Extracting scientific results from research articles

Lucas Pluvinage

Master IASD

Internship report

Extracting scientific results from research articles

Lucas Pluvinage

Advised by Pierre Senellart,
École Normale Supérieure de Paris

April 20, 2020 — September 18, 2020

Abstract

From April to September I've been doing my research internship with Pierre Senellart in École Normale Supérieure computer science department's *VALDA* team[1]. My goal is to develop and test methods to perform information extraction from PDF articles, in particular focusing on theoretical statements such as theorems or definitions. We tackle the topic of data management and extraction, performing research on hierarchical feature extraction, conditional random fields for document segmentation, software engineering for accelerated dataset creation, and leveraging deep-learning methods for information extraction. While no groundbreaking state-of-the-art results have been obtained here, this internship results in flexible reusable software and promising methods paving the way for future research. The software is available on Github, in the following repository: github.com/PierreSenellart/theoremkb

Acknowledgements

I would like to thank my advisor Pierre Senellart for his guidance throughout this internship, in spite of these hectic times. I'm also thanking Théo Delemazure with whom I collaborated on the project. I'm grateful for the hospitality of my friends Camille Gobert and Leila Slaoui who hosted me and helped me bear the COVID-19 lockdown in better conditions than in my little parisian appartment. At last, I'm thanking Maëlle Puéchoultres who has always been there, even in my hard times.

Contents

[1] team.inria.fr/valda

1 Introduction

TheoremKB [1] is a project led by Pierre Senellart whose goal is to build semantic knowledge from a collection of scientific articles in the PDF format. One of the key goals of TheoremKB is to be able to compute the graph of theoretical results from a given research topic. In this graph, an edge is drawn from a result A to result B when B is used in the proof of A. Having a finer resolution than the citation graph, as nodes are individual results instead of whole documents, this graph should allow deducing interesting facts on scientific research, such as whether we can find proof cycles among papers (which would be possible as some authors cite not yet published content), or what results are impacted if a proof is found to be incorrect. The project aims at making bibliographical work easier by having a deeper understanding of how papers are related to each other.

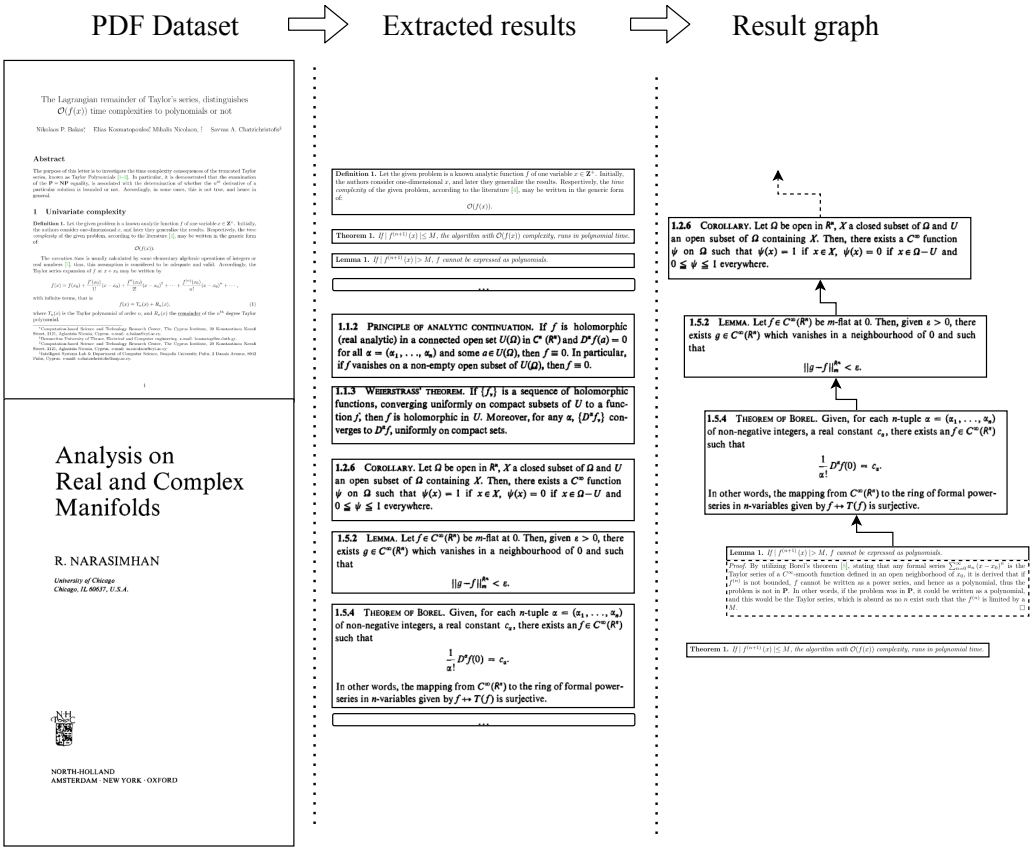

Figure 1: Two steps for result graph extraction in TheoremKB.

My co-intern Théo Delemazure and I were very interested in this project, so we decided to focus on two initial tasks to advance on this goal. The tasks are the following:

- Given a PDF article, identify the list of theoretical results along with their proofs. Results include theorems, lemmas, corollaries, definitions. Identifying the results is necessary to assess the contributions of an article before matching them across articles. The method will probably need to take into account visual features as that is how results and proofs are mostly distinguished from the rest of the paper.
- Given a corpus of articles, build the result graph by identifying when a result is used in a proof. It is a challenging task as there is no standard way of mentioning content from another article. For example, when an author uses a theorem proved in another publication, they might cite the whole paper without referring in particular to the theorem used.

My internship focuses on the first task. While the second task depends on the results of the first one, Théo was able to work in parallel by using a synthetic corpus built from LaTeX sources. At last, the following constraints need to be taken into account:

- The technique have to work on PDF documents, as it is the main form of publication of scientific papers. That enables using huge open access databases to build an exhaustive corpus, which is needed to properly construct the results graph.
- There is no publicly available dataset that can be used to compare our results. Most research article datasets are made for metadata extraction tasks and are not focused on theoretical results. Therefore,

we will need to create our own dataset which should have as less bias as possible. The creation of this dataset should be semi-automatic: bulk of the work being done automatically but it should be possible to fine-tune it manually.

- Being preliminary research for an important project, the work should be re-usable and extensible.

1.1 State of the art

The field of knowledge understanding has not yet tackled the problem of result extraction from PDF documents. However, similar approaches have been found and experimented with, they serve as inspirations for our system. As a starting point, *CiteSeerX* introduces on this page software that performs some kind of information extraction from publications: csxstatic.ist.psu.edu/downloads/software

1.1.1 PDF Extraction tools

In any case, the first step is to extract PDF data into a machine-readable format while preserving layout information. A solution can be to render PDFs, for example using the PyMuPDF library [2]: by rendering the PDF no visual information is lost, at the expense of having no parsed semantic content. If the document layout is not needed, there are tools to extract the raw textual content of the PDF. Being text-only, it is particularly suitable as an input for natural language processing frameworks.

Balancing the trade-off between semantics and visual features, a number of tools were created to transform the PDF into an XML file. The XML format captures a reading order of the PDF and may be able to embed visual features such as token position and how the text is displayed. The 4 following software were tested before settling on PDFAlto.

- PDFMiner: github.com/euske/pdfminer
- PDFtoXML: github.com/kermitt2/pdf2xml
- PDFAlto: github.com/kermitt2/pdfalto
- PDFBox: pdfbox.apache.org

PDFAlto outputs an XML under the ALTO format (Analysed Layout and Text Object)[2] which has the advantage of keeping a lot of layout information. The PDF is described as a set of pages, cut in blocks that are composed of lines made of textual tokens and spaces. Each node has a position and size, and text can be styled with a font. The tool is also able to extract PDF annotations and images. The following nodes and properties are particularly useful:

- `<TextStyle>`: Font used – ID, font family, font size, font color
- `<Page>`: Document page – Number, dimensions
- `<TextBlock>`: Textual block, such as a paragraph – Position, dimensions
- `<TextLine>`: Textual line – Position, dimensions
- `<String>`: Textual token – Position, dimensions, ID of style used, content
- `<SP>`: Space – Position, width

1.1.2 TheoremKB preliminary project

In the setting of a small research project, Daria Pchelina (ENS) explored some possibilities on how to accomplish our goal. The first promising idea has been to use LaTeX sources to automatically build an annotated dataset of articles, as there is no dataset available to benchmark the results. This can be seen as problematic because the performance of tested methods directly depends on the quality of the LaTeX extraction step, which consists of injecting a LaTeX plugin in a corpus of LaTeX sources.

PDFMiner[3] is used to parse the PDF content into an XML file and a matching algorithm is performed to identify the results and build the training target. A relatively small number of features were handcrafted and 3 methods were compared: Bayes, conditional random fields, and a naive algorithm. Tested on a corpus of 600 documents downloaded from arXiv, the results are not very good because of the lack of features but they set a baseline for work presented in this internship. The code developed as part of this project can be found on Github[4].

[2] `https://www.loc.gov/standards/alto/`
[3] pdfminersix.readthedocs.io
[4] github.com/tooticki/IE_project

1.1.3 Cascading-style extractors

NaDevEx In the field of nanoscale research, NaDevEx [3] – Nanocrystal Device Automatic Information Extraction Framework – is a software developed to extract domain-specific information. The project aims to extract textual information sorted in 8 categories using a cascade of conditional random fields. In a cascading-style information extractor, the model output can be fed in another model. A dataset has been developed for this task, comprising 392 manually annotated sentences and totaling 2870 terms.

The key ideas in this project are the usage of multiple models in cascade, and the features they used. Linguistic features such as part-of-speech tags and domain-specific vocabularies seem to be useful for the accomplishment of this task.

CERMINE CERMINE [4] – Content ExtRactor and MINEr – is a software that extracts article metadata, bibliography, and outline. It's an open-source project written in Java that aims at flexibility. Content classification is done in hierarchical steps using support vector machines and conditional random fields, based on a set of 97 features. They are geometric as well as lexical, related to formatting or based on heuristics. It produces a file in the JATS (Journal Article Tag Suite) format. The project is benchmarked on the GROTOAP2 dataset. This tool is interesting but has not been maintained since 2018.

Grobid Competing with CERMINE, Grobid [5] is an actively maintained open-source project led by Patrice Lopez written in Java. It is a production-ready tool for information extraction from PDF articles, that can extract the title, abstract, header metadata, find citations in the body, and parse the bibliography. It makes an extensive use of Wapiti [6] CRF library but it supports other sequence taggers such as bi-LSTM, through DeLFT [7] (deep learning framework for text). Grobid features a web interface, a REST API, and batch processing options.

1.2 Internship overview

The first months of the internship were more of an exploratory period, getting familiar with the project and the state of the art. One goal has been to create a training corpus from arXiv and reproduce Daria's results on it. I added some features and compared line-based and word-based tokenization in terms of CRF performance. When the results were satisfying as a baseline, my goal was to integrate the work in a general-purpose information extractor to take advantage of pre-built features and training procedures. Grobid was therefore the focus point: written in Java, already managing dozens of annotation models, adding my work as another annotation model seemed like a manageable task. Time has been spent reading about Grobid, digging in the code while figuring out if the project was a good candidate. But after several failed attempts at extending Grobid in a way that was satisfying to me, I decided to write my own, hopefully simpler, information extractor and manager. This was approximately two months since the beginning of the internship. The hope was not to have better performances than Grobid in any way but to have more extensible and flexible code, allowing fast iterations between updating models, features and evaluating it on training data. While building this software, I tested several models, improved the LaTeX extraction script and extended the task to perform document segmentation and title identification. More than seeking state-of-the-art results, the greatest challenge in this internship has been to properly design a machine learning workflow in a field where – in opposition to computer vision or natural language processing – the tools are not ready and there is no clear consensus on how things should be done.

2 Preliminary research

2.1 The LaTeX extraction script

As there is no publicly available dataset of PDF articles and their theoretical results, the first step in this project is to create one. Thankfully arXiv, a database of more than 1M open-access research papers, has an API to download articles PDFs and their sources. The source files that we are interested in are LaTeX sources because they have a lot of semantic content. LaTeX is a software and programming language designed to facilitate the writing of textual documents, letting the user focus on semantic content by taking care of formatting considerations. In the case of theoretical results, they are often signaled by appropriate environments such as `\begin{theorem}...\end{theorem}` to indicate LaTeX that a special formatting should be used there. These environments are defined using the `\newtheorem` command. This also means that one could use these markers to identify where are theorems are in a given paper. During Daria's project, a LaTeX plugin has been developed to extract all results from a given source.

It works by re-defining the `\newtheorem` command, adding a hook to output each result in its own page. As the plugin re-defines the `\newtheorem` command, it needs to be applied before the command is used but after its definition. That list of extracted results can then be used for machine learning purposes. However, this method is somewhat unwieldy because it can not be used to quickly compute if a word in the article is part of a result. To generate a sequence of labeled tokens, the PDF must be read through in order while sequentially matching content with the list of extracted results.

As an alternative to this ad-hoc algorithm, I suggested changing the LaTeX script to generate custom PDF links while keeping the original layout. The output of the LaTeX extraction step became a set of labeled bounding boxes highlighting the results of the article. Associated with a spatial index, we can now perform a fast lookup of each token's kind: result or not.

This method was applied to 6000 articles downloaded from arXiv in the field of complexity theory. Some losses are encountered: for example, 258 papers are not written in Latex. Compilation errors were encountered when the extraction script was included in the wrong place (for example after results environments have been defined). And even if there is a de facto standard way of describing results in Latex, there is a multitude of small variations that makes it hard to identify all of them. Theorems that have a special name, written in another language, or not defined using `\newtheorem` are not extracted by this method. This is not that problematic as the goal is to have a large enough dataset to train machine learning models, but it can introduce a bias as the extracted dataset is not representative of the real paper distribution. For example books (that are under paywalls) and old publications (which fails to be extracted using the LaTeX method) are underrepresented in this dataset.

2.2 Conditional random fields for theorem extraction

Using the LaTeX extraction method, we obtain a dataset of PDF papers with custom annotations to identify the results. The next step is to convert this to a sequence of pairs of features and labels that will be fed into the machine learning system. To do that the PDF is transformed into an XML file using PDFalto. This tool extracts PDF content in the ALTO format presented earlier and PDF annotations in a separate file. Then the Python `lxml` library is used to parse the XML content, producing a set of bounding boxes from the annotations and a sequence of tokens corresponding to the lines (or words, depending on selected granularity) of the document. These tokens can then be associated with numeric and categorical features, computed from XML information. The initial features were the number of words, average word length, the proportion of italic and bold symbols, the usage of a math font, whether the first word is bold, capitalized, equal to "proof" or a heading ("theorem", "definition", ...). I also added more geometric features such as font size, vertical space between the line and the previous line, and whether there is indentation or not.

I focused on linear chain Conditional Random Fields [8] (CRFs) because they are known to be good for sequence labelling. CRFs form a class of graphical models: it is described as follows as a graph (V, E) that has the following properties:

- $V = X \cup Y$, X being the observations and Y the latent variables.
- Y conditioned to X depend only on its neighbors, thus following the Markov property: $p(Y_i|X, Y_j, j \neq i) = p(Y_i|X, Y_j, j \sim i)$ ($\sim$ being the graph neighbor relation)

In the special case of linear chain CRFs, X is the sequence of features, Y is the sequence of labels and we model p as sum of *unary* and *pairwise* potentials:

$$p \propto \sum_i U_{\theta_U}(Y_i, X_i) + P_{\theta_P}(Y_{i-1}, Y_i)$$

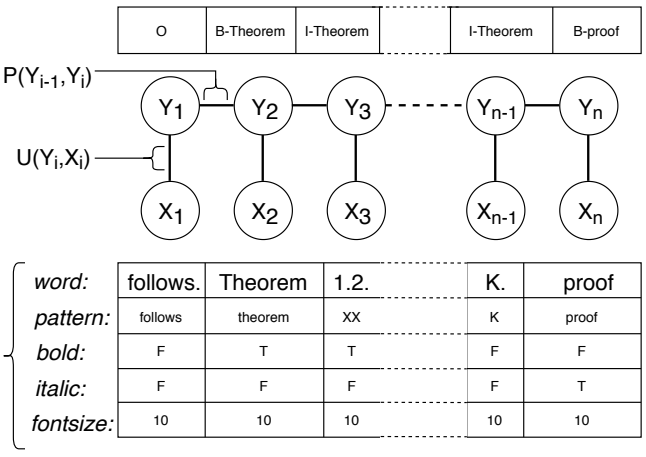

Figure 2: Visualisation of a linear-chain CRF with features and labels.

$U_\theta(Y, X) = \theta_Y \cdot X$ gives the chance of having Y given the features X.

$P_\theta(Y_{i-1}, Y_i) = \theta_{Y_{i-1}, Y_i}$ gives the chance of having a transition from Y_{i-1} to Y_i.

This is a simple model with few parameters but it has the advantage of having efficient algorithms for training and inference. Training is done by usual optimization algorithms such as L-BGFS [9] and inference uses the Viterbi algorithm [10]. There exist more powerful variations of linear-chain CRFs, such as semi-CRFs [11] which locally drop the Markov property to enable more complex behaviors, while keeping computational tractability.

2.3 Choosing metrics

We have a dataset and a model, but now metrics are needed to be able to compare our results. As it is a classification problem, the focus will be on precision and recall measures, combined into the F1 score. However the dataset is prone to class imbalance, as non-results dominates results, so we focus on two metrics: *macro (unweighted) average* and *weighted average* of F1-scores.

2.4 Implementing CRFs on Grobid

After obtaining somewhat good preliminary results (0.90 F1-score on identifying theorems). I decided that taking advantage of Grobid infrastructure would be a good idea to further improve results. To achieve this, the following tasks needed to be accomplished:

- Add a new word-based model to extract results, based on what was done for full-text content.
- Create and test new layout features.
- Generate an annotated dataset that Grobid can handle, using the LaTeX extraction tool.

Adding a new model Model definitions are located in `grobid-core/src/[...]/engines`, with one file per model. Each model is a class that can generate features from a document, apply a tagger (a CRF model for instance) and output a TEI XML file. TEI (Text Initiative Encoding) is the main output format for Grobid: it's a lossy format that describes the document in a more high-level fashion. Even if models are supported to behave similarly, there is no code abstraction for this and a lot of content is redundant between files. That being said, adding a new model would not benefit from the work that already has been done for the other models.

Creating new features As feature generation is not shared between models, improving a model's set of features won't improve the others'. Moreover, even if the dataset is distributed in the Github repository, its format makes it hard to add new features: the training data is distributed as, for each model, a set of pairs of input features and target XML TEI file. Therefore updating features would require finding the original papers before applying a step to re-generate these features. At this date, there is no way to automatically download the training dataset's PDFs. Therefore, even if we only want to add new features, either the original articles need to be found on the internet or a new custom dataset has to be created. But creating a new dataset is time-consuming: for each PDF, a dummy TEI file needs to be generated, before annotating it using the target's nodes. The flow of the document needs not to be changed, or else the matching procedure done by Grobid will fail.

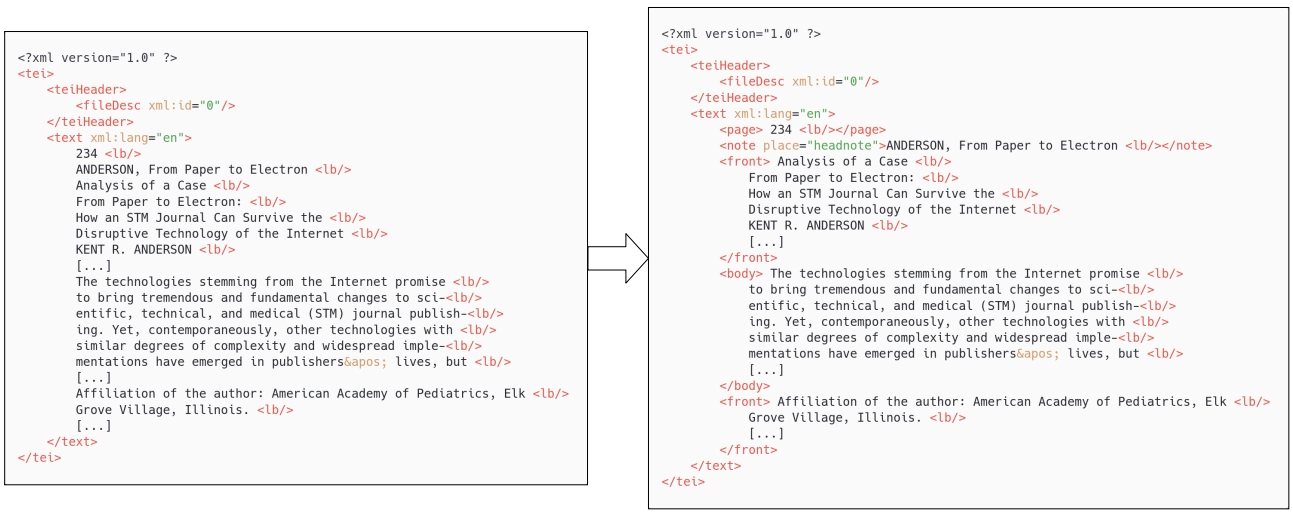

Figure 3: The annotation process in Grobid. The annotator has to follow through the parsed document text in order to add the tags corresponding to each section.

Generate an annotated dataset At last, given what has been explored before, generating an annotated dataset is probably the easiest task. Basically, the algorithm needs to take for input a list of PDFs, generate the non annotated TEI file using Grobid, and then parse the TEI file and the PDF in parallel, using PDF annotations to figure out the label for each node, grouping neighbors under the same node parent.

Grobid is a very good software for production purposes: it is fast and provides very accurate results for most use cases. However, I found that developing models, features, and testing them on training data was hard and error-prone. Even if Grobid could benefit from a results model as an extension, I found that integrating the work into Grobid would have taken too much time and not be flexible enough for future iterations. I therefore chose to develop a new software, with a focus on flexibility and improved dataset management.

3 TKB: an annotated paper management system

The main work of this internship is the TKB software. It comprises a set of interfaces and libraries to manage data and machine learning algorithms. More precisely, TKB is a database of articles that can be annotated, be it manually or using previously trained machine learning models. It allows updating these annotations easily, to design and manage machine learning algorithms, and features in a flexible way, and to provide batch processing of documents. The project began as a simple annotation tool that quickly became a dataset management system.

3.1 TKB software architecture

TKB is made of a central library and several modules that make use of this library.

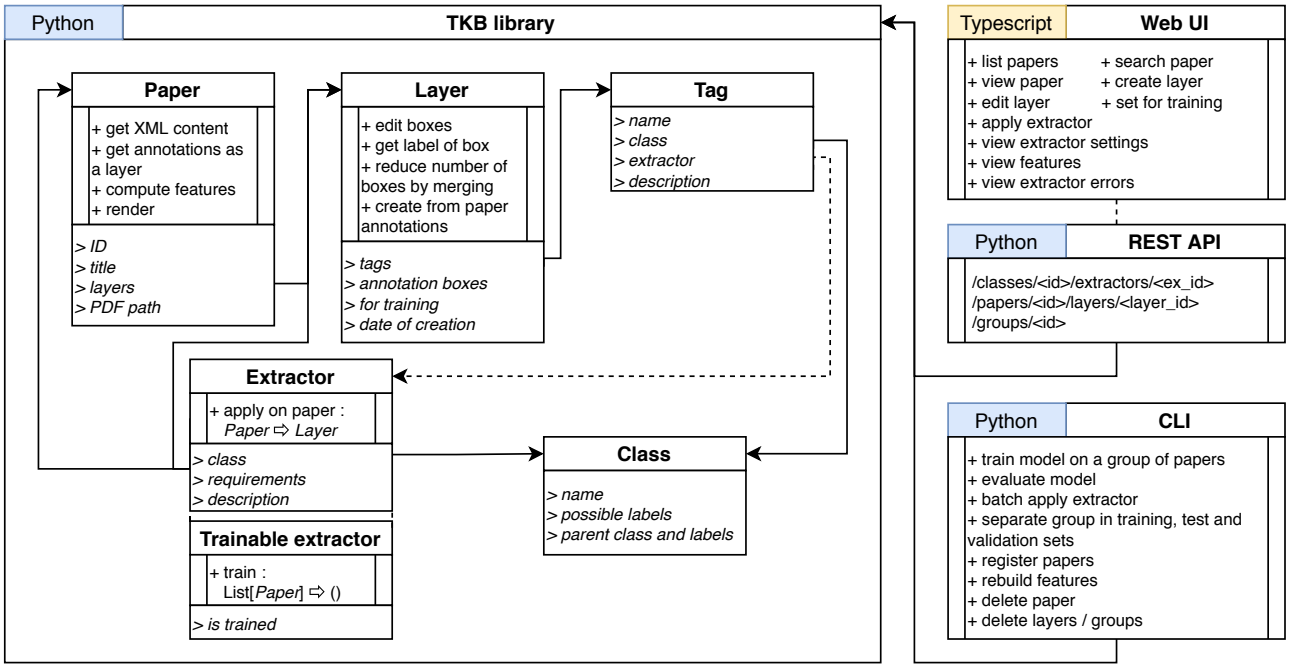

Figure 4: Functional description of the TKB architecture.

`lib:` TKB-lib is the Python library managing the database. It introduces and manage the main concepts that are required in this project:

- Paper: a PDF document that can be annotated through annotation layers.
- Annotation layer: a set of labeled bounding boxes, labels being elements of a given annotation class. Annotation layers can be tagged.
- Tags: a set of annotation layers. Tags make dataset management easier, as they, for example, can be used to identify data provenance or virtually separate data in training and test sets.
- Annotation class: a set of possible labels and where they can appear. For starters, three classes were introduced:
  - `segmentation` makes a coarse partitioning of document elements: `[front,headnote,footnote,page, body,appendix,acknowledgements]`
  - `header` labels elements from the document header: `[title]`
  - `results` identify results in document body and annex: `[lemma,theorem,proof,definition,...]`

- Annotation extractor: a module that can automatically create an annotation layer for a given article. This module is usually a machine learning model, it can be trained using manually annotated samples. Extractors may depend on other annotation classes in order to take advantage of previously extracted information. This hasn't yet been used but for example, we could imagine that a results extractor would make use of the location of specific items such as bibliographical references or formulas, that would be identified by another extractor.

`cli`: A command-line interface manages bulk operations on the dataset. Training and applying models is done through this interface.

`server`: The API frontend exposes TKB-lib's features through a REST API. It allows paper annotations management such as editing annotations or applying an extraction to create a new annotation layer.

`web`: The web interface features a user-friendly way of editing annotations, that are viewed as bounding box overlays on the PDF document. A goal of the web interface is to easily iterate over machine learning designs by visualizing how the algorithm is behaving.

3.2 Data management and representation

The list of papers is stored in a SQLite database file. The title, PDF location and list of annotation layers are stored in this list. Annotation layer content is stored in separate files, in specific papers metadata directories. Initially stored as a simple Python dictionary, the SQLite database usage was motivated by the need for access in a multi-process context. Locks and synchronization are therefore delegated to the SQL driver. `sqlalchemy` has been used to interface with the database. It is quite efficient and features an ORM (object-relational mapping), meaning that data is represented as natural Python objects that can be used (and eventually mutated) in the library.

Paper annotations form a list of bounding boxes, each bounding box containing its coordinates in the PDF document, its label, and optional data for the Web front-end to display. The `pickle` Python module is used to write this data structure to files. This is not the most efficient way of storing data but the usage is straightforward and this format could be used to export data to other platforms. As it's expected to perform spatial operations over annotation layers, a spatial index is populated when loading bounding boxes. This spatial index is used to perform low-cost lookups of bounding box intersections, therefore being useful to label every token in the document. The `rtree` library is used for this purpose, it's a binding for `libspatialindex` implementing the R*-tree data structure.

3.3 Building features while keeping hierarchical information

Feature management is something I wanted to have more flexibility in comparison to Grobid. In our cases, features are automatically extracted descriptors of XML tokens that can be fed into machine learning systems. As XML is a hierarchical format, I wanted to be able to easily create and compose features that can live on different hierarchy levels. For example, featuring the ALTO format can be done be designing `<Page>`-level features, `<TextBlock>,<TextLine>,<String>`-level features and composing them altogether. Given an output hierarchy level, these features and joined and aggregated to produce one single feature table synchronized with the selected tokens. I also developed document-wise feature normalization and token-wise deltas which are both useful preprocessing methods in the case of CRFs.

Example: let's say we have this document and we want to generate `<TextLine>`-wise features.

```
<Page>
  <TextLine>
    <String CONTENT="I'm" FONTSIZE="12"/>
    <String CONTENT="an" FONTSIZE="10"/>
    <String CONTENT="example" FONTSIZE="10"/>
  <TextLine/>
  <TextLine>
    <String CONTENT="the" FONTSIZE="10"/>
    <String CONTENT="end." FONTSIZE="10"/>
  <TextLine/>
  <TextLine>
    <String CONTENT="footnote" FONTSIZE="8"/>
  <TextLine/>
</Page>
```

First step for each tag type a set of features is generated and stored in a `pandas` Dataframe. For this there are several feature extractors working independently, performing a linear pass over the whole document. In order not to repeat that costly phase, feature tables are cached to disk.

<Page>	Position
0	Position.BEGIN

<TextLine>	<Page>	Position
0	0	Position.BEGIN
1	0	Position.IN
2	0	Position.END

<String>	<TextLine>	Position	text	fontsize
0	0	Position.BEGIN	I'm	12
1	0	Position.IN	an	10
2	0	Position.END	example	10
3	1	Position.BEGIN	the	10
4	1	Position.END	end	10
5	2	Position.BEGIN	footnote	8

Figure 5: Example set of extracted features for each token type

Second step join all features, aggregating `<String>` features. All of this is also done efficiently through pandas operations. As we selected the `<TextLine>` hierarchy level, parents (`<Page>`) are simply joined while children (`<String>`) are aggregated before joining. The aggregation step computes the min, max, mean and standard deviation of children's numerical values, makes a copy of the first, second and last values, and count non-numerical values in a bag-of-words fashion.

<TextLine>	Page.Position	TextLine.Position	String. mean_fontsize	String.text
0	Position.BEGIN	Position.BEGIN	10.66	`{"I'm": 1, "an": 1, "example": 1}`
1	Position.BEGIN	Position.IN	10	`{"the": 1, "end": 1}`
2	Position.BEGIN	Position.END	8	`{"footnote": 1}`

Figure 6: Features of document's `<TextLine>` nodes.

Third step (optional) expand features by adding contextual deltas. This is done by taking the discrete derivative of numerical features. Textual features are simply copied. In this example the only numeric feature is `mean_fontsize`. There are other choices to add contextual information, such as simply copying the previous/next value or computing multiplicative changes instead of additive. This is an arbitrary choice based on the ease of implementation in the case of finite differences. It will not be hard to extend this step to compare the behavior of such features.

<TextLine>	...	String.mean_fontsize	String. mean_fontsize_prev	String. mean_fontsize_next
0		10.66	0	-0.66
1		10	-0.66	-2
2		8	-2	0

Figure 7: Features with contextual deltas.

Fourth step (optional) standardize features document-wise. For each numerical feature, an affine mapping is applied to bring the mean to zero and the standard deviation to one. This is also an important step because it enables the identification of variations from the standard layout, therefore being robust to global layout changes.

3.4 User interface

The web frontend is very useful to create a training dataset and to inspect how algorithms are behaving. It features the list of articles in the database and a way to display them with their annotations. Annotations can be edited by moving bounding boxes around and new layers can be created using previously trained models. These features enable a form of "annotation boosting", where a small set of articles is annotated manually before training a model to perform pre-annotation of the dataset. Annotation boosting enables very fast iterations on the documents, meaning that a usable dataset (50 articles) can be created in less than a day.

Typescript and React are used to build the frontend. Typescript is a typed extension of Javascript which makes the development phase more efficient as it avoids a lot of common bugs when writing code. React is a declarative UI library well known for building dynamic web applications. It has a lot of pre-developed components ready to use, such a `react-pdf` making use of Mozilla's PDFjs rendering library and `react-rest-hooks` to communicate with the API.

3.4.1 List papers

Figure 8: Home page: the list of papers in the database. (1) General information on each paper. The title is automatically extracted using the header annotation class. (2) Search in title. (3) Filter papers that have a layer of the given tag. (4) Annotation status for each class: checkmark mean annotated for training, a number means annotated but not validated for training, and a cross means no annotation.

The list view is here to allow searching for a specific paper, and quickly identify which papers are used for training purposes. As we aim to support an arbitrary amount of papers, the list is virtualized, meaning that only the visible subset of the list is pre-loaded while the rest of the content is loaded on demand. This page enables searching for a paper name and filter papers that have a specific tag.

3.4.2 View paper

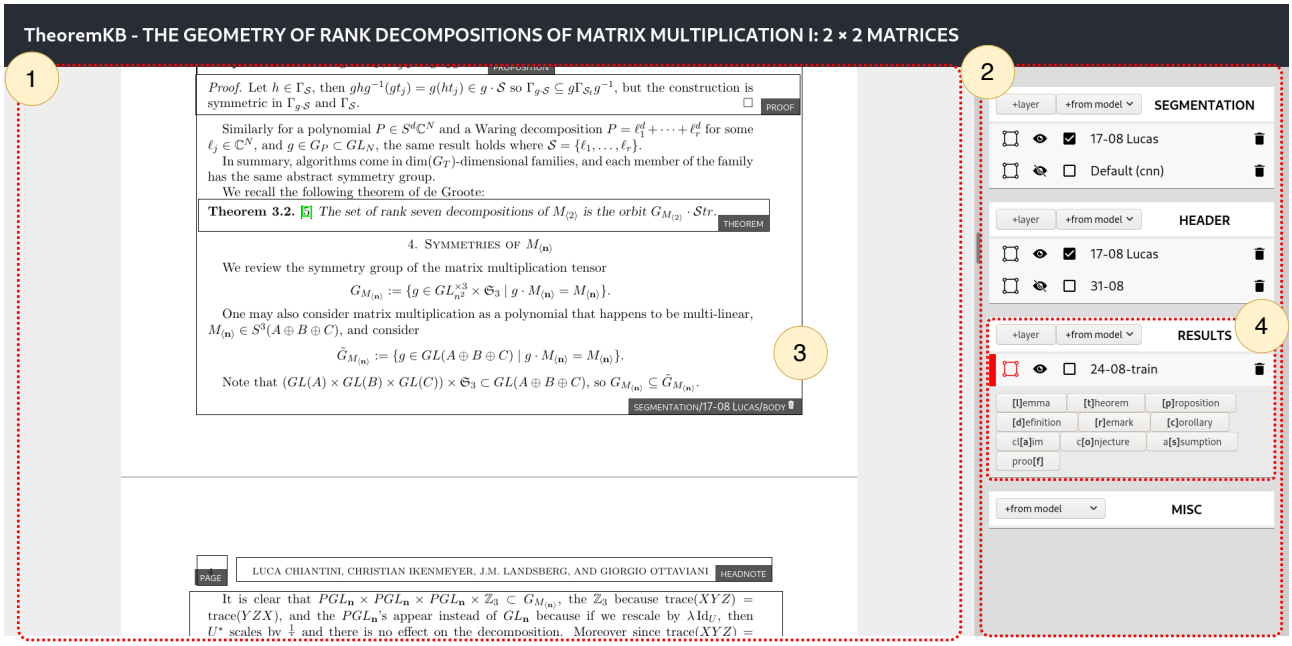

Figure 9: Paper view. (1) PDF rendering. (2) Annotation classes and layers for this document. (3) Annotation boxes as an overlay on the document. (4) Editor menu for an annotation class. It offers the possibility to create a new layer, toggle the layer visibility, edit layer tags and annotations.

Papers are rendered using PDF.js, Mozilla's general-purpose PDF renderer. Similar to the paper list, the display is virtualized, and pages are rendered only when they are needed, improving overall responsiveness when loading

big documents. Pages are rendered onto individual canvases, thus adding an overlay layer is quite simple. One just needs to find the conversion between the page units and the rendered unit (pixels). This is a simple linear transformation that is applied to each bounding box before being positioned over the document.

Annotation boxes are editable: they can be resized, dragged, and deleted using appropriate controls. The annotator can select the layer they want to modify using the right-side menu and clicking the annotate button. Then the type of box is selected either by choosing one of the buttons or by using the automatically generated keyboard shortcuts. The algorithm for shortcut generation is quite simple: for each label, the first letter that is not already used for a shortcut is chosen.

Lastly, the right-side menu can be used to create a new layer by applying an extractor. Depending on the extractor it can take up to a few seconds, as the extraction process may use heavyweight machine learning machinery.

3.4.3 Layer tags and extractors

From the home page, it's possible to access the *layers and extractors* page which gives an overview of all the layer tags. Each tag has a description of its provenance (user or extractor) and can describe extractor settings. When iterating on machine learning models, this is a great tool to keep track of which settings and algorithms have been tested.

4 Models, features and extractors

4.1 Building features

The chosen set of features is a mix of geometrical and lexical properties, extracted from 4 types of ALTO nodes:

- **Page features** A single feature is used, corresponding to the position in the document (begin, in, end).
- **Block features** For blocks, the features are the position in the page, the vertical distance from the last block, the horizontal distance from page border, block dimensions.
- **Text line features** In text lines we start to have more content: features include tabulation (horizontal distance from the left side of the whole block), vertical distance with previous and next block, and whether the textual pattern has already been encountered. This notion of pattern helps to identify headers and footers as they are often repeated across the document.
- **Word features** From word tokens we extract the word, the word transformed into a pattern (keep only letters and transform digits to a special symbol), font size, font styles such as italic or bold, if it contains a number, if it contains special characters, the word position in the sentence, word size and distance with previous and next word.

That makes a total of 25 features. This is not much, but the features aggregation and augmentation steps described in section 3.3 makes them already very powerful.

4.2 Trainable extractors

4.2.1 Conditional random fields

The CRF extractor feeds the paper's extracted features into a linear-chain conditional random field, using its output to tag every token with a label. CRF labels are encoded using a variation of the inside-outside-beginning notation, introduced in [12]: labeled tokens are tagged with 'B-*label*' when they are the first token of the sequence, 'I-*label*' when they are inside the sequence, 'E-*label*' when they are the last of the sequence, and unlabeled tokens are tagged with 'O'.

Several CRF libraries were tested in order to get the most efficient results: Wapiti, CRF++, CRFSuite, and CRFSharp. I used `sklearn-crfsuite` as it is integrated within the sklearn framework. However, this library is not as optimized as I hoped for: it doesn't support multicore training.

CRF++ and Wapiti are written in C/C++, but they do not support feature scaling: meaning that numeric features need to be discretized. That would have not been a problem as the discretize4crf project[5] can take care of this issue. But while benchmarking Wapiti against a small dataset (30 documents) I found that there was no performance gain even if the library claimed to take advantage of multi-core systems. One interesting experiment to make would be to compare discretized and continuous features: discretized features enable non-linear behavior

[5] https://gforge.inria.fr/projects/discretize4crf/

but they might lead to overfitting. An advantage of CRF++ and Wapiti is the support for bi-gram features: basically, the transition weights can be parameterized by the features instead of being constants. This gives more power to the model, at the expense of having even more parameters to fit.

The CRF extractor is therefore built on top of sklearn-crfsuite and uses a generic architecture that can be used for any sequence labelling problem. It has been tested for segmentation, header, and result annotation classes. In order to improve the results, the CRF can be trained to focus on a single class. Also, a sampling mechanism has been added to counter class imbalance. Instead of using the whole document, we only use a fixed-length context around blocks containing target classes. This length is chosen so that there is roughly 2 times more of the default class than of the target class.

Lastly, the CRF regularization parameters can be tweaked to avoid overfitting and obtain the best generalization properties. It is mandatory as when word features are used it is very easy to overfit. The CRF can simply look for rare words and learn the dataset. To counter that crfsuite throws away features that occur less than the minimum frequency threshold, which will be set to 5 for small datasets. This threshold is also useful to improve training speed as training on millions of features can take more than a day. Moreover the L1-regularization parameter is important to perform some kind of feature selection and make the model lighter.

4.2.2 Recurrent neural networks

The use of recurrent neural networks has been considered, but in the end was not implemented. The use of LSTM (Long-Short Term Memory) [13] is a very popular choice in the field of sequence to sequence learning. It is however heavyweight (380M parameters in the introduction paper), and has been developed for short sequences (less than 128 tokens). Our sequences being large (for a 12 pages document: >500 text lines and >7000 words tokens), the classical algorithm to learn weights – backpropagation through time (BPTT) – fails, thus another algorithm needs to be used: Truncated BPTT. I decided not to go on with that cost of implementation before testing other, simpler methods.

But this is something that should be explored. CRF inference and the application of RNNs are related: it has been shown that mean-field inference can be rewritten as the application of a specific recurrent neural network [14]. This means that we may be able to get the structured learning advantages of CRFs while having more computational flexibility.

4.2.3 1D Neural Networks

Another approach to the sequence labelling problem is to build a classifier that will operate on a sliding window over the input sequence. The use of traditional non-recurrent classifiers has been explored in [15], they have the advantage over recurrent methods to use bounded memory, and they can be trained using classical methods. This classifier is a 1-dimensional neural network, with 8 layers and 1.5M parameters, operating on word (`<String>`) tokens and taking for input a set of numerical and categorical features and the embedded word. Words are transformed into vectors using an embedding layer with a vocabulary of size 10K.

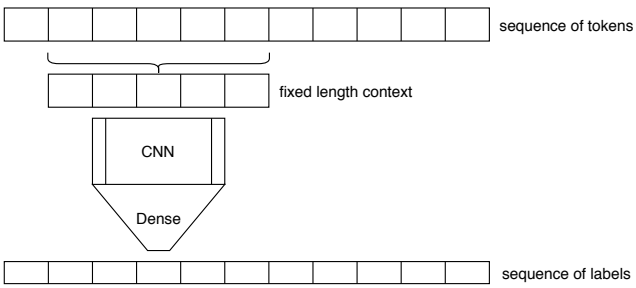

Figure 10: Visual description of a 1-dimensional neural network architecture.

These kind of models can be augmented using conditional random fields as an activation layer to enforce output consistency. In this case the output of the network controls the unary weights of the CRF, while the transition weights are constants learned during training.

4.2.4 2D Convolutional Neural Networks

The main issue with the previous methods is that they take for input already processed features, handcrafted with some domain knowledge. If we want to extract these features automatically, a solution can be to make the PDF rendering as an input and perform a segmentation task. The model takes an image as an input and outputs a pixel-wise segmentation of the document. Then for each token box a majority vote is applied to figure

out which label is given. The CNN has been implemented in Keras/TensorFlow, it is a U-net model with 23 layers and 3.8M parameters.

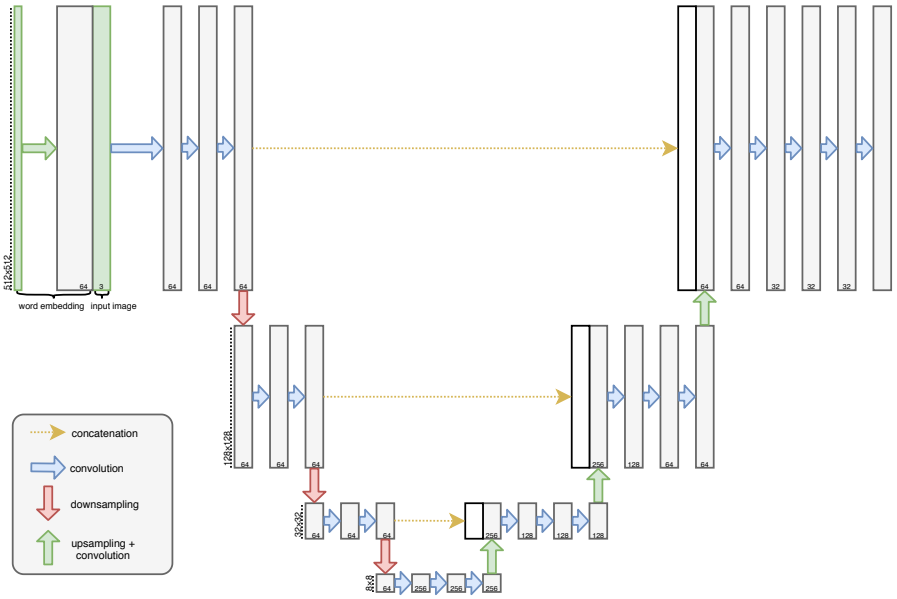

Figure 11: Network architecture description.

The input is a PDF render of resolution 512 × 512px with three color channels and one integer channel that represents words, each word being mapped to an integer using a vocabulary defined during training and words that are not in the vocabulary having a special value. An embedding layer is used to transform this word channel into a 64 channels image using a vocabulary of 10K words. That resolution may seem low, but it is high enough so that visual features can still be recognized. We don't need to be able to read the words as the word embedding layers can take care of that task.

4.3 Extracting LaTeX metadata

To train our extractors, there are two options: annotate manually (as done for the segmentation and header models) or find a way to generate a pre-annotated dataset.

The LaTeX extractor makes use of LaTeX source code to figure out where the results are in a given paper. The steps are the following: get the LaTeX sources, inject some code – as an additional package copied in the working directory – to identify document results in the form of tagged links (of uri `tkb.<result_kind>.<result_number>`) in the PDF and at last read those links as bounding boxes for the results. Two strategies have been found to find results in LaTeX sources.

4.3.1 Redefining the command that defines the environments.

As results are mostly created through the `\newtheorem` command, the extraction package redefines the command to add a hook which creates the desired links when a result environment is used. To insert the extraction package, a `\usepackage` command needs to be inserted in the document in a valid position. Indeed it needs to lie after the initial definition of `\newtheorem` and before the first use of the command. This is actually quite a hard task, as it can happen anywhere in the beginning of the document: it can happen in a document class definition, in an included file or package. Therefore extraction code is inserted by parsing the LaTeX file and exploring potential use of `\input` or local `\usepackage`.

4.3.2 Redefining the environments

Another way to proceed is to redefine environments that we know will contain results. Basically the extraction package encodes a mapping of well-known environments and their associated results: `thm,theo,theorem => Theorem`, `lem,lemm,lemma => Lemma`, etc. Insertion of this code is easier as we only need to find the beginning of the document when the environments are defined. It has however the drawback of potentially missing results that are not part of our list of environments.

4.3.3 Comparison

To compare the two methods, extraction has been performed on 6000 articles and the number of extracted articles (compiled with at least one result) along with the number of results found for each category was stored.

	Override newtheorem	Override environments
Successful extraction	3480	**4387**
Theorems	20678	28486
Lemmas	21228	27974
Definitions	13436	17821
Propositions	5791	8299
Proofs	36854	51534

Table 1: Extraction results on the 6000 Arxiv dataset in the field of computational complexity.

As shown in table 1, the second method, overriding a specific list of known results environments, is clearly more robust. Indeed by not having to find a specific place in the source code to include the package, the risk of having a compilation error is reduced.

As both methods have flaws, it would be a good idea to combine the two techniques, thus improving the number of extracted results. However, that implies more complex and probably slowed code, so we settle for the *override environments* method. Therefore we obtain a dataset of **4387** articles that we can use to train result extraction algorithms.

4.4 A naive algorithm for result extraction

To set a baseline for result extraction, I developed a simple algorithm that performs the following labelling:

- Set the *target words* to the list of results names
- For each textual token in the paper:
  - If the token is the first of its line, is in bold or italic and contains one of the *target words*, set the label to the target word in question.
  - Else if the token is the first of its block, and the first of its line, set the label to `other`.
  - Else set the label as the previous label.

Being a very simple heuristic, we don't expect good results from it. It should at least find a good chunk of results' beginning, the hard part being to find when they end.

4.5 Special extractors: agreement and features

Special extractors have been created to debug and visualize what's happening. The Web UI handles them separately, enabling them to display additional data.

agreement: takes two models for input and outputs where these two models disagree. It can take one as the ground truth to compute precision, recall, and the F1-score.

features: generates a box for each token and associate the computed features on it. These features can be examined by activating this extractor on a paper, displaying the layer, and hovering sections of the document. As most features are based on visual traits, it makes it more easy to find bugs as one can observe values along with the visual token in the document.

5 Results

With datasets for each annotation class and extractors developed, our system's performance can be evaluated. Initially, all developed extractors were supposed to be tested for each annotation class. However, methods based on neural networks have yet to be successful. As the initial focus was on conditional random fields, it was too late in my internship to obtain sufficient resources to correctly train neural networks. Results on CRF methods are promising, meaning that at least features and datasets are correct, and that the other methods should work given more time to fix machine learning issues.

5.1 Segmentation class

To train segmentation models, a small dataset has been created, composed of 50 semi-automatically annotated documents. To create this dataset, I manually annotated 10 documents before training a first model that helped me annotate 40 other papers. Segmentation is not the main focus but it's a good benchmark as the problem seems easier than extracting results. The dataset is split between 35 documents for training and 15 documents for testing.

CRF This small quantity of documents is sufficient to train conditional random fields, as long as they are sufficiently regularized. We use L1-regularization and a minimum frequency threshold for features, chosen manually to avoid overfitting. The comparison for L1-regularization is available in appendix A.1.

One interesting fact is that using a line-based model is sufficient for this task. It yields the same performance as a word-based model while being faster to train (as the dataset size is reduced). We can see that difficulty resides in finding the annex. It's hard to differentiate an annex section from a body section because they practically have the same content. One could think that the fact that the annex being separated from the body by a bibliographical section should be something that the CRF could learn, but the problem is that `page` tokens break the Markov property, making information unable to propagate between two pages. That's where non-linear-chain CRFs could help, by allowing that kind of information to pass through. An alternative would be to start by extracting page numbers before performing the rest of the document segmentation with page numbers eliminated.

	precision	recall	f1-score
acknowledgement	0.680	0.383	0.490
appendix	0.622	0.107	0.183
bibliography	0.990	0.989	0.989
body	0.918	0.985	0.951
footnote	0.799	0.790	0.794
front	0.718	0.947	0.817
headnote	0.889	0.729	0.801
page	0.971	0.952	0.962
macro avg	0.823	0.735	**0.748**
weighted avg	0.894	0.911	**0.887**

Table 2: Results on the test set for **line-based** CRF. It took 220s to train, and has *939* active features (over 20K). *Parameters: minimum frequency of 5, l1-regularization of 1.5.*

precision	recall	f1-score	support
0.953	0.383	0.546	316
0.634	0.117	0.197	17874
0.989	0.883	0.933	10447
0.914	0.996	0.953	189265
0.864	0.539	0.664	2190
0.873	0.869	0.871	3228
0.971	0.702	0.814	660
0.970	0.933	0.951	315
0.896	0.678	**0.741**	224295
0.894	0.912	**0.887**	

Table 3: Results on the test set for **line-based** CRF. It took 20 minutes to train, and has *1094* active features (over 13K). *Parameters: minimum frequency of 5, l1-regularization of 3.0.*

5.2 Header model

As it's very simple to identify the document title when the front has been found, we obtain perfect results while using the line-based CRF. It took 2 minutes to train and the resulting model consists in 39 features. Looking at the features confirms the intuition: it looks for a font size that is larger than usual font size and a large gap with the next block.

5.3 Results

When training the result extractors we used a set of 1936 documents for training and 431 documents for testing. Only a subset of the 4387 documents dataset obtained using the LaTeX extraction method is used, as CRF training time was already high when using half of the documents.

Naive Presented in section 4.4, the naive algorithm performs well in precision but fails in recall. It confirms the intuition that it's easy to find when theorem starts but it's harder to find when they end. Overall we obtain a macro F1-score of **0.56**.

	precision	recall	f1-score	support
assumption	0.036	0.622	0.068	283
claim	0.468	0.657	0.546	17031
conjecture	0.316	0.664	0.428	2825
corollary	0.813	0.700	0.752	32371
definition	0.726	0.552	**0.627**	140873
lemma	0.737	0.640	**0.685**	173266
proof	0.874	0.285	**0.430**	1723296
proposition	0.709	0.664	0.686	34270
remark	0.760	0.736	0.748	28520
theorem	0.633	0.640	**0.637**	138122
macro avg	0.607	0.616	**0.561**	2290857
weighted avg	0.832	0.370	**0.487**	

Table 4: Results for the naive algorithm.

Conditional random fields We find that line-based conditional random fields perform much better than the naive method. It has a macro f1-score of **0.72** and is especially good at finding theorems (0.88), lemmas (0.89), definitions (0.81), and proofs (0.81), which are the main results we are looking for.

Finding the right parameter for these models is hard because they are take a lot of time to train and I'm not yet satisfied with the results. In particular, line-based and word-based results are not comparable because I took less time tweaking the parameters for word-based CRFs, as training time usually takes 4 times longer than it's line-based counterpart. The two main variables are the *minimum frequency threshold* to limit the number of features and accelerate training, and the *L1-regularization* to avoid overfitting.

As the dataset is unbalanced and no techniques are used to counter it, we can observe that the **assumption** class is lost in regularization, its support being so small compared to other classes such as proof or lemma. Proofs are especially over-represented because they can span whole pages of documents.

	precision	recall	f1-score
assumption	0.000	0.000	0.000
claim	0.628	0.839	0.718
conjecture	0.903	0.629	0.742
corollary	0.874	0.847	0.860
definition	0.835	0.775	**0.804**
lemma	0.901	0.874	**0.888**
proof	0.781	0.834	**0.806**
proposition	0.848	0.828	0.838
remark	0.714	0.691	0.702
theorem	0.886	0.872	**0.879**
macro avg	0.737	0.719	**0.724**
weighted avg	0.800	0.834	**0.816**

Table 5: Results on the test set of *line-based* CRF. It took 5 hours to train and has 37K active features (over 72K). *Parameters: minimum frequency of 50, l1-regularization of 5.0, l2-regularization of 0.1, 300 iterations.*

precision	recall	f1-score	support
0.000	0.000	0.000	283
0.818	0.542	0.652	17031
0.883	0.356	0.508	2825
0.882	0.596	0.711	32371
0.747	0.649	**0.695**	140873
0.892	0.793	**0.840**	173266
0.701	0.893	**0.786**	1723296
0.833	0.632	0.719	34270
0.775	0.252	0.380	28520
0.836	0.786	**0.810**	138122
0.737	0.550	**0.610**	2290857
0.733	0.844	**0.777**	2290857

Table 6: Results on the test set of *word-based* CRF. It took 21 hours to train and has 350K active features (over 800K). *Parameters: no minimum frequency, l1-regularization of 0.3, l2-regularization of 0.1, 500 iterations.*

6 Conclusion

This internship feels complete, but there are a lot of things to improve in order to take full advantage of this work. The web interface can be improved to better present extracted information: it should enable for example searching for extracted results. In the machine learning workflow, TKB could feature training monitoring and support multiple versions of the same model. Moreover there is no standard for box-annotated document datasets: something should be done to exchange this kind of data.

In terms of machine learning, the project could benefit from more NLP research and experimentation. Future work should include testing recurrent models, and further research on 2D CNNs over document rendering in order to get a machine learning workflow without handcrafted features. Semi-CRFs are an option to extend the computational capacity of CRFs, but careful feature engineering could do the trick. Lastly, improving training time when using big datasets and/or a large number of features could be very useful to accelerate the workflow and perform more advanced model selection techniques such as cross-validation to find the right parameters.

At last, the TKB library could be groundwork for the TheoremKB project. It gives a clear workflow to generate features and test models while allowing to manage the dataset in a user-friendly manner. By being extensible, I hope that this project will enable creating new methods for information extraction from structured documents. In any case, this internship was very insightful: it proved to me that the core of machine learning may not be the algorithms, but the workflow around it: data preprocessing, feature generation, results management, etc. While there are very powerful toolboxes to build algorithms (TensorFlow/Keras, PyTorch), the rest of our machine learning workflow is usually very ad-hoc, and that's why I wanted to explore what could be done when focusing on the data, and not the algorithms.

References

[1] P. Senellart. TheoremKB: Towards a knowledge base of mathematical results. SinFra Symposium 2019 on Artificial Intelligence, 2019.

[2] Ruikai Liu Jorj X. McKie. PyMuPDF. https://github.com/pymupdf/PyMuPDF.

[3] Thaer M Dieb, Masaharu Yoshioka, Shinjiro Hara, and Marcus C Newton. Framework for automatic information extraction from research papers on nanocrystal devices. *Beilstein journal of nanotechnology*, 6(1):1872–1882, 2015.

[4] Dominika Tkaczyk, Pawel Szostek, Mateusz Fedoryszak, Piotr Jan Dendek, and Lukasz Bolikowski. Cermine: automatic extraction of structured metadata from scientific literature. *International Journal on Document Analysis and Recognition (IJDAR)*, 18(4):317–335, 2015.

[5] GROBID. https://github.com/kermitt2/grobid, 2008–2020.

[6] Thomas Lavergne, Olivier Cappé, and François Yvon. Practical very large scale CRFs. In *Proceedings the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 504–513. Association for Computational Linguistics, July 2010.

[7] DeLFT. https://github.com/kermitt2/delft, 2018–2020.

[8] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[9] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

[10] G David Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[11] Sunita Sarawagi and William W Cohen. Semi-markov conditional random fields for information extraction. In *Advances in neural information processing systems*, pages 1185–1192, 2005.

[12] Lance A. Ramshaw and Mitchell P. Marcus. Text chunking using transformation-based learning. *CoRR*, cmp-lg/9505040, 1995.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[14] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1529–1537, 2015.

[15] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

A Experiments

A.1 Choosing the L1-regularization parameter in the case of CRFs

This experiment is done using the line-based conditional random field with no L2-regularization and a minimum feature frequency threshold of 5. We observe perfect fit when no regularization is used, decreasing as the L1 parameter increases. The best regularization parameter found is of 1.5, yielding a F1-score of **0.75** for test and **0.98** for training. This remains a situation of overfitting but from there, reducing overfitting doesn't improve the results, meaning that the dataset might be too small to cover the real distribution.

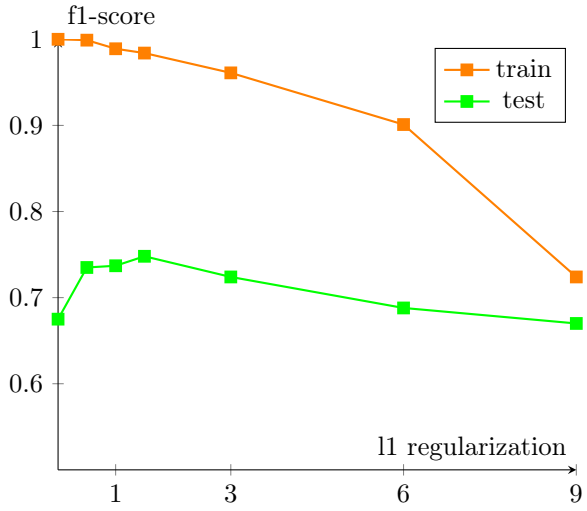

Figure 12: CRF performance in function of the L1-regularization parameter.