Implementing PAKET
A Production-Ready AI enhanced Keyword Extractor

MASTER THESIS

Marlon Weghorn

Submitted on 23 September 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 23 September 2022

License

Erlangen, 23 September 2022

Abstract

Keywords are fragments of the core content of a text and can be used to cluster documents, visualize information or enrich metadata. The extraction process is a well-researched topic in the information retrieval community and existing solutions work well, although they are usually not designed for production, but for scientific experimentation. Being suitable for production means developing for the real world, i.e. anticipating how potential stakeholders can be satisfied best.

This thesis presents a solution for keyword extraction that is intended to be production-ready. Quality building criteria are applied throughout the software development cycle, e.g. by drafting requirements demanding the software architecture to be sustainable and following general principles like design by contract. Separately, a graphical UI is provided, which demonstrates the main functionality and serves as a proof-of-concept.

The result is a deployable application, which not only extracts keywords from text, but also text from files. Over fifteen different MIME types are supported. The keyword extractor ranks keywords by replicating the YAKE! algorithm for 1-grams and filters them in a postprocessing step. Filtering is performed by using language-specific, pre-trained NLP pipelines provided through the spaCy library, and fuzzy matching. Currently, the two languages implemented are English and German, but the design allows the number of languages to be extended upon request. The application enables the integration via web service offering a RESTful-API.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CI Continuous Integration

CLI Command Line Interface

CPU Central Processing Unit

DbC Design by Contract

DRY Don’t Repeat Yourself

IoT Internet of Things

KISS Keep It Small and Simple

MIME Multipurpose Internet Mail Extension

NLP Natural Language Processing

OAI Open Archives Initiative

PoS Part of Speech

PoC Proof of Concept

UI User Interface

UML Unified Modeling Language

1 Introduction

The advancing digitization of global economies is shifting the landscape of conventional business models further into virtual space (World Economic Forum, 2022). For example, more and more people are paying cashless in supermarkets and the first prototypes are already reality (‘Bargeldloses Zahlen nimmt zu’, 2022; ‘Bezahlen ohne Kasse bei Aldi: Sieht so der Supermarkt der Zukunft aus?’, 2022). Automobiles are increasingly equipped with Internet of Things (IoT) devices, feature persistent internet connections and gather large amounts of data, especially for autonomous driving purposes (‘Connected car’, 2022; Tesla Germany, 2022). Companies of different sectors (and sizes) are increasingly entering and making use of a symbiotic relationship with the virtual world, and will continue to do so if they are to remain competitive (Gruhn & von Hayn, 2020). Precisely because of this connection, more data is created than ever before (Reinsel et al., 2018). Analyzing these large datasets can offer unique insights and enable a business to get a competitive advantage.

The following fictional use case gives an illustration and shows the driving direction of the thesis: Imagine a digital strategist wants to find out which business related topics, especially technical ones, have been subject of discussion over a period of time. By associating emails and chat histories with a few keywords that capture the core content, she might better estimate upcoming expenses. This raises the question: how can she get those summarizing keywords in the first place? The **P**roduction-Ready **A**I enhanced **K**eyword **E**x**t**ractor, **PAKET** for short, is supposed to provide an answer.

Apart from functionality, it is of high priority that the probability of problems occurring in production use, unplanned maintenance or arising change costs is small. In this regard, high software quality throughout the entire software development life cycle is to be aimed at. It begins with the requirements that are imposed on the product. One study shows that up to 85% of rework costs are incurred, because the formulated requirements contained errors (Wiegers, 2022).

If the requirements form a well-defined basis, the individual aspects of software design automatically benefit. Nevertheless, a quality standard should be defined for the individual aspects. Indeed, another study from the U.S. (2018) shows that poor software quality has direct economic implications: $2.26 trillion in costs incurred without technical debt and $2.84 trillion with technical debt (Wiegers, 2022).

However, at the end of the day, perfect software can never be written, so it is crucial that the product is good enough to deliver value to the stakeholders.

In the course of development, it has also become apparent that tools from the *AI* world can be very useful. Always looking at software development today with a glimpse on AI can open up non-negligible opportunities to solve problems.

In summary, the thesis attempts to answer the following questions:

- How to extract keywords from documents? How can tools from the AI world help us?
- How to realize a product so that assessments by users and important stakeholders are perceived as positive as possible?
- How to keep future development and maintenance costs within a tolerable scope?
- How to credibly evaluate design and implementation of the requirements?

Accordingly, the thesis consists of the following chapters:

Chapter 1 explains why keyword extraction is an issue, but software development must also be understood from an economic point of view.

Chapter 2 contextualizes the thesis by highlighting related work from research.

Chapter 3 provides a detailed overview of the requirements that are imposed on PAKET.

Chapter 4 describes the responsibilities of architectural elements and the relationships between them.

Chapter 5 deals with the logical structure of individual components.

Chapter 6 looks at the design from a user experience perspective.

Chapter 7 evaluates the design and implementation of PAKET. To achieve this, metrics and analysis techniques will be used.

Chapter 8 summarizes the work of the thesis and makes suggestions on further opportunities for improvement.

General conventions:

- *Italic* font indicates new or emphasized terms.
- Typewriter font references program elements or artifcats.
- The word ‘we’ encompasses the reader and the author.
- Footnotes contain sites for additional information or marginalia that would otherwise impair the reading flow.

2 Literature Review

As already mentioned in chapter one, it is the goal to increase the chance that software in the production use, but also at development time, causes as few problems as possible and simultaneously can fulfill current and upcoming requirements. Research shows that a vast number of approaches and possible solutions exist that can have positive impact on individual phases of the software development life cycle. It is up to this chapter to select, narrow down, and prepare appropriate literature. Special attention is devoted to the comparison of unsupervised, document-only keyword extraction methods.

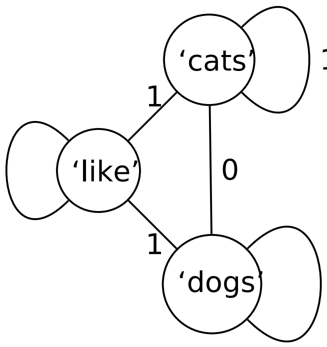
The first work we look at is from Michael T. Nygard which delivered the idea for the thesis to always link software development with the end-state notion of delivering a system to the real world (Nygard, 2018). It conveys an almost paranoid but justifiable view of software design, since it focuses not on what the system should do, but on what it *should not* do. Even though this work is primarily about distributed systems, it offers tools that can also be useful for simple applications, e.g. *stability patterns* like *timeouts*, *steady state* or *let it crash*. Furthermore, if an application is running on a single machine today, it does not mean that it might not be split into microservices, orchestrated via Kubernetes[1], tomorrow.

Related work by Neal Ford et al. aims to achieve the survival of software systems and gives guidance on how to prevent architectural erosion over time (Ford et al., 2017). To a set of selected quality attributes for an architecture, the meta quality attribute *evolvability* is added, which protects all other attributes and further architectural characteristics. It adds the property of *guided*, *incremental* change accross multiple dimensions. Incremental change describes how teams develop software incrementally and how to deploy it. Guided change describes the mentioned protection behavior, which is achieved by so-called fitness functions that assess the integrity of the objectives. Examples of fitness functions are process or architecture metrics, but also unit or integration tests.

[1] https://kubernetes.io

Now that the claim of the thesis is framed in terms of architecture and design, we move to the actual functionality. Three extraction methods and respective papers, which would be, *RAKE* [2] (graph-based), *EmbedRank* [3] (embeddings-based) and *YAKE!* [4] (statistics-based) should provide a rough overview (Rose et al., 2010; Bennani-Smires et al., 2018; Campos et al., 2020).

We begin with the simple RAKE method, which behaves as follows: Text is split into words using word delimiters. This list of words is then used to generate n-gram word phrases using stopwords and phrase delimiters. Afterwards a graph is built using the cross product over the word phrases, resulting in word-word pairs with the number of word *co-occurences* as edges. For example, the sentence 'I like cats, like i like dogs.' would be splitted in the 2-gram words 'like cats', 'like dogs' and 'like', which results in following graph:



Afterwards, a so-called *degree metric* can be calculated, which favors words that occur frequently and appear in long word phrases. It is calculated for each word by summing up the edge values to all other words. If a phrase consists of multiple words, all degrees are summed up to a score. The higher the score, the higher the word relevance. Sticking with the example, the summed degree of 'like cats' is seven and 'like' is five.

Nevertheless, needing $O(V^2)$ space (worst case), where V are the vertices of the graph, is not very efficient, especially not if the text vocabulary is very large.

Mapping words as vectors in continuous vector space (with low dimensions) might remedy the situation. However, the actual reason of these *embeddings* is the semantic relatedness between words, sentences and documents. EmbedRank draws

[2] https://github.com/csurfer/rake-nltk
[3] https://github.com/swisscom/ai-research-keyphrase-extraction
[4] https://github.com/LIAAD/yake

on this and calculates the *cosine similarity* between word phrase embeddings and the document embedding, and ranks them.

However, these word phrase or sentence embeddings are without any context, i.e. they have no relationship to the sentences, paragraphs or documents in which they exist. For this reason, the so-called *Transformer* models are created, which are currently state-of-the-art (Tunstall et al., 2022).

Although vector space models provide good results and are steadily optimized, they still have a relatively high inference time (especially for long texts) and as soon as text becomes domain-specific, it is difficult for them to understand the semantics without training. Moreover, in the wild, one encounters text that has no real semantics, let alone complete sentences. Examples are spreadsheet or powerpoint files. For these reasons, a hybrid solution is implemented in PAKET with light-weight YAKE! algorithm as substructure and word vector models as superstructure. Further details can be found in chapter five.

YAKE! is based on statistical information collected about words, more precisely terms, which are aggregated in a total *term* score. Generally speaking, a term is a word which is used unambiguously (Adler & Van Doren, 2014). An example will illustrate this in context of YAKE!: If the words ‘Keyword’, ‘keyword’ and ‘KEYWORD’ were to occur in a text, they would all be declared as exactly one term, therefore the term frequency would be three. The German words ‘Ei’ and ‘Eis’ look almost the same syntactically, but have completely different meaning and would be declared as two separate terms.

Thus, after breaking down the text into sentences, sentences into n-gram words and n-gram words into 1-gram terms via syntactic rules, five statistical features are collected for each term based on empirically derived assumptions:

1. Casing (T_{case}): Capitalized terms are usually more relevant than lowercase terms.

2. Term position (T_{pos}): Important keywords tend to be found in sentences at the beginning of a document, e.g. in an outline, introduction, or abstract. To avoid that words, appearing at the end of a document, vanish in irrelevancy, a logarithmic smoothing is applied over the sentence indices.

3. Term frequency normalization (T_{norm}): As mentioned above, the term frequency is incremented on each occurrence. It is useful assigning a higher relevance to often occurring terms. To prevent long texts from causing a bias, the term frequency is divided by the sum of the mean of term frequencies and one time its standard deviation.

4. Term relatedness to context (T_{rel}): The higher the number of different terms surrounding a term, the less important they are. The rationale is that terms which are used repeatedly in the same context are of higher relevance.

5. Term different sentence (T_{sen}): Terms that appear in many sentences tend to be of higher relevance. The exception is stopwords, which obviously occur in many sentences.

Still, only the interaction of all statistical features provides a conclusive assessment. These flow into a total term score which is calculated as below:

$$S(t) = \frac{T_{rel} \cdot T_{pos}}{T_{case} + \frac{T_{norm}}{T_{rel}} + \frac{T_{sen}}{T_{rel}}}$$

As can be seen, T_{rel} is mitigating the effect of T_{pos}, T_{norm} and T_{sen} if the term is used in multiple contexts. A capitalized term contributes fully to the equation.

3 Requirements

The success of a project is rooted in the ability to transfer the needs of key stakeholders into a product that satisfies all of those stakeholders (Wiegers, 2022). In other words, the likelihood that a product is suitable for use is greatly increased if the very first step in the transfer process is to translate those needs into *requirements*. If these requirements are good enough, the next phase of development (architectural design) is able to continue and the amount of future rework is reduced. Whether requirements are good enough is determined by its appropriateness of detail, that allows the product to be developed and implemented.

Section 3.1 establishes a common basis for requirements and presents a representation technique called *utility tree*. Section 3.2 tries to achieve shared understanding of the product. In sections 3.3, 3.4 and 3.5 requirements are formulated, which eventually result in the design and implementation of PAKET.

3.1 Fundamentals

Requirements: A broad and comprehensive definition of a requirement is given by Wiegers (2022, p. 8). It is ‘a statement of a customer need or objective, or of a condition or capability that a product must possess to satisfy such a need or objective. A property that a product must have to provide value to a stakeholder’. It is also assumed that requirements are only available in textual form, but in general they can also be proof-of-concepts (PoCs), mockups etc. (Wiegers, 2022). The provided definition lays a common ground for understanding, as requirements are not clearly defined in the literature. It also leads to several requirement types such as *Functional requirements*, *Quality attribute requirements* and *Transition requirements*, types that help to further differentiate requirements.

A functional requirement describes the product’s behavior and conditions under which this behavior occurs and tells the developers *what to build* (Bass et al., 2013). An example would be to have the application sort keywords in descending order by their relevance. However, if only pure functionality is considered, one misses the big picture.

Functionality is irrevocably interwoven with quality, because the way functionality is mapped in software structures determines the support for quality by the architecture. Otherwise put, the shape of an architecture constrained by the quality attributes which are chosen and considered most relevant, is an essential part of building an effective system (See section 4.1). Quality attributes are specified as a measurable or testable property of a system that indicates *how well* the system satisfies the needs of its stakeholders, and the description accordingly as a quality attribute requirement. An example would be, having a low memory footprint or CPU usage so that the application doesn't interfere with other user processes.

A transition requirement describes conditions the product must meet, or activities the project must complete to enable a successful migration from a current state to a future state (Wiegers, 2022). An example would be to present new functionality to an influential stakeholder (e.g. the CEO or customers) via PoC and guarantee the continuity of the project through acquired budget.

As a uniform template, the *FunktionsMASTER* technique, viewable in figure 3.1, is applied for all the requirements (Rupp, 2021).

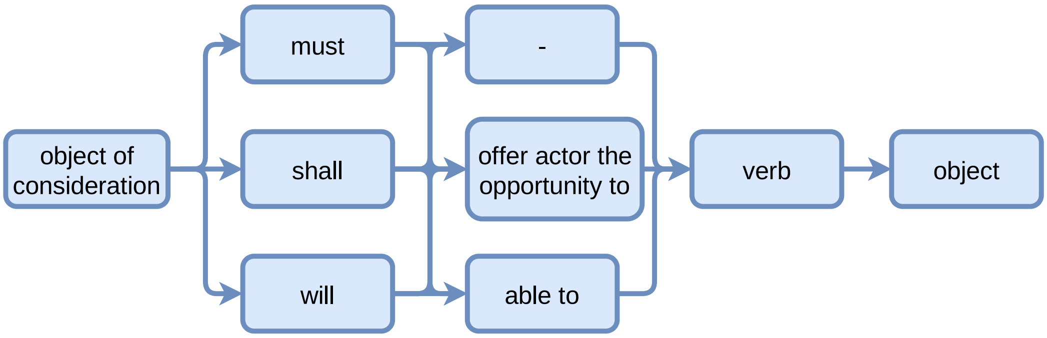


Figure 3.1: Requirements template of *FunktionsMASTER*.

Utility Tree: The goal of a utility tree is to document quality attribute requirements and structure them (Bass et al., 2013). The starting point is a utility root node, which is an expression of the overall quality compliance of the system. Attached to the root node are the still very abstract quality attributes the product is considered to have. Examples are usability, performance or security.

Becoming more concrete, the specific meaning of the quality attributes are refined, considering relevant aspects only. For instance, it can be specified that special attention is drawn to learnability and accessibility when it comes to usability.

Lastly, the quality attribute requirements are collected and prioritized, whereby prioritization can take different forms. In the thesis, the three-level classification over two categories is used, meaning that each of these categories is estimated as either high, medium or low. The first category describes the extent to which the quality requirements imply substantial changes to the architecture and the second category describes the extent to which the quality requirements generate high business value for the key stakeholders. For example, the requirement to improve a hard-to-use User Interface (UI) can have a large impact on both UI design and customer value and could be noted as ‘(H, H)’.

MIME types: A MIME type is the expression of a standardized format, which specifies data as it exists in its original (natural) form (Network Working Group, 2022). It consists of a general top-level type, a specific subtype separated by a slash ‘/’ and individual information. For example, for a text document with ascii characters, text/plain; charset=us-ascii is sufficient for the specification.

3.2 Vision Statement

For researchers or digital strategists who need an automated summarization technique for their pile of documents, PAKET is a keyword extractor for european languages that frames the core content of a document by a specifiable number of keywords, supporting over fifteen different MIME types. Unlike keyword extractors which are hastily built for experimenting, mainly for the scientific community, PAKET provides a solid, interoperable, production-ready solution.

3.3 Functional Requirements

Table 3.1 lists the functional requirements that must, should or will constitute PAKET.

Functional Requirement	**ID**
The system must be able to extract keywords from documents.	F-REQ-1
The system must be able to extract keywords using a unsupervised, statistics-based method.	F-REQ-2
The system must be able to extract keywords independent of domain or document corpus.	F-REQ-3
The system must be able to extract plain text content from MIME types listed in table 3.2.	F-REQ-4
The system will be able to extract plain text content from MIME types xml/html, text/calendar, application/epub+zip, and application/vnd.ms-outlook.	F-REQ-5
The system must be able to extract keywords from English and German documents.	F-REQ-6

The system must be able to extract keywords from documents whose languages originate from european countries.	F-REQ-7
The system will be able to extract keywords from documents whose languages originate from non-european countries.	F-REQ-8
The system must be able to extract keywords from plain text content.	F-REQ-9
The system must be able to extract keywords from files referenced by one or multiple file paths.	F-REQ-10
The system shall be able to extract keywords from files referenced by one or multiple directory paths.	F-REQ-11
The system must be able to return results as a dictionary, where file paths are keys and lists of extracted keywords are values.	F-REQ-12
The system must not be able to (recusively) extract keywords from directories referenced by directory paths.	F-REQ-13
The system must ignore files whose MIME type is not supported.	F-REQ-14
The system must ignore documents whose language is not supported or couldn’t be identified.	F-REQ-15
The system must ignore documents with more than 10^7 characters.	F-REQ-16
The system must ignore documents if any other errors arise during the extraction process.	F-REQ-17
The system must offer users the opportunity to use a Command Line Interface (CLI).	F-REQ-18
The CLI must be able to print the results to stdout as a json string.	F-REQ-19
The CLI must enforce the number of extractable keywords to between one and one hundred.	F-REQ-20
The CLI will offer users the opportunity to adjust the maximum number of characters a document is allowed to have.	F-REQ-21
A keyword must be taken from the document itself.	F-REQ-22
A keyword must be a condensed form of relevant content of part of a document.	F-REQ-23
A keyword must be a 1-gram word.	F-REQ-24
A keyword will be a 2-gram or 3-gram word (keyphrase).	F-REQ-25

The relevance of keywords must have been proven by a well-researched extraction algorithm.	F-REQ-26
The relevance of keywords must be measured by how well they are accepted by key stakeholders.	F-REQ-27

Table 3.1: Functional requirements with identifiers.

Extension	MIME type
csv	text/csv, application/csv
html, htm	text/html
rtf	text/rtf
txt	text/plain, text/x-tex
tex	text/x-tex
doc	application/msword
docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
pdf	application/pdf
odp	application/vnd.oasis.opendocument.presentation
ods	application/vnd.oasis.opendocument.spreadsheet
odt	application/vnd.oasis.opendocument.text
xls	application/vnd.ms-excel
xlsm	application/vnd.ms-excel.sheet.macroEnabled.12
xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet

Table 3.2: MIME types that must be handled by PAKET.

3.4 Quality Attribute Requirements

The quality attribute requirements, as listed in Table 3.2, are intended to set the scope concerning architecture and design for PAKET. There are obviously tradeoffs in their importance, for example it is crucial, regarding architecture and business value, that PAKET is interoperable by being reachable via HTTP requests as well as offering a standard CLI. However, to have building blocks with clear responsibilities may be of little interest to a customer or product manager, but shapes the architecture substantially.

Quality Attribute	Attribute Refinement	Attribute Requirement	ID
Interoperability	Exchanging Information via Interfaces	The CLI must offer users the opportunity to extract keywords via terminal. (H, H)	A-REQ-1
		The CLI must offer users the opportunity to extract keywords via web server. (H, H)	A-REQ-2
Modifiability	Modularity	The system shall be understandable bottom-up, because the software structure is broken down into meaningfully related units. (H, L)	A-REQ-3
	Hierachization	The system shall be understandable top-down, because the architecture is free from cycles. (H, L)	A-REQ-4
	Pattern consistency	The system shall be understandable top-down, because complex structures are deduced by design patterns. (H, L)	A-REQ-5
Extensibility	Adding new languages	The system must be expandable by new languages on request, within one working day. (H, H)	A-REQ-6
Performance	Throughput	At normal load, the system shall be able to return 1, 10, 100 keywords for 1MB files in less than 15, 25, 50 seconds. (M, M)	A-REQ-7

	Ressource Utilization	The system shall work under user disk constraints of minimum 32GB. (L, M)	A-REQ-8
		The system shall have no more than 3GB peak RAM usage. (L, M)	A-REQ-9
Portability	Platform Dependencies	The system must be deployable on current *Linux* and *Windows* distributions via *Docker* containers. (M, H)	A-REQ-10
Transparency	Logging	The system must be able to log human-readable log messages. (H, H)	A-REQ-11
		The system must differentiate between logging levels *info*, *warning*, *debug* and *error*. (M, H)	A-REQ-12
		The CLI must offer users the opportunity to select between logging levels, which are supported by the system. (L, H)	A-REQ-13
Usability	CLI	The CLI shall offer users the opportunity to print the output json string in a human-readable format. (L, L)	A-REQ-14

Table 3.3: Tabular Form of the Utility Tree with identifiers. The utility node is implicitly assumed.

3.5 Transition Requirements

Between the state of development to the final production, there may be an obligation to demonstrate intermediate results of functionality to key stakeholders. To meet this insistence, it may be useful to demonstrate some of the functionality by creating a PoC. Table 3.4 lists the transition requirements of the PoC.

Transition Requirement	**ID**
The PoC shall have a logo.	T-REQ-1
The PoC must offer users the opportunity to insert plain text of their choice into a text box, from which the keywords are to be extracted.	T-REQ-2
The PoC must offer users the opportunity to drag-and-drop a file to a file box, from which the keywords are to be extracted.	T-REQ-3
The PoC must offer users the opportunity to select an upper limit of one to one hundred keywords.	T-REQ-4
The PoC must display, which file extensions are supported by the system.	T-REQ-5
The PoC must display the keywords and weights in a result table, sorted, with the first keyword being the most relevant.	T-REQ-6
The PoC shall offer users the option to delete text in the text box and remove the results table.	T-REQ-7
If a file with unsupported MIME type is inserted, the PoC must display a warning message and continues working.	T-REQ-8
If a text with unsupported language is inserted, the PoC must display a warning message and continues working.	T-REQ-9
If a file with unsupported language is inserted, the PoC must display a warning message and continues working.	T-REQ-10

Table 3.4: Transition requirements with identifiers.

4 Architectural Design

With the list of requirements from chapter three, we are prepared for the next step, namely, sketching out the architecture. Good architectural decisions are to be embeded early by considering principles that increase the chance of getting a production-ready system. For this reason the system is built upon *modularity*, *hierarchization*, and *pattern consistency*, architectural principles which are part of a *sustainable software architecture* as described by Lilienthal (2020). Additionally, the maxim *KISS* will be followed (Starke, 2020).

It should be noted that architectural design differs from other design aspects insofar that while architecture tries to find a compromise between all quality attributes, the application of design is an architectural decision in itself, improving maintainability (Toth, 2019). The use of design might even reduce architectural work by keeping the structure of the software solution flexible. Either way, good design reduces the time it takes to read and understand the system internals, thus reducing development time and maintenance costs (Lilienthal, 2020).

Section 4.1 provides an outline of technical debt, sustainable software architecture and KISS. Section 4.2 presents a general overview of the project structure and addresses the integration and deployment of PAKET. Section 4.3 and section 4.4 decompose the packages of PAKET into modules and describe them in context of the aforementioned principles.

4.1 Fundamentals

Technical Debt: Technical debt arises when incorrect or suboptimal technical decisions are made consciously or unconsciously, causing additional expense at later date (Lilienthal, 2020). The following types of technical debt are to be considered: Implementation (code smells), design and architecture, test as well as documentation debt.

Sustainable Software Architecture: Sustainability can be understood as using resources in a way that ensures long-term satisfaction by maintaining the regenerative capacity of systems involved (Brundtland, 1987). An analogy to software development can be provided as follows: The resources development and maintenance time should be used as efficiently as possible over the software lifetime (Lilienthal, 2020). More precisely, the effort to pay off technical debt should be bearable or, in other words, software should be capable of regenerating to the extent that developers can work on it properly. All systems involved (e.g. stakeholders) are supposed to benefit.

To counteract erosion at architecture level, *modularity*, *hierarchization* and *pattern consistency* are to be resorted to. These principles are the logical continuation of human cognition when it comes to better reading and understanding program code.

Modularity exploits *chunking*, the fact that people combine smaller knowledge units (chunks) with little information per knowledge unit into larger, more condensed knowledge units with more information. If the line is drawn to programming, this means that developers combine the lines of program code they read (bottom-up) into higher and higher order knowledge units until an understanding is reached. If the knowledge units are related in a meaningful way, this works out more easily. Based on this insight, the following criteria should be strived for in order to achieve a modular architecture:

1. Building blocks (e.g. packages, modules, ...) are highly cohesive and have exactly one responsibility.

2. Building blocks are encapsulated via interfaces, which are explicit, minimal and delegating.

3. Building blocks are loosely coupled.

Content that is also hierarchically structured and ordered (trees, acyclic graphs) is easier for people to learn, process, but also, once in memory, to recall (top-down). The construction of hierarchies is supported by the fact, that in software systems methods are contained in classes, classes in modules, modules in packages and so on. However, as a condition, the architecture must be designed *cycle free*. This implies that it must not be possible to backtrack to each individual building block via the relationships it has to other building blocks.

In the first instance, our brain is primarily a pattern recognition apparatus that derives so-called *schemes* from situations in our environment. Complex coherences are bundled as schemes in chunks in order to achieve a speed boost in recalling memory contents (top-down). Such a chunk consists of an abstract and a concrete level. The former attributes the relationships it schematically represents and the latter represents the prototypical instances of the scheme. The following example shall symbolize this: The scheme *car* generates abstract notions in us, attributing e.g. that it has a chassis, an engine, a steering wheel, four tires, and so on. Concretely, however, each of us will have different vehicles in mind, which we have stored as prototypes of the car schema.

Schemes can be transferred congruently to patterns of software development. Here it is crucial to use generally known patterns from practice and to implement them consistently.

Keep It Small and Simple (KISS). Albert Einstein is alleged to have said: 'Everything should be made as simple as possible. But not simpler'. This is absolutely true for the development of software systems (Starke, 2020). Systems that are kept simple are easier to maintain, because they are easier to understand and they do not hide potential problems through unnecessary complexity. Nevertheless, problems have their own inherent complexity that cannot be simplified arbitrarily.

4.2 PAKET

4.2.1 Project Tree

Figure 4.1 shows elements of the directory structure and provides a bird's eye view of the project. The project itself is a *GitLab*[1] project, therefore making use of version control. The folder gitlab additionally suggests that a CI pipeline is used which is described in section 4.2.2.

The two folders text_extractor and keyword_extractor encapsulate package-relevant details, such as the package itself as well as dependencies and tests. The *Python*[2] programming language is used for the overall project. This architectural decision arose from the fact that Python is considered standard in the data science and AI field offering many open-source keyword extraction implementations.

The decision to keep the tests next to the package instead of the respective modules is due to the fact that they would otherwise have to be removed with additional effort upon delivery (Krekel, 2022). You will also notice the pyproject.toml file, which has been introduced as a new configuration file to specify build dependencies and to reduce cognitive complexity (Cannon et al., 2022).

Since the final product will be delivered as a *Docker*[3] container, it will be tested in a similar environment beforehand. docker_integration is used to build executable binaries and test them. docker_deploy is intended to preserve the bare application with its dependencies and should therefore be seen as separate.

[1] https://about.gitlab.com/
[2] https://www.python.org/
[3] https://www.docker.com/

```
/
├── docker
│   ├── docker_integration
│   └── docker_deploy
├── gitlab
├── keyword_extractor
│   ├── keyword_extractor
│   ├── tests
│   └── pyproject.toml
├── text_extractor
│   ├── text_extractor
│   ├── tests
│   └── pyproject.toml
├── Makefile
└── ...
```

Figure 4.1: Project tree.

In addition, a Makefile is used to simplify local testing and other administrative tasks.

4.2.2 Continuous Integration

In order to get immediate feedback of the software state after each code commit a CI pipeline is used. Figure 4.2 shows the four states that are passed by.

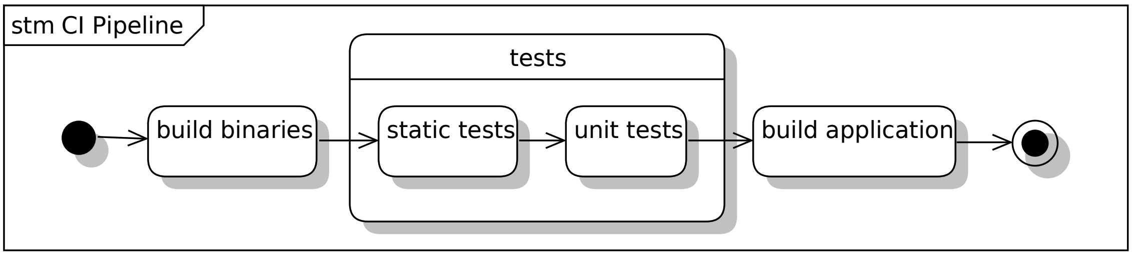


Figure 4.2: A UML diagram showing the different pipeline stages. The abbreviation *stm* stands for Statechart Diagram.

The following static tests are automated:

- Conventions about coding style and docstring structure.
- Error linting in source files.
- Metrics like cyclomatic complexity and code coverage.

- Since Python is a strong, dynamically typed language, a type checker that combines duck typing and static typing.

4.2.3 Deployment

As figure 4.1 already revealed, PAKET is divided into two packages. One handles the extraction of text from document files and the other handles the extraction of keywords from those texts. It is the very first step towards a modular architecture, because the packages have clear responsibilities. Figure 4.3 exemplifies this relationship and we see that the Keyword Extractor depends on the Text Extractor as well as their resulting artifacts.[4] The two artifacts and other dependencies are then bundled into a light-weight Docker container with docker_deploy, ready for delivery.

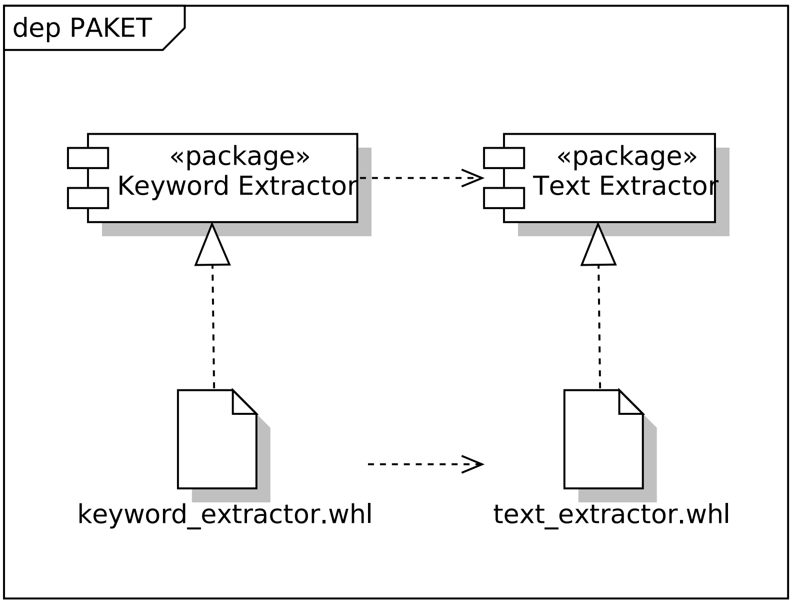


Figure 4.3: A UML diagram of PAKET showing the constituent packages, their relationships and example artifacts. The abbreviation *dep* stands for Deployment Diagram.

[4] A Python wheel is an example instance of an artifact. However, the design is kept as general as possible, so basically any programming language is suitable for implementation. The Python notation is used throughout the thesis.

4.3 Text Extractor

Let's take a closer look at the Text Extraction. Since documents could be analyzed not only in terms of keyword extraction, the package is designed as a library. In figure 4.4 we can see how it is composed of four modules: reader, strategy, logger and dependency and how they depend on each other.

The reader module provides the simple method interface:

```
def text_from_file(path: str) -> str:
    ...
```

The method receives a file path as parameter which points to a file whose raw text content is returned if it is one of the MIME types listed in table 3.2.

It depends on the strategy module, whose name is directly derived from the well-known[5] *Strategy pattern*. This means that different read strategies are selectable at runtime, depending on the document type.

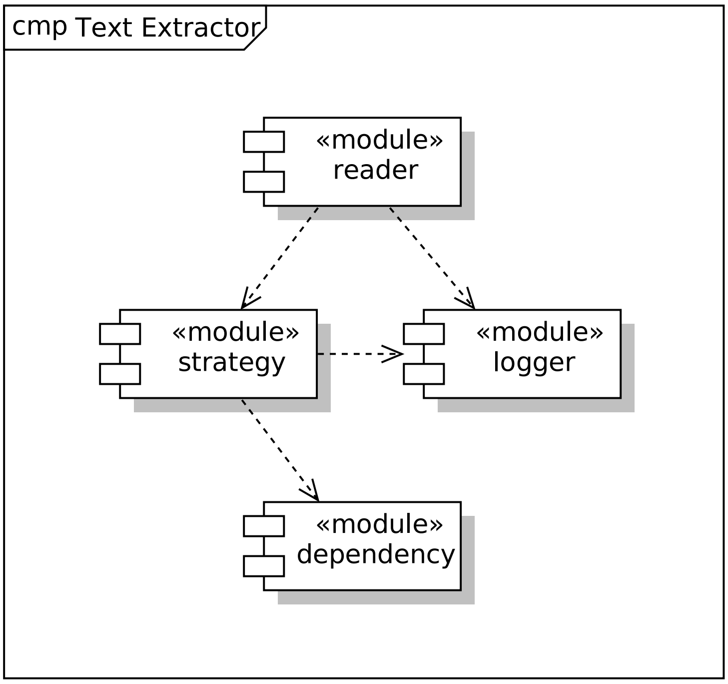


Figure 4.4: A UML diagram of the Text Extractor package showing constituent modules and their dependencies. The abbreviation *cmp* stands for Component Diagram.

[5] Design patterns that are unknown do not help understanding the program.

However, the multiplicity of document types which have to be handled, automatically leads to a multiplicity of required (third party) libraries. Consequently, it may be necessary to resort to native libraries, because those libraries are not available, not mature enough or slow. Native libraries cannot be loaded directly via the package or dependency management tool of the programming language. This is why the dependency module exists. It checks at compile time[6] whether these external dependencies are installed on the host system and aborts if that is not the case (fail-fast).

The logger module can of course be used for inspection purposes. For building a library, it might be a good idea to design the logger to be configurable for the application developer, because the developer should be free to decide what role logging should play in his application (Sajip, 2022).

4.4 Keyword Extractor

Now that we have text in our possession, we continue and extract keywords. Figure 4.5 shows eight modules constituting the Keyword Extractor.

The module cli offers integration points via the CLI, once in form of a classical terminal, once in form of a web server (service) via HTTP requests. If the decision is made in favor of the web server, a *POST request* can be sent that returns one to one hundred keywords for each file listed. Appendix A contains more detailed information on its usage.

Both approaches make use of the keywords module, which offers following method interfaces:

```
def keywords_from_files(paths: List[str], upper_limit: int) \
                        -> Dict[str, List[Keyword]]:
    ...
def keywords_from_text(text: str, upper_limit: int) \
                       -> List[Keyword]:
    ...
```

The first method receives a list of file paths, as well as a number, which indicates the upper limit of keyword objects to be returned.

The first parameter paths refers to one or more files and/or to one or more folders. These paths are resolved non-recursively and made unique via the module of the same name, paths. Among other things, it is to be achieved that if the input contains e.g. two directory paths, both pointing to the same folder, corresponding file paths of the same hierarchy level are taken from the directory path only once.

[6] Import time in python.

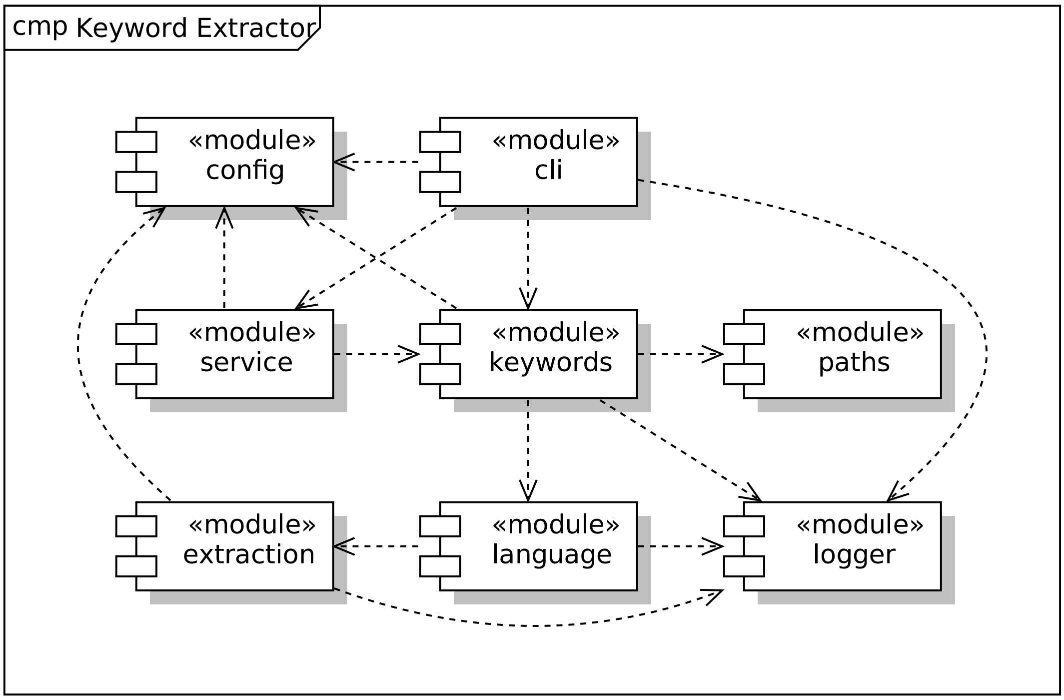


Figure 4.5: A UML diagram of the Keyword Extractor package showing constituent modules and their dependencies. The abbreviation *cmp* stands for Component Diagram.

The second parameter is called upper_limit, because it is practically possible that the text contains fewer words than the required keywords. Accordingly, fewer of them would be returned. The return value is a dictionary that maps the file paths to list of keyword objects. The choice for a keyword object was made because, firstly, it is descriptive and secondly, it allows the client to use only the attributes it needs, e.g. the word, weight or other encapsulated attributes.

The second method does almost the same, but receives a text as first parameter and returns a list of keywords. Because all these peculiarities are not directly recognizable in the method signature itself, they must be included in the interface documentation (not shown).

The language and extraction modules are strongly coupled, because both use the *spaCy*[7] library, whose concepts are explained in chapter five. Nevertheless, these are seperate modules, because the former governs the composition of the NLP pipeline and the selection of the language. The latter cares about the actual extraction and ranking of keywords.

[7] https://spacy.io/

Finally, the modules config and logger. You can see in the diagram that they unite many arrows, which either indicates that the modules are not very cohesive or that they are cross-cutting concerns and thus cannot be easily modularized. The latter is the case.

5 Detailed Design

In this chapter we focus on the logical structure of individual program components. To keep the quality level high, the same guidelines as stated in chapter 4 are followed, but also additional ones. On the one hand, *design principles* such as *DRY* and *DbC* are used. On the other hand *development practices* like extensive testing, coding standards and documentation, which are to be checked by the CI pipeline, are applied (Thomas & Hunt, 2020).

Section 5.1 touches on design principles and provides a detour into the spaCy library, thereby explaining important NLP concepts. How to proceed to determine the similarity of strings is another topic. Section 5.2 covers aspects of the actual keyword extraction process.

5.1 Fundamentals

Don't Repeat Yourself (DRY): This principle is defined as follows: 'Knowledge must have a single, unique, and authoritative representation within a system' (Thomas & Hunt, 2020, p. 31). If adhered to, this has a positive effect on maintainability (Thomas & Hunt, 2020). By having only one single point of truth in the system, the programmer has fewer chunks to retrieve and hold in memory when she makes a change. It should be emphasized that the same code may have different knowledge representations.

Design by Contract (DbC): Contracts between people bind them to explicitly written clauses that provide rights as well as obligations and entail consequences. Contracts between software modules and their routines work the same way (Thomas & Hunt, 2020). Contractually defined preconditions apply to a routine before it is allowed to be called. The routine then guarantees all postconditions and invariants as soon as it returns. If caller and callee fulfill these bindings then they work correctly according to specification. As soon as one of the routines violates the contract, the consequences follow immediately and e.g. an exception is thrown or the program terminates.

spaCy Library: The quality requirements imposed on a product are at least reflected in the libraries used. The NLP library spaCy was chosen, because it was explicitly developed for production use (Explosion AI, 2022). It is characterized by the fact that the API is kept simple, makes comprehensible architectural specifications, but does not cut back on performance. In addition, spaCy makes pre-trained NLP pipelines for English, German and many other languages not only available, but also easy to integrate. The following paragraphs delve into the core concepts that significantly shape PAKET.

The design of spaCy is based on the *Pipe-and-Filter* architectural pattern, which is characterized by the fact that data streams are successively transformed and forwarded (Bass et al., 2013). In spaCy, these text-processing pipelines are called *Languages*, which may consist of several interconnected, reusable, components (filters). Only the so-called *Tokenizer* component is fixed, which segments text into *Tokens* (i.e. words, punctuations, symbols, whitespaces, etc.) and creates a *Doc* object. This Doc object is then potentially passed on and processed by components responsible for other linguistic features. Each Language accesses different data types and implements components in distinct ways.

Language data can be e.g. *stopwords* or *punctuation* rules. Stopwords are words like ‘a’, ‘an’, ‘and’, ‘the’ and so forth, words which carry little useful information in most contexts and are better to be removed. Punctuations are characters like ‘=’, ‘ !’ or ‘ ?’, which are used to split tokens or sentences via regular expressions or rules.

Linguistic features that are relevant for us is the already mentioned *tokenization*, but also other features like *sentence segmentation*, *Part of Speech (PoS) tagging*, *morphology* and *word vectors*. Custom components can also be developed and appended. This is exactly the procedure used for the Keyword Extractor, as can be seen in section 5.2.

Sentence segmentation at sentence boundaries is done by the *Sentencizer*. It seperates sentences on punctuation like ‘.’, ‘?’ or ‘!’.

PoS tagging refers to the process of assigning tags to tokens. Such tags can be for example: A lemma (root word), coarse or fine-grained PoS tags or a conditional, e.g. whether the token is part of a stop list. In PAKET, in particular, coarse-grained PoS tags are needed, which are generated by the *Morphologizer*. Coarse-grained PoS tags have, for example, the identifiers *NOUN* for noun or *ADJ* for adjective.[1] The Morphologizer either generates them using statistical models or derives them by token text and fine-grained PoS tags. The fine-grained PoS tags further differentiate words, language-specifically. For example, an English noun can be further broken down as *NN* (singular or mass) or *NNS* (plural).[2] Fine-

[1] https://universaldependencies.org/u/pos/index.html
[2] https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

grained PoS tags are created by the *Tagger*, which also works with statistical models.

To round everything up, we revisit the concept of word vectors touched in chapter two. Word vectors or word embeddings embody the learned representation of a word, expressed in an n-dimensional vector space. The closer these word vectors lie in space, the closer their relationship. SpaCy offers pre-trained word vector tables for different languages, which are used by components and their statistical models to increase the accuracy of their predictions. The tables contain mainly common words, whose vector representations can be several hundred megabytes in size (Table 7.2). For example, the words 'keyword' and 'language' are included and thus mapped as a vector. A non-existent word is mapped as a vector of zeros and has no effect on the decision making of the statistical models. Such words are called out-of-vocabulary (OOV) words.

Fuzzy Matching: In contrast to exact matching, where two strings must be identical, fuzzy matching is intended to find the similarity, an approximate matching ('Approximate string matching', 2022). Since language is often ambiguous, it is not always desirable to check two strings for exact equality as can be seen in postprocessing, section 5.2.2.

Usually, approximate matching algorithms calculate the 'edit distance between strings A and B ... as the minimum number of edit operations needed in converting A into B or vice versa. Typically the allowed edit operations are one or more of the following: an insertion, a deletion or a substitution of a character, or a transposition between two adjacent characters.' (Hyyrö, 2004, p. 79).

A well-known representative is the *Levenshtein distance*, which performs the first three operations (Hyyrö, 2004). In PAKET, the *Indel distance* is used, which performs only the first two operations. More precisely, the *normalized Indel similarity*[3] is used, which sets the distance in relation to the length of both strings and outputs a comparable value between zero and one, where one means that the strings are perfect matches.

[3] https://maxbachmann.github.io/RapidFuzz/Usage/distance/Indel.html

5.2 Keyword Extraction

5.2.1 Language Integration

Since users can have documents with different languages, it is necessary to react quickly and flexibly to future requirements, offering new languages on request. At the same time, it helps to be able to scale the number of languages, because of the mentioned size of the word vector models.

As with the read strategies of the Text Extractor, we make use of the Strategy pattern. During runtime, it should be possible to get the respective strategy, i.e. the concrete extractor, which was generated at compile time (or import time) by means of the according text language.

Figure 5.1 shows the corresponding inheritance hierarchy. It can be seen that the concrete classes EnglishExtractor, GermanExtractor or XYZExtractor (representative for any language) inherit from the abstract class Extractor. The concrete classes must implement the get_language hook method provided by the abstract class and return the respective spaCy Language without components attached, besides the Tokenizer. When the extractors are instantiated, the template method build_pipeline is called (e.g. via the constructor), which assembles and returns the pipeline.

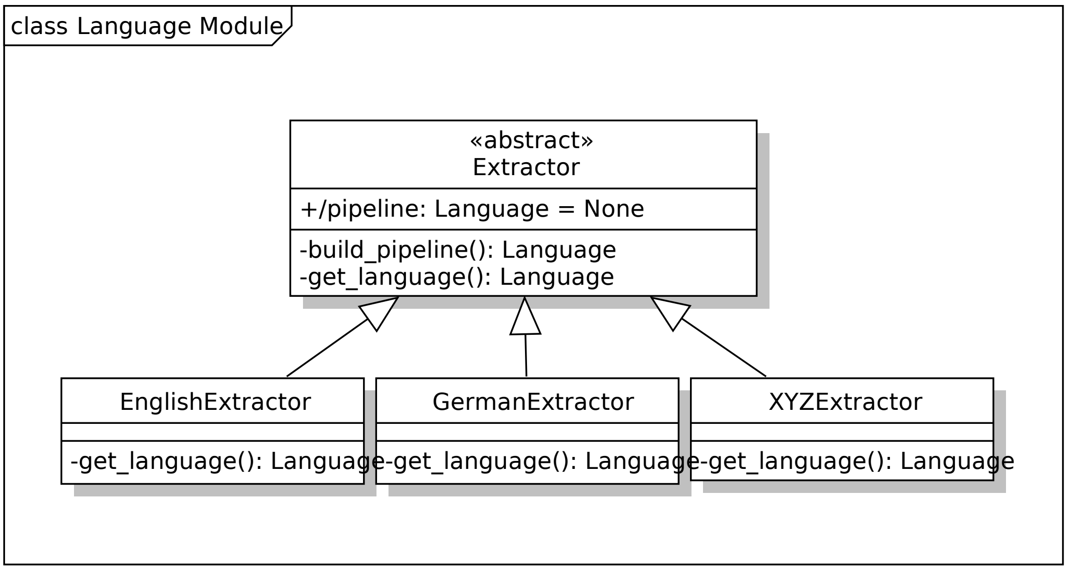


Figure 5.1: A UML diagram of the language module showing the inheritance relationship of Extractor classes. The abbreviation *class* stands for Class Diagram.

The following code excerpt is intended to illustrate the pipeline assembly:

```
@classmethod
def _build_pipeline(cls) -> Language:
    pipeline = cls._get_language()
    ...
    pipeline.max_length = 10 ** 7
    pipeline.add_pipe("sentencizer")
    pipeline.add_pipe("keyword_extractor")
    pipeline.initialize()

    return pipeline
```

As can be seen, the sentence segmentation and keyword extraction components are added to the pipeline. It should be pointed out that those components do not use any statistical models and work by rules or regex. For this reason, the maximum text length that spaCy sets by default can be increased. The limit is changed from 10^6 to 10^7 characters to handle large books, for example. Texts with more characters would be ignored. The limit is usually intended to prevent memory overflows by the models. These allocate temporary memory of about 1GB per 100,000 characters[4].

5.2.2 Processing Pipeline

Figure 5.2 shows the pipeline on a higher level. We will now look at the individual steps of extracting keywords, i.e. Preprocessing, Keyword Weighting and Postprocessing, and assume that tokens and sentences are already attached to the Doc. Roughly speaking, the first two steps handle the implementation of the YAKE! algorithm for 1-gram keywords. The third step builds on this and ranks, filters and collects the keywords.

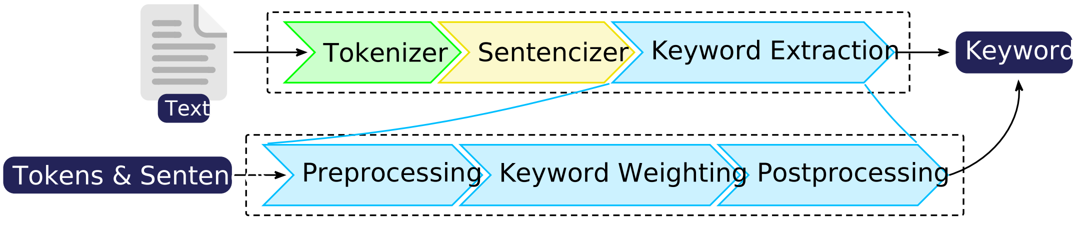


Figure 5.2: Model of a keyword extraction pipeline.

But before that, an explanation of why the decision was made to reimplement YAKE! instead of relying on existing software: The decision in favor of third-party (open source) software can be justified in terms of reusability and division of labor, as the functionality can be very complex and the software is usually

[4]https://github.com/explosion/spaCy/blob/master/spacy/language.py

actively maintained (Winters et al., 2020). Nevertheless, it can sometimes be advantageous to write the functionality in-house, especially if the previous criteria are not or hardly fulfilled.

First of all, from a purely scientific point of view, it makes sense to replicate the results of scientific papers, even more so considering the *replication crisis*[5]

From a business perspective, in-house implementation can mean knowledge gain and control. The acquired knowledge of the person(s) might not only increase the competence, but also radiates into the organizational structure (if knowledge sharing is cultivated). Control over code also means control over quality and flexibility in functionality. For example, if the decision is made to prioritize keyword ranking on other statistical features than those provided by YAKE! this would hardly be possible if externalized. These thought processes are particularly part of a future-oriented and sustainable strategy. Nevertheless, existing software should usually be the first choice, especially if delivery is time-critical.

Preprocessing

During Preprocessing, the first step is to decide whether the tokens of the text are valid or not, and if not, they are skipped. This creates an initial preselection of words that will bias the weighting. A token is considered valid if it is marked as a parsable content, uppercase or acronym. A parsable content is anything that is not an unparsable content and is neither a digit nor a number. An unparsable content is a token that consists of at least two punctuation characters (e.g. an URL or email), that is a combination of one or more digits or alphabetic characters, or that has neither digits nor alphabetic characters.

After that preselection, the valid tokens are mapped to terms. In the best case, this mapping is an unambiguous, many-to-one relationship, i.e. many tokens are mapped to one term. Unfortunately, in reality one does not always encounter unique words. These have differences due to spelling errors, changing spelling reforms or grammar (singular, plural), for example. To obviate this, the tokens are lowercased, punctuations are removed and the lemma is taken. This approach is the most an approximation and a compromise is made between accuracy and performance. Not to leave unmentioned are weaknesses that become apparent from the mapping itself, and thus impacting the statistical approach. Words that are homonyms or homographs are treated free of any context.

At last, information is collected that influences the statistical analysis. If the term is a stopword, it will be marked as such and weighed less during Keyword Weighting. In addition, the frequency of the term is counted and the indices of the sentences as well as the co-occurrences relative to the term are collected.

[5] https://en.wikipedia.org/wiki/Replication_crisis

The following code gives a rough impression:

```
def _preprocessing(doc: Doc) -> List[Term]:
    ...
    for i, sentence in enumerate(sentences):
        for token in sentence:
            if _token_is_invalid(token):
                continue
            term = ...     # Map tokens to terms

            # Information that influences weighting
            term.is_stop = ...
            term.term_frequency += 1
            term.mark = _get_mark(token)
            _append_sentence_index(term, i)
            _append_co_occurrences(term, token)

            terms.append(term)

    return terms
```

Keyword Weighting

In Keyword Weighting or actually Term Weighting, a total weight is calculated for all terms, which is composed of the individual statistical features as described in chapter 2.

Postprocessing

In Postprocessing, the keywords are finally identified. For this purpose, the terms are sorted in ascending order according to their weights (low values mean higher relevance) and each one is checked for certain properties. If it is a stopword, it is skipped right away. Otherwise, a pre-trained NLP pipeline is applied and the term is checked if it is part of the word vector vocabulary. If not, it is a more exceptional term, which is kept. If yes, an additional check is performed whether the (coarse-grained) PoC tag is in a list of allowed PoC tags. If this is not the case the term is skipped.

Finally, redundant keywords are to be avoided. For example, the words ‘Game-of-Thrones’ and ‘gameofthrones’ are prevented from appearing as two different words. To achieve that, a heuristic based on *fuzzy matching* is used.

Firstly, the candidate term is compared to all the keywords, which are already included in the list. The term and the keywords are lowercased, and the normalized Indel similarity between them is calculated. Two words are considered similar if the result is a value greater than 0.9, i.e., there is little leeway when inserting deleting or replacing characters. This value is empirically developed by specifiying a (steadily increasing) list of word pairs, which are considered similar or different. For each integrated language the list of word pairs should be extended and the heuristic might have to be adjusted.

The ‘game-of-thrones’-example would have a value of around 0.928 and thus be rejected, because already contained. The syntactical almost identical, semantically different, German words ‘freiheit’ and ‘freizeit’ with a normalized Indel similarity of 0.875 would both be included.

At first glance, this approach seems to be quite sufficient. Nevertheless there are words (tendentially shorter ones) which are semantically very close, but would both be included. Examples are ‘ice’ and ‘iced’ with a value of 0.857 and the German words ‘fürst’ and ‘fürsten’ with a value of 0.833. Additionally there are semantically different, German words like ‘butter’ and ‘mutter’ with a value of 0.833, which would not be included. As can be seen, there are syntactically similar words, having almost the same normalized Indel similarity, but are semantically very different. Therefore another differentiating factor is needed.

The implemented solution checks in an additional step whether the term is a substring of the keyword or vice versa, and considers them equal if they have a normalized Indel similarity of more than 0.8. The lower bound of 0.8 is chosen, because of the German words ‘ei’ and ‘eis’, which have a similarity of exactly 0.8.

The following code serves as an overview:

```
def _postprocessing(doc: Doc) -> List[Keyword]:
    ...
    for term in sorted_terms:
        if len(keywords) == upper_limit:
            break
        if term.is_stop:
            continue
        ...
        candidate = ...  # Result of NLP pipeline
        if not candidate.is_oov and candidate.pos_ not in
                                        INCLUDED_POS_TAGS:
            continue
        if _term_exists(term.text, keywords):
            continue
        keyword = Keyword(term.text, term.weight)
        keywords.append(keyword)
    return keywords
```

As well as the fuzzy matching approach:

```
def _term_exists(term: str, keywords: List[Keywords]) -> bool:
    ...
    for keyword in keywords:
        ...
        similarity = ...  # Normalized Indel similarity
        if similarity > 0.9:
            return True

        is_substring = ...
        if is_substring and similarity > 0.8:
            return True
    return False
```

6 User Experience Design

A system liked by the users is at least as important as its technical functionality and implementation. If the usability of the system is difficult to understand or cumbersome, this raises the question of its suitability for productional use. To avoid such mistakes, individual *Usability Principles* by Dix et al. (2004) are selected.

Section 6.1 presents those principles. Section 6.2 and section 6.3 show how the CLI and the structure of a logging message can be designed to positively affect usability. Since user experience also includes the opportunity to gain initial experience with the product, the PoC is presented in section 6.4.

6.1 Fundamentals

Usability Principles: Three categories of usability supporting principles are differentiated: *Learnability*, *Flexibility* and *Robustness* (Dix et al., 2004). Only a subset of principles is considered, namely those that also had an impact on the product or PoC.

The first category, learnability, comprises principles that aim to make the (first) system interaction for new users as performant as possible. Herein, the principles *operation visibility* and *familiarity* are of relevance. Operation visibility means that only those operations will be displayed, which the user is actually able to execute, therefore reducing the cognitive load. Familiarity is defined as 'the correlation between the user's existing knowledge and the knowledge required for effective interaction' (Dix et al., 2004, p. 264). The higher the correlation, the easier the system can be used.

Flexiblity refers to the different ways user and system exchange information. In the thesis, a restriction is made by letting user and system communicate via dialogs only. Either the user (user pre-emptive) or the system (system pre-emptive) can *initiate* these *dialogs*. A good system guarantees that the user has as much freedom as possible and that the system only intervenes when it is really necessary.

Interaction with the system implies that the user wants to achieve certain (task-specific) goals with it. Robustness includes principles providing support, such as *browserability* and *defaults*. The former presents users a limited view of the internal system state via the interface, tailored to the task at hand. It basically follows the KISS principle. The latter insists on the availability of default values, which are set within the system or queried and initialized at system startup. They provide guidance to the user and reduce the number of physical actions.

6.2 Command Line Interface

The use of a (user pre-emptive) CLI has the immediate advantage of achieving familiarity, since developers are usually accustomed to its use. It is divided into three task dialogs with the goal of achieving operation visibility and browserability. The first task dialog (Appendix B, fig. 2) provides a general overview of its use, a functional description and a description of the possible options and commands. For example, the four logging levels error, warning, info and debug are provided for logging, with a warning set as default. If the extract command (Appendix B, fig. 3) is chosen, the task dialog looks similar, because the dialogs are kept consistent. The --prettify option is provided, which indents the json string, making it more legible for humans. The run command (Appendix B, fig. 4) offers the option to change the port with --port. In addition, the port range is limited to registered and dynamic ports.

6.3 Logging

In order to make the system more transparent and to debug it more efficiently in case of an error, various demands are made on the structure of the log message: Each message should be uniquely identifiable, human-readable, categorizable, traceable and easily parsable. To achieve this, each log message has the following structure:

[Time][Level][File][Function][Message]

Time is the totally ordered and unique identifier. Level is one of the log levels referenced in section 6.2, File specifies the location in the file system and Function the location in the code, Message contains the concrete concern.

6.4 Proof of Concept

The PoC is implemented as a *streamlit*[1] web application and as already mentioned, acts as a persuasion medium for relevant stakeholders. Two task dialogs are implemented (Appendix C, fig. 5 and fig. 6). To establish an identity to the product (or alternatively to the company), a prototypical logo is created. The overall design is kept minimalistic and modern, it is recommended to use colors matching the product or company and keep it consistent. It is also recommended to limit the maximum text length or file size to avoid overflows. By default, the input option is set to a text and the upper limit of the slider is set to five. In case of an error, feedback should be displayed to the user (system pre-emptive) in the form of a warning (Appendix C, fig. 7, 8 and 9).

[1]https://streamlit.io/

7 Evaluation

Finally, we assess how comprehensively and appropriately the requirements of chapter 3 have been realized.

Overall, an agile approach was conducted, meaning there were many refinement iterations on the requirements, design and evaluation.

Section 7.1 provides an overview of metrics to help to classify the architecture. Section 7.2 provides the specification of the computer, datasets and language models for replicability. In section 7.3, 7.4 and 7.5 PAKET is evaluated, i.e. all requirements are tested if satisfied, partly satisfied or not satisfied.

7.1 Fundamentals

Architectural metrics: Hereafter, metrics are presented which express the structural quality of the architecture in numbers, as described in Dowalil (2020). These can serve as maintainability indicators of a system. It should be emphasized that a measure must remain a measure and might not become the target itself (Goodhart’s law).

The first metric, called *Relational Cohesion*, states that if a module has a much higher number of connections relative to components, it is an indication of poor cohesion. Recommended values range between 1.5 and 4 (Dowalil, 2020, p. 98). It is calculated as follows:

$$\text{Relational Cohesion} = \frac{\text{Number of connections} + 1}{\text{Number of components}}$$

More metrics that are used are those of John Lakos. These are based on the so-called *Depends Upon Values*, which accumulate, per component, the number of components that directly or indirectly depend on it, including the component itself.

With these the *Cumulative Component Dependency* (CCD) can be calculated:

$$CCD = \sum \text{Depends Upon Values}$$

This directly results in the *Average Component Dependency* (ACD) with the following calculation rule:

$$ACD = \frac{CCD}{\text{Number components}}$$

And results in the *Relative Average Component Dependency* (RACD):

$$RACD = \left(\frac{ACD}{\text{Number components}}\right) \cdot 100$$

Lower values mean fewer side effects, which is preferable. To differentiate between architectures, the RACD may be taken, which sets the ACD in relation to the number of components. According to literature, it is recommended to aim for a RACD of less than 25% (Dowalil, 2020, p. 103). The smaller the percentage, the lower the coupling between components. RACD is usually smaller for systems with a high number of components.

The last metric is *Relative Cyclicity*, which indicates the percentage of components involved in a cycle. A value of zero is to be aimed at. The following calculation formula is used:

$$\text{Relative Cyclicity} = \left(\frac{\text{Number of cyclic components}}{\text{Number components}}\right) \cdot 100$$

7.2 Test Environment

Experiments were performed on a computer with following specifications:

- OS: Ubuntu 20.04 focal.
- Kernel: x86-64 Linux 5.15.0-46-generic.
- CPU: AMD Ryzen 7 1700X, 3.4GHz with eight cores.
- RAM: 16GB.
- Disk: Samsung SSD 960 EVO, 500GB.
- Python version 3.9.

Experiments were performed with datasets and pre-trained spaCy models, to be taken from table 7.1 and table 7.2. *Schutz2008* and *SemEval2017* are officially available and extracted from the ACM Digital Library and PubMed Central respective. They include not only documents, but also *gold keywords* annotated by skilled researchers. Because finding a German dataset with gold keywords proved to be a challenge, documents and keywords were scraped from *peDOCS*[1], a repository for educational science literature, using their OAI interface.

Dataset	Language	Size	# Files	MIME types
Schutz2008	en	31MB	1232	text/plain, text/x-tex
SemEval2017	en	2MB	494	text/plain
peDOCS2022	de	405MB	174	application/pdf

Table 7.1: Used datasets for experiments.

Name	Release	Language	Size	Vectors	Licence
en_core_web_lg	3.4.0	en	560MB	514k keys, 514k unique vectors	MIT
de_core_news_lg	3.4.0	de	541MB	500k keys, 500k unique vectors	MIT

Table 7.2: Used spaCy models for postprocessing.

[1] https://www.pedocs.de/

7.3 Evaluating Functional Requirements

Table 7.3 lists the functional requirements, their degree of fulfillment and the reason(s) for that.

ID	Satisfied	Reason
F-REQ-1	yes	Section 5.2 proves that keywords are extracted.
F-REQ-2	yes	Section 5.2 proves that only statistical features and pre-trained models are used.
F-REQ-3	yes	Section 5.2 proves that no domain specific features and no document corpus are used.
F-REQ-4	yes	Section 4.3 proves that plain text content is extracted. Additionally, tests are written for each document MIME type listed in table 3.2.
F-REQ-5	no	It is not yet foreseeable that these MIME types will have to be supported.
F-REQ-6	yes	Section 5.2 proves that English and German languages are integrated.
F-REQ-7	partly	Section 5.2 proves that the integration of new languages is possible. It must be mentioned that the word vector models offered by spaCy do not have commercially usable licenses for every language.
F-REQ-8	no	It is not yet foreseeable that these lanugages will have to be supported.
F-REQ-9	yes	Section 4.4 proves that keywords can be extracted from text.
F-REQ-10	yes	Section 4.4 proves that keywords can be extracted from file paths referencing one or multiple files.
F-REQ-11	yes	Section 4.4 proves that keywords can be extracted from file paths referencing one or multiple directories.
F-REQ-12	yes	Section 4.4 proves that results are returned as a dictionary, where file paths are keys and lists of extracted keywords are values.
F-REQ-13	yes	Tests are written to guarantee that directories referenced by directory paths are not resolved.
F-REQ-14	yes	If there is no read strategy for corresponding MIME type, the file will be ignored. Tests are written to guarantee that.

F-REQ-15	yes	If the language could not be identified or there is no extraction strategy the file will be ignored. Tests are written to guarantee that.
F-REQ-16	yes	Section 5.2 proves that a max length of 10^7 characters is set for the extraction pipeline.
F-REQ-17	yes	If there is an unspecific exception for a file, the file will be ignored.
F-REQ-18	yes	Section 4.4 proves that a CLI is implemented.
F-REQ-19	yes	Section 6.2 proves that the extract command prints the results to stdout as a json string.
F-REQ-20	yes	Section 6.4 proves that the upper limit of keywords can only be between one and one hundred on CLI side.
F-REQ-21	no	The maximum text length of 10^7 might suffice for now. On request, the requirement is able to be implemented within one working day.
F-REQ-22	yes	Section 5.2 proves that terms which result in keywords are directly taken from the document.
F-REQ-23	yes	Section 5.2 proves that relevant content is captured by statistical features.
F-REQ-24	yes	Section 5.2 proves that keywords are 1-gram words.
F-REQ-25	no	It is not yet foreseeable that keyphrases will have to be supported.
F-REQ-26	yes	Implementing the YAKE! algorithm as a substructure results in the benefit of using one of the most cited keyword extractors, which is, according to the peer-reviewed paper, extensively tested.
F-REQ-27	yes	An ad hoc review was conducted via PoC and accepted by the key stakeholders.

Table 7.3: Evaluation of functional requirements.

7.4 Evaluating Quality Attribute Requirements

Table 7.4 lists the quality attribute requirements, their degree of fulfillment and the reason(s) for that.

ID	Satisfied	Reason
A-REQ-1	yes	Section 4.4 proves that the CLI offers users the opportunity to extract keywords via terminal.
A-REQ-2	yes	Section 4.4 proves that the CLI offers users the opportunity to extract keywords via web server.
A-REQ-3	yes	Table 7.5 shows that Relational Cohesion is on average within reasonable range for both packages. RACD is way beyond the recommended 25%, which is due to the small number of components. Section 4.3 and section 4.4 prove that modules are cohesive in general. The modules config and logger are cross-cutting concerns and can't be easily modularized.
A-REQ-4	yes	Table 7.5 proves that the architecture is free from cycles by having a Relative Cyclicity of zero.
A-REQ-5	yes	Section 4.3 and section 5.2 prove that design patterns are used where possible.
A-REQ-6	yes	Section 5.2 proves that languages can be integrated with almost no effort. One working day should suffice.
A-REQ-7	yes	Figure 7.1 proves that 10, 100 keywords for 1MB files are returned in less than 15, 25, 50 seconds.
A-REQ-8	yes	If it is assumed that a word vector table for each language needs a maximum of 600MB of memory and that all 23 languages of spaCy (version 3.4.1) would be used, a total of 13.8GB of disk space would be needed. For the other packages used, no more than 500MB is assumed and is negligible in relation.
A-REQ-9	partly	Figure 7.1 shows that the peek RAM usage is around 2,2GB. As there is variance between different datasets but not between different upper limits, the assumption is that the extracted text as well as the word vector tables contribute to these memory values. Anyway, the variance is no more than 300MiB. Further examination is needed here.

A-REQ-10	yes	Section 4.2.3 proves that the system will be deployed via Dockercontainer, thus working on Linux and Windows systems.
A-REQ-11	yes	Section 4.4 and 4.4 prove that a logging mechanism is implemented. Section 6.3 proves that the logging message is human-readable.
A-REQ-12	yes	Section 6.2 proves that there are the four logging levels error, warning, info and debug.
A-REQ-13	yes	Section 6.2 proves that the default logging level can be changed.
A-REQ-14	yes	Section 6.2 proves that there is an option to prettify the json string.

Table 7.4: Evaluation of quality attribute requirements.

Metric	**Text Extractor**	**Keyword Extractor**
Relational Cohesion	1.25	1.875
CCD	9	31
ACD	2.25	3.875
RACD	56%	48%
Relative Cyclicity	0	0

Table 7.5: Results of the architectural metrics.

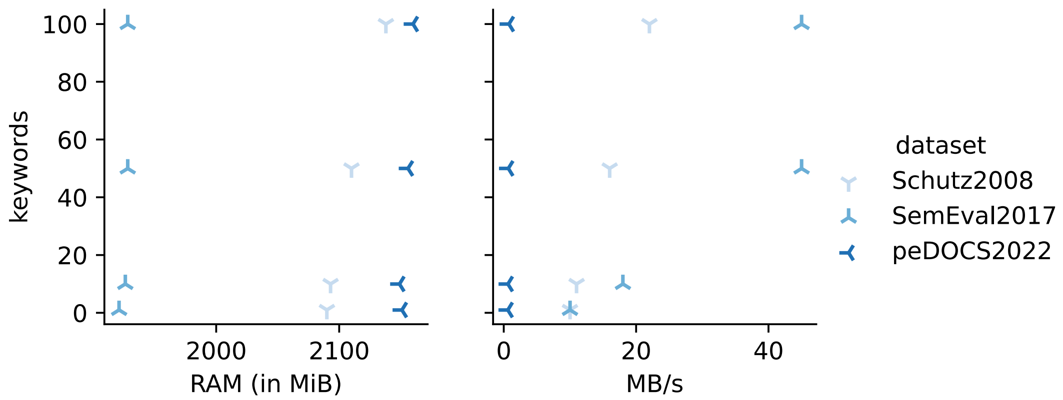


Figure 7.1: Performance measurements.

7.5 Evaluating Transition Requirements

Table 7.6 lists the quality attribute requirements, their degree of fulfillment and the reason(s) for that.

ID	Satisfied	Reason
T-REQ-1	yes	Section 6.4 proves that the PoC has a logo.
T-REQ-2	yes	Section 6.4 proves that users can extract keywords by inserting plain text via text box.
T-REQ-3	yes	Section 6.4 proves that users can extract keywords by drag-and-drop a file to a file box.
T-REQ-4	yes	Section 6.4 proves that users can select the upper limit via a slider.
T-REQ-5	yes	Section 6.4 proves that the file box shows all the supported file extensions.
T-REQ-6	yes	Section 6.4 proves that keywords and weights are displayed in a result table, in sorted order with the first keyword being the most relevant.
T-REQ-7	yes	Section 6.4 proves that there is a clear button to delete text in the text box and remove the results table.
T-REQ-8	yes	Section 6.4 proves that there is a warning message displayed when inserting a file with unsupported MIME type.
T-REQ-9	yes	Section 6.4 proves that there is a warning message displayed when inserting text with unsupported language.
T-REQ-10	yes	Section 6.4 proves that there is a warning message displayed when inserting a file with unsupported language.

Table 7.6: Evaluation of transition requirements.

8 Conclusions

As we have seen, the development of pure functionality is only half of the battle. Software engineering must always be considered from an economic point of view. In the end, it is about developing a product that will be used by other people and should provide value for them. We have seen that PAKET can be such a solution for keyword extraction.

At the same time software engineering should be thought of in a sustainable and future-oriented way, because software must grow with people and adapt to their needs. It makes sense strategically to be able to react efficiently and effectively (agile) to new challenges.

It starts with the requirements, which are clearly defined and forced into a structural corset in order to create a common basis of understanding for all stakeholders. This predefined structure has the effect of avoiding ambiguities in requirements, making them prioritizable, traceable and verifiable. Whether all requirements are actually found or described in sufficient detail depends on the feasibility of the development phases based on them.

Throughout the design phases, quality-building criteria are applied, mainly with the aim of minimizing development time and maintenance costs. The architecture is approved only if it is modular, hierarchical and pattern consistent. The same applies to the implementation. The user experience was not left to chance by complying with Dix’s principles.

We saw that there are different keyword extraction algorithms and decided to use the statistical approach. These words can be extracted from files with over 15 different MIME types. After weighing, the words are passed through a filtering process, thus making use of the power of word-vector models and fuzzy matching.

However, there is still room for improvement. The following list provides suggestions and ideas:

- Supporting additional MIME types.
- Extracting 2-gram and 3-gram words.
- Selecting the maximum allowed length of a document.
- Guaranteeing integration capability for currently non-commercially usable language models.
- Trying out other unsupervised, document-only keyword ranking algorithms, especially state-of-the-art transformer models that weight words according to their semantic relatedness to the document.
- Enriching the result list using words from the document title.
- Enriching the result list using named entities.
- Benchmarking against gold keywords, before and after postprocessing.
- Increasing throughput.
- To be elicited from the stakeholders.

The closing words: A world which is riddled with uncertainty should at least strive for reliable technology. Let's devote our energy ensuring that high quality comes first ... that our software environment is sustainable ... that the symbiosis of the real and virtual world continues to succeed.

Appendices

A Service Description

post **/extract_keywords**

extract representative keywords from files

Parameter

no parameter

Body application/json

```
{
    " paths " : [ " path 1 " ," path 2 " , ... ] ,
    " upper_limit " : number
}
```

Response application/json

200 ok

```
{
    " path 1 " : [
        [ " keyword 1 ", weight 1 ] ,
        [ " keyword 2 ", weight 2 ] ,
        ...
    ] ,
    ...
}
```

500 internal server error

Figure 1: Description of an HTTP POST request extracting a minimum of upper_limit keywords from files, referenced by paths.

B Command Line Interface

```
Usage: paket [OPTIONS] COMMAND [ARGS]...

  Command Line Interface, PAKET.

  A natural language processing tool that extracts keywords
  from files.

Options:
  --log_level [error|warning|info|debug]
                                  [default: warning]
  --version                       Show the version and exit.
  --help                          Show this message and exit.

Commands:
  extract:   Run extractor via terminal.
  run:       Run extractor via web service.
```

Figure 2: Main dialog of the CLI.

```
Usage: paket extract [OPTIONS] [PATHS]... UPPER_LIMIT

  Run extractor via terminal.

  The results will be printed to stdout as a json string.

  PATHS must be directory and/or file paths.
  UPPER_LIMIT must be a number between 1 and 100.

Options:
  --prettify     Pretty print json output.
  --help         Show this message and exit.
```

Figure 3: Extract dialog of the CLI.

```
Usage: paket run [OPTIONS]

  Run extractor via web service.

  The results will be included in the response body
  as a json string.

Options:
  -p, --port INTEGER RANGE  The port where the server will
                            listen for connections.
                            Defaults to 8096. [1024<=x<=65535]
  --prettify                Pretty print json output.
  --help                    Show this message and exit.
```

Figure 4: Service dialog of the CLI.

C Proof of Concept

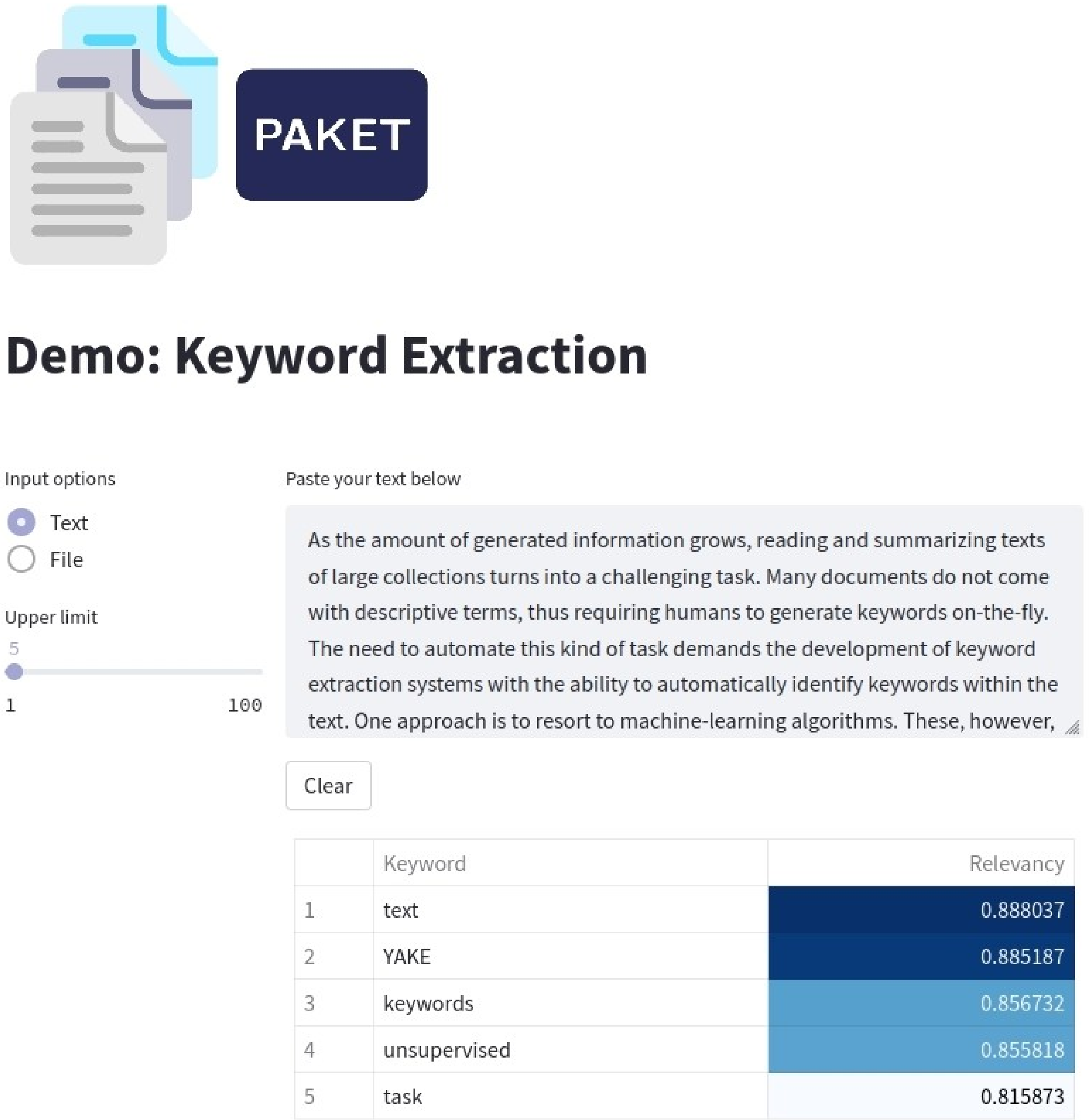


Figure 5: The PoC is displaying the extracted keywords in sorted order. The input is a text.

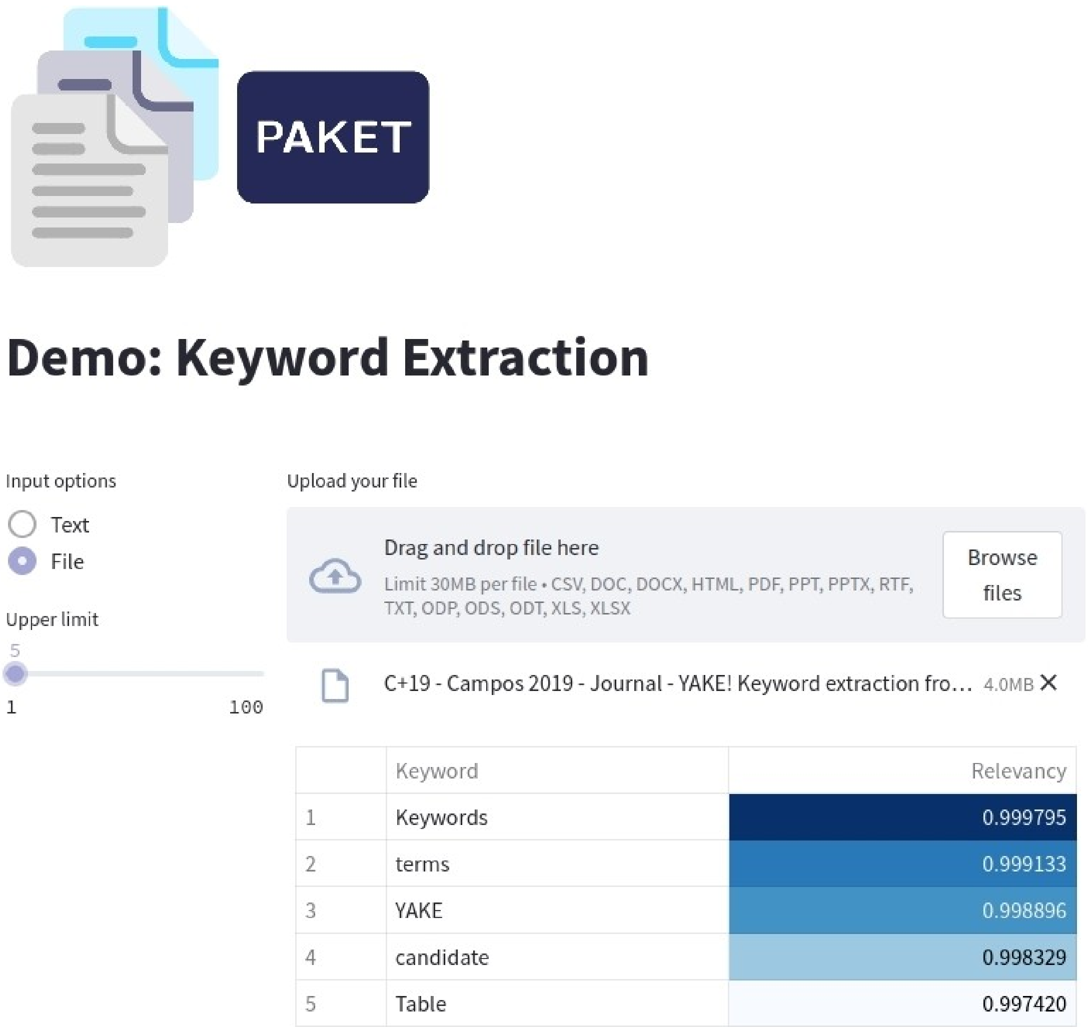


Figure 6: The PoC is displaying the extracted keywords in sorted order. The input is a file.

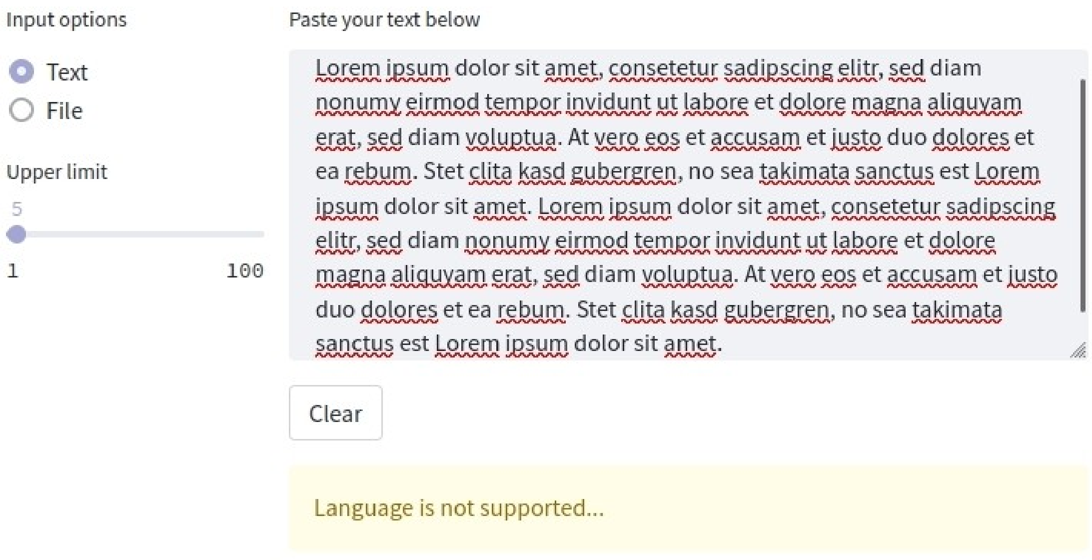


Figure 7: The PoC is displaying a warning message if the text language is not supported.

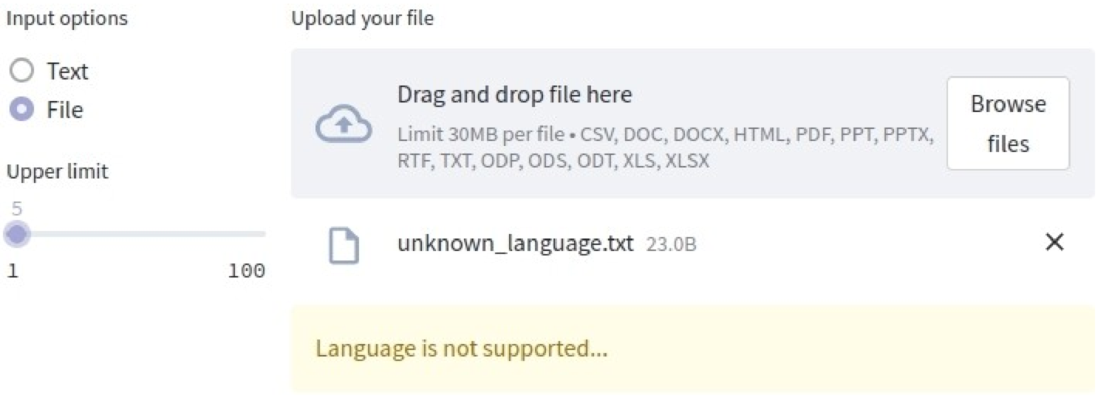


Figure 8: The PoC is displaying a warning message if the file language is not supported.

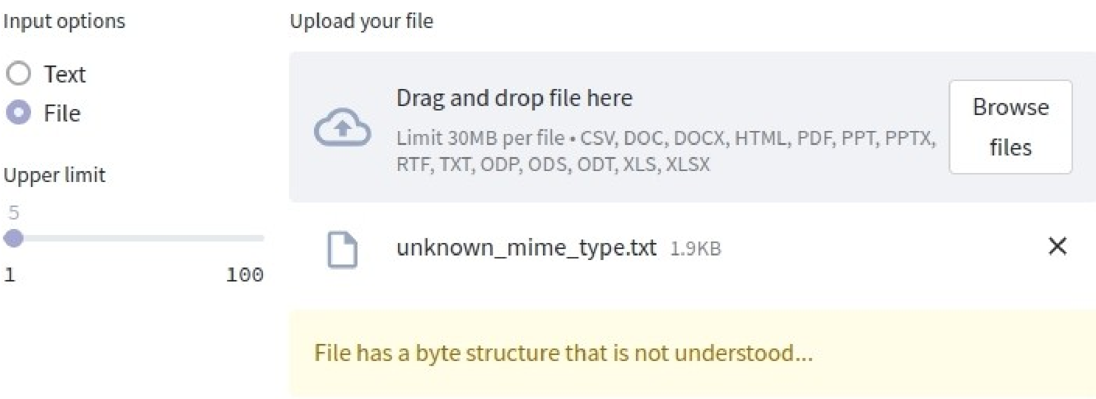


Figure 9: The PoC is displaying a warning message if the MIME type is not supported.

References

Adler, M. J. & Van Doren, C. (2014). *How to Read a Book: The Classic Guide to Intelligent Reading* (1st ed.). Touchstone.

Approximate string matching. (2022). Retrieved September 15, 2022, from https://en.wikipedia.org/wiki/Approximate_string_matching

Bargeldloses Zahlen nimmt zu. (2022). Retrieved August 12, 2022, from https://www.tagesschau.de/wirtschaft/bargeld-bundesbank-studie-zahlungsverhalten-101.html

Bass, L., Clements, P. & Kazman, R. (2013). *Software Architecture in Practice* (3rd ed.). Addison-Wesley.

Bennani-Smires, K., Musat, C. C., Hossmann, A., Baeriswyl, M. & Jaggi, M. (2018). Simple unsupervised keyphrase extraction using sentence embeddings. *CoNLL*. https://doi.org/10.48550/ARXIV.1801.04470

Bezahlen ohne Kasse bei Aldi: Sieht so der Supermarkt der Zukunft aus? (2022). Retrieved August 12, 2022, from https://www.infranken.de/ueberregional/wirtschaft/bezahlen-ohne-kasse-bei-aldi-sieht-so-der-supermarkt-der-zukunft-aus-art-5503638

Brundtland, G. (1987). *Our Common Future: Report of the World Commission on Environment and Development*. United Nations Brundtland Commission.

Campos, R., Mangaravite, V., Pasquali, A., Jorge, A., Nunes, C. & Jatowt, A. (2020). Yake! keyword extraction from single documents using multiple local features. *Information Sciences*, *509*, 257–289. https://doi.org/10.1016/j.ins.2019.09.013

Cannon, B., Smith, N. & Stufft, D. (2022). *Pep 518 – Specifying Minimum Build System Requirements for Python Projects*. Retrieved September 20, 2022, from https://peps.python.org/pep-0518/

Connected car. (2022). Retrieved August 12, 2022, from https://en.wikipedia.org/wiki/Connected_car#Single-vehicle_applications

Dix, A., Finlay, J., Abowd, G. D. & Beale, R. (2004). *Human–Computer Interaction* (3rd ed.).

Dowalil, H. (2020). *Modulare Softwarearchitektur: Nachhaltiger Entwurf durch Microservices, Modulithen und SOA 2.0* (2nd ed.). Hanser.

ExplosionAI. (2022). *Industrial-strength Natural Language Processing*. Retrieved September 12, 2022, from https://spacy.io/

Ford, N., Parsons, R. & Kua, P. (2017). *Building Evolutionary Architectures: Support Constant Change* (1st ed.). O'Reilly.

Gruhn, V. & von Hayn, A. (2020). *Ki verändert die Spielregeln: Geschäftsmodelle, Kundenbeziehungen und Produkte neu denken* (1st ed.). Hanser.

Hyyrö, H. (2004). A Note on Bit-Parallel Alignment Computation, 79–87.

Krekel, H. (2022). *Good integration practices*. Retrieved August 30, 2022, from https://docs.pytest.org/en/7.1.x/explanation/goodpractices.html

Lilienthal, C. (2020). *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen* (3rd ed.). dpunkt.verlag.

Network Working Group. (2022). *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Retrieved September 12, 2022, from https://datatracker.ietf.org/doc/html/rfc2045#section-5

Nygard, M. (2018). *Release It!: Design and Deploy Production-Ready software* (2nd ed.). The Pragmatic Programmers.

Reinsel, D., Gantz, J. & Rydning, J. (2018). The Digitization of the World: From Edge to Core. *IDC White Paper*.

Rose, S. J., Engel, D. W., Cramer, N. & Cowley, W. (2010). Automatic keyword extraction from individual documents. https://doi.org/10.1002/97804706896 ch1

Rupp, C. (2021). *Requirements-engineering und -management: Das Handbuch für Anforderungen in jeder Situation* (7th ed.). Hanser.

Sajip, V. (2022). *Logging HOWTO*. Retrieved August 17, 2022, from https://docs.python.org/3/howto/logging.html#library-config

Starke, G. (2020). *Effektive Softwarearchitekturen: Ein praktischer Leitfaden* (9th e Hanser.

Tesla Germany. (2022). *Fahren in der Zukunft*. Retrieved August 12, 2022, from https://www.tesla.com/de_DE/autopilot

Thomas, D. & Hunt, A. (2020). *The Pragmatic Programmer: Your journey to mastery* (1st ed.). Pearson Education, Inc.

Toth, S. (2019). *Vorgehensmuster für Softwarearchitektur: Kombinierbare Praktiken in Zeiten von Agile und Lean* (3rd ed.). Hanser.

Tunstall, L., von Werra, L. & Wolf, T. (2022). *Natural Language Processing with Transformers: Building Language Applications with Hugging Face* (1st ed.). O'Reilly.

Wiegers, K. (2022). *Software Development PEARLS: Lessons from Fifty Years of Software Experience* (1st ed.). Addison-Wesley.

Winters, T., Manshreck, T. & Wright, H. (2020). *Software Engineering at Google: Lessons Learned from Programming Over Time* (1st ed.). O'Reilly.

World Economic Forum. (2022). *The Digital Transformation of Business: New Digital Business Models*. Retrieved August 12, 2022, from https://intelligence weforum.org/topics/a1G0X000006DIDZUA4