Označavanje, izlučivanje i prikazivanje ključnih riječi iz višejezičnih tekstova

Aljević, Dino

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:593934

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-06**

University of Rijeka – Department of Informatics

Jednopredmetna informatika

Dino Aljević

Keyword tagging, extraction and visualisation from multilingual texts

Bachelor's thesis

Mentor: prof. dr. sc. Sanda Martinčić - Ipšić dipl. ing.

Rijeka, 18.9.2019.

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetna informatika

Dino Aljević

Označavanje, izlučivanje i prikazivanje ključnih riječi iz višejezičnih tekstova

Završni rad

Mentor: prof. dr. sc. Sanda Martinčić - Ipšić dipl. ing.

Rijeka, 18.9.2019.

Abstract

Application described in this thesis is a web application which performs keyword extraction and visualisation on texts written in Croatian, English and Italian language. Keyword extraction is a process of determining which terms best describe a given document with a caveat that such terms must be present in the document itself. Manual keyword extraction is a very time consuming and tedious task making it a perfect candidate for automation.

Application is written in Python on top of Django framework and uses Maui topic indexing algorithm developed at the University of Waikato. The thesis serves as a documentation of practical use of the algorithm focused on keyword extraction task without the use of controlled vocabulary and as a documentation of the application architecture and design decisions behind it.

Keywords: *keyword extraction, Maui, web application*

Sažetak

U ovome radu je opisana web aplikacija koja izlučuje i vizualizira ključne riječi iz tekstova napisanih na hrvatskom, engleskom i talijanskom jeziku. Izlučivanje ključnih riječi je proces određivanja termina koji najbolje opisuju dani dokument uz uvjet da se termini moraju pojavljivati u samom dokumentu. Ručno izlučivanje ključnih riječi je dugotrajan i zamoran posao, pa je njegova automatizacija poželjna.

Aplikacija je napisana u programskom jeziku Python na temelju Django frameworka i koristi Maui topic indexing algoritam razvijen na Sveučilištu u Waikatu. Ovaj završni rad služi kao svojevrsna dokumentacija korištenja algoritma u postupku izlučivanja ključnih riječi bez uporabe vokabulara i dokumentacija arhitekturu aplikacije, te odluka koje stoje iza nje.

Ključne riječi: izlučivanje ključnih riječi, Maui, web aplikacija

Table of Contents

1. Introduction

Keyword extraction is a process of determining which terms best describe a given document with a caveat that such terms must be present in the document itself. Keywords can be used in wide array of tasks such as: (Beliga et al., 2015)

- automatic generation of document index,

- querying in information retrieval systems (e.g. search engines),

- categorisation and classification (e.g. categorising news articles).

Manual keyword extraction is a time consuming and tedious task which makes it a perfect candidate for automation. It is no surprise then, that a number of different keyword extraction algorithms have been developed in the past. Algorithms are usually supervised or unsupervised, using standard machine learning methods (Beliga et al., 2015). Recently, graph enabled keyword extraction has gained much attention.

The algorithm featured in the thesis is Maui (Medelyan, 2009). The algorithm has been chosen because of it's human-competitive performance and proven multilingual capabilities.

The first part of the thesis will describe practical use of the algorithm in the keyword extraction process including the steps needed to adapt it to work with Croatian and Italian language.

The second part discusses the web application's architecture focusing on the different key components that make up the application. This part also glosses over the steps needed to implement additional algorithms.

2. Methodology and background terminology

This chapter aims to clear up confusion by bringing up similarities and differences between terms revolving around information extraction and retrieval. In addition, it lists metrics traditionally used to evaluate information retrieval systems.

Information retrieval and information extraction may sound identical, but the former pertains to retrieval of documents, often as the result of answering a query. The latter refers to acquiring information from a text (unstructured data) by looking for a particular class of object and it's relationships. (Russel & Norvig, 2009)

Keyword extraction algorithms examine different words in text, choosing them based on their properties (e.g. frequency, length, etc.). Thus we can conclude that keyword extraction falls under the latter category.

Another pair of terms which appear to be the same thing when not looking beneath the surface are term assignment and keyword extraction. While both serve ultimately the same goal, the underlying principle is different.

In term assignment we assign keywords to a document based on controlled vocabulary. In keyword extraction, controlled vocabulary may or may not be used, however, a key difference is that extracted keywords must be present in the original text. Topic indexing is a more general term that encompasses both term assignment and keyword extraction (Medelyan, 2009).

Another way of assigning terms to a document is by tagging. No criteria are placed on a word used as a tag and therefore it can be chosen freely. Tagging is mainly used on collaborative sites (e.g. YouTube content, blog posts) (Medelyan, 2009).

Performance of IR systems is measured by scoring set of queries and corresponding result sets with respect to human judgement. Traditionally, two measures have been used: precision and recall (Russel & Norvig, 2009).

The standard formula for precision P, and recall R as defined in (Medelyan, 2009) are:

$$P = \frac{\# \, correct \, extracted \, topic}{\# \, all \, extracted \, topics}$$

$$R = \frac{\# \, correct \, extracted \, topic}{\# \, manually \, assigned \, topics}$$

(Medelyan, 2009) also provides formula for F-measure, a combination of precision, recall and a factor β which can be used to give more significance to either precision or recall.

$$F_\beta = (1+\beta) \frac{PR}{\beta^2 P + R}$$

By assuming β = 1, it is giving equal significance to both precision and recall. Thus we get F_1-measure as a harmonic mean of recall and precision:

$$F_1 = (1+1) \frac{PR}{1^2 P + R} = \frac{2PR}{P+R}$$

2.1. Maui and Kea

Maui is topic indexing algorithm built on top of Kea algorithm, inheriting many of Kea's components. Unlike Kea whose sole purpose was keyword extraction; Maui can also perform term assignment using controlled vocabulary or Wikipedia (Medelyan, 2009).

The algorithm can be summarized in four steps: (Medelyan, 2009)

1. generate candidate topics,

2. compute their features,

3. build topic indexing model and

4. apply the model in topic indexing tasks.

Maui inherits supervising learning approach from Kea when it comes to building the model.

(Russel & Norvig, 2009) define supervised machine learning as observing input-output pairs in order to determine a function that maps input to output. In contrast, in unsupervised learning an agent attempts to discover patterns without explicit labelling of the input.

In Maui, input-output pairs consist of a document and manually assigned keywords.

As is often the case, before applying (supervised) machine learning one has to prepare (annotate) the data used in the process.

3. Implementation of keyword extraction

3.1. Data preparation

In data preparation process documents containing texts in Croatian, English and Italian language from TriKEDS corpus (Beliga, 2019) were used. Initially all documents to UTF-8 encoding and cleaned up.

The following steps were performed:

1. all documents were converted to UTF-8 encoding.

2. duplicate keywords were removed and

3. encoding errors were fixed.

First two steps were done automatically using scripts written in Python, encoding errors were fixed by hand, but offending characters were located using Python script.

First two steps are very straightforward, so let us focus on the third.

Documents used to build the model were encoded differently. *file* program present on most Linux distributions was used to determine the encoding of original documents. The following encodings were reported:

- ASCII,

- ISO-8859-1,

- UTF-8 Unicode,

- UTF-8 Unicode with byte order mark and

- Non-ISO extended-ASCII (unknown).

In cases where document encoding had been determined, it was simply a matter of reading the document in it's encoding and writing it back in UTF-8. Non-ISO extended-ASCII encoding was assumed to be Windows-1250. The assumption is an educated guess based on the prevalence of Windows operating system on desktop and the fact that Windows-1250 is used to represent texts in Eastern European languages.

In the original corpus, some of the characters have been incorrectly written or encoded, for example, surnames ending in "-ić" written as "-iæ" or "srđ" encoded as "srð". Because keywords from multiple sources pertaining to the same texts had been collated into one text file, the text file ended up having both the correct spelling and the incorrect one. A simple algorithm was devised to locate such misspellings.

1. For every token in file iterate over tokens appearing after it.

2. If a token can be encoded in ASCII, skip it because it's not the one containing non English characters.

3. If Levenshtein distance between token is 1, it is a possible misspelling. Indeed, Levenshtein distance between "srđ" and "srð" is exactly 1.

4. Output the location of the tokens in file.

The algorithm is not sophisticated, it picks up valid tokens like "Riječke" and "Riječka" as potential misspellings, but the set of potential misspelled words was small enough to make this algorithm and subsequent manual corrections reasonable. Although the whole process could be improved and automated from start to finish, the solution was efficient enough for task at hand.

3.2. Building the model

As previously mentioned, Maui expects pairs of text documents and keywords to build the model. Files containing keywords should have the same file name as the corresponding text, but with *.key* extension instead of *.txt*.

A model can be built by running *MauiModelBuilder*. It requires three parameters:

- *-l <directory name>* specifies path to the directory containing *.txt* and *.key* files,

- *-m <model name>* specifies file path where generated model will be saved,

- *-v <vocabulary name>* specifies path to the vocabulary to use. If vocabulary is not used, -v none should be passed instead[1].

---

1   Newer versions of Maui don't require explicit -v none, it is implied instead.

Assume class files are stored in *bin* directory, required libraries in *lib* and *.txt* and *.key* files in *data*, we can build a model by executing following command in the terminal of our choice:

```
java -cp "lib/*:bin" maui.main.MauiModelBuilder -l data -m model -v none
```

3.3. Extracting keywords

Assume the same directory structure as in the previous section and assume we want to extract keywords from a text file located in *target* directory containing a single file. We can run the following command:

```
java -cp "lib/*:bin" maui.main.MauiTopicExtractor -l target -m model -v none
```

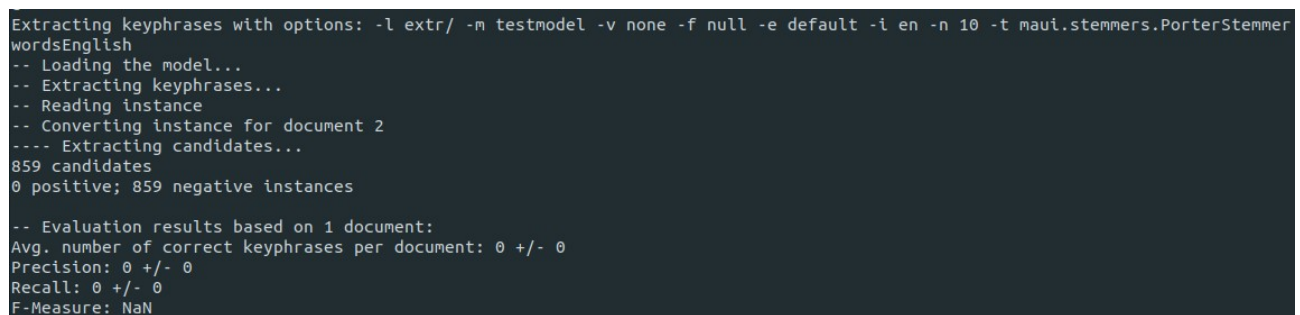Output will look similar to the figure below:

```
Extracting keyphrases with options: -l extr/ -m testmodel -v none -f null -e default -i en -n 10 -t maui.stemmers.PorterStemmer
wordsEnglish
-- Loading the model...
-- Extracting keyphrases...
-- Reading instance
-- Converting instance for document 2
---- Extracting candidates...
859 candidates
0 positive; 859 negative instances

-- Evaluation results based on 1 document:
Avg. number of correct keyphrases per document: 0 +/- 0
Precision: 0 +/- 0
Recall: 0 +/- 0
F-Measure: NaN
```

Figure 1: MauiTopicExtractor output

Extracted keywords are stored in *.key* files. If .key files already exist, Maui will use them as a gold standard to evaluate the performance of extracted keywords.

We can use -d parameter to turn on debugging output. Among other information, Maui will also output extracted keywords. The Python application parses (debugging) output to retrieve extracted keywords and evaluation metrics.

```
Extracting keyphrases with options: -l extr/ -m testmodel -v none -f null -e default -i en -n 10 -t maui.stemmers.PorterStemmer
wordsEnglish -d
-- Loading the model...
-- Extracting keyphrases...
-- Reading instance
-- Converting instance for document 2
---- Extracting candidates...
859 candidates
10 positive; 849 negative instances
-- Processing document: 2
-- Keyphrases and feature values:
museum,Museum,0.013501,1.475907,0.019926,0.104506,0.993289,0.888782,1,1,0,0,0,0,0,0.90119,1,True
vučedol,Vučedol,0.008845,3.555348,0.031449,0.004794,0.987536,0.982742,0,1,0,0,0,0,0,0.821069,2,True
project,project,0.003259,0.154151,0.000502,0.720038,0.91371,0.193672,17,1,0,0,0,0,0,0.630128,3,True
'locat vučedol','Vučedol location',0.001397,3.555348,0.004966,0.004794,0.36721,0.362416,0,2,0,0,0,0,0,0.568638,4,True
danub,Danube,0.000931,3.555348,0.00331,0.019175,0.300096,0.28092,0,1,0,0,0,0,0,0.492347,5,True
'cultur vučedol','Vučedol culture',0.002793,3.555348,0.009931,0.024928,0.704698,0.67977,0,2,0,0,0,0,0,0.475152,6,True
architectur,Architectural,0.003259,1.070441,0.003488,0.152445,0.644295,0.49185,4,1,0,0,0,0,0,0.464757,7,True
vukovar,Vukovar,0.001862,2.862201,0.00533,0,0.088207,0.088207,0,1,0,0,0,0,0,0.402284,8,True
'site vučedol','Vučedol site',0.000931,3.555348,0.00331,0.023011,0.25791,0.234899,0,2,0,0,0,0,0,0.401971,9,True
cultur,Culture,0.005121,0.847298,0.004339,0.025887,0.840844,0.814957,1,1,0,0,0,0,0,0.393824,10,True
-- 10.0 correct
```

Figure 2: MauiTopicExtractor output with debugging turned on

Maui can also be used as a Java library, but since the web application is written in Python, using Java library is not possible[2].

3.4. Croatian and Italian languages

Maui is capable of working with languages other than English (Medelyan, 2009). This was the main reason for selecting the Maui among many available solutions for keyword extraction. Additionally, Maui provides examples for French and Spanish as well. To use a language with Maui, it requires a list of stop words and a stemmer for the language.

Stopwords are common words which do not carry strong semantic meaning, but are required by language syntax (Beliga et al., 2015). Because of their lack of semantic meaning, they are often removed during text preprocessing.

In addition to removing stopwords, words in text are often stemmed, meaning they are reduced to their root form. Such algorithms are called stemming algorithms (Segaran, 2007).

Adapting Maui to work with other languages is a rather straightforward task, even more so if you have readily available stemmers for a language you want to use. By looking at Maui source code, we can see that a stemmer is class which inherits from *Stemmer* abstract class. A stemmer class must implement *stem* method which accepts a string (word) and returns it's stemmed version.

---

2   It may be possible to use Maui as a library from Python using Py4J, but due to personal lack of experience in Java ecosystem, using it as a command line tool seemed like a simpler choice.

Stop words are implemented as a class inheriting a *Stopwords* abstract class. Such classes hold reference to file path containing a list of stop words and a *isStopword* method which accepts a string and returns boolean value depending or whether the argument is a stopword or not.

Fortunately both Croatian and Italian stemmers are available online with permissive licenses (GPL 3 and the 3-clause BSD license respectively).

Croatian stemmer was developed by Nikola Ljubešić and Ivan Pandžić (Ljubešić & Pandžić, 2012). The stemmer is a refined version of stemmer presented in (Ljubešić et al., 2007). It was originally written in Python, but Java adaptation was introduced in (Batanović et al., 2016). The code was slightly modified to satisfy Maui's interface requirements and unnecessary code was removed (e.g. Cyrillic to Latin conversion).

Italian stemmer is a part of libstemmer library originally written by Dr. Martin Porter and later adapted for Java by Richard Boulton (Porter & Boulton, 2002). Using the stemmer in Maui is a simple matter of writing a stemmer class which calls libstemmer to perform the actual stemming.

Stopword classes are implemented in the same way as Maui's *StopwordsEnglish* class with path to file containing stop words hard coded in the it's constructor.

With everything in place, we can tell Maui to use new stemmer class by passing *-t* parameter when executing either *MauiModelBuilder* or *MauiTopicExtractor*. Stopwords class can be specified using *-s* parameter.

For completeness, here is an example of how we can build a model using *CroatianStemmer* and *StopwordsCroatian* classes.

```
java -cp "lib/*:bin" maui.main.MauiModelBuilder -l data -m model -v none
-t CroatianStemmer -s StopwordsCroatian
```

4. Application architecture

Maui is only a part (an important part) of a larger web application written in Python on top of Django web framework (Django, 2019) and a host of other web development technologies. The choice of framework and keyword extraction algorithm had great influence on the overall design of the application itself.

Following sections will discuss application architecture, starting from Django framework and the way Django projects are structured, narrowing down to the specific components and program units.

4.1. Project structure

Django projects consist of a project configuration and a single or multiple applications where each application performs some task in the overall web application. The term "application" may be a bit misleading here. Each Django application is just a Python package that provides set of features, interacts with the rest of the framework and often has it's own configuration. To avoid confusion, here we refer to the overall web application built on top of Django framework as web application or project and the aforementioned Python packages as Django applications.

Because individual Django application performs some task, it is logical to separate multiple applications based on some criteria, e.g. area of responsibility. This approach of creating decoupled, reusable application is even encouraged by Django and it's DRY[3] approach to development. After all, isn't it better to write an application once and then reuse it across different projects than to write the same application over and over again?

When considering how to separate the web application into multiple areas of responsibility, this approach presented itself as the most efficient:

- User interface and user interaction are one area of responsibility; therefore it will be a responsibility of a single Django application. This application also handles form validation and algorithm selection. Because this application serves as an entry point it has been named "core".

---

3   Don't repeat yourself, software engineering principle of reducing repetition of software patterns.

- Keyword extraction and visualisation will be another application aptly named "kwextractor".

Let us consider the benefits of such approach:

1. Keyword extraction application can be reused across different projects.

2. Because the application doesn't have any views or templates, only model classes, we can do away with user interface entirely and implement keyword extraction differently (e.g. web API).

The question now is whether keyword visualisation should be a part of *kwextractor*, *core* or it's own application? The reason keyword visualisation functionality has been included in *kwextractor* application is grounded in model migrations and will be discussed later.

4.2. Database

Almost every application needs a way to store data it works with and web application presented in this thesis is no different. The web application uses SQLite 3 (SQLite, 2018) for storing text, keywords and other relevant data.

Initially, it was intended for text to be stored in the same JSON file along with related data such as keywords, word cloud image path, etc. In the end, the idea was scrapped in favour of using SQLite database. There are three primary reasons for that decision:

1. SQLite is already supported by Django framework and works nicely with Django object relational mapper. This avoids writing the code tasked with CRUD[4] operations on JSON files from scratch.

   Furthermore, text files and keywords have clear relationship and any other scheme would either mimic relational model or be close enough not to warrant sidestepping DRY principle.

2. SQLite is extremely lightweight compared to other relational database management systems such as PostgreSQL (PostgreSQL, 2019). The whole database is stored in a single file.

3. We can leverage Django's migrations feature.

---

4   Create, read, update and delete.

To elaborate on reason number three, one needs to know a bit about Django's models.

In Model-View-Controller (MVC) frameworks such as Django[5] models represent definitive source of information about data and include any logic which operates on said data. Models are also used by Django to create and maintain database, mapping each model to a single table. In other words, model definitions determine final database layout.

Django keeps track of changes made to the models and database using migrations. Django's documentation compares migrations to version control for database schemas. They are invaluable tool for keeping database consistent across different revisions, especially in the early stages when final database layout is not known.

---

5   Actually, Django is a Model-Template-View framework: the view decides which data to present, template specifies how the data should be presented and the controller part is handled by the framework itself.

4.3. Keyword extraction

While Maui is de facto algorithm for keyword extraction used in this (web) application, the application has eben designed with multiple algorithms in mind and can be extended with almost zero effort spent on code refactoring. This is facilitated by clean design where working components depend on abstractions rather than concrete implementations.

The following UML class diagram describes relationship between classes working together to extract keywords.
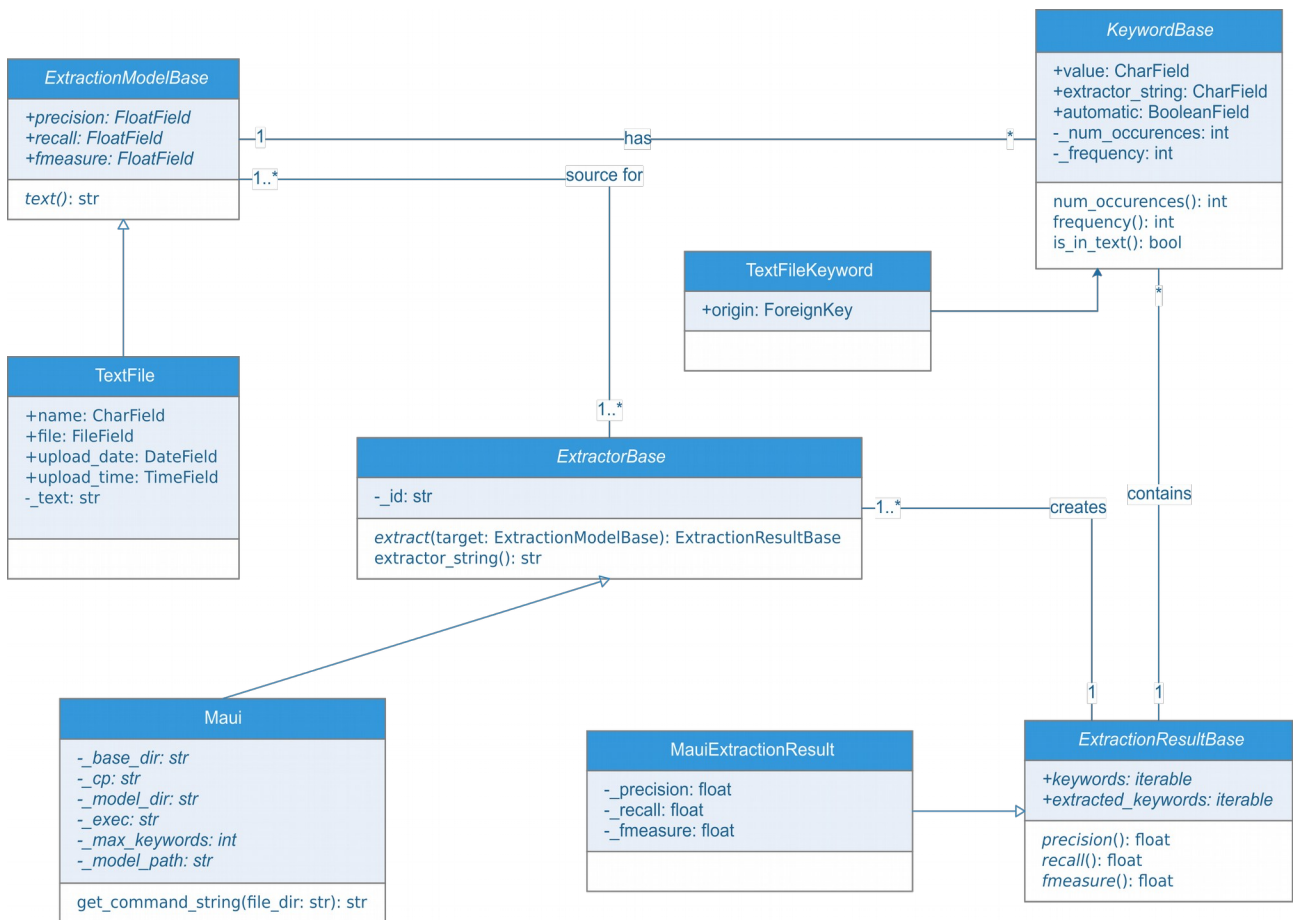

Figure 3: UML class diagram of classes related to keyword extraction task

To understand what is going on, we can look at the way the web application calls Maui to perform extraction task. Extraction algorithms are implemented in extractor classes which inherit from *ExtractorBase*. Extractors use ExtractorModelBase as the source of text from which to extract keywords creating a collection of *KeywordBase* subclasses. Because both *ExtractionModelBase* and *KeywordBase* are Django models

and need to be bound by foreign key, it is not possible to have a generic keyword class.

extract method returns concrete *ExtractorResultBase* from which the view will read evaluation metrics and save the metrics in the extractor model.

The whole process begins upon submitting a form. The corresponding view fetches extraction model from the database (i.e. text file), instantiates *Maui* extractor and calls it's *extract* method to perform keyword extraction. The important thing to note here is that view automatically assumes Maui as the extraction algorithm, it doesn't consider any other algorithm. Refactoring this view so that it selects an algorithm based on input would be the first step in introducing new extraction algorithms.

To summarise, in order to expand algorithm selection, we need to:

1. Refactor the view so it can select keyword extraction algorithm based on user input. If done correctly, this has to be performed only once.

2. Related to the first point, we need to update user interface to allow user interaction.

3. Most importantly, implement the algorithm by subclassing *ExtractorBase*.

Maui extractor performs couple of things. First it creates *.txt* from which to extract text and *.key* file to use for evaluation. These files are created based on data stored in the extractor model. The extractor then runs Maui algorithm as a subprocess, passing the correct arguments and parsing the output.

How does the extractor determine what command line arguments to pass to Maui, specifically what model, stemmer and stopword class to use? Command line arguments can be configured in the project settings. A string passed from the form is used as an index into *MAUI_EXTRACTOR_CONFIGURATION* dictionary whose values are tuples containing path to the model and names of stemmer and stopword classes. This allows us to configure different combinations of models, stemmers and stopwords without modifying the extractor code.

The extractor needs to return an extractor result instance. Extractor result classes need to have references to assigned and extracted keywords to be able to calculate precision, recall and F-measure. Maui already outputs those metrics, so there is no need to recalculate them, instead we pass them to the *MauiExtractionResult* constructor.

We can see the importance of using Maui's *-d* parameter here. Maui outputs extracted keywords to a *.key* file if it does not exist. If it does exist, it reads a *.key* file and calculates evaluation metrics, it does not overwrite *.key* files with automatically extracted keywords. Had we not used *-d* parameter, we would have to run Maui twice, first with existing *.key* file to calculate evaluation metrics and then again without *.key* file to retrieve keywords. By using *-d* parameter, we can parse keywords from standard output instead.

Automatic keywords 5

Keyword	Number of occurrences	Frequency	Extractor string
arheološkog	3	0.05	maui
arhitekture	4	0.07	maui
kulture	7	0.11	maui
muzej	12	0.20	maui
vučedolske kulture	4	0.07	maui

Manual keywords 4

Keyword	Number occurrences	Frequency
dunav	0	0.00
muzej	12	0.20
vukovar	1	0.02
vučedol	5	0.08

Statistics

Precision	20.00
Recall	25.00
F_1 Measure	22.22

Figure 4: Text file detail view

Maui assumes F_1-measure, giving equal significance to both precision and recall. The question is, should other extractors use the same value for β? While not enforced programmatically, consistency should be favoured instead of versatility. User interface further enforces this idea by explicitly labelling the field "F_1 Measure".

4.4. Keyword visualisation

The web application uses word clouds to visualise keywords. Word clouds are rendered on the server anytime keywords are extracted or assigned. This has considerable performance impact as rendering is an intensive task.

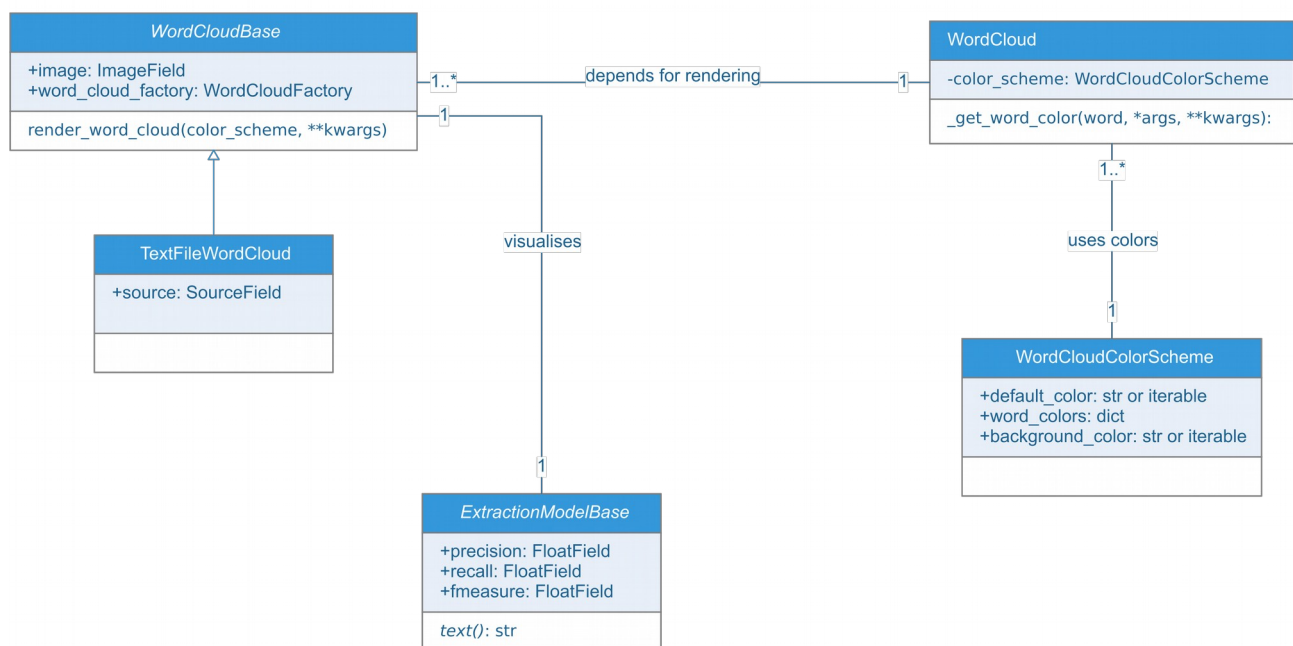Again we start with UML class diagram showing a subset of relevant classes.

Figure 5: UML class diagram of classes related to keyword visualisation

WordCloudBase is a Django model used for storing path to rendered word cloud image, as well as maintaining link to extractor model used as a source. Due to same reasons as *KeywordBase*, concrete implementations of *WordCloudBase* are bound to concrete extractor models.

WordCloudBase depends on *WordCloud* class to do actual rendering. Because we might want to swap *WordCloud* implementations in the future, creation of word cloud objects is delegated to *WordCloudFactory class*.

Final piece of the puzzle is the *WordCloudColorScheme* class. It is a simple class used to define colouring for word cloud images. Specific class is used for this task because we want to have the ability to easily swap colour schemes and introduce complex logic for colouring individual words.

Figure 6: Example of a generated word cloud

5. Conclusion

In this thesis we have discussed keyword extraction and shown example of how Maui algorithm can be used to extract keywords. We even have production ready web application capable of performing keyword extraction by leveraging different keyword extraction algorithms. Despite that, we have seen only a glimpse of natural language processing field and barely scratched the surface Maui's workings. It serves as a testament to how complex the field and the algorithms really are when we can build a full functional application without really knowing what goes behind the algorithms.

There is much more to be said, not just about natural language processing or Maui algorithm, but about the web application itself. Further work can be focused on implementing new extractors, offloading keyword visualisation to client in order to improve performance or even on improving stability and robustness. One idea is to implement some sort of user account system. In the current version the only way to limit access to the application is to configure web server authentication. Custom account system would enable interesting features such as per-user model selection, limits on text document length for each user and even allow administrators to configure the application from the interface. Perhaps, such features coupled with general improvements on all fronts could in time make the application commercially viable.

References

S. Beliga, A. Meštrović, S. Martinčić-Ipšić. (2015). An Overview of Graph-Based Keyword Extraction Methods and Approaches. Journal of Information and Organizational Sciences, 39(1)

O. Medelyan. (2009). Human-competitive automatic topic indexing (Thesis). The University of Waikato, Hamilton, New Zealand. Retrieved from https://hdl.handle.net/10289/3513

S. Russel, P. Norvig (2009). Artificial Intelligence: A modern approach. Upper Saddle River, NJ: Pearson

S. Beliga, (2019). Keyword Extraction Based on Structural Properties of Language Complex Networks (Dissertation). The University of Rijeka

T. Segaran (2007). Programming Collective Intelligence: Building Smart Web 2.0 Applications. Sebastopol: O'Reilly Media

N. Ljubešić, I. Pandžić (2012). Stemmer for Croatian. The University of Zagreb. Retrieved from http://nlp.ffzg.hr/resources/tools/stemmer-for-croatian/

N. Ljubešić, D. Boras, O. Kubelka. (2007). Retrieving Information in Croatian: Building a Simple and Effcient Rule-based Stemmer. Retrieved from http://nlp.ffzg.hr/data/publications/nljubesi/ljubesic07-retrieving.pdf

V. Batanović, B. Nikolić, M. Milosavljević. (2016). Reliable Baselines for Sentiment Analysis in Resource-Limited Languages: The Serbian Movie Review Dataset. Retrieved from http://www.lrec-conf.org/proceedings/lrec2016/pdf/284_Paper.pdf

M. Porter, R. Boulton (2002). Java libstemmer library. Retrieved from http://snowball.tartarus.org/

Django (Version 2.2) (2019). Retrieved from https://djangoproject.com/

SQLite (Version 3.22.0) (2018). Retrieved from https://sqlite.org/

PostgreSQL (2019). Retrieved from https://www.postgresql.org/

Appendix

Appendix includes CD-ROM containing a copy of the web application featured in the thesis and the accompanying documentation.

Table of Figures