UNIVERSITATEA POLITEHNICA BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Căutarea Semantică în Biblioteci Virtuale și Corecții Automate ale Textelor

Roxana Frunză

Coordonator științific:
Conf. dr. ing. Mihai Dascălu

BUCUREŞTI

2019

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Semantic Search in Digital Libraries and Automatic Text Correction

Roxana Frunză

Thesis advisor:
Conf. dr. ing. Mihai Dascălu

BUCHAREST

2019

CONTENT

SINOPSIS

În ultimii ani, s-a observat apariția tendinței de digitalizare a documentelor stocate în format fizic, scopul principal fiind salvarea acestora într-un format care facilitează căutarea de informații. Cu toate acestea, datorită numeroaselor programe software care pot fi folosite în procesul de digitalizare și a structurilor variate pe care le au cărțile, este necesară găsirea unei soluții de stocare a acestor documente cât mai eficiente. Proiectul de față oferă o soluție de extracție a informației, de detecție automată de greșeli din textul extras si un motor de căutare semantică pentru găsirea ușoară a informațiilor. Procesul de extracție analizează elemente care, în general, se găsesc în toate cărțile cum ar fi cuprinsul, notele de subsol, tabelele și imaginile. Acestea sunt combinate si rezultă într-o structură definită în prealabil care nu depinde de aspectul inițial al documentelor. După procesarea cărții, există posibilitatea de a detecta și de a oferi sugestii pentru orice greșeală de scriere care poate apărea la crearea documentului. În cele din urmă, oferim un motor de căutare semantică pentru a găsi ușor informații în cărțile salvate. Acesta se va uita la expresia căutată și contextul semantic al acesteia pentru a oferi rezultate similare din toate cărțile salvate în baza de date.

ABSTRACT

The past few years have seen the emergence of the trend for digitizing paper-based documents with the main purpose of better information storage and retrieval. However, due to the many software that can be used in the digitization process and the various layouts books have, it is necessary to find a way to store these documents as efficiently as possible. This thesis presents a solution that offers a way to extract information, automatically detect mistakes from the extracted text and a semantic search engine for easy information retrieval. The extraction process looks for specific elements that, generally, are found in every book, such as tables of contents, footnotes, tables, and images. These are combined together and result in a pre-defined structure independent of the original layout. After processing the book, we offer the possibility to automatically detect and find suggestions for any spelling errors that might appear in the creation or extraction process. Finally, we provide a semantic search engine for easy information retrieval from processed books. Our project looks at the query and its meaning and offers similar results from all the books that were saved in the database.

1 INTRODUCTION

1.1 Background

1.1.1 Document Formats

The use of digital documents has allowed users to manage information in a more effective manner. Electronic documents have certain advantages compared to printed books: easy storage, cost-effectiveness, and search convenience. This type of document is most suitable for libraries, as the same book can be easily shared between multiple people, removing the limitations caused by physical books.

The Portable Document Format (PDF) has become a widely used format for electronic books thanks to the possibility of representing the data independent of the operating system used to create it or of the one that is used to view it [1]. Thanks to its popularity, PDF has been widely used in different domains, especially where archiving is required [2].

The introduction of PDF/Universal Accessibility (PDF/UA) [3] has led to the creation of documents where the information is saved in a structured format. With this standard, there is a clear distinction between the parts representing the intent of the document's author and the elements which don't have any semantic significance. All structures type must follow a standard format and all images that represent real content may be referred to in the text using a tag as well. However, as there are numerous documents which were created before PDF/UA was introduced, changing older documents in this format is still a problem looking for a solution.

At the most fundamental level, a PDF document is made up of a stream of bytes that can be grouped together to form objects that will be later decoded with a filter in order to produce the original data. However, there is no rule specifying in which order these objects are encoded or what an object represents (e.g. it can represent only a character or a full line of text). This could become an inconvenience, as it's usually preferred to go through the text in the logical reading order rather than the encoding one.

1.1.2 Automatic Spell Checking

In the past few years, there has been an increase in storing textual data in an electronic format. This trend has been noticed in the use of scientific articles and libraries which choose to store books in an electronic format [4].

However, data with spelling errors are harder to process and will lead to even more inaccuracies in subsequent processing steps. If we think about extracting the information from a book which people will later use for looking up information, unfixed errors that appear in the data may lead to unmatched results in the query. A standard full-text search will not give every result possible as a misspelled version of the query will not be taken into consideration.

The length of the electronic document and the brain tendency to deduce the word based on context rather than reading it letter by letter [5] makes a manual spell correction system inefficient and prone to errors. This has led to the necessity of creating a software that automatically detects and corrects spelling mistakes.

A basic spell-checking system should determine if a word exists in the language vocabulary and offer suggestions for a correct form. However, some mistakes can result in words that exist in the dictionary but are misused in the context (e.g. *advise* and *advice*). An automatic spell-checking system should focus on correcting not only vocabulary mistakes but also fixing contextual errors like the misuse of homonyms or grammatical errors. Due to the multiple languages that may be used to write an electronic document, the spell-checking system should be able to support multiple languages.

1.1.3 Semantic Search

While electronic documents can store any kind of content like magazines or works fiction, scientific books and articles might be the most popular thanks to the fact they can be efficiently be distributed between different individuals. As a result, a lot of these documents might be used for research where it may be useful to have a search engine to find the required information across multiple documents efficiently.

Semantic search offers a solution more complex than simply checking if a word or expression is found in a document. It enables the search engine to offer more accurate results by looking at the word not only as a sequence of characters but also at what it represents with its meaning [6]. Semantic search systems and research can be classified by the data type and the search paradigm [7]. The data type refers to natural language text, knowledge bases or a combination of these two. While the text is considered a collection of documents written in the natural language, knowledge base refers to records in a database that stores what is known about this world. There are three search paradigms used to describe the query that the user asks: keyword search (searching for a few words), structured search (queries in languages like SQL) or natural language (a complete question).

1.2 Problem Description

The digitization of books has led to the apparition of several issues related to document storage, search, and text correction. Along with the increase in the number of books saved in an electronic format, the question arises as to how information can be saved in an easy-to-access way making information search as effective as possible.

While nowadays a book is most likely published in both physical and electronic format, there is the demand to digitize previously published books that were published only in physical form. The emergence of technologies like Tesseract [8] offered the possibility to save physical documents as PDFs. In order to process the text and perform other operations like table or image recognition, it is essential to extract the text from PDFs correctly in the logical structure of the text.

A PDF document saves not only the textual data but also font name and size, coordinates for the text and other visual elements. In order to facilitate further processing on the extracted text, we want to have a certain structure of the document by knowing chapter titles and each chapter’s content, identifying tables and image extraction. Due to its structure, a PDF document doesn’t distinguish this kind of information. For example, a table is seen as two completely separate elements: lines of text representing the data inside the table cells and graphics elements representing the lines that delimit the table cells. This causes the necessity of an algorithm that correctly identifies tables in order to save them in a user-friendly format.

On a scanned document, we can use Optical Character Recognition (OCR) to extract the text in an editable format. The process of converting books into an electronic file is influenced by the quality of the original documents. The first versions of page-reading systems succeeded in recognizing characters and words with an accuracy of over 90% [9]. The evolution of OCR based system can improve accuracy with different techniques over 99% [10]. For example, assuming an average length of words of 5 character, even with a 99%-character accuracy, one in 20 words might be spelled incorrectly. Additionally, the distance between the letters of the same word that may vary depending on the original document may lead to incorrect words being identified. These are only a few of the factors that influence text extraction. A more detailed paper about the accuracy of text extraction in historical books can be found in [11]. Because of these challenges, book digitization is prone to spelling errors that need to be fixed before storing the book in its final state.

While looking for a word or an expression in a text can be quite simple, this limits the results to those that perfectly match the input. For languages where there aren’t many forms derived from the root of the words, this might not seem a problem. For example, in English, when searching for a noun we can do only two queries: one for the singular form and one for the plural form, and most likely will get satisfying results. However, for Italic languages, where besides singular and plural forms there are multiple articulated forms, searching independently for each can become inconvenient. As a result, we want a search engine that automatically looks at all the forms of the word from the query. In addition, when searching in a book, we might be more interested in the information we are looking for rather than a perfect match. Ideally, a search engine will look behind the meaning of the query and offer similar results, not only a perfect match.

1.3 Goals

This project has three main goals:

- Extract the text from a PDF document and identify structure elements. such as chapters, footnotes, tables, and images;
- Automatically check and correct spelling errors;
- Search based on queries across multiple books using a semantic search engine

1.3.1 PDF Text Extraction

Starting from a PDF document, the main purpose of this stage is to create a basic structure that is independent of the initial file structure. This step includes identifying the titles of chapters, their text, including footnotes in the place where they are referenced and removing unnecessary data such as page numbers, headers repeating the book title or the chapter title. Additionally, we want to identify tables, extract the text inside the cells, save them in a tabular format and identify where it is placed in correlation with the rest of the page elements. Images will also be extracted and saved for later reference.

At the end of this step, we generate a structured document that may be stored more easily. This facilitates the storage of a large number of documents in a structure which is the same for all files no matter the initial structure of the PDF.

1.3.2 Automatic Spell Checking

After processing the document and formatting it in a standard structure, we want to identify potential spelling errors that might appear in the extraction stage or that might exist since the publication of the document. This step is based on an automatic spellchecking tool adapted for the language of the book and identifies any potential spelling errors and will offer different suggestions. We look into different spell checkers that will try to find errors for the words that are not in the language thesaurus, grammatical or homonyms errors.

This step automates the text correction process leading to a faster and more accurate document processing by identifying errors that might be missed by the human eye.

1.3.3 Semantic Search

At this step, we look into different ways to analyze a search query and return more than fragments that match perfectly with the query. We want to find the meaning behind the query, analyzing the words to offer similar results that might be of interest. Every result has a score based on how accurate is based on the initial query and the results are shown in descending order according to their score.

1.4 Outline of the Thesis

This thesis presents a project with the goal to save a document in a formatted structure, automatically detect and offer suggestions to spelling errors and offer a semantic search engine to look for information in processed books. Chapter 2 presents the state of the art discussing the current research context and alternatives to this project. Chapter 3 focuses on our method, giving details about the architecture and the algorithms used. Towards the end, we present in Chapter 4 the results and explain how we reached them. Chapter 5 offers a conclusion to our project summarizing what we've done and what possible areas for improvement there are.

2 STATE OF THE ART

Since the introduction of PDF format and the beginning of publishing books in an electronic format, many documents published earlier have been scanned and converted in PDF files. From a scanned document as bitmap images, a PDF document text extraction can be achieved using OCR. Over the years, scientific literature has studied how OCR accuracy and speed can be improved. Solutions offered may include combining results from multiple OCR systems followed by string alignment [12] or scanning the same page multiple times and running a "consensus sequence algorithm" [13], leading to OCR systems with accuracies up to 99% [10].

Discovering the logical structure of a PDF document has been a well-studied problem for more than 20 years. In the beginning, solutions for extracting the layout of the structure have been classified as *top-down* [14] and *bottom-up* [15]. Afterward, research has focused on extracting the information according to the layout for scientific papers. In general, scientific articles follow the structure known as *AIMReDCaR* (Abstract, Introduction, Methodology, Result, Conclusion, and References) [16]. This enables the possibility to create algorithms that automatically extract information based on pre-defined structure.

Although contemporary OCR systems offer accuracies up to 99%, various reasons such as low quality of the bitmap image can lead to spelling errors. As a result, along with improving OCR systems, research has also focused on finding techniques on how to automatically detect and correct words in the text. The automatic spell checking research focuses on three main problems: (1) nonword error detection, (2) isolated-word error correction and (3) context-dependent word correction [17]. For each of these problems, different solutions have been presented in specialized articles. Researchers have offered multiple answers for all of these problems including dictionary lookups, the usage of n-grams [18] and weighted finite-state frameworks [19]. In recent years there has been a tendency to look for automated checking solutions in other languages than English such as Arabic [20] or Hungarian [21] and looking up for solutions for fixing contextual and grammar errors [22].

The research regarding semantic search has classified the systems using two dimensions: data type and search paradigm, resulting in nine categories: keyword search on text, on knowledge bases and on combined data, structured data extraction from text, from knowledge bases and from combine data, question answering on text, on knowledge bases and on combined data [7]. For each of these categories, there have been different amounts of research done, with the first category being the most well documented and the last one having still relatively little research to date. Basic solutions in matching include lemmatization or stemming, synonyms and proximity [23]. Structured data extraction on text focuses on extracting relationship from natural language and construction of base knowledge [24]. Currently, semantic search is still growing. In the years to come, we will see the knowledge databases grow comparing to the present and basic techniques which are now used will work in a more effective way.

Moving forward, we will not get into details about the OCR systems and how they can be used in document structure extraction as this topic is beyond the scope of this thesis. This project starts from the premise that the books used as the data set were created with an accurate enough OCR system for the text to be extracted with minimal errors.

In the following section, we primarily focus on projects that offer a solution to transforming the PDF text in a structured form by strictly analyzing the text format and other related information. We move on to projects focused on improving the extracting text by identifying and offering suggestions for potential mistakes and, finally, we present recent work that offers different kinds of solutions for semantic search.

2.1 PDF Text Extraction

Document structure analysis has been a field of interest for several years leading to various solutions and research papers. The multiple formats that a PDF document can be found in led to plentiful algorithms using different techniques. Ramakrishnan [25] and Jiang and Yang [26] offer solutions using bounding boxes in order to identify if a document is in a single or double column format and later unifies the extracted text based on the proximity. Anjewierden [27] suggests a statistical analysis based on geometry and textual content. Wu, *et al.* [28] offer a framework that proposes to integrate multiple algorithms that extract different semantic entities such as book title, author, header, citations and figure captions.

Gao, *et al.* [29] proposes an algorithm to extract structure from books in three main steps. Firstly, identify areas that should be treated together based on the idea they have the same format and group them in clusters. Afterward, eliminate blocks that might have been wrongly put in a cluster by applying the XY-cut algorithm and, finally, order blocks in logical reading order. The analysis in the first step happens at three levels: block level, page level which takes into consideration the information from a block and its neighbors, and document level which reflects features from blocks within the same cluster. Applying a learning method, called Support Vector Machine, text blocks are labelled accordingly to their three-level features previously identified. Using a weighted bipartite graph created to model the blocks and their probability as neighbors, the reading order is deduced.

CERMINE [30] accepts a scientific publication in PDF format and saves the information using the XML format. This solution focuses on three processing paths: (1) basic structure extraction, (2) metadata extraction, and (3) bibliography extraction. The first path focuses on character extraction along with their page coordinates and dimensions, page segmentation to construct document’s hierarchical architecture structure, reading order solving and classifying document’s zone in four main categories: metadata, body, reference, and other. Metadata extraction consists of classifying zones in specific metadata classes and extracting atomic metadata for each labelled zone. The final path focuses on dividing the reference zones in individual references strings and extracting metadata information from them.

Besides the solutions that focus on offering an extraction system that includes parsing multiple elements like headers, footnotes, figure caption, there are also solutions that focus on a specific step during the extraction process. For example, [31, 32] offer solutions for table of contents extraction from heterogeneous books and scientific articles, with the former being able to identify both chapters and subchapters.

2.2 Automatic Spell Checking

The problem of finding techniques for spell checking has been researched in parallel with OCR techniques, focusing on two main directions: spell detection and spell correction. Pirinen. and Lindén [33] offer a spell checking functionality based on a finite-state library HFST [34], while Hodge and Austin [35] offer a methodology based on two Correlation Matrix Memories: one to store the words for n-grams and the Hamming Distance for the typing matching step and one to store phonetic codes used in the phonetic matching step. In the following paragraphs, we will go into more details about in some more recent projects.

Korektor [36] is a statistical text correction tool with no language-specific parts other than the trained models, making it suitable for multiple languages with available resources. Authors focused on developing a model that would work on the Czech language. This paper looks at the task of spelling correction as a noisy-channel model. A transmitter sends a sequence of symbols, with some symbols being transmitted erroneously, leaving the receiver with the task to correct the information using the knowledge of the source and the channel properties. The source model is described using features based on language model probability (words form feature, morphological lemma feature, morphological tag feature), while the error model considers that between the misspelled words and the correct word exists only a single edit operation with each edit operation have a distinct probability.

Priya, *et al.* [37] present a detection and correction system based on 1,2 or 3-character sub-arrangements of strings, called n-grams, and the edit distance referring to the minimum number of insertions, deletions, substations, and transpositions necessary to transform one string into another. In this method, input and output pairs are specified as training data from which rules will be extracted that later will be used in a binary classifier to specify whether a rule should be applied to the text or not.

Schmaltz, *et al.* [38] propose a solution to identify grammatical errors at the sentence level based on a binary classifier for prediction and a sequence-to-sequence model trained for error correction. For the binary prediction task, the paper proposes two encoder-decoder architectures: word-based and char-based. The encoder neural network has the goal to summarize the source sequence and the decode neural network generates a distribution and tags the source at each step using deletion tags for tokens that should be eliminated and insertion tags for tokens that should be added.

2.3 Semantic Search

Semantic search is a well-documented field, with research focusing on finding solutions on query matching and recommendations across various types of documents. Lee and Tsai [39] offer a solution for an engine to look through web pages by offering an interactive solution that adapts gradually by looking at the meaning the user is thinking of. Syeda-Mahmood *et* *al.* [40] propose a solution to look through XML repositories using both name and type semantics to determine the equivalence between attributes.

Guha *et al.* [41] propose a solution starting from the Semantic Web which is based on the idea of creating relations not only between Web resources but between different kinds of resources such as people, events and organizations. They focus on augmenting search with data retrieved from the Semantic Web and improve the search results by understanding what the user means with their query. For the first part, they want to understand the meaning of the query and determine what data is relevant from the Semantic Web. The application runs as a client of TAP interface which gives the option to map the search terms to the Semantic Web directed graph. The application decides which information to retrieve by creating a subgraph by walking to the main graph in a breadth-first order for a limited number of nodes starting from the previously found nodes. For improving the search results using Semantic Search, they ask the user to tell the engine if the results match their intended meaning so they can filter the results by using a knowledge-based approach.

Ceglowski, Coburn, and Cuadrado [42] proposed a solution for searching through a document repository starting from latent semantic indexing (LSI). This method is based on a contextual network graph, where the nodes are documents and terms. The main principle is that every term has links to the node documents it appears in and every document has links to all the terms is contained in that document. An edge between a document and a term has the weight equal to the frequency values in the corresponding field of the term-document matrix. The search process consists of starting from a query node and moving to the connected nodes.

SEMEDICO [43] is a search engine designed to look for information in articles belonging in the biological sciences field. This solution involves multiple processing steps such as resolving acronyms analyzing genes/proteins relationships and involves indexing documents and all the concepts present in them in Elasticsearch. When searching for a query, each result receives a score to prioritize the results. Firstly, each result receives a score based on linguistic signals (e.g. expressions in the original document such as “we believe”, “might” or their negated forms) to calculate the likelihood value that the information matches the query. Furthermore, results are then ordered by using the proximity of each token from the search query in the searched text.

2.4 Relevance for the Current Research Context

Previous sections have presented similar solutions to this the three main parts of this project: PDF layout extraction, automatic spell checking, and semantic search. While currently there are multiple publications for each part of our project, there hasn’t been developed a solution that integrates all three. We offer a solution that extracts text from PDF books, detects and suggests spelling errors and offers an engine to search for information in the stored books.

PDF layout extraction is similar to [29]. Our solution looks at the information for each part of the page and groups them together. One important step in the extraction process that it’s missing from other works is table recognition. Most solutions start from the assumption that a table is preceded or followed by a line of text that clearly specifies table information. However, we look into table recognition by identifying the lines that delimit the table and saving that information in JSON format.

For spell checking, we offer a similar solution to those already being researched by tokenizing each sentence into words, finding misspelled words that aren't in a dictionary and offer suggestions from n-gram data. For semantic search, this project offers a search engine alike [42] by relying on Elasticsearch indexing and apply an analyzer to look for synonyms to the words in the query.

2.5 Integrated Technologies

In order to implement this project, we have used Apache PDFBox [44] for PDF document text extraction, Tabula [45] for table recognition and data extraction, LanguageTool for Java [46] as the spell checking tool and Elasticsearch [47] for indexing and storing the documents.

2.5.1 Apache PDFBox

Apache PDFBox is an open source Java library that allows to create, edit and encrypt PDF documents and advanced extraction of text, images, and forms, including their metadata such as font size and page coordinates. Being named an Open Source Partner Organization of the PDF Association [48], the possibility to integrate with Lucene, the number of features previously mention and also the possibility to easily personalize the extraction process make PDFBox a reliable solution for our project.

PDFLib TET [49] offers a commercial tool extraction text alternative to Apache PDFBox, both having similar features. However, [28] performed a baseline comparison between these two using 1000 PDF non-academic documents with extractable text. Out of the data set, both couldn't extract data from 40 documents, while TET couldn't get data from an additional number of 17 documents. Even if the difference is relatively low, considering TET is also a paid software, moving forward, we use PDFBox.

2.5.2 Tabula

As PDFBox doesn't offer the feature of correctly identifying tables, we need a technology to extract tables respecting the initial layout and save them in a format where each cell text can be easily identified. For this, we use Tabula, a tool which detects table and extracts data to save them in Excel Spreadsheets, CSV or JSON files. Tabula can identify tables with or without lines and rows separators and supports tables with multiline rows [50]. As, in our project, we focus on books with a heterogeneous format, this is useful as we don't have a pre-defined table structure in our documents.

An alternative technology considered is TrapRange [51], a solution that extracts tables in a matrix of strings or in the HTML format. The main drawback to this project is that is a solution only to extract data from the tables, not to detect them. In order to correctly extract rows and columns, the user needs to manually give as input the page of the tables and indexes of lines that don't belong in a table. Additionally, this solution extracts rows starting from the premise that words with the same horizontal alignment belong to the same row, making it an unfit solution for tables with multi-line rows.

Taking into consideration the shortages of TrapRange and the necessity of either manually extracting lines that don't belong in the table or building a solution to detect tables, we use Tabula in our project implementation.

2.5.3 LanguageTool

LanguageTool in Java is a spell checker tool that offers spelling detection and correction, grammar checks and is highly customizable by offering the possibility to add new rules to look for in the text. Most of the languages supported by LanguageTool have a spell-checking system based on the dictionaries used by LibreOffice/OpenOffice.

Alternatives studied for this tool were Jazzy Spell Checker [52] and Hunspell [53]. Jazzy Spell Checker is offering a solution based on phonetic matching and sequence comparison. However, as this comparison shows [54], Jazzy fails to offer suggestions for concatenated words or contextual errors and doesn't offer support for multiple languages. Hunspell is a spell checker tool that offers extended support for multiple languages, morphological analysis and stemming. However, a report shows as far as performance is considered, Hunspell is much slower that Language Tool, with rules from the latter analyses 1400 sentences on average while the former processes only half with the number being much lower when suggestions system is turned on [55]. Taking this into consideration and the facts that LanguageTool offer support for multiple languages with a variety of rules to detect mistakes for every one of them and the possibility to customize by adding our own rules, we use this technology for the spell-checking system.

2.5.4 Elasticsearch

Elasticsearch is a search engine based on a database that offers the possibility to store, retrieve and manage document-oriented or semi-structured data [56]. It provides users an easy and fast way to retrieve data by offering a REST API to organize, get, search and edit the document collection. Using the Lucene's inverted-index [57] which maps every term with the documents it appears in, and considering the better performance in search queries comparing to the classical SQL management systems [56], Elasticsearch is the most suitable for our project.

An alternative is the open-source search platform, Solr [58] which offers similar features to Elasticsearch in terms of indexing and searching. The reasoning behind our choice is that Elasticsearch is faster in terms of search time, with a mean search time of 0.10s and with 99% searches under 0.22s, while for Solr, it was 0.12s and 0.54s [59]. Taking this into consideration and other differences [60] such as dynamic nature of the shard and user-friendly APIs like Kibana that in the long run might lead to better performance, we have decided to move forward using Elasticsearch.

3 METHOD

The following chapter focuses on the proposed solution by offering a high-level overview of our project, describing the workflow for each step in the layout extraction process, how to connect and run the spellchecker and the lookup procedure. We look into solutions for each of these steps, presenting a general description, algorithms used, special configurations that were necessary and motivating our choices at certain points in the developing process.

3.1 Corpus

The corpus used in this project consists of 50 books in Romanian from various domains such as philosophy, history, and literature. The quality of the documents varies significantly due to the age of documents. There are recent books where the information is clearer and easier to understand. However, there are documents published over 50 and even 100 years ago which means they have a lower quality of text and images and may also contain handwritten notes. The book collection is saved in the PDF format and they were created from scanned documents using OCR so the text can be extracted from them. The OCR process was realized using different software resulting in different document properties that influence the extraction process.

3.2 Architecture and Processing Pipeline

This project consists of three main parts that are completely independent of each other. One part of the application focuses on the layout extraction process. This is where the table of contents, footnotes, images, and tables are identified and arranged in a structured form. The second part consists of a server that is responsible for the spell-checking system which is called when we want to identify errors in the uploaded document. The final part consists of an Elasticsearch database that is used to index all processed documents. When a new query is initiated, we look for matches in this database. As *Figure 1* shows, all three components are managed by the PDF Processing Application which can be accessed from the web browser. The spellchecking component connects to a local server on port 8081 where the main spellchecking application is, while for the search process we connect to an Elasticsearch database and look through the books index.

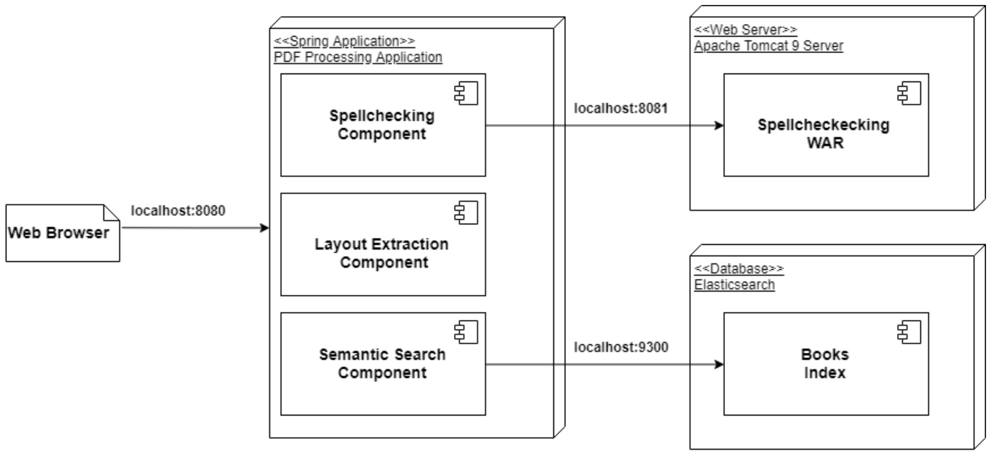


Figure 1. System Architecture Overview.

There are two branches in the workflow of the project: one for document processing and one for the search. The former focuses on layout extraction and spell checking, while the latter is responsible for searching the query in the stored information. Moving on we will present in more detail each step in these two branches.

3.3 Document Processing Pipeline

Figure 2 shows the main steps that happen when processing a PDF document. After the file was uploaded, the extract layout step focuses on creating a JSON object containing each chapter title, their contents with inserted tags for images and tables, and extracted data from tables.

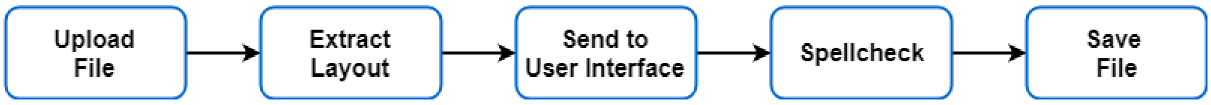


Figure 2. The workflow for document processing.

The JSON object is sent to the user interface where it's displayed using a TinyMCE text area [61]. For this step, we are transforming the JSON object to an HTML format and replacing the table tags with the corresponding data by creating an HTML Table Object. This format is interpreted by TinyMCE and displayed in the text area. At this point, the user can make further modifications such as marking chapter titles, editing text paragraphs or inserting, modifying and deleting tables. The spell check is an optional step and can be ignored at the user's discretion. If spellchecking is chosen, a request is sent to the spell checker server by sending the whole text and the language in a JSON format. The spell checker analyses the text and sends back a list of potentially misspelled words and their possible fixes. In the end, saving the files means extracting the text from TinyMCE in HTML Format, replacing tables with a corresponding tag and changing it back to JSON format. This format is sent to Elasticsearch to be indexed.

3.3.1 Layout Extraction

The layout extraction process can be seen as a processing pipeline with multiple steps resulting in a formatted structure of the document. *Figure 3* presents an overview of the workflow of this process.

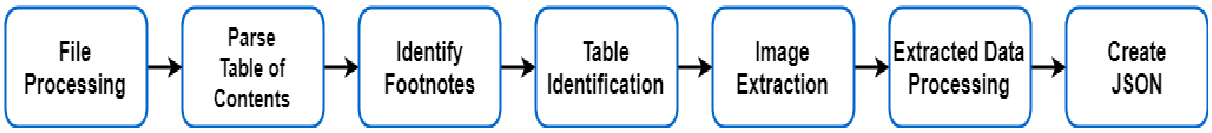


Figure 3. The workflow for layout extraction.

The document processing starts by loading the document from the stream in a PDDocument instance. This instance is used in the processing steps since it allows us to extract the whole text, from a certain area given by coordinates or page by page, paragraph by paragraph or line by line. These properties are used when we extract the layout elements and group the text by chapters.

Parsing the table of contents means looking for pages which are placed at the beginning and at the end of the document and respect a table of contents template. Footnotes identification means looking at the end of each page to find certain references that were cited in the text on that same page. Table identification refers to identifying tables on each page and applying an extraction algorithm for each result. The image extraction refers to looking at the graphics on each page to identify pictures. At this step, we also look to merge close enough pictures that might be part of the same element but were extracted separately.

After extracting all required elements, we look at the text page by page, removing unnecessary lines and merging together lines in order to form paragraphs. In the end, we create a JSON object where we save the chapter title, paragraphs, and extracted tables.

3.3.1.1 Parsing the Table of Contents

The table of contents identification and chapter extraction is based on a similar structure these pages have. A general rule we have noticed is that most of the tables of contents have lines that end with digits, representing the start page of that chapter. This is used when we try to determine pages that are part of the table of contents and in the identification of chapter titles.

Algorithm 1 shows the process applied in order to identify the first page of the table of contents and extract the text from several pages which are later used to identify chapter titles and page numbers.

Algorithm 1: Identify and extract table of contents pages

```
1  $startPage \leftarrow -1$;
2  for $i = 0$ to $numOfPages$ do
3      $currentPage \leftarrow pages[i]$;
4      if currentPage is ToC firstPage then
5          if $i <$ beginLimit or $i >$ endLimit then
6              $startPage \leftarrow i$;
7              break;
8  if startPage == -1 then
9      table of contents doesn't exist;
10 else
11     $endPage = startPage + maxNumberOfTocPages$;
12     extract text from startPage to endPage
```

Finding the first page starts from the premise that a book marks the start of the table of contents with a keyword specific to the language the book is written in. When processing a page, we look through a list of keywords and check if there exists a match. However, due to the lower quality of some books, a match between any of the keywords and the text on the page wasn't enough as the page might have had the words split in a various number of words due to white spaces. As a result, we created a regular expression where we check if the keyword with any number of spaces between its letter is found on the page.

After finding the keywords on a page, we want to make sure the page identified is really part of a table of contents. As a keyword can be used in the content of a book, we use a regular expression that matches any line ending in numbers and consider that at least 5 matches and the presence of a keyword mark that page as part of a table of contents. An extra step in the first-page identification consists of the checks made at line 5 in Algorithm 1. This is a precaution in case the previous steps return multiple pages. Generally, the table of contents can be found at the beginning or at the end of the book, so, when we find a match, we check to be in the first or last ten pages.

In order to determine the previously mentioned rules, we have looked at 20 documents. The only requirement is for the table of contents to have the same format throughout the whole book. In this list of documents, there are three keywords that mark the first page of the table of contents: “cuprins”, “cuprinsul” and “tabla de materii”, with the first word in the list being present in 15 of the documents. We look for all three keywords, with the possibility to easily add new ones. As for the minimum number of matches for the lines ending in digits, we have decided to choose 5 since out of the 35 pages part of any table of content, only 2 have less than 5 entries, with those usually being the last pages of a table of contents spread across multiple pages. We have decided to look at only the first and last 10 pages after looking at the data set and realizing that only one book has the table of contents pages outside of this interval.

In order to extract chapter titles, we apply a regular expression on the whole text. To create this regular expression, we have studied the same data set as before. Every document except one has entries that end in numbers. 15 of those documents follow a pattern by writing the chapter name, which may be preceded by a number, followed by a various number of full stops and ending with a page number. In four other documents, an entry consists of the chapter name, a various number of whitespaces, followed by a number. Only one document had a special format where the chapter name and number were separated by the ‘/’ character.

As a result, we created a regular expression to match one entry in the table of contents. The areas of interest are the chapter titles which is a sequence of any characters and the page number. In order to separate these two, we capture two groups. *Figure 4* shows how the used regular expression matches the most popular types of chapter entries. The parts highlighted in green and orange are extracted from the match using the group() function.

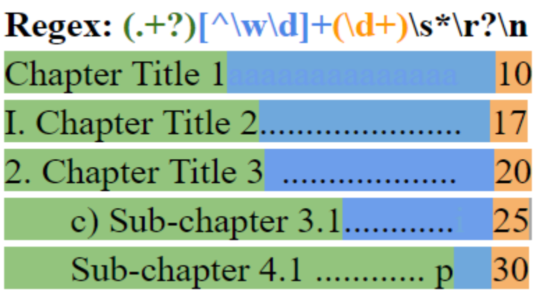

Regex: (.+?)[^\w\d]+(\d+)\s*\r?\n
Chapter Title 1 10
I. Chapter Title 2...................... 17
2. Chapter Title 3 20
c) Sub-chapter 3.1............ 25
Sub-chapter 4.1 .... p 30

Figure 4. Regular expression matches.

As *Figure 4* shows, an entry in the table of contents might start with a number or letter used to have an ordered list or with a various number of spaces. In this case, we have to separately remove these parts by using three regular expressions that look at the beginning of each line for digits, small letters or symbols specific for Roman numerals. A particular case we have encountered is the "p" symbol before the number page. In this case, the first group includes the separating marks between the chapter titles and the page number. This case is treated separately by removing any extra symbols that were included in the group for the chapter title.

After extracting the chapter title and its corresponding page number, we have to perform additional checks in order to be sure it has been identified correctly. This is necessary since closer to the table of contents pages, there can be pages with general book information such as the ISBN number, publisher name or publishing year. Before adding the chapter to the final list, we check to see if the identified page number is within the number of pages of the book. Furthermore, since tables of contents are ordered ascendingly by the page number, we always save the last number identified and when we extract a new number, we check to see if it's bigger than the one saved and only then we add it to the final chapters list.

3.3.1.2 Identifying and Parsing Footnotes

Footnotes are placed at the bottom of the page, usually preceded by a horizontal line that separates them from the rest of the page. A footnote can be made of one or more lines of text with the remark that the first line must start with a special character or a number so the reader can identify the place where this footnote is referred to. Our footnote identification process is based on these two mentioned rules. *Figure 5* shows an overview of the algorithm we applied to identify footnotes on the page.

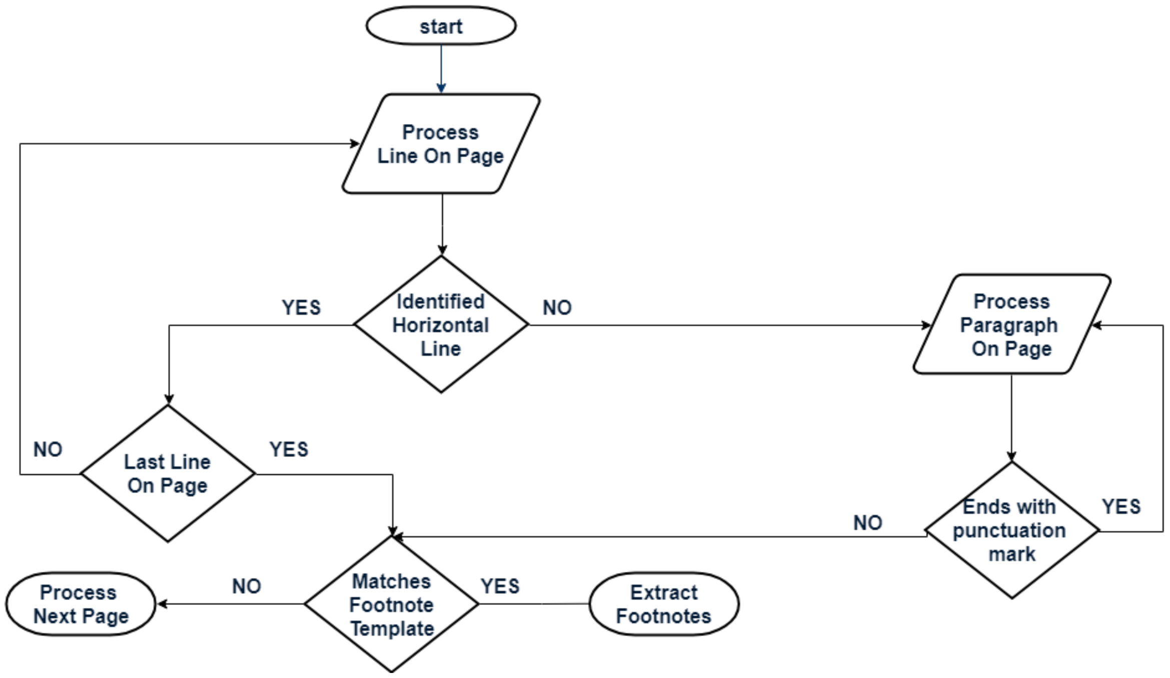

start
Process Line On Page
Identified Horizontal Line
YES
NO
Process Paragraph On Page
Last Line On Page
NO
YES
Ends with punctuation mark
NO
YES
Matches Footnote Template
NO
YES
Process Next Page
Extract Footnotes

Figure 5. Flowchart to identify footnotes.

Before starting the parsing process for footnotes, we first need to identify the lines of texts that are part of the footnote and then separate them in groups and store them in an array of dictionaries.

We start from the premise that footnotes are separated from the rest of the page by a horizontal line. A straight line is created using the following steps:

- Move to the starting point of the line and start a new path using moveTo(x, y) function;
- Draw a path from the current point to the end of the line using lineTo(x, y) function;
- Stroke the path previously created using strokePath() function;

At the end of the first two steps, the result is a path which is a set of coordinates that define the shape. It is possible to identify information such as width and height using formulas from analytical geometry. However, stroking the path results in the final shape which contains information such as the width, the height and the coordinates relative to the page with all this information being saved as a rectangle. As a result, if we analyze a line in the strokePath() function we get all the information needed to identify horizontal lines. At this point, we create a list with the horizontal lines which is a rectangle with the height close to 0.

The process presented earlier is done for each page in the analysis process resulting in a list of lines from headers, footnotes, and tables. As lines are processed from the top to bottom, we are interested in the last line processed as it is most likely the separating lines between text and footnotes. Using the line coordinates and page dimensions we create a rectangle that is used to delimit the area from where we extract the footnotes. The rectangle is defined using the following values:

- Width which is equal to the page width;
- Height which is equal to the difference between the page height and Oy coordinate for the extracted line;
- Bottom-right corner, with the Ox value equal to 0 and the Oy value equal to page height.

The previous algorithm offers a reliable solution in identifying and extracting footnotes when lines and rectangles are created using a graphics stream engine. However, for scanned books, depending on the creation process, lines and figures are considered an image or part of a bigger image. This led us to look for an alternative solution when we can't identify the horizontal lines.

When looking at the documents, the main difference noticed was the creation software used in order to create the PDF files. Table 1 shows a list of used software and either there were able to identify lines or not. If the software doesn't support OCR, most likely another software was used for the PDF creation step.

Table 1. PDF Creation Software Summary.

Software	Supports OCR	Identifies Lines
ABBYY FineReader	YES	NO
Adobe Acrobat Pro Paper Capture Plug-in with Clear Scan	YES	YES
Acrobat Distiller	NO	YES
Adobe Image Conversion Plug-in	YES	YES
GPL GhostScript	YES	YES
OpenOffice Org	NO	YES
Virtual PDF Printer	YES	YES

The main problems were the documents created using ABBYY FineReader. The problem with this type of documents is that there are multiple overlapping layers for the original scanned page, images, and extracted text. However, the footnote line is not considered an image in the OCR process so there is no layer to contain it except the original scanned page. However, the layer containing the original scanned page is seen as an image without object encodings for every element which makes line identification not possible.

Algorithm 2 offers a solution by looking at the paragraphs and check if they end with any of the following punctuation marks: full stop, question mark, and exclamation mark. The reasoning behind this check is if a paragraph doesn't end with a punctuation mark, it's because the continuation is on the next page. When analyzing a paragraph, if it's noticed that is continued on the next page, we check to see if it's the last paragraph on the current page. If that is not the case it means that the next paragraph is not part of the main text and can be either a footnote, page number or other chapter or book information.

```
Algorithm 2: Identify and extract footnotes using the paragraphs on page
paragraphs ← delimitPage(page, paragraphDelimiter);
foundFootnote ← false;
foreach paragraph in paragraphs do
    foreach line in paragraph do
        if !foundFootnote and !hasPunctuationMark and isFootnoteStart then
            foundFootnotes ← true;
            append line to footnote text
        else if foundFootnote then
            append line to footnote text
```

In order to differentiate the footnotes from other elements that might be at the bottom of the page, we look at the beginning of the line. As a rule, the footnote starts with a special character such as an asterisk or with a number each may or may not be between brackets. If the processed paragraph has a format like the start of the footnote, then we append the rest of the paragraphs on the page to the footnote text.

As the final goal to the footnote parsing is to add them into the text at the place they were referenced, we need to separate the footnote content from the rest. This is done using a regular expression with two groups, one for the content and one for the number or symbol used for identification. After the separation is done, we save the footnotes identified on a page in a dictionary where the key is the identification part and the value is the footnote content. We are guaranteed the key is unique because if two footnotes on the same page had the same number or symbol, it wouldn't be possible to distinguish where each of the two footnotes is referenced in the text.

3.3.1.3 Table Identification

Table identification is using Tabula algorithms and has two steps: table detection and table data extraction. The detection refers to identifying where a table is positioned on a page. For each table, there are 4 values that uniquely identify a table: width, height, x and y coordinates for the right-upper corner. The extraction refers to identifying each cell from the table by trying to delimit the area identified in the detection step into cells.

Firstly, we focus on detecting the tables on each page by applying the Nurminen Detection Algorithm [62]. *Figure 6* shows the six main steps followed by this algorithm.

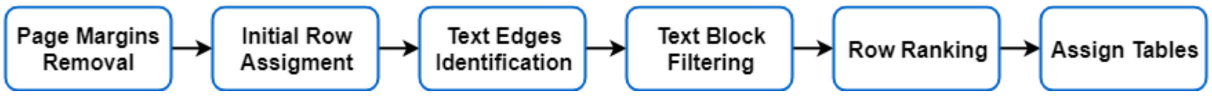


Figure 6. Overview of the Nurminen Detection Algorithm.

Firstly, the information irrelevant to the table data such as page numbers, logos or other publishing information is removed. The initial row assignment unifies rows that have similar vertical coordinates, while text edges identification looks for multiple rows that have the left and right limit similar. Text block filtering looks to identify blocks of justified text that so they won't be considered as columns in the table in the next steps. Row ranking has the goals to order each row on a page by the probability of being part of a table. In calculating this probability there are taken into consideration the number of text edges and the justified text blocks identified previously. The final step is to find the boundaries of the tables by looking for rectangular areas or by unifying the rows computed at the last step and placing them in a rectangular area. [62].

After applying the Nurminen Detection Algorithm on each page, we have a list of rectangles defined by the coordinates of their top-left corner, the width, and the height. These lists are saved as values in a map with page number as a key.

In order to extract the data, we have to process each of the rectangles and apply an extraction algorithm. Tabula offers two methods: stream, where the delimiter between columns is white space, and lattice, where the delimiter is a vertical line. For the former, Tabula looks uses different heuristics to check for whitespaces, while the latter looks at the visual elements of the document encoding. A more detailed comparison is offered in Table 2.

Table 2. Lattice vs. Stream in Extracting Table Data.

Method	Column delimiter	Advantages	Disadvantages	Additional Notes
Lattice	Vertical Line	- Can identify more accurately data from each cell - Can identify cells which have multiple rows of text and groups them together	- Imposes a fixed structure for the table	The lines between columns need to be elements that can be extracted from the stream. This method doesn't recognize lines from images.
Stream	Whitespace	- Offers flexibility as it can identify tables that don't necessary	- Cells with more than one line are split across multiple cells resulting in lines in the table with NaN values - Can identify false positives as lines with variant space dimensions can be identified as tables.	

For table extraction, we analyze each rectangle identified at the detection step using the lattice algorithm. For each table detected, we use the coordinates to create a new temporary page object that contains only the data from that rectangle. This guarantees us the area detected contains exactly one table and it doesn't merge multiple tables into one. On the extracted area of the page, we apply the extraction algorithm which has as a result an array of tables. However, since we specified an area each can contain only one table, this array always has exactly one element which is the table we wanted to extract. The extracted table is saved in a matrix structure as a JSON object. Besides the JSON Array containing all the tables, we also keep a dictionary for each page where the value is the array of tables identified on their page including, as it is used later in the process of inserting the table tag in the paragraphs at the right point.

For a more accurate table extraction process, additional steps would be necessary to identify what table template the processed document uses in order to choose the best extraction method. Moving on, we use the lattice method, as the stream method requires additional filtering steps to remove the false positives identified. We leave finding a method to combine these two algorithms for further developments.

3.3.1.4 Image Extraction

As shown in *Figure 7*, image extraction focuses on identifying images on a page, filtering images that might not be of interest, merging images that were extracted separately but are part of the same image and saving them to a local file. Extracting the images is an optional step and is only done when the user checks the "Extract Images" in the user interface. The extracted images are saved in PNG files, and a tag is inserted at the place where the image was extracted from. We also save images as objects using a class we created where, besides the image stream, we keep information such as width and height in pixels and points or image coordinates in the coordinate system determined by the page boundaries.

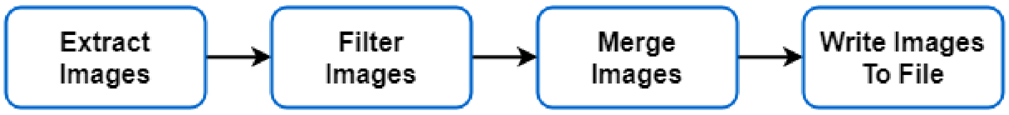


Figure 7. Workflow for the image extraction process.

The first step is extracting from the PDF stream the images objects. For extracting images, we look at the stream engine, being interested in the "do" operation and its operand. This operation is the one that adds elements such as images and forms to the stream. From the operand used in processing a page, we get a list of resources which are objects that offer not only a data dictionary but the image data as well. The image data is the one where we can find the stream of bytes that form the image. The advantage of extracting images when they are processed rather than looking at resources on every page is that we look for images in a recursive way, analyzing each layer, instead of looking at the top level only.

In the filtering step, we remove images that might have been created during the document creation process but aren't part of the actual content of the book. As a result, we remove any images that have the size close to the whole page dimensions, as these most likely are images extracted due to the creation process that contains the scanned page and are not considered a figure in the original book. In the end, we have an array of extracted images with all their data which is used in the next processing steps.

Merging the images is an important step as, due to the creation process, an image can be split into different images which is individually extracted. An image can be separated on the Oy axis resulting in two images that need to be merged vertically, one on top of the each other, or on the Ox axis resulting in two images that need to be concatenated horizontally, one next to each other. We have noticed this behavior in documents created using GPL GhostScript and we suspect it's caused by multithreaded rendering, which parses a page in bands rendered at the same time [63]. *Figure 8* shows an example of how an image is split amongst multiple pieces with each red rectangle being extracted as a separate image.

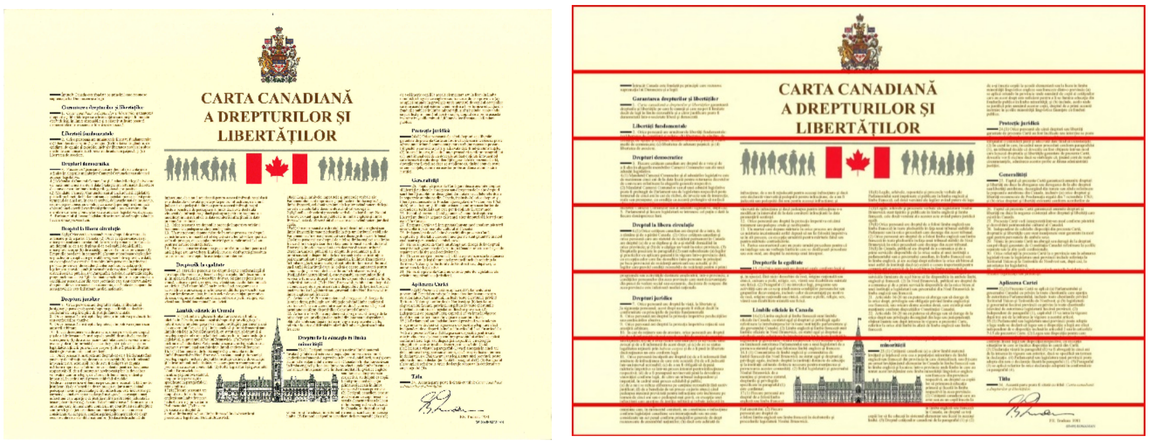

CARTA CANADIANĂ A DREPTURILOR ŞI LIBERTĂŢILOR
CARTA CANADIANĂ A DREPTURILOR ŞI LIBERTĂŢILOR

Figure 8. Original image vs. extracted images.

Figure 9 shows the workflow followed in the merging process. The merging algorithm looks at the list of images saved and compares their data to see if they should be merged horizontally, vertically or left separate. At this step, we look for similarity in the dimension and coordinates for the bottom right corner between the two images to determine how close they are.

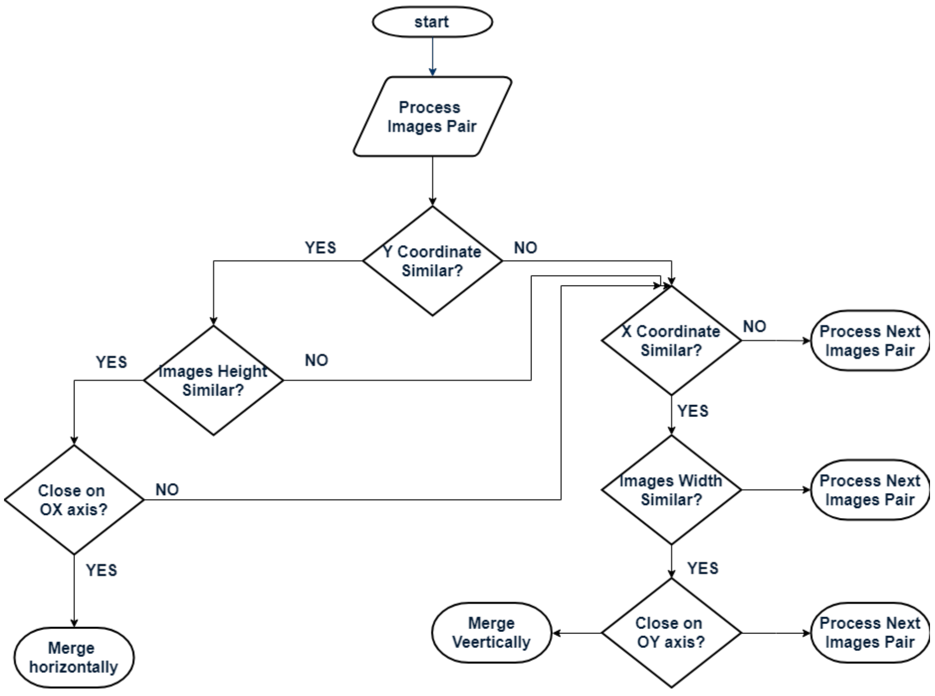


Figure 9. Workflow to merge images horizontally and vertically.

The merging process is applied on each page and looks at the list of images saved. It starts from the last image that is extracted and analyzes them in pairs to see either they should be concatenated or not.

Algorithm 3 shows the conditions that need to be true in order to merge two images one next to each other. In this algorithm, if we were to draw a vertical line between the two images, the first image is considered the image on the left of the line and second image the one on the right. For two images to be merged, we first check to see if they have similar height and bottom-right corners with a similar Oy coordinate. If these statements are true, it means they might be part of the same images. In order to be merged, they need to be next to each other, meaning the x coordinate from the right side of the first image needs to be almost equal to the x coordinate of the left of the second image.

Algorithm 3: Check if two images should be merged horizontally

```
input: image1, image2 - two image objects
if $image1.y == image2.y$ and $image1.height == image2.height$ then
    $image1End = image1.x + image1.width$;
    $image2Start = image2.x$;
    if $image1End == image2Start$ then
        return true;
```

The algorithm for checking if two images should be merged vertically is similar to the previous algorithm and can be seen in Algorithm 4. In this case, two images are on top of each other and if they were to be separated a horizontal line, the first image would be the one below the line and the second would be the one above the line. In both cases, the coordinates system is considered with the origin in the bottom-right corner of the page with the Oy increasing upwards and the Ox increasing to the right.

Algorithm 4: Check if two images should be merged vertically

```
input: image1, image2 - two image objects
if $image1.x == image2.x$ and $image1.width == image2.width$ then
    $image1End = image1.y + image1.height$;
    $image2Start = image2.y$;
    if $image1End == image2Start$ then
        return true;
```

In the end, we create an image where we add smaller images accordingly to the concatenation mode. If we merge two images horizontally, the final image has the biggest height and the width the sum between the widths of the two images. We choose the biggest height because, while they might be similar in height, they are not equal. We choose the maximum, so we won’t lose data from any of the images. If we merge them vertically, the result has the width equal to the maximum between widths and the height equal to the sum between the heights of the smaller images.

3.3.1.5 Processing Extracted Data

The final step before we have the book saved in a structured format is to process the information extracted until this moment and put it all together. At this point, we have a list of chapter titles, footnotes, extracted images, coordinates and data of tables. All these elements need to be integrated at the right point in the final structure of the document. Also, we look at the font information for each line of text in order to offer an alternative to identifying chapter titles when we are unable to identify a table of contents. *Figure 10* shows the main steps followed when processing the extracted data and creating the final structure of the documents.

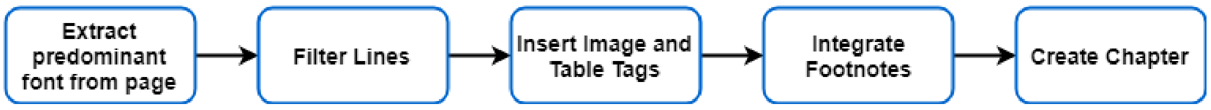
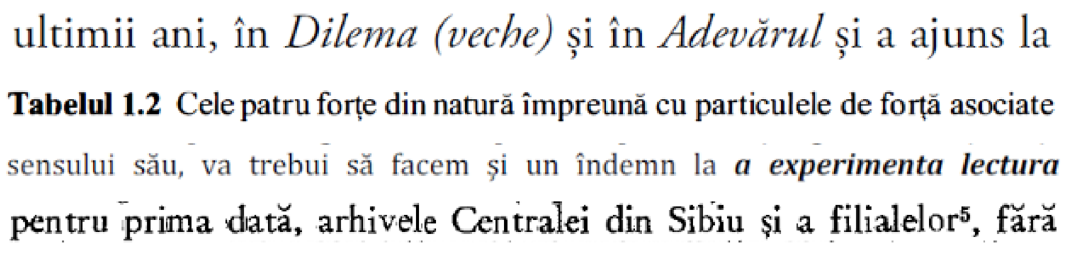

Extract predominant font from page
Filter Lines
Insert Image and Table Tags
Integrate Footnotes
Create Chapter

Figure 10. Flow for final processing steps.

We start by analyzing each page line by line. We customize the PDFBox text stripper which uses a series of process operations that divide the page into smaller and smaller parts. As we are interested to analyze each line, we use the writeString() method, the operation which uses the character as part processed.

Usually, the lines from the same paragraphs have the same font. However, there are cases where, within the same paragraph, there are used multiple fonts in order to highlight certain expressions, citations or footnotes. *Figure 11* shows some examples of such lines from our corpus. It can be seen that a line uses the same font except for a few words which may be written in italics, bold or superscript.



ultimii ani, în *Dilema (veche)* și în *Adevărul* și a ajuns la

Tabelul 1.2 Cele patru forțe din natură împreună cu particulele de forță asociate

sensului său, va trebui să facem și un îndemn la ***a experimenta lectura***

pentru prima dată, arhivele Centralei din Sibiu și a filialelor^5, fără

Figure 11. Examples of lines with multiple font styles and sizes.

Since we are interested in only extracting the paragraphs and chapter titles, we need to ignore any style within the same paragraph. As a result, when we process each line, we also analyze the font name and size for each character and create a dictionary where we store the number of occurrences for each font. Using this dictionary, we identify the most predominant font in the line and considered the whole line is written using that font.

After creating a list of fonts for the page, one for each line, we look to see the most common font on the whole page. This helps us differentiate between the chapters and the paragraphs. Algorithm 5 presents how paragraphs and chapters are differentiated using the font. After finding the most common font on the page, we look at each line and compare it to the predominant font. We consider part of a paragraph anything that has a smaller or equal size. We also look at smaller size because there might be footnotes that weren't correctly identified. They are usually written in a smaller font and, in case there weren't detected in the previous steps we want them to be integrated into the whole paragraph. If we previously identified a table of contents, we consider a chapter title only lines that are in the chapter titles list. If this is not the case, the line is marked as a chapter title if it has the largest font on the page.

Algorithm 5: Identify chapters and paragraphs using the most common font

```
1 $mostCommonFont \leftarrow$ most common font on page;
  foreach line in page do
2     if line.fontSize $\leq$ $mostCommonFont$ then
3         append to paragraphs
4     else if line is in chaptersTitleList then
5         create new chapter
6     else if chaptersTitleList is empty and line.fontSize is largest on page then
7         create new chapter
8     else
9         append to paragraphs
```

After identifying a new chapter, anything that follows until another chapter title is identified is considered part of the paragraphs of the newly created chapter. We have chosen to look at the predominant font on a page rather than the whole book as some chapters such as the preface of a book or different articles in the same magazine might use a different font. However, we have noticed that, in general, a chapter uses the same font for each of their paragraphs. We have chosen the approach to look at the font size to have an alternative to look for chapter titles. If we have taken into consideration only the chapter titles parsed from the table of contents, if we were unable to identify it, we would have had a book that consisted of only paragraphs. While the predominant font method needs improvement, it offers an alternative to identifying the chapter structure of the book.

The identified footnotes are integrated into the line where they are referenced. When processing a line, we look at the footnotes identified on the page and see if any of them has their number or symbol referenced in the line. If that is the case, we replace the symbol from the line with the footnote content between brackets. However, adding the footnote to the line is not enough because when we normally extract text from the page the footnote is still extracted. This can lead to having footnotes appearing twice in the text. As a result, for each line, we check if it's part of a footnote. If that is the case, we ignore it and don't append it to the current paragraph.

Integrated footnotes aren't the only lines that are ignored. Usually, a book has the page number on a separate line. Also, depending on the template of the original books, we can have headings that contain the book title or the current chapter title. Since they can be followed or preceded by a number, we created a regular expression to find if the line that is currently processed matched the book or a chapter title accompanied by zero or more digits. The book title is taken from the form the user completes when uploading a document. The chapter title is the one we identified from the chapter list or by using the most common font method. All these lines are not taken into consideration in forming the paragraph as they don't offer any useful information.

Integrating tables involves removing the lines which are extracted but in an unformatted way and adding the tag “Insert Table <NUMBER>” where <NUMBER> is a counter that keeps track of the number of table tags added to the current chapter. Algorithm 6 presents how to replace the table lines with the corresponding tag.

```
Algorithm 6: Replace table lines with corresponding tag
  input: line - line currently processed
1 tableLines ← lines from area delimited by table outline;
  if line is in tableLines then
2     if line is first in tableLines then
3         create tag;
          append tag to paragraph;
          add table to chapter JSON;
4     else
5         ignore line
6 else
7     append line to paragraph or chapter title
```

We first start by identifying what lines are part of the tables identified in the page we process currently. For each identified table, we extract the text from the area determined by the rectangle that forms the outline of the table. For each table, we extract the text as lines since the processing it’s done at the line level. After the extraction process, we have a list of lines for each identified table. When a processing line from text, we also check to see if it’s part of any of the lists for table lines. If this is not the case, then the analysis process continues as described above. However, if it is a table line, we look at its position in the table lines list. If it’s the first one, it means we only start to process lines of tables and we need to append the tag to the current paragraph. Otherwise, the table already has a tag in text and the line can be ignored.

Images are also referenced in the text by adding the “Insert Image <IMAGE NAME>” tag to the place they were extracted from. <IMAGE NAME> refers to the filename the images are saved as locally. During the image extraction process, we saved the coordinates of the images so we can create a rectangle using the upper limit of the page and the upper limit of the extracted image. This rectangle is used to mark an area from which we extract lines of text. For each image, we have a list of lines from the start of the page to the place the image is positioned. We are interested in the line right above the image so from each list, we keep only the last line. When processing the lines on a page, when the line being processed matches any of the lines above the extracted image, it means we found the point where the image needs to be so, at the end of this line, we append the corresponding image tag.

After processing the extracted data, we have a list of chapters. A chapter is an object which contains three important fields: chapter title, paragraphs, and tables. In the paragraph fields, we have the extracted text with the footnotes integrated and with tags for every table and image detected in that chapter. A chapter is transformed in a JSON object and, along with the other chapters, we have a JSON array which is sent to the user interface to be displayed in TinyMCE.

3.3.2 Spellchecker

The spellchecking step is based on a client-server application, with the client being the user interface where the document is uploaded and displayed after the layout extraction process. The spellchecker is hosted on a server where requests can be sent in a specific JSON format.

Figure 12 shows the main steps followed in the spellchecking process. The server processes the request, extracts the text and analyzes it in two steps: identifies potential mistakes and creates a list of suggestions for each misspelled word. The list of suggestions for each identified error is sent to the user interface where it is underlined in the original text. For each identified mistake, the user has the possibility to see the suggestions list, correct it manually or ignore it.

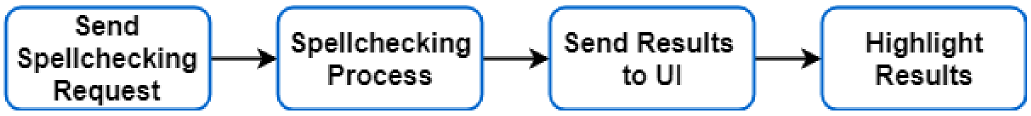
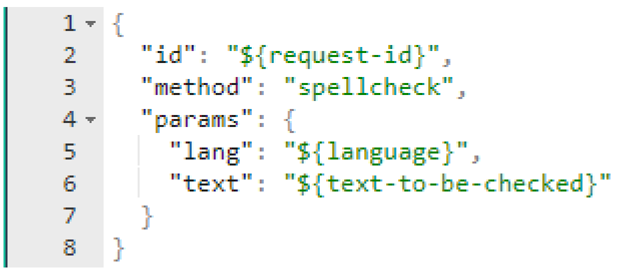


Figure 12. Overview of steps in the spellchecking process.

The spellchecking application is a Java servlet application created starting from the HttpServlet class. We have chosen to extend this interface as it offers methods for HTTP specific requests. This server is deployed as a WAR on a server running Apache Tomcat 9 and can be accessed on port 8081 from localhost.

This server waits for requests using the POST method with the body encoded in JSON Format. The format of the request can be seen in *Figure 13*. The id is an identifier established by the client, with the response using the same id to associate the context between these two objects. The method field specifies what kind of check to be performed. At this point, the only supported value is "spellcheck" with the possibility to implement in the future different checks or different customization. The params field includes the language and the text. The lang field represents a code to the language the text is written in (e.g., "ro", "en-us", "en-gb"). The text is a string that is analyzed for spelling errors.



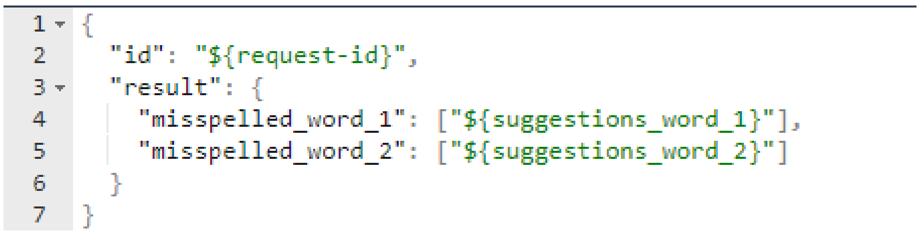
```
{
  "id": "${request-id}",
  "method": "spellcheck",
  "params": {
    "lang": "${language}",
    "text": "${text-to-be-checked}"
  }
}
```

Figure 13. Request body for spellchecker server.

When sending a request, the server receives it and analyses in the DoPost() method. Here we extract the text and pass it to the spellchecker which is based on JLanguageTool. The spellchecker is initialized in two steps: creating an instance for the language used for spellchecking and instantiation of a new LanguageTool using the language instance. In order to improve the performance, we don't want to create a spellchecker every time a request is made. As a result, we have a map where we save the instantiated spellchecker for each language. Before starting the spellchecking process, we load the spellchecker from the cache or, if it doesn't exist, we create a new one and save it in the cache.

Spellchecking using JLanguageTool has two steps: identifying mistakes that match a set of rules specific to the language and finding the suggestions for every match. At this point, we extract a substring from the text from the start of the match to the end of it in order to use it in the response sent to the client.

The response is also encoded in a JSON format similar to the request as it can be seen from *Figure 14*. Every identified spelling error is sent as a field in the response with the value set to a list of strings, each element from the list being a potential correction for the found error.



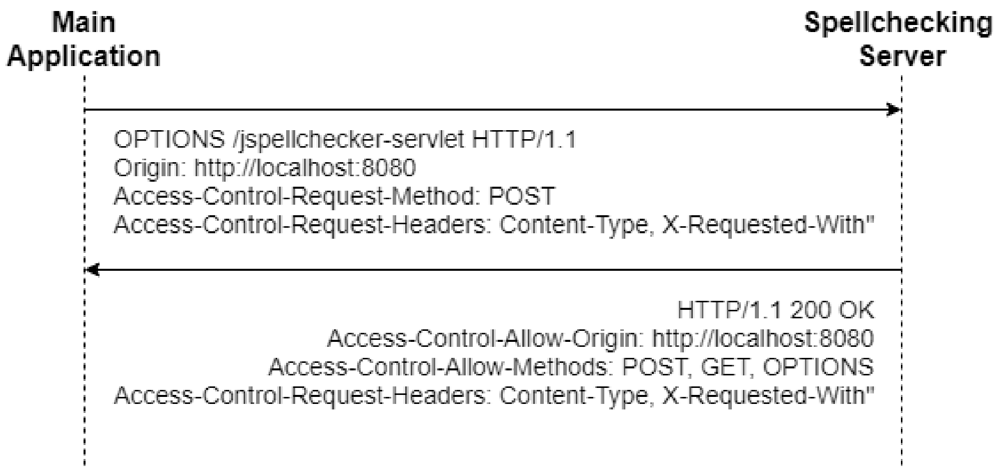
```
{
  "id": "${request-id}",
  "result": {
    "misspelled_word_1": ["${suggestions_word_1}"],
    "misspelled_word_2": ["${suggestions_word_2}"]
  }
}
```

Figure 14. Response body for spellchecker sever.

The detection of mistakes doesn't require a server and could have been done in the same step with the layout extraction. However, we have chosen this approach to offer the possibility to choose the correction, rather than automatically correct it. In order to correctly highlight the mistakes, we took advantage of the TinyMCE functionality for spellchecking method and make a call to our own server.

When initializing the TinyMCE text area, we override the spellcheck callback used when invoking the spellchecker. The new callback sends a JSON-RPC over HTTP POST request where the format is the same as *Figure 13*. As our application supports only Romanian spellchecker, the language field is always set to "ro". The text field is extracted from the TinyMCE contents in a raw format, removing all the HTML tags that are used to format paragraphs, chapter titles or tables. After receiving the response, TinyMCE automatically interprets the JSON and highlights each occurrence of the words from the result field.

Since the spellchecker server and the main application work on two different domains and the request is made from within a script, HTTP restricts cross-domain requests. Before sending the POST request, the browser sends an OPTIONS request to check if the domain is allowed to access the server. As a result, we need to do an extra step in order to accept the requests from the main application domain. Namely, we need to set the response headers for the OPTIONS to clearly specify that it allows requests that come from the origin with address localhost and port 8080. *Figure 15* shows the request the browser does automatically and the headers we need to add in the server as a response for this request. As we can see, the server explicitly needs to allow the main application domain so the request is accepted. The same headers need to be added to the POST response as well along the classic headers for cache control, encoding, and content type.



```
OPTIONS /jspellchecker-servlet HTTP/1.1
Origin: http://localhost:8080
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type, X-Requested-With"
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:8080
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Request-Headers: Content-Type, X-Requested-With"
```

Figure 15. Pre-flight request to check if CORS supported.

The main disadvantage of TinyMCE is that it doesn't take into account the position where an error was found. When analyzing the response, it takes any misspelled word and highlights all the occurrences of that word in the text. Due to this, contextual errors such as one word being written with a small letter instead of a capital letter and with no other mistakes, are highlighted for every word even if the rest were written correctly. As a compromise, we have chosen to identify only dictionary-based errors leaving the search for an alternative to TinyMCE to future work.

3.4 Information Search

When performing a query to search for information, a series of filters are applied to the whole phrase to transform them into tokens. *Figure 16* shows the four filters used in this project in both the analysis process during indexing the data into the database and the search.

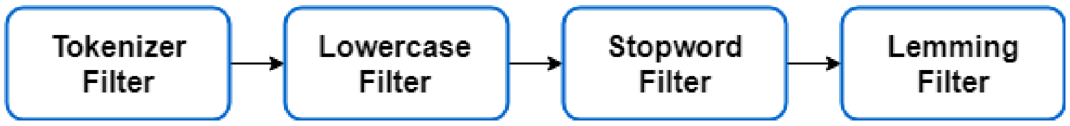


Figure 16. Filters applied in the analysis process.

The tokenizer filter splits the phrase into words called tokens, while the lowercase filter changes the tokens to be written only in small caps. The stop words filter removes words that are very common in a language such as conjunctions, prepositions or demonstrative nouns. This removal has the goal to reduce the index size and improve search performance. The final step has the goal to bring a word at its base form eliminating suffixes which represent possessive articles, plural forms or verbal forms.

Bringing a word to its root form by removing suffixes can be done in two ways: using stemming or lemming. The stemming process eliminates common suffixes or prefixes used in a certain language while lemming analyses the words at a morphological level. Lemming has some certain advantages since it creates the base word taking into consideration its part of speech. The stemming can remove prefixes or suffixes that are important in the searching progress. This can result in more results with some that may not fit with the original meaning of the query.

The previously mentioned filters need to be applied both at the index time, when saving a book in the database, and at query time. In order to do this, we create an analyzer in the index settings called "default" which will be referred in both situations. Elasticsearch offers implementations for Romanian of all the four filters we previously mentioned. However, due to the fact they use a stemmer that removes important parts of the word and a relatively small list of stop words, we decide to override some of the default filters.

Creating our own synonyms filter involves two steps: creating a synonyms file in the required format and defining and adding the filter to the analyzer. To create the synonyms file, we started from a SQL database dump created from the Romanian Online Dictionary. This database contains a table with a list of over 48 thousand words and their synonyms. After extracting the table from the database into a CSV format, we created a parser in order to remove the examples phrases for different words, abbreviations used for parts of speech or to for country regions where a certain form is used. After the parsing process, we have a file with the format as shown in *Figure 17* where one line contains words that are considered synonyms. This file was added to the synonym analysis path in Elasticsearch.

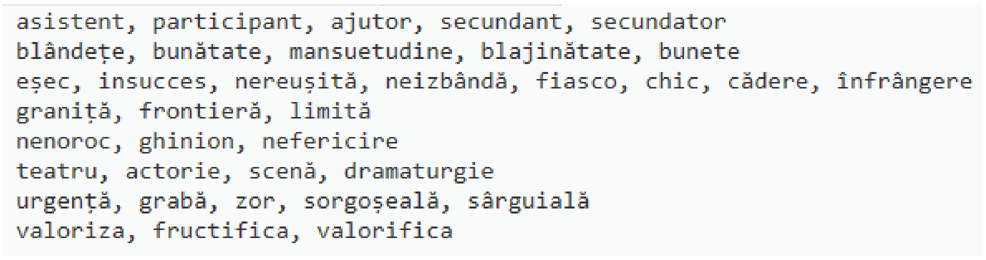

```
asistent, participant, ajutor, secundant, secundator
blândețe, bunătate, mansuetudine, blajinătate, bunete
eșec, insucces, nereușită, neizbândă, fiasco, chic, cădere, înfrângere
graniță, frontieră, limită
nenoroc, ghinion, nefericire
teatru, actorie, scenă, dramaturgie
urgență, grabă, zor, sorgoșeală, sârguială
valoriza, fructifica, valorifica
```

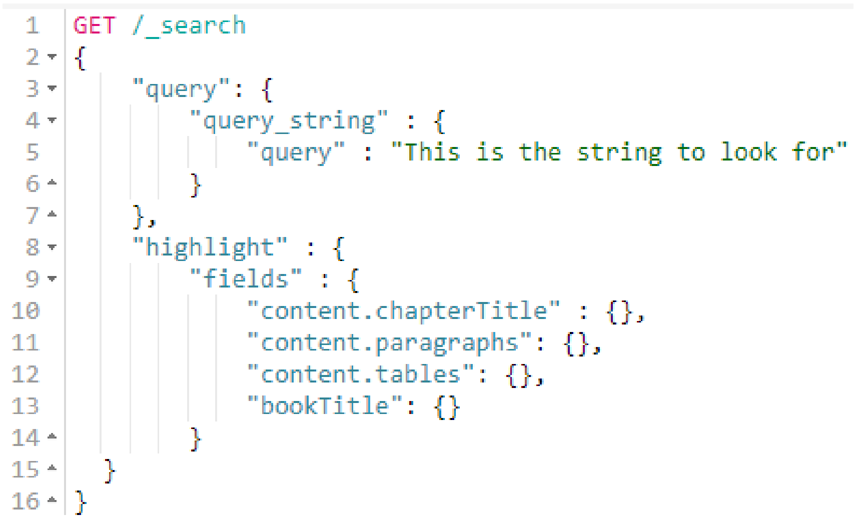
Figure 17. Part of the synonyms file used in Elasticsearch.

We choose to override the default stemming filter with one based on lemming. The default Elasticsearch stemming filter for Romanian is Snowball. However, after checking the way it works, we realized that for some words it can remove essential parts of the word. Table 3 shows a few examples of Romanian words and the root extracted after the stemming and lemming process. It can be seen that for some words they are removed so many characters that for different parts of speech it results in the same root, which might lead to results that don't fit the meaning of the initial query. Our lemming filter is based on a file with over 300 thousand words and their root form, including different parts of speech such as verbs, nouns, and adjectives, all with various articulated forms.

Table 3. Extracting word root using stemming and lemming.

Word	Root word	Stemming Root	Lemming Root
gândul	gând	gând	gând
gânditoare	gânditor	gândit	gânditor
gândirea	gândire	gând	gândire
gândiseră	gândi	gând	gândi
ocolită	ocolit	ocol	ocolit
ocolului	ocol	ocol	ocol
ocolește	ocoli	ocol	ocoli

The search request is created and sent using the Elasticsearch Java API. We create a query_string query that parses the input and splits the text around different operators. We used the default operator OR to look for any of the tokens from the query. By default, we look in all the indexes from the book including tables, chapter paragraphs, chapter titles and book title. In order to visualize the results after the search, we use a highlighter for all the previously mentioned fields. *Figure 18* shows the CURL equivalent to the request sent from Java. The results are displayed at the user interface in JSON format containing the book title and a list of fragments with the results highlighted using the <em> tag.

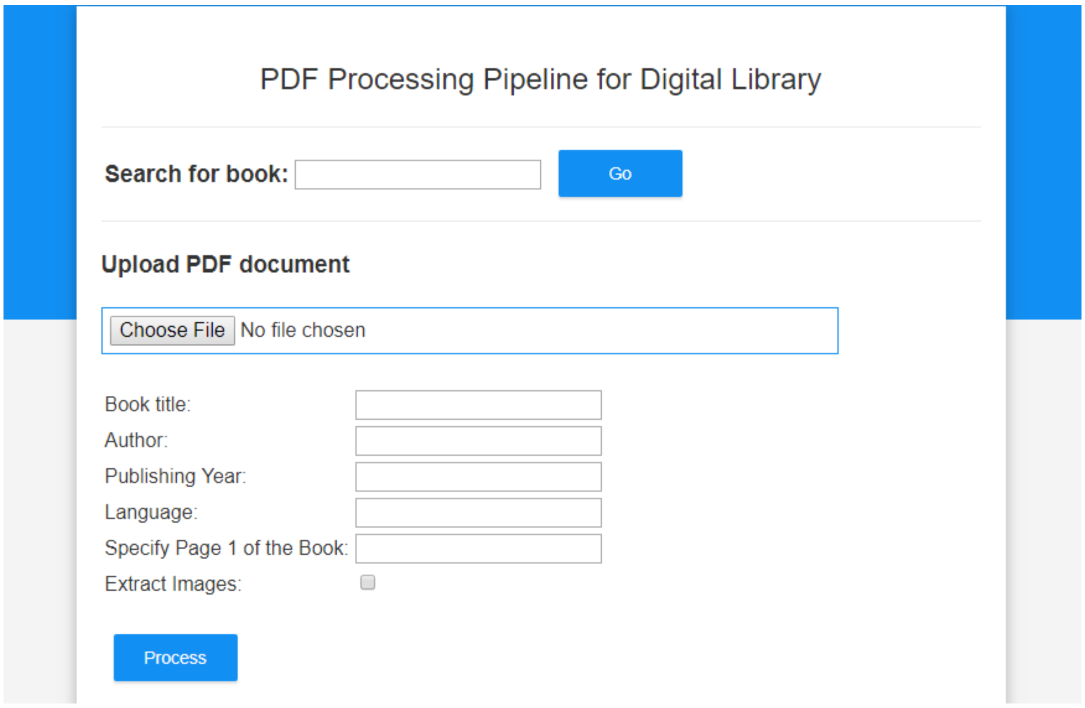


```
GET /_search
{
    "query": {
        "query_string" : {
            "query" : "This is the string to look for"
        }
    },
    "highlight" : {
        "fields" : {
            "content.chapterTitle" : {},
            "content.paragraphs": {},
            "content.tables": {},
            "bookTitle": {}
        }
  }
}
```

Figure 18. Elasticsearch query example.

3.5 User Interface

The main application is a Spring application running on the port 8080 on the localhost. On this interface, the user has the possibility to upload, edit and store a document and search within all the saved books stored in Elasticsearch.


PDF Processing Pipeline for Digital Library

Search for book: Go

Upload PDF document

Choose File No file chosen

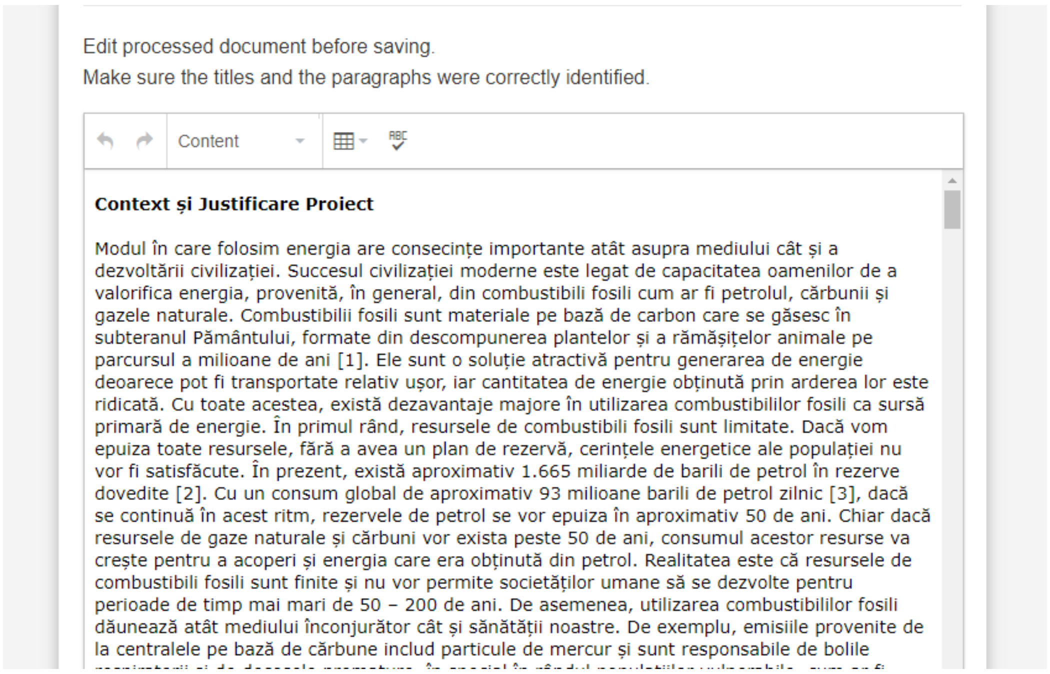
Book title:
Author:
Publishing Year:
Language:
Specify Page 1 of the Book:
Extract Images:

Process

Figure 19. User interface for the uploading process.

Figure 19 shows the interface for uploading a file in order to be processed. After choosing a file and before starting the processing stage, the book title, author and the first page of the book fields need to be completed. The first two are used when saving the processed book in Elasticsearch, while the last one is used in the layout extraction process. This last field is used to ignore first pages of the book that don't contain any useful information such as the ISBN number of the book or the publishing date. The other 3 fields are optional, with the publishing year and language being added as fields when indexing the book in Elasticsearch. If these two fields are left uncompleted and the document is indexed in Elasticsearch, we use the default values "2019" for publishing year and "ro" for the language. As it can be a costing process and decrease the performance, a user also has the possibility to extract or not the images from the book.

After processing the book, the extracted content is shown in the TinyMCE interface area. The identified paragraphs are written normally while the chapter titles are written in bold. At this step, the user can modify chapter titles or paragraph content, correct any misidentification between paragraphs and headers and add or modify tables. At this point, the user can also invoke the spellchecker to identify potential misspelling with all the detected errors underlined in red. *Figure 20* shows the interface where a book has been processed and displayed in TinyMCE.



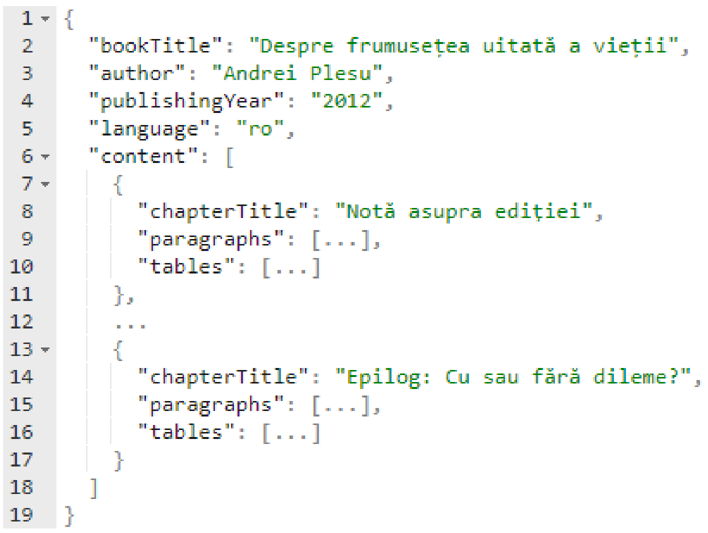
Edit processed document before saving.
Make sure the titles and the paragraphs were correctly identified.

Context și Justificare Proiect

Modul în care folosim energia are consecințe importante atât asupra mediului cât și a dezvoltării civilizației. Succesul civilizației moderne este legat de capacitatea oamenilor de a valorifica energia, provenită, în general, din combustibili fosili cum ar fi petrolul, cărbunii și gazele naturale. Combustibilii fosili sunt materiale pe bază de carbon care se găsesc în subteranul Pământului, formate din descompunerea plantelor și a rămășițelor animale pe parcursul a milioane de ani [1]. Ele sunt o soluție atractivă pentru generarea de energie deoarece pot fi transportate relativ ușor, iar cantitatea de energie obținută prin arderea lor este ridicată. Cu toate acestea, există dezavantaje majore în utilizarea combustibililor fosili ca sursă primară de energie. În primul rând, resursele de combustibili fosili sunt limitate. Dacă vom epuiza toate resursele, fără a avea un plan de rezervă, cerințele energetice ale populației nu vor fi satisfăcute. În prezent, există aproximativ 1.665 miliarde de barili de petrol în rezerve dovedite [2]. Cu un consum global de aproximativ 93 milioane barili de petrol zilnic [3], dacă se continuă în acest ritm, rezervele de petrol se vor epuiza în aproximativ 50 de ani. Chiar dacă resursele de gaze naturale și cărbuni vor exista peste 50 de ani, consumul acestor resurse va crește pentru a acoperi și energia care era obținută din petrol. Realitatea este că resursele de combustibili fosili sunt finite și nu vor permite societăților umane să se dezvolte pentru perioade de timp mai mari de 50 – 200 de ani. De asemenea, utilizarea combustibililor fosili dăunează atât mediului înconjurător cât și sănătății noastre. De exemplu, emisiile provenite de la centralele pe bază de cărbune includ particule de mercur și sunt responsabile de bolile

Figure 20. User interface for the editing process.

The processed information is saved in Elasticsearch by sending it as a JSON to the cluster. An example of the JSON format can be seen in *Figure 21*. All the processed books are saved in an Elasticsearch index called */books*. To send the information, we use a transport client which forwards the request to a node in the cluster. The request is made to the localhost on the port 9300, the configured port for Java clients.



```
{
  "bookTitle": "Despre frumusețea uitată a vieții",
  "author": "Andrei Plesu",
  "publishingYear": "2012",
  "language": "ro",
  "content": [
    {
      "chapterTitle": "Notă asupra ediției",
      "paragraphs": [...],
      "tables": [...]
    },
    ...
    {
      "chapterTitle": "Epilog: Cu sau fără dileme?",
      "paragraphs": [...],
      "tables": [...]
    }
  ]
}
```

Figure 21. Processed book as JSON.

The search can be performed from the user interface as well, by introducing the search query in the field from the top of the page. The connection is established using a transport client, similarly to the saving request. In the end, the search results are displayed to the user in a JSON format containing the book title and fragments that matched the search.

4 RESULTS AND DISCUSSION

Starting from our corpus, we have chosen different datasets in order to test each of the steps in the layout extraction pipeline. Below, we will present how accurate are our algorithms in identifying tables of contents, footnotes, tables, and images. We will also discuss how the spellchecker and semantic search work on different inputs.

For the table of contents identification, we used a set of 50 books from which 34 have a table of contents. Table 4 shows how well our algorithm identifies if a book has a table of contents or not.

Table 4. Confusion matrix for table of contents identification algorithms.

	Predicted Class	
Actual Class	**Book has table of contents**	**Book doesn't have table of contents**
Book has table of contents	31	3
Book doesn't have table of contents	9	7

The false negatives are caused by two main reasons. For two of the books, the table of contents wasn't placed in the first or last ten pages of the book. The table of contents is placed at the end of the book, followed by the book appendix which has more than 20 pages. The other reason is the quality of the original document. The quality of the PDF is low, with words and characters being extracted on different lines even if they should be considered together. This means we are unable to identify any of the keywords.

Our algorithm identifies a table of content even if it doesn't exist in the documents that don't necessarily respect the structure of an usual book. These documents are articles from literary magazines or offer bibliographies for other books, resulting in multiple pages that end with numbers. Additionally, the use of one of the keywords but with a different meaning makes our algorithm to identify a table of content even if it doesn't exist.

For the 31 books correctly classified, we have tables of contents with over 887 entries. In Table 5 we present how correctly our extraction algorithm identifies the name of the chapters.

Table 5. Classification of the entries identified.

Type of Entry	Number of Entries
Entries identified correctly and completely	782
Entries not completely identified	46
Entries not identified	59
Entries wrongly identified	64

Entries identified correctly and completely are the entries that are the ones that were extracted exactly as they appear in the book. Entries that are not completely identified are entries that appear on multiple rows or pages. This case can also happen due to the low quality of the book where words that should be extracted on the same line are considered as being on multiple rows. The main reason behind not identifying an entry is the quality of the book. Most of the documents from which we were unable to extract the lines had handwritten notes close to the page number which at the text extraction process were seen as other characters. This means our regular expression to look for lines that end in numbers didn't match that certain entry. Entries were identified wrongly especially in the books where following the table of contents, we had pages containing different references or other book information with certain lines matching our regular expression and not being filtered by our further processing.

For footnotes identification, we looked at 10 books which combined had 145 pages with footnotes. Created with different software, only for two books we were able to identify horizontal lines. The other 8 had the lines as part of an image and we had to rely on the paragraph algorithm in order to identify the footnotes. Table 6 presents how our two algorithms performed in identifying footnotes.

Table 6 . Number of pages with footnotes identified.

Identification Method	Pages Correctly Identified	False positives	Total pages
Line recognition	45	0	60
Paragraph method	52	3	85

For the line recognition algorithm, in the two books, there were 60 pages that had footnotes at the bottom of the page. This algorithm identified the lines on all 60 pages and was able to extract the text below the line. The 15 pages that weren't identified correctly were from the same book and the cause was the footnote identifier. If written correctly, it would have been extracted as 1, however, it was written using small font size, that the character was extracted as an apostrophe which isn't considered a valid start of the footnote.

For the algorithm using the paragraph method, we weren't able to identify pages with footnotes that had the paragraph above ending in a punctuation mark. Another case where footnotes weren't correctly identified even if the paragraph didn't end in a punctuation mark was the documents with lower quality. Due to age, on the scanned image there were different marks or handwritten notes, which in the extraction process were transformed in different characters including a punctuation mark. The exact opposite situation, where the low quality led to characters being unclear or completely erased, is the reason why we also have false positives.

Regarding the tables, our corpus of 50 books has around 20 tables. However, using our algorithm we are able to extract only one table. Of the 20 tables found in the corpus, one doesn't have the required template, being separated using lines only between rows and not columns as well. The other tables we are unable to extract, while they might respect the necessary template, the lines are part of images, which Tabula is unable to extract. We have created our own file with 20 tables that can be extracted in order to see how our algorithm works. We have tables with multiple columns, with rows with single or multiple lines of text, merged rows and columns. We use different fonts, different sizes and different text alignments within the cell. Our algorithm identified and extracted all the data with some places needing improvement. For example, when multiple columns on one row are merged, the result splits the column and places the text in the first column. The heuristics used to find spaces between words don't work when within a cell we have spaces of different dimensions. Consequently, the result is a table where words separated by spaces larger than the predominant space size are merged into the same word.

We extracted 975 images from a data set formed of 30 books. While our algorithm succeeds in extracting all images from the documents, we also have a lot of files that aren't necessarily part of the book content. This may include graphic elements such as lines, page edges from the scanning process and elements specific to the document owner. Table 7 shows a more detailed presentation of all the extracted images and their number of occurrences. For one book, the algorithm extracted each figure in pieces resulting in seven different images for each extracted figure. However, after applying the merging algorithm, all the images were brought to their initial form. Furthermore, the filtering step was able to remove over 5000 images that appear due to the software that was used in the creation process. These images represented the whole page with all the text and other graphic elements and didn't provide any additional information than the extraction steps. Also, please note that we haven't taken into consideration the digital added watermark. 22 out of the data set have a watermark on each page resulting in approximate 5600 extracted images only for the watermark.

Table 7. Types of extracted images and their number of occurrences.

Image Content Type	Number of occurrences
Content Figure	356
Scanning Traces (e.g., page edges)	256
Graphic Elements (e.g., lines)	159
Font Enhancements (e.g., characters in bold)	140
Owner-specific elements (e.g., library stamp)	64

We ran the spell checker service over a few of the books from our corpus and here are the most interesting noticed observations. First, if a word is split into two or more parts, which is quite frequent in our books due to the OCR process, the spellchecker identifies each part as a mistake and offers suggestions for each part without the possibility to consider them part of the same word. Second, while it is seen as a mistake, if multiples words are united there are cases when there are no suggestions. If more than two words are united, there are no suggestions while, usually, for two merged words the right suggestion is offered. Additionally, some technical or newer terms borrowed from different languages and unusual proper nouns are not always identified. For examples, Romanian words such as “Aureliu”, “sustenabilitate” and “terawatt” are identified as mistakes although they are part of the Romanian dictionary. Also, for words with a length between 5 and 10 characters, the spell checker usually offers only one suggestion which is also correct. However, for smaller words, like those formed with 3 letters, the number of suggestions can be as high as ten. This was noticed especially with words that are abbreviated. Regarding abbreviated words, certain abbreviations such as “ing.”, “dir.” are identified as mistakes, while more popular ones such as “dr.” and “cuv.” are considered correct.

Regarding the semantic search, we searched for different queries on a book that has all the chapters identified and marked correctly and on which we also applied the spellchecker and corrected all the identified mistakes with their correct form. We performed different queries formed with one or multiple words. For queries with one word, we searched for verbs, nouns, and adjectives. For verbs, we searched in three forms: infinitive form (e.g., “a gândi”), present form for the singular first person (e.g., “întreb”) and past form for the plural second person (e.g., “avuseră”, “au vrut”). For nouns and adjectives, we used singular, plural and articulated forms (e.g., “frumoasă”, “vieții”, “politicului”). Our search engine is able to identify multiple forms of the word no matter what form the initial word has. Regarding synonyms, the search engine looks and finds similar words to the ones in the query if it exists in our synonyms file. Due to our synonyms file structure, some words are treated as having the same meaning even if that is not the case. We started from an SQL dump of the Romanian Dictionary which offers a word followed by a list of synonyms for it. However, after the parsing process, we have a list of words separated by commas, where all the words in one line are considered to have the same meaning by Elasticsearch. This leads to words being treated as synonyms when they actually have the same meaning with only the first word from the line. This is true especially for words that have multiple meanings depending on the context. For queries using multiple words, our system shows results for the whole searched expression and for individual words. Even in the results containing the whole expression, the search engine considers the derived forms of the word. As a result, even if only one word has a different form and the other has the same form as in the query, it still is identified and returned as a result.

5 CONCLUSIONS AND FUTURE WORK

The project presented in this thesis has the goal to provide a storage and search solution for digital documents by saving files in a structured format independent of the original layout of the document, by correcting the potential mistakes that occurred in the extraction process and by providing a search engine that doesn't search only for exact matches but at its meaning as well.

We offer a solution that starts by identifying and extracting the table of contents entries on which the final structure is based on. We identify and extract footnotes on each page using two methods: by extracting text below the last horizontal lines on a page and by looking at paragraphs that aren't completed on the same page and still followed by lines of text. The next step is to identify and extract tables and save them in a matrix format saved as a JSON object. We extract images, analyze them to see if we should merge them together in case they were extracted in pieces and filter them in order to reduce the number of objects that aren't parts of the actual text. In the end, we analyze the text, combine all the extracted elements to insert footnotes, table and images tags at the place they were extracted from and create a JSON array with an object for each chapter where we save information such as title, paragraphs and extracted tables.

For the automatic text correction, we developed a server where we can send a JSON request with the text in which we want to identify spelling errors. The server looks at the text and creates a list of all the potential misspellings and suggestions for each identified error. This list is sent as a response to the initial request and every occurrence of each entry in the list is highlighted at the user interface where correction can be chosen based on the offered suggestions.

For easy information retrieval, we offer a semantic search engine which looks for the query in the saved books from the Elasticsearch database. We apply an analyzer at indexing and at searching time by splitting the words in tokens, transform each token in small caps, remove any stop words and bring the word to its roots form. The search process also looks in a list of synonyms for each token to find similar entries to the one being searched.

Our algorithms were tested with books that were created with different software and that had different layouts. We were able to identify 782 of the table of contents entries out of a total of 887 from a data set of 50 books, while for footnote extraction we were able to correctly extract 97 out of the 145 pages found across 10 books. We were able to extract all images from our data set and, if it was the case, to merge them when they have been extracted separately. Due to the creation software of our data set, we were able to correctly detect and extract data for only one table. The spellchecking server is able to identify and offer suggestions for a wide range of words with room for improvement in neologisms and abbreviations. The semantic search engine offers multiple results for our queries, taking into consideration various forms and synonyms of each word from the searched expression.

For future work, there are four areas that could be improved. Firstly, there is room for improvement in identifying chapters titles when we are unable to extract the table of contents from the book. This can be done by looking at more information about the text not only at the font size. We could create different heuristics that look at the position of a line of text on the page or at the fonts used across the whole book and identify the ones that are used only on a few lines every couple of pages. Secondly, we would like to improve our tables detection algorithm either by looking at a way to detect lines on images and apply the existing extraction algorithm or by finding a way to filter more accurately the tables extracted using an algorithm that looks at the spaces between columns. Additionally, we would like to improve the current spellchecking process to identify words that are split and offer the suggestion to merge them or to detect more than spelling errors such as grammar or punctuation errors. The final area of improvement is the semantic search where we would like to create a system that searches for more than synonyms and also looks at the intent of the query.

REFERENCES

1. *Document Management – Portable Document Format – Part 1: PDF 1.7, First Edition*. 2008; Available from: https://www.adobe.com/devnet/pdf/pdf_reference.html.
2. Evans, T.N. and R.H. Moore, *The use of PDF/A in digital archives: a case study from archaeology.* International Journal of Digital Curation, 2014. **9**(2): p. 123-138.
3. Oettler, A., *PDF/A in a Nutshell: PDF for long-term archiving*. 2014: Satzweiss. com.
4. Ashcroft, L., *Ebooks in libraries: an overview of the current situation.* Library Management, 2011. **32**(6/7): p. 398-407.
5. Dehaene, S., *Reading in the brain: The new science of how we read*. 2009: Penguin.
6. Balog, K., *Entity-Oriented Search*. The Information Retrieval Series. 2018: Springer.
7. Bast, H., B. Buchhold, and E. Haussmann, *Semantic Search on Text and Knowledge Bases*. Vol. 10. 2016. 119-271.
8. Kay, A., *Tesseract: an open-source optical character recognition engine.* Linux J., 2007. **2007**(159): p. 2.
9. Rice, S.V., F.R. Jenkins, and T.A. Nartker, *The fifth annual test of OCR accuracy*. 1996: Information Science Research Institute.
10. Holley, R., *How good can it get? Analysing and improving OCR accuracy in large scale historic newspaper digitisation programs*. Vol. 15. 2009.
11. Pirker, J. and G. Wurzinger, *Optical Character Recognition of Old Fonts–A Case Study.* The IPSI BgD Transactions on Advanced Research, 2016: p. 10.
12. Handley, J., *Improving OCR accuracy through combination: a survey*. 1998. 4330-4333 vol.5.
13. Lopresti, D. and J. Zhou, *Using Consensus Sequence Voting to Correct OCR Errors*. Vol. 67. 1997. 39-47.
14. Pavlidis, T. and J. Zhou, *Page segmentation and classification.* CVGIP: Graphical Models and Image Processing, 1992. **54**(6): p. 484-496.
15. Gorman, L.O., *The document spectrum for page layout analysis.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1993. **15**(11): p. 1162-1173.
16. Pardede, P., *Scientific Articles Structure*. 2012.
17. Kukich, K., *Techniques for automatically correcting words in text.* ACM Comput. Surv., 1992. **24**(4): p. 377-439.
18. Riseman, E.M. and A.R. Hanson, *A contextual postprocessing system for error correction using binary n-grams.* IEEE Transactions on Computers, 1974(5): p. 480-493.
19. Beaufort, R. and C. Mancas-Thillou. *A weighted finite-state framework for correcting errors in natural scene OCR*. in *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. 2007. IEEE.
20. Magdy, W. and K. Darwish. *Arabic OCR error correction using character segment correction, language modeling, and shallow morphology*. in *Proceedings of the 2006 conference on empirical methods in natural language processing*. 2006. Association for Computational Linguistics.
21. Siklósi, B., A. Novák, and G. Prószéky, *Context-aware correction of spelling errors in Hungarian medical documents.* Computer Speech & Language, 2016. **35**: p. 219-233.
22. Miłkowski, M., *Developing an open-source, rule-based proofreading tool*. Vol. 40. 2010. 543-566.
23. Li, H. and J. Xu, *Semantic Matching in Search.* Found. Trends Inf. Retr., 2014. **7**(5): p. 343-469.

24. Sarawagi, S., *Information extraction.* Foundations and Trends® in Databases, 2008. **1**(3): p. 261-377.
25. Ramakrishnan, C., et al., *Layout-aware text extraction from full-text PDF of scientific articles*. Vol. 7. 2012. 7.
26. Jiang, D. and X. Yang, *Converting PDF to HTML approach based on text detection*, in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. 2009, ACM: Seoul, Korea. p. 982-985.
27. Anjewierden, A. *AIDAS: incremental logical structure discovery in PDF documents*. in *Proceedings of Sixth International Conference on Document Analysis and Recognition*. 2001.
28. Wu, J., et al., *PDFMEF: A Multi-Entity Knowledge Extraction Framework for Scholarly Documents and Semantic Search*. 2015. 1-8.
29. Gao, L., et al., *Structure extraction from PDF-based book documents*, in *Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries*. 2011, ACM: Ottawa, Ontario, Canada. p. 11-20.
30. Tkaczyk, D., et al., *CERMINE: automatic extraction of structured metadata from scientific literature.* International Journal on Document Analysis and Recognition (IJDAR), 2015. **18**(4): p. 317-335.
31. Wu, Z., P. Mitra, and C. Lee Giles, *Table of Contents Recognition and Extraction for Heterogeneous Book Documents*. 2013. 1205-1209.
32. Klampfl, S. and R. Kern, *An Unsupervised Machine Learning Approach to Body Text and Table of Contents Extraction from Digital Scientific Articles*. 2013.
33. Pirinen, T. and K. Lindén. *Finite-state spell-checking with weighted language and error models*. in *Proceedings of LREC 2010 Workshop on creation and use of basic lexical resources for less-resourced languages*. 2010.
34. Lindén, K., M. Silfverberg, and T. Pirinen, *HFST Tools for Morphology – An Efficient Open-Source Package for Construction of Morphological Analyzers*. Vol. 41. 2009. 28-47.
35. Hodge, V.J. and J. Austin, *A comparison of standard spell checking algorithms and a novel binary neural approach.* IEEE Transactions on Knowledge and Data Engineering, 2003. **15**(5): p. 1073-1081.
36. Richter, M., P. Straňák, and A. Rosen, *Korektor – A System for Contextual Spell-checking and Diacritics Completion*. 2012.
37. Priya, M., R. Kalpana, and T. Srisupriya, *Hybrid optimization algorithm using N gram based edit distance*. 2017. 0216-0221.
38. Schmaltz, A., et al., *Sentence-level grammatical error identification as sequence-to-sequence correction.* arXiv preprint arXiv:1604.04677, 2016.
39. Lee, W.-P. and T.-C. Tsai, *An interactive agent-based system for concept-based web search*. Vol. 24. 2003. 365-373.
40. Syeda-Mahmood, T., et al., *Semantic search of schema repositories*, in *Special interest tracks and posters of the 14th international conference on World Wide Web*. 2005, ACM: Chiba, Japan. p. 1126-1127.
41. Guha, R., R. McCool, and E. Miller, *Semantic search*, in *Proceedings of the 12th international conference on World Wide Web*. 2003, ACM: Budapest, Hungary. p. 700-709.

42. Ceglowski, M., A. Coburn, and J. Cuadrado, *Semantic search of unstructured data using contextual network graphs.* National Institute for Technology and Liberal Education, 2003. **10**.
43. Faessler, E. and U. Hahn. *Semedico: A Comprehensive Semantic Search Engine for the Life Sciences*. 2017. Vancouver, Canada: Association for Computational Linguistics.
44. *PDFBox Apache*. 2014 [cited 2019 15th June]; Available from: https://pdfbox.apache.org/index.html.
45. Manuel Aristarán, M.T., Jeremy B. Merrill, Jason Das, *Tabula*, in *https://tabula.technology/*. 2018.
46. LanguageTool. *Java Spell Checker*. 2013 17th May 2017 [cited 14th June 2019]; Available from: http://wiki.languagetool.org/java-spell-checker.
47. *Elasticsearch*. [cited 2019 14th June]; Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.5/index.html.
48. Zellmann, T. *Apache PDFBox named an Open Source Partner Organization of the PDF Association*. 2015 [cited 2019 15th June]; Available from: https://www.pdfa.org/apache%C2%99-pdfbox%C2%99-named-an-open-source-partner-organization-of-the-pdf-association/.
49. *PDFlib TET*. [cited 2019 15th June]; Available from: https://www.pdflib.com/products/tet/.
50. Tigas, M.A.a.M. *Introducing Tabula*. 2013 [cited 2019 15th June]; Available from: https://source.opennews.org/articles/introducing-tabula/.
51. Luong, T.Q. *TrapRange: a Method to Extract Table Content in PDF Files*. 28th April 2015 [cited 2019 15th June]; Available from: https://dzone.com/articles/traprange-method-extract-table.
52. Idzelis, M. *Jazzy - The Java Open Source Spell Checker*. [cited 2019 14th June]; Available from: http://jazzy.sourceforge.net/.
53. *Hunspell*. [cited 2019 14th June]; Available from: http://hunspell.github.io/.
54. *Comparison table of Basic Suggester and other popular spellchecking software*. 2007 [cited 2019 15th June]; Available from: http://www.softcorporation.com/products/spellcheck/comparison.html.
55. Milkowski, M. *Hunspell Support*. 2012 20 November 2017 [cited 15th June 2019].
56. *What Is Elasticsearch And How Can It Be Helpful For My Business?* [cited 2019 15th June]; Available from: https://www.marutitech.com/elasticsearch-can-helpful-business/.
57. Chang, E. *How to monitor Elasticsearch performance*. 2016; Available from: https://www.datadoghq.com/blog/monitor-elasticsearch-performance-metrics/#what-is-elasticsearch.
58. *Solr*. [cited 2019 16th June]; Available from: https://lucene.apache.org/solr/.
59. *Elasticsearch vs. Solr performance: round 2*. 2015 [cited 16th June 2019]; Available from: http://www.flax.co.uk/blog/2015/12/02/elasticsearch-vs-solr-performance-round-2/.
60. Kuć, R. *Top 15 Solr vs. Elasticsearch Differences*. 2015 11th June 2018; Available from: https://sematext.com/blog/solr-vs-elasticsearch-differences/.
61. *TinyMCE*. [cited 2019 16th June]; Available from: https://www.tiny.cloud/.
62. Nurminen, A., *Algorithmic extraction of data in tables in PDF documents.* 2013.
63. *How to Use GhostScript*. 2019 [cited 2019 17th June]; Available from: https://www.ghostscript.com/doc/current/Use.htm#Improving_performance.