PDFMEF: A Multi-Entity Knowledge Extraction Framework for Scholarly Documents and Semantic Search

Jian Wu[†], Jason Killian[‡], Huaiyu Yang[†], Kyle Williams[†], Sagnik Ray Choudhury[†],
Suppawong Tuarob[‡], Cornelia Caragea[*], C. Lee Giles[†‡]
[†]Information Sciences and Technology
[‡]Computer Science and Engineering
Pennsylvania State University, University Park, PA, 16802 USA
[*]Computer Science and Engineering, University of North Texas, Denton, TX, 76203 USA

ABSTRACT

We introduce PDFMEF, a multi-entity knowledge extraction framework for scholarly documents in the PDF format. It is implemented with a framework that encapsulates open-source extraction tools. Currently, it leverages PDFBox and TET for full text extraction, the scholarly document filter described in [5] for document classification, GROBID for header extraction, ParsCit for citation extraction, PDFFigures for figure and table extraction, and algorithm extraction [27]. While it can be run as a whole, the extraction tool in each module is highly customizable. Users can substitute default extractors with other extraction tools they prefer by writing a thin wrapper to implement the abstracts. The framework is designed to be scalable and is capable of running in parallel using a multi-processing technique in Python. Experiments indicate that the system with default setups is CPU bounded, and leaves a small footprint in the memory, which makes it best to run on a multi-core machine. The best performance using a dedicated server of 16 cores takes 1.3 seconds on average to process one PDF document. It is used to index extracted information and help users to quickly locate relevant results in published scholarly documents and to efficiently construct a large knowledge base in order to build a semantic scholarly search engine. Part of it is running on CiteSeerX digital library search engine.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: [Information filtering]; H.3.7 [**Digital Libraries**]: [Systems issues]; H.3.4 [**Systems and Software**]: [Performance evaluation (efficiency and effectiveness)]

General Terms

metadata extraction

1. INTRODUCTION

K-CAP 2015, October 07 - 10, 2015, Palisades, NY, USA

ACM 978-1-4503-3849-3/15/10$15.00
DOI: http://dx.doi.org/10.1145/2815833.2815834.

Digital libraries have completely changed the way researchers search for relevant work in almost all knowledge domains. In 2014 it was estimated that the total number scholarly articles on the web in English was at least 114 million [19]. With unprecedented growth in the publication of papers [20], authors find it difficult to read all relevant papers in order to locate useful results. In some cases, an author cites a paper just because he/she is interested in a particular *entity*, such as the conclusion, an approach, a results table, and/or a figure. Having a scholarly search engine that can utilize these entities would help in such a search and could be the foundation of a semantic scholarly search engine.

In general, a scholarly document consists of several of these entities, if not all: a header, a text body, a bibliography, figures, tables, math and algorithms (even chemical formulae [26]). The header encapsulates a title, author names, author affiliations, author emails, an abstract, a publication year, a venue (conference proceedings or journals), a volume number, pages, and/or series. The text body contains all descriptive text including ASCII and non-ASCII characters. The bibliography is a list of publications cited by the current paper, usually appearing at the end of a scholarly document and can be named "bibliography", "references", or even "notes". Figures are very illustrative entities used to present results with certain trends and are used more frequently in modern papers. Tables are effective in presenting structured data, such as experimental results. Algorithms, usually expressed in pseudo-codes, are effective to present a way to solve a computational problem. Math appears as equations. Algorithms appear not only in computer science papers, but also in other science domains, such as mathematics and physics. Our goal is to construct a knowledge base of each scholarly document that then can be encoded in a scholarly document ontology.

There has been a great deal of effort in developing methods for extracting specific entities from scholarly documents. For example, Apache PDFBox has been widely used to extract text and attributes of PDF documents. Similar tools include Poppler [2] and PDFLib TET [1]. In crawl-based digital libraries, the title and authors are important for clustering near-duplicate papers based on the assumption that no or very few research papers have exactly the same title and authors. Existing open-source header parsers include SVMHeaderParse [18] and GROBID [24]. There are bibliography parsers such as ParsCit [13] (GROBID parses citations as well). Existing table extraction tools include Tabula

[3], automatic table extractors [22], and PDFFigures [9]. A figure extraction tool [7] has been developed for general academic documents and for captions and figures in biomedical PDF documents [23]. An algorithm extraction tool has all been developed[27]. In addition, a crawl-based digital library search engine needs to filter returned documents by classifying them as scholarly or not [5].

While there are many existing tools for extracting *semantic* entities, there has not been a framework proposed to integrate them all together. The Unstructured Information Management Architecture (UIMA) provides a framework to do text-based analysis by converting unstructured data into structured data [16]. However, it focuses on text mining, such as analyzing logs and clinical notes, through automatic annotations. The framework here is aimed at efficiently performing multi-entity extraction using existing open-source software, including but not limited to plain text, header, citations, figures, tables, and algorithms. This framework is designed to be modular and scalable. It has built-in wrappers for a selection of extraction tools, such as PDFBox, but others can easily be substituted such as another text extractor, e.g., PDFLib TET, by writing a simple wrapper to implement the interface. Parallelization is also configurable by specifying the number of concurrent processes, allowing the code to process a large corpus of scholarly documents on a multi-core machine.

This paper is organized with Section 2 describing a typical metadata extraction system of CiteSeerX, that illustrates challenges and the design of the new extraction framework, including all modules and how we choose and implement an extraction tool. In Section 3, we discuss design implementations. In Section 4, we run this framework on a random sample of data and show its performance. Section 5 describes applications that could potentially make use of the framework. Section 6 summarizes and discusses future work.

2. DESIGN

2.1 Current Design

The metadata extraction system of CiteSeerX includes the following modules: text extraction, academic document filter, citation extraction, and header extraction.The four modules are assembled in a Perl script, with a Java wrapper. The wrapper acts like a job dispatcher. Once started it creates a job folder and launches multiple threads, each of which works on a batch job by retrieving a list of unextracted documents from the crawl database and then processing by the Perl script. The text extraction module uses PDFLib TET for plain text extraction; the academic document filter implements a rule-based classifier, which looks for keywords/phrases, such as "bibliography", "references", and their variants from the full text [29]; the citation extraction module uses *ParsCit* [13]; the header extraction module employs *SVMHeaderParse* (hereafter SVMHP). Both ParsCit and SVMHP require plain text as input. The output of ParsCit and SVMHP are further compiled into a single XML file used for ingestion. The ingestion populates a production database with new document metadata, adds documents into a production repository, and indexes full text. The extraction process has a very limited logging function. The current extraction module has the following drawbacks and limitations:

1. The Perl script is difficult to maintain and has poor portability; Python would be better.
2. Although the rule-based document filter achieves a generally high accuracy of 80%–90% [29], the recall is below 80%, indicating that it misses a significant fraction of scholarly documents. A sophisticated classification algorithm, such as the one developed by [5] based on structural features and supervised machine learning, would do better.
3. The SVMHP has been evaluated against other header extraction tools [21], and the results indicate that GROBID [24] outperforms other competitors. An independent evaluation is performed to verify the superior extraction performance of GROBID (see below).
4. The current extraction is limited to textual content. It is desirable to integrate multiple extraction tools for figures, tables, and algorithms.
5. The current extraction module has limited logging function, making it difficult to trace the reason causing a failed extraction.
6. The current design is optimized for TET, SVMHP, and ParsCit, making it difficult to switch to other extraction tools. This ad hoc design significantly limits the applicability of the extraction system.

2.2 New Design

Redesigning the extraction framework faces several challenges. First, the framework should be highly portable and modular, i.e., it should be relatively easy to fit another extraction tool into the framework and/or to plug/unplug one or more modules from the framework, for example, using TET for plain text extraction instead of PDFBox, or to remove the academic document filter. In most cases, this requires developing an "adapter" or a thin wrapper to a specific extraction tool, because different tools are written in different programming languages, have different API input/output requirements, etc. To accomplish this, the framework should define an abstract interface to implement the wrapper and that deals with input, output files, and returning standard extraction status. Specifically, the wrapper makes a system call to run an application API (either as a command line or as a service). The output and error messages are retrieved and parsed by the wrapper to determine the status (successful or fail), which is used to control the work flow and logged. The pluggable feature requires a centralized design, so a module change needs minimal modification of the source code or configurations. Second, the design should consider information shared between modules to make sure that the dependencies are generated before they are used and the framework does not waste time to duplicate some workload across multiple modules. This requires us to define required and optional modules and arrange them in a certain order. For example, while GROBID takes a PDF file as input, other header extraction tools accept a plain text. Because full text is a prerequisite for many extraction tools, the framework by default always extracts full text at the first place. Another example is that if GROBID is used for both header and citation extraction, it only needs to be run once when extracting headers, and the citation extraction module can work directly on its TEI output. Third, the

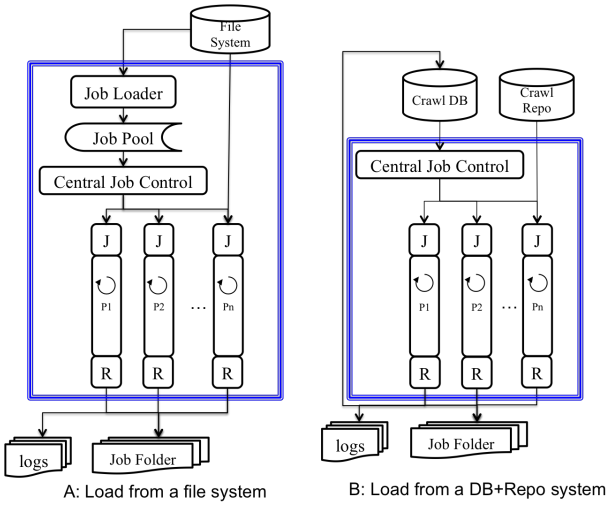

Figure 1: Proposed architecture. "P_i" represents individual process. The blue boxes enclose the frameworks.

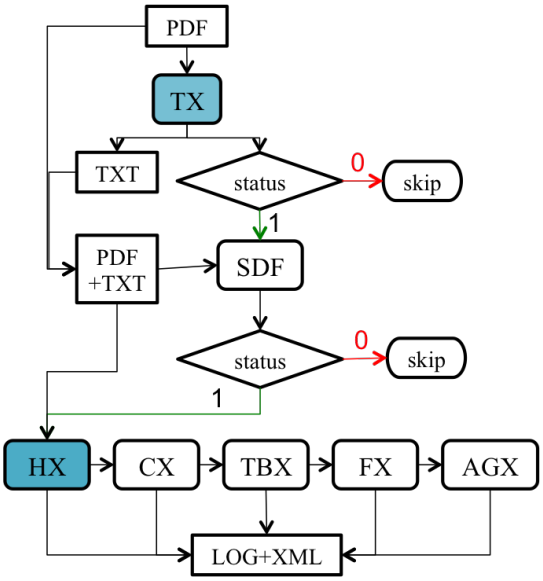

Figure 2: The PDFMEF flowchart.

framework must handle exceptions from different extraction tools and their results appropriately. For example, if the full text contains zero line, it is treated as a failure extraction even if an empty text file is generated.

The high level design of the proposed framework is depicted in Figure 1. A and B are very similar except that they support different input modes. In Mode A, the *Job Loader* loads documents directly from the file system, i.e., a folder containing PDF documents to be extracted. This is a convenient way to run a batch extraction job without setting up the crawl database and repository. The *Job Loader* then creates a *Job Pool*, which works like a set or dictionary with document paths (or derived equivalents) as keys. Each process has a *Job Controller* ("J") and ends with a a *Job Cleaner* ("R"), which cleans temporary files, integrates extraction results and writes extraction status to log files. The "J" module switches the process between on and off. The *Central Job Control* works like a commander, which reads external command, e.g., a stop command, from users by periodically checking the settings in a configuration file. The signal is then passed to each job controller, which stops fetching new documents after the current job list is finished. This *soft-termination* feature is especially helpful when the framework is running in Mode B, because it retains the integrity of extraction jobs by avoiding partially extracted documents. When the framework is brought up again after a shutdown, it does not rerun the same documents it had run before, and it does not deal with documents that are partially extracted without status updated in the database. In Mode B, the document list is retrieved from the database and the job cleaner reports extraction status of each document to the database.

Because working in Mode B requires extra coding which depends on specific database schemas, we focus on design and experiments on the framework itself. Database queries are usually much faster than extraction, so they are generally not bottlenecks.

In the extraction framework, each process executes a number of extraction modules (EMs). The current default settings include a text extraction module (TX), a scholarly document filter (SDF), a header extraction module (HX), a citation extraction module (CX), a table extraction module (TBX), a figure extraction module (FX), and an algorithm extraction module (AGX). Two design options are considered. In the first design option, *within each process*, the EMs are arranged sequentially, i.e., each EM is invoked one after the previous one is finished. In the second design option, *within each process*, each EM is called by a *subthread* and EMs are executed in parallel. We choose the first. This is because of the substantial time difference it costs to execute different EMs. For example, depending on the extractor, citation extraction can be ten times slower than header extraction (see experiment results below). If we choose the second design option, the subthread executing the header extraction will be idle while the citation subthread is working. In contrast, the sequential arrangement utilizes CPUs more efficiently and it is easier to scale it up by adding more cores. The sequential arrangement is also easier to implement because there is no need to deal with synchronization issues. For example, as the table extraction is based on full text, it has to keep idle until full text is extracted. The flowchart of the sequential design are depicted in Figure 2 (job controller and job cleaner are not displayed). The full text and header EMs are required, the other EMs are optional. In the next section, we discuss default extraction tools for each EM.

2.3 Full Text Extraction

Commercial text extraction tools such as PDFLib TET was believed to perform better than open source counterpart, e.g., Apache PDFBox. However, both of them have been under active development, so it is reasonable to evaluate them based on their current versions.

The testing sample is comprised of 1000 PDF documents randomly selected from the crawl database of CiteSeerX, so the sample contains non-academic documents. Because building a ground-truth for text extraction is not feasible

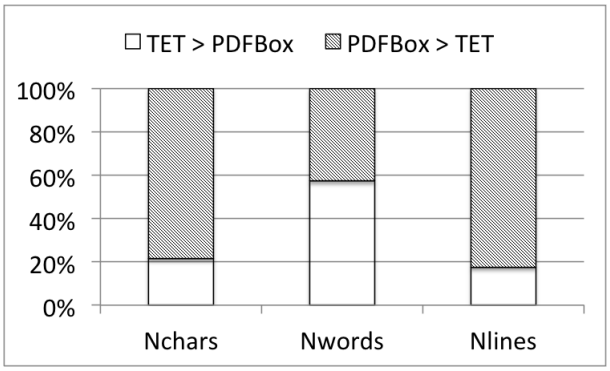

Figure 3: TET vs. PDFBox on numbers of extracted characters, words, and lines.

due to extremely complicated formatting of PDF files and the lack of standard transformation schema from PDF files to text files, we only perform *baseline* comparison of one against the other. For this paper, we compare the number of extracted PDF files, number of lines, characters, and words for each extracted PDF file. We use the most recent releases of TET (version 4.4) and PDFBox (version 1.8.6) at the time of writing. We then use the Linux built-in `wc` command to count lines, characters, and words. The results are tabulated in Table 1. This table implies that PDFBox 1.8.6 has a better performance in terms of the number of extractable PDF files, although there are 40 files that neither of them can extract. In fact, there is only one file that TET can extract but PDFBox cannot, but there are 17 files otherwise. In Figure 3, we compare the number of characters (Nchars), words (Nwords), and lines (Nlines) of files extractable by *both* of them. This chart indicates that PDFBox extracts more characters and lines than TET, but extracts less words for most of documents. Given the current results, it is difficult to conclude whether TET or PDFBox can absolutely beat the other. We implement both of them in the framework. Because TET is a paid software and PDFBox has a free license, we will experiment on PDFBox only.

Either TET or PDFBox can be executed by launching a system call. A successfully extracted document is passed to the SDF. Otherwise, it is dropped and logged.

Table 1: TET vs. PDFBox on extractable files.

		PDFBox Fail	PDFBox Success	TET Overall
TET	Success	1	942	943
TET	Fail	40	17	57
PDFBox	Overall	41	959	1000

A successful extraction output contains at least one line.

2.4 Academic Document Filter

A rule-based classifier discriminates between academic and non-academic documents by identifying textual patterns such as "bibliography" and "references" terms from full text. A more sophisticated approach has been developed by [5], which is based on a set of structural features using supervised machine learning algorithms. The four types of structural features and examples are tabulated in Table 2. To enhance classification quality, we re-train the classifier with a new sample. The new sample extends the training sample described in [5] by adding in 2,000+ manually labeled documents randomly selected from the crawl repository of an existing digital library. We also supplement the sample set with manually curated sets for theses, resume, slides, and books. The labeling was performed by three people independently and verified by a domain expert. The revised training data yield excellent testing results. The precision and recall from 10-fold cross validations are both over 95%. To our knowledge, this is the best classification approach to separate scholarly documents from a corpus of crawled PDF documents.

Table 2: Examples of structural features.

File specific features
File size in kilobytes; page number.
Text specific features
Document length in characters, words, and lines; Ratio between reference mentions and tokens.
Section specific features
Appearance of section names, e.g., abstract, etc.
Content features
Appearance of "this paper", "this book", etc.

For a given document, the filter takes full text, and the PDF file (the filter needs to obtain some structural features directly from the PDF file) as input, and output a boolean value.

2.5 Header Extraction

In [21], Lipinski et al. compares a list of metadata extraction tools for scientific PDF documents based on a sample selected from arXiv digital library. A partial score scheme is used to assess performance of individual tools on the following metadata fields: title, authors, and abstract. They also evaluate authors' last names and year information in their second and third round of evaluation. In their evaluation, GROBID delivers the best results, which significantly beats SVMHP and other competitors.

Here, we attempt to verify the conclusion in [21] using a more heterogenous sample selected from CiteSeerX. The goal is to investigate the *best* performance of GROBID over SVMHP. This sample is comprised of papers whose titles and/or authors were mistakenly extracted by SVMHP, and were corrected by end users. We manually extract paper titles, and authors as the ground-truth and extract titles, and authors using GROBID. Then, we compare automatically extracted metadata against the ground-truth. We focus on titles and authors because they are the most essential fields in a crawl-based digital library search engine. The evaluation was performed by three people independently, each of whom visually compares the ground truth with automatically extracted metadata. The only guideline in this user study is to be consistent with their own judging criteria. The posterior survey indicates that these users generally follow two principles: (1) all words must be parsed correctly to qualify a correct title extraction, with exceptions of letters with accent or ligatures; (2) all authors must be parsed cor-

rectly to qualify a correct author extraction, with exceptions of letters with accent or ligatures. The final judgements are reported based on the majority vote for each field. Table 3 shows the evaluation results. The results are in general consistent with those in [21]. Given the heterogeneous property of our sample, it is clear that GROBID outperforms SVMHP significantly in terms of both titles and authors by at least 30% in the best senario.

Because the sample is biased against SVMHP, the results in Table 3 represent the *best* performance of GROBID over SVMHP. Empirically, in most cases, GROBID achieves a comparably quality when SVMHP extracts well. The overall performance over SVMHP strongly motivates us to employ GROBID for header extraction of academic documents.

The header extraction executes on research papers only so the corresponding EM is arranged after the SDF. The GROBID can be executed either stand-alone or as a service. The latter is more efficient because it loads required library files only once. The EM handles errors output by GROBID appropriately, such as the service is not responding. It first run GROBID to generate the TEI file [12], and then extracts fields corresponding to a predefined and customizable metadata schema into an XML file.

Table 3: GROBID vs. SVMHP.

Title Extraction				
		GROBID Incorrect	GROBID Correct	SVMHP Overall%
SVMHP	Correct	5	43	65.7%
SVMHP	Incorrect	1	24	34.3%
GROBID	Overall%	8.2%	91.8%	100%
Author Extraction				
		GROBID Incorrect	GROBID Correct	SVMHP Overall%
SVMHP	Correct	4	38	57.5%
SVMHP	Incorrect	3	28	42.5%
GROBID	Overall%	9.6%	90.4%	100%

2.6 Citation Extraction

ParsCit is a citation extraction tool developed by [13]. Since GROBID parses citations, we perform a *baseline* comparison between them, by looking at the numbers of reference items they can extract out, and the quality of extracted titles, authors, and years. These three fields appear in almost all reference items, regardless of conference papers, journal articles, or books. There are a few exceptions, such as citing a website, in which the authors field is empty, or citing a forthcoming paper, in which the year information is missing.

There are existing datasets used for benchmarking citation extraction, such as the Cora dataset [25], and the CiteSeerX dataset used in [13]. However, they usually do not provide original PDF files, which are input of GROBID. Therefore, we build the ground-truth ourselves by manually extracting citations from a list of selected papers.

In the first experiment, we randomly select 100 scholarly documents from CiteSeerX, and compare the number of automatically parsed reference items against the real numbers. The comparison results are tabulated in Table 4. The results indicate that while both ParsCit and GROBID have similar numbers of under-parsed papers, GROBID tends to over-parse citations.

Table 4: ParsCit vs. GROBID on number of parsed items.

Parser	Under-parse	Match	Over-parse
ParsCit	28	44	28
GROBID	29	26	45

Here, "under-parse" is the number of papers for which the automatic-parsed items is less than real numbers. The meanings of "number-match" and "over-parse" are then self-explanatory.

A comprehensive comparison of citation extraction quality requires a large number of manually extracted references from papers, and an accurate automatic title and author matching algorithm. Another issue is to align reference items in automatic parsed items with manually extracted results. These are beyond the scope of this paper. Here, we focus on 12 papers that cover typical citation styles.

For most conference and journal papers for which the plain text is well extracted, the differences between GROBID and ParsCit are minor. However, there are a few cases that can make their parsing results different. First, it seems that GROBID does not work well with reference items starting with acronyms, e.g., [KH12]. When parsing these references, it tends to combine two consecutive titles to form a wrong title. On the other hand, if a paper contains an appendix following the reference section, ParsCit treats the appendix as part of the last reference string. Both of them make mistakes when there are control characters inserted among the textual content. Basically, both GROBID and ParsCit applies conditional random field, which explains in most cases they work nearly equally well. The differences are likely due to the training samples.

One of the advantages of ParsCit is that it extracts citation context, which are snippets of text around the location where a reference mention. GROBID does not extract citation context directly, but the output TEI file contains labels in the text body that correspond to labels of reference items. Thus it is still possible to extract citation context by extra coding. For this project, we use ParsCit as the default citation extracting tool.

2.7 Table and Figure Extraction

One of the improvements of the new framework compared with the current one is the integration of table, figure, and algorithm EMs. There are couples of existing open source tools to extract tables and figures, such as the table extraction algorithm proposed by [22], and the figure extraction algorithm proposed by [8]. Recently, [9] developed a package called PDFFigures, that can extract figures and tables in one command. An independent evaluation suggests that it takes 0.1–0.2 seconds on each page and can achieve an F1-measure of 90%. Therefore, it is arguably one of the best open source figure/table extraction tools. Besides, it integrates figure and table extractions so we do not need to implement individual EMs.

PDFFigures is executed by calling a compiled binary with output directories for figures and JSON files. We set a timeout of 20 seconds in case the program hangs when it is pro-

cessing an unusual file. The output JSON files are concatenated into a single JSON file by the wrapper. The extracted figures and tables are saved as PNG files into the output folder for the paper.

2.8 Algorithm Extraction

Algorithm extraction automatically extracts pseudo-codes and reference context from papers, and generates synopsis for each pseudo-code. There is a handful of open-source software dedicated on algorithm extraction. The most recent and successful one was developed by [4] and [27] (hereafter *AlgEx*).

The core program is written in Perl with a Java wrapper. It takes a text file, and the output directory as input parameters, and writes an XML file to the supplied directory. We set a timeout of 20 seconds.

3. EXTRACTION

One of the goals of this framework is to abstract away the logic of coordinating and running the extraction process and let the user focus on defining the actual logic for extracting metadata.

The framework was implemented in Python. Python has been shown to be more usable than Perl [28], and is widely used in academics. One of the challenges with Python is the Global Interpreter Lock (GIL), present in Python's standard implementation, i.e., cPython. The GIL prevents the Python interpreter from executing more than one thread at a time. As a result, the GIL essentially prevents CPU-bounded jobs from being able to gain performance benefits through multi-threading. However, multi-processing does not suffer from the GIL limitations because it allows multiple instances of Python interpreters. Therefore, the multi-processing is used instead of multi-threading in order to meet the performance goals for the framework.

The code is split up into four main modules. The `core` module is responsible for the main functionality of the framework. It runs the overall extraction process. It contains the `ExtractionRunner` class, which is where users of the software library configure and execute extractions. The `core` module internally manages all parallelization. The `runnables` module lets users define their own runnables. It contains the base `Runnable` class, which is inherited by the `Extractor` and `Filter` classes. To define extractors and filters, a user extends either the `Extractor` or `Filter` class and overrides its `extract` or `filter` method, respectively.

The `utils` module contains various useful tools. For example, the function called `external_process` provides users with an easy way to start, pass data to, and get the result from an external process while specifying a time-limit for the process. Finally, the `log` module contains logging handlers that the extraction framework uses by default. The full source code for the extraction framework is available online on Github [1].

4. EXPERIMENTS

To validate the framework, we run it on a dedicate virtual server with 16 logical cores of Intel Xeon E5649 @ 2.53GHz. The server has 8GB of RAM.

The first experiment is set to run 1000 PDF documents randomly selected from the existing digital library with a single process. The goal of this experiment is to investigate the overall performance of the framework in terms of runtime for each module and system resource consumed. The average runtime for one document is about 10 seconds. Figure 4 illustrates the distribution of the average runtime across all EMs. Among all submodules, the algorithm extraction is currently the slowest followed by the citation extraction. The header extraction is the fastest, taking only 0.16 seconds per document on average. If we do not perform algorithm extraction, it takes on average 5 seconds to process one PDF file using a single thread. The framework handles all exceptions produced by specific extraction tools and runs without a hang or a crash. The average CPU usage is less than 15% with peak CPU usage of 55% happening in only two spikes. The runtime distribution depends on the actual extraction tools. The numbers reported here give a guideline in terms of which EMs are opted in for time sensitive jobs.

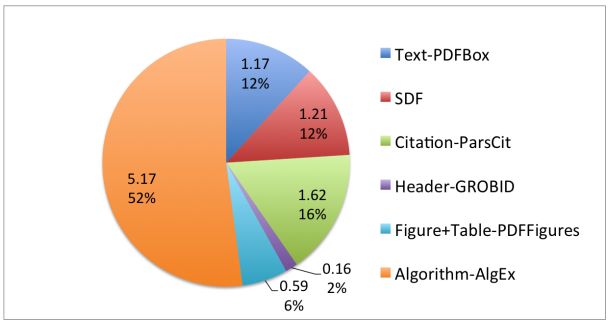

Figure 4: Average runtime (per document in seconds) distribution of EMs.

In the second experiment, we run the framework on the same dataset by varying the number of processes (#process). The goal is to investigate how system resources (CPU, memory) and the average runtime for each PDF change with #process. Figure 5 shows that the CPU usage increases linearly starting with about 13% when #process is less than 6. It then asymptotically increases to around 90%, and keeps steady even #process increases. Clearly, CPU is fully used with the rest of 10% used by the operating system. The memory, however, increases very slowly starting from 56% (4582MB) and it is only used by less than 70% when #process= 17. The average runtime per document (normalized to the single thread runtime) decreases quickly in a power-law fashion when the #process is less than 6, and then asymptotically reaches a limit around 14.3% (1.3 seconds) with respect to the single thread runtime.

The coherence of the CPU usage and average runtime implies that the default implementation of the framework is CPU bounded. The CPU usage does not follow a strict linear relation out to 16 processes (the total number of cores). We believe this is due to resources used by specific extraction tools as they may contain internal processes using multiple cores and memory cache, which stacks up more overhead with more processes. The slow increase of memory indicates that the framework leaves a negligible footprint in the memory, so most of its memory is occupied by library modules.

[1] `https://github.com/SeerLabs/new-csx-extractor`

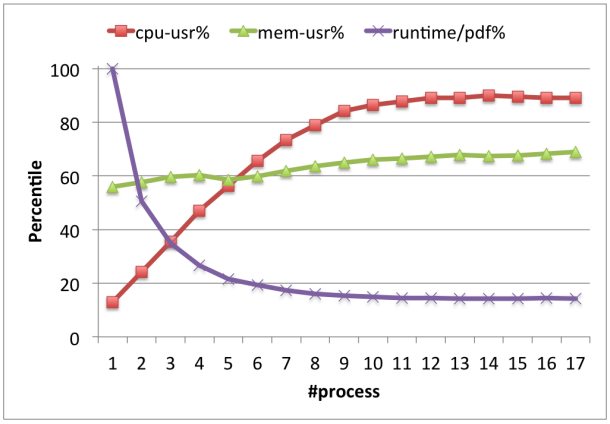

Figure 5: Overall performance with multiple processes.

5. APPLICATIONS

The PDFMEF can be viewed as a knowledge capture framework by itself, as it is capable of encapsulating runnable knowledge capture tools and organizing results. Because of its usable, maintainable, modular, and scalable features, it can be applied to many knowledge capture projects.

In case of the CiteSeerX digital library search engine, which provides free access to six million academic documents, while metadata of this project has been frequently requested and widely used in various projects, the metadata quality has an urgent need to be improved due to mistakenly extracted titles, authors, and citations. There are also a fraction of mis-classified documents, i.e., non-academic classified as academic. This digital library hosts a crawl database of more than 21 million PDF files. It would significant improve metadata quality by running PDFMEF on the big repository and build a knowledge search engine out of the extracted entities using the automatic knowledge indexing technique [14]. It can be a valuable platform for authors to quickly locate existing experimental results and related work.

It has been long proposed that the next generation of search engine should be semantic, e.g., [17, 15]. The current solution by Google uses a knowledge graph with billions of entities gathered from a wide variety of sources. These entities are then logically linked to answer user questions. Actually, recent works have attempted to develop systems to extract simple knowledge statements from plain text [10] or from science textbooks [6]. A slight modification of PDFMEF can fit such a system into the framework and use plain text as input for knowledge extraction. Likewise, it would be useful to build a semantic academic search engine with a large knowledge base populated by entities extracted from academic documents. AllenAI has launched a project called *Semantic Scholar* and published a number of works on knowledge extraction, e.g., [11]. The PDFMEF framework can be an appropriate and efficient solution for such projects to obtain a collection of metadata and paper knowledge entities.

6. SUMMARY AND FUTURE WORK

We have developed PDFMEF, a framework for extracting multiple knowledge entities within scholarly documents. It consists of the following modules: a plain text extractor using PDFBox, a scholarly document filter, a header extractor using GROBID, a citation extractor using ParsCit, figures and tables extractors using PDFFigures, and an algorithm extractor. It is designed to be easily used, maintainable, modular, and scalable. Except for the full text extraction and header extraction modules, all modules are pluggable, so users can easily substitute the default extraction tools with their preferred alternatives. We have tested it on a dedicated server with 16 cores @2.53GHz. It takes on average 10 seconds per document to run all default extraction tools with a single thread and a minimum of 1.3 seconds for launching 10 processes. The framework leaves a very small footprint in the memory. This is a first step in constructing a knowledge base for a scholarly ontology that uses scholarly big data.

Future work will be to build a semantic scholarly ontology at scale. This could also be extended to work on other scholarly related documents, e.g., scholar resumes, slides or syllabi. Integrating the system into the Hadoop or Spark framework would significantly boost system efficiency.

7. ACKNOWLEDGMENTS

We thank Zhaohui Wu for useful discussions and gratefully acknowledge support from the National Science Foundation.

8. REFERENCES

[1] Pdflib tet. http://www.pdflib.com/products/tet/. Accessed: 2015-05-12.

[2] Poppler. http://poppler.freedesktop.org. Accessed: 2015-05-12.

[3] Tabula. http://tabula.technology. Accessed: 2015-05-13.

[4] S. Bhatia, S. Tuarob, P. Mitra, and C. L. Giles. An algorithm search engine for software developers. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 13–16, New York, NY, USA, 2011. ACM.

[5] C. Caragea, J. Wu, K. Williams, S. D. Gollapalli, M. Khabsa, and C. L. Giles. Automatic identification of research articles from crawled documents. WSDM 2014 Workshop on Web-scale Classification: Classifying Big Data from the Web, 2014.

[6] V. K. Chaudhri, B. E. John, S. Mishra, J. Pacheco, B. Porter, and A. Spaulding. Enabling experts to build knowledge bases from science textbooks. In *Proceedings of the 4th International Conference on Knowledge Capture*, K-CAP '07, pages 159–166, New York, NY, USA, 2007. ACM.

[7] S. Choudhury, P. Mitra, A. Kirk, S. Szep, D. Pellegrino, S. Jones, and C. L. Giles. Figure metadata extraction from digital documents. In *12th International Conference on Document Analysis and Recognition*, ICDAR '13, pages 135–139, 2013.

[8] S. R. Choudhury, S. Tuarob, P. Mitra, L. Rokach, A. Kirk, S. Szep, D. Pellegrino, S. Jones, and C. L. Giles. A figure search engine architecture for a chemistry digital library. JCDL '13, pages 369–370, 2013.

[9] C. Clark and S. Divvala. Looking beyond text: Extracting figures, tables, and captions from computer science paper. AAAI 2015 Workshop on Scholarly Big Data, 2015.

[10] P. Clark and P. Harrison. Large-scale extraction and use of knowledge from text. In *Proceedings of the Fifth International Conference on Knowledge Capture*, K-CAP '09, pages 153–160, New York, NY, USA, 2009. ACM.

[11] P. Clark, P. Harrison, N. Balasubramanian, and O. Etzioni. Constructing a textual kb from a biology textbook. In *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction*, AKBC-WEKEX '12, pages 74–78, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[12] T. Consortium. *TEI P5: Guidelines for Electronic Text Encoding and Interchange 2.3.0*. TEI Consortium, 2013.

[13] I. Councill, C. L. Giles, and M.-Y. Kan. Parscit: an open-source crf reference string parsing package. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, may 2008. European Language Resources Association (ELRA).

[14] I. G. Councill, C. L. Giles, H. Han, and E. Manavoglu. Automatic acknowledgement indexing: Expanding the semantics of contribution in the citeseer digital library. In *Proceedings of the 3rd International Conference on Knowledge Capture*, K-CAP '05, pages 19–26, New York, NY, USA, 2005. ACM.

[15] O. Etzioni. Search needs a shake-up. *Nature*, 476(7358):25–26, 08 2011.

[16] D. FERRUCCI and A. LALLY. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10:327–348, 9 2004.

[17] R. Guha, R. McCool, and E. Miller. Semantic search. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 700–709, New York, NY, USA, 2003. ACM.

[18] H. Han, C. L. Giles, E. Manavoglu, H. Zha, Z. Zhang, and E. A. Fox. Automatic document metadata extraction using support vector machines. In *Proceedings of the 3rd ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '03, pages 37–48, 2003.

[19] M. Khabsa and C. L. Giles. The number of scholarly documents on the public web. *PLoS ONE*, 9(5):e93949, May 2014.

[20] P. Larsen and M. von Ins. The rate of growth in scientific publication and the decline in coverage provided by science citation index. *Scientometrics*, 84(3):575–603, 2010.

[21] M. Lipinski, K. Yao, C. Breitinger, J. Beel, and B. Gipp. Evaluation of header metadata extraction approaches and tools for scientific pdf documents. In *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '13, pages 385–386, New York, NY, USA, 2013. ACM.

[22] Y. Liu, P. Mitra, C. L. Giles, and K. Bai. Automatic extraction of table metadata from digital documents. In *Proceedings of the 6th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '06, pages 339–340, New York, NY, USA, 2006. ACM.

[23] L. D. Lopez, J. Yu, C. N. Arighi, H. Huang, H. Shatkay, and C. Wu. An automatic system for extracting figures and captions in biomedical pdf documents. *2013 IEEE International Conference on Bioinformatics and Biomedicine*, 0:578–581, 2011.

[24] P. Lopez. Grobid: Combining automatic bibliographic data recognition and term extraction for scholarship publications. In *Proceedings of the 13th European Conference on Research and Advanced Technology for Digital Libraries*, ECDL'09, pages 473–474, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] K. Seymore, A. McCallum, and R. Rosenfeld. Learning hidden markov model structure for information extraction. AAAI'99 Workshop on Machine Learning for Information Extraction, 1999.

[26] B. Sun, P. Mitra, C. Lee Giles, and K. T. Mueller. Identifying, indexing, and ranking chemical formulae and chemical names in digital documents. *ACM Trans. Inf. Syst.*, 29(2):12:1–12:38, Apr. 2011.

[27] S. Tuarob, S. Bhatia, P. Mitra, and C. L. Giles. Automatic detection of pseudocodes in scholarly documents using machine learning. ICDAR, pages 738–742, 2013.

[28] L. Wang and P. Pfeiffer. A qualitative analysis of the usability of perl, python, and tcl. In *Proeedings of the tenth International Python Conference*, 2002.

[29] J. Wu, K. Williams, H.-H. Chen, M. Khabsa, C. Caragea, A. Ororbia, D. Jordan, and C. L. Giles. Citeseerx: Ai in a digital library search engine. In *The Twenty-Sixth Annual Conference on Innovative Applications of Artificial Intelligence*, IAAI '14, 2014.