

KONLA Documentation

Contents

1. Introduction	3
2. Deployment Guide	4
2.1 Running the project locally	4
2.2 Deploying the project online.....	5
3. User Manual.....	6
3.1 How to use KONLA?	6
3.2 Using KONLA via user interface	6
3.3 Using KONLA via backend API.....	7
4. Project Dependencies	7
5. KONLA Architecture.....	8
5.1 Technology Overview	8
5.2 Communication in Examples	9
5.2.1 User accesses KONLA Web-Based User Interface	9
5.2.2 User uses KONLA Backend API.....	9
5.3 KONLA Architecture Diagram	10
6. Backend	11
6.1 Backend Structure and Control Flow.....	11
6.2 API.....	12
6.2.1 Upload (binary&url)	13
6.2.2 Upload (start).....	14
6.2.3 Whole Summarisation.....	15
6.2.4 Partial Summarisation	16
6.2.5 Keywords Extraction	17
6.2.6 References	18
6.2.7 Metadata.....	19
6.2.8 Metrics	20
6.3 PaperProcessor (NLP).....	21
6.3.1 PaperProcessor Initialization & PDF-to-text conversion	21
6.3.2 Computation of Text Metrics	21

6.3.3 Extraction of Meta Data.....	22
6.3.4 Extraction of References.....	22
6.3.5 Keyword Extraction through Word Frequency	22
6.3.6 Section Extraction	23
6.3.7 Whole Paper Summarization	23
6.3.8 Partial Paper Summarization	24
7. Frontend	24
7.1 Frontend Structure	24
7.2 Useful Commands and Frontend Configuration.....	25
8. Legal Considerations	25
9. Possible Improvements & Future Work	26

1. Introduction

Project Title: Knowledge Organisation through Natural Language Analysis (KONLA)

Last Updated: 27.03.2022

Authors and their emails (alphabetically):

Bartosz Grabek bartosz.grabek.20@ucl.ac.uk

Suraj Kothari suraj.kothari.20@ucl.ac.uk

Minyi Lei minyi.lei.20@ucl.ac.uk

Client: Lacibus Ltd.

Client Representative: Dr Chris Harding, CEO Lacibus Ltd.

Summary:

This is the documentation for KONLA (Knowledge Organization through Natural Language Analysis) project which has been developed between Oct 2021 and Mar 2022 by UCL Students as part of COMP0016 Systems Engineering module in partnership with Lacibus Ltd. represented by Dr Chris Harding. The project aims to help researchers to speed up the process of reading through research papers with the use of natural language processing techniques.

GitHub Repository: <https://github.com/hzlmy2002/konla>

2. Deployment Guide

2.1 Running the project locally

What you will need

- Docker installed (<https://docs.docker.com/get-docker/>)
- Python

Setup

Firstly, download project .zip folder and unpack it in desired directory or clone the project from the GitHub repository.

To run the webserver, you will need to have **Docker** installed on your machine. Navigate to the project folder, open the terminal and follow these steps to start the server locally:

Step 1 Build the docker image

```
docker build -t konla .
```

Note: this may take some time depending on your machine and internet connection

Step 2 Run the server using the safe port 443

```
docker run -p 443:443 -d konla
```

Step 3 Open the browser, navigate to <https://127.0.0.1>

You should be able to see the application interface now and use the UI or the underlying API to analyse your research papers. For information on how to use KONLA, read [User Manual](#).

Note: Some browsers may block the website due to security issues. For example, Google Chrome may show a message "Your connection is not private", but you can still click "Advanced" and then on the label "Proceed to 127.0.0.1 (unsafe)" to use the app. (see Figure 1)

Using project locally again

If you want to run the webserver again and did not remove the docker image from the first build, you can skip step 1 and follow steps 2 and 3 only.



Your connection is not private

Attackers might be trying to steal your information from **127.0.0.1** (for example, passwords, messages or credit cards). [Learn more](#)

NET::ERR_CERT_AUTHORITY_INVALID



To get Chrome's highest level of security, [turn on enhanced protection](#)

Hide advanced

Back to safety

This server could not prove that it is **127.0.0.1**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

[Proceed to 127.0.0.1 \(unsafe\)](#)

Figure 1 Overcoming browser block

2.2 Deploying the project online

(TBA)

3. User Manual

3.1 How to use KONLA?

You can use KONLA in **two** different ways:

- 1) If you are a researcher looking for **a tool to analyse your paper**, you can use our intuitive user interface (see Using KONLA via user interface).
- 2) If you are a researcher or developer who wants to **make use of the technology behind KONLA in their software or other setting**, you can use our backend API (see Using KONLA via backend API)

Please, see the Project Demo Video [link here] which includes a short tutorial on how to use the service.

3.2 Using KONLA via user interface

Navigate to the website hosting KONLA service. You should see a container titled "Upload research paper".

Selecting features

Select the features of the system you want to use by checking the corresponding boxes. You can also click select all to select all of the features at once.

Uploading document & starting analysis

After you selected the features you should upload your .pdf paper. You can do that by entering a URL link of a paper and clicking "Analyse URL Paper" or you can upload your local .pdf file using the grey area at the bottom of the container. Drag and drop your file or click "browse" to select a file from your local file system. After you selected one click the "Analyse Uploaded Paper" button to start the analysis process (see Figure 2).

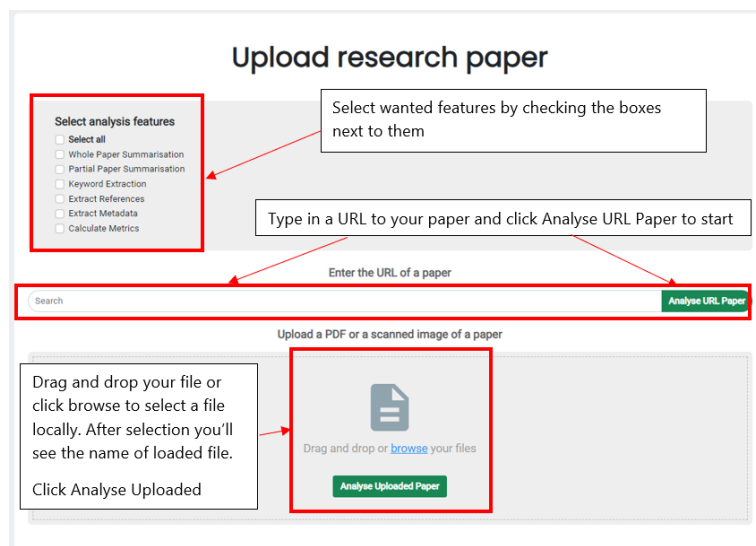





Figure 2 Using KONLA UI

You will see a new view after a few seconds. The features you selected will appear on the left side and they will be color-coded:

-  Yellow – Your feature is still being run in the background. Please wait.
-  Green – Your feature has finished. You can click on the block to see the results.
-  Red – An error occurred while trying to run this feature. Try again or report an issue.

Note: some specific feature blocks can show yellow for a long time. It is not an error. Some specific features such as summarization may just take longer than expected.

3.3 Using KONLA via backend API

As long as the KONLA webserver is running, you can query the API using appropriate endpoints. These are described in detail on GitHub repository in [doc/Endpoint.md](#) as well as in [Section 6.2 API](#) in this document.

4. Project Dependencies

List of main dependencies

- Python 3.9+
- Docker

List of libraries (requirements.txt)

- PyPDF2
- spacy==3.2.0
- redis
- Django
- requests
- django-cors-headers
- uwsgi

Other tools used

- nginx - a web proxy
- supervisord – client/server system for controlling processes on UNIX-like OSs
- npm - default package manager for JavaScript runtime environment Node.js
- redis - used for caching uploaded files

List of language models used

- en_core_web_trf (spacy model)

All the dependencies are available for free and on an open-source license.

5. KONLA Architecture

5.1 Technology Overview

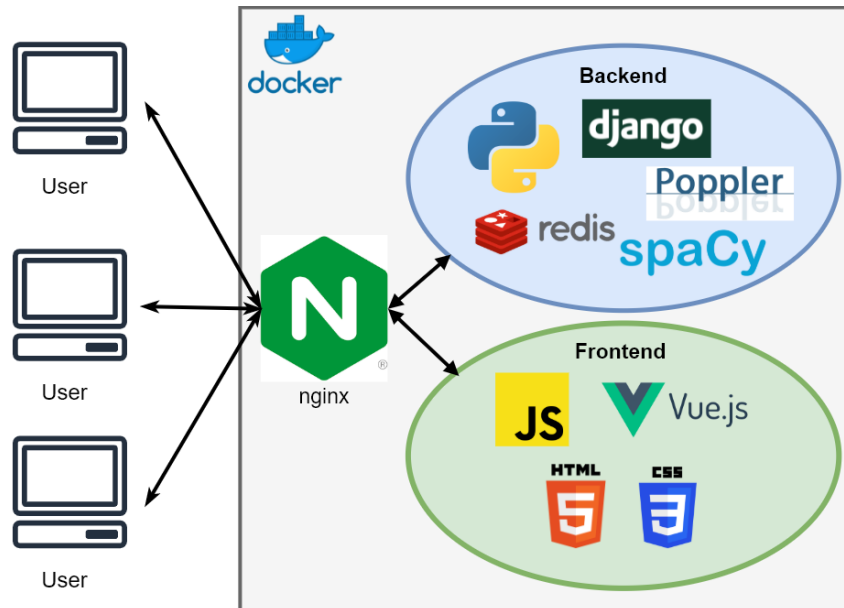


Figure 3 KONLA Technology Overview

KONLA was developed using Docker to unify and simplify the development process and easily manage the project dependencies. The whole project is packaged within a single docker container that is based on Ubuntu version 20.04. Please, refer to [Dockerfile](#) for initial configuration details.

The system comprises of two parts: the backend and frontend which are inside a single docker container, yet they are separated from each other (see Figure 3). We use Nginx webserver/proxy to establish communication between the backend and the frontend. This simplifies the application architecture and the development process for possible future work or upscaling.

The backend was developed using Python Django framework. The backend also uses Poppler to extract plain text from .pdf files, Redis for caching frequently analysed papers and spacy for Natural Language Processing (NLP) tasks. More information on Backend in [Section 6](#).

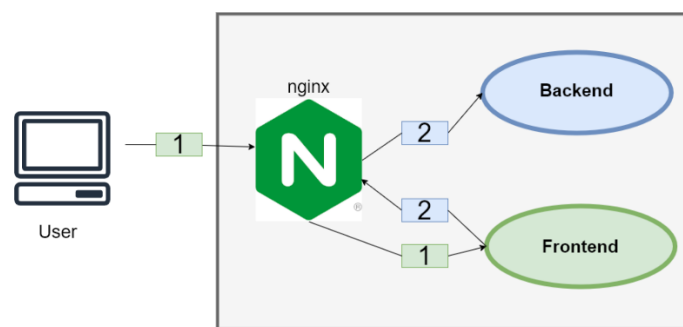
The frontend was developed with the use of a popular JavaScript framework called Vue.js (version 3) along with standard frontend languages such as HTML5 and CSS. More information on Frontend in [Section 7](#).

5.2 Communication in Examples

This section shows with the use of diagrams the communication between different parts of the system for two example ways of using the app.

5.2.1 User accesses KONLA Web-Based User Interface

User navigates to a website where KONLA is deployed. The web address is fed to nginx which routes it to frontend. The user now sees the frontend (see Figure 4). After choosing settings, uploading the file and clicking the “Analyse” button, the frontend queries analysis results from the backend using Nginx server.

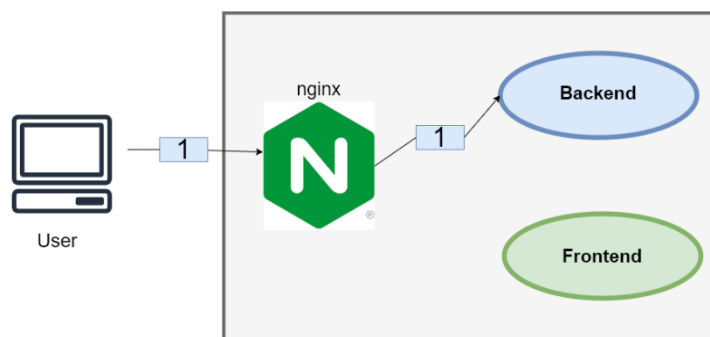


- 1 User enters website via web address
- 2 After document upload, the Frontend continuously queries Backend for analysis results through nginx.

Figure 4 Communication diagram for example use 1

5.2.2 User uses KONLA Backend API

User queries the backend using the KONLA API (see Figure 5). The frontend is not used.



- 1 User queries the Backend using API

Figure 5 Communication diagram for example use 2

5.3 KONLA Architecture Diagram

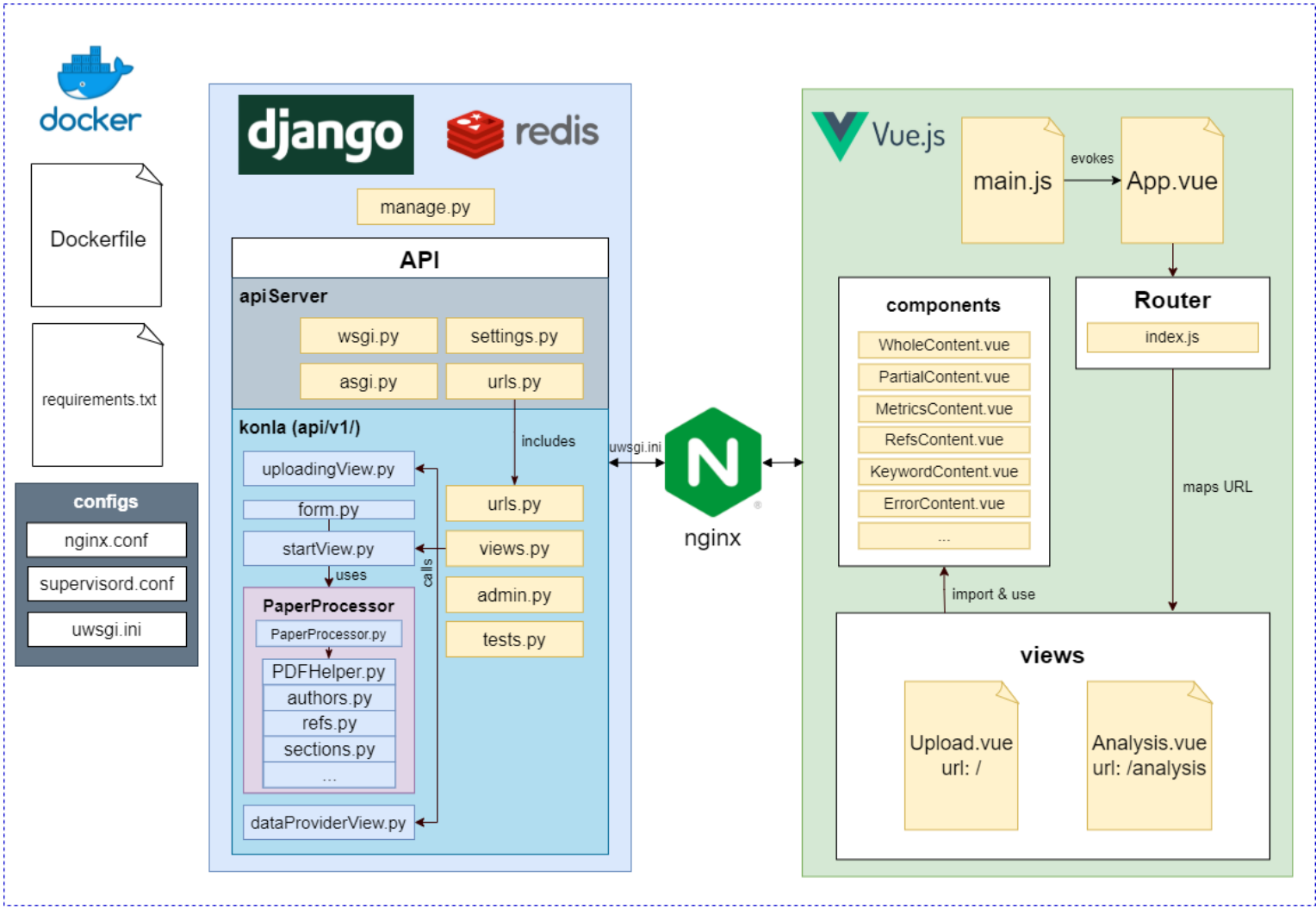


Figure 6 KONLA Architecture Diagram

6. Backend

The backend API is responsible for answering both requests from users using the frontend and those who query the API directly (see [Section 5.2](#)). Nginx communicates with Django Backend API using uWSGI, which is configured to run with a uwsgi.ini file.

6.1 Backend Structure and Control Flow

The backend server consists of two directories: apiServer and konla (see [Section 5.3 Architecture Diagram](#)). The apiServer is responsible for configuring the whole Django server and includes a urls.py file that includes the urls from the second directory konla available under endpoint "api/v1/".

```
# apiServer/urls.py
urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/v1/", include("konla.urls")),
]
```

Both the apiServer and konla follow the structure of a standard Django application. The urls.py in konla directory maps the API endpoints with corresponding views from views.py (functions that receive a web request and return a web response). The Architecture Diagram from Section 5.3 shows standard Django application files as yellow, while the new files added to better separate the functions are marked blue.

Views.py in konla uses three views files to separate different working stages:

- [uploadingView.py](#) which accepts URL or a file as request, converts a pdf to plain text and saves it a temporary directory. It also computes a fingerprint of the text using SHA256. The paper fingerprint and uploaded file path in temporary directory is saved in request.session. The function returns a web response informing about the success of upload, fingerprint and plain text.
- [startView.py](#) which receives a form containing information which features to run. It receives back the checksum (paper fingerprint) from the session and caches the feature selection form with a checksum as an access key. Next, It starts the mainThread. The mainThread initializes a PaperProcessor object which is responsible for all the text processing and analysis tasks. It also creates corresponding threads for each of the features called. This allows to show the results of specific features as they are completed, so that the user does not have to wait for all functions to finish in order to see any results. Each of the threads uses a specific method in PaperProcessor for text processing and analysis. The results are cached using checksum as a prefix and a name of the feature so that it can be read later by dataProvider.py.
- [dataProviderView.py](#) which on given request for a feature, checks the cache to retrieve the results if they are ready. It returns a web response containing current status of the feature (whether it is completed or not) and the results if they are already available.

The next section documents the API endpoints and [Section 6.3 PaperProcessor \(NLP\)](#) describes the structure and implementation of text processing and analysis tasks.

6.2 API

This section is a documentation for the KONLA API. The latest version of the API documentation is always available in the project repository under [/doc/Endpoint.md](#).

Endpoint

<https://konla.thinktank007.com/api/v1/>

Structure

```
endpoint (/api/v1/)
├─ upload/
│   ├── binary
│   ├── url
│   └─ start
├─ summarization/
│   ├── whole
│   └─ partial
├─ keywords/
└─ info/
    ├── refs
    ├── metadata
    └─ metrics
```

The next pages describe each of the endpoints in detail.

6.2.1 Upload (binary&url)

Endpoint: /api/v1/upload

Description: The upload api is used for uploading binaries like pdf or images. The user can also submit an url so that the server can download and analyse it.

Methods: POST for binary, GET for url

Parameters: None

Request:

- binary: POST the content directly, endpoint is /api/v1/upload/binary
- url: GET request. Using http parameter with key "link=".
E.g. "/api/v1/upload/url?link=<https://ucl.ac.uk/cs/shapes.pdf>".
Endpoint is /api/v1/upload/url.

Response:

```
{
  "current_status": 1, # -1 means incomplete, 0 means failed, 1 means completed and success
  "errors": [],
  "messages": [],
  "result": {
    "sha256": "5c061a8549a38daee917491da054ad37c3edf8dd06656c5e0743a1f4dfb42f5a",
    "text": "content of the paper"
  }
}
```

6.2.2 Upload (start)

Endpoint: /api/v1/upload/start

Description: After uploading the paper, the backend needs to know which functionalities will be activated, cookie should contain session id

Methods: POST (in html form)

Parameters (For all the following parameters they are all either 0 or 1, where 0 represents disabled and 1 represents enabled):

- whole: Control whether enable whole summarisation
- partial: Control whether enable partial summarisation
- keywords: Control whether enable keywords extraction
- refs: Control whether enable references extraction
- metadata: Control whether enable meta data extraction
- metrics: Control whether enable metrics calculation

Request:

Request body:

```
whole=1&partial=1&keywords=1&refs=1&metadata=1&metrics=1
```

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {}
}
```

6.2.3 Whole Summarisation

Endpoint: /api/v1/summarisation/whole

Description: The whole summarisation function is used to get summarised paper.

Methods: GET

Parameters: None

Request:

- Access this endpoint directly with cookie that contains session id.

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "whole_summarisation": "Whole summarised paper content"
  }
}
```

6.2.4 Partial Summarisation

Endpoint: /api/v1/summarisation/partial

Parameters: TBC

Description: The partial summarisation function is used to summarise the paragraphs of the paper.

Methods: GET

Request: TBC

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "partial_summarisation": {"section1 title":"summarized text for
section1","section2 title":"summarized text for section2"}
  }
}
```


6.2.5 Keywords Extraction

Endpoint: /api/v1/keywords

Description: This api return a list of most frequent keywords with their occurrences.

Methods: GET

Parameters:

- max: int. This parameter indicates the max number of most frequent key words, should within [1,100]. Default value is 100.
- ignorecase: int. The default value is 0. This parameter indicates whether ignore uppercase. 0 represents not ignore, 1 represents ignore
- extractlemma: int. The default value is 0. This parameter indicates whether extract the lemmas instead of exact words. 0 represents disable, 1 represents enable.

Request:

- get <https://konla.thinktank007.com/api/v1/keywords?max=10&ignorecase=0&extractlemma=0>

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "keywords": {"Hello": 10, "World": 6, "KONLA": 3}
  }
}
```

6.2.6 References

Endpoint: /api/v1/info/refs

Description: This API returns the information of extracted references in the paper.

Methods: GET

Parameters:

- None

Request:

- get <https://konla.thinktank007.com/api/v1/info/refs>

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "refs": ["ref1", "ref2"]
  }
}
```

6.2.7 Metadata

Endpoint: /api/v1/info/metadata

Description: This API returns the metadata from the paper. This includes authors, title, and other file information

Methods: GET

Parameters:

- None

Request:

- get <https://konla.thinktank007.com/api/v1/info/metadata>

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "metadata": {
      "authors": ["Nikhil Parasaram", "Earl T. Barr", "Sergey
Mechtaev"],
      "creator": "Appligent AppendPDF Pro 5.5",
      "producer": "pdfTeX-1.40.21; modified using iText® 7.1.16
©2000-2021 iText Group NV (IEEE; licensed version)",
      "subject": "IEEE Transactions on Software Engineering;
;PP;99;10.1109/TSE.2021.3124323",
      "title": "Title Of Research Paper"
    },
  }
}
```

6.2.8 Metrics

Endpoint: /api/v1/info/metrics

Description: This API returns the metrics calculated from the text. This includes character count, word count, reading and speaking times.

Methods: GET

Parameters:

- None

Request:

- get <https://konla.thinktank007.com/api/v1/info/metrics>

Response:

```
{
  "current_status": 1,
  "errors": [],
  "messages": [],
  "result": {
    "metrics": {
      "wordCount": 1920,
      "readingTime": 120,
      "speakingTime": 243
    }
  }
}
```

6.3 PaperProcessor (NLP)

PaperProcessor directory contains all the code associated with processing and analysis of texts. The [PaperProcessor.py](#) file includes a PaperProcessor object class which is used for managing all of the other files in the directory. The PaperProcessor object is created during execution of [mainThread](#) in [startView.py](#), in other words after the file is uploaded and the analysis starts.

```
./PaperProcessor/
├── PDFHelper
│   ├── PDFHelper.py
│   └── __init__.py
├── PaperProcessor.py
├── __init__.py
├── authors.py
├── refs.py
├── sections.py
├── summary.py
└── wordFrequency.py
```

6.3.1 PaperProcessor Initialization & PDF-to-text conversion

PaperProcessor is initialized with a string parameter `path` which is a path to the file saved temporarily in the backend. After that, PDFHelper opens a subprocess running pdf-to-text conversion. After the conversion finishes, the result is decoded using utf-8, fed into the preprocessing function and saved in text. Next, another PDFHelper's method is used to extract and save metadata. During initialization PaperProcessor also loads and saves the spacy language model and uses the standard spacy pipeline on the text to save it in self.doc attribute for further use.

```
# PaperProcessor/PaperProcessor.py
def __init__(self, path:str) -> None:
    text = self.preprocessText(PDFHelper.pdf2text(path))
    self.metadata = PDFHelper.getMetaData(path)
    self.nlpTRF = spacy.load("en_core_web_trf")
    self.doc = self.nlpTRF(text)
```

The methods of the PaperProcessor reflect the backend API and are used to return corresponding results. The next subsections describe each of the features in more detail.

6.3.2 Computation of Text Metrics

Computation of Text Metrics is contained within PaperProcessor object under metrics method. By text metrics or text statistics we mean reading time, speaking time and word count. The metrics method of PaperProcessor returns a dictionary containing these computed metrics.

```
# PaperProcessor/PaperProcessor.py
def metrics(self) -> dict:
    result={}
    result["wordCount"] = self.wordCount()
    result["readingTime"] = 60*self.wordCount()//238
    result["speakingTime"] = 60*self.wordCount()//140
    return result
```

Note: self.wordCount() is calculated as the number of tokens in doc that are not punctuation marks.

Reading Speed

We assumed an average rate for conversational speech which is about **140** words per minute, according to Tools for Clear Speech between 110-150wpm [1].

Speaking Speed

We assumed an average rate of reading time of 238wpm for researchers according to the study by Marc Brysbaert from Ghent University in Belgium whose work calculated an average reading speed based on 190 of research studies conducted between 1901 and 2019, collectively involving 17,887 participants [2].

[1] Speaking Rate [Internet]. [cited 2022 Mar 27]. Available from:

<https://tfcs.baruch.cuny.edu/speaking-rate/>

[2] Most Comprehensive Review To Date Finds The Average Person's Reading Speed Is Slower Than Previously Thought – Research Digest [Internet]. [cited 2022 Mar 27]. Available from: <https://digest.bps.org.uk/2019/06/13/most-comprehensive-review-to-date-suggests-the-average-persons-reading-speed-is-slower-than-commonly-thought/>

6.3.3 Extraction of Meta Data

Metadata are extracted in two phases. Firstly, PDFHelper uses PyPDF2 to retrieve document information and saves it in self.metadata upon PaperProcessor initialization. The result is then complemented by a search of Author. AuthorParser (from [authors.py](#)) uses spacy's Named Entity Recognition to find the first entities labelled as 'PERSON'.

6.3.4 Extraction of References

Ref object located in [refs.py](#) is responsible for reference extraction. It uses a combination of spacy matcher object for rule based matching and regular expressions to capture the reference region and return a list of references.

6.3.5 Keyword Extraction through Word Frequency

Keyword Extraction is performed in [wordFrequency.py](#). Firstly, a wordlist is created by iterating through the tokens in doc and filtering out whitespaces, punctuation marks, digits, numbers urls and stopwords. Depending on the request parameters, the words appended to the list can be lemmatised and/or made lowercase. For each word in the wordlist a number of

occurrences is calculated. The results are returned with a dictionary containing the word and the number of occurrences of this word in the document.

6.3.6 Section Extraction

Although section extraction is not a feature that can be directly called from an API, it is used by other features such as partial summarization. By section extraction we mean extracting the titles of sections and subsections of the documents and the text regions under that particular headings.

Section Extraction can work efficiently under the following assumptions:

- section and subsection titles are clearly separated from main paragraph text
- section and subsection titles are within a single line
- section and subsection titles follow standard numerical ordering: 1, 1.1, 1.2, ... etc. or 1., 1.1., 1.2., ... etc.
- section and subsection titles follow writing standards, i.e. they are either capitalised or uppercase (stopwords can be lowercase)

Section Extraction is performed in [sections.py](#) in a following order:

1. Spacy Matcher object uses several patterns to distinguish section and subsection titles
2. A round of checking of matches is performed, letter case is taken into account and possible reference section is excluded as there is no point in summarising references
3. Multiple patterns can match to same or overlapping texts. Therefore, another stage is the removal of duplicates and overlap between matches. If there is an overlap between two matches, the one that starts earlier is used, and the second one is discarded. If two matches start at the same position, longest match is used, shorter is discarded.
4. By iterating over a list with section titles, start tokens and end tokens section range is computed for each of the section and subsection match. Furthermore, if a section contains several subsections, its range is computed with regards to the next section, i.e. it includes the text range from each of its subsections. For example, section "2" includes the combined text range of "2.1", "2.2", "2.3", ... up to the start of section "3". The last section is cut off at reference start if reference match was found, otherwise it continues till the end of the text.

6.3.7 Whole Paper Summarization

Note: The current version of whole paper summarization is likely to change.

Whole Paper Summarization is done with the use of Summarizer object in [summary.py](#). It performs extractive summarization on spacy doc (pre-processed text). Firstly, the relative frequency of words in the document is computed. Whitespaces, numbers, stopwords, punctuation marks are ignored. Next, each sentence is given a score which is the sum of relative frequencies of its words. The function returns top `WHOLE_SUMMARY_LENGTH` sentences, where `WHOLE_SUMMARY_LENGTH` is the class attribute of Summarizer and defaults to 10.

6.3.8 Partial Paper Summarization

Note: The current version of whole paper summarization is likely to change.

Partial Summarization is performed similarly to whole paper summarization (see Section 6.3.7), but the calculation of sentence scores and extraction of sentences is performed for each section separately. Another difference is that it returns an ordered dictionary, where the keys are the section and subsection titles and values are summarised contents of this their text regions. Each region summary is made up of SEGMENT_SUMMARY_LENGTH sentences, where SEGMENT_SUMMARY_LENGTH is the class attribute of Summarizer and defaults to 10.

7. Frontend

7.1 Frontend Structure

The frontend follows a standard [Vue 3](#) web application structure generated by vue-cli. The current user interface folder is located in src/frontend/vue-web-app.

```
./vue-web-app/src
├── components
│   ├── ErrorContent.vue
│   ├── Footer.vue
│   ├── Header.vue
│   ├── KeywordsContent.vue
│   ├── LoadingIcon.vue
│   ├── MetadataContent.vue
│   ├── MetricsContent.vue
│   ├── PartialContent.vue
│   ├── RefsContent.vue
│   └── WholeContent.vue
├── router
│   └── index.js
├── views
│   ├── Upload.vue
│   └── Analysis.vue
├── App.vue
└── main.js
```

As in standard Vue application, in [main.js](#) the app is created and mounted to the "#app" div in the html template specified in [index.html](#) in vue-web-app/public directory.

The index.js in router maps the endpoints to the corresponding views. ("/" for Upload.vue and "/analysis" for Analysis.vue).

The views use templated components for displaying different parts of the webpage to create a better and more comprehensible structure.

The user interface consists of two views:

- upload view ([Upload.vue](#)), which is the default page when the service is accessed
- analysis view ([Analysis.vue](#)), which is visible once the research paper was uploaded and analysis features were selected

The upload view retrieves the selected features and the research paper in .pdf and pushes it as a JSON object to the Analysis Vue shortly after the user clicks on “Analyse Paper”. Next, the analysis view is displayed, checks the selected features and sends request to upload the paper from frontend and start selected analysis tasks. Once it receives a confirmation that analysis tasks started, it continuously sends requests for results of analysis to the backend. After it receives a particular feature, the view is dynamically updated.

7.2 Useful Commands and Frontend Configuration

You can test solely frontend by opening terminal, changing directory to `konla/src/frontend/vue-web-app` and using npm commands below.

Project setup

```
npm install
```

Compiles and hot-reloads for development

```
npm run serve
```

Compiles and minifies for production

```
npm run build
```

Lints and fixes files

```
npm run lint
```

Customization

For frontend project customization, see Vue [Configuration Reference](#).

8. Legal Considerations

[TBA]

9. Possible Improvements & Future Work

One of the most difficult decisions during the design of KONLA was the choice between accuracy and performance. This trade-off significantly limited our capabilities of producing a system that could be both fast and accurate.

The performance is dependent on the file size, language model and the algorithms behind the text analysis tasks.

The accuracy is dependent on the text analysis model and algorithms, but it is also limited by the noise in data. The PDF to text conversion works well in simple, formatted texts including standard characters, while richly formatted research papers with mathematical equations generate a lot of noise in the data making it more difficult to derive useful insights.

Another important factor is the language model itself. The more general it is the worse the results can be for papers that include texts in a very specific domain. Ideally, the model should be fine-tuned for analysing research papers from a specific domain of knowledge. While this method requires a lot of domain-specific data for training, it can be rewarding in terms of system accuracy.

Here are some general suggestions on how to improve the existing system:

1. Make the analysis tasks and model domain specific
2. Experiment with other summarization algorithms to increase accuracy
3. Implement new features such as image extraction, search tool, multiple paper analysis, similar paper detection etc.

As the project is well constructed, it is easy to expand its capabilities. In fact, new frontend or new backend can be easily fitted to existing solutions. The backend API version 1 can be copied and improved creating a newer version, for example under "api/v2" endpoint.