Jakob Weichselbaumer

Concept Extraction from User Comments using NLP Methods - Advanced Practical

Summer Semester 2021

Advisors: Prof. Dr. Barbara Paech, Michael Anders

Institut für Informatik

Ruprecht-Karls-Universität Heidelberg

July 22, 2021

Contents

1. Introduction

1.1. Motivation

Requirements engineering is the process of formulating, documenting, and systematically maintaining software requirements [1]. While conventional techniques of requirements engineering usually involve the exchange of formal design specifications, whitepapers, and feature requests between stakeholders and developers, recent research in the domain of web-apps has highlighted that *app end-users* provide important explicit and implicit requirements feedback whose satisfaction is vital for ensuring the enduring success and popularity of an app.[2] This feedback is almost exclusively in the form of natural language data.

Stakeholders have historically had difficulty integrating these forms of end-user feedback into their requirements engineering process because online feedback is unstructured and noisy, doesn't use a systematic lexicon and terminology to identify features and software components and doesn't tailor its feedback to the needs of any particular type of stakeholder, e.g. project manager vs. a software developer. [2] Finally, the sheer volume of online submissions makes an **automatic analysis** preferable to manual processing, but the lack of tools and techniques in this domain means that this valuable and abundant source of requirements information is often underutilized. [2]

The open-source project OpenREQ [3] is funded by the European Union Horizon 2020 Research and Innovation programme under grant agreement No 732463 which has as the goal, among others, to "monitor[s] the actual software usage, collect[s] stakeholders' and users' feedback (e.g. from social media), aggregate[s] and visualize[s] this information as predictive analytics" [4]. The *feed.ai* system, a part of OpenREQ which was developed in Christoph Stanik's PhD thesis[2], monitors Twitter for user discussion and automatically extracts key sentiment/topic insights.

UVL began as a collaboration between teams at the Universities of Heidelberg and Hannover. The goal is to derive a language to describe the 'view from outside' onto software from the perspective of the end user, analogously to how the Unified Modelling

Language (UML) describes the view of software from 'within'. (Translated from [5]). Feed.UVL is an analysis platform that was built on *feed.ai* as part of the UVL project. UVL is funded by the German Research Society [6].

This report documents the implementation, in the context of the UVL project, of NLP algorithms and their integration into Feed.UVL for automatic extraction and visualization of key concepts from user statements.

2. Feed.UVL

Feed.UVL is a Client - Server Architecture whose server side is implemented in what is known as a "Microservices" (MS) Architecture. This form of program design satisfies business requirements through the use of highly modularized, often containerized components whose implementation details are hidden from the rest of the application.

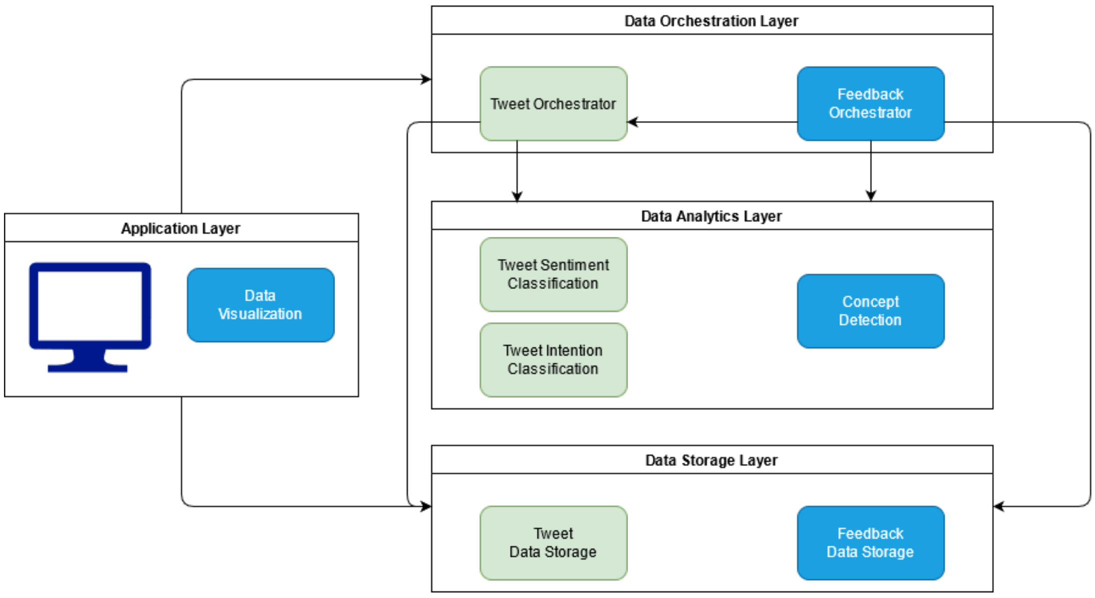


Figure 2.1.: Architecture of the Feed.UVL system.

Figure **??** shows the architecture of the Feed.UVL system: Submodules of the Data Orchestration, Analytics, and Storage layers are implemented as microservices. The algorithms implemented as part of this project are deployed as Docker containers in the Data Analytics Layer.

This type of program design differs from the conventional "Monolithic" architecture in a few ways that are favorable to the Feed.UVL project:

- Reduced code coupling: Microservices don't necessarily share their code and communicate through APIs - this minimizes coupling and reduces communication overhead between developers

- Freedom of implementation: Developers are not bound to any one programming language or framework and are free to use what is best suited to the task at hand
- Modularity: Microservices can be swapped out with another microservice that implements the same API
- Independence at runtime: Isolation of processes means that the failure of one microservice does not have to crash the rest of the program
- Understandability: Microservices "encapsulate a core business capability and the implementation of the service". [7] Functionality is localized, and the learning difficulty for new developers is proportional to the scope of their responsibilities.

To understand the complexity advantage of a microservices architecture over a suboptimally implemented, highly coupled monolith, consider the following analysis:

If we model a program implementing n separately defined business objectives (components) as a graph, we can interpret the code implementing these components as nodes and the total number of references between components as edges. These edges can be modeled as an adjacency matrix A, with the value of $A_{ij}, (i \neq j)$ representing the code complexity required to couple/communicate between components i and j, and A_{ii} representing the complexity of the code implementing component i.

We can interpret the sum $C(A) := \sum_{i=1}^{i=n} \sum_{j=1}^{j=n} A_{ij}$ as a measure for the total complexity of our program.

In a highly coupled implementation, the number of references between components grows *on the order of* the complexity of the components. Let k denote the complexity of the least complex component. Then $\forall i, j \in \{1, ..., n\} : c \cdot k \leq A_{ij}$, where $c \cdot k$ is a constant lower bound on the complexity of coupling between components.

Accordingly, the total complexity $C(A) \in \Omega(k \cdot n^2)$ grows **quadratically** in the number of components.

However, in a loosely coupled program (e.g. microservices), coupling between components remains constant and independent of the complexity of the components. This is because components only need to access an exposed API without knowledge of the implementation details of other components, which internally may grow arbitrarily more complex.

Accordingly, as k grows, the values of elements on the main diagonal of A grow (at least) linearly, with the other $A_{ij} \in \mathcal{O}(1)$, and the total complexity $C(A) \in \Omega(k \cdot n)$ **remains linear**.

In software development practice, this translates to **substantial time and communications savings** when compared to a highly coupled implementation, and lowers the amount of knowledge required to make effective changes to the project's source code.

As will be made clear, the two algorithms share many of the same steps and procedures, allowing a single implementation of certain steps (e.g. tokenization) to be reused, further reducing the total complexity of the implementation.

3. Requirements

The purpose of this practical is to consider the four NLP algorithms mentioned for their suitability for concept extraction from natural language user statements and to implement the most promising approaches.

Core criteria are:

1. Low data overhead - Amount of input data required by the algorithm to produce meaningful results should be low
2. Implementability - Algorithms should be implementable with the help of existing libraries, or else feature low complexity of implementation
3. Performance - Algorithms should feature low runtime complexity and deliver results for complete datasets within seconds or minutes
4. Understandability - Algorithm outputs should be visualizable and interpretable by a human for use in requirements engineering. Extracted concepts should be traceable to the parts of the dataset that produced them.
5. Modularity - It should be possible to add or remove implemented algorithms from the software without disrupting execution
6. Simple Input Format - Inputs should be formatted in .csv form, with the text to be analyzed stored in some particular field (specified by the implementation), with one line per document.[1]

[1] Originally, the tool was intended to support the output of the data analytics software MAXQDA (`https://www.maxqda.de/`), but this requirement was relaxed.

3.1. System Functions

System functions describe important functions of a program on an abstract level. The following system functions capture the essential functionality of an implementation that satisfies the stated requirements:

SF: Upload Dataset	
Preconditions	File f exists
Input	f
Postconditions	Data in f is added to storage and is ready to use
Output	Dataset View
Exception	(E1) f is corrupt, is too large, IO Exception occurs, or other error occurs (E2) f is of the wrong type or format (E3) User cancels the operation
Rules	(R1) Catch errors of type E1 or E2 as they occur. If an error occurs, the user is alerted with a human-understandable error message and the technical error message. (R2) Errors of type E3 are handled by returning to the original view WS1.1

SF: Start Concept Detection	
Preconditions	Dataset d exists, NLP Algorithm a exists and is applicable to the dataset
Input	d, a
Postconditions	None
Output	Algorithm's outputs after being applied to the dataset are visualized
Exception	(E1) Error while parsing file or executing algorithm
Rules	(R1) If an error of type E1 occurs, a human-readable error message is displayed as well as a technical error description

SF: Display Run Result	
Preconditions	Algorithm a has been executed on dataset d, producing result r
Input	r
Postconditions	None
Output	A visualization of r is shown in the UI
Exception	None
Rules	r must satisfy the data format constraints required by the visualization function for a

SF: Delete Dataset	
Preconditions	Dataset d exists in storage
Input	d
Postconditions	Dataset d removed from storage
Output	Snackbar alerting user to deletion or failure status
Exception	None
Rules	None

SF: Display Run Result	
Preconditions	Description
Input	End users perform concept extraction on datasets.
Postconditions	End users upload and store datasets.
Output	End users are able to filter and delete datasets.
Exception	
Rules	

SF: Show Dataset	
Preconditions	Dataset d exists in storage
Input	d
Postconditions	None
Output	d shown in Dataset View
Exception	None
Rules	None

3.2. User Tasks

The user tasks to be supported are listed in table 3.2:

UT1 - Analyze Natural Language User Statements	
Subtask Name	Description
UT1S: Extract Concepts from Datasets	End users perform concept extraction on datasets.
UT1S: Persist natural language data	End users upload and store datasets.
UT1S: Manage Datasets	End users are able to filter and delete datasets.

4. Algorithm Comparison

What follows is an analysis of the suitability of each of the four considered algorithms with regard to the specified requirements. The following essential definitions will be important to understand the definition of the algorithms.

Definition 1 (Token) *A **token** is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. [9] In other words: a word, term, abbreviation, interjection or similar. All tokens are transformed into lower-case form for processing.*

Definition 2 (Lemmatisation, Lemma) ***Lemmatisation** is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by a new token called the word's **lemma**. Derived from: [10] We define the lemma of a non-word token t as t.*

Definition 3 (Concept) *A **concept** of length $= n$ is an ordered list of n lemmas. Derived from: [11]*

For convenience, we introduce the following definition:

Definition 4 (Sentence) *A **sentence** of length n is an ordered list of n tokens derived from an input text, such that the sentence is bound from the left and the right by the end of the input stream or a punctuation mark delimiting a natural language sentence, e.g. '.', '?', '!'.*

We say that a concept c *occurs in* a sentence s if the lemmatisation of s produces a sentence s' such that c is a sublist of s'.

4.1. Finding comparatively important concepts between texts (FCIC)

4.1.1. Summary

The method in this paper is based on a comparison between the frequencies of concepts as they occur in a baseline (or corpus) text and a comparison (or input) text. The baseline text is usually defined as a normative corpus such as the British National Corpus, whereas the comparison text is chosen to be the text under analysis (in the case of Feed.UVL, user comments).

The algorithm uses stopword elimination to remove irrelevant words from the texts and lemmatisation to reduce related words (e.g. „ran“ and „runs“) to a common form. During this preprocessing stage, it identifies all concepts with $\mathit{length} \leq N$ for some natural N.

After the text has been preprocessed in this way, the **relevance** of a concept is computed for every identified concept using the formula:

$$\frac{F_a(c)}{F_a(c)+F_b(c)}$$

, where $F_a(c)$ is the number of occurrences of concept c in the input text, and $F_b(c)$ is the number of occurrences in the corpus.

In the following step, the algorithm prunes the set of considered concepts by eliminating all concepts from consideration with a frequency score beneath a certain threshold. Note that this is equivalent to only selecting the top M concepts with the best frequency scores for consideration. Optionally, one may remove those concepts whose occurrences $F_a(c)$ in the input text are below some cutoff.

Then, the algorithm proceeds to train a **decision tree** using the new set of M candidate concepts together with a sentence dataset S. S is derived by combining all sentences from the input text together with some subset S' of sentences of the corpus. Note that this is not a set in the mathematical sense because duplicates are allowed.

To do this, the algorithm processes the corpus in the same way the input was processed (this step may be done before execution of the algorithm) and after gathering all sentences to be used, maps each sentence s to its *concept vector* $X(s)$, defined as the result of

$$\begin{aligned} X\colon S \to B &:= \{0,1\}^{M+1} \\ s \mapsto x \in B\colon x_{M+1} = 0 &\iff s \text{ was mined from the corpus} \\ \wedge \forall m \in \{1,\dots,M\}\colon x_m = 1 &\iff c_m \text{ occurs in } s \end{aligned}$$

, where c_j is defined as the concept with the j'th highest frequency score.

Now, any standard decision tree algorithm can use the training set $\{X(s)\colon s \in S\}$ to train a decision tree to map an unseen input sentence s' to a binary-valued vote. Each node in this tree can be interpreted as a test on the occurrence of a particular concept c_n in s'. Leaf nodes represent a categorization of s' as either part of the input text or part of the corpus text.

Note that the order in which concepts are tested for induces a new ranking of importance on the candidate concepts. The set of candidate concepts together with this new ranking, as well as an array encoding the decision tree (see 6.4.2) are the output of the algorithm and used for visualization of the result in the browser.

4.1.2. Suitability for Feed.UVL

This algorithm was deemed suitable for Feed.UVL because it straightforwardly derives document keywords from an input text. The simplicity of its design and the availability of the XML version of the British National Corpus, which provides annotations that minimize the need to perform complex calculations such as lemmatisation and stopword elimination, simplify implementation. Additionally, there are many implementations of decision tree algorithms available, which allows for code reuse and keep the risk of introducing bugs low.

4.2. Relevance-based Abstraction Identification (RBAI)

4.2.1. Summary

RBAI is a similar approach to FCIC in that it is also a frequency-based approach that computes a relative frequency score by comparing concept occurrences in sentences from the input with occurrence data from a normative corpus. Like FCIC, RBAI starts by first lemmatising and removing stopwords from all sentences in the input and the corpus.

However, a key difference lies in how scores are computed for multi-word items: While

FCIC counts the occurences of a multi-token concept in the straightforward way for both sentences in the input as well as the corpus, RBAI looks for multi-word concepts **in the input only**, and computes the frequency score for such multi-word concepts using the corpus frequencies of the individual lemmas that make up that concept.

In other words, the total score for a concept discovered in the input is an arithmetic combination of *corpus frequency* scores for the individual lemmas in that concept, and not a function of the whole concept per se.

Using the notation that c_i denotes the i'th lemma in concept c, the concept score function S for concept c is:

$$\begin{aligned} S\colon C &\to R \\ c &\mapsto \sum_{i=1}^{i=N} k_i LL_{c_i} \end{aligned} \tag{1}$$

With:

$$\begin{aligned} LL_{c_i} &= 2 \cdot \left(d(c_i) \cdot ln\frac{d(c_i)}{E_d(c_i)} + h(c_i) \cdot ln\frac{h(c_i)}{E_h(c_i)} \right) \\ E_d(c_i) &= \frac{n_d(d(c_i)+h(c_i))}{n_d+n_h} \\ E_h(c_i) &= \frac{n_h(d(c_i)+h(c_i))}{n_d+n_h} \end{aligned} \tag{2}$$

, where n_d refers to the total number of tokens occurring in the input text, and n_h refers to the total number of tokens occurring in the corpus text. $d(w)$ and $h(w)$ refer to the total number of occurrences of lemma w in the input and the corpus, respectively. k is a custom weight vector that per specification most heavily weights the first lemma in a concept. The authors propose as an example, for $N = 3$: $k = (0.5, 0.3, 0.2)^T$.

Line for line, the interpretation of equations 1 and 2 is: "The score for a concept c is the weighted average of the concept scores of its constituent lemmas c_i. The score of a lemma c_i is given by the value of LL_{c_i}."

Finally, the algorithm returns a ranking of concepts. In this implementation, only the top M concepts are returned, ordered by concept score.

4.2.2. Suitability for Feed.UVL

RBAI is an elegant and straightforward approach that is shown to perform reliably on problems of this nature, and has hence been included for implementation in Feed.UVL.

4.3. Concept mapping as a means of requirements tracing

4.3.1. Summary

A complete requirements specification usually involves several different documents written in different registers and at different levels of abstraction. For instance, a low-level document may describe the specific functionality of a login page on a university website for prospective students or applicants. A higher-level document may describe the functionality of this webpage from the perspective of the end user. This approach serves three main purposes in this context:

- Extraction of domain-specific concepts
- Concept mapping (map similar concepts to each other within and across documents)
- Tracing (find related text passages based on the concepts they discuss)

Algorithm Overview:

1. Through the application of RBAI, concepts are extracted for each document under analysis
2. Once concepts are extracted, distance metrics between individual words (based on WordNet graph distance) and for concepts (computed from word distances for the words in each of two concepts) are defined.
3. All concepts in low-level documents are mapped to concepts in higher-level documents using the similarity metric defined in step 2.

4.3.2. Suitability for Feed.UVL

Because the implementation is only interested in identifying concepts, and the concept identification step of the algorithm is based entirely on RBAI, this approach does not have to be considered further for the purposes of this practical.

4.4. Automated Extraction of Conceptual Models from User Stories via NLP

4.4.1. Summary

This approach uses heuristic rules to analyze syntactic structures of *user stories* in order to derive a UML-based model.

This approach is not applicable to the practical because input data in the practical are not necessarily phrased as user stories and may lack proper orthography or grammatical structure. The method is also not applicable to non-English contributions, unlike the other approaches.

5. Pre-Implementation Considerations

5.1. Research Phase

Following the considerations laid out in chapter 4, the selection was narrowed down to two algorithms: **Relevance-Based Abstraction Identification** [12] (RBAI) by Gacitua et al and **Finding Comparatively Important Concepts Between Texts** [11] (FCIC) by Renaud Lecoeuche.

5.2. Planning Phase

Once the preliminary evaluation of the algorithms had been completed, it was time to plan their implementations.

To begin with, variable factors affecting the implementation of the two candidate algorithms had to be considered:

5.2.1. Choice of Corpus

Both algorithms rely on a comprehensive language corpus to provide statistical information about the occurence of English words in addition to stemming/lemmatization and stopword detection steps in their training and execution phases. It follows that a corpus annotated with grammatical information on a word-for-word basis would be the easiest choice for an effective implementation.

Furthermore, the author's implementation of RBAI uses the British National Corpus[15] (BNC) as a training corpus for the algorithm, making it the desired choice for use in the implementation. The British National Corpus homepage makes available a version of the BNC in XML format [16] with part-of-speech tags and lemmas included for each word; this was the corpus that was chosen for use in the implementation.

5.2.2. Technical Considerations

The most important initial consideration was in what language the algorithms were to be implemented. The primary motivation was to find a programming language which provided the following characteristics:

Control over Performance

While dynamically typed languages like JavaScript and Python offer speed and flexibility of development, they give up the ability of compiled languages to optimize memory access and running time on a working memory level. In order to satisfy the stated requirement of fast running times, the choice was made to use a statically typed, compiled language - either Java, C++, or Go.

Reduce Developer Time

In this area, Java and Go are the clear favorites. Go provides a convenient console interface for downloading, managing and including packages for projects, and Java has well-established build frameworks such as Maven or Gradle for this purpose.

The most well-known build system for C++ is CMake, which is in the judgement of the author a confusing and messy legacy system whose maintenance takes up an undue amount of development time. In the experience of the author, using a modern option rather than C++ with CMake saves 90% of the initial setup and build/toolchain maintenance time for projects.

Simplify string Manipulation

Because this is a natural-language-processing application, it seemed likely that fine-grained control over strings and their constituent characters would prove useful and necessary.

It is known that string manipulations in Java can be tricky - iterative concatenations in a "for" loop create a copy of both the base string as well as the added string at every step, resulting in quadratic complexity as the total length of the string increases.

While certain constructions such as Java's stringBuilder help avoid problems of this nature, C++ provides string containers that dynamically resize on extension with another string (a specialization of the data structure "dynamic array"), avoiding most of the complexity overhead involved with the Java approach.

C++ provides a view of a string as a "C-string", i.e. an array of characters. When optimizing for performance, such lower-level interfaces provide reliable ways to access file data in-place with no unintended copies or side-effects.

Additionally, passing a string to a method in Java creates a copy of the underlying string object at every invocation. When the number of method invocations and manipulations is high, this can result in a substantial slowdown of the program. This behavior can be circumvented by passing wrapper objects instead, which maintain references to the underlying string rather than copying it directly.

Ultimately, while knowledgeable use of Java constructs probably allows similar performance to C++, it is "hard to go wrong" with C++, with little programming or research overhead required to get efficient and fast performance on string data.

Because the author didn't have much experience in using Go and didn't want to take a risk, C++ was chosen as the likely favorite in this point of consideration.

Library Support for XML parsing

Finally, once a general preference for C++ had been established, the author investigated library support for rapid and space-efficient XML-parsing in C++. The rapidxml [17] library was selected for use with this application because of its simplicity (it is provided in a single header file) and proven track record as a robust and highly optimized choice for this type of problem.

6. Algorithm Implementation

In the following sections, we shall discuss the details of the technical implementations of the two algorithms, together with the most interesting problems and challenges that were encountered.

6.1. Server-Side Algorithm Implementation

Each algorithm is implemented in C++. For the reasons described in section 2, each algorithm was implemented to run in its own separate container, with separate *main* files controlling the flow of their executables. The source files for both algorithms are bundled in a git repository [18] including, for each algorithm:

- C++ Sources: These source files contain the concrete implementations of each algorithm, together with their *main* files (which provide a shell interface for other programs to call). Source files are compiled into an executable file by the Dockerfile
- Resource files: These files include statistical information on word frequencies (generated by the program) and lists of stopwords, as well as a lemmatization file, for use during the execution phase of the algorithms
- Flask Endpoint: This Python file starts the server endpoint which redirects client requests to the corresponding algorithm for processing (through the shell using Python's subprocess module)
- A Dockerfile: This file defines a Docker container, which hosts the Flask server and whose instructions include the compilation of the C++ sources into an executable file.

Finally, a Jenkins build is configured for each algorithm, which launches the container based on the current state of the git repository and automatically provides it with access to the host filesystem as well as instructing it to expose the correct port on which to handle incoming client requests.

6.2. Client-Server Interface

For the implemented algorithms to be useful and understandable to the general user, algorithm results must be visualizable and run parameters configurable through a user interface. This requires the formalization of a program interface to communicate data in four stages:

1. User inputs such as input text, concept word length, and number of returned concepts are aquired through two-way data bindings between an underlying JSON *params* object and a set of user interface elements. This is made simple and robust against bugs by the web development framework Vue [19].

2. When the user triggers the execution of one of the algorithms, the *params* object is serialized into a string, attached to an HTTP-GET request, and routed to the algorithm's Flask endpoint. Here, it is deserialized back into the *params* object, and its fields used as inputs to the algorithm. The Flask endpoint at this point calls the executable with the program inputs and waits for it to finish executing.

3. Once the executable finishes the processing of the algorithm, its resultant data structures are encoded as a JSON-formatted object string and written to the executable's output buffer, which is then read by the Flask server and from there immediately returned to the client in the response body.

4. Now, the user can choose to render the corresponding visualizations in the browser - the JSON object from the server is deserialized and its data fields used to create the algorithm output visualizations.

Figure 6.1 shows the UI-Structure Diagram for workspaces in the web-app. Concept detection is launched from workspace **WS2** and the results (see section 6.8) are displayed in **WS3**.

6.3. Text Parsing Data Structures

The steps involved in computing the two algorithms are quite similar. In the pre-execution stage, resource files need to be either generated from training data (a corpus)

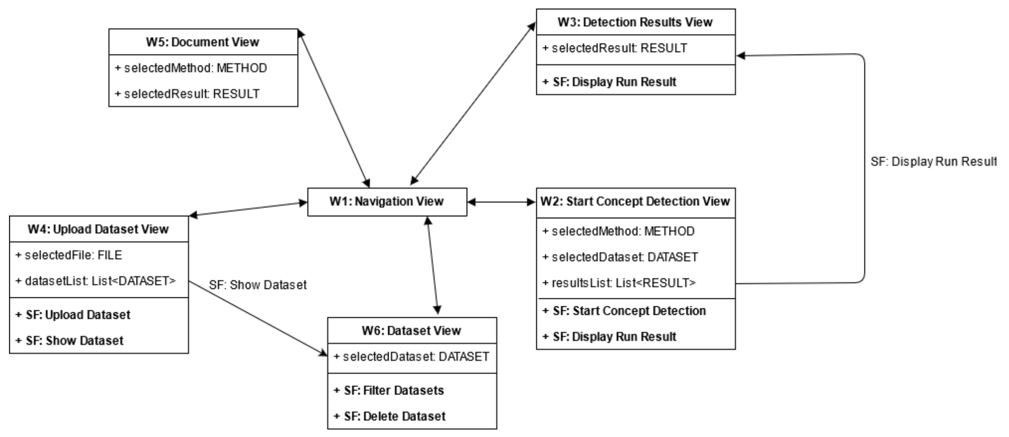


Figure 6.1.: UI-Structure Diagram

or acquired through other means and made available for use by the algorithms at run-time.

Both algorithms need statistical information about the relative frequency of individual English concepts in a baseline corpus. Note that the following section describes the author's personal choice of implementation; the white papers do not prescribe any particular data structure of approach for implementing the algorithms.

Broadly speaking, the objective of both algorithms is to **derive an ordered list of the most important concepts** occurring in a given input text, where we consider only **concepts formed by consecutive lemmas occurring in the same sentence**.

Ideally, we would allow for concepts for $length \leq N$ for any reasonable N, e.g. $N = 5$. However, given 50000 base English lemmas, the number of possible $length \leq N$ concepts is equal to $3.1250625e23$, an amount that is computationally unfeasible to store and would require an unfeasibly large corpus in order to accurately capture multi-word frequency data.

Accordingly, while the algorithms were written to compute the general case of concepts of any integer (int) length, in practice user inputs are restricted to concepts search with $N \leq 2$.

It is important to note that the algorithms differ in that FCIC requires information about the relative frequency of concepts of any $1 \leq length \leq N$, while RBAI requires only information about the relative frequency of concepts of $length = 1$, i.e. tokens, for all N. The relevance score for multi-word concepts in RBAI is computed by arithmetically combining the relevance scores for the tokens that make up that concept. However, we can still express concept search for RBAI in terms of the general algorithm outlined below (using $N = 1$).

In the following subsections, we will consider the result of parsing the following input:

“He held those in high esteem who held him in the lowest. This is another sentence.”

To find all concepts for $N = 3$ in this text, we first tokenize each sentence and then lemmatise its constituent words. We then scan each word to see if it matches a stop word, and discard it from consideration if it does.

The result of this process are two new sequences of tokens:

$$\text{hold high esteem hold low} \tag{3}$$

$$\text{another sentence} \tag{4}$$

6.3.1. Concept Storage

While processing the corpus text, we will store our tokenized corpus data in a tree data structure. We initialize the tree with the empty token as the root node with depth 0, which we add for convenience of representation. A node at depth $k > 0$ represents the final token in a concept of $length = k \leq N$, with its parent representing the preceding token (and so on).

Note that this way, concepts c_1, c_2 of $length = k$ with a common root $\forall j < k\colon c_{1j} = c_{2j}$ require a combined storage space of $k + 1$ tokens, because only the last nodes are different, with a reference to the same parent. There is a 1-1 correspondence between nodes in the tree (except the root) and unique concepts in the input corpus.

Each node in the tree is labeled with an occurrence counter initialized to 1 - when the same concept is discovered again, it is incremented. A node's children are stored in alphabetical order of the tokens they represent, allowing a binary search to quickly perform lookups and insertions in logarithmic time.

We say we *update* node n with token t when we either insert a new node n' representing t as a child of n if no such node exists, or else we increment the frequency counter of that child.

Definition 5 (Pointer Operations) *For a node n, the function $p(n)$ returns a pointer*

to n.
We write $n(p)$ to mean the node referred to by pointer p, i.e. $n = n(p(n))$

Figure 6.2 shows the final tree structure for the example input text. Note that the frequency counter is set to 1 everywhere (not written) except for "hold" at $depth = 1$, with $c = 2$. This is because the word "hold" occurs twice in the input. The two child nodes of "hold" represent the following lemmas that occured in the input.

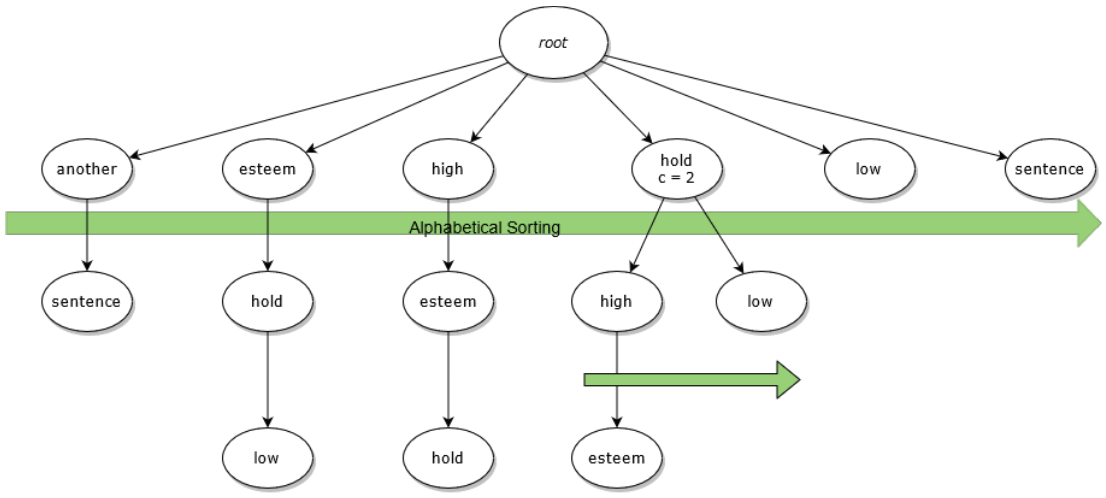


Figure 6.2.: The concept tree for the example text.

6.3.2. Pipelining

In order to mine concept frequencies from an input text and populate the concept tree, the implementation used a pipeline-based approach to capture all concepts of $length \leq N$.

Definition 6 (Pipeline) *A pipeline of size N is an array of N pointers p to nodes $n(p)$ in the concept tree.*
As long as at least one element in the pipeline is NULL, we refer to the pipeline as "not loaded".

While this definition is tailored to the context of this application, similar structures can be found e.g. in CPU design to solve similarly occuring problems. For more information see e.g. [20].

Algorithm 1 shows how the preprocessed input is fed into the data structure.

Algorithm 1: Populating the Concept Tree

```
Tokenize, lemmatize and remove stopwords for all sentences in text;
Initialize the concept tree to root node $root$;
Initialize a pipeline $P$ of length $N$;
while $unread\ sentences\ remaining$ do
    Set each pointer in the pipeline $P$ to NULL;
    for $lemma\ in\ sentence$ do
        for $p \in P : p \neq NULL$ do
            if $depth(n(p)) < N$ then
                update $n(p)$ with $t$ ;
            else
                update $root$ with $t$, yielding node $n$ with $depth = 1$;
                replace $p$ with $p(n)$ in $P$;
            end
        end
        if $pipeline\ is\ not\ loaded$ then
            update $root$ with $t$, yielding the node $n$;
            replace the first NULL element in $P$ with $p(n)$;
        end
    end
end
```

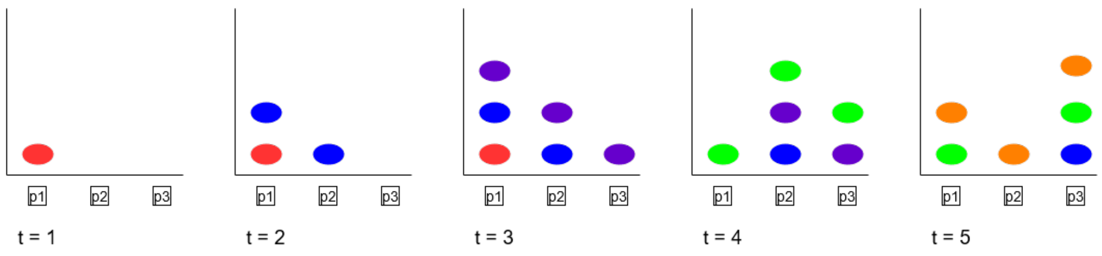


Figure 6.3.: The first 5 steps of the pipeline (color coded according to the example). The pipeline becomes loaded at step 3. After the pipeline becomes loaded, every step involves one replacement with a $depth = 1$ node.

At each step of the execution, the next token t of the currently processed sentence s is read in. Once the pipeline is loaded, each step (each new processed token) produces/updates N concepts in the tree. The pipeline is reset for every sentence (in this implementation, after every sentence-like punctuation mark such as commas, exclamation marks, question marks, etc.).

Note that building the concept tree is done once using the normative corpus (BNC) before program execution (to generate the frequencies dataset used by the algorithm during execution), and then once at every program execution with the user-provided input text.

Accordingly, algorithm 1 has been implemented in two forms: Once to parse annotated, lemmatized BNC XML files, and once to parse arbitrary input texts provided by the user. The differences lie primarily in how lemmatisation and stopword removal are performed: In the former case, lemmas are already provided and stopwords are detected based on part-of-speech tags; tokens such as pronouns, prepositions, etc. are removed. In the latter case, each token is mapped to a lemma using a lemmatisation dataset (this implementation uses [21]), and the resulting lemma is scanned against a pre-defined list of stopwords and excluded from analysis if a match is found. Tokens are presumed to be lemmas if they do not occur in the lemmatisation dataset.

The stopwords, computed frequencies, and lemmatisation files are all stored under a *resources* folder in the program directory. The two implementations of algorithm 1 can be found in *frequency_ manager::get_ tokens* for program inputs and in *frequency_ accepter::word_ search* for reading BNC files.

6.4. FCIC

6.4.1. Modifications

Now that the first part of FCIC, which it shares with RBAI, has been introduced, its distinguishing characteristics will be discussed.

Once the algorithm is called to execute (given a desired concept length N, an input text and the number of concepts M to find), the algorithm proceeds to build the concept tree from the input text according to algorithm 1. After it has done this, it computes the frequency score for each identified concept (for details see appendix [[[]]]) and orders the concepts in descending order of their frequency score.

Practical results showed that the vast majority of domain concepts never occurred in the corpus, giving them the highest possible frequency score 1. For example, in an input document about the online learning platform "moodle", neither the concept "moodle" nor the concept "sidebar" occurred in the corpus, so that both concepts got the frequency score 1, even though "moodle" was mentioned more often in the text.

Because nodes are stored in alphabetical order in the concept tree, this meant that if the user restricted the number of concepts to a certain degree, the algorithm would return the concept "sidebar", but not the concept moodle, because it has higher alphabetical order. Because this contradicts the intuitive expectation that ties should be broken by which concept occurs more frequently, the algorithm was modified to prefer concepts that occured more frequently in the case that two concepts earned the frequency score 1. This modification does not violate the design set out in the paper - it is merely a more specific realization of the design.

6.4.2. Decision Tree Implementation

The specification of the FCIC algorithm calls for the use of a decision tree to learn to classify sentences as either part of the training corpus or the input text. Each node in this decision tree is a test on the presence of a certain concept in the input sentence. The concepts used in these tests are the top M concepts with the best frequency scores ("candidate concepts").

The implementation of FCIC was originally meant to reuse an existing C++ implementation of a decision tree algorithm, but it was surprisingly difficult to find a generally applicable algorithm for use. Finally, after some disappointing attempts to use various implementations to no success, an implementation from github [22] by Nikhil Iyer which uses information gain as a splitting condition was selected for use.

This implementation did not compile correctly, and with some reverse-engineering, the compile errors were eliminated.

This implementation was written to be as general as possible - input data were vectors of strings, assigned to one of a set of enumerated string labels. Each feature was meant to be a string value taking one of arbitrarily many enumerated values.

While the code did eventually run and perform the desired computations, it did so in an extremely suboptimal way. The code relied on hash maps in most places where one could have used vectors, so that it was orders of magnitude slower than a similar implementation using more suitable container classes. A single execution of the algorithm on a small BNC subset took multiple minutes to complete.

In the end, an entirely new implementation of the decision tree algorithm was written, this time operating on boolean feature vectors $x \in \{0,1\}^{M+1}$ representing each sentence s from the input and a subset of the sentences occurring in the BNC, with x_i encoding the presence or absence of candidate concept i in s (see also section 4.1.1).

This new boolean-vector based implementation achieved a speedup factor over the github version of around 500. Almost all of the runtime of the final FCIC implementation is spent gathering training sentences from the corpus and preprocessing the input.

Finally, the decision tree induces a new ordering on the candidate concepts. More informative concepts are split on earlier in the decision tree structure. The new decision tree implementation was altered to save and return the *information gain* value for each split, returning it to the client together with the list of candidate concepts and a vector-based representation of the whole decision tree.

6.4.3. Decision Tree Representation

A decision tree is a type of binary tree - each node has at most two children. There is a "natural" bijective mapping between node position in a binary tree and the natural numbers revealed by the following scheme: Starting from the root node with value 1, traverse the children at every following level from left to right (breadth-first search starting at root, preserving left-to-right node order at every level).

We shall assert that the natural index of the left child of a node n with index k is $2 \cdot k$, and the natural index of the right child of n is $(2 \cdot k) + 1$. Note that the index increases exponentially with the depth of the tree.

The use of this mapping is that it lets us send the decision tree back to the client packed in an array. There, it can be turned back into the original tree using the same strategy.

The naive approach to this strategy would be to compute the index for every node in the decision tree and insert the node label into a vector at that index. The problem with this approach is that if the top M concepts never occur in the corpus (which is in practice often the case), then the resulting decision tree will be highly degenerate, with every node except the last having exactly one leaf child. Such a tree achieves a depth of M, meaning that for a value like $M = 30$, the highest occuring index of a node in this decision tree will be 2147483648. Naively initializing an array to this length will probably throw a memory exception, not to mention that it would take too long to send to the client.

The solution is instead to simply append the node index to the end of an output array together with the node's label. A single node now takes up twice the space in the array, but in the naive case with $M = 30$, the ratio of array components that actually store a node and don't sit empty is $\frac{(30\cdot2)+1}{2^{31}}$, which is about 1 out of 50 million, so the savings realized in practice make this the better choice.

There is one more problem to consider: In the case where we choose M to be something greater than 31, we can no longer calculate the index using the type *unsigned int* using the formula above, because we exceed the size of the underlying numerical type.

The solution that was implemented was to use the type *unsigned long long int*, which holds twice as many bytes as a standard *unsigned int*, allowing us to use up to a comfortable 62 concepts, give or take, before the numerical limits are reached. If it becomes clear that this is still not enough concepts to satisfy requirements, an unbounded type provided by an external library should be used instead.

6.5. RBAI

As described in sections 4 and 6.3, the computations involved in executing RBAI are a subset or derivation of the computations involved in computing FCIC. The only modification of note which had to be made (besides ignoring the last decision tree step) was to change the computation of the score function for individual lemmas and adapt the score function for multi-word concepts to become a linear combination of the scores of its constituent lemmas.

6.6. Diagrams

The following diagrams illustrate the class structure for RBAI and FCIC. Figure 6.4 shows the class diagram from the pre-execution corpus-based training step. Figures 6.5 and 6.6 show the class diagram for the execution stage of RBAI and FCIC, respectively.

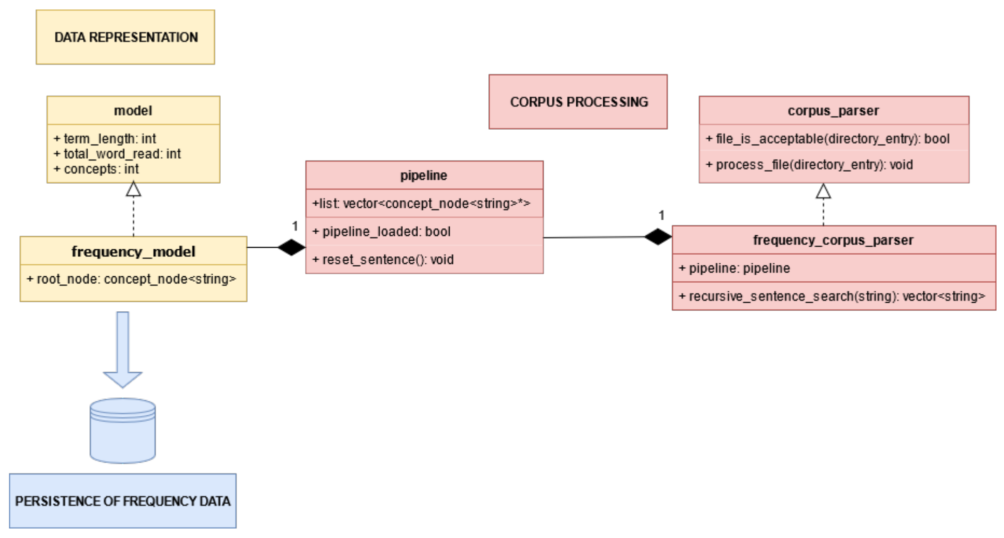

DATA REPRESENTATION
model
+ term_length: int
+ total_word_read: int
+ concepts: int
frequency_model
+ root_node: concept_node<string>
PERSISTENCE OF FREQUENCY DATA
1
pipeline
+list: vector<concept_node<string>*>
+ pipeline_loaded: bool
+ reset_sentence(): void
CORPUS PROCESSING
corpus_parser
+ file_is_acceptable(directory_entry): bool
+ process_file(directory_entry): void
1
frequency_corpus_parser
+ pipeline: pipeline
+ recursive_sentence_search(string): vector<string>

Figure 6.4.: Pre-Execution Stage for RBAI and FCIC (gather frequencies)

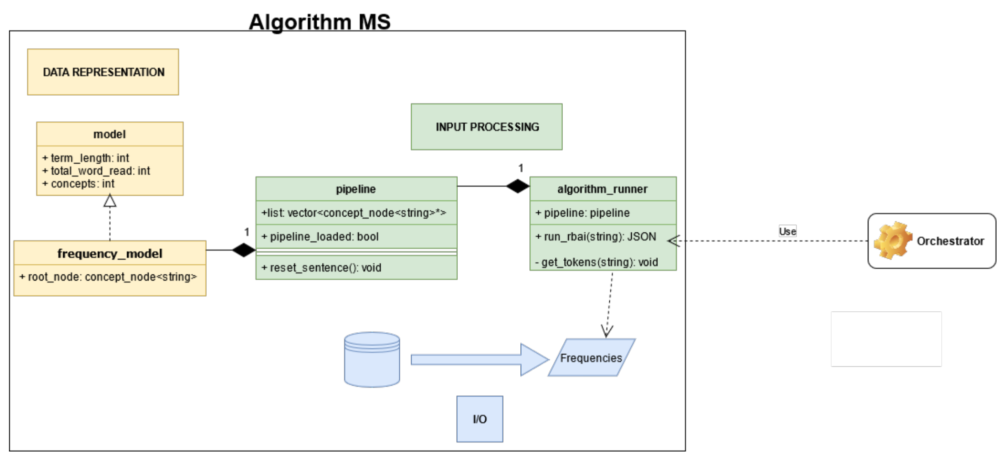


Figure 6.5.: Execution Stage for RBAI

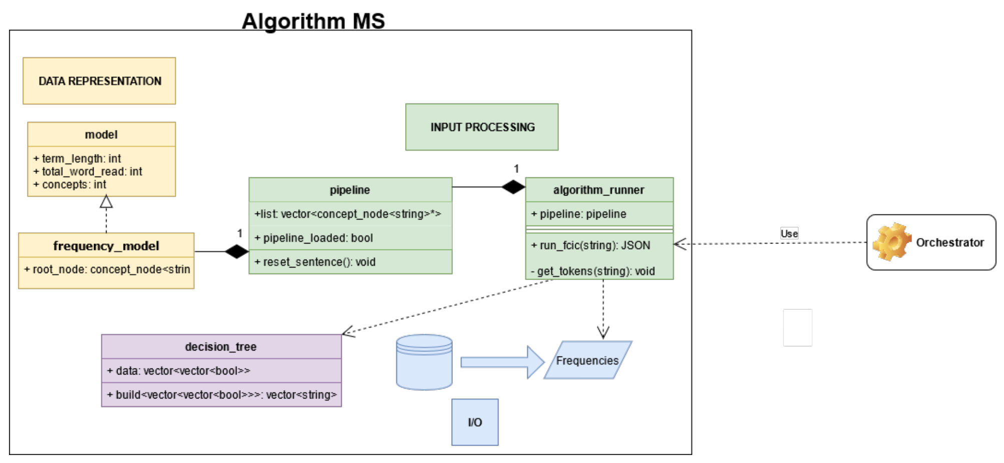


Figure 6.6.: Execution Stage for FCIC

6.7. Concept Tracing

In order to trace extracted concepts to their occurrence in the input, a second program was written which reuses components from the preprocessing stages of FCIC/RBAI.

In short, a list of extracted concepts is passed as an argument to the program together with one of the entries in the dataset. The entry is preprocessed (lemmatised, stopwords removed) as in normal execution of the algorithm and watches for occurrences of the any of the provided concepts. Finally, it returns a vector of boolean values encoding the occurrence of a concept in this entry.

This program is called once for each entry to yield the total occurrences of all concepts in all entries of the dataset and visualizes as a heatmap in the results page of the user interface.

6.8. Front-End Visualization

Algorithm outputs were visualized using three UI elements (figures show results for an RBAI run):

1. Word cloud (figure 6.7)
2. Occurrence table (figure 6.8)
3. Heat Map (figure 6.9)

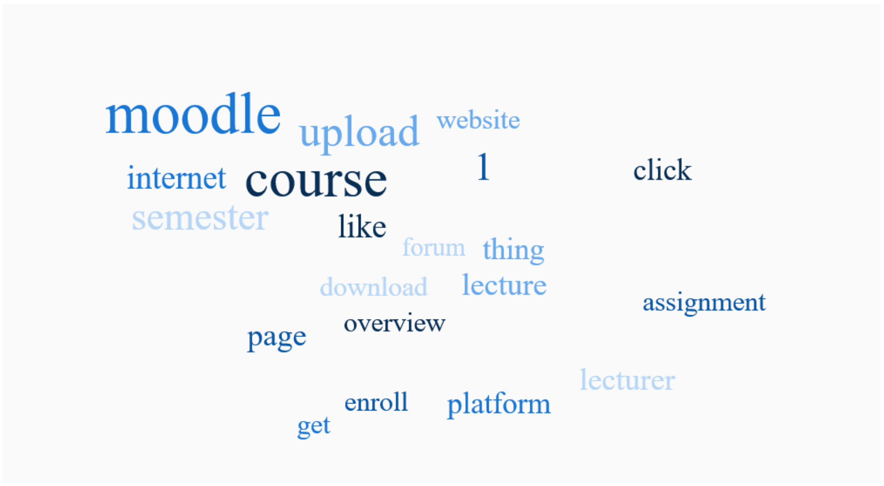
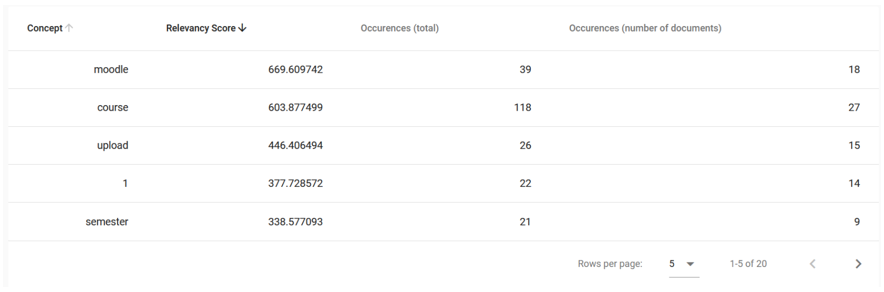


Figure 6.7.: Word cloud for best concepts



Concept ↑	Relevancy Score ↓	Occurences (total)	Occurences (number of documents)
moodle	669.609742	39	18
course	603.877499	118	27
upload	446.406494	26	15
1	377.728572	22	14
semester	338.577093	21	9

Rows per page: 5 ▾ 1-5 of 20 ‹ ›

Figure 6.8.: Table giving details

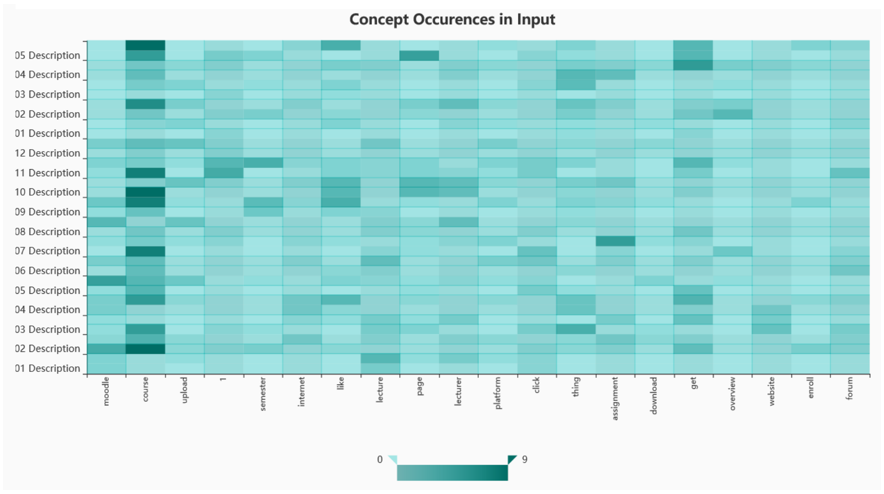


Figure 6.9.: Tracing concepts to input texts

7. Algorithm Evaluation and Results

7.1. Methodology

FCIC and RBAI concept outputs were compared to a human-curated list of target concepts and validated using the information retrieval quality measures of **precision**, **recall** and **F1-score**. The input dataset was a collection of interviews on the subject of the online learning platform "Moodle", carried out by the University of Heidelberg. Target concepts were compiled by human researchers on the basis of the input text before the choice was made to use these particular algorithms.

Before computing the quality metrics for algorithm outputs, the target concepts need to be extracted from the reference document and preprocessed for comparison with RBAI/FCIC.

The target concept set provides a list of non-lemmatised keyword segments which may contain stopwords such as "the". Additionally, the provided keyword segments may be of any length, and can therefore not be directly compared to algorithms outputs. Because of this, the implementation of precision, recall and F1 only checks for the *presence of extracted lemmas* in the target concepts. This way, the presence of stopwords in the target concepts has no influence on the scores for these metrics because the extracted concepts contain no stopwords, and returned concepts can be compared with target concepts of any length. Extracted concept length is a configurable parameter of the algorithms (see chapter 4) and was restricted to lengths 1 or 2 for RBAI and length 1 for FCIC.

Finally, duplicate concepts were removed from the set of target concepts. The list of target concepts obtained from this preprocessing has been included in appendix A.

7.1.1. Implementation

Given an algorithm concept set C with index set $\bar{C}$ and the set of preprocessed target concepts T with index set $\bar{T}$, we define the 'match' function for concepts $c_i \in C$, $t_j \in T$ with $c_i =< c_{i1}, \ldots, c_{iN} >$ and $t_j =< t_{j1}, \ldots, t_{jM} >$ as:

$$\begin{gathered}\delta \colon \bar{C} \times \bar{T} \to \{0, 1\} \\ \delta(i, j) = 1 \iff \exists k \in \{1, \ldots, N\}, k' \in \{1, \ldots, M\} \colon c_{ik} = t_{jk'}\end{gathered}$$

i.e. if two concepts share a lemma.

With this notation, the number of **true positives (tp)**, **false positives (fp)** and **false negatives(fn)** becomes:

$$\begin{aligned}\text{tp} &= |\{i \in \bar{C} \colon \exists j \in \bar{T} \colon \delta(i, j) = 1\} =: TP| \\ \text{fp} &= |\bar{C} \backslash TP| \\ \text{fn} &= |\{j \in \bar{T} \colon \forall i \in \bar{C} \colon \delta(i, j) = 0\}|\end{aligned}$$

These values are applied to the conventional definitions of precision, recall, and one of the various equivalent definitions of F1:

$$\begin{aligned}\text{precision} &= \tfrac{tp}{tp+fp} \\ \text{recall} &= \tfrac{tp}{tp+fn} \\ \text{F1} &= \tfrac{tp}{tp+\frac{1}{2}\cdot(fp+fn)}\end{aligned}$$

The full implementation of the validation methodology can be found in <**uvl-analytics-concepts-frequency**>/**validate_test_input.py**[1].

7.2. Results

Results show a generally high precision score for both algorithms. Because a single lemma l can be part of multiple different extracted concepts with length $N \geq 2$, all

[1] Available at: https://github.com/feeduvl/uvl-analytics-concepts-frequency. To reproduce the validation, uncomment the call to the validation method at the bottom of the "post_classification_result" function in the corresponding algorithm's Flask server file. Then execute the algorithm in the UI using the moodle interview test dataset, which can be found under the "lib/res" path of the project sources.

extracted concepts containing that lemma will be considered matching if there is a target concept containing l, leading to an optimistic judgment of precision for concept lengths > 1. Therefore, the precision of RBAI with $N = 1$ is more informative: Precision showed a gradual downward trend, as expected, with increasing number of concepts returned. At the same time, recall and F1 scores increase consistently for all concept set sizes. As the number of extracted concepts approaches 175, the improvements in performance in these latter metrics begin to lessen with increasing concepts.

RBAI with concept length $N = 1$ achieved recall and F1 scores at 200 extracted concepts of about 25% and 0.36, respectively. Increasing length to $\leq N = 2$ yielded an improvement in these metrics of about 20%.

FCIC attained lower scores than RBAI across the board. Precision was initially variable but eventually stabilized at about 60%. As with RBAI, recall and F1 trended upwards in a quasi-linear manner as the number of concepts increased, with the highest value of about 18% and 0.27, respectively, attained at 200 concepts. These data suggest that recall and F1 would have kept increasing for some time as the number of concepts is increased.

7.2.1. FCIC Decision Tree

Because the input corpus does not feature software-focused texts (and in fact features no text from after 1993), the implementation has no knowledge of the correct relative frequency of domain keywords such as "click", "upload", "download" etc.

This results in a highly degenerate tree structure (see figure 7.2), where all identified keywords are immediately associated with the input text (since they do not occur in the corpus).

The solution to this problem is to use a more current-day or domain-focused corpus which can provide more accurate estimates of software term word frequencies.

7.3. Interpretation

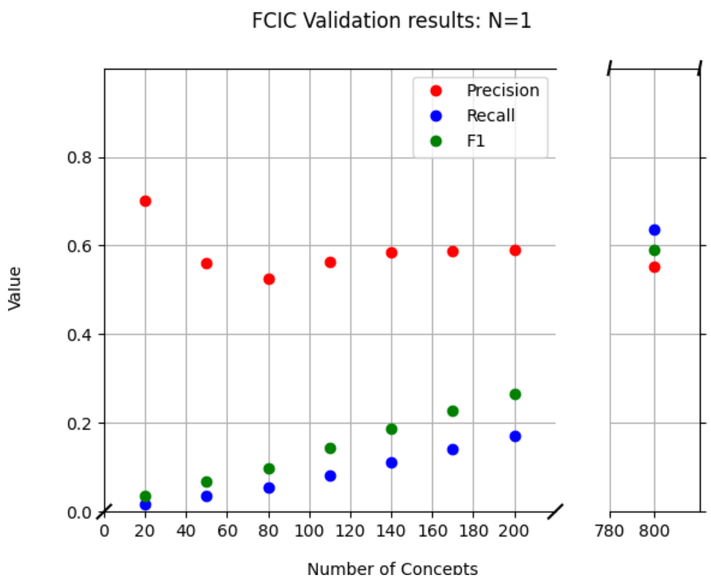
There is some difficulty in extrapolating these results to general statements about the algorithms' effectiveness. Caveats include:

1. Noisy data and lack of concept ranking - The target concept set contains some

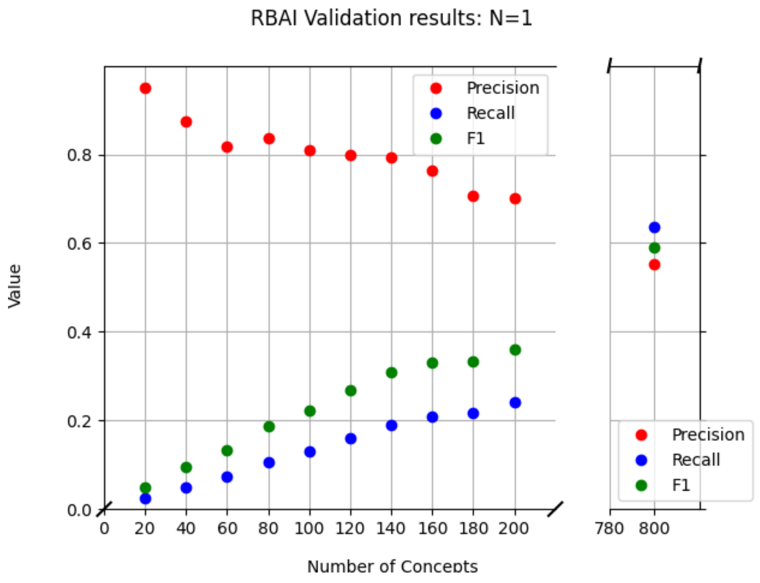
highly important and informative concepts, and many which are less informative. The validation strategy that was implemented ignores the informative value of concepts returned, even though both algorithms also produce a ranking of concept importance. It treats all concepts as equal, and when there is a target set of 800 concepts (after preprocessing), the best strategy is to produce as many concepts as possible, regardless of their value, because the probability of scoring a match is high, even if the quality of the concept is low. This goes against the algorithms' stated purpose of extracting the key, most informative concepts from the text.

2. Typographic errors and non-words - the target contains some non-words such as "the chronologic". This concept can never be found by the algorithms as long as the input data is free of typographic errors because "chronologic" is not a word and "the" is a stopword. The only exception occurs when "chronologic" also occurs in the input data.

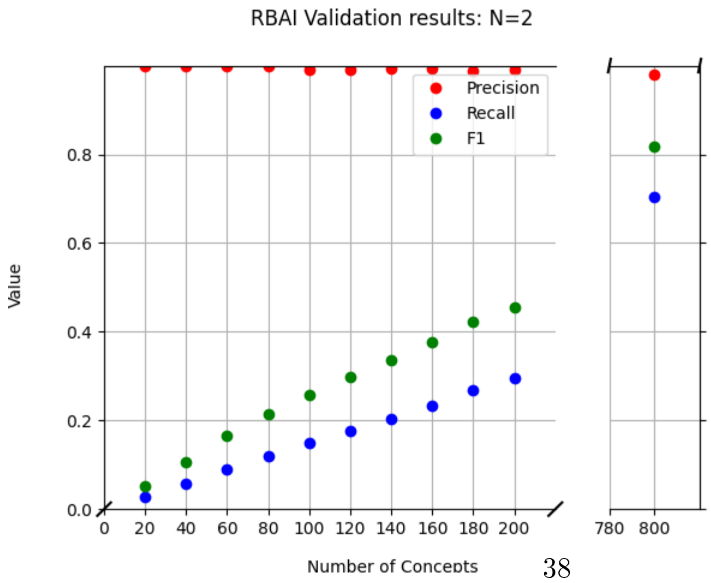
3. Scoring methodology: Because the ground truth features targets of varying length (some > 5), the evaluation metrics had to be made more lenient in identifying matches. A target list of single-word concepts would have allowed for more accurate evaluations.



(a) Validation results for FCIC



(b) RBAI with concept length 1



(c) RBAI with concept length ≤ 2

Figure 7.1.: Validation results for RBAI and FCIC

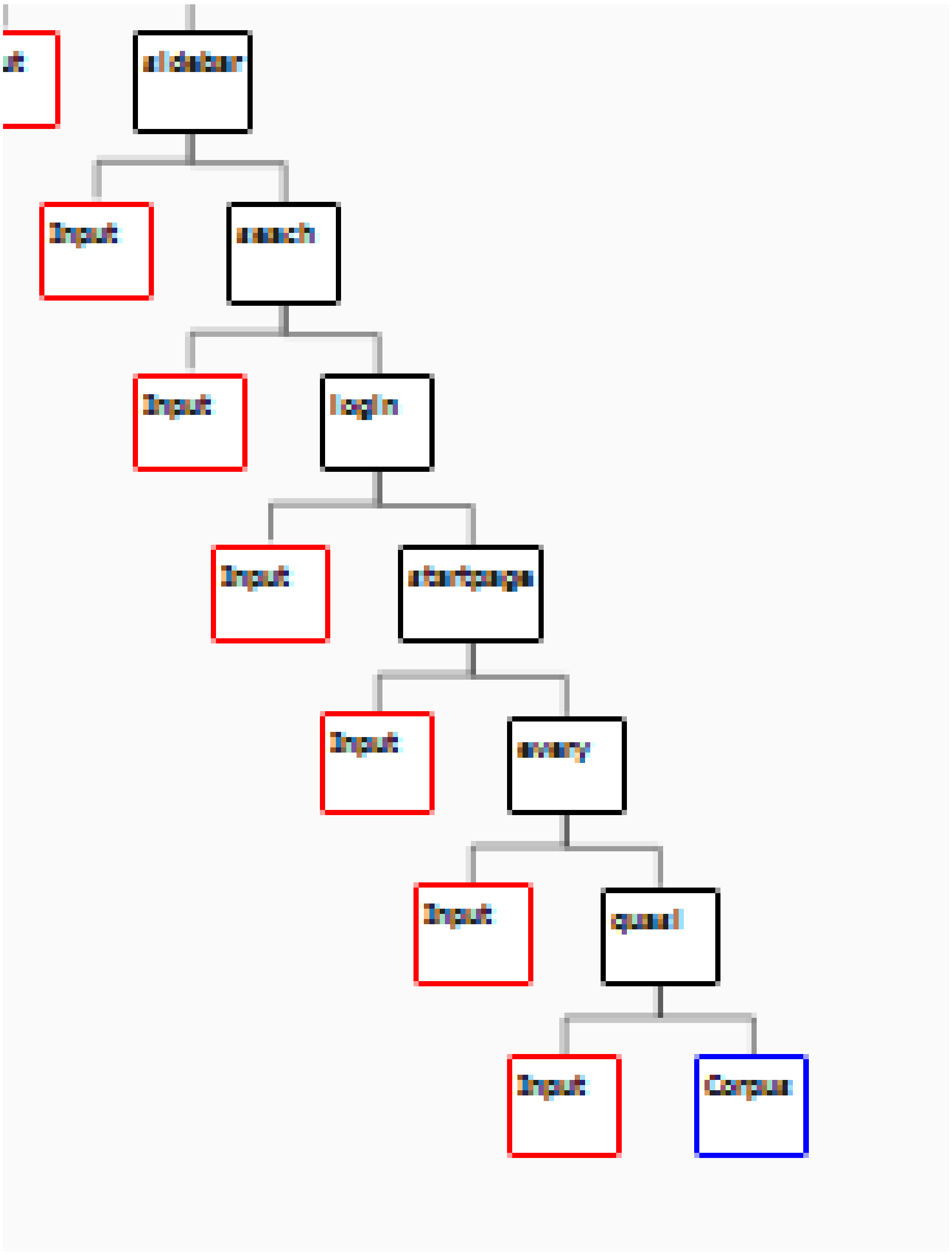


Figure 7.2.: Decision tree results for FCIC in practice

8. Summary

Requirements engineers seek to utilize end user feedback as part of their strategy to develop and maintain software. Because of the sheer volume and noisyness of online contributions, automatic analyses are the preferred tool for deriving insights from this abundant but underutilized data source.

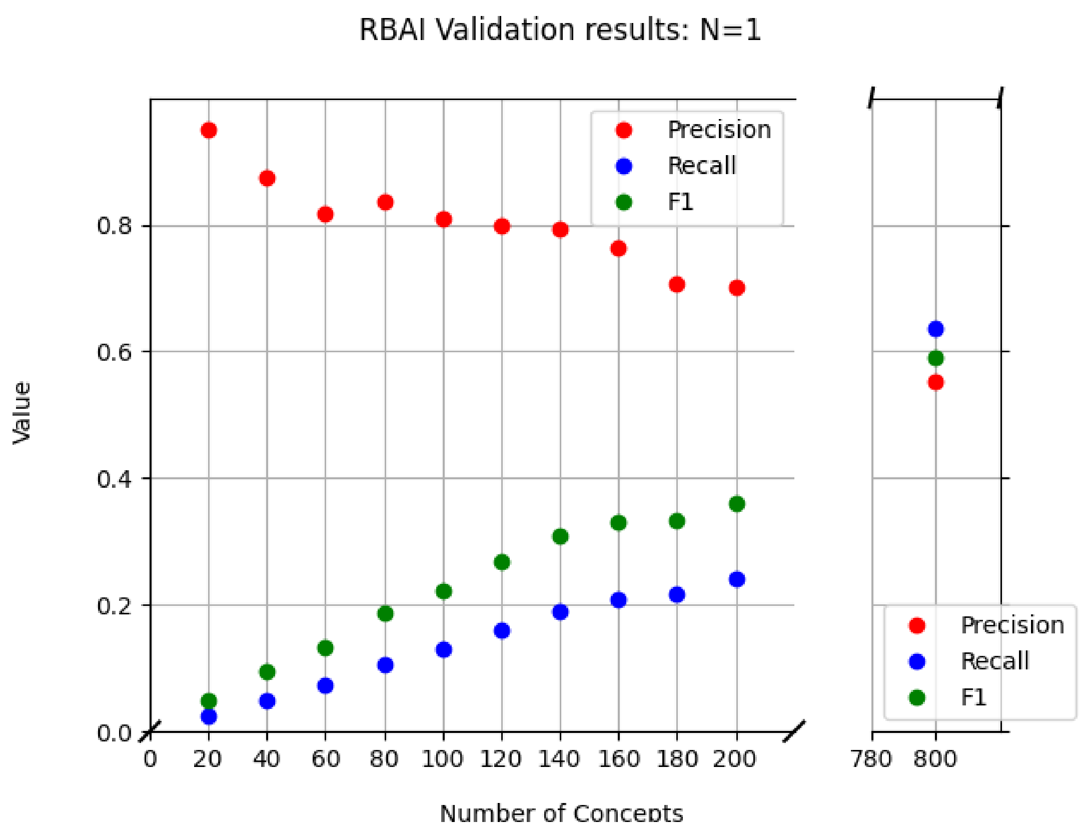
UVL[6] is a research project in the field of requirements engineering that is focused on using natural language processing and machine learning methods to develop a language model that reproduces the view that the end-user has on software.

As part of this software project, the two natural language processing algorithms **Finding Comparatively Important Concepts**[11] (FCIC) and **Relevance-based Abstraction Identification**[12] (RBAI) were implemented and integrated into the text analysis platform Feed.UVL [5].

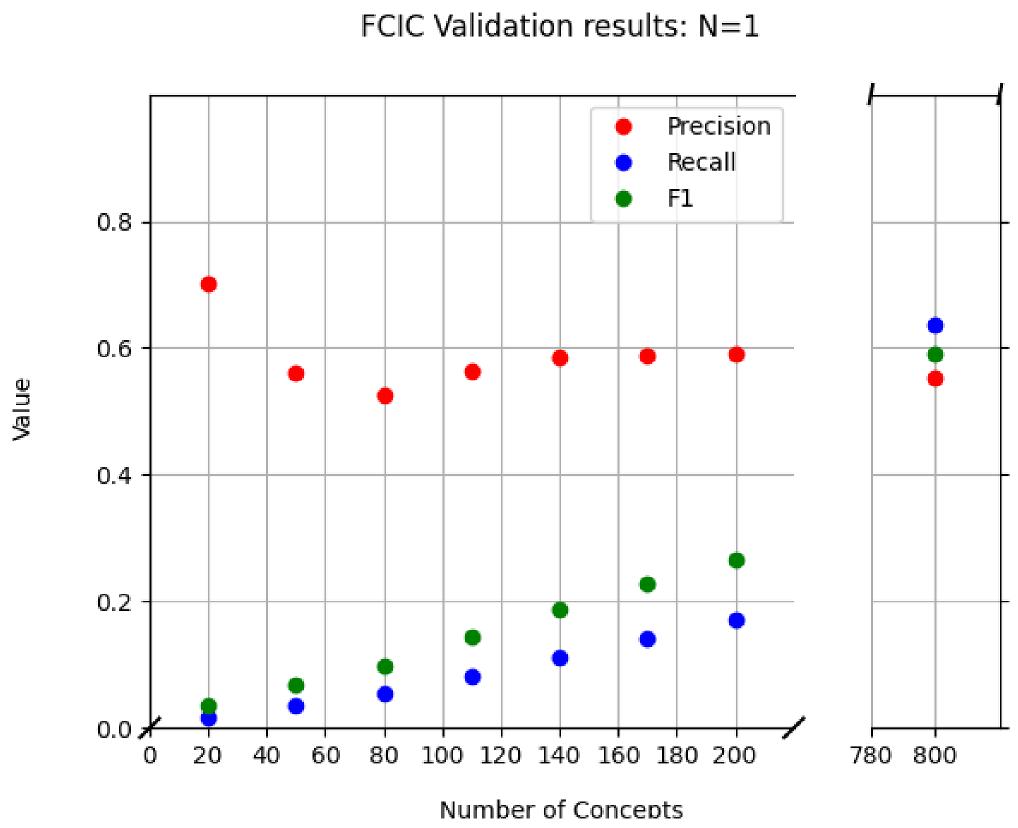
The algorithms were implemented in C++ and derive statistical language information from the *British National Corpus*[15] (XML Version)[16]. Potentially interesting concepts are derived through a frequency-based analysis and are reduced to a lemmatised form. Each algorithm outputs a ranking of the most interesting concepts.

Evaluation of the algorithms (figure 8.1) using the metrics of *precision*, *recall* and *F1-score* on a ground-truth dataset demonstrated the utility especially of RBAI in identifying key concepts in an input text. This suggests that frequency-based approaches hold promise at extracting end-user language models that may be relevant to the field of requirements engineering.

Future work may focus on the **development of domain-specific corpora**; because the BNC exclusively features texts from before the internet and digitalisation, the effectiveness of frequency-analysis-based algorithms at identifying concepts relevant for requirements engineering may be improved further through the use of corpora reflecting current-day language use.



(a) RBAI with concept length 1



(b) FCIC with concept length 1

Figure 8.1.: Validation Results

9. References

[1] G. Kotonya & I. Sommerville, *Requirements engineering: processes and techniques.* Wiley Publishing, 1998.

[2] C. Stanik, "Requirements intelligence: On the analysis of user feedback," Ph.D. dissertation, Universität Hamburg, 2020.

[3] Christoph Stanik. "Openreq." (2020), [Online]. Available: https://github.com/OpenReqEU (visited on Jul. 14, 2021).

[4] European Commission. "Intelligent recommendation decision technologies for community-driven requirements engineering." (2020), [Online]. Available: https://cordis.europa.eu/project/id/732463 (visited on Jul. 14, 2021).

[5] Sofware Engineering Group Heidelberg. "User view language." (2021), [Online]. Available: https://se.ifi.uni-heidelberg.de/research/projects/uvl.html (visited on Jul. 14, 2021).

[6] Institut für Praktische Informatik. "Uvl." (2021), [Online]. Available: https://www.pi.uni-hannover.de/de/se/forschung/projekte/forschungsprojekte-detailansicht/projects/uvl/ (visited on Jul. 14, 2021).

[7] MuleSoft LLC. "Microservices vs monolithic architecture." (2021), [Online]. Available: https://www.mulesoft.com/resources/api/microservices-vs-monolithic (visited on Jul. 11, 2021).

[8] MAXQDA. "Maxqda." (2021), [Online]. Available: https://www.maxqda.de/ (visited on Jul. 14, 2021).

[9] P. R. Christopher D. Manning & H. Schütze. "Introduction to information retrieval." (2008), [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html (visited on Jul. 14, 2021).

[10] Collins English Dictionary. "Definition of lemmatisation." (2021), [Online]. Available: https://www.collinsdictionary.com/dictionary/english/lemmatize (visited on Jul. 13, 2021).

[11] R. Lecoeuche, "Finding comparatively important concepts between texts," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, 2000, pp. 55–60. DOI: 10.1109/ASE.2000.873650.

[12] R. Gacitua, P. Sawyer, & V. Gervasi, "Relevance-based abstraction identification: Technique and evaluation," *Requir. Eng.*, vol. 16, pp. 251–265, Sep. 2011. DOI: 10.1007/s00766-011-0122-3.

[13] L. Kof, R. Gacitua, M. Rouncefield, & P. Sawyer, "Concept mapping as a means of requirements tracing," October 2010, pp. 22 –31. DOI: 10.1109/MARK.2010.5623813.

[14] M. Robeer, G. Lucassen, J. M. E. M. van der Werf, F. Dalpiaz, & S. Brinkkemper, "Automated extraction of conceptual models from user stories via nlp," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, 2016, pp. 196–205. DOI: 10.1109/RE.2016.40.

[15] Oxford Text Archive, IT Services, University of Oxford. "British national corpus." (2015), [Online]. Available: http://www.natcorp.ox.ac.uk/ (visited on Jul. 16, 2021).

[16] Oxford Text Archive. "British national corpus - xml version." (2007), [Online]. Available: http://www.natcorp.ox.ac.uk/XMLedition/ (visited on Jul. 11, 2021).

[17] M. Kalicinski. "Rapidxml." (2006), [Online]. Available: http://rapidxml.sourceforge.net/ (visited on Jul. 11, 2021).

[18] Jakob Weichselbaumer. "Github: Uvl-analytics-concepts-frequency." (2021), [Online]. Available: https://github.com/feeduvl/uvl-analytics-concepts-frequency (visited on Jul. 12, 2021).

[19] E. You. "Vue." (2014 - 2021), [Online]. Available: https://vuejs.org/ (visited on Jul. 12, 2021).

[20] Wikipedia. "Pipeline (computing)." (2021), [Online]. Available: https://en.wikipedia.org/wiki/Pipeline_(computing) (visited on Jul. 13, 2021).

[21] M. Měchura. "Github: Lemmatization-lists." (2021), [Online]. Available: https://github.com/michmech/lemmatization-lists (visited on Jul. 13, 2021).

[22] N. Iyer. "Github: Decision-tree-implementation." (2021), [Online]. Available: https://github.com/nikhil-iyer-97/Decision-Tree-Implementation (visited on Jul. 13, 2021).

List of Figures

List of Tables

Appendices

A. Cleaned and Preprocessed Target Concepts

['have two rough areas ', ", 'opening timeslots ', 'opening ', 'you can look ', 'all this here ', 'where you can click ', 'message ', 'with the teacher ', 'new view ', 'choose ', 'enrolled ', 'here the announcement ', 'can get ', 'digitally ', 'different subsections ', 'lecture notes ', 'now have calendar view ', 'browser ', 'discussion forums ', 'hear each other ', 'weeks ', 'faculty ', 'doc ', 'also have evaluations ', 'these chapters ', 'the title ', 'are listed already ', 'announcement ', 'between both groups ', 'room information ', 'window actually the connecting visual element ', 'you press ', 'almost nothing filled yet ', 'what there ', 'sheets paper ', 'you have selected earlier ', 'few buttons ', 'presentations ', 'emblem the university ', 'relating documents ', 'the options ', 'assignment ', 'you can send ', 'page ', 'you have written that ', 'great demand ', 'swap offer any offers for help job offers ', 'the next cable ', 'get something ', 'start page ', 'name ', 'adding ', 'lecturer ', 'you have scroll and down ', 'identify yourself ', 'there material yet ', 'learning materials ', 'online ', 'the selected ', 'exercises ', 'collects ', 'the middle ', 'several semesters ', 'handed out ', 'here above ', 'large box ', 'voice memos ', 'start screen ', 'tab ', 'there the start ', 'hand ', 'have them home ', 'assignment due soon ', 'that like user interface ', 'times ', 'click ', 'you also have overview ', 'contain further files ', 'click them ', 'different functions ', 'documents ', 'provided ', 'other the data center ', 'different subjects ', 'mainly call ', 'the script ', 'things they have published ', 'have written you ', 'semesters ', 'buttons ', 'someone else ', 'that use ', 'watch ', 'material ', 'learning ', 'that directly integrated ', 'main page ', 'school ', 'time ', 'school days ', 'then the lecture ', 'person ', 'would open ', 'have the overview ', 'the student ', 'get them answered ', 'you can enroll there ', 'this platform ', 'worksheets ', 'find ', 'upper right corner ', 'the right seminar ', 'also possibility ', 'are suggested you ', 'options ', 'communicate ', 'credit ', 'redundance ', 'sub items ', 'tutors ', 'learning contents ', 'for questions ', 'frontend software ', 'are marked ', 'connection ', 'get into ', 'the individual meetings ', 'both groups ', 'heidelberg university ', 'things that refer the user view ', 'clicked ', 'data ', 'announcement forum ', 'button ', 'categories ', 'polls ', 'essays ', 'communication ', 'solution ', 'mark them color ', 'down the right ', 'were held ', 'from home ', 'you can communicate with ', 'new assignment ', 'job offers ', 'where you see ', 'can talk ', 'feedback ', 'can access ', 'job training ', 'web ', 'the course ', 'you have entered ', 'have uploaded ', 'everybody can have ', 'there not only one server ', 'certain synchronous order ', 'these people ', 'view ', 'lists ', 'slides ', 'new things will uploaded ', 'individual entries ', 'for the respective week ', 'you can learn things ', 'piece paper ', 'mails ', 'services ', 'press ', 'your faculty ', 'support ', 'see what one still missing ', 'can upload them there ', 'two larger banners ', 'learning content ', 'platform ', 'modules ', 'video conferences ', 'bulletin board ', 'classified ads ', 'names ', 'voice notes ', 'there ', 'uploaded ', 'you not only can see ', 'have this course ', 'several things them ', 'switch ', 'course ', 'you can call ', 'kind meet ', 'video ', 'the top ', 'already have selected ', 'work with ', 'the upper right ', 'internet page ', 'you could arrange ', 'chats ', 'provide ', 'upper menu ', 'something that you take ', 'the right course ', 'direct contact ', 'call someone ', 'interface ', 'papers ', 'these courses ', 'activities ', 'they offer ', 'participants ', 'cable ', 'files ', 'meeting ', 'direct ', 'how many credits you got ', 'exchange with ', 'select ', 'the faculties ', 'adapt ', 'linked ', 'internet ', 'open ', 'screen ', 'the week ', 'get more ', 'protected ', 'personal assistant ', 'can entered ', 'paper ', 'problems ', 'headers ', 'the materials ', 'option ', 'module ', 'cameras ', 'whoever maintains moodle ', 'internal messages ', 'the studies ', 'listed ', 'browser bar ', 'how find the lecture room ', 'your teachers ', 'messages ', 'here the side ', 'pdf ', 'different semesters ', 'last entry ', 'the lecture dates ', 'somewhere ', 'can correspond with ', 'different options ', 'window ', 'and lecturers ', 'moodle ', 'account ', 'semester ', 'attend ', 'the beginning ', 'within this discussion forum ', 'moodle logo ', 'enters ', 'selection ', 'stud thing ', 'login ', 'contact information ', 'can this ', 'what happens which week ', 'takes place ', 'the different lectures ', 'books ', 'week week ', 'bar ', 'lectureres ', 'way distribute ', 'upload ', 'register ', 'call frontend ', 'picture ', 'the headers ', 'email ', 'can provide ', 'access ', 'the forum ', 'with whom you want communicate ', 'you can choose ', 'with the lectures ', 'other participants ', 'download ', 'different group ', 'then load ', 'others ', 'you somehow get ', 'big bar ', 'you can log ', 'the tab ', 'central ', 'that you have observe ', 'kind program ', 'students ', 'discussion forum ', 'changes ', 'entered ', 'you get ', 'registered ', 'the chronologic ', 'other people ', 'get ', 'have been added ', 'preferences ', 'the name ', 'distance ', 'evaluation ', 'work materials ', 'you can start ', 'feature ', 'build like ', 'pupils ', 'overview ', 'want have the evaluation ', 'assignments ', 'are shown there ', 'automatically ', 'the material ', 'document ', 'the left side ', 'studies ', 'reminder ', 'password ', 'transfer ', 'you don know how this function works exactly ', 'they can upload ', 'date ', 'simply scripts ', 'session ', 'participant ', 'pictures ', 'that the current semester ', 'the same possibility ', 'last semester ', 'recordings ', 'the university ', 'organization ', 'lectures ', 'different types people ', 'submissions ', 'have dates ', 'the right side ', 'the timetable ', 'the startpage ', 'authenticate ', 'can work ', 'everyone ', 'classes ', 'the internet ', 'language ', 'timetable ', 'own courses ', 'video recordings ', 'seminar ', 'exercise sheet ', 'can organized ', 'schedule ', 'chosen ', 'make appointments ', 'the overview ', 'can also submit ', 'there are different options ', 'corresponding dates ', 'kind email program ', 'lesson ', 'faculties ', 'homescreen ', 'can seach for ', 'post ', 'there faculties ', 'welcome text ', 'tells you ', 'not open for everybody ', 'bulettin board ', 'other students ', 'that means online ', 'the start page ', 'the background ', 'you have listed ', 'collection materials ', 'use ', 'uploading ', 'meetings ', 'everything online ', 'work ', 'the bottom ', 'enroll ', 'you can then print out ', 'evaluations are due ', 'lecturers ', 'can look ', 'welcome screen ', 'individual meetings ', 'what really see ', 'not everybody can just enter ', 'entries ', 'meet ', 'does not matter ', 'the front page ', 'function ', 'introductory text ', 'uploads ', 'for submission ', 'profile ', 'there timetable ', 'these are the functions that are most important ', 'evaluated ', 'lecture slides ', 'call ', 'when you choose this ', 'still use them now and then ', 'write ', 'orientation ', 'few faculties ', 'which linked ', 'the tutors ', 'this website ', 'functions ', 'log ', 'you can also download ', 'just server well ', 'runs parallel ', 'different faculties ', 'teachers ', 'home ', 'lecture ', 'links ', 'the header ', 'print ', 'this the overview ', 'cannot get there ', 'can ', 'server ', 'make information available you ', 'via moodle ', 'currently called ', 'main box ', 'word ', 'wireframe ', 'read ', 'you had the past ', 'the lectures ', 'below that ', 'questions ', 'homework ', 'when the next date ', 'the end ', 'take place ', 'you attend ', 'photos ', 'content ', 'big folder ', 'pdfs ', 'part the backend ', 'this has been updated recently ', 'news ', 'offer ', 'arranged ', 'you can choose between ', 'professors ', 'material accumulates ', 'forum ', 'used ', 'there search bar ', 'see ', 'you can see ', 'home screen ', 'have register ', 'behind ', 'detailed ', 'search bar ', 'these should all functions ', 'your lecturer ', 'rooms ', 'chat function ', 'here have chronologic ', 'are registered ', 'from time time something lights ', 'you log ', 'setting ', 'then have the address ', 'you can see things ', 'work assignments ', 'list offer request and such ', 'the new assignments ', 'you can navigate ', 'text ', 'pages ', 'title ', 'videos ', 'super well structured ', 'there platform ', 'subjects ', 'box ', 'personal companion ', 'can answer ', 'the upper left side ', 'offers request job offers things know tutoring and transfer ', 'the website ', 'uses php ', 'special website ', 'weekly view ', 'have download ', 'learning platform ', 'called ', 'can hand them there ', 'have get ', 'the submissions ', 'menu bar ', 'make them available ', 'and then the top ', 'smartphone ', 'places ', 'can organised ', 'you can clearly see ', 'send ', 'pity that there uniform setup ', 'immediately shown ', 'the profile ', 'look things ', 'you can upload ', 'forums ', 'backend software ', 'these two side panels ', 'titles ', 'chronological order ', 'chat ', 'take part ', 'can just click ', 'enter ', 'are being uploaded ', 'areas ', 'the beginning the semester ', 'storage ', 'appointment ', 'written with ', 'submission ', 'you can open ', 'work sheets ', 'the side ', 'other files ', 'the message function ', 'internet connection there ', 'you can ask questions ', 'the courses ', 'put these things ', 'what comes next ', 'part ', 'can only accessed via ', 'there also the possibility ', 'the teacher ', 'you can then listen ', 'you can show things ', 'new things ', 'that you have selected ', 'also certain assignments ', 'until you find something ', 'below ', 'received ', 'clipboard ', 'class ', 'overviews ', 'privat ', 'what time ', 'mail ', 'here ', 'enlist ', 'you can unsubscribe ', 'scheduling ', 'the left ', 'write someone ', 'link ', 'database ', 'usually have ', 'you can also log yourself ', 'the same possibility manage ', 'literature ', 'all the weeks ', 'power point presentations ', 'when you have downloaded ', 'info box ', 'can log ', 'additional functions ', 'here are the times ', 'down here ', 'virtual ', 'information ', 'other courses ', 'directly integrated ', 'computer ', 'you can select ', 'seminars ', 'see each other face face ', 'this page ', 'not present ', 'you get small view ', 'list ', 'and which grade ', 'the information ', 'where teachers ', 'keep track ', 'calendar function ', 'dashboard ', 'there the right ', 'security ', 'your courses ', 'from server ', 'you have direct access ', 'the right ', 'can easily multiplied ', 'passwords ', 'usually navigate back and forth ', 'users ', 'many other functions ', 'you can then see ', 'the weeks ', 'details ', 'the next dates ', 'added ', 'signing out ', 'methods communication ', 'then here only have the overview ', 'can upload things ', 'that are using ', 'word documents ', 'social network ', 'user ', 'there also this dicussion forum ', 'exercise ', 'you can get ', 'the individual courses ', 'the address bar ', 'courses ', 'alarmbell ', 'cancelled ', 'each other ', 'can delivered ', 'get back ', 'another page ', 'icons ', 'short information ', 'file ', 'like forum ', 'sometimes checked automatically ', 'can upload ', 'calendar ', 'looking glass ', 'collect ', 'checkmarks ', 'you can store ', 'there are these other functions ', 'handed ', 'looks ', 'things know ', 'that currently selected ', 'the seach bar ', 'drawn ', 'then there are evaluations ', 'distribution ', 'the material still missing ', 'important function sort things ', 'that all can access ', 'the pupil ', 'where you will forwared ', 'lock ', 'newly ', 'ask ', 'you had lists ', 'corresponding solutions ', 'register for ', 'kind screen ', 'this the second page ', 'you can scan ', 'opens ', 'sometimes not chronological ', 'the other side ', 'the leaders the course ', 'time table ', 'this the main page ', 'you can exchange with ', 'first screen ', 'make sure get ', 'you enter ', 'going ', 'gives certain orientation ', 'sessions ', 'emails ', 'usually heiconf ', 'the possibility ', 'methods ', 'look ', 'would add third component ', 'texts ', 'ascending descending order ', 'presence ', 'the link ', 'you have various options ', 'survey ', 'hyperlinks ', 'you can kind check ', 'you can also communicate ', 'relatively many threads ', 'have read ', 'directly ', 'have the navigation ', 'then the right side ', 'shown ', 'the chapters ', 'that you did ', 'enrolment ', 'you want take ', 'you can look together ', 'subfolders ', 'folders ', 'discuss things ', 'website ', 'tool ', 'the different weeks ', 'just ordinary links ', 'rectangular window ', 'the teachers ', 'find out all kinds things ', 'you are inside ', 'university contents ', 'miscellaneous ', 'request ', 'sheet ', 'user name ', 'what you find ', 'with the weeks ', 'exams ', 'offers ', 'materials ', 'talk others ', 'you have kind quick access ', 'available ', 'fellow student ', 'switch button ', 'content delivery network ', 'give ', 'can then hand ', 'you then have the meeting ', 'the file ', 'exchange ', 'chest drawers ', 'navigate ', 'event ', 'have log ', 'corresponds ', 'has been running ', 'subject ', 'then here ', 'fulfil ', 'laptop ', 'individual area ', 'you can click ', 'call them ', 'alarm clock ', 'many faculties ', 'search ', 'announcements ', 'studip ', 'can view them ', 'the left and right ', 'deadline ', 'folder ', 'that means account ', 'are shown ', 'website content ', 'where courses are ', 'the initial screen ', 'sheets ', 'logged ', 'that can call ', 'some materials ', 'pdf file ', 'this the site ', 'framework ', 'typical view ', 'the monitor ', 'the software divided ', 'kind register ', 'layout ', 'upper part ', 'found ', 'software ', 'talk ', 'for this course ', 'this varies from course course ', 'university lecturers ', 'data exchange ', 'university ', 'separate ', 'faculities ', 'reload ', 'people ', 'student ', 'learn ', 'stud ', 'sidebar ', 'room ']