Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

PDF document representation for automated analysis

Bc. Jakub Váca

Supervisor: Ing. Martin Rehák, Ph.D.
Field of study: Software Engineering
Subfield: Open Informatics
May 2020

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Váca Jakub**	Personal ID number:	**457119**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

PDF document representation for automated analysis

Master’s thesis title in Czech:

Reprezentace PDF dokumentů pro automatickou analýzu

Guidelines:

1. Study PDF format and existing options for its processing (PDFMiner).
2. Review serialization formats suitable for machine learning, for instance Google Protobuf, and propose a method that converts PDF into the chosen format.
3. Implement application in Python language that takes PDF document, extracts its features and stores them into chosen format.
4. Further implement several examples of PDF document modification detectors, such as word overlap detector.
5. Test feature extraction modification detection on appropriately chosen or created dataset.
6. Compare performance of PDF loading for future analysis (machine learning) from chosen format and using PDFMiner library.

During the analysis and implementation take the GDPR legislation into account and try to save the maximum amount of information allowed by GDPR regulations.

Bibliography / sources:

[1] Adobe System Incorporated, PDF Reference - Sixth edition, Version 1.7, November 2006

Name and workplace of master’s thesis supervisor:

Ing. Martin Rehák, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **11.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Martin Rehák, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank my supervisor Martin Rehák for his guidance and for providing me with an opportunity to work on such an interesting and evolving project at Resistant.AI.
I would also like to thank Martin Vejman for introducing me to the intrigues of PDF and for his suggestions for my thesis.
My thanks also go to my family and especially to my girlfriend Eva for all their support.

Declaration

I declare that presented work was developed independently and that I listed all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague, 22nd May 2020

Abstract

PDF documents are the most popular electronic form of information exchange between businesses and individuals. With its popularity in business usage, forgery of PDF documents is an ever-growing threat. In this thesis, we address the problem of preprocessing of PDF documents for automated analysis. We extract relevant features and store them in serialization format suitable for machine learning. We additionally implement several modification detectors covering various types of PDF modifications. We experimentally evaluate modification detectors on labelled and real-world data and show that our application reliably detects various types of document modifications. We also show that by storing the results of preprocessing, any subsequent training cycle can be sped up to 18 times for scanned dataset. All of the above is done while taking the GDPR legislation into account, so an application is ready for business use.

Keywords: PDF, information extraction, forgery, serialization, preprocessing, GDPR

Supervisor: Ing. Martin Rehák, Ph.D.

Abstrakt

PDF dokumenty jsou nejpopulárnější forma výměny informací mezi byznysy a jednotlivci. S vysokou popularitou PDF mezi firmami roste i problém padělání PDF dokumentů. V této diplomové práci se zabýváme problémem předzpracování PDF dokumentů pro automatickou analýzu. Z dokumentů extrahujeme relevantní informace, které následně ukládáme do formátu vhodném pro zpracování strojovým učením. Navíc implementujeme několik detektorů modifikací, které pokrývají různé typy PDF modifikací. Detektory modifikací otestujeme na uměle vytvořených i veřejně dostupných datech a ukážeme, že spolehlivě detekujeme různe typy modifikací dokumentu. Dále ukážeme, že uložením výsledku předzpracování dat můžeme zrychlit jakýkoliv následující trénovací cyklus až 18krát pro skenované dokumenty. Během celého vývoje zohledňujeme ukládání dat v souladu s legislativou GDPR, tak aby byla aplikace připravena pro použití v praxi.

Klíčová slova: PDF, extrahování vlastností, falšování, serializace, předzpracování, GDPR

Překlad názvu: Reprezentace PDF dokumentů pro automatickou analýzu

Contents

Figures

Tables

Chapter 1
Introduction

Electronic documents are becoming more and more common method of information interchange, with the de-facto standard being the Portable Document Format (PDF). There are billions of new PDF files created every day and the number of opened PDF files inside Adobe Acrobat tripled between the years 2016 and 2018.[6] The format is used to represent a vast variety of document types ranging from invoices, bank statements, presentations, ebooks and tickets to scanned ID cards. In addition, business documents, such as bank statements, are increasingly relied upon as proof of income or financial stability. With such popularity, PDF document forgery becomes a major problem. It is estimated that one in seven small or medium-sized enterprises has been victims of invoice fraud and that financial services are loosing more than £165 million to some type of invoice fraud every year.[7] Developing an effective PDF analysis system that would parse PDF information and determine if a document was manipulated or not is crucial to protect businesses from document forgery related fraud.

This thesis addresses a problem of PDF document fraud and focuses on transforming raw PDF file into anonymized representation suited for automated analysis. The main steps are PDF parsing, extraction of information and storing of extracted information in the anonymized state.

1.1 Thesis Outline

The thesis is organized as follows:

In Chapter 2, we introduce and explain the concept of PDF format.

In Chapter 3, we review several serialization frameworks, compare their performances and select the most suitable framework for our thesis. We then introduce the main points of GDPR and explain how are we going to comply with them. Lastly, we introduce a library used for PDF parsing and explain how PDF parsing is done.

In Chapter 4, we explain types of PDF fraud, then we define extracted features. We sort features into multiple categories and in detail explain why and how we extract those features.

In Chapter 5, we first give a high-level overview of our application, then we introduce how we represent a PDF page and explain the extraction and representation of multiple modification features in detail. In the end, we explain how the serialization and deserialization of extracted features are done.

In Chapter 6, we introduce project's most used libraries and structure.

In Chapter 7, we explain evaluation metrics and introduce used datasets. We then evaluate PDF modification detection and loading of a parsed PDF file from our representation. We give a detailed analysis of results and compare modification detection classifiers. We also show the advantages of storing results from preprocessing.

In Chapter 8, we summarize the results, give suggestions for future work and conclude this thesis.

1.2 Thesis Goals

Based the assignment of this thesis we set out 3 specific goals:

1. Implement application that takes a PDF file and converts it into an anonymized selected format with extracted information.
2. Implement several PDF document modification detectors.
3. Speed up the process of analyzing PDF files.

Chapter 2

Portable Document Format

The Portable Document Format (PDF) is a file format developed by Adobe in the 1990s to present documents, including text formatting and images, in a manner independent of application software, hardware, and operating systems.[8, Section 1] Based on the PostScript language every file encapsulates everything it needs for displaying it. PDF was standardized as ISO 32000 in 2008 and nowdays it is one of the most used file formats. It became the de facto standard for the electronic exchange of documents [8, Section 1] mainly because of its great portability. PDF is used from invoices, payslips, bank statements to presentations, ebooks, school textbooks or tickets, which makes PDF one of the most diverse file formats.

During its development Adobe released many versions of PDF starting from version 1.0 in 1993 and ending at version 1.7 in 2006. PDF version 2.0 was released by International Organization for Standards (ISO) in July 2017. In this thesis we will focus at the most widely spread and used version 1.7.

2.1 Overview

A document's page can contain any combination of text, graphics, and images. A page's appearance is described by a PDF *content stream*, which contains a sequence of graphic objects to be painted on the page. This appearance is fully specified; all layout and formatting decisions have already been made by the application generating the content stream.[8, Section 2]

A document can furthermore contain interactive elements such as annotations used for text notes, hypertext links, file attachments and higher-level information that can be used for interchange of content among applications or can contain logical structure information that allows the document to be searched or edited.

2.1.1 Page Description Language

PDF serves as a *page description language*, a language for describing the graphical appearance of pages with respect to an imaging model.
The PDF operators for setting the graphical state are similar to the coresponding operators in PostScript language. But unlike PostScript, PDF is not a full scale programming language. It trades reduced flexibility for improved efficiency.[8, Section 2.1.] The main differences are:

- PDF lacks control constructs such as variables and procedures.
- PDF enforces strictly defined file structure that allows applications to acces parts of document in arbitary order.
- PDF file contains additional information to ensure viewing fidelity.

2.1.2 Random Access

The order of object occurrence inside a PDF document has no semantic significance. Instead, applications should process a PDF document by following references from object to object, rather than by processing sequentially. This is important for interactive document viewing or for any application where are documents pages or other objects are accessed out of sequence.[8, Section 2.2.5]
To support random access to individual objects, every PDF file contains *cross-reference table* We talk more in detail about *cross-reference table* in subsection 2.3.3.

2.1.3 Incremental Updates

Applications may allow the user to modify the content. To avoid waiting for an entire file, which can be couple hundreds of pages long, PDF allows modifications to be appended to a file, leaving the original data intact. Incremental update allows to save modifications in the amount of time proportional to the size of modification rather than the size of the file.[8, Section 2.2.7]

2.1.4 Security

PDF possesses two security features that can be used without any dependancy on each other.

1. **Encryption** - A document can be encrypted so that only authorized users can access it. Users access can also be selectively restricted to allow only certain operations, such as viewing, content copying, printing or editing.
2. **Digital signature** - A document can be digitally signed to verify its authenticity. The signature may take many forms, from private/public

key signature to some form of biometric signature. Any subsequent changes to PDF will invalidate the signature.[8, Section 2.2.6]

2.2 Building Blocks

PDF supports 8 object types that are used to represent a file:

- **Boolean** - values `true` or `false`
- **Integer and real numbers** - e.g., 666, -28.6
- **Strings** - literal and hexadecimal, e.g, `(example)`, `<6578616d706c65>`
- **Names** - sequence of characters starting with a slash (/), e.g., `/Root`
- **Arrays** - one dimensional heteregeneous collection, e.g., `[/SomeName (John) 3.14 false]`
- **Dictionaries** - table containing key-value pairs, key must be a name object e.g., `«/Offset 42»`
- **Streams** - more details in 2.2.1
- **The null object** - `null`

2.2.1 Stream Objects

A *stream object* is a sequence of bytes, which a PDF application can read incrementally. Streams can be of an unlimited size so objects with large amounts of data, such as images and page descriptions, are represented as streams.[8, Section 3.2.7]

The stream begins with a dictionary followed by 0 or more bytes bracketed between keywords `stream` and `endstream`. Example:

```
<<\Size 9 \Type \Image ... >>
stream
a42c97648b687...
endstream
```

2.2.2 Indirect Objects

Objects may be labeled so that they can be referenced to by other objects. A labeled object is called an indirect object. This gives the object a unique *object identifier* by which other objects can refer to it.[8, Section 3.2.9] The object identifier consists of two parts:

- A positive integer *object number*,

- A non-negative integer *generation number*. In a newly created file all indirect objects have a generation number of 0.

The combination of object number and generation number uniquely identifies an indirect object.

The definition of an indirect object in the file consists of its object number and generation number followed by the value of the object surrounded between keywords **obj** and **endobj**. The object than can be referred to from elsewhere in the file by an *indirect reference* of the object number, the generation number and keyword **R**.[8, Section 3.2.9] Example of indirect reference to an object with object number 16 and generation number 0:

```
16 0 R
```

2.2.3 Object Streams

Object stream is a stream that contains a sequence of PDF objects. The purpose of object streams is to allow a greater number of PDF objects to be compressed, therefore reducing the size of PDF files. The objects in the stream are refered to as *compressed objects*.[8, Section 3.4.6] Object streams were added to PDF in version 1.5.

2.2.4 PDF Comments

Comments inside PDFs are preceded by % sign granted the % is outside of string or stream. The comment consists of all characters between the percent sign character and the end of the line including space and tab characters. PDF ignores comments and treats them as if they were single white characters.[8, Section 3.1.2] Comments have no semantics apart from PDF version and end of file marker described in 2.3.

2.3 File Structure

A Typical PDF file consists of four main elements, whose order can be seen in Figure 2.1.

- A one line, *header* which specifies the PDF version in which the file belongs.
- A *body* containing the objects that make up the document.
- A *cross-reference table* containing information about indirect objects in the file.
- A *trailer* giving the location of the cross-reference table and of certain special objects within the body, such as root element.

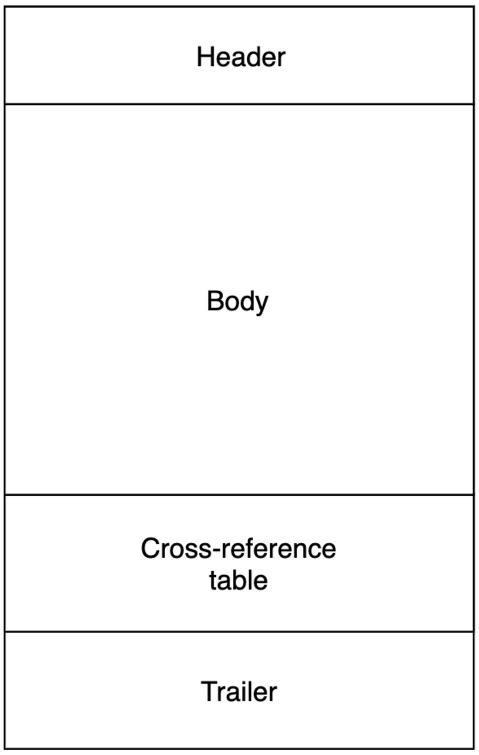


Figure 2.1: Structure of a PDF file

2.3.1 Header

The header is the first line of PDF file. Identifying the version of PDF specification. The header starts with `%PDF-` followed by a version number. At this moment, versions 1.0 to 1.7 and 2.0 are applicable. For a file belonging to version 1.7, the header would look like `%PDF-1.7`.

PDF format is backward compatible so any earlier version of PDF also conforms to version 1.7.

2.3.2 Body

The body consists of a sequence of indirect objects representing the contents of a document.[8, Section 3.4.2] Objects represent components such as fonts, pages and images. The body can also contain object streams, which contain a sequence of indirect objects. Example:

```
23 0 obj
% data of object 23
endobj
9 0 obj
% data of object 9
endojb
```

2.3.3 Cross-Reference Table

The *cross-reference table* contains information crucial for random access to indirect objects within the body so that the entire file doesn't need to be read to locate any particular object.[8, Section 3.4.3] The table contains one line entry for each indirect object, specifying its location inside PDF's body.

One *cross-refence table* can contain more than one *cross-reference sections*. Cross-reference section begins with keyword **xref**.
Example of cross-reference section with two subsections:

```
xref
0 2          % subsection at object 0 with 2 entries
0000000000 65535 f % first entry always free with generation number 65535
0000001565 00000 n % object number 1, offset 1565, generation 0, in use
4 1 % subsection at object 4 with 1 entry
0000002514 00002 n % object number 4, offset 2514, generation 2, in use
```

2.3.4 Trailer

The file trailer enables an application reading file to quickly locate the cross-reference table and other special objects. All applications should read PDF files from the end, where trailer is located.

The trailer starts with a keyword **trailer** followed by a trailer dictionary and a series of key-value pairs.
The last three lines of the file are in order keyword **startxref**, byte offset from the beginning of the file to the start of the **xref** keyword in the last cross-reference section. The last line of the file contains only the end of file marker `%%EOF`. All possible values inside trailer can be seen in table 2.1.

Key	Type	Required	Value
Size	integer	True	The total number of entries in the file's cross-reference table.
Root	dictionary	True	Indirect reference. The catalog dictionary.
Info	dictionary	False	Indirect reference. The document's info dictionary.
Encrypt	dictionary	False	Encryption dictionary.
ID	array	False	Recommended, an array of two byte strings constituting a file identifier.
Prev	integer	False	Present only if the file has more than one cross-reference section.

Table 2.1: Possible values inside trailer

Example of a file trailer:

```
trailer
<< /Size 22 /Root 2 0 R /Info 1 0 R
   /ID [ <77fa14e5e882571aeb5fd2f92d2ea001>
     <77fa12e5e882571aeb5fd2f92d2ea001>]
>>
startxref
18799
%%EOF
```

2.3.5 Linearized PDF

A linearized PDF file is an optional file organization whose primary goal is to achieve the following behavior:

- When a document is opened, display the first page as quickly as possible. The first to be viewed does not have to be the first page of document.
- When a user clicks on a link that leads to another page of the document, display it as quickly as possible.
- When page data is delivered through a slow connection, display the page incrementally as it arrives.
- Permit user interaction even before the entire page has been received and displayed.

Enhanced viewer applications can recognize that a PDF file has been linearized and can take advantage of that organization. It is intended that linearized PDF is view-only, an incremental update is allowed, but the resulting PDF is no longer linearized and treated as ordinary PDF.[8, Appendix F]

2.4 Document Structure

A PDF document can be regarded as a hierarchy of objects contained in the body section of the PDF file. At the root of the hierarchy is the documents' *catalog* dictionary. Most of the objects in the hiearchy are dictionaries. Parent, child and sibling relationships within the hiearchy are defined by dictionary entries whose values are indirect references to other dictionaries.[8, Section 3.6]

2.4.1 Catalog

The root of the document hiearchy is the catalog dictionary. The catalog is located by the **Root** entry in the trailer of PDF file (see 2.3.4). The catalog contains references to other objects defining the document's contents, outline, pages and other attributes. To see all atributes go to pages 139-142 in [8, Section 3.6.1].

2.4.2 Page Tree

The pages of a document are accessed through a structure known as a *page tree*. A *page tree* defines the orderding of pages in the document.[8, Section 3.6.2]. The tree contains nodes of two types:

- intermediate nodes - *page tree nodes*
- leafs - *pages*

Page

A *page object* is a dictionary specifying the attributes of a single page of a document. Typical entries are Parent, LastModified, Resources, MediaBox and Contents. To see all atributes go to pages 145-148 in [8, Section 3.6.2].

2.4.3 Content Streams

Content streams are the primary means of describing the appearance of pages and other graphical elements.

A *content stream* is a PDF stream object whose data consists of a sequence of instructions describing the graphical elements to be painted on the page. The instructions are represented in the form of PDF objects. However, content streams are intended to be interpreted and acted upon sequentially. Each page of a PDF document is represented by one or more content streams. Content streams are also used to package sequences of instructions as graphical elements, such as forms, fonts, and annotation appearances.[8, Section 3.7.1] These entries are stored inside so called *Resource* dictionaries.
Depending on the internal file structure, content streams might occupy just a small percentage of the overall file size or almost an entire document.[9]

2.4.4 Marked Contents

Marked-content operators identify a portion of the PDF content stream as a *marked-content element* of interest to a particular application or PDF plug-in extension.[8, Section 10.5] Marked-content elements fall into two categories, one where the operator designates a single *marked-content point* in the content stream and other, where the operator marks a *marked-content sequence* of objects within the content stream.

Marked content might identify a set of related graphical objects as a group to be processed as a single unit or a text-processing application might use it to maintain a connection between a footnote marker in the body of a document and the corresponding footnote text at the bottom of the page.[8, Section 10.5]

Chapter 3

Analysis

This chapter starts by introducing and comparing several popular serialization formats. Next, we are going to introduce fundamentality of the General Data Protection Regulation and after that, we will focus on PDF parsing and PDF page structure.

3.1 Serialization Formats

Serialization is the process of transforming data structure or object state into a format that can be stored, transmitted or reconstructed (deserialized) later on. The main distinguisher between serialization formats is whether data is stored in a human-readable way (text store) or in a computer-readable way (binary store). While the text formats are generally more used, binary formats achieve smaller file sizes and faster serialization times in trade-off for more complex implementation. For purposes of this thesis, we are going to compare the most popular serialization formats from both worlds.

3.1.1 JSON

Javascript Object Notation (JSON) is the most widely used text data interchange format. JSON is a nested format meaning records are stored in an n-level hierarchy and there exists a schema to describe its structure. Records consist of attribute-value pairs and array data types. Bellow, we can see an example of JSON syntax:

```
{
"firstName": "John",
"lastName": "Doe",
"age": 24,
"address": {
"streetAddress": "21 2nd Street",
"city": "New York",
"state": "NY"
},
"phoneNumbers": [{
```

```
"type": "home",
"number": "212 555-1234"
}]
}
```

JSON is supported by almost every popular programming language.

3.1.2 Protocol Buffers

Protocol Buffers (Protobufs) are a flexible, efficient, binary, language-neutral automated mechanism for serializing structured data. For use in communication protocols, data storage, and more.[10] Protobuf is developed by Google under an open-source licence.

The user specifies data structures, called messages inside `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers, booleans, strings, raw bytes, or even other protocol buffer message types, allowing the user to structure data hierarchically. It is also possible to specify optional fields, required fields, and repeated fields. Example of `.proto` file:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = MOBILE];
  }
  repeated PhoneNumber phone = 4;
}
```

Once the message is specified, language specific-protocol buffer compiler compiles `.proto` files and generates data access classes. These classes provide simple accessors for each field as well as methods to serialize/parse the whole structure to/from raw bytes.[10] Protobuf currently supports C++, C#, Dart, Go, Java and Python programming languages.

3.1.3 Apache Thrift

Thrift is an interface definition language (IDL) binary communication protocol used for serializing structured data.[11] Although originally developed at Facebook, Thrifts are now developed as an open-source project under the Apache Software Foundation.

The usage of Thrift is similar to Protocol Buffers. After defining structures inside `.thrift` file, specific language compiler is then used to generate data access classes for serialization and deserialization. Example of `.thrift` file can be seen below.

```
enum PhoneType {
  HOME = 0;
  WORK = 1;
  MOBILE = 3;
}

struct PhoneNumber {
  1: string number;
  2: optional PhoneType type;
}

struct Person {
  1: string name;
  2: i32 id;
  3: optional string email;
  4: list<PhoneNumber> phones
}
```

Apache Thrift currently supports a wide variety of programming languages.

3.1.4 Apache Avro

Avro is a row-oriented data binary serialization framework developed within Apache's Hadoop project.

Avro is very similar to Thrift and Protocol Buffers, the main difference is that Avro does not require code generation because data is always accompanied by JSON schema. This results in less type information needed to be encoded with data and thus resulting in smaller file sizes.

3.1.5 Comparison

We are going to compare file formats mentioned above by following main criteria:

- Serialization/deserialization speed

- Resulting file size
- Quality of documentation

Python support is also required because the code of this thesis will be written in python language according to thesis assignment. All mentioned serialization formats have support for Python language so going forward we can disregard this requirement.

Benchmark

The performance of each framework differs depending on whether we serialize/deserialize small or big objects.[12] We assume that a typical PDF file we process is about 1-5 pages long. The PDF file size for this kind of file may differ between tens or hundreds of kB, for digital file, to units of MB for a scanned file. With this assumption we based our decisions on benchmark for bigger objects from CriteoLabs.[12] Note that the real desired file size benchmark is probably between CriteoLabs small and big objects. Benchmark was conducted in 2017 using C# implementations of frameworks.

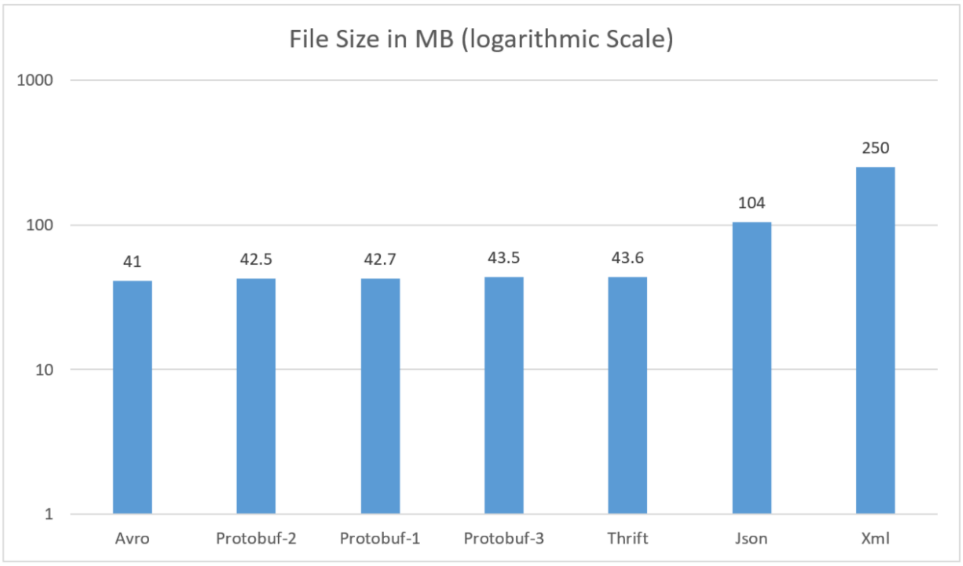


Figure 3.1: Comparison of file sizes [1]

As presented in [1] (seen in Figure 3.1), we can see that binary formats are considerably smaller than text formats with Avro being the smallest. Note that the Protobuf-1-3 shown above differs in schema implementation, we are going to take a mean time.

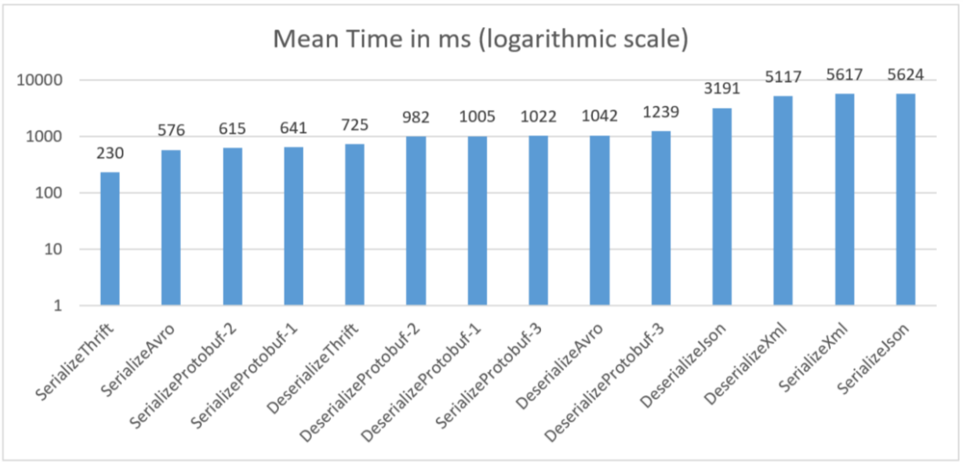


Figure 3.2: Comparison of serialization/deserialization times [2]

From speed benchmark (seen in Figure 3.2) we can see that Thrift is fastest, followed by Avro and Protobuf. We can see that for binary formats serialization times are always smaller than deserialization times and for text formats, it's the other way around.

Results

Protobuf, Thrift and Avro have similar performances in terms of file sizes and serialization/deserialization times. But the slightly better speed of Thrift does not overweight detailed Protobuf documentation. Both Thrift and Avro are lacking in that department and thus for our purposes, we chose Protobuf as our serialization file format.

3.2 General Data Protection Regulation

General Data Protection Regulation (GDPR) is a European Union legislation, which came into full effect on 25 May 2018, on data protection and privacy for all individual citizens of European Union (EU) and the European Economic Area (EEA). One of its aims is to give individuals control over their personal data. GDPR applies to all business in EU, but if business processes the personal data of EU citizens or residents, or offers goods or services to such people, then GDPR applies to it even if the business is not in EU.[13]

3.2.1 Personal Data

According to GDPR *personal data* means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person.[14] In practice, the typical examples of personal data include (but not limited to these):

- Full name
- Date of birth
- Address
- Email Address
- IP Adress
- Telephone number
- Religious belief
- Place of work

3.2.2 Consent and Privacy Rights

To be able to process personal information explicit consent must be given and documentary evidence of the consent must be kept. Individual then has the right to be informed, right to erasure, right of access, and others.

3.2.3 Data Security

Bussines is required to handle data securely by implementing appropriate technical and organizational measures.[14]
Technical measures mean anything from requiring your employees to use two-factor authentication on accounts where personal data are stored to contracting with cloud providers that use end-to-end encryption.[13]
Organizational measures are things such as staff training, adding a data privacy policy to your employee handbook or limiting access to personal data to only those employees in your organization who need it. GDPR urges to limit access to personal data only to those who need it.

3.2.4 Key Takeaways

To comply with the main GDPR points mentioned above, our application will have to be designed to exclude the actual contents of documents from being serialized as much as possible. The application will instead focus mainly on graphical elements, page layout, histograms, document metadata and similar features.

3.3 PDF Parsing

Most PDF files look like they contain well-structured text. But the reality is that a PDF file does not contain anything that resembles paragraphs, sentences or even words.[15] This makes extracting meaningful pieces of information, mainly text ones, from PDF files difficult. The characters that compose a paragraph are no different from those that compose the table, the page footer or the description of a figure. Luckily, as mentioned in 2.1, the PDF file contains some structural information. To help us reconstruct some of these structures we will use a PDFMiner[1] library.

[1]Specifically PDFMiner.six https://github.com/pdfminer/pdfminer.six

Here is a look at PDFMiner's most notable features:

- Parse PDF objects into Python objects.
- Analyze and group-text in a human-readable way.
- Extract text, images and more.
- Support for various font types.
- Support for basic encryption (RC4).
- Layout analysis.

3.3.1 Layout Analysis

PDFMiner uses heuristics on the positioning of characters for layout analysis. Layout analysis consists of three stages: it groups characters into words and lines, then it groups lines into boxes and finally it groups textboxes hierarchically.[15] All of this is done using three margins: `char_margin` (M), `word_margin` (W) and `line_margin` (L). If the distance between two chunks is smaller than specified margin they get grouped into one. Furthermore, it may be required to insert blank characters (spaces) between words, because blank between words may not be represented as spaces. The graphical portray of these margins can be seen in Figure 3.3.

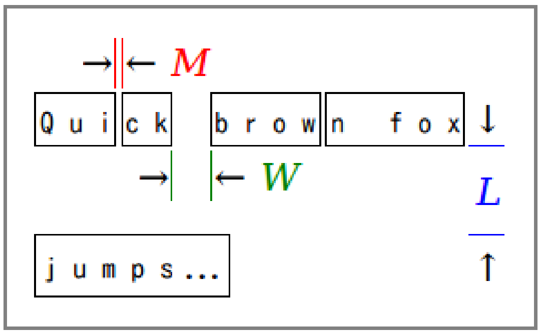


Figure 3.3: Text group margins [3]

The output of the layout analysis is an ordered hierarchy of layout objects (seen in Figure 3.4). Every object has its own *bounding box* and *sequence id*. *Bounding box* is a box surrounding object and thus defining its area. Bounding box is defined by four coordinates min(y), max(y), min(x), max(x). If two objects are merged their bounding boxes are merged as well. An example of bounding boxes can be seen around texts in Figure 3.3. *Sequence id* represents the order in which layout objects render. Object with sequence id 0 is rendering first.

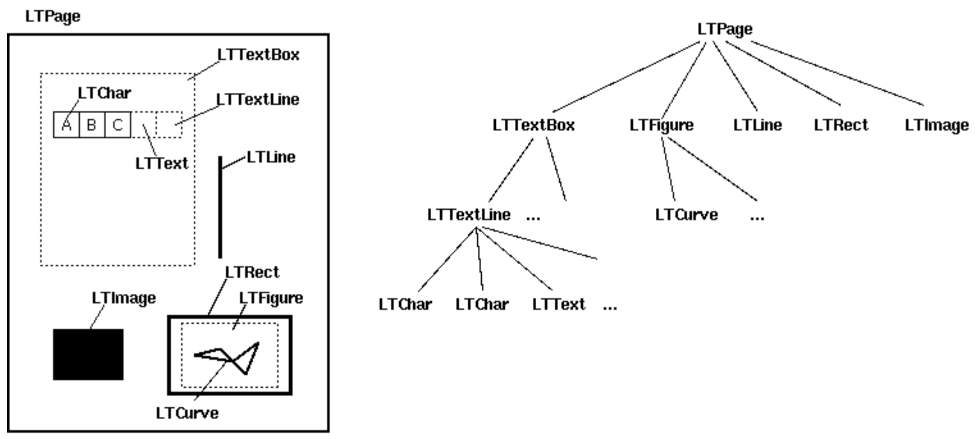


Figure 3.4: Hierarchy of layout objects [4]

The layout analysis is done per page, as shown in Figure 3.4. PDFMiner, PDF parsing and layout analysis will be the first thing that we do once we input PDF file into our application.

Chapter 4

Information Extraction

In this chapter, we first introduce types of PDF forgery and most used techniques, then we provide a detailed description of extracted features that will allow us to represent format as diverse as PDF in uniform form and at the end we are going to take a look at how we are going to comply with GDPR legislation.

4.1 PDF Forgery

With PDF being the de facto standard for the exchange of documents and with the modernization of many business processes, invoices, bank statements, payslips and other electronic documents are increasingly relied on as proof of income, identity, financial stability or proof of ownership. These facts may draw fraudsters into forgery of PDF documents.

We identify three main types of PDF forgeries:

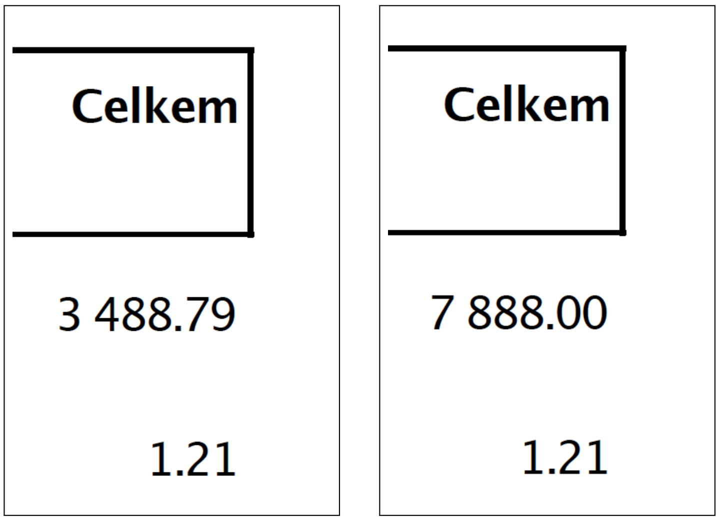
- **Transaction modifications** - These modifications include change of invoice total and invoice items, increased salary on payslips or changes to amounts on bank statements. For instance an attacker may increase the number of items in invoice thus recalculating (increasing) the invoice total and then have it paid by factoring firm.
- **Payment details modifications** - Fraudster takes a legitimate invoice and only modifies an account number to direct funds into his account. Profits are usually laundered.
- **Identity modifications** - Fraudster takes a legitimate document and modifies the document's original identity to another entity. For instance, this allows the attacker to claim payslips or bank statements as his own, thus proving a higher income and better financial stability.

Every business working with PDF documents can be affected by forgery of PDF documents. The most commonly affected types of businesses are internet lending, invoice financing and car financing.

The most common way of modifying a PDF document's info is either by addition of new content over old, we call this type of modification **word overlap**, or by appending new content to old one, we call this type of modification **word append**.

4.1.1 Word Overlap

Word overlap is when information is hidden or replaced by another object. Since the most important information on invoices, payslips and bank statements are text ones (meaning characters or numbers), we investigate only when any text information is overlapped by another object (text, annotation, image, line, rectangle, curve). An example of word overlap, where invoice total was completely rewritten can be seen in Figure 4.1



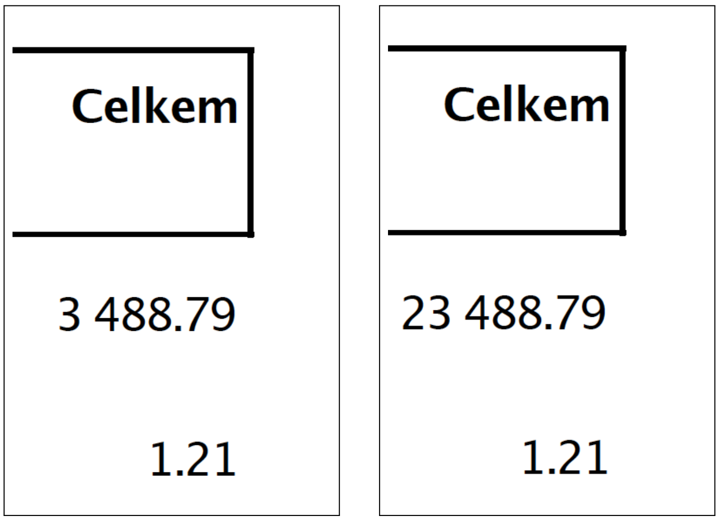
(a) : Original document **(b) :** Changed total

Figure 4.1: Example of word overlap

By word overlaps fraudster may change anything on PDF document and it won't be detectable by the human eye unless he uses incorrect font or font size. Even then the modification can be found only if someone deems the document as suspicious and performs a detailed check. A typical use case is change of invoice total and items, change of identity or change of payment details.

4.1.2 Word Append

Word append is when new content is appended to the original. The difference between word overlap and word append is that with word appends no information is covered, only added. For instance, by appending (or rather prepending) number 2 in front of some total, value is suddenly increased by one order with minimal changes to the document. Visualization of this example can be seen in Figure 4.2.



Celkem

3 488.79

1.21

Celkem

23 488.79

1.21

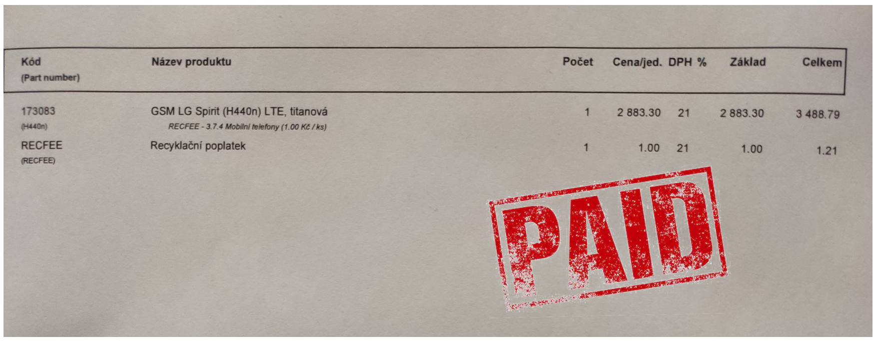
(a) : Original document **(b) :** Changed total

Figure 4.2: Example of word append

Again, this type of modification is not recognizable by the human eye. A detailed description of how we detect word overlaps and word appends is described in Representation chapter 5.

4.1.3 Image Splicing

Word overlaps and word appends are modifications made on digital document but what about if the document is scanned, meaning the document is made only out of images (more on scan types in 4.2.2) so there is no text (by text we mean PDF text components 4.2.2) information to cover. Image splicing is a simple and common image tampering operation, where region from one image is pasted into another image with the aim to change its content. We detect image splicing by finding noise inconsitencies within image. We call regions with noise inconsistencies *noise tampered regions*. An example of image splicing can be seen below in Figure 4.3.



Kód (Part number)	Název produktu	Počet	Cena/jed.	DPH %	Základ	Celkem
173083 (H440n)	GSM LG Spirit (H440n) LTE, titanová *RECFEE - 3.7.4 Mobilní telefony (1.00 Kč / ks)*	1	2 883.30	21	2 883.30	3 488.79
RECFEE (RECFEE)	Recyklační poplatek	1	1.00	21	1.00	1.21

PAID

Figure 4.3: Paid stamp added to invoice

A typical use case is to use header from another document and paste it into the original one. Details about the process of detecting image splicing by noise variances are described in section 5.6.

4.2 Properties and Features

We decided to sort extracted properties and features into 5 following categories according to its type:

- **Simple features** - represented simply by its value, e.g., file size, sha256 file name
- **Structure features** - objects describing PDF internal structure mentioned in 2.3 and 2.4, e.g., catalog, trailer
- **Layout features** - features describing document/page visual layout, e.g., scan type
- **Origin features** - features describing document origin, e.g., creation date
- **Modification features** - features describing modification of document, e.g., word overlaps

Table 4.1 shows in what category every feature belongs. Note that we extract documents property and we can create a feature from it by assigning a boolean value to it, e.g., signature and has signature. Both of these values are in table 4.1. Also note that feature can be inside more than one category, that is because feature can be used to describe the origin of the documents and as well if the origin is fraudulent therefore if there was any modification.

Category	Properties and features
Simple	file size, sha256 file name, number of pages, PDF version, character histogram, lenght of texts, noise, number of components, number of texts, number of images, number of marked contents, number of annotations, number of rectangles, number of curves, number of rectangles, number of lines, number of fonts, number of Windows/Mac fonts, title, keywords, number of horizontal/vertical lines
Structure	catalog, xrefs, trailer, pages, PDF comments, object positions, font character sets, stream operations, stream whitespaces, fonts encryption parameters, has encryption parameters, signature
Layout	images, texts, lines, curves, rectangles, component matrix, links, scan type, is scanned, is scanned with OCR, is digital, file annotations, text annotations, marked contents
Origin	metadata, xmp metadata, author, producer, creator tool, creation date, modify date, Mac fonts, has Mac fonts, Windows fonts, has Windows fonts, program indicators listed in modification category
Modification	word overlaps, has word overlaps, word appends, has word appends, signature, is signed, has signature invalid byte range, signature date, has Windows and Mac fonts, noise tampered regions, has noise tampered regions, unused fonts, has unused fonts, text annotations, has text annotations, text modified regions, has text modified regions, is modified per modify date, is modified with Photoshop, is modified with Gimp, is modified with Affinity, is modified with iText, is created by PDF Expert, is created by Sejda, is created by PDF Pro, is created by PDFfiller, is created by PDF Guru, is created by LibreOffice, is created by Google Sheets, is produced by Microsoft, is produced by Microsoft Word/Excel/PowerPoint/Access, is created by microsoft print to PDF

Table 4.1: Properties and features splitted into categories

Features and properties are further divided into page and document features. A detailed description of important features follows.

4.2.1 Structure Features

Encryption Parameters

A PDF document can be encrypted to protect its content from unauthorized access. Encryption applies to all string and streams in PDF document but not to integers or booleans.[8, Section 3.5] If a PDF document is encrypted we extract information about the type of encryption and documents user access permission.

Information about encryption is stored inside document's *encryption dictionary*, which is the value of the `Encrypt` entry inside document's trailer dictionary (see table 2.1). The absence of this entry from trailer dictionary means that the document is not encrypted.

We extract:

- Algorithm - integer indicationg type of algorithm is under value of entry `V`. Possible values are:
  - 0 - Undocumented and no longer supported algorithm
  - 1 - AES RC4 with encryption key of length 40 bits
  - 2 - AES RC4 with length of encryption key above 40 bits
  - 3 - An unpublished algorithm
  - 4 - *Security handler*
- Filter - Identifies the file's *security handler*, a software module that implements various aspects of encryption process and controls access to encrypted document.
- Revision - A number specifying which revision of the standard security handler should be used to interpret encryption dictionary.
- Length - The length of encryption key in bits.
- User access permissions - set of flags specifying which operations are permitted when the document is opened. We further parse these flags according to user access permission table (see [8, Section 3.5, pages 123,124]). Possible permissions are:
  - Fill forms
  - Insert or delete pages
  - Modify the document
  - Print the document in high/low quality
  - Create or modify forms and signatures
  - Copy and extract text from document
  - Add or modify text annotations

Fonts

We extract information about fonts used inside PDF document. Information about fonts are stored inside catalog as dictionaries with value of entry `Type` Font. If there is a font present, we save the following attributes:

- Subtype - PDF type of font, for more see table at [8, Section 5.4, page 411].
- Base font - The PostScript name of the font, e.g., Helvetica.
- Encoding - Specification of the font's character encoding. Encoding is either dictionary or one of the three following predefined values:
  1. `MacRomanEncoding` - Font comes from Mac OS.
  2. `WinAnsiEncoding` - Font comes from Windows.
  3. `MacExpertEncoding`- Associated with experts fonts that contains additional characters.
- Character set - If there is associated *font descriptor*, a character set is saved in string listing the character names divided by slash (/). We further parse these names directly into characters.

We use font encoding to determine documents origin operating systems. When PDF document contains both fonts from Mac and Windows operating systems, we consider the file as modified since it is unlikely that one PDF document was created and modified on different operating systems without fraudulent intent.

4.2.2 Layout Features

Scan Type

Scan type is mainly a page feature. Scan type of pages determines the final scan type of whole document. A PDF page can be one of three scan types:

- `NOT_SCANNED` - PDF page is created inside computer using appropriate sofware. If it isn't content copy protected, you can select and copy-paste text without any problems.
- `SCANNED_WITH_OCR` - PDF page is created using a scanner with *Optical Character Recognition* (OCR). As name reveals, OCR tries to convert scanned text (printed or handwritten) into machine-encoded text. In this type of document you can copy-paste but unfortunately OCR accuracy is good only for printed text, accuracy for handwritten text is mostly lacking.
- `SCANNED_WITHOUT_OCR` - PDF page is created using a scanner without OCR. The electronic document is then composed of image or images.

A PDF document than can be one of four scan types:

- `DIGITAL` - All pages of the document are `NOT_SCANNED`.
- `SCANNED_WITH_OCR` - All pages of the document are `SCANNED_WITH_OCR`.
- `SCANNED_WITHOUT_OCR` - All pages of the document are `SCANNED_WITHOUT_OCR`.
- `MIXED_SCAN` - Combination of `SCANNED_WITHOUT_OCR` and `SCANNED_WITH_OCR` pages.
- `MIXED` - Combination of `NOT_SCANNED`, `SCANNED_WITHOUT_OCR` and `SCANNED_WITH_OCR` pages.

How we determine page and document scan type from binary information will be explained later on in Chapter 5.

Components

Let's call a building block obtained from layout analysis (3.3.1) a component. Types of blocks are texts, images, lines, curves and rectangles. As mentioned in 3.3.1, every component has a bounding box and sequence id. In addition, as layout analysis is done per page we also obtain *page id*, id of a page on which component is.

Every type of compontent has its special attributes, for instance text component has a text and font id attribute. All components attributes can be seen in Appendix B. All components are saved in anonymized state to comply with GDPR (see 4.3).

Text Annotations

Text annotations represent a text attached to a point in PDF. Some programs represent text annotations as sticky notes but some (for instance Preview) allow to attach text inrecognizable from the rest of the document by a human eye. It is one of the easiest ways of adding a text into PDF and thus making it one of the easiest type of modification to make.

4.2.3 Origin Features

Metadata and XMP Metadata

Document's metadata contain general information about document, such as the document's title, creator tool, and creation and modification dates. Document's metadata may be added or changed by software or plug-in extension. Unfortunately, there is no history of metadata so we can retrieve only the last version of document's metadata. This means that if PDF document was created by Adobe Acrobat and then modified by iText, only information about iText will be contained in relevant metadata entries.

Metadata can be stored in PDF document in both following locations:

1. Inside document information dictionary (mentioned in 2.1).
2. In *metadata stream* associated with the document.

To distinguish between these two we call data from information dictionary **metadata** and data from *metadata stream* **XMP metadata**, XMP stands for *Extensible Metadata Platform.* Typical and for us valid entries for both metadata are:

- Title - Document's title.
- Author - Document's author.
- Creator - If the document was converted to PDF from another format, the name of the application that created the original document from which it was converted.
- Producer - If the document was converted to PDF from another format, the name of the application that converted it to PDF.
- Creation date - The date and time the document was created, in human-readable form. Date includes precision up to miliseconds.
- Modification date - The date and time the document was most recently modified, in human-readable form.

XMP metadata are stored inside `Metadata` entry in document's catalog. Metadata are defined in XML using XMP framework, which provides a way to use XML to represent metadata describing documents and their components and it is intended to be adopted by a wider class of applications than just those that process PDF.[8, Section 10.2.2] If you want to know more about document's metadata, see [8, Section 10.2] inside PDF reference.

4.2.4 Modification Features

Word overlaps, word appends and noise tampered regions were already, in theory, explained in 4.1. Details about how we detect them in practice are covered in Chapter 5 as it is not trivial and page representation needs to be explained first. Description of other interesting modification features follows.

Digital Signature

A digital signature can be used to authenticate the identity of a user and the document's content.[8, Section 8.7] Signatures are created by computing a *digest* of data (or part of the data) in the document and storing within the same document. Signature information is contained inside *signature dictionary.* If document is signed we extract following signature attributes:

- Name - The name of the person or authority signing the document.
- Type - Must be `Sig` if signature is present.
- Filter - The name of preffered signature handler to use when validating this signature.
- Sub-filter - A name that describes the encoding of the signature value and key information in signature dictionary. An application may use any handler that supports this format to validate the signature. [8, Section 8.7]
- Contents - The signature value.
- Byte range - An array of pair of integers describing the exact byte range for digest calculation. Multiple discontiguous byte ranges are used to describe a digest that does not include signature value (`Contents` entry) itself.[8, Section 8.7]
- Certificate - An array of byte strings representing the X.509 certificate chain used when signing and verifying signatures that use public-key cryptography, or a byte string if the chain has only one entry.[8, Section 8.7]
- Signature date - The time of signing.

Signatures can be verified by recomputing the digest of the data according to specified filter, sub-filter, certificate and byte range, and by comparing it to the one stored inside the document. Note that when a digest is computed, the actual value of `Contents` entry is excluded.

We did not implement recomputing of the digest, but we included a check for validity of signature byte range followingly:

$$0 == doc\_start < sig\_start < sig\_end < doc\_end$$

If the formula above holds true, we say that document's signature has valid byte range.

Is Modified per Modify Date

We check whether document's `creatation date` and `modification date` obtained from document's metadata and XMP metadata match. We first look into metadata for `creation/modification date` value and, if the value is not present there, we look into XMP metadata.

We say the document is modified per modify date if `creatation date` and `modification date` is filled and dates do not match and if there is `modification date` present and there is no `creatation date` present. This check is added because if there is a document without metadata and it is then modified with software that fills the `modification date` entry of metadata we end up with modified document without `creatation date` but with `modifification date` inside metadata.

Program Indicators

We included indicators for specific software derived from metadata imprint of these PDF or graphical programs. These indicators were derived from try and examine method. We currently have checks for the following programs:

- Adobe Photoshop
- Gimp
- Affinity
- iText
- PDF Expert
- PDF Pro
- PDFfiller
- PDF Guru
- Sejda
- LibreOffice
- Google Sheets
- Microsoft Word
- Microsoft Excel
- Microsoft PowerPoint
- Microsoft Access
- Created by Microsoft print to PDF

As you can see, the that software in the list ranges from graphical software (Photoshop, Gimp, Affinity) to common document software (Microsoft Office). We do not detect any specific modification, only the imprint. But some software is more suspicious than other. For instance, editing PDF file in Photoshop is not a typical use case, but creating PDF from Word document might be. Because of this, program indicators fall not only into the modification category but also into the origin category.

Text Modified Regions

Fonts in 4.2.1 are all the fonts available in the document, but we don't know where the fonts are used. To determine *text modified regions*, which are regions where was added text from a different operating system, we extract page associated fonts from page resources (mentioned in 2.4.3). Every extracted front from page resources has a unique font id, which we can use to match with font id attribute of text component.

If page contains both fonts and associated texts from Windows and Mac operating systems, we declare shorter texts (sum of all characters written by that OS on specified page) as fraudulent.

Unused Fonts

We deem suspicious if PDF document contains a font that is unused. The font is just increasing the size of a file and should be optimized out. This can occur when text written by specific font is deleted from document.

4.3 Anonymization

To comply with GDPR we are not going to save any data that could identify any person or entity. However, we are not going to anonymize extracted modification features, because that is an added value of our application. This also includes metadata that contain only general information about the document as software that was used to create the document.

Things that fall under anonymization:

- File name - We save only sha256 hash of the file name.
- Texts - All numbers are changed to `9`, all lowercase letters are changed to `a`, all uppercase lettter are changed to `A`, and all special characters are changed to `#`.
- Images - We will not save any actual image info, we save image height and width, position on page (bounding box) and color space.
- PDF catalog - PDF streams inside of catalog, so an original PDF document could not be reproduced.

Rectangle/curve/line components don't contain any compromising data.

Chapter 5
Representation

In this chapter, we firstly look at an overview of the application, then we explain how we represent a page of PDF document and further explain extraction and representation of features dependent on it. Lastly, we explain how serialization and deserialization of extracted information is done.

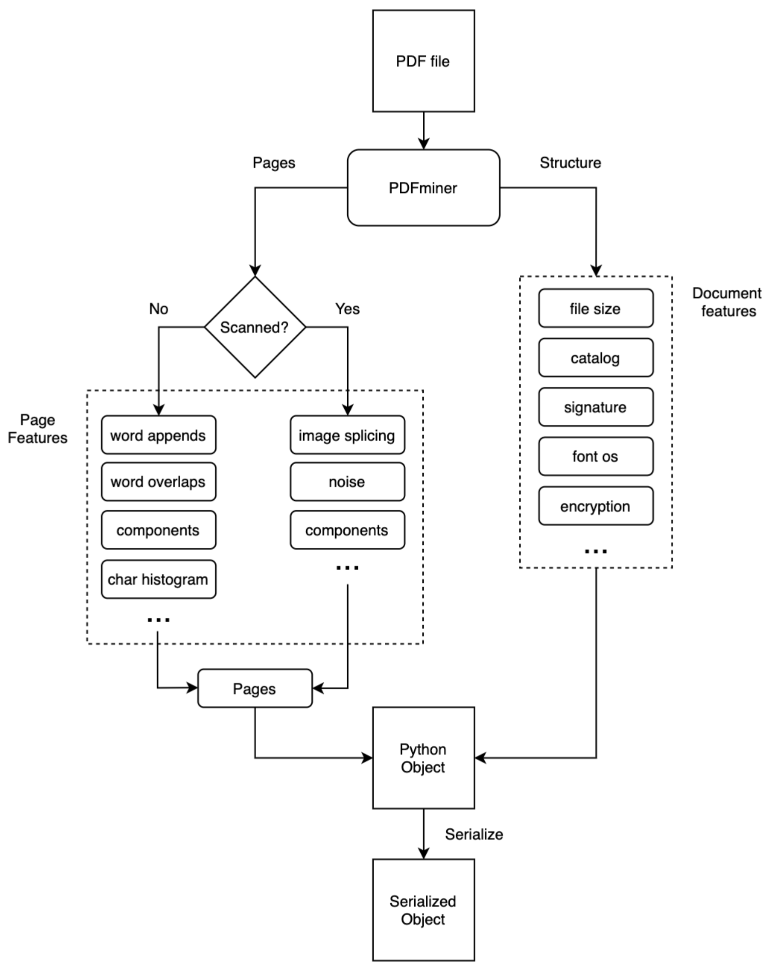


Figure 5.1: High-level application overview

5.1 High-level Overview

In Figure 5.1 you can see a high-level overview of application architecture. A PDF file is first loaded and parsed using PDFminer library, then we separately extract document features and page features. Page features are extracted per page and extracted feature set differs depending on the page's scan type. Pages and their features are then merged together with document features into python object. Python object is then serialized into a protobuf file.

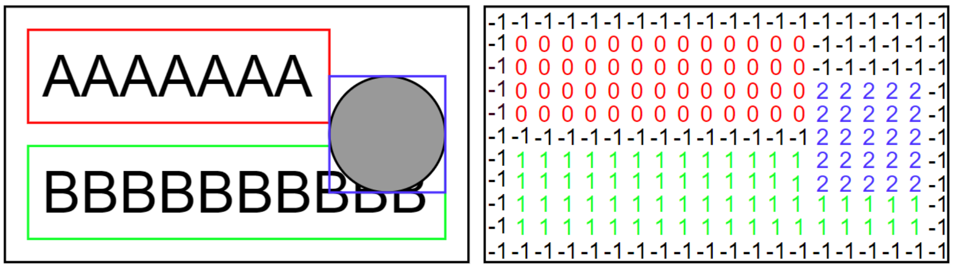
5.2 Page Representation

As we know from layout analysis (3.3.1), every component has its own sequence id and bounding box. Having a list of page components does does not provide the knowledge of how the page looks because components can overlap each other. However, with a list of page components and their sequence ids, we can reconstruct a visual representation of page followingly:

```
Data: List of page components
Result: Component Matrix
initialization: create array of page size initialized to -1
for component c in increasing sequence id order do
    if c influences visual page in any way then
        write c.seq_id to page array according to it's bounding box
    end
end
```

Algorithm 1: Creating Component Matrix

That way we end up with an array representing the page. We decided to call this representation a **Component matrix**. An illustration of a small component matrix can be seen in Figure 5.2.



```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1 -1
-1  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1 -1
-1  0  0  0  0  0  0  0  0  0  0  0  0  0  2  2  2  2  2 -1
-1  0  0  0  0  0  0  0  0  0  0  0  0  0  2  2  2  2  2 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  2  2  2  2  2 -1
-1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2 -1
-1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2 -1
-1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 -1
-1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Figure 5.2: Example of small component matrix

On the left, you can see an example PDF page consisting of two texts and one overlapping image. Their respective bounding boxes are shown as well. On the right, you can see its corresponding component matrix representation. Sequence ids are shown with the same color as their respective bounding boxes. Note that `-1` is the default value, meaning nothing is rendered at this position. We will further use component matrix to extract scan type, word overlaps and word appends features from PDF page.

5.3 Word Overlap Detection

Component matrix gives us a visual representation of a page, so we know which components are visible but we don't know any information about components underneath. To detect which components are overlapped, and specifically by what component is component overlapped, we need to create *component overlap map*.
Component overlap map is a list of tuples where the first value is a component and the second value is a list of overlapping components. To create component overlap map, we recreate component matrix, but instead of writing sequence id value into page array, we look at current component's area in component matrix and examine all present sequence ids. Because we know that sequence ids written at the desired area in component matrix are visible, all sequence ids that do not match sequence id of our currently investigated component are overlapping our current component. Sequence ids are then mapped into component objects. Pseudocode follows:

```
Data: List of page components, component_matrix
Result: overlap map
initialization: overlap map = Dict[int,Set[int]]
for component c in increasing sequence id order do
    if c influences visual page in any way then
        for unique id in component_matrix[c.area] do
            if id!=-1 and id!=c.seq_id then
                add id to dictionary with key c.seq_id
            end
        end
    end
end
```

Algorithm 2: Creating overlap map

Component overlap map will be further used when extracting scan type 5.5.

To extract word appends, we will first flatten the representation to pairs of component (`c_under`) and its overlapping component (`c_over`) called component tuple. Since word overlap can occur only when text component is overlapped by another component (4.1.1), we can filter component tuples and disregard any pair where `c_under` is not text. Next, we apply a series of filters to obtain the final result.

We filter cases when:

- `c_over` is rectangle or curve component and is not filled - Text in PDF may be inside of a table and that table can be made out of rectangle components. However, the rectangles have no fill so text can be visible. We want to filter these occurrences.
- `c_over` is text and text value of `c_over` is same as text value of `c_under` - Overlap of text with same text does not hide any information.
- `c_over` is text and text is not visible (text component attribute, meaning not rendered) - Text is not rendered in PDF so no information is hidden.
- `c_under` text is composed only of whitespace characters - Overlap of whitespace character is not covering any information.
- `c_under` text is not visible - Overlap of invisible text does not hide any information.
- `c_over` is not covering more than one character of `c_under` - `c_over` covers `c_under` only slightly, this can be for instance caused by low vertical spacing.

Final selection is then converted to `overlap` object, which is later serialized, containing page id, hidden text, replacement text and `c_over` bounding box. When `c_over` is not text component, the value is name of component type, e.g., Image.

5.3.1 Improving Accuracy

During tests, we noticed some false positive (FP) overlaps findings in near proximity of images. The text was still visible yet we had a finding. The reason behind them was that PDF supports transparent images and we only took into account bounding boxes when creating component matrix. An artificially created example of such case can be seen in Figure 5.3.

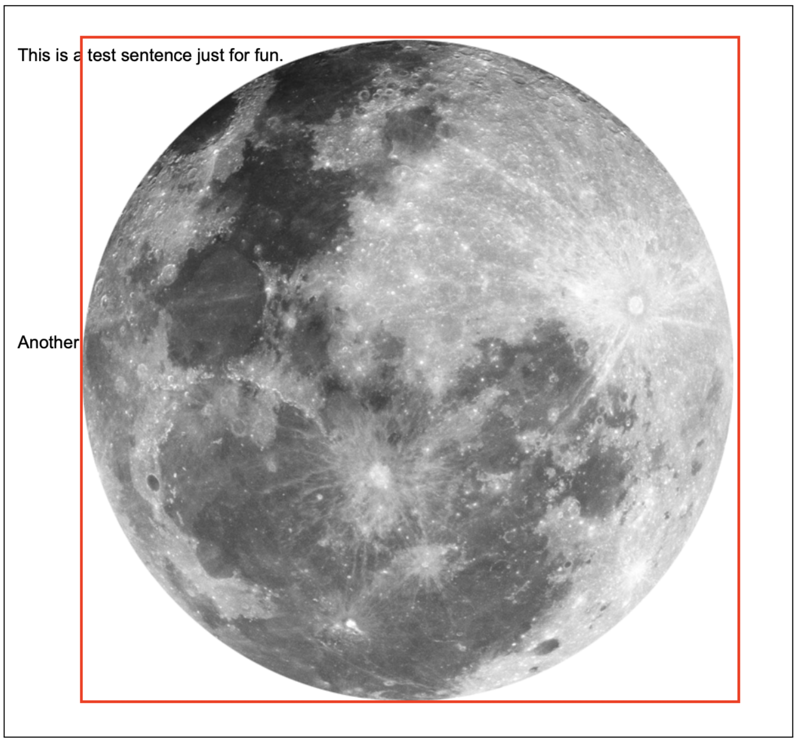


Figure 5.3: Transparent image with its bounding box drawn in red and in top left corner text falsely overlapped by transparent image

To avoid such cases, we implemented support for image transparency into the application.

Image Transparency in PDF

In PDF transparency is achieved through a usage of mask. There are three types of mask entries inside image dictionary:

- `ImageMask` - Mask is already present in the image.
- `Mask` - Either specifies separate image to be used as an explict mask specifying which areas of the image to paint and which mask out or specifies range of colors to be masked out whenever they occur within the image. This technique is known as *color key masking*.[8, Section 7.5.4] Both types do not iplement true transparency, they only mask out specific points by masking color (white or black).
- `SMask` - Or an alpha mask, is a *soft-mask image* specifying alpha channel for original image, where alhpa mask is specified image data is left out and objects bellow are visible.

Results

The algorithm 1 is changed by adding a check if the current component is an image component and if it has specified alpha mask. If so, we write sequence ids into component matrix only at positions where values inside alpha mask are above specified transparency threshold. For algorithm 2, it is similar. First, we have to check if a component is an image component, then we examine only the areas of component matrix where values inside alpha mask are above specified threshold. Example of component with one overlapping image supporting transparency can be seen in Figure 5.4.

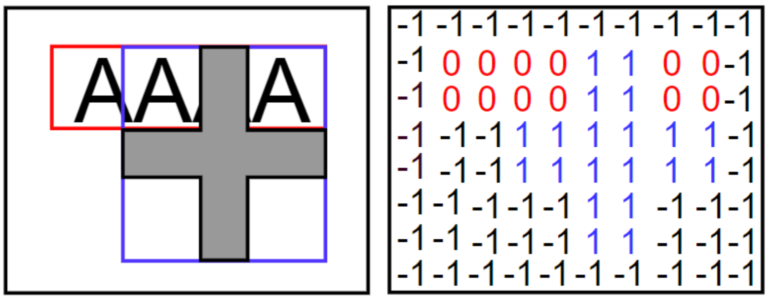
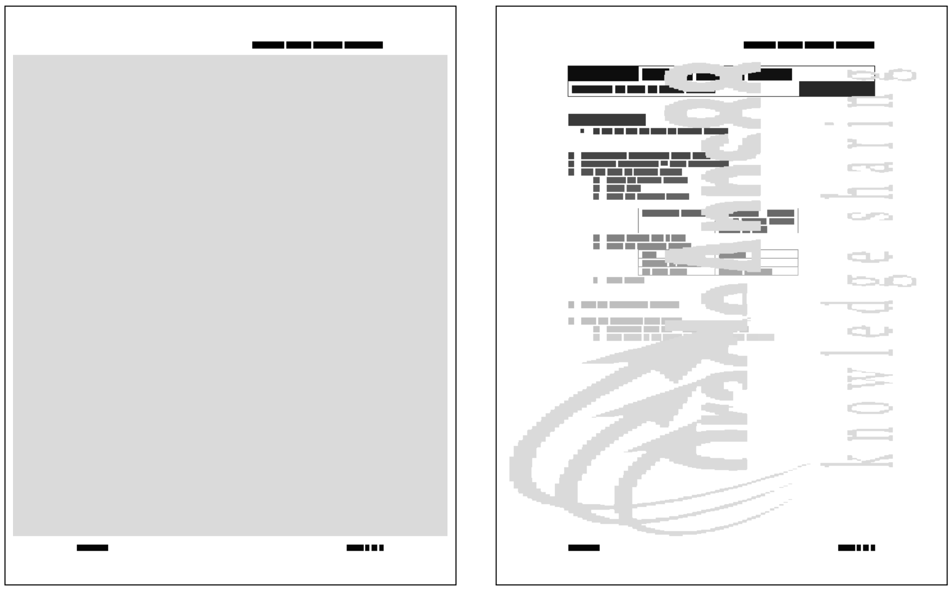


Figure 5.4: Component matrix with support for image transparency

A real-world example of improved accuracy (reduction of PFs) can be seen in Figure 5.5 (each sequence id is represented by different RGB value). The original document has an image over the whole page. If we only took into account bounding boxes, almost all content of the document is classified as covered, but with added image transparency support we can see that higher precision was achieved.



(a) : Before **(b) :** After

Figure 5.5: Visualized component matrix with and without image transparency support

5.4 Word Append Detection

Detection of word appends is based on a succession of sequence ids. Usually, PDFs are created and rendered as they are read, from top to bottom and from bottom to left. Meaning that if we go through components on a page from top to bottom and left to right the value of sequence id should gradually increase for each component, ideally by 1. If we encounter any sequence id that is unproportionally different than sequence id before or after it is suspicious and it could be an instance of word append. To detect such cases, we once again use component matrix. In Figure 5.6, you can see a visualized example of appended 8 to number 60. Normally, number 60 would be one component so it would have one sequence id, but for sake of example, we split it into two components, one for each digit. Image shows only cutout of component matrix, more components are present so, when 8 is appended as a modification of already created file, its sequence id is equal to maximum sequence id +1 (in this case there are 10 components present in matrix, also counting the 8).

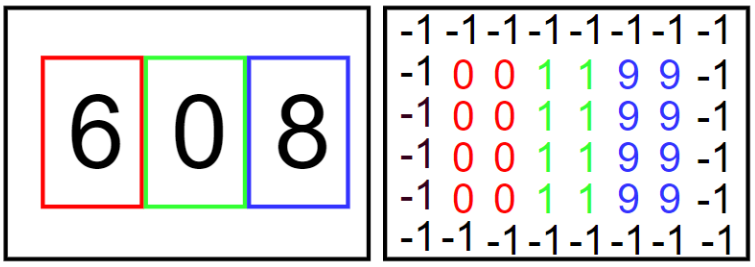


Figure 5.6: Word append shown in component matrix

We start by creating pairs of subsequent sequence ids. To create them, we iterate through component matrix from top to bottom and left to right. We disregard any pair containing default value (-1) or where the numbers are the same. For example 5.6, this would yield pairs `(0,1),(1,9)`. Since we are doing this per line the result would be 8 pairs total. So the next logical step is to filter duplicate pairs and then map sequence ids into component objects. Next, we apply a series of filters to obtain the final result. We filter cases when:

- Components sequence ids are consecutive - If absolute of differences between components sequence ids is 1-3 everything is okay.
- One or both components are not text components - Same as word overlaps word appends are defined only on text components.
- One or both texts are not visible - Word appended by invisible word does not add any additional information.

- One or both texts are not whitespace characters - Append of whitespace character does not add any additional information.

- Components are too far away from each other (not same word) - If components are not seen as one word it can be an instance of word append. The check is done using x coordinates and width of a gap between components.

- Components are not on the same line - Due to different font sizes, we can get pair where words are not on the same line. We check it by comparing y coordinates and height of components.

Final selection is then converted to `append` object, which is later serialized, containing page id, first word, second word, merged bounding box and sequence id distance.

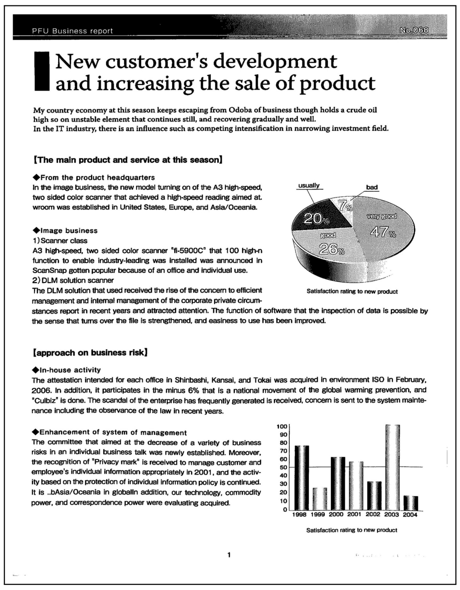
5.5 Scan Type Detection

Each out of three page scan types (`SCANNED_WITHOUT_OCR`, `SCANNED_WITH_OCR`, `NOT_SCANNED`) has its own specific traits.

- `SCANNED_WITHOUT_OCR` - Composed of image or images, the overall number of components is small.

- `SCANNED_WITH_OCR` - Scanned text is either converted to invisible text components that are above the original scanned texts inside images (when you try to select the text you select the invisible top one) or as texts hidden behind original images.

- `NOT_SCANNED` - Are made out of all types of components and contain a significant number of visible texts.



PFU Business report No.068

New customer's development and increasing the sale of product

My country economy at this season keeps escaping from Odoba of business though holds a crude oil high so on unstable element that continues still, and recovering gradually and well.
In the IT industry, there is an influence such as competing intensification in narrowing investment field.

【The main product and service at this season】

◆From the product headquarters
In the image business, the new model turning on of the A3 high-speed, two sided color scanner that achieved a high-speed reading aimed at. wroom was established in United States, Europe, and Asia/Oceania.

◆Image business
1) Scanner class
A3 high-speed, two sided color scanner "fi-5900C" that 100 high-n function to enable industry-leading was installed was announced in ScanSnap gotten popular because of an office and individual use.
2) DLM solution scanner
The DLM solution that used received the rise of the concern to efficient management and internal management of the corporate private circumstances report in recent years and attracted attention. The function of software that the inspection of data is possible by the sense that turns over the file is strengthened, and easiness to use has been improved.

Satisfaction rating to new product

【approach on business risk】

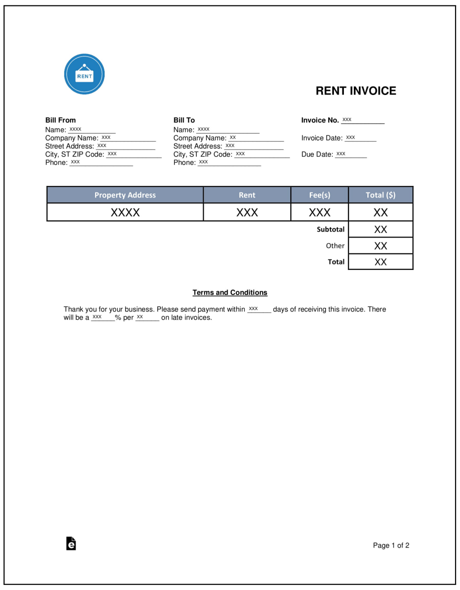
◆In-house activity
The attestation intended for each office in Shinbashi, Kansai, and Tokai was acquired in environment ISO in February, 2006. In addition, it participates in the minus 6% that is a national movement of the global warming prevention, and "Culbiz" is done. The scandal of the enterprise has frequently generated is received, concern is sent to the system maintenance including the observance of the law in recent years.

◆Enhancement of system of management
The committee that aimed at the decrease of a variety of business risks in an individual business talk was newly established. Moreover, the recognition of "Privacy mark" is received to manage customer and employee's individual information appropriately in 2001, and the activity based on the protection of individual information policy is continued. It is ..bAsia/Oceania in globalIn addition, our technology, commodity power, and correspondence power were evaluating acquired.

Satisfaction rating to new product

1

(a) : Scanned page [16]



RENT INVOICE

Bill From	**Bill To**	**Invoice No.** xxx
Name: xxxx	Name: xxxx	
Company Name: xxx	Company Name: xx	Invoice Date: xxx
Street Address: xxx	Street Address: xxx	
City, ST ZIP Code: xxx	City, ST ZIP Code: xxx	Due Date: xxx
Phone: xxx	Phone: xxx	

Property Address	Rent	Fee(s)	Total ($)
XXXX	XXX	XXX	XX
		Subtotal	XX
		Other	XX
		Total	XX

Terms and Conditions

Thank you for your business. Please send payment within xxx days of receiving this invoice. There will be a xxx % per xx on late invoices.

Page 1 of 2

(b) : Not scanned (digital) page [17]

Figure 5.7: Examples of scan types

To detect these traits and classify page's scan type, we first start by calculating image to page ratio (ItP). We first calculate the sum of bounding box areas of non-overlapping image components, which we divide by page bounding box area. If $ItP < 0.2$ and page contains any visible text that is outside of header and footer (header and footer are defined as 15% of a page from top and bottom respectively), we classify page as `NOT_SCANNED`. The check for visible text outside of header and footer is added for cases when a single text, for instance, automatic header or footer, is added to the scannned document. Visible texts are extracted from overlap map. Text is visible if it is visible (in PDF way), it is not made out of whitespace characters and entry for that component inside overlap map yields an empty list.

Next, using component tuple, we extract text components behind images and invisible text in front of images. If more than half of overall text components on page are invisible texts in front of images or if more than half of overall text components on page are texts behind images, we classify a page as `SCANNED_WITH_OCR`. Otherwise as `SCANNED_WITHOUT_OCR`. An example of scan types can be seen in Figure 5.7.

5.6 Noise Variance Detection

To detect image splicing by finding noise inconsistencies withing image, we combined methods from two articles [5], [18]. First, we used a similar process of determining tampered regions as in [5], and second, we used method for calculation of local noise estimate from [18].
The whole process is built upon fact that the noise variance of the image region cropped from one source image is typically different from that of the remaining part of that image, which is obtained from the other source image.[5]

5.6.1 Noise Estimate

Noise estimation is based on the fact that image structures like edges have strong second order differential components and noise estimator should be insensitive to the Laplacian of an image. John Immerkær ([18]) suggests using the difference between two masks L_1 and L_2, each approximating the Laplacian of an image. The used elements of L_1 and L_2 are:

$$L_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} L_2 = \frac{1}{2}\begin{bmatrix} 1 & 0 & 1 \\ 0 & -4 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

And the final noise estimation operator K is the mask operation using mask (kernel):

$$K = 2(L_2 - L_1) = \begin{bmatrix} 1 & -2 & 1 \\ -2 & -4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

Estimate of noise n in image I with width W and height H can be then computed as:

$$n = \frac{1}{6(W-2)(H-2)} \sum_{imageI} (I(x,y) * K)^2 \tag{5.1}$$

The method works only with greyscale images where each pixel has an integer value 0 - 255.

The method performs well for a large range of noise variance values. In highly texturer images or regions, though, the noise estimator perceives thin lines as noise.[18] This can be later a cause of false positives findings.

5.6.2 Region Classification Method

Article [5] proposes that image is first segmented into non-overlapping image blocks. The noise variance at each local image block is then computed using an effective noise estimation method. A clustering step is then employed to separate these image blocks into clusters based on the similarity of their estimated noise.[5] Refined segmentation is then applied to achieve more accurate results. Procedure is visualized in Figure 5.8.

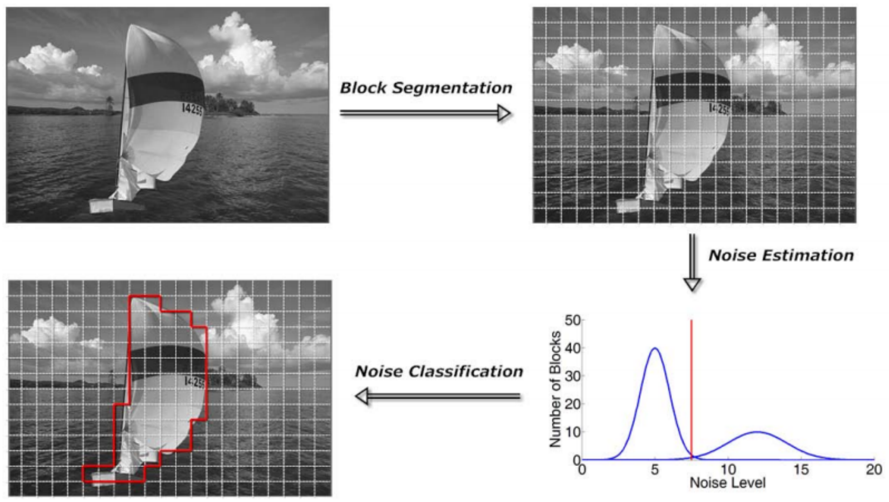


Figure 5.8: Illustration of detection method [5]

5.6.3 Combining Methods

We first have to extract all images that make up the scanned page and then merge them into a single image. Then following the procedure from article [5], we convert image into greyscale (5.6.1) and then segment it into non-overlapping image blocks. The block size is 64 $\times$ 64 pixels. This size reaches the best tradeoff between the stability of the noise estimation and the precision of locality.[5] The noise estimate is then computed for each block using fast noise estimation 5.1. Use of different noise estimation method is one of the main differences between method proposed in [5] and our method. We then apply k-means clustering algorithm to classify the estimated noise levels and group all image blocks into two clusters. The cluster with the less block is then classified as a tampered cluster (we do not assume that fraudster modifies more than half of document's area), we call blocks inside tampered cluster tampered blocks. If the distance between two centers of clusters is bigger then defined threshold, we further apply refined detection using a block size of 32$\times$32 pixels to classify tampered regions more accurately.

Document Specific Method Adjustments

Method proposed in [5] is focused on general image splicing. We are focused on image splicing within scanned PDF documents. The difference is that standard scanned page of a document consists of white background (paper) and black foreground (texts), whereas general image has a wide variety of colors. This color distribution for scanned pages automatically creates two clusters with different noise variances. If the distance between two cluster centers is above specified threshold we further implement a series of checks to combat such behaviour. The checks in order are:

- We disregard any image blocks containing only white color above the chosen threshold.
- We identify text blocks as blocks containing a defined amount of black pixels. If more than 70% of pages text blocks are inside tampered cluster, meaning one cluster is made out of white background and another of text blocks. We classify an image as free of noise tampered regions.
- We merge neighbouring tampered blocks into regions, we then disregard any isolated small tampered regions left.
- If the number of tampered regions is above a specified number, we classify an image as free of noise tampered regions.

The final selection is then converted to `noise tampered region` object, which is later serialized, containing page id and bounding box.

5.7 Serialization/Deserialization

To convert extracted information into protobuf representation, we first need to specify message structure inside `.proto` file. The full specified message can be seen in the Appendix B. The important thing is that page features and document features are parsed separately (see 5.1). Each page of the PDF document is represented by one page object. These page objects are then grouped into a list, which is then field (`pages`) of a PDF document python object, which is serialized. The structure of specified message inside `.proto` file corresponds to the structure of PDF document python object.

5.7.1 Serialization

Each serialized class has its specified protobuf class counterpart. This protobuf counterpart was generated by protobuf python compiler. The class is then imported and each class field is serialized. If a field is of primitive value (string, bytes, int, double, float, boolean), the value is serialized right away. If a field is an object, we recursively repeat the same procedure until we get to primitive values. If a field is a collection of non-primitive values, we repeat the procedure for each item in the collection. Some types of field can't be serialized into protobuf, for instance, NumPy's `nd.array`, `to_proto` function is then specified. If class has function `to_proto` defined, the function is called instead of recursively repeating above procedure. In the example of `nd.array` `to_proto` functions saves the `nd.array` as 1D list with additional information on array's original width and height so `nd.array` can be correctly loaded (deserialized) later.

5.7.2 Deserialization

As for serialization, inverse mapping is used to determine python class counterpart for protobuf class. Since speed is of the essence, as the speed of loading from generated protobuf files is later on compared to the speed of PDFminer library, `from_proto` function is defined for each serialized/deserialized class. Deserialization function then consists only of importing target class and then invoking `from_proto` function with the input of the function being its protobuf counterpart.

Chapter 6
Implementation

In this chapter, we introduce projects most used libraries and introduce project structure and where you can locate the most important files.

6.1 Most Used Libraries

The most important library is PDFminer.six (mentioned in 3.3). On top of PDFminer is part of Resistant.AI codebase, which the author is part of. The Resistan.AI[1] codebase takes output of PDFMiner layout analysis and transforms it into its own parsed python objects with additional functions.

All mathematical and array operations are done using a python NumPy library[2] and all image operations are handled using libraries Pillow[3] and OpenCV[4].

6.2 Project Structure

The root project package is called `diploma` and inside are files named `miner_*.py` (main one called just `miner.py`) used for intial parsing (layout analysis) and converting of raw PDF files into python objects (Resistan.AI) using PDFminer library, and six subpackages:

- `structure` - Package contains all dataclasses representing PDF in various processing states. A `catalog.py` contains definition of catalog calling other functions to parse desired data (metadata, fonts, etc.). A `core.py` contains definition of PDF document python object representation, in other words, final output of our application before serialization into protobuf.

- `extractors` - Package contains extraction of features that are not directly parsed from catalog. For instance `component_matrix.py` for creating

[1]https://resistant.ai/
[2]https://numpy.org/
[3]https://pillow.readthedocs.io/en/stable/
[4]https://opencv.org/

page represenation and `word_append_detection.py` for detection of word appends. Masking and converting of scanned pages into one or more images is handled by `image_extraction.py`.

- `core` - Package contains one file named `pdf_processor.py`. The class inside handles all logic of converting output of `miner.py` and extracting all additional informating which is then exported into objects defined in `core.py`.
- `proto` - Package contains all serialization and deserialization functionalities. `serde.py` contains serialization/deserialization logic and `sample.proto` contains protobuf defined message.
- `tests` - Package contains various PDF files used in tests.
- `utils` - Package contains all utility functions ranging from date parsing located inside `time_utils.py` to input functions of application (load, extract, serialize) located inside `file_utils.py` file.

Each package, apart from `tests` and `utils` packages, has its own `test` subfolder where pytests, ensuring functionality of this package, are located.

Chapter 7
Evaluation

In this chapter, we firstly evaluate modification features detectors described in Chapters 4 and 5 on labeled and real-world data. We start by describing evaluation metrics and datasets used for evaluation. Then, we use them to evaluate modification features detectors.
Secondly, we conduct PDF loading performance comparison between loading from generated Protobuf files and loading PDF files using PDFminer library. At the end, we explain results and describe the advantages of our solution.

7.1 Modification Detection Evaluation

In this section, we evaluate the functionality of implemented modification features and explain their strengths and weaknesses.

7.1.1 Evaluation Metrics

In binary classification problem (document is modified or document is not modified), there are four possible outcomes:

- If instance is positive (modified) and it is classified as positive, it is called *True Positive* (TP).
- If instance is negative (unmodified) and it is classified as negative, it is called *True Negative* (TN).
- If instance is negative and it is classified as positive, it is called *False Positive* (FP).
- If instance is positive and it is classified as negative, it is called *False Negative* (FN).

To further visualize performance of classification algorithm, we use confusion matrix. The structure of confusion matrix can be seen in Figure 7.1.

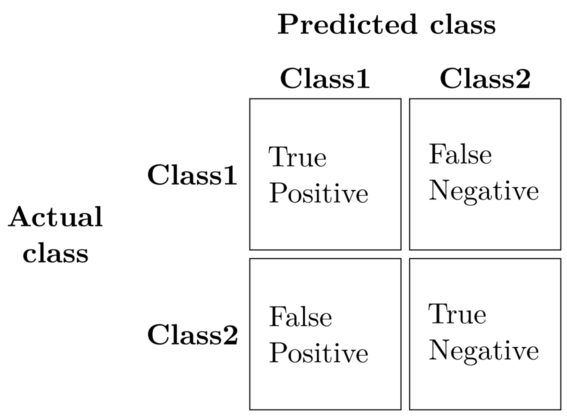


Figure 7.1: Structure of confusion matrix

One way to evaluate the classification algorithm is to count positive hits ($TP + TN$) against all processed files, this metric is called *accuracy*. Unfortunately, accuracy can be a misleading metric for imbalanced datasets, but other metrics do not suffer from this problem. Namely *recall* and *precision* metrics will be used. In the context of PDF document modification detection, recall, also called *True Positive Rate* (TPR), with perfect score of 1.0 means that all modified PDF documents are detected, but tells nothing about how many unmodified documents are misclassified. Whereas precision score of 1.0 means that every PDF document classified as modified is indeed modified but tells nothing about the portion of detected modified documents. *False Positive Rate* (FPR) measures false positive detections against all negative instances.

$$\begin{aligned} accuracy &= \frac{TPs + TNs}{TPs + TNs + FPs + FNs}, recall = \frac{TPs}{TPs + FNs}, \\ precision &= \frac{TPs}{TPs + FPs}, FPR = \frac{FPs}{FPs + TNs} \end{aligned} \tag{7.1}$$

7.1.2 Datasets

We use 2 datasets, first (`SET1`) is a labeled dataset where frauds are manually created to reflect possible types of modification on real-word data, and second dataset (`SET2`) is the first one but enriched by documents from publicly available sources (ulozto[1], sribd[2], etc.). Documents from publicly available sources are presumed to be genuine, but we have no guarantee hence the two datasets used. Both datasets consist mainly of invoices that are the easiest to find. Some bank statements and payslips are also present but in a minority. These types of financial documents are prime targets of fraud modification. More detailed characteristics of both datasets follow:

[1]https://ulozto.cz/
[2]https://www.scribd.com/

name	SET1	SET2
number of files	186	1088
% of scanned documents	16%	12%
number of modified documents	118	118
number of legitimate documents	68	970

Table 7.1: Characteristics of datasets used for experimental evaluation

In table 7.1 you can see that the difference between `SET1` and `SET2` is that in `SET2` there is 902 documents added from ulozto. The ratio of legitimate documents to modified documents in `SET2` more matches the real world distribution of documents, where the majority of documents are legitimate.

7.1.3 Validation

We selected 13 modification features (`has_word_overlaps`, `has_word_appends`, `has_Windows_and_Mac_fonts`, `has_noise_tampered_regions`, `is_modified_with_Photoshop/Affinity/Gimp`, `has_signature_invalid_byte_range`, `is_created_by_Sejda/PDFPro/PDFfiller/PDFExpert/PDFGuru`) as classification elements. The classification was rather strict, we classified the document as modified if any one of the listed features was found present in the document. For instance, a document is classified as modified if there is any word overlap present, we do not examine if there is 1 or 20 word overlaps present. This could be added in the future to further specify how strict should the classification algorithm be. Also note that we did not include `is_modified_per_modify_date` feature, because it turned out that it is a common occurrence in template files to have different time of creation and modification.

In Figure 7.2, you can see confusion matrix and derived metrics for `SET1`.

$$cm = \begin{bmatrix} 101 & 17 \\ 5 & 63 \end{bmatrix} \rightarrow \begin{aligned} recall &= 0.86 \\ precision &= 0.95 \\ FPR &= 0.074 \end{aligned} \tag{7.2}$$

A precision score of 0.95 is very good and tells that almost all documents classified as modified are indeed modified. Meanwhile, recall score of 0.86 tells us that we detected 86% of modified documents from the dataset. These are good results, but this validation was done only on set of 118 labelled files. Next, we are going to examine results for a set with real-word data (`SET2`).

Confusion matrix and derived metrics for `SET2` can be seen in Figure 7.3.

$$cm = \begin{bmatrix} 101 & 17 \\ 41 & 929 \end{bmatrix} \rightarrow \begin{aligned} recall &= 0.86 \\ precision &= 0.71 \\ FPR &= 0.042 \end{aligned} \tag{7.3}$$

Recall stayed the same because SET2 is SET1 enriched by real-word data that we presume as legitimate, whereas precision score is lower at 0.71. Good thing is that FPR is lower for SET2 at 0.042. The undetected documents (FNs) were mostly PDF documents that were flattened (converted to images) after an edit or somehow edited by editor without leaving a trace in document's metadata. Further, we are going to examine the cause of FPs and look at best performing classifiers (features).

False Positives

Since we are interested in a minimal number of FPs, we examined these files and looked at what features caused the most FP hits. All examination was done on data from SET2. You can see a bar graph of feature FP hits in Figure 7.2.

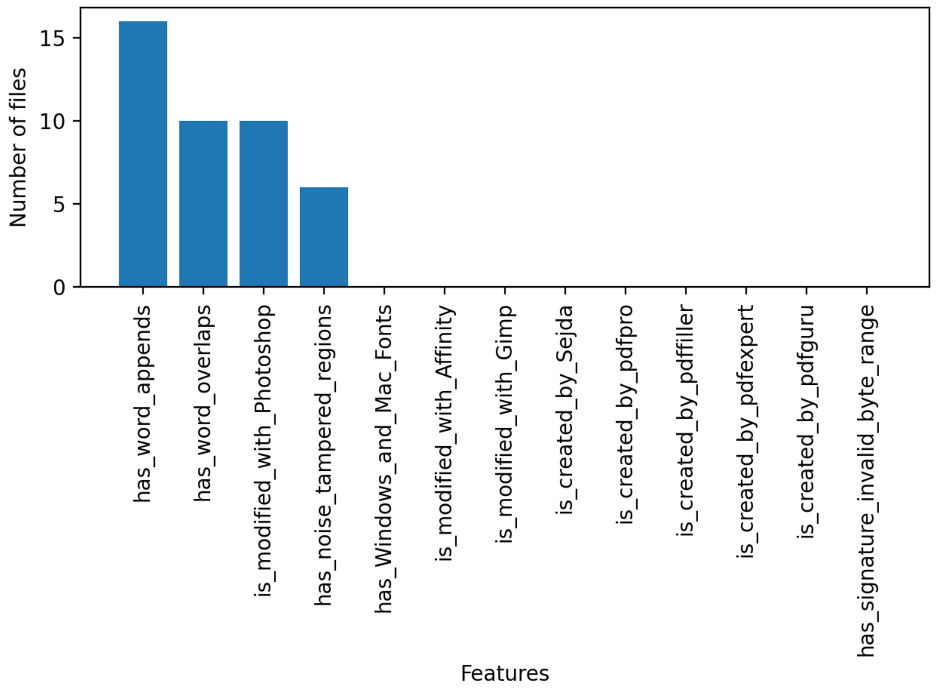


Figure 7.2: # of false positive hits for each classifier (modification feature)

Upon examination, we noticed that most of false positive feature hits are the same features for the same type of file (type of invoice). Meaning that, for instance, detected word append for Vodafone invoice is repeating in the same place for every Vodafone invoice. A more detailed description follows:

- Word appends - Out of 16 detected samples with word appends present, we identified 2 types of templates, that yielded 12 files, where word appends findings repeated. The appends were caused by large sequence id distance and a small actual distance between value and its unit, caused by a strange rendering of the PDF. You can see an example in Figure 7.3.

The word append detected on image 7.3 is by definition correct because the value and unit seem as one word but in reality, it is not a case of fraudulent modification of a file.

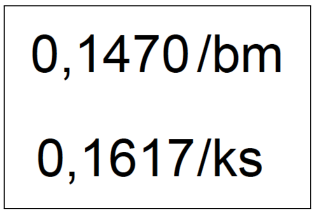
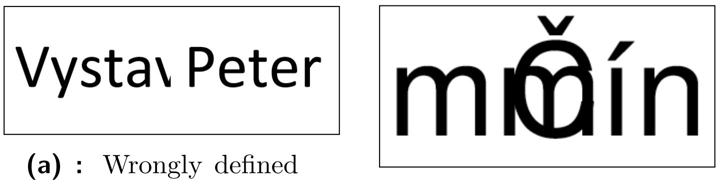

0,1470/bm
0,1617/ks

Figure 7.3: Value and its unit without space between words that triggered word append detection feature

Rest of word appends were caused by incorrectly determining if two objects are part of one word or not.

- Word overlaps - Out of 10 detected samples with word overlaps present, we identified 2 types of documents that in total yielded 6 files where we detected word overlap correctly by word overlap definition (text is covered by something else), but in reality, it is not a case of fraudulent modification. Both are caused by wrongly filled templates. You can see both findings in Figure 7.4.


Vysta\ Peter
mǒín

(a) : Wrongly defined template **(b) :** Overflowed text

Figure 7.4: Example of by definition correctly word overlaps

The remaining 4 samples are of the same type and are caused by rectangle and text objects to create dots.

- Modified with Photoshop - All 10 finding are invoices of the same type. Seem like the issuer uses photoshop to create PDF, which is strange. The detections of Photoshop were correct.
- Noise tampered regions - All FPs are caused by highly textured regions, where the noise estimator can perceive thin lines as noise (5.6.1).

If we disregarded correctly classified modification features or counted the same finding on 10 files from the same issuer as one finding, our precision score would be higher, specifically 0.84 if we were to completely disregard these findings.

True Positives

In Figure 7.5 you can see TP hits for each classifier on `SET2`.

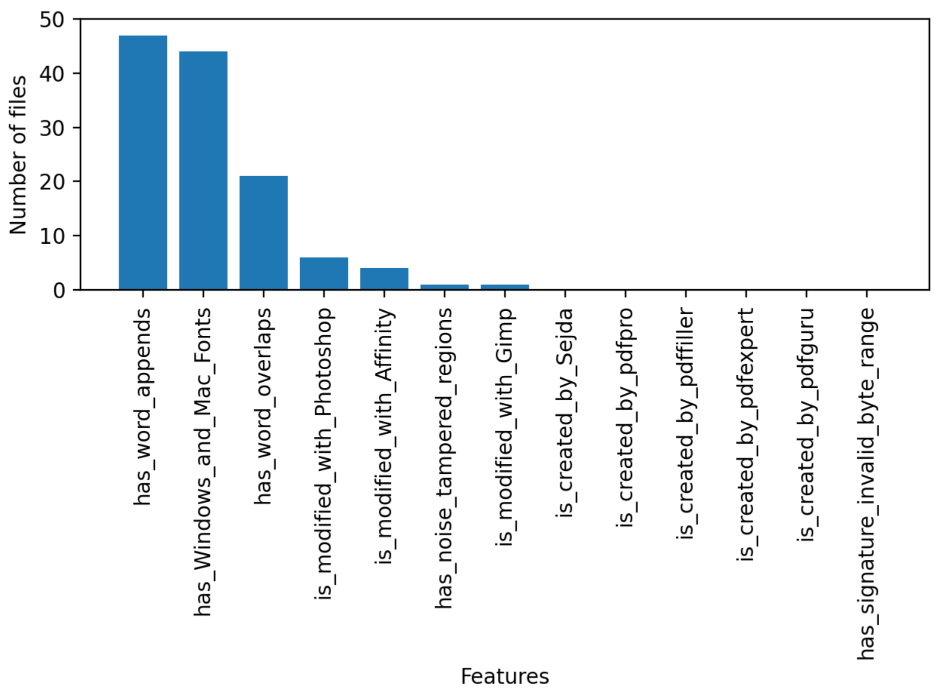


Figure 7.5: # of true positive hits for each classifier (modification feature)

Note that some files can contain modifications that trigger more that one modification feature, but most of the files triggered only one.

Comparison of Classifiers

From graphs 7.2 and 7.5, we can see that while `has_word_appends` scores the most true positives, it also scores the most false positives, so his error rate is high, but TPs outweigh a lesser count of FPs. The best performing modification feature turns out to be `has_Windows_and_Mac_fonts` while detecting the second highest number of TPs it maintained zero counts of FPs making it the perfect modification classifier. Problem is that unlike word overlaps or word appends and other modification features, it depends on modification being made on different operating system than on which was PDF document originally created. A fraudster may own computer with the same operating system as the issuer and the modification would not be detected (if other detection features would not detect it either). The worst performing classifier in terms of the ratio between TPs and FPs is `has_noise_tampered_regions` classifier. Improvement of noise estimate technique is needed to combat high textured regions in files.

7.2 PDF Loading Benchmark

We compare performances (time) of PDF loading for future analysis of the three following use cases:

1. Loading a raw PDF using a PDFminer library. The result is loaded PDF with layout analysis done on each page.
2. Loading raw PDF using PDFminer library and then extracting its features. The result is python PDF object with all features extracted (simple, structure, layout, origin, modification), ready to be serialized.
3. Loading PDF from serialized Protobuf object. The result is python object with all features extracted.

7.2.1 Method

As layout analysis and most of the extracted features are done per page the amount of extracted information increases with the document's number of pages. The assumption is that load duration time is increasing linearly with the number of document's pages. In addition, we are also interested if there is any difference between the performances of digital and scanned documents.

To verify above assumption we mixed documents from ulozto with scientific papers from CSVPR[3] and split them into folders by the number of pages. We then took random 10 documents (10 digital, 10 scanned) from each folder to conduct benchmark on. We then converted these files into protobufs using our application so we can measure performance of loading time from Protobuf. The evaluation was carried out 4 times for each folder, with the first time disregarded as warm-up period. The mean time was then calculated from 3 remaining times. The mean time was then divided by a number of documents in the folder to get average time for a document.

Benchmark was conducted on MacBook Pro 2017, with 2.3 GHz Dual-Core Intel Core i5 processor and 8 GB 2133 MHz memory.

[3]https://www.kaggle.com/paultimothymooney/cvpr-2019-papers

7.2.2 Results

In Figure 7.6 you can see the comparison of three mentioned use cases. On the left, you can see a comparison for digital documents and on the right, you can see the comparison for scanned documents. All functions have a linear trend.

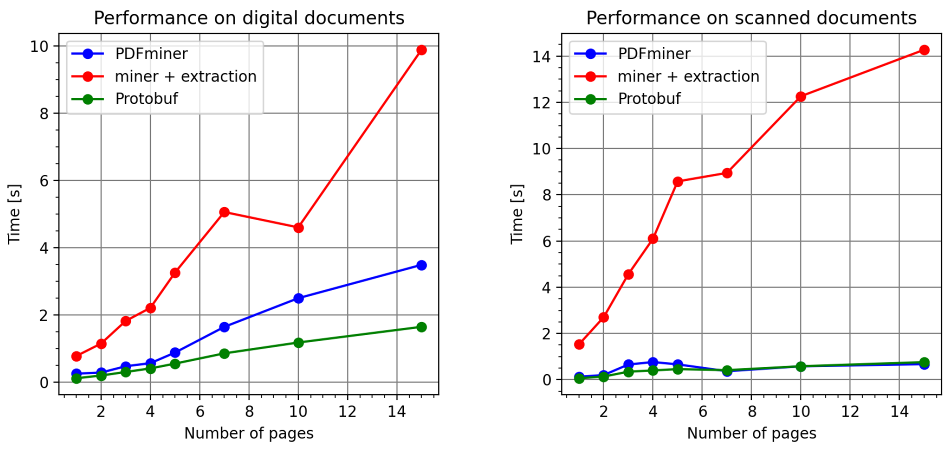


Figure 7.6: Comparsion of PDF loading times

The fastest method is loading from Protobuf file averaging 0.111s for digital page and 0.077s for a scanned page. Second is loading using PDFminer averaging 0.2s for digital page and 0.117 for scanned page, keep in mind that this method only loads and parses PDF but do not perform any additional information extraction. The slowest is as expected loading using PDFminer and subsequent information extraction averaging 0.638 for digital page and 1.406 for scanned page. The spike for digital documents with 7 pages can be contributed to the composition of files in the source folder. The more information PDF contains the longer it loads and the longer duration of additional information extraction. Summarized result can be seen in table 7.2.

Page type	PDFminer	Extraction	Protobuf
digital[s]	0.2	0.638	0.111
scanned[s]	0.117	1.406	0.077

Table 7.2: Summary of average loading times per PDF page

In Figure 7.7, you can see detailed comparison of digital versus scanned files for each method.

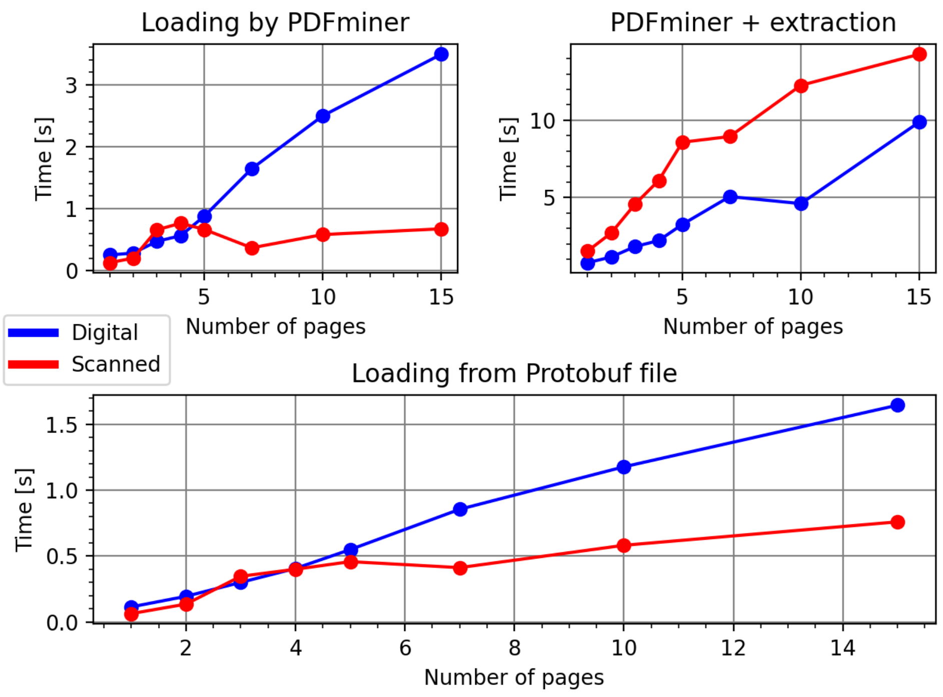


Figure 7.7: Comparison of loading digital versus scanned documents for each method

You can see that loading time for scanned documents is lower for PDFminer and Protobuf, but it is higher for the extraction of information. Lower loading times are caused by scanned documents or pages having fewer components than the digital ones (5.5), whereas information extraction time is higher for scanned documents because of image operations that are slow. Mainly masking of images (transparency), composing images into 1 image and additional noise features.

Advantages

Storing the result of preprocessing for future analysis can save a lot of time in the development pipeline. Extracted features might be used to train a machine learning model. A model might be trained multiple times or retrained on new data. Without any storing of preprocessing results in place, each time the model needs to be retrained, the whole process of extracting information and parsing needs to be redone. In table 7.3 you can see calculated durations of data preprocessing using various methods. Durations were calculated for a dataset with 1000 pages and with various digital to scanned documents ratios. Only PDFminer with subsequent information extraction and loading from generated Protobuf files were compared as they achieve the same output.

Method	100% digital	75%:25%	50%:50%	25%:75%	100% scanned
Extraction[s]	638	830	1022	1214	1406
Protobuf[s]	111	102.5	94	85.5	77

Table 7.3: Example preprocessing time method comparison for dataset with 1000 pages

You can see that storing the results of preprocessing can speed up the data preparation part of the training cycle up to 18 times for dataset made solely out of scanned documents.

Chapter 8

Conclusion

The main goal of this thesis was to create an application that takes PDF file, parses it, extracts relevant information from it and stores it in anonymized state. In Section 1.2, we defined 2 additional goals. List of all goals with description of how we fulfilled these goals follows:

1. **Implement application that takes a PDF file and converts it into an anonymized chosen format with extracted information.** We reviewed 4 serialization formats, compared their speed, file size and quality of documentation and chose Protocol Buffers as our serialization format. We then defined extracted features and explained how we sort them into categories according to their added information value, we also in detail explained why and how we extract and represent those features. We then described how the anonymization is done and which information is anonymized and which is not. Finally, we explained how serialization and deserialization are done and provided a full Protobuf message definition representing a processed PDF.

2. **Implement several PDF document modification detectors.** We first introduced the types of PDF forgery and showed examples of the most common modifications. We then listed 25 modification features (counting detectors for specific software) and in detail explained extraction and representation of the most important ones. We also introduced a page representation that is used to help extract these features. From 25 listed features we selected 13 features whose presence classifies document as modified. We evaluated them on 2 datasets, one labelled and second enriched by real-world data. We achieved recall 0.86, precision 0.95 and FPR 0.074 for first dataset and recall 0.86, precision 0.71 and FPR 0.042 for second dataset. We manually examined false positive samples and revealed that most of our findings were correct by definition, but files themselves were not fraudulent. Without these samples classified as FPs, our precision score increased to 0.84 for the second dataset. We then compared modification classifiers according to their TP to FP ratio and concluded that word overlaps, word appends and fonts from both OS are our best performing modification classifier. On the contrary, noise variance detection did not meet our expectations.

3. **Speed up the process of analyzing PDF files.**
   We performed a comparison of PDF loading time from parsed Protobuf file, using PDFminer library and using PDFminer library with subsequent information extraction. We performed this evaluation separately on digital and scanned documents and found out that the scanned documents are faster to load but slower to extract features from. We gave reasoning as to why and concluded that loading time of PDF grows linearly with the number of document's pages. Lastly, we demonstrated that by storing the results of PDF preprocessing we can speed up the data preparation part of the training cycle up to 18 times.

The resulting application is able to efficiently parse and extract relevant features from PDF files. This information is then stored for future use. Based on our evaluation, we can say that we reliably detect various types of PDF modification.

In future work, we suggest training an anomaly detection model to suppress recurring findings on documents from the same issuer and creating a user interface to visualize possible findings.

Appendix A

Bibliography

[1] “File Sizes,” [Accessed: 08-04-2020]. [Online]. Available: http://labs.criteo.com/wp-content/uploads/2017/05/File-Sizes.bmp

[2] “Serialization/Deserialization speed,” [Accessed: 08-04-2020]. [Online]. Available: http://labs.criteo.com/wp-content/uploads/2017/05/Table-3-Small-objects-serialization-time-in-micro-seconds.bmp

[3] “Text group margins,” [Accessed: 02-01-2020]. [Online]. Available: https://pdfminer-docs.readthedocs.io/_images/char-line-word_margin.png

[4] “Layout Analysis,” [Accessed: 02-01-2020]. [Online]. Available: https://pdfminersix.readthedocs.io/en/latest/_images/layout_analysis_output.png

[5] X. Pan, X. Zhang, and S. Lyu, “Exposing image forgery with blind noise estimation,” *MM and Sec’11 - Proceedings of the 2011 ACM SIGMM Multimedia and Security Workshop*, 09 2011.

[6] Duff Johnson, “PDF statistics – the universe of electronic documents,” 2018, [Accessed: 08-05-2020]. [Online]. Available: https://www.pdfa.org/wp-content/uploads/2018/06/1330_Johnson.pdf

[7] Angelique Ruzicka - Thisismoney.co.uk, “Business invoice scams rise, here is how to stay safe,” 2019, [Accessed: 08-05-2020]. [Online]. Available: https://www.thisismoney.co.uk/money/smallbusiness/article-6804459/Business-invoice-scams-rise-heres-stay-safe.html

[8] Adobe Systems Incorporated, “Pdf Reference - Sixth edition, Version 1.7,” November 2006.

[9] Evermap, “Understanding PDF File Size,” [Accessed: 09-04-2020]. [Online]. Available: https://www.evermap.com/PDFFileSize.asp

[10] Google, “Protocol Buffers,” [Accessed: 26-12-2019]. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview

[11] Apache, “Apache Thrift,” [Accessed: 08-04-2020]. [Online]. Available: https://thrift.apache.org/

[12] CriteoLabs, "Data Serialization Comparison," [Accessed: 08-04-2020]. [Online]. Available: https://labs.criteo.com/2017/05/serialization/

[13] B. Wolford, "What Is Gdpr?" [Accessed: 27-12-2019]. [Online]. Available: https://gdpr.eu/what-is-gdpr/

[14] Council of European Union, "Council regulation (EU) no 2016/697," 2016, [Accessed: 27-12-2019]. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1501688126470&uri=CELEX:32016R0679

[15] PDFMiner.six, "Converting a pdf file to text," [Accessed: 02-01-2020]. [Online]. Available: https://pdfminersix.readthedocs.io/en/latest/topics/converting_pdf_to_text.html#layout-analysis-algorithm

[16] Fujitsu, [Accessed: 24-04-2020]. [Online]. Available: https://www.fujitsu.com/global/support/products/computing/peripheral/scanners/scansnap/sample/sv600.html

[17] eforms, [Accessed: 24-04-2020]. [Online]. Available: https://eforms.com/invoice-template/monthly-rent/

[18] J. Immerkær, "Fast noise variance estimation," *Computer Vision and Image Understanding, Vol. 64, No. 2*, pp. 300–302, 09 1996.

Appendix B

Protobuf Message Definition

```
message PdfSampleRaw {
    message BoundingBox {
        float x = 1;
        float y = 2;
        float width = 3;
        float height = 4;
    }
    message Page {
        message Overlap {
            uint32 page = 1;
            string hidden_text = 2;
            BoundingBox bbox = 3;
            string replacement_text = 4;
        }
        message Append {
            uint32 page = 1;
            BoundingBox bbox = 2;
            string first_word = 3;
            string second_word = 4;
            uint32 id_distance = 5;
        }
        message NoiseTamperedRegion {
            BoundingBox bbox = 1;
            uint32 page_id = 2;
        }
        message Image {
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            string name = 4;
            uint32 src_width = 5;
            uint32 src_height = 6;
            BoolValue image_mask = 7;
            uint32 bits = 8;
            repeated string color_space = 9;
            repeated float matrix = 10;
            sint64 stream_length = 11;
            uint32 stream_obj_id = 12;
        }
        message Curve {
            message Point {
                float x = 1;
                float y = 2;
            }
            message Color {
                enum ColorType {
                    FLOAT = 0;
                    STRING = 1;
                }
                ColorType type = 1;
                string value = 2;
            }
            enum CurveType {
                CURVE = 0;
                LINE = 1;
                RECTANGLE = 2;
            }
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            CurveType curve_type = 4;
            bool even_odd = 5;
            bool fill = 6;
            float line_width = 7;
            bool stroke = 8;
            repeated Color n_stroking_c = 9;
            repeated Color stroking_c = 10;
            repeated Point points = 11;
        }
        message Char {
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            string char = 4;
            float adv = 5;
            repeated float matrix = 6;
        }
        message Text {
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            string text = 4;
            string font_id = 5;
            uint32 render_style = 6;
            bool upright = 7;
            StringValue cs_name = 8;
            UInt32Value cs_components = 9;
            repeated Char chars = 10;
        }
        message MarkedContent {
            enum Type {
                POINT = 0;
                SEQUENCE = 1;
            }
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            string name = 4;
            Type type = 5;
            map<string, string> properties = 6;
            repeated Text tagged_text = 7;
        }
        message Annotation {
            uint32 page = 1;
            uint32 seq_id = 2;
            BoundingBox bbox = 3;
            StringValue subtype = 4;
            StringValue text = 5;
        }
        message CompArray {
            repeated int64 matrix = 1;
            int64 height = 2;
            int64 width = 3;

        }
        message ModifiedTextRegion {
            BoundingBox bbox = 1;
            uint32 page = 2;
            string text = 3;
        }
        uint32 page_id = 1;
        uint32 obj_id = 2;
        BoundingBox layout_bbox = 3;
        bool is_scanned = 4;
        bool is_scanned_with_ocr = 5;
        repeated Annotation annotations = 6;
        repeated Append word_appends = 7;
        repeated Overlap word_overlaps = 8;
        repeated Image images = 9;
        repeated Curve curves = 10;
        repeated Curve lines = 11;
        repeated Curve rectangles = 12;
        repeated Text texts = 13;
        uint32 length_of_texts = 14;
        map<string, uint32> char_histogram = 15;
        repeated MarkedContent marked_contents = 16;
        CompArray component_matrix = 17;
        repeated NoiseTamperedRegion noise_tampered_regions = 18;
        google.protobuf.FloatValue noise = 19;
        repeated ModifiedTextRegion modified_text_regions = 20;
    }
    message StreamOperations {
        message StreamOperation {
            message StreamOperand {
                string type = 1;
                string value = 2;
            }
            string operator = 1;
            repeated StreamOperand operands = 2;
        }
        uint32 obj_id = 1;
        repeated StreamOperation operations = 2;
    }
    message StreamWhitespaces {
        uint32 obj_id = 1;
        repeated string whitespaces = 2;
    }
    message ObjPos {
        uint32 byte_pos = 1;
        uint32 obj_id = 2;
        uint32 gen_no = 3;
    }
    message PdfCatalog {
        message Xrefs {
            message Xref {
                uint32 obj_id = 1;
                uint32 gen = 2;
                uint32 pos = 3;
                UInt32Value stream_id = 4;
            }
            string pdf_miner_type = 1;
            repeated Xref xrefs = 2;
            string json_trailer = 3;
        }
        map<uint32, string> json_catalog = 1;
        repeated Xrefs xrefs = 2;
    }
    string sha256_name = 1;
    uint64 file_length = 2;
    repeated bytes pdf_comment_lines = 3;
    repeated Page pages = 4;
    repeated StreamOperations stream_operations = 5;
    repeated StreamWhitespaces stream_whitespaces = 6;
    repeated ObjPos obj_positions = 7;
    PdfCatalog pdf_catalog = 8;
}
```

Appendix C
CD Contents

Due to the privacy concerns of datasets (invoices, payslips, bank statements), the data that was used in this thesis is not present on the CD. However, anonymized text files, which were later used to generate statistics are present. Also, some modified example files, used for tests, are present inside the application folder.

The CD has a following structure:

```
CD
├── Diploma
│   ├── diploma
│   ├── diploma.py
│   └── pyproject.toml
├── data
└── thesis
    ├── src
    └── pdf_document_representation_for_automated_analysis_vaca_jakub.pdf
```

- Diploma→diploma - Contains source codes of an application.
- Diploma→`diploma.py` - A command line script used to interact with the application.
- Diploma→`pyproject.toml` - A project requirements file.
- data - Contains anonymized text files with result sfor each dataset folder (legit, fraud, ulozto) where each row correnspoonds to one file and where each column boolean value corresponds whether modification feature is detected inside of a corresponding file.

  ```
  is_modified overlaps appends OS_fonts noise_reg Photoshop Gimp Affinity Sejda PdfPro PDFfiller PDFexpert PDFGuru Signature
  ```

- thesis→src - Contains LaTeX source files for this thesis, figures used within the thesis are also enclosed.
- thesis→src - Contains a PDF version of this thesis.

Appendix D
Usage Manual

First, Python 3.7 and Poetry dependency manager is needed. If you don't have Poetry installed follow instructions at https://python-poetry.org/docs/#installation.

Navigate into `Diploma` folder and from command line run:

```
poetry install
```

If any package or packages could not be found, try installing these packages with pip or preferably pip3. For instance package `cv2` is installed by:

```
pip install opencv-python
```

All inputs are handled by `diploma.py`. After successful installation of packages you can run the command:

```
poetry run python diploma.py --input_path={path_to_input_folder}
--output_path={path_to_output_folder}
```

where `input_path` is a path to the folder with raw PDF files and `output_path` is a path to the folder where resulting protobufs files will be stored.

Only files with `.pdf` suffix are processed.