



University of
Zurich^{UZH}

Parsing Information from PDF Papers to Match With EUREKA Addresses

Lucas Pelloni
Zurich, Switzerland
Student ID: 13-722-038

Supervisor: Bruno Rodrigues, Prof. Dr. Thomas Bocek
Date of Submission: March 15, 2019

Abstract

The EUREKA platform is a scientific rating-publishing platform built on the top of the Ethereum blockchain. EUREKA leverages blockchain technology and smart contracts to enable an more efficient and trustworthy publishing process by introducing time stamping for discoveries and a rewarding system for all stakeholders involved in the publishing process of a certain article using micro-payments consisting in EUREKA (EKA) tokens. Therefore, authors who either directly (being a main author) or indirectly (being a cited author in the reference section) contributed to the publication of an article, have the right to obtain the reward. However, to be able to obtain the reward, authors need to have a valid address in the EUREKA ecosystem for receiving their tokens. As of today, the EUREKA platform only enables authors with a valid Ethereum address to write their articles using online editor in the EUREKA platform. This allows extracting metadata of the article quickly. However, submitting external PDF documents are not supported in its current version. PDFs are the most common format for scientific publications and they became ubiquitous as a digital replacement for paper and hold all kind of critical business data. Thus, it is essential to support a feature able to extract author information from these documents. The main aim of this thesis, is to implement a tool that automatically extracts author information and matches the gathered data with the EUREKA platform and see if there is a match. This includes, information retrieval and text mining techniques for authors as well as an address scheme proposal for an EUREKA addresses. Furthermore, it also contains a way on how to include an EUREKA address in new and upcoming papers.

Contents

Abstract	i
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	2
2 Related Work	3
3 Design	5
3.1 Address Design	6
3.1.1 Address Proposal	8
3.2 Parsing PDF	11
3.2.1 First Phase	11
3.2.2 Second Phase	14
4 Implementation	15
4.1 Scope	15
4.2 Frontend	16
4.2.1 React	16
4.2.2 Structure	17
4.2.3 EUREKA Citation	21
4.2.4 PDF Parser	25

4.3	Backend	33
4.3.1	Technical stack	33
4.3.2	Security provisions	34
5	Evaluation	35
6	Final Considerations	39
	Bibliography	41
	List of Figures	45
	List of Tables	47

Chapter 1

Introduction

1.1 Motivation

The EUREKA [1, 2] platform is a scientific rating-publishing platform using the public permission-less Ethereum [4] blockchain as its central infrastructure. EUREKA leverages blockchain technology and smart contracts to enable an more efficient and trustworthy publishing process by introducing time stamping for discoveries and rewarding reviewers as well as editors, linked and referenced authors using micropayments consisting in EUREKA (EKA) tokens. On the one hand, referenced (or cited) authors are those that contributed indirectly to a paper and are listed at the end of the scientific work in the References section. On the other hand, linked authors are those that have actually a direct contribution to a paper by being main or co-authors. Both are used as input data for the SC that locks the amount paid in EKA tokens for a submission and keeps track of all stakeholders that need to be rewarded at the end of the peer-review and editorial assessment process. By using a blockchain-based system, the EUREKA platform provides transparency, giving the authors immediate ownership rights of their discoveries. Finally, it incentives token holders with a loyalty program and in future, research funding will be supported as well. The potential use cases for EUREKA are substantial and global.

As of today, the EUREKA platform only enables authors with a valid Ethereum address to write their articles using the draft-js [5] online editor provided by the EUREKA platform. Since authors are able to add other co-authors by looking for other users already stored in our database, extracting metadata of the article is easier and quicker. A simple cross-match in the EUREKA database is enough to identify a user and extracts its corresponding Ethereum address. However, submitting external PDF documents is not supported in its current version. PDFs became ubiquitous as a digital replacement for paper and hold all kind of critical business data, and therefore, it is essential to support a feature able to detect author information from these documents. In fact, the constant growth in the number of scientific publications makes it more difficult to deal with collections of academic literature and to assess the quality of scientific output [6]. This leads to the need for automatizing the process of enriching documents with author-related metadata.

There are several reasons ranging from technical issues to practical workflow obstacles that make extracting data from PDF a challenging task. For example, even though PDFs are the most common format for scholarly articles, they lack of structural information and can assume different encodings and have different citation mechanisms. Also, one important factor of scientific literature is that they cannot be treated as isolated units, but they do need to be placed in a specific context. In fact, several aspects can contribute to the creation of a scientific paper, such as the author's affiliations or the journal where the article has been published. In this scenario, the challenge lies on how to extract data from PDF documents in an automated fashion reliably, *i.e.*, without having too much manual effort.

1.2 Description of Work

The main goal of this Independent Study (IS) is to implement a tool for parsing scientific papers in PDF format, that is able to extract author information and match it with EUREKA addresses using well-known information extraction techniques. Indeed, the tool presented in this report implements well-known text-mining techniques and smart entity extraction recognition to match author names as well as address schemes for EUREKA addresses. Furthermore, it contains a way on how to include an EUREKA address in new and upcoming papers by generating a *Bibtex* scheme based on the passed addressed. Finally, part of the result of the IS will be integrated into a second step into the current EUREKA platform and used for integrating existing publishing platforms and offering more options to users for uploading articles.

1.3 Thesis Outline

Chapter 2 describes the different approaches in recent literature for metadata extraction in scientific publications. *Chapter 3* presents the overall design architecture of the tools presented in this thesis. *Chapter 4* continues with the implementation details *Chapter 5* describes the evaluation of the empirical study has been conducted. *Chapter 6* closes the thesis with the final considerations.

Chapter 2

Related Work

This chapter presents similar and related proposed works in the field of automatic extraction of author information from PDF documents. All of them tackle the goal of automatizing the process of extracting contextual information from scientific publications presenting different techniques.

Indeed, [6] presents a novel approach using both machine learning and named entity recognition techniques. In this approach, a scientific paper is considered as a chain of contiguous text blocks, which represent the basic building blocks of PDF documents. These blocks are in turn clustered based on specific type of information (*i.e.*, metadata categories) they contain. In order to deal with a large variety of layouts of scholarly articles, they implemented a system that uses unsupervised and supervised machine learning techniques. Unsupervised learning (clustering) is used to analyse the physical layout of a scientific paper by extracting contiguous text blocks, while supervised learning is used to extract author and affiliation metadata, identifies references and collect funding information from these blocks. Afterwards, heuristics are applied to identify the reference section of a certain scientific output (*e.g.*, looking for specific keywords that indicates the beginning of the reference section) and segment it into individual reference tokens. Finally, these strings are clustered and then with named entity recognition techniques information such as research grants and funding agencies are extracted. This tool, has been built on the top of the Apache PDFBox library [7].

In [9], the authors implemented a tool that is meant to extract basic metadata from PDF papers for Digital Library Ingest. Their approach consists of a combination of several techniques including low level PDF parsing and similarly to the approach presented in [6] they apply machine learning-based layout analysis. In addition, in order to identify authors in PDF papers they use a well-known citation database called DBLP [10] to assist the process of information extraction and tested their system with more collections of paper selected from recent conference proceedings. In their paper, they argue that automatic extraction of administrative metadata from PDF documents has become the *de-facto* standard for documents stored in digital libraries. This is the reason why they proposed a tool that is fully compatible with the digital library software package Greenstone [11]. The latter, is a open source library that provides different tools for organizing information and makes it available for everyone, with a user-friendly interface to access

all documents in a collection. Their approach treats a PDF paper as a set of different layers. As first step, JPedal [12] package is used to extract the basic layout blocks. These blocks provide additional information such as textual objects, relative position within the document and style properties. The information is secondly serialized in order to obtain A XML tree which contains the different features of each region in the page. Afterwards, information gathered is processed and features are extracted with a numerical feature vector. These vectors are then used as input for a Multi-Layer Perceptron (MLP) classifier that tries to identify which layout blocks could contain author information. Finally, information is exported in a XML format that can be interpreted by Greenstone. The latter displays the author metadata in a structured way.

In [8], is presented a novel combination of algorithms for extracting metadata from online texts and secondly linking full-text documents together based on their set of references. Firstly, PDF documents are transformed into a more readable and comprehensible format such as ASCII or XHTML (XML version of HTML) with the help of well-known conversion tools [8]. Afterwards, they identify and split all citations in the references section in order to connect more scientific literature by following references links of a certain paper. Similarly to the approach presented in [9], they apply well-known heuristics based on specific labels for locating the references section. In order to identify more references in the text, the approach parses anchor formats found in D-Lib [15] using different algorithms and regular expressions. Afterwards, they extract document names from the list of references using different text-mining techniques looking for strings in the format of citation standards such as CrossRef [14] or DOI [13]. Finally, author information extraction is handled with the Wand's Author Name algorithm [8]. This algorithm considers the different names extraction considering that names can be spelled in many different ways.

In [16], a overall evaluation of different tools for metadata extraction from PDF articles is presented. The authors applied a strict methodology based on three main requirements. First, tools have to present a reliable interface that can be easily integrated into custom projects. Secondly, they have to obtain reliable results without any manual effort from an user. Third, the output has to be presented in a readable format such as XML or XHTML. In order to test a variety of PDF articles, they selected a dataset from arXiv [18], a scientific publication archive, which contains scientific papers with different disciplines having various layouts and encodings. Three evaluations are performed with two independent setups. Based on the results presented in [16], it was pointed out noteworthy results, which were assigned to their accuracy scores, respectively. GROBID [17] performs best based on the information extraction accuracy in the following categories: titles, authors, author's last names, abstracts and publication date.

GROBID is a machine learning open-source library for extracting information from scholarly documents. It parses and re-structures raw PDF documents putting the focus on technical and scientific publications. It has been made open-source in 2012 and its code is available here [19]. This tool presents information extraction techniques including bibliographical information extraction (*e.g.*, title, abstract, authors, affiliations, keywords, etc.) and References extraction and parsing from articles in PDF format.

Chapter 3

Design

The main idea driving this thesis is to identify an approach that uniquely identifies and recognizes authors metadata in scientific publications and links the gathered information to the corresponding (if any) valid EKA addresses. Since the EUREKA SC [2] performs micropayments in EKA tokens at the end of the peer-review and editorial assessment process for compensating all stakeholders that contributed in the publishing process, a list of Ethereum addresses have to be locked in order to know where to send EKA tokens in case an author gets cited. This implies, that each time a new article is submitted in the EUREKA platform with either the EUREKA draft-js online editor [5] or as external PDF a list of valid EKA addresses have to be served as input data for the SC. Thus, the challenge lies in finding a reliable mechanism for parsing external PDF scientific papers, since differently from documents generated with the draft-js online editor, they lack of structural information, layout consistency and are harder to handle.

The approach defined herein consists of two different mechanisms based on the information present in the input PDF document:

- The first one consists of retrieving information based on a direct mapping between an author and a valid EKA address stored in the paper itself. In this case, a Ethereum address needs to be published together with the author credentials in a paper on the title page as well as in the references. However, in order to obtain compact addresses that are easy to recognize in PDF publications, a conversion mechanisms between Ethereum and new EKA addresses is needed (the EKA Address scheme is discussed in the section 3.1). This approach is accurate since EKA addresses can be easily detected in PDF publications based on addresses checksums and known EKA prefixes.
- The second one consists of storing the Author-Address mapping in the convenience layer of the EUREKA platform [1]. Authors metadata are then extracted using well-known information retrieval techniques from PDF publications and afterwards cross-matched with existing author credentials in the EUREKA database for linking the corresponding EKA address. This approach allows to support already published papers. However, the gathered information cannot be considered 100% accurate

and thus still need manual effort from the user for confirming the guessed author metadata found in a paper.

3.1 Address Design

As of today, addresses in the EUREKA ecosystem are represented using the Ethereum [4] address scheme. This implies, that addresses are formatted in hexadecimal (base 16 notation) and composed by 40 characters (which leads to a combination of 20 bytes). Because of a convention, most of the Ethereum addresses are displayed with two leading characters "0x". The following line illustrates a common Ethereum address.

`0xcd2a3d9f938e13cd947ec05abc7fe734df8dd826`

Ethereum addresses are not compact enough and are hardly recognizable if not accompanied by the leading characters. In order to tackle this challenge, hereby is introduced a new address scheme for EKA addresses with the following goals:

- EKA addresses should be as compact and short as possible.
- Need to have a recognizable and EUREKA-related prefix at the beginning.
- Must include a mechanism for avoiding copy and paste errors or typos (*e.g.*, by using printable characters)

To prevent typos and misspelled EKA addresses, it was integrated for all encoding schemes a checksum [20] mechanism. In this case, a checksum is represented by one or more bytes placed in a certain and known position of the input address against which later comparisons can be performed to confirm the integrity of the address and detect potential errors. The checksum algorithm is discussed in Chapter 4. In order to achieve the goals listed above, is required an experiment using different encoding schemes for a given input Ethereum address. The analysis procedure considers a list of criteria, where to each criteria it was assigned a relative weight based on its importance. Afterwards, a rating scale from 1 to 10 (where 1 is slight extent, 5 some extent and 10 great extent) was established for each of the following criterion:

1. The number of characters present in the encoded output address (weight 30%)
2. Whether the base notation of the output address is conform to the BibTex [22] standard, *i.e.*, the styled-independent text-based file format for listing bibliography items such as theses, books or articles (weight 55%).
3. Addresses can be easily copied from some visual source and pasted in modern browsers or editors without any additional effort (*e.g.*, a double-click select the whole string without having to use the cursor) (weight 10%).
4. Addresses' characters can be easily distinguished and thus be unambiguously processed by humans (weight 5%).

It was assigned to the second criterion the highest relative weight in our scoring model, since we retain that new EKA addresses have to be easily integrated into a scientific publication using the standards provided by the Latex ecosystem [21]. Secondly, the first criterion has a relative weight of 30% because we want to keep new EKA addresses as short as possible for saving space and characters in scientific papers. Finally, the third and fourth criterion assume lower relative weights but should also be taken into account in the scoring model.

The following encoding schemes were considered:

- **Base 32** encoding [23]

Base 32 uses a restricted set of symbols *i.e.*, a 32-characters set of twenty-six upper-case letters from A to Z, and the digits from 2 to 7, that can be conveniently processed by Computers and are easily readable by humans. In some circumstances, padding has to be added to the output string and can be inferred from the length of the string modulo 8. Commonly, padding is represented by the "=" symbol.

- **Base 58** encoding [24]

Subset of Base 64 that has been adapted to avoid both letters and non-alphanumeric characters which may be ambiguous when printed (*e.g.*, the number 0 and the upper-case letter O). Base 58 has several applications, for instance in the crypto field, the Base 58 encoding has been modified and used as Base58Check [27] for encoding Bitcoin addresses and includes error detection with a checksum in the last few characters.

- **Base 62** encoding [25]

Subset of Base 64 that has been modified in order to exclude all symbols and keep only lower-case and upper-case letters and numbers from 0 to 9.

- **Base 64** encoding [26]

Base 64 has the largest alphabet of all encodings schemes listed here. This means, it translates a given ASCII string into a radix-64 representation, where each digit can be represented with exactly 6 bits of data.

Table 3.1 summarizes the different alphabets used by the encoding algorithms presented above.

As illustrated in Table 3.1, the Base64 encoding scheme represents the superset of all other encoding algorithm. It includes all letters lower-case and upper-case, all digits from 0 to 9 and two additional symbols "+" and "/". Base62 excludes the two symbols from its alphabet and further base58 removes all ambiguous characters that are hardly differentiable for humans. In order to choose the most appropriate encoding schema for the new EKA addresses, it was selected three different Ethereum addresses that have been used as input data for the different encoding algorithms. The Ethereum addresses used for the evaluations are presented below and have been randomly extracted from the

Encoding Scheme	Alphabet
Base 32	234567 ABCDEFGHIJKLMNOPQRSTUVWXYZ
Base 58	123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Base 62	0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Base 64	0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz+ /

Table 3.1: Different Alphabets for Encoding

Ethereum network using the Blockchain Explorer EtherScan [28].

```
0x9456562ff5ab0919b5c0093c400000a63dedf638
0x546780150c610c27342289932f577673786d4a79
0xd7b9a9b2f665849c4071ad5af77d8c76aa30fb32
```

All addresses contain exclusively 40 hexadecimal characters (with two leading characters at the beginning, *i.e.*, "0x") and are represented with 20 bytes. In order to have as input data purely hexadecimal characters, only the relevant part for the Ethereum address was encoded, *i.e.*, without the two leading characters at the beginning used as a convention. It is worth to point out, that the different encoding algorithms were performed without adding any additional checksum bytes in the input address. This means, that the final number of characters may differ depending on the number of bytes reserved for later comparison using the checksum. Furthermore, for each of the following encoding mechanisms a Buffer representation was used for manipulating streams of characters using an array of Bytes. Since Ethereum addresses are encoded using the hexadecimal system, exactly 4 bits are needed for representing each character (16 possible combinations). Thus, in order to have one stream of bytes, exactly 8 bits are necessary for formally representing two hexadecimal characters. This conversion was performed using binary data in order to prevent any loss of characters in the encoding mechanism. In the Table 3.2 the different resulting output addresses are illustrated with the corresponding encoding schemes.

3.1.1 Address Proposal

As illustrated in the Table 3.2, the base notation adopted from the different schemes substantially affect the resulting number of characters of the input address. Table 3.3 shows the evaluation of the different encodings based on our weighted scoring system. The scale is defined from 1 to 10, where 1 means that the criteria has been slight fulfilled, 5 to some extent and 10 to a great extent. The criterion used for the evaluation have been presented in Section 3.1.

Encoding Scheme	Output addresses	# characters
Base 32	HE2DKNRVGYZGMZRV MFRD AOJRHF RD KYZQGA4TGYZUGA YDAMBQME3DGZDFMRTDMMZY — GU2DMN ZY GAYTKMDDGY YT AYZSG4ZTIMRSHA4TSMZSMY 2TONZWG4ZTOOBWMQ2GCNZZ — MQ3WEOLB HFRDEZRWGY2T QNBZMM2DANZRMFSDKYLGG4 3WIODDG43GCYJTGBTGEMZS	64, 64, 64
Base 58	34rzyMkFQms632c1BfhgUTtXjq67 — 2BChynFFDPdnqhFUwEuthSkmEUHe — 41KDakHUnQH DWmfSnG3A jvCEuNm w	28, 28, 28
Base 62	LAFdkL45qCDuhSknV2FQ4JUCQuW — C2fr4CpfDXPct4FGO6htk7bLGUj — UmOR60dE7DXUpUndWHILMoN6WiA	27, 27, 27
Base 64	IFZWL/WrCRm1wAk8QAAApj3t9jg= — VGeAFQxhDCc0IomTL1d2c3htSnk= — 17mpsvZlhJxAca1a932Mdqow+zI=	28, 28, 28

Table 3.2: Encoding Scheme Experiment Results

Encoding Scheme	1st. Criterion (30%)	2nd. Criterion (55%)	3rd. Criterion (10%)	4th. Criterion (5%)	Results
Base 32	2	10	5	8	7
Base 58	9	10	10	10	9.7
Base 62	10	10	10	2	9.6
Base 64	9	8	3	8	7.8

Table 3.3: Weighted Scoring System Results

On the one hand, Base 62 [25] performs best in terms of number of characters for a given Ethereum address. Indeed, using Base 62 as encoding scheme, the input address starts from a number of 40 characters and is reduced to an output address with 27 characters in radix 62. In this sense, there is a substantial difference (minus 32.5%) in the number of characters prior to and after the conversion. On the other hand, both Base 64 [26] and Base 58 [24] result into an output address of 28 characters (again, with a difference of minus 30% compared to the input address). Base 64 adds padding to its encoding mechanism, that is the reason why even with a higher notation the number of characters are the same as for Base 58. However, taking into consideration only the first criterion, we can see only a slight difference between Base 58, 62 and 64. Base 32 [23] performs the worst by obtaining more characters compared to the input address.

A part of Base 64, all output addresses are fully compatible with the Latex ecosystem and thus can be easily compiled using the LaTeX Compiler [30]. Base 64 adds two special symbols to its alphabet (*i.e.*, "+" and "/"). Although these two symbols are supported by most of the LaTeX editors, it may happen that addresses containing them can be error-prone if copied and pasted into a writing system that does not support specific industry standards for encoding strings.

Taking into account the third criterion, Base 64 is the worst performer in terms of portability of the resulting addresses into other visual sources such as text editors or modern browsers. The reason relates to the additional symbols added in the Base 64 alphabet. Indeed, it acts as breaking point for splitting the string into different parts when it is selected with a double click. This makes the output addresses hardly portable. Base 32 fulfils the criteria to some extent since its output addresses have a considerable length and thus hardly displayable into one single line. Finally, both Base 58 and Base 62 addresses can be copied and pasted into different writing system without requiring any additional effort from the user (*e.g.*, without having to highlight with cursor the whole string for selecting it).

The fourth criterion represents the main substantial difference between the performances of Base 58 and Base 62. Indeed, although using Base 62 a character would have been saved in the resulting addresses, its alphabet contains similar-looking characters which may look ambiguous when printed. This implies, that some of the characters belonging to the same address may be hardly distinguished when processed by humans. In fact as illustrated in Table 3.2, if the third output address generated by Base 62 is considered, it can be found in the same address both similar-looking characters O (capital o) and 0 (zero). In this sense, Base 58 has a subset alphabet which is especially designed for humans who manually enter the data or copying the addresses from some visual source into another. Therefore, its alphabet is free from ambiguity in distinguishing characters and performs best taking into account this criterion.

For all reasons explained above, it is identified as most suitable performer the Base 58 [24] encoding algorithm for creating the new EKA address scheme. Thus, in the following we propose our final address proposal:

EKA34rzyMkFQms632c1BfhgUTtXjq67

In order to achieve the initial goals presented in 3.1, the **EKA** prefix is added at the beginning of the output address in order to make it more recognizable present in scientific publications. Furthermore, a checksum mechanisms is added to detect errors and to avoid typos. The number of bytes reserved for the checksum are located at the end of the string. Its implementation details is presented in the Chapter 4.

3.2 Parsing PDF

The approach for parsing scientific publications in PDF format consists of two phases. As illustrated in Table 3.1, the first process consists of uploading and pre-processing a specific PDF by extracting structural and physical layout information from it. In this phase PDFs are first normalized and then divided into different layers where each layer can be viewed as a PDF page instance. In turn, PDF instances are divided into streams of contents, *i.e.*, blocks of content containing textual objects together with style proprieties. Text contents represent the building blocks of scientific PDFs and are clustered in the second phase into different collections for extracting enhanced information about the authors.

3.2.1 First Phase

The main information processing cycle in the first phase begins with the elaboration of a single text item. In this cycle, each block triggers three parallel processes that analyse the text item from different angles. These processes are summarized as follows:

- *Process Text Blocks*

The first, puts the focus on the unstructured textual object that can be extracted from a text item. In this procedure, our algorithm looks for a potential EKA address using multiple regular expressions augmented with the EKA prefix and the checksum present in the address. Simultaneously, it tries to extract named entities such as author names using Named-entity recognition [31] techniques. The latter, tries to identify with a grammar-based mechanism a set of predefined categories for the entity person such as first name or last name. Furthermore, our entity information retrieval algorithm supports abbreviations in this class. For example, entities such as **Mario Rossi** or **M. Rossi** or **Mario R.** will result into the same person. Finally, our algorithm checks whether the found entity is present in the list of well-known English names [32] and contemporaneously not present in the list of the 20 thousand most common English words [33]. Based on our experiment, this increases precision in the results. More details are described in the Chapter 4.

- *Sentence Map*

The second process iterates all text items present in a PDF page instance and puts them into a map of sentences. For each text item, the font size is extracted from its text and it is used as a entry key for the sentence map. Thus, the value linked to that specific key is an array of text strings. For each PDF page instance, a different sentence map is composed by different text items clustered by their font sizes. In

PDF scientific publications all main and co-authors are listed in the first page (next to title) and usually using a larger font-size. Therefore, it is possible to state that text items clustered in one of the first groups in the sentence map (ordered by font-size) belong to the same authors list. As a result, the sentence map can be used in the second phase for augmenting the precision in the process of guessing authors.

- *Heuristics for Reference Section*

The third process applies to each text item different text heuristics in order to identify the start of the reference section in a PDF paper. As explained in [8], common keywords for indicating the begin of the reference section are:

References

Bibliography

Notes and References

Note and References

Section <number>: References

Once a text item matches one of the above mentioned heuristics, a global variables in our algorithm is set stating that the reference section has been located. It is worth to point out that only section headings for the bibliography keyword are taken into account. For instance, text items matching heuristics in the table of contents are not considered valid indicators. Once the reference section has been identified, the coming text items are iterated and treated as segments of references. Finally, based on a regular expression that matches square brackets around numerals, text items are split into tokens and the corresponding list of references is created.

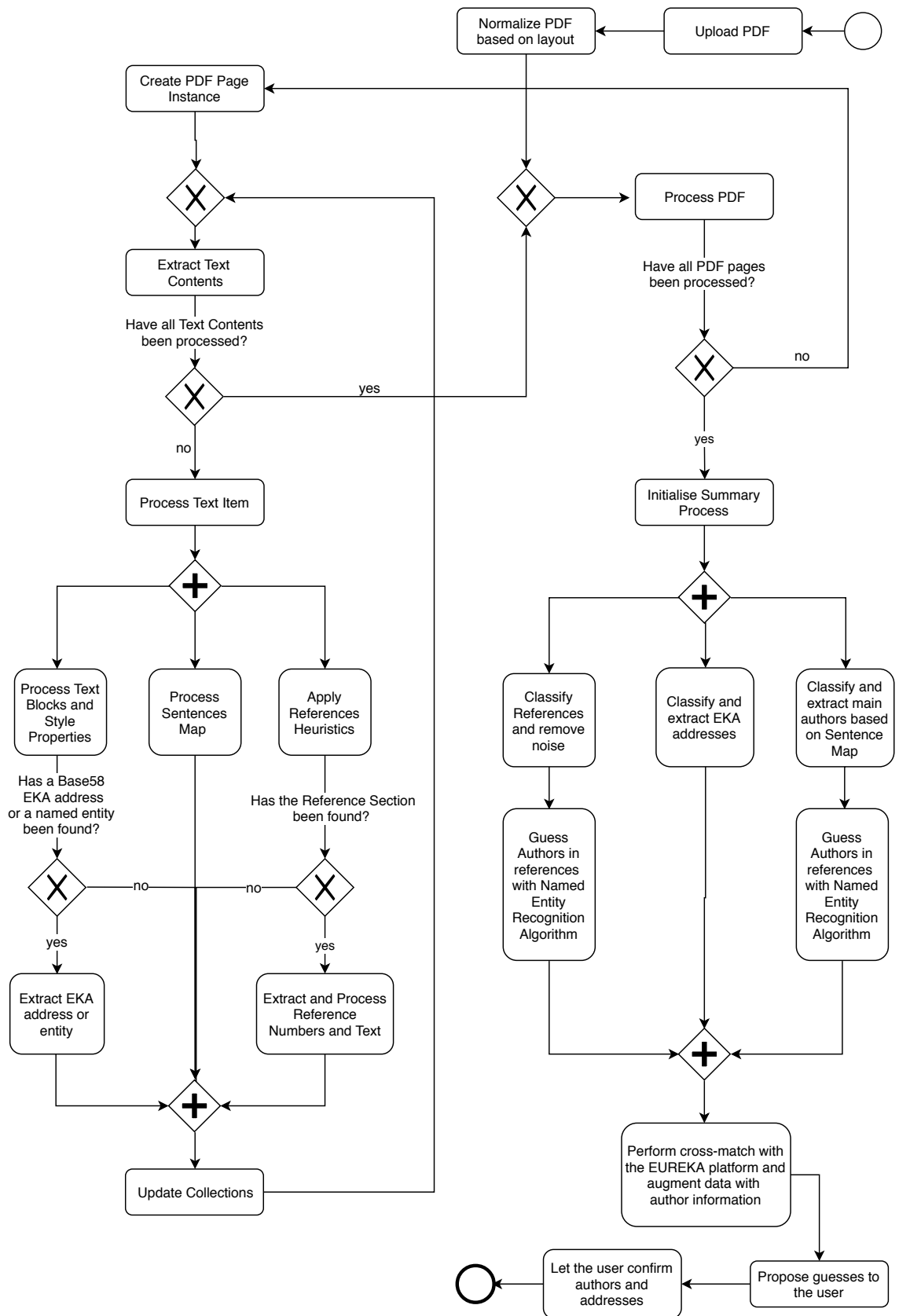


Figure 3.1: PDF Parsing Algorithm

3.2.2 Second Phase

The second phase begins when all PDF pages have been processed and consists of three processes that are executed in parallel and have the common goal to elaborate data coming from the first phase. The three processes are classified as follows:

- *Classify References with NER [31]*

In this task, after removing noises in the collection, the NER algorithm is applied to the references list created in the first phase. The goal is to extract entities from these bibliography sources, where for each reference there is a list of guessed authors displayed to the user in the summary view and awaiting for confirmation. Once a specific reference is confirmed by the user, a cross-match request is sent to the EUREKA platform to check whether there is a mapping between the author credentials and the user object stored in the EUREKA database. In a positive case, a link to the corresponding author is displayed where the new EKA address can be generated as well.

- *Classify and extract EKA addresses*

As explained before in the Section 3.1, extracting EKA addresses is an accurate process since they can be identified using our checksum mechanisms and the EKA prefix. Therefore, this task extracts and classifies EKA found in PDF publications and processes them without any additional interaction required from the user. It is worth to point out, that duplicates are removed for entities and EKA address that result into the same person. For instance, if a main author is identified and it leads to (after a cross-match in the EUREKA platform) the same address as a EKA address that is extracted using checksums and the EKA prefix, it is not counted twice.

- *Classify Authors with NER and sentence map*

In this process, main or co-authors of the papers are classified using the NER algorithm augmented with data coming from the sentence map. Indeed, once a named entity has been identified we look for its position in the sentence map. In other words, it is looked for its key (font-size). If most of the retrieved linked authors corresponds to the same key (*i.e.*, to the same group clustered by font-size in the sentence map), it is possible to assume that all values present in that specific group are all linked authors. If this is the case, all text items present in this cluster are displayed to the user, who confirms the different guesses. Similarly to the first task, once a specific author is confirmed by the user, a cross-match request is handled by the EUREKA platform and a potential Author-Address mapping is returned. Again, a link to the authors is shown to the user where the new EKA addresses can be retrieved.

Chapter 4

Implementation

This chapter presents the concrete implementation details of the design presented in the previous Chapter. Although relying on different components implemented in the EUREKA platform [1] in terms of UI style convention, the different Web applications that we present in this Chapter have been implemented from scratch.

4.1 Scope

In order to achieve the design goals proposed in the previous Chapter, the implementation among different Web applications is split as follows:

- **EUREKA Citation**

This Web application implements the Base58 [24] encoding algorithm to create the mapping between Ethereum and EKA addresses. Indeed, it is able to encode or decode Ethereum respectively EKA addresses and to generate the corresponding QR code and BibTex [22] template that can be easily integrated into different LaTeX [21] formats.

- **EUREKA Scientist Tracker**

Web application that links an email address to a valid EKA address. An author can be identified with its address and all his relevant information (track records) are displayed to the user. This Web application encourages transparency since the lookup can be performed by everyone logged in the EUREKA platform. However, it relies on EUREKA's convenience layer for users information.

- **EUREKA PDF Uploader**

Core Web application that uploads and parses scientific papers. This application parses all relevant EUREKA information and extracts information for authors that have been cited and have the right of being rewarded with EKA tokens.

The Web applications listed above will be addressed later more in details in this Chapter.

4.2 Frontend

4.2.1 React

For the User Interface (UI) of the different applications, the ReactJs Library [3] is used. ReactJs is a JavaScript lightweight library for building fast and efficiently web apps. For all applications, React is used in combination with JSX [35], a XML-like syntax extension built on top of JavaScript. In fact, JSX adds XML elements to JavaScript such as tags, which can have attributes, children and other properties. Each JSX element must be in line with the strictly hierarchical tags structure. Finally, JSX can be easily combined with CSS and HTML [34] and thus add power to the React rendering process.

UI

The web applications relies on EUREKA's design Palette. This has been implemented from scratch, including design components (such as Modals, Buttons, Input fields, Tables, Label, etc.) and color schemas. However, concerning the styling of basic design elements, EUREKA's design is inspired by the Argon Design System from Creative Tim [37]. . Indeed, EUREKA does not use any external design Framework but relies on reusable React components. Concerning style maintenance and HTML tags management, EUREKA relies on the well-known external library *styled-components* [36].

This library, enhances CSS in order to style React component systems and provides several advantages [36], such as:

- No more bugs with unique Class Names
The library generates unique class names during the rendering process and ensures that there are no duplicates for different HTML tags.
- Conditional Styling
CSS elements are like React components. Using props [38] in combination with Template Literals syntax [43] CSS attributes can be dynamically styled without having to manually manage tons of external CSS classes.
- Maintenance
Searching for styled elements is the same as searching for React components. Everything is compact and stays in the same file. In fact, styled-components avoids time-consuming searches for finding the styling affecting a specific component.

Redux

In order to easily manage different states simultaneously, Redux [39] was integrated. It is a popular state management tool for React and Javascripts applications. Redux helps the application to know what and where is the current application state. Since our application data is dynamically shared among different components may be tedious to track every

internal state change. Most of the components in the React ecosystem are connected with the Redux Store[42]. Redux wraps a component by injecting the requested states and actions as props.

4.2.2 Structure

The folder structure of the different web application on the client side follows the EUREKA's structure. The code structure follows the rules for each React ecosystem, which states that a Web application must be divided among different reusable Components. In this sense, a component can be either functional, *i.e.* a stateless view [49] that displays a set of passed properties (props) [38] or a class, *i.e.*, a stateful component [49] that encapsulates and manages its own state [38] and logic. Usually, the latter are those components which are in charge of sending asynchronous requests to our Back-end, change the state that is locally encapsulated within the component and pass the fetched data through the application tree using read-only props. In other words, stateless components are responsible for "*How things look*" in your application, while stateful components are responsible for "*How things work*" by considering dynamic states and actions that can change at any time in the components life cycles. To summarize, the general file structure that has been adopted for each application can be divided into three main categories:

- **React components**

React components are used to split the UI into independent and isolated pieces that can be reused along the application. Conceptually, React components are like native JavaScript functions that take props [38] as input and renders React elements into the screen. This category mainly contains stateless and stateful React components and can be divided into six sub-categories:

- *Stateful components, React Routers* [48]

Stateful components represents the logical layer for EUREKA. They are responsible for storing application data and defining the different behaviour of the child components based on an internal state change. They can be presentational (*e.g.*, manage the state for an input field) or logical (*i.e.*, fetching the data from our server and pass these data along the application tree).

In addition to the classic stateful components, this category contains React Routers as well. React Routers are built at the top of React and are used for core dynamic routing in our Front-end. They defined, based on certain conditions such as matching a URL path, which component will be rendered.

- *Constants*

This category consists of a variety of Javascript exported variables which can be statically shared along the platform. For instance, a string for a common message in a Modal or the EUREKA colors palette.

- *Design components*

Styled components [36] represents the foundation of EUREKA's design. This module enhances normal CSS and removes the mapping between components and styles by creating normal React components instead of external CSS classes.

EUREKA exploits this module in order to have reusable styled components that can be shared along the whole platform and thus improving the style consistency. For instance, a Button or a Input field can be imported into different React components and have the same style structure. Based on different props, they can be conditionally styled as well. The following code snippets illustrates an simplified example of how React components coupled with styled-components can be conditionally styled.

```

1  const ParentContainer = styled.div `
2    // other CSS attributes here
3    display: ${props => (props.isHidden ? 'none' : 'flex')});
4  `;
5
6  const TextContainer = styled.div `
7    // other CSS attributes here
8  `;
9
10 export default class Notification extends Component {
11   constructor() {
12     super();
13     this.state = {
14       isHidden: false
15     };
16   }
17   hideAlert() {
18     this.setState({isHidden: true});
19   }
20   render() {
21     return (
22       <div>
23         <ParentContainer isHidden={this.state.isHidden}>
24           {/*Other HTML tags...*/}
25           <TextContainer onClick={() => this.hideAlert()}>{
26             this.props.text}</TextContainer>
27           {/*Other HTML tags...*/}
28         </ParentContainer>
29       </div>
30     );
31   }
32 }

```

Listing 4.1: Styled Components - Conditional Styling

The code snippet above shows a stateful React Component that stands for a simple Notification label in our Front-End. As you can see, there is no mapping between components and styles using external CSS classes, but everything can be represented in the same component. Indeed, in the first part (until line 12) there are only native HTML tags overwritten with a styled-components instance called *styled* (line 1 and 9). Starting from line 13, we can see the logic and rendering process of the stateful component. The conditional statement in order to have different styling behaviours can be found at the lines 3-5. Using Template literals syntax [43] a condition can be embedded in the expression for the specific CSS attribute. For instance, the *ParentContainer* passes as props to its styled-container the internal state that describes whether a certain

notification in the Front-End has been hidden or not. As specified in the constructor, the initial state is set to false which means that the notification is displayed at the beginning. Based on the result returned by the binary operator [46] at line 3, the CSS expression that gets rendered for the *ParentContainer* is the following:

```
1 display: flex;
```

As soon as the user clicks on the *TextContainer* the internal state changes (line 17-19) and thus the whole component gets re-rendered. This leads to get the first operand as a result for the binary expression for the *display* attribute (and so hiding the notification in the UI):

```
1 display: none;
```

As you can see, conditional styling with styled-components can be done using Template literals syntax [43] and binary operators [46] and so allow to save a substantial amount of work that external CSS classes would require.

– Reducers

This section is composed by Redux [39] components: Reducers [41], Store [42] and Actions [40]. Redux is a well-known predictable state container for JavaScript-based applications that allows your application to easily manage different local and global states among different components. Actions are used for payloads of information that are directly sent to the global store, which in turn holds the different application states together. Reducers define how a state shall change when a certain action is executed. In other words, an action can be viewed as a trigger for a reducer which defines how an application's state changes. The store keeps track of the application states and gives access to different component to access them in the application tree. It is worth to point out, that usually stateful component do subscribe to the Redux state and dispatch different actions through the application while presentational views read data and invoke callbacks using props.

– Stateless components (i.e., views) and Guards

This category represents the foundation of EUREKA's front-end. In fact, there are more than 50 views in our application. Conceptually, stateless components do not manage any internal state and do not have the component lifecycle methods [49]. They just take props as an argument and returns a react element. In our Web applications, they are used as presentational views for components like *Address.js*, *Avatar.js*, *Figure.js*, etc. In fact, the main purpose of these components is how to display the input data in the UI and not how to manipulate them. They can be viewed as a sort of black-box components, where an input comes in and the output results into a styled view. For instance, the *Address.js* view can be used to display the address of the logged User in the Header as well as the sender address of a Ethereum transaction.

```
1
2 const SpanContainer = styled.span `
3   color: ${props => props.color};
4   font-size: ${props => props.fontSize};
5   word-break: break-all;
```

```

6  ' ;
7
8  const Address = ({ethereumAddress, content, ...otherProps}) =>
    {
9      return (
10         <SpanContainer href={`'/app/users/${ethereumAddress}`} {
            ...otherProps}>
11             {content}
12         </SpanContainer>
13     );
14 };
15
16 export default Address;

```

Listing 4.2: Stateless Component - Reusable Address

As you can see, the *Address* view can be completely reused and anonymised since it just takes a set of parameters and returns a React element displaying the passed content that points to an external link.

In addition to traditional Views, this category includes Guards as well. Guard is a *angular-like* term. They are stateless components that protect certain routes within the application that can be accessed if a set of needed criteria are fulfilled.

• Helpers Modules

This section consists of a variety of utility elements, *i.e.*, descriptive methods or variables that can be reused among the whole application. The majority of these elements, are highly related to the style and layout of EUREKA. For instance, this category contains the *colors.js* and *layout.js* file. The first one, describes the entire color palette by exporting the hexadecimal representation of the EUREKA colors. The second one, contains a collection of CSS Media queries [45] in order to have different screen breakpoints. By doing so, EUREKA's design components behave differently on each side of the breakpoint and make the EUREKA's Front-end compliant with the Responsive Web Design principles [45].

The following Application tree illustrates a simplified version of the EUREKA's Front-End structure.

```

├─ frontend
│   └─ helpers
│   └─ webpack
│       ├── components
│       ├── constants
│       ├── design-components
│       ├── reducers
│       └─ views

```

4.2.3 EUREKA Citation

As mentioned in Chapter 3, the Base 58 [24] encoding scheme has been selected for generating a mapping between EKA and Ethereum addresses. In the *EUREKA Citation* web application, a mechanism is provided for encoding and decoding Ethereum respectively EKA addresses in one single app. Indeed, this application consists of two main routes that are encapsulated in the *MainRouter.js* as described below:

```

1  class MainRouter extends Component {
2    render() {
3      return (
4        <Container>
5          <Title>What do you want to do?</Title>
6          <OptionPicker
7            onChange={value => {
8              this.props.history.push('/${value}');
9            }}
10         />
11          <Route path="/encoding" exact render={() => <Encoding />} />
12          <Route path="/decoding" exact render={() => <Decoding />} />
13        </Container>
14      );
15    }
16  }

```

Listing 4.3: EUREKA Citation - Main Router

The application simply offers to the user the possibility to choose between two options, *i.e.*, encoding or decoding a certain address. The action selector is illustrated by the code at lines 5-9, which triggers an *onChange* event based on the user's choice.

- **Encoding**

The encoding process starts with a check that states whether the inserted address is a valid Ethereum address (line 2). The algorithm uses Web3 [57] utils library that exposes a method called *isAddress* that returns whether the input value is a valid Ethereum address. Indeed, this method controls whether the input string contains the exact number of hex characters (*i.e.*, 38 or 40, depending on the leading characters at the beginning). In addition to that, it checks whether it is checksummed. The checksum mechanism of an Ethereum address works as follows:

1. All leading characters at the beginning of the input address are removed
2. The address is hashed using SHA-3 [58].
3. All characters are iterated and if the *ith* character is in the range between 9 to f (in hex format) then it should be upper-case. Otherwise, lower-case. If all the statements assert a true condition, then we can state that the given address is correctly checksummed.

```

1  const NUMBER_OF_CHECKSUM_BYTES = 2;
2  encode(value) {
3    if (web3.utils.isAddress(value)) {
4      value = value.toLowerCase().replace(/~0x/i, '');

```

```

5      let address = new Buffer(value, 'hex');
6      let hash = new Buffer(sha256(sha256(address)));
7      let checksum = hash.slice(0, NUMBER_OF_CHECKSUM_BYTES);
8      let addressChecksum = Buffer.concat([address, checksum]);
9      return bs58.encode(addressChecksum);
10   }
11 }

```

Listing 4.4: EUREKA Citation - Encoding with Base 58

After checking the validity of the input address in terms of number of characters and checksum, our encoding algorithm removes all leading characters as well. Afterwards, it creates a buffer of 8-bit unsigned integers (*Uint8Array* [59]) to deal with binary data directly and prevent the lost of bytes during the conversion mechanism. Since Ethereum addresses are purely composed by hexadecimal, we need exactly 1 byte of data for representing 2 hex characters. The following example gives a brief overview of the conversion between hex characters and streams of unsigned integers.

```

1  const example = new Buffer("1234", "hex");
2  // Uint8Array(2) --> [18, 52]
3  const extreme = new Buffer("ff", "hex");
4  // Uint8Array(1) --> [255]

```

Listing 4.5: EUREKA Citation - Buffer of Hexcharacters

As shown in 4.5, in the first line exactly two bytes are needed for four hex characters. At line three, the maximal value is represented that can be obtained with two hex characters, *i.e.* 255. This is because, the value *f* in the hex alphabet is represented by 1111 in a binary scale. Since we have two *f* in the extreme example, we obtain 11111111 as result in binary. The latter translated in base 10 corresponds exactly to 255 ($2^8 - 1$). The encoding algorithm continues with the hash function at line 5 in Listing 4.4. Similarly to 256 [60], one well-known cryptographic hash algorithm. The reason behind this, is to add a checksum into the match request is performed to gather additional information behind that specific address. Therefore, w

Ethereum Address

0x49F13DC1c04D05a8A937944Ab715154d81A61B12




0x49F13DC1c04D05a8A937944Ab715154d81A61B12
lucas@sciencematters.io

ENCODE
↑↓

Evaluating the following address:
0x49F13DC1c04D05a8A937944Ab715154d81A61B12

EKA Address

EKAM4zLLinCrnUS7hgCgG7ekdaWF6CENR

BibTex

```

@article{EKAM4zLLinCrnUS7hgCgG7ekdaWF6CENRxy,
  author = {"Lucas Pelloni"},
  year = {tbd},
  title = {tbd},
  journal = {tbd},
  volume = {tbd},
  number = {tbd},
  pages = {tbd},
  doi = {tbd},
  isbn = {tbd},
  issn = {tbd},
  pmid = {tbd},
  url = {tbd}
}

```

QR Code


Figure 4.1: EUREKA Encoding Algorithm - Screenshot

As shown in the Figure 4.1, an user has been associated to an Ethereum address and thus the BibTex template has been consequently adjusted by inserting already the author's name.

- **Decoding**

Similarly to the encoding process, it is needed to ensure the validity of the given EKA address before decoding it. Thus, it is necessary to check whether the inserted address is correctly checksummed. First of all, as illustrated at line 2-4 of Listing 4.6, all characters in the string belong are checked for the Base 58 alphabet. If this is the case, the algorithm continues with decoding the given value by creating the corresponding buffer

of unsigned integers. Afterwards, using the same number of checksum bytes stored in the encoding algorithm, the bytes representing the address and the checksum itself (lines 7-8) are extracted. Then, the hash of the address buffer is computed again using double hashing SHA-256. Finally, at lines 10-13 the checksum digits are iterated and it is checked whether the hashed characters correspond to the checksum values (which have been hashed previously in the encoding process). If the flag returns true, the EKA address can be considered checksummed.

```

1  const isCheckSum = value => {
2    if (!areCharactersAllowedByBS58(value)) {
3      return false;
4    }
5    let flag = true;
6    const buffer = new Buffer(bs58.decode(value));
7    const address = buffer.slice(0, -NUMBER_OF_CHECKSUM_BYTES);
8    const checksum = buffer.slice(-NUMBER_OF_CHECKSUM_BYTES);
9    let hash = new Buffer(sha256(sha256(address)));
10   checksum.forEach((digit, i) => {
11     if (digit !== hash[i]) {
12       flag = false;
13     }
14   });
15   return flag;
16 };
17
18 const bs58decode = value => {
19   if (value.toString().includes(InitialPrefix)) {
20     value = value.replace(InitialPrefix, '');
21   }
22   if (isCheckSum(value)) {
23     const buffer = new Buffer(bs58.decode(value));
24     const address = buffer.slice(0, -NUMBER_OF_CHECKSUM_BYTES)
25       .toString('hex');
26     return web3.utils.toChecksumAddress('0x' + address);
27   }
28   return null;
29 };

```

Listing 4.6: EUREKA Citation - Checksum mechanism

If the EKA address is checksummed, a buffer is created with the given value that has been decoded using again Base 58 (line 23). Then, at line 24 the bytes in the buffer reserved for the address are extracted and then converted into a hex string representation. Finally, the returned address is converted into a checksummed Ethereum address using a utility function of Web3 [57] (line 25).

Figure 4.2 illustrates the encoding algorithm. As illustrated, the resulting Ethereum address exactly matches the input address given to the encoding algorithm of Figure 4.1.



Figure 4.2: EUREKA Decoding Algorithm - Screenshot

4.2.4 PDF Parser

The PDF Parser Web application consists of different phases as explained in the Figure 3.1. Indeed, the first phase starts with the upload of a PDF paper and the consequently pre-processing of its different layers in terms of layout and textual objects. For achieving this and for the complete analysis of a PDF document, the PDF library PDF.js [63] as been used. This library has been proposed by Mozilla [62] and it is well established in the Javascript ecosystem known as a web standards-based platform for parsing and rendering PDFs. In this sense, the library is composed by three different layers [63]:

1. Core
Core layer which represents the foundation of PDF.js and all subsequent layers. It is in charge of parsing and interpreting a binary PDF. Its API is considered an advanced usage.
2. Display
Display layer that acts as a middleman for the core layer and exposes an easy-to-user API that returns well-structured information out of a document.
3. Viewer
Viewer layer represents the UI for viewing a PDF in other browsers.

Since the full library is available only in pure Javascript, a content delivery network (CDN) [64] has to be used to integrate the PDF.js library into our React application. Furthermore, it has been installed as external dependency a generic pre-build of the source code known as PDF.js - dist [65] to have a set of automatically generated build scripts for calling specific methods in the PDF.js API. Concerning the rendering of PDFs document in our UI, the DOM is accessed directly using Refs [66] because of the integration of a third-party DOM library. Indeed, each PDF page that it rendered has its own viewport that defines the size in pixels and the initial rotation. The page is defined and gets rendered with a canvas object that we manage using Refs in React.

Upload

As described in our design approach in Figure 3.1, we start uploading a generic PDF document in our platform. For PDF file storage we rely on Amazon Simple Storage Service (Amazon S3) [67]. This service from Amazon is flexible and we can store and retrieve any amount of data dynamically. Furthermore, it provides a easy-to-use management of our storage information so that we can easily organize our PDF documents. Listing 4.7 illustrates the uploading process of a PDF object into S3.

```

1 S3Upload.prototype.uploadToS3 = function(file, url, public_url) {
2   let this_s3upload;
3   let xhr;
4   this_s3upload = this;
5   xhr = this.createCORSRequest("PUT", url);
6   if (xhr) {
7     xhr.addEventListener("load", () => {
8       if (xhr.status === 200) {
9         this_s3upload.onProgress(100, "Upload completed.");
10        return this_s3upload.onFinishS3Put(public_url);
11      }
12      return this_s3upload.onError("Upload error: " + xhr.status);
13    });
14    xhr.addEventListener("error", () => {
15      return this_s3upload.onError("XHR error.");
16    });
17    xhr.upload.addEventListener("progress", e => {
18      let percentLoaded;
19      if (e.lengthComputable) {
20        percentLoaded = Math.round((e.loaded / e.total) * 100);
21        return this_s3upload.onProgress(
22          percentLoaded,
23          percentLoaded === 100 ? "Finalizing." : "Uploading."
24        );
25      }
26    });
27  } else {
28    this.onError("CORS not supported");
29  }
30  xhr.setRequestHeader("Content-Type", file.type);
31  xhr.setRequestHeader("x-amz-acl", "public-read");
32  return xhr.send(file);

```

```
33 };
```

Listing 4.7: EUREKA PDF Uploader - S3 AWS

The XMLHttpRequest (XHR) [68] protocol is used for handling the communication between the S3 Web Server and our Web Browser.

First Phase

As illustrated in Figure 3.1, the process continues with the instantiation of a PDF instance. This is achieved in the *PDFReader.js* and represents the foundation of a PDF and all subsequent layers. Since PDF.js heavily relies on the use of Promises [69], each interaction with the library is handled using Promises and asynchronous operations. Listing 4.8 is extracted from the class *PDFReader.js*, which illustrates the creation of a global PDF instance.

```
1 import * as PdfJs from "pdfjs-dist";
2 import { WORKER_URL } from "../WorkerLink";
3
4 async componentDidMount() {
5   const url = this.props.pdfs[0];
6
7   // The workerSrc property shall be specified.
8   PdfJs.GlobalWorkerOptions.workerSrc = WORKER_URL;
9   // Asynchronous download of PDF
10  const loadingTask = PdfJs.getDocument(url);
11
12  const that = this;
13  loadingTask.promise.then(pdf => {
14    that.setState({ pdf });
15  });
16 }
```

Listing 4.8: EUREKA PDF Uploader - PDF.js

The loading of a PDF instance is asynchronous. Furthermore, the location of the PDF.js worker has to be specified together with the url where the PDF is located. Because of AWS S3 [67], the url of the location of the PDF is returned once it gets successfully uploaded. This is then passed using props to the *PDFReader.js* component. In line with the design approach, the process continues with three sub processes. First of all, the total number of pages of the PDF is extracted using the PDF instance we created before. Afterwards, for each page is created a new instance which asynchronously loads its text content. Thus, in the *Viewer.js* component, a for-loop for iterating the number of pages is created. Listing 4.9 illustrated initialization process of PDF pages.

```
1 async componentDidMount() {
2   let { pdf } = this.props;
3   const TOTAL_PAGES = pdf ? pdf._pdfInfo.numPages : 0;
4   let pages = [];
5
6   for (let p = 1; p <= TOTAL_PAGES; p++) {
7     let page = await this.getTextContent(p, pdf, TOTAL_PAGES);
```

```

8     pages.push(page);
9 }
10 // ..more code here..
11 }

```

Listing 4.9: EUREKA PDF Uploader - PDF Page Instance

After that, for each PDF instance its content is extracted. Listing 4.10 illustrates the promise resolution generated by the text content of a PDF page.

```

1  getTextContent(pageNum, PDFDocumentInstance, TOTAL_PAGES) {
2    // Return a Promise that is solved once the text of the page is
      retrieval
3    const that = this;
4    return new Promise((resolve, reject) => {
5      PDFDocumentInstance.getPage(pageNum).then(pdfPage => {
6        // The main trick to obtain the text of the PDF page, use the
          getTextContent method
7        pdfPage
8          .getTextContent({
9            normalizeWhitespace: true,
10           disableCombineTextItems: true
11         })
12         .then(textContent => {
13           const textItems = textContent.items;
14           const sentenceMap = that.getMap(textItems);
15           const text = that.getText(textItems);
16           const references = that.getReferences(
17             textItems,
18             pageNum,
19             TOTAL_PAGES
20           );
21
22           let obj = { sentenceMap, text, pageNum, references };
23           resolve(obj);
24         });
25       });
26     });
27 }

```

Listing 4.10: EUREKA PDF Uploader - Text Content of PDF Instances

In line with the approach described in the Section ??, the algorithm continues with three sub processes that happen once the promise is fulfilled (line 12).

- *Process Text Blocks*

The first, puts the focus on the unstructured textual object that can be extracted from a text item. Indeed, all text items are iterated from the promise callback and an array of sentences is created, as illustrated below:

```

1  getText(textItems) {
2    let sentences = [];
3    for (let i = 0; i < textItems.length; i++) {
4      const item = textItems[i];
5      sentences.push(item);

```

```

6      }
7      return sentences;
8  }

```

Listing 4.11: EUREKA PDF Uploader - Text Items

In a second step, these sentences are processed and entities are extracted using Named Entity Recognition [31]. Additionally, EKA addresses are clustered as well. Indeed, for each sentence a method named *extractEntity()* implemented in the *ExtractEntityFactory.js* is called.

```

1      page.map(sentence => {
2          let text = sentence.str;
3          let entity = extractEntity(text);
4      });

```

Listing 4.12: EUREKA PDF Uploader - Entity Extraction

This factory is in charge of extracting from a set of normal sentences a list of potential entities and EKA addresses. In this sense, for performing entity extraction, the algorithm relies on the well-known Javascript library called *Compromise* [70] which is used for natural-language processing. In addition to that, the resulting entities are augmented using different regular expressions such as splitting strings using commas or other separators that usually are used when authors are listed in scientific papers. Another indicator is the letter capitalisation. Usually, first names and last names present the first character upper-case. Furthermore, entities are filtered by checking if they are contained in the list of the most famous English person nouns and simultaneously not included in the list of the most used common words in English.

Concerning the extraction of EKA addresses, it is easier and more accurate. In fact, for all sentences in a PDF page, those strings that contain the EKA prefix, the correct amount of characters and that are correctly checksummed as described in the Section 4.2.3 are extracted. Listing 4.13 illustrated the extraction mechanism for EKA addresses.

```

1  isEKA(string) {
2      if (string.toString().includes(InitialPrefix)) {
3          const split = string.toString().split(InitialPrefix);
4          const EKAish = split[split.length - 1];
5          const EKA = EKAish.substr(0, NUMBER_OF_EKA_CHARACTERS);
6          if (isChecksum(EKA)) {
7              return EKA;
8          }
9      }
10     return null;
11 }

```

Listing 4.13: EUREKA PDF Uploader - EKA Extraction

- *Sentence Map*

The second process iterates all text items present in a PDF page instance and puts them into a map of sentences. For each text item that gets processed, a set of style properties are encapsulated into a single Javascript object:

```

1 {
2   str:"one two three four",
3   dir:"ltr",
4   width:151.67308125000002,
5   height:20.6625,
6   transform:[20.6625,0,0,20.6625,110.854,651.853],
7   fontName:"g_d0_f1"
8 }

```

Listing 4.14: EUREKA PDF Uploader - Text Item Object

The transform array contains a set the viewport information, which includes scale, rotation and Y-axis flip. This means, that with a simple Math operation, the font size can be extracted from the transform attribute.

```

1   getMap(textItems) {
2     let sentenceMap = new Map();
3     for (let i = 0; i < textItems.length; i++) {
4       let item = textItems[i];
5
6       const tx = item.transform;
7       const fontSize = Math.sqrt(tx[2] * tx[2] + tx[3] * tx[3]);
8
9       let entrySentences = sentenceMap.get(fontSize)
10        ? sentenceMap.get(fontSize)
11        : [];
12       if (fontSize) {
13         entrySentences.push(item);
14         sentenceMap.set(fontSize, entrySentences);
15       }
16     }
17     return sentenceMap;
18   }

```

Listing 4.15: EUREKA PDF Uploader - Font-Size Extractor

Listing 4.15 illustrates the extraction of the font-size for each text item object (line 7). After assigning the font-size to a local variable, this can be used as a entry key for the sentence map, where the value linked to that specific key is an array of text strings called *entrySentences*.

- *Heuristics for Reference Section*

The third process applies to each text item different text heuristics in order to identify the start of the reference section in a PDF paper. Indeed, once there is a match with one of the heuristics listed in the Section ??, the state of our React components is change, as shown in Listing 4.16.

```

1   getReferences(textItems, pageNum, TOTAL_PAGES) {
2     let references = [];
3     for (let i = 0; i < textItems.length; i++) {
4       const line = textItems[i].str;
5       if (
6         (REFERENCES_HEURISTICS.includes(line))
7       ) {
8         this.setState({ refStartPointHasBeenFound: true });
9       }

```

```

10      // ..more code here..
11  }

```

Listing 4.16: EUREKA PDF Uploader - Reference Heuristics

From that moment, the subsequent iterations are treated as part of the reference section with the goal to create an array of objects. These objects have in turn two attributes: the reference number and the corresponding text. Therefore, the algorithm start extracting all references by iteratively calling the method shown in Listing 4.17.

```

1  extractReferenceNumber(text) {
2    let match = text.toString().match(/\s*([0-9]+)/g);
3    if (match) {
4      return match.map(m => m.trim());
5    }
6    return null;
7  }

```

Listing 4.17: EUREKA PDF Uploader - RegEx for References

The regular expression which is charge of extracting the reference is illustrated at line 2. It extracts all strings that have square brackets around numerals.

Second Phase

The second phase can be viewed as a summary in the UI for the results of the algorithms performed in the first phase. In this sense, this phase lets the user confirm the author that have been extracted previously and creates a bridge with the EUREKA platform for looking up these authors if the cross-match request returns a successful mapping. The whole summary process in handled in the *Summary.js* class and in all its child components. Each time a cited or linked author is confirmed by the user, the following cross-match request is sent to EUREKA's API.

```

1  async fetchAuthorInformation(field) {
2    let url = queryString.stringify(field);
3    return fetch(`${getDomain()}/api/users?${url}`, {
4      method: "GET",
5      headers: {
6        "Content-Type": "application/json"
7      },
8      credentials: "include"
9    })
10     .then(response => response.json())
11     .then(async response => {
12       if (response.success) {
13         if (response.data) {
14           return response.data;
15         }
16         return null;
17       }
18     })
19     .catch(err => {
20       alert(err);

```

```
21     });
22   }
23
24
25   async addFieldToConfirmedList(field, key, list) {
26     if (field) {
27       let user = await this.fetchAuthorInformation(field);
28       let personal = {};
29       if (user) {
30         personal = {
31           ethAddress: user.ethereumAddress,
32           avatar: user.avatar,
33           email: user.email
34         };
35       }
36       this.addField(key, list, {
37         personal,
38         fName: field.fName,
39         lName: field.lName,
40         refNumber: field.refNumber,
41         id: uuidv1()
42       });
43     }
44   }
```

Listing 4.18: EUREKA PDF Uploader - Cross-Request with EUREKA's API

As presented in Listing 4.18, every time the user confirms a certain field, this field is sent to the EUREKA's API that either returns the author information in case of a successful mapping or null. This information is then added to the UI and displayed to the user. Afterwards, the user is able to click on a link that points to the author lookup in the EUREKA platform. Figure 4.3 illustrates the summary component of an uploaded PDF. On the left side, it can be found authors that have been guessed and references that have been identified. On the right, authors and references that have been confirmed by the user. As shown in the Figure, for certain authors, an avatar together with a link and the Ethereum address is displayed. This means, that the algorithm was able to identify a mapping between the author's credentials and a valid Ethereum address in the EUREKA database.

SUMMARY FUNCTIONAL COMPONENT

Guessed Authors

First Name	Last Name	OK	delete
Burkhard	Stiller	OK	delete
Yehia	El-khatib	OK	delete
Literature review	Literature review	OK	delete

Confirmed Authors

First Name	Last Name	OK	delete
Fabio	Victoria Hecht	OK	delete
David	Hausheer	OK	delete
Lucas	Pelloni	OK	delete

References we found

First Name	Last Name	OK	delete
E	Adar	OK	delete
H	Bloom	OK	delete

Confirmed References

First Name	Last Name	OK	delete
Thomas	Bocek	OK	delete
W	Kun	OK	delete
V	Hecht	OK	delete

Figure 4.3: PDF Parser - Summary Component

4.3 Backend

Since the web applications do not need any additional end points that are not included in EUREKA's backend, we used the API exposed by the EUREKA platform. In the following it is disclosed a simple description of how it works.

4.3.1 Technical stack

The backend is created like a typical back-end application. We chose Javascript as the programming language to assure parity with the front-end stack. Particularly the reason to use the same library for interaction with the Frontend led us to this decision. Using npm [50] as the package manager provides us with a big variety of well established libraries and frameworks.

We use Node.js [51] with the Express web application framework [73]. Node.js is a platform built on Chrome's Javascript runtime for easily building fast and scalable network applications. Express is a light-weight web application framework, which helps to organize your web-application into a Model-View-Controller [71] architecture on the server side. The Frontend application communicates through a REST-Interface with it. For the storage of data the back-end application sends data to a database. We decided to setup the database with MongoDB [53], as this technology allows us to adapt data structure quickly. In the other hand we use the library Mongoose [54], which extends the function-

alities by schema validation, constraints definition and value inspection. The combination of these two technologies allow us to set limitations and control were we need it in order to assure data integrity and allows us to be flexible in areas where it is necessary.

4.3.2 Security provisions

Passport[55] was selected to provide the authentication service. It acts as middleware and developed a local authentication strategy. For a secure encryption of the users password we make use of the javascript module Bcrypt [72], which uses the bcrypt-algorithm to assure security and store only the encrypted password on our database. In order to grant proper authorization to the platforms users, we implemented our own Role-based access control. Each users holds some roles within the database. Roles, which are coupled with a wider access privilege. This way we assure correct authorization (*e.g.*, only a users registered in the EUREKA platform can have access to the user lookup routes in the API). Express-session provides us the functionality of working with session, which are stored in our databases. User, which have successfully logged in within our platform are provided with a session id. Sessions expire after a defined amount of time has past. All this assures a safe and secure authentication process for our future platform users.

Chapter 5

Evaluation

The PDF Uploader and the accuracy of the retrieved author information was evaluated by conducting an empirical study on a smaller dataset of scientific publication in PDF format. The dataset has been randomly selected from a small set of publications found in the database of the Communication System Group of the Department of Informatics at the University of Zurich [74]. The criteria we chose for evaluating our approach concern the accuracy and precision of the retrieved author information. The metric we used consists of:

1. How many EKA address have been correctly identified and extracted independently of their position in the PDF layout structure.
2. How many linked (main) authors have been correctly identified and guessed by the Named Entity Recognition algorithm with the assumption that usually main authors appear in the first pages of a scientific paper.
3. How many references have been correctly extracted and identified with the assumption that the reference section are usually placed in the last pages of a scientific publication.
4. How many cited (referenced) authors have been correctly identified and guessed by the Named Entity Recognition algorithm from the list of retrieved references in the third point.

In order to obtain numerical values from our evaluation, we used two well-known metrics used in pattern recognition as a measure of relevance called *Precision* and *Recall* [75]. Precision is the ratio between the relevant elements among the set of retrieved instances while Recall is the ratio of all relevant instances that have been retrieved compared to the total amount of relevant instances. In other words, precision states "how useful the retrieved results are", while recall defines "how complete the retrieved elements are". The following equations summarize the two metrics:

$$precision = \frac{d_{relevant} \cap d_{retrieved}}{d_{retrieved}} \quad recall = \frac{d_{relevant} \cap d_{retrieved}}{d_{relevant}}$$

Document Number	Relevant EKA	Retrieved EKA	Relevant & Retrieved EKA	Precision Recall (in%)
1	4	4	4	P = 100, R = 100
2	6	6	6	P = 100, R = 100
3	9	9	9	P = 100, R = 100
4	6	6	6	P = 100, R = 100
5	2	2	2	P = 100, R = 100

Table 5.1: EKA Address Retrieval - Precision and Recall

For instance, if one scientific publication has six main authors, they would form the set of relevant data in our measurements. Therefore, if our NER [31] algorithm is able to find three of them (true positives) and in addition it returns one false positive, it would have a precision of $3/4$ and a recall of $3/6$. Precisely, we tested 10 scientific publications which have different layout structures, citation mechanisms and ways of listing main authors. Five documents in our dataset contain EKA addresses and named authors in their publication while the other five only contain named authors. We split our results into different tables, where each one addresses one of the criteria we explained at beginning of this Chapter. Taking into account only the retrieved EKA addresses, Table 5 shows how our algorithm performed in terms of recognition of the EKA pattern. As we explained in Chapter 3, parsing and identifying EKA addresses in scientific publications is accurate since we can use checksum mechanisms and well-known prefixes for extracting the addresses pattern. Because of this, the extraction of EKA address returned a precision respectively a recall of 100% for all documents in our dataset.

Concerning the retrieved linked authors, Table 5 shows how the NED algorithm was able to identify most of them in their scientific publications. The ratio between Recall and Precision is not elevate, which implies that those authors that have not been identified can be considered edge cases that do not affect the accuracy of the algorithm. Furthermore, a part of document number two, all relevant main authors of a specific paper have been retrieved by our algorithm. In most of the cases (three out of five), we observed the presence of at least on false positive.

Table 5 shows how the algorithm performed in terms of the retrieval of bibliography references within scientific publications, independently of the presence of EKA addresses. As observed, the algorithm was able to identify for all documents present in our dataset, the reference section and its corresponding list of references.

Table 5 illustrates the algorithm performance concerning the retrieval of author metadata in the reference list of a specific paper. We can observe, that for the vast majority of documents the ratio between Precision and Recall is smaller than one. This means, that the algorithm was not as precise as for the retrieval of main authors. This is mainly because, references in a scientific paper are harder to parse since they usually have abbreviations for author names, different entities on the same line in the same reference and inconsistently distributed punctuation.

Document Number	Relevant Main Authors	Retrieved Main Authors	Relevant & Retrieved Main Authors	Precision Recall (in %)
1	5	6	5	P = 83.3, R = 100
2	6	4	4	P = 100, R = 66.7
3	4	5	4	P = 80, R = 100
4	7	7	7	P = 100, R = 100
5	5	6	5	P = 83.3, R = 100

Table 5.2: Main Authors Retrieval - Precision and Recall

Document Number	Relevant References	Retrieved References	Relevant & Retrieved References	Precision Recall (in %)
1	21	21	21	P = 100, R = 100
2	35	35	35	P = 100, R = 100
3	22	22	22	P = 100, R = 100
4	18	18	18	P = 100, R = 100
5	7	7	7	P = 100, R = 100
6	12	12	12	P = 100, R = 100
7	25	25	25	P = 100, R = 100
8	15	15	15	P = 100, R = 100
9	17	17	17	P = 100, R = 100
10	31	31	31	P = 100, R = 100

Table 5.3: References Retrieval - Precision and Recall

Document Number	Relevant Cited Authors	Retrieved Cited Authors	Relevant & Retrieved Cited Authors	Precision Recall (in %)
1	17	22	15	P = 68.1, R = 88.2
2	26	31	24	P = 77.4, R = 92.2
3	14	16	12	P = 75.5, R = 85.7
4	14	14	12	P = 85.5, R = 85.7
5	2	4	1	P = 25, R = 50
6	5	8	2	P = 25, R = 40
7	20	21	11	P = 52.3, R = 55.5
8	12	15	8	P = 53.3, R = 66.7
9	18	23	17	P = 73.9, R = 94.4
10	19	25	15	P = 60, R = 78.9

Table 5.4: Cited Authors Retrieval - Precision and Recall

Chapter 6

Final Considerations

The main goal of this thesis was to implement a tool for the extraction of author information from PDF scientific papers and match it in a second step with EUREKA addresses using well-known information extraction and named entity recognition techniques. To achieve our goal, we planned intermediary sub-goals covering different aspects of the implementation. The first specific objective was to provide an address scheme proposal for EUREKA addresses in order to easily recognize short and compact EKA address patterns in scientific papers. The second stated goal was to implement a platform for converting Ethereum addresses into new EKA addresses. After that, a strategy was needed to include an EUREKA address in new and upcoming papers, by supporting the LaTeX ecosystem and providing QR codes as well. All stated objectives have been successfully achieved. The main factors that contributed to the success can be attributed to (i) the vastness of research works in the same area of this thesis presenting different approaches for extracting author information from scientific papers and (ii) the availability of the EUREKA source code for different API endpoints and UI components. Furthermore, it has been important to have the support of EKA Blockchain Solutions GmbH [1] that offered cutting-edge infrastructures, cloud services such as AWS S3 for uploading PDF papers and experts in the field for exchanging different ideas and opinions.

Despite the lack of structural information in PDF papers, they can still be viewed as unstructured artefacts which may contain pieces of structured data. Indeed, the main challenge of this work was not located in parsing and extracting textual information from PDF documents, but consisted of the interpretation of the gathered data. In this sense, the usage of PDF.js as instrument for navigating PDF documents has been essential. For the analysis of the data, different regular expressions based on punctuations together with Named Entity Recognition has been applied to the text streams. On the one hand, our algorithm performed relatively well in terms of Precision and Recall and most of the time has been able to identify all main authors of a scientific publication and its reference section. However, the dataset that has been used covers a minimal fraction of the scientific world and its corresponding publications. For instance, different citation mechanisms have not been properly tested yet. Furthermore, in order to have the algorithm to perform as expected, a set of pre-conditions have to meet the input PDF, such as having the main authors of the first page, the reference section in the last pages, etc. The main challenge of

this work, consisted of standardizing the retrieval of main (linked) and cited (referenced) authors with a single algorithm. Indeed, linked authors are usually listed in the first page with both first and last name. Instead, cited authors are located in the reference section and lack of consistency in their textual representation. If on the one hand, the way of listing main authors was pretty similar among different papers in the data set, on the other hand cited authors were cited using different standards and punctuation.

The relationship between the proposed schedule and the work schedule has been respected and the milestones punctually achieved. However, the time allocated for the documentation part has been a bit underestimated. Nonetheless, no modifications occurred during the execution of the work between the scope defined in the Project phase and in what was developed.

Future work consists of testing the algorithm with a bigger dataset with documents coming from different scientific fields. In addition, different citation mechanisms will be integrated and tested.

Bibliography

- [1] EUREKA. The Next-generation Open Science Platform Powered by Blockchain, <https://eurekatoken.io/> (Last accessed March 2019).
- [2] EUREKA Platform (MVP), <https://eurekaplatform.herokuapp.com/> (Last accessed March 2019).
- [3] ReactJS: A JavaScript library for building user interfaces, <https://reactjs.org/> (Last accessed January 2019).
- [4] Ethereum Community. Ethereum Blockchain App platform, <https://www.ethereum.org/> (Last accessed March 2019).
- [5] Draft-JS. Rich Text Editor Framework for React JS, <https://draftjs.org/> (Last accessed February 2019).
- [6] Klampfl, Stefan, and Roman Kern. *Machine learning techniques for automatically extracting contextual information from scientific publications*. Semantic Web Evaluation Challenge. Springer, Cham, 2015. (Last accessed February 2019).
- [7] Apache PDFBox - A Java PDF Library, <https://pdfbox.apache.org/> (Last accessed February 2019).
- [8] Bergmark, Donna. *Automatic extraction of reference linking information from online documents*. Cornell University, 2000. (Last accessed February 2019).
- [9] Marinai, Simone. *Metadata extraction from PDF papers for digital library ingest*. 2009 10th International Conference on Document Analysis and Recognition. IEEE, 2009. (Last accessed February 2019).
- [10] DBLP: Citation Network Dataset, <https://aminer.org/citation> (Last accessed February 2019).
- [11] Greenstone Digital Library Software, <http://www.greenstone.org/> (Last accessed February 2019).
- [12] JPedal: A Java PDF Library, <https://www.idrsolutions.com/jpedal/> (Last accessed March 2019).
- [13] DOI System: Digital Object Identifier System, <https://www.doi.org/> (Last accessed March 2019).

- [14] Crossref: an official Digital Object Identifier Registration Agency, <https://www.crossref.org/> (Last accessed March 2019).
- [15] D-Lib: The Magazine of Digital Library Research, <http://www.dlib.org/> (Last accessed March 2019).
- [16] Lipinski, Mario, et al. *Evaluation of header metadata extraction approaches and tools for scientific PDF documents*. Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries. ACM, 2013. (Last accessed March 2019).
- [17] GROBID: A machine learning software for extracting information from scholarly documents, <https://github.com/kermitt2/grobid> (Last accessed March 2019).
- [18] arXiv.org: a scientific publication archive, <https://arxiv.org/> (Last accessed March 2019).
- [19] GROBID: Source code, <https://github.com/kermitt2/grobid> (Last accessed March 2019).
- [20] Wikipedia: Checksum, <https://en.wikipedia.org/wiki/Checksum> (Last accessed March 2019).
- [21] LaTeX - A document preparation system <https://www.latex-project.org/> (Last accessed March 2019).
- [22] BibTex File Format, <http://www.bibtex.org/Format/> (Last accessed March 2019).
- [23] Wikipedia: Base32, <https://en.wikipedia.org/wiki/Base32> (Last accessed March 2019).
- [24] Wikipedia: Base58, <https://en.wikipedia.org/wiki/Base58> (Last accessed March 2019).
- [25] Wikipedia: Base62, <https://de.wikipedia.org/wiki/Base62> (Last accessed March 2019).
- [26] Wikipedia: Base64, <https://de.wikipedia.org/wiki/Base64> (Last accessed March 2019).
- [27] Base58Check encoding for Bitcoin, https://en.bitcoin.it/wiki/Base58Check_encoding (Last accessed March 2019).
- [28] EtherScan, Ethereum Blockchain Explorer, <https://etherscan.io/> (Last accessed March 2019).
- [29] Node.js Documentation: Buffer, <https://nodejs.org/api/buffer.html> (Last accessed March 2019).
- [30] LaTeX Compiler, https://www.overleaf.com/learn/latex/Choosing_a_LaTeX_Compiler (Last accessed March 2019).
- [31] Wikipedia: Named-entity Recognition, https://en.wikipedia.org/wiki/Named-entity_recognition (Last accessed March 2019).

- [32] GitHub: A list of well-known Names, <https://github.com/eureka-blockchain-solutions/PDF-Parser/blob/master/src/data/AllNames.js> (Last accessed March 2019).
- [33] GitHub: 20K most common English Words, <https://github.com/eureka-blockchain-solutions/PDF-Parser/blob/master/src/data/EnglishWords.js> (Last accessed March 2019).
- [34] W3Schools: HTML Styles - CSS, https://www.w3schools.com/html/html_css.asp (Last accessed January 2019).
- [35] Introducing JSX <https://reactjs.org/docs/introducing-jsx.html> (Last accessed January 2018).
- [36] Documentation for Styled Components <https://www.styled-components.com/docs/basics> (Last accessed January 2019).
- [37] Argon Design System https://demos.creative-tim.com/argon-design-system/index.html?_ga=2.268656502.625706937.1549100533-387788188.1536925437 (Last accessed February 2018).
- [38] Agiliq: Understanding State and Props in ReactJS, <https://www.agiliq.com/blog/2019/05/understanding-react-state-and-props/> (Last accessed January 2019).
- [39] Redux Framework: A JavaScript predictable state container <https://redux.js.org> (Last accessed January 2019).
- [40] Redux Actions <https://redux.js.org/basics/actions> (Last accessed January 2019).
- [41] Redux Reducers <https://redux.js.org/basics/reducers> (Last accessed January 2019).
- [42] Redux Store <https://redux.js.org/basics/store> (Last accessed January 2019).
- [43] Template literals (Template strings) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals (Last accessed February 2019).
- [44] Redux DevTools <https://github.com/reduxjs/redux-devtools> (Last accessed March 2019).
- [45] Responsive Web Design: Media Queries https://www.w3schools.com/css/css_rwd_mediaqueries.asp (Last accessed March 2018).
- [46] Binary Operator <https://www.techopedia.com/definition/23953/binary-operator> (Last accessed February 2019).
- [47] Destructuring assignment https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment (Last accessed February 2019).

- [48] React Router: Basic Routing <https://reacttraining.com/react-router/web/guides/quick-start> (Last accessed March 2019).
- [49] Stateful vs. Stateless Functional Components in React <https://code.tutsplus.com/tutorials/stateful-vs-stateless-functional-components-in-react--cms-29541> (Last accessed March 2019).
- [50] NPM: The package manager for JavaScript and the world's largest software registry <https://www.npmjs.com/> (Last accessed March 2019).
- [51] Node.js: a script runtime built <https://nodejs.org/en/> (Last accessed March 2019).
- [52] Express: a web application framework for node.js <https://expressjs.com/> (Last accessed March 2019).
- [53] MongoDB: a document database with scalability and flexibility <https://www.mongodb.com/> (Last accessed March 2019).
- [54] MongooseJS: elegant mongodb object modeling for node.js <https://mongoosejs.com/> (Last accessed March 2019).
- [55] PassportJs: Simple, unobtrusive authentication for Node.js <http://www.passportjs.org/> (Last accessed March 2019).
- [56] Passport npm module: NPM package for passport.js, compatible with node.js <https://www.npmjs.com/package/passport> (Last accessed March 2019).
- [57] Web3.js: Ethereum JavaScript API, <https://web3js.readthedocs.io/en/1.0/index.html> (Last accessed March 2019).
- [58] Wikipedia: SHA-3 Cryptographic Hash Algorithm, <https://en.wikipedia.org/wiki/SHA-3> (Last accessed March 2019).
- [59] MDN: Uint8Array, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array (Last accessed March 2019).
- [60] SHA-256 Cryptographic Hash Algorithm , <https://www.movable-type.co.uk/scripts/sha256.html> (Last accessed March 2019).
- [61] BS58 JavaScript component, <http://cryptocoinjs.com/modules/misc/bs58/> (Last accessed March 2019).
- [62] Mozilla, <https://www.mozilla.org/en-US/> (Last accessed March 2019).
- [63] PDF.js: A general-purpose, web standards-based platform for parsing and rendering PDFs., <https://mozilla.github.io/pdf.js/> (Last accessed March 2019).
- [64] Wikipedia: Content delivery network, https://en.wikipedia.org/wiki/Content_delivery_network (Last accessed March 2019).

- [65] PDF.js - dist: Generic build of PDF.js, <https://github.com/mozilla/pdfjs-dist> (Last accessed March 2019).
- [66] React Documentation: Refs and DOM, <https://reactjs.org/docs/refs-and-the-dom.html> (Last accessed March 2019).
- [67] Amazon S3: Object storage, <https://aws.amazon.com/s3/> (Last accessed March 2019).
- [68] Wikipedia: XMLHttpRequest, <https://en.wikipedia.org/wiki/XMLHttpRequest> (Last accessed March 2019).
- [69] MDN: Promises, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (Last accessed March 2019).
- [70] Compromise: natural-language processing in javascript <https://github.com/spencermountain/compromise> (Last accessed March 2019).
- [71] Model-View-Controller Architecture <https://en.wikipedia.org/wiki/Model-view-controller> (Last accessed March 2019).
- [72] Bcrypt <https://www.npmjs.com/package/bcrypt> (Last accessed March 2019).
- [73] Express Session <https://www.npmjs.com/package/express-session> (Last accessed March 2019).
- [74] CSG Publications <http://www.csg.uzh.ch/csg/en/publications.html> (Last accessed March 2019).
- [75] Wikipedia: Precision and Recall https://en.wikipedia.org/wiki/Precision_and_recall (Last accessed March 2019).

List of Figures

3.1	PDF Parsing Algorithm	13
4.1	EUREKA Encoding Algorithm - Screenshot	23
4.2	EUREKA Decoding Algorithm - Screenshot	25
4.3	PDF Parser - Summary Component	33

List of Tables

3.1	Different Alphabets for Encoding	8
3.2	Encoding Scheme Experiment Results	9
3.3	Weighted Scoring System Results	9
5.1	EKA Address Retrieval - Precision and Recall	36
5.2	Main Authors Retrieval - Precision and Recall	37
5.3	References Retrieval - Precision and Recall	37
5.4	Cited Authors Retrieval - Precision and Recall	37