Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Katedra softwarového inženýrství

Master's thesis

Recipes Detection and Recognition on the Web

Bc. Stanislav Fifik

Supervisor: Ing. Jiří Novák, Ph.D.

4th May 2015

Declaration

In Prague 4th May 2015 . . . .

Citation of this thesis

Stanislav Fifik. *Recipes Detection and Recognition on the Web: Master's thesis.* Czech Republic: Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstract

Using natural language processing tools in time and space limited environment of web browser is interesting problem, which solution can provide enhanced end user experience by augmenting website information with extra information from external sources, text categorization and useful information highlighting. One of the possible application is recipe extraction from page in user's browser.

Keywords nlp, javascript, text mining, information retrieval, web

Abstrakt

Využití technologií pro zpracování přirozeného jazyka v dnešních prohlížečích, představuje zajímavý problém s omezeními jak na paměť tak čas. Řešení tohoto problému nám otevře cestu ke zlepšení uživatelova prožitku UX pomocí doplnění relavantních informací, automatické kategorizace nebo zvýrazněním důležitých informací. Jedním z možných využití této technologie je získání receptu z uživatelem prohlížené stránky.

Klíčová slova nlp, javascript, text mining, vytěžování informací, web

Contents

List of Figures

List of Tables

Introduction

Motivation

Web is huge source of useful information, but in most cases this information are not well tagged and automatized information retrieval is hard. Microformats, microdata and RDFs are trying to change this, but we're just at the beginning of their long journey. At the current state most websites don't use these tagging methods or misuse them. By understanding information on the website, we can provide user relevant information for viewed content and allow user save relevant parts of viewed content into other storing services.

Tools nowadays provides such features, but they are usually designed for desktop or server applications. Moving these tools into user's browser decreases needed processing power on servers, minimize network communication and allows running data extraction on any site loaded in user's browser even without his direct interaction.

Personal Motivation

I'm particularly interested in recipe extraction since I'm working on a social network allowing user save recipe from any site to a single place. To search and categorize recipes on this site, I need to extract recipe ingredient, their amount and preparation steps. Using this information we can implement features like shopping lists, nutrition information, searching by diets, automatic categorizations and recipe suggestions. But to do so we need to break down the recipe, which is time-consuming for a user.

Approach

In order to extract recipe from a web page, I chose an approach using corpus linguistic and implemented elemental stack for text processing, annotating and corpus querying in JavaScript. On the top of this generic library, I built a domain-specific application for language identification and recipe extraction. In the following chapters, Libraries, NLP stack, Language identification and Recipe extraction I'll describe libraries and tools I've used, implemented library for corpus linguistic, n-gram based approach for language identification and methods used for recipe extraction, respectively.

CHAPTER 1

Introduction into NLP

Natural Language Processing (NLP) is field where computer science, artificial intelligence and linguistic meets. NLP uses text mining methods for creating some knowledge about input text. One of the used processes describes Figure 1.1. The input document is processed in pipeline of multiple methods. First the text is parsed and basic structures like paragraphs and sentences are extracted, further more granular structures, like word tokens are created. These structures are processed by Part-Of-Speech (POS) tagger which adds word syntactic information. From these annotated structures are extracted entities and finally relationship of these entities are formed.

1.1 Corpus

Most NLP tools for text processing need collection of texts, which are used to create statistical models of language. This collection is called corpus. A corpus is a collection of machine-readable texts that have been produced in a natural communicative setting. They have been sampled to be representative and balanced with respect to particular factors; for example, by genre - newspaper articles, literary fiction, spoken speech, blogs and diaries, and legal documents.[18]

Public corpora Almost every country has its national corpus, which is created by tagging documents in national language. These corpora are created by linguistic institutes and contains billions of words, from various sources. Beside language corpora exists corporas with specialized content like text produced by non-native speaker. For Czech corpora containing 2.2 billions words, see *www.korpus.cz*.

Corpus allows us statistical test on representative sample of language and help us create statistical models of language. These statistical models

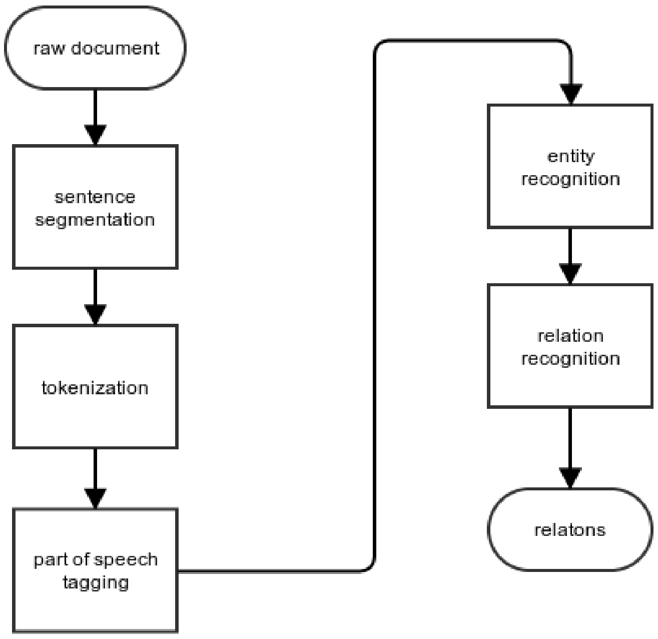


Figure 1.1: Processing document

extracted from such corpus can provide useful information about similar documents or even its amount of similarity to the corpus. One of important application is language identification.

1.2 Language identification

For languages with a unique alphabet not used by any other languages, such as Greek or Hebrew, language identification is determined by character set identification.[10] For more difficult cases, such as European languages that use exactly the same character set but with different frequencies final identification can be performed by training models of byte/character distributions in each of the languages. [10] To improve success of such model we can enhance it with prediction of next letter or word. We formalize this idea of word prediction with probabilistic models called N-gram models, which predicts the next word from the previous N - 1 words.[12] This n-gram model creates Hidden Markov Model of N - 1 order. Which can be used for prediction sequence of letter in language. By comparing n-gram model generated from training corpus with model generated from unknown document, we can estimate probability of document language or even a topic.

1.3 Tokenization

Tasks in NLP often require breaking input text into meaningful substrings - Tokenization. One type of these substrings is a word. This process can take many forms, depending on the language being analyzed. For English, a straightforward and effective tokenization strategy is to use white space and punctuation as token delimiters. This strategy is simple to implement, but there are some instances where it may not match the desired behavior. [13]

Another important substring is sentence, which shares similar problems with word tokens in their identification. For English text, it is almost as easy as finding every occurrence of punctuation like "."; "?"; or "!" in the text. However, some periods occur as part of abbreviations or acronyms.[13]

While punctuation characters are usually treated as separate tokens, there are many cases when they should be "attached" to another token. The specific cases vary from one language to another, and the specific treatment of the punctuation characters needs to be enumerated within the tokenizer for each language.[10]

Some of these problems can be solved by adding language specific rules into tokenizer. Most of tokenization rules can be expressed using standard and widely implemented language - Regular Expression (RegExp).

1.4 Regular expression

The regular expression is used for specifying text strings in all sorts of text processing and information extraction applications.[12]

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus, they can specify search strings as well as define a language formal way. [12]

Sample of RegExp used for matching number is displayed in Listing 1.1. This RegExp matches beside simple integers English format of numbers (1/2 or 1,000.99). Diagram describing matching process (Figure 1.2) shows three matching groups. These groups will be returned by RegExp engine when RegExp is applied on a string and matched. Group #1 is number itself, Group #2 is explicit sign of the number and Group #3 is its terminator. First group is simple, $([-+]?)$ states for optional plus or minus sign is a group. Similar matching is used in Group #3 ($?(?: st|nd|rd|th)?)$ but this time we can't use bracket for defining list of possible patterns, instead we use vertical line symbol (pipe). By grouping all possible suffixes into unnumbered group (defined by ?: symbols at the beginning of the group) we can make this group optional using question mark, which states for one or zero occurrences. Final, middle part matches number digits. Its first part matches 1 and more occurrences of digits from 0 to 9 $([0-9])$. Further it matches either number in formats (1,000.00)

```
(
  ([-+]?) // Group #2
  [0-9]+ // integer matching
  (?:(?:
      (?:[ ,] ?[0-9]{3})* //thousands
      (?:[.][0-9]+)? // floats
    )|(?:
      [/][0-9]+) // fraction
    ?)
  )
(\ ?(?:st|nd|rd|th)?) // Group #3
```

Listing 1.1: English number matching regex

```
<(script|style).+?<\/\1>
```

Listing 1.2: RegExp matching style and script tags

or fractions or it skippes directly to Group #3 matching in case of small or undivided integer number.

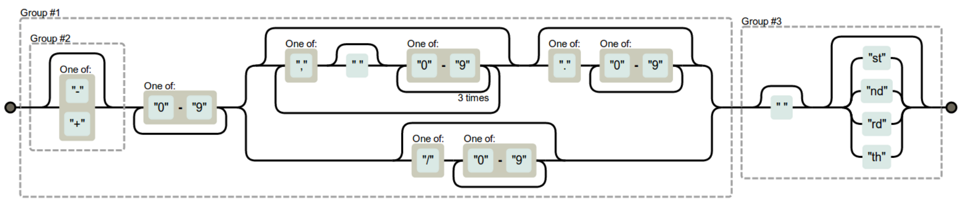


Figure 1.2: Diagram for number matching regex

Beside tokenization, regular expression has important place in document cleaning. Many texts contain certain classes of character sequences which should be filtered out before actual tokenization; these include existing markup and headers (including HyperText Markup Language (HTML) markup), extra whitespace, and extraneous control characters. [10] One example of HTML cleaning RegExp is displayed in Listing 1.2, which matches all script or style tags in page.

1.5 Stem and Lemma

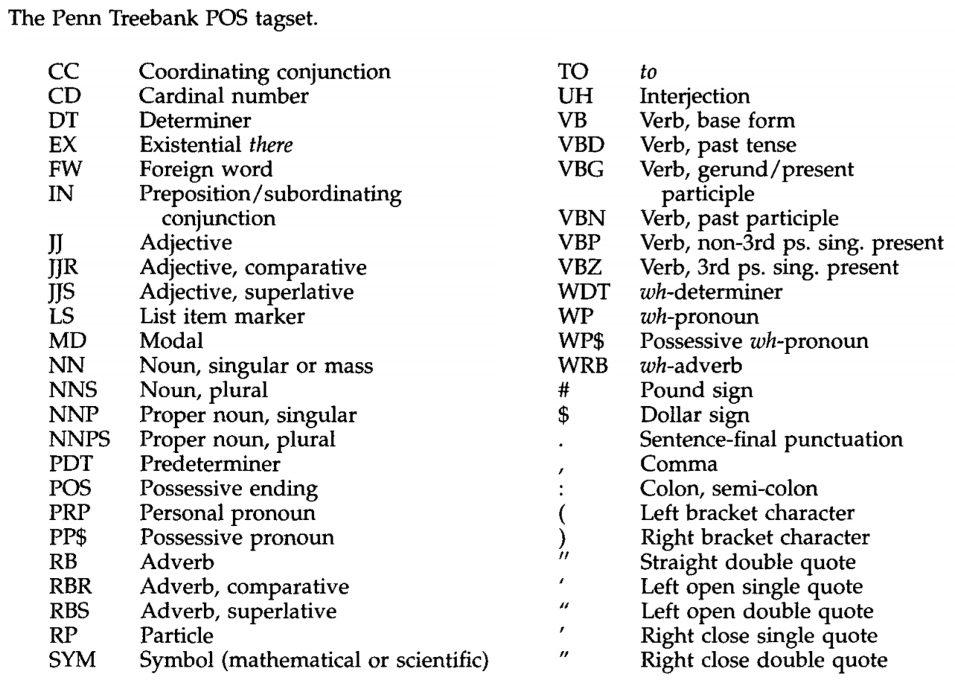
A lemma is morphology root of a word; it's the form that can be found in dictionaries and represents all its forms - lexeme (Figure 5.4). Lexemes are created from lemma by language rules, which are complex, sophisticated and with lot of exceptions. This makes automatic lemmatization - the process of converting any lex to its lemma, a complicated problem. Lemmatization is required in case of analyzing languages with rich language resources such as Czech, Dutch or German.

A stem is an artificial root of a word created by removing suffixes and prefixes. In contrast with lemmatization, creating stem from the word - stemmatization, doesn't require extensive processing resources by using simple rules. The stemmatization is suitable for cases where deeper semantic meaning is not crucial or for performance focused application. It is widely used for a fulltext search.

Both of these methods are considered as content normalization methods; mapping different forms of tokens describing same concept to single form.

1.6 POS tagger

POS tagging is critical step in many NLP applications, since it is important to know what category a word is assigned to in order to perform subsequent analysis on it.[18] POS tagger classify basic lexical category of words, like verb, noun or adjective and even its subcategories. There are a number of part-of-speech coding schemes, based on tagsets of between 40 and 200 tags.[12]


The Penn Treebank POS tagset.

CC	Coordinating conjunction	TO	*to*
CD	Cardinal number	UH	Interjection
DT	Determiner	VB	Verb, base form
EX	Existential *there*	VBD	Verb, past tense
FW	Foreign word	VBG	Verb, gerund/present participle
IN	Preposition/subordinating conjunction	VBN	Verb, past participle
JJ	Adjective	VBP	Verb, non-3rd ps. sing. present
JJR	Adjective, comparative	VBZ	Verb, 3rd ps. sing. present
JJS	Adjective, superlative	WDT	*wh*-determiner
LS	List item marker	WP	*wh*-pronoun
MD	Modal	WP$	Possessive *wh*-pronoun
NN	Noun, singular or mass	WRB	*wh*-adverb
NNS	Noun, plural	#	Pound sign
NNP	Proper noun, singular	$	Dollar sign
NNPS	Proper noun, plural	.	Sentence-final punctuation
PDT	Predeterminer	,	Comma
POS	Possessive ending	:	Colon, semi-colon
PRP	Personal pronoun	(	Left bracket character
PP$	Possessive pronoun	)	Right bracket character
RB	Adverb	"	Straight double quote
RBR	Adverb, comparative	‘	Left open single quote
RBS	Adverb, superlative	“	Left open double quote
RP	Particle	’	Right close single quote
SYM	Symbol (mathematical or scientific)	”	Right close double quote

Figure 1.3: The University of Pennsylvania (Penn) Treebank Tag-set

CHAPTER 2

Goals

The goal of my work was to create suitable web browser application capable of extraction recipe information, like ingredients and their amount and preparation steps, from any website in supported language without any server side text processing. This goal can be represented by following requirements, the application should:

- load and process web page within a second
- not load more than few hundred kB of external resources into users browser
- identify ingredients, amount and preparation of a single recipe
- be capable of identification of a supported language
- process web page in browser, server should act as storage of application code and dictionaries

Text categorization and extraction are common task in NLP, but existing NLP stacks are not suitable for usage in browser, they would require additional plugins to be installed in clients browser. Implementation of lightweight and on browser focused library for NLP is necessary, which defines more requirements for my work:

- it should be possible to use implemented library in other, similar NLP application
- library should be extensible

CHAPTER 3

Problem analysis

Recipe extraction can be modeled as information retrieval task, which can be solved by NLP methods. Most of these methods can be generalized and implemented as generic tools, that are not limited to recipe extraction. Standard approach in information retrieval is identifying important parts of document, use database of word relationships to build some basic knowledge about document.

3.1 Application structure

To avoid server side processing, these NLP methods have to be implemented in a programming language interpretable by a browsers - JavaScript. Since this problem requires document in various languages and because of language dependency of NLP methods, language identification is required. By identifying the language, loading of a language dependent parts of the library can be delayed and only those that are required will be loaded. In order to conditionally load parts of the library, the library has to be divided into modules.

3.2 Document

Some wrapper of the document text should be implemented in order to provide unified access to its content and metadata. HTML documents have to be parsed and only meaningful content stored as a plain text. This way we can process HTML and plain text documents using same tools. Because of possible live changes of Document Object Model (DOM), restricted access to DOM in some cases (e.g. userscripts, iframes), an HTML parser is required. One possible approach is creating DOM in a memory and extracting content from there. With this approach there is a problem with separating text content and tags, while keeping track of relationship between these two. Other approach is creating custom HTML parser without building DOM.

3.3 Annotations

Document's metadata gained directly from input document or extracted using NLP tools, has to be stored and have access to them provided. This metadata can be implemented as a document annotations. The annotation has it's starting and the ending point within the document text, each annotation has a type which defines its relationship to the document text and allows easier filtering. Finally, the annotation implementation needs to provide some key-value storage for storing an additional data with various structure. Annotations are usually stored as a set, which provides filtering methods. These filtering methods should return view of a filtered annotation set implementing same methods as an original one, which allows additional and effective filtering.

3.4 NLP toolkit

Implemented NLP tools should use interface provided by document and annotations classes described above and should not access document in other way. Using this approach results of these tools will be available as new annotations and accessible by other tools.

3.5 Tokenizer

Document content needs to be divided into classifiable parts like sentences or words. Tokenizer is a basic tool from NLP toolkit that has to be implemented. It might be language specific and, thus it needs to be implemented in extensible manner.

3.6 Gazetteer

Gazetteer is a word dictionary that provides various types of information about contained words. Most common types of information are POS category (e.g. noun, verb) and topic information (e.g. cooking, furniture). This type of dictionary is necessary for understanding text at least in terms of its categorization. Thus, a gazetteer of cooking related words is required. Since I have access to database of recipes with ingredients and preparation in separate columns I can build gazetteer using this data. Data in this database doesn't contain POS tagging, so I have to use some external tool to tag them.

3.7 Stemmer

In order to minimize number of gazetteer I need to map different forms of same word into a single form which would be stored in gazetteer. Usual approach is using Stemmer. Some stemmers for JavaScript already exists and can be used. I just need to implement unified interface to use stemmers from different sources in the same way and in case of missing Asynchronous Module Definition (AMD) interface, wrap stemmer code into AMD module.

3.8 Extraction

Recipe extraction will require identifying language of document, and loading appropriate tools. After application of these generic tools onto a document, a domain specific tools has to be executed. These tools will extend knowledge about document entities and identify important ones. Extracted information will be filtered in order to identify the content of the most important recipe on the page.

CHAPTER 4

Implementation

4.1 Library

The library is developed in JavaScript, which is present in all nowadays browsers, both mobile and desktop, doesn't need to be explicitly allowed by user like Java and provides stable cross-platform and cross-browser Application Programming Interface (API) unlike C/C++ using Native Client (NaCl). JavaScript is single-threaded event driven functional programming language with prototypical inheritance. All JavaScript code on a single page is executed within one thread and during it's computing the web page is not responding to user interaction. This forces developers to write application that have very short execution time or splitting runtime into multiple short time frames, in order to avoid long delays of user interaction events.

4.1.1 Compatibility

Proposed library was implemented using latest ECMAScript 6 (ES6) syntax, which is not at the time of writing this thesis natively supported by any browser. But thanks to thes nature of javascript, there are transpilers, source-to-source compilers, which can compile an ES6 code into an ECMAScript 5.5 code; most feature complete are traceur developed by Google and babeljs. These transpilers solves only syntax problems, for accessing new API features like Promises or TextDecode we need to load polyfills. Polyfilling is technique where missing standard native browser feature is emulated by the loaded javascript code. Babeljs provides library polyfilling most of ECMAScript 6 features. Thanks to transpilers and polyfills, we can write a future javascript code today without losing the support of older browsers. Hopefully in the near future, the code we write today will work natively.

4.1.2 Modular system

Module system is a must-have for applications with multiple classes and internal dependencies. Days when we wrote javascript code to the end of the HTML document are long gone; nowadays complex application requires complex structure. Modular system allows developer creating large libraries and users cherrypick just the parts that they actually need.

ECMAScript 6 provides module system which works the same on the server and in the browser. Each module imports its dependencies in header and then publicates its API using export keyword. This library was originally modularized using AMD which still is a strong module system but since ES6 syntax is much cleaner, can be compiled into AMD and CommonJS module format and it's the future of javascript I've migrated library to ES6 modules. This allows using library as compiled library with single access point in global scope (like jquery) or loading sources as dependency into users project and importing only part that user need, automatically with all its dependencies. The later usually leads to a smaller application code and cleaner global scope.

4.1.3 Used tools

Building and testing tasks are covered by gulp.js. This tool is JavaScript equivalent of MAKE that is commonly used in C projects. It provides easy to use command line interface and automate building and testing process, which can be even automatically executed after each file save. These build tools are especially useful for complex building processes, which usually consists of code checking, linting, testing, transpiling, sourcemap generating, bundling, minification and compression.

Testing Tests are written using mocha.js and should.js assert framework, which allows fast, multi-browser testing in Behavior Driven Development (BDD) style. Since tests are just a web page, they require browser to start the test framework, that runs tests. Using test framework that allows testing in any real browser is crucial in current state of Internet environment, where web browser implements new feature on a daily basis and multiple browsers in multiple versions exist simultaneously. I've chosen phantom.js as browser that runs tests automatically in the background during development because of its low resource impact, which is achieved by its headlessness, and fast feedback which is provided directly into a terminal or a test console of most Integrated Development Environment (IDE). To make sure that library is fully tested, a cover test library called blanket.js is included within test suit. This library provides overview of which lines of code were executed during testing.

Dependency management Dependencies are defined following Node package manager (NPM) and bower standards. Bower is a tool for dependency management, developed by Twitter and is widely used in web developer's community. NPM is node.js packaging manager which is primarily used for managing server side and desktop JavaScript libraries and application. As NPM dependencies I'm listing desktop tools for testing and building, like grunt, it's tasks and bower. Listed bower dependencies are third party libraries that are directly used by developed library and developing tools, like mocha and blanket. Bower also defines this library as possible dependency, which makes it easy to find and include in other projects.

Lodash is a javascript library implementing many useful generic functions. Most known functions are the collection iterators and transformations: map, filter, reduce, any. Some of these functions are built into the browser, despite that, the lodash implementation takes their performance to maximum and it is significantly faster then native version in most browsers. Since I'm doing a lot of array operation, I'm using lodash for its performance boost. Using its ES6 module version, I was able to cherry-pick only a few functions I needed (map and filter) and define it as a dependency in files which need them.

4.1.4 Summary

In this chapter, I've explained why I used JavaScript and with which versions is implemented library compatible. Described nowadays best practices in JavaScript and how are these practices applied. Described dependency management in the scope of project, application and development tools. Library source code, documentation and showcase of few features is available on the GitHub: ficik.github.io/nlpjs. Only dependency of this library is lodash.

CHAPTER 5

NLP Stack

The library I've implemented for generic NLP tasks is called nlp.js and is available on github. Library is split into three parts, core, extras and helpers (Figure 5.1). Each of this part contains classes which are defined using AMD API, which way defines dependencies on each other.

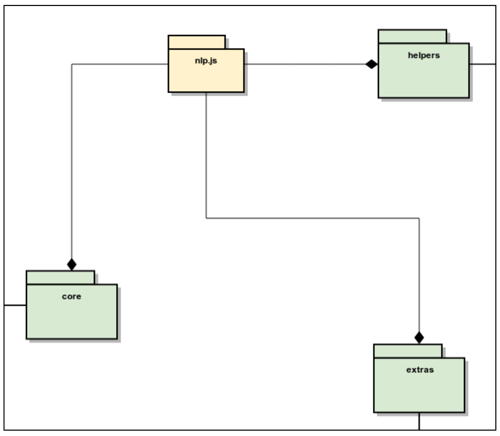


Figure 5.1: Package hiearchy

- Core module defines documents, corpora and annotations and is purely focused on loading documents, storing its data and metadata and providing access to it.
- Extras module defines NLP tools which accesses and modifies documents.
- Helpers module defines functions and classes used by both, core and extras.

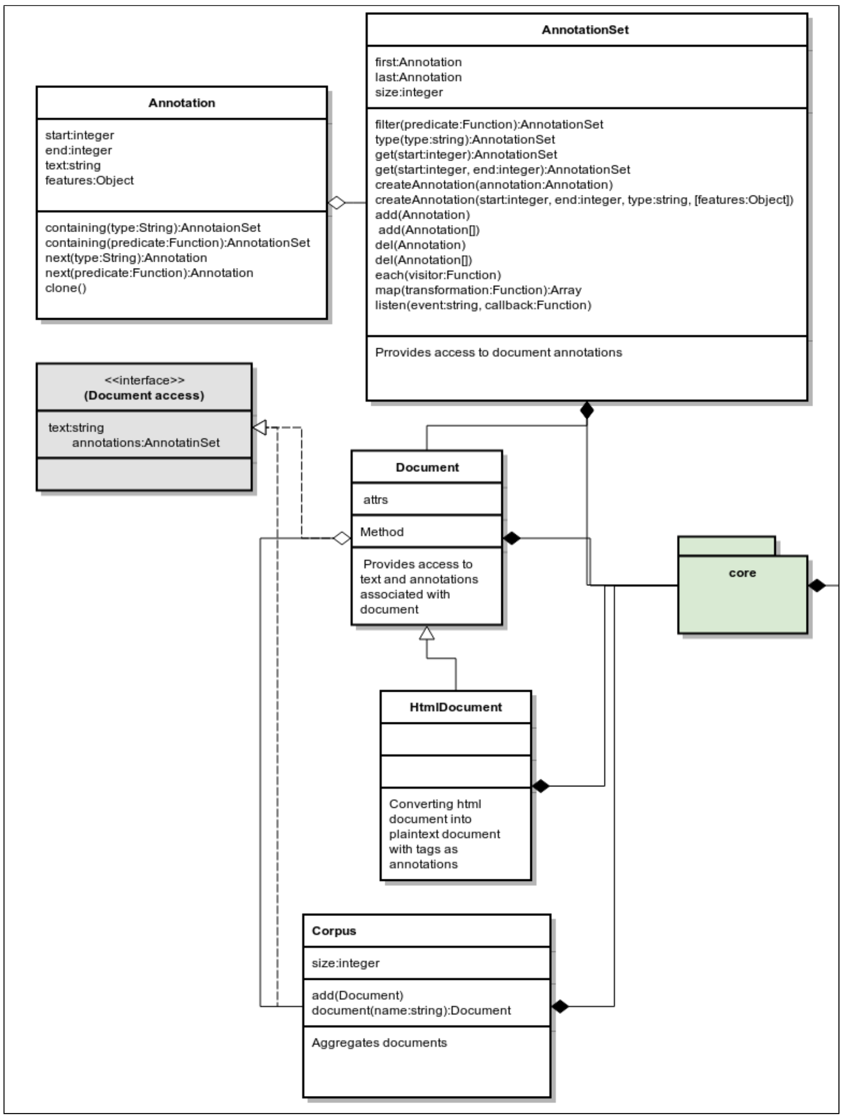

AnnotationSet
first:Annotation
last:Annotation
size:integer
filter(predicate:Function):AnnotationSet
type(type:string):AnnotationSet
get(start:integer):AnnotationSet
get(start:integer, end:integer):AnnotationSet
createAnnotation(annotation:Annotation)
createAnnotation(start:integer, end:integer, type:string, [features:Object])
add(Annotation)
add(Annotation[])
del(Annotation)
del(Annotation[])
each(visitor:Function)
map(transformation:Function):Array
listen(event:string, callback:Function)
Prrovides access to document annotations
Annotation
start:integer
end:integer
text:string
features:Object
containing(type:String):AnnotaionSet
containing(predicate:Function):AnnotationSet
next(type:String):Annotation
next(predicate:Function):Annotation
clone()
<<interface>>
(Document access)
text:string
annotations:AnnotatinSet
Document
attrs
Method
Provides access to text and annotations associated with document
core
HtmlDocument
Converting html document into plaintext document with tags as annotations
Corpus
size:integer
add(Document)
document(name:string):Document
Aggregates documents

Figure 5.2: UML diagram of core package

5.1 Documents and Annotations

Document and annotations are represented by the Document and AnnotationSet classes respectively and primary structures of the core module (displayed in Figure 5.2) and the library. This library follows pattern commonly used in other NLP tools and wraps the processed document into an object separating text of the document and the metadata. The annotation set is created by the document and a user should never need to create a new instance of the annotation using its constructor. Same rule goes with annotation set. Since annotations can be created by providing necessary informations about new annotation to the annotation set in a multiple forms, there should be no need to create new instances of annotations manually. s

5.1.1 Annotations

Annotations are stored in annotation set which is bound with document and provides methods for storing, removing and querying annotations. Annotations and annotation set are the most important part of this library. Most of the higher level methods in NLP work with annotations, instead of text and uses results of lower level methods (e.g. tokenizer, stemmer), which are stored as annotations and stores its results also as annotations.

Implemented annotation set acts as a collection that provides standard filter, map and each method. All of these methods iterates over annotations in left-to-right, top-to-bottom order, which provides a stable and easy to understand interface for processing words in sentence, lines or sentences in document e.i. Annotation set is most frequently accessed object in this library, thus any further performance optimizations should start here. So far I was satisfied with its performance when I was working with thousands of annotations but it can be pushed way further by switching internal data array for some sorted structure like a heap.

Some common filtering methods were added to simplify usage of the library. There are methods for filtering by annotation type, first annotation after specified position and annotations overlapping selection. The latest is very important for queries like words of sentence or sentence containing some word. These methods are just wrappers above more powerful filter method, which can create filtered annotations set using any predicate provided as a function.

5.1.2 Annotations querying

When implementing annotation set, I was thinking of developing regex-like engine to query annotations, but syntax for such engine was either too complicated to use or made engine too weak. This is because possible overlapping annotations, user defined feature set and types. I decided not to implement regex-like querying and instead I've implemented querying methods directly into annotations, which provides reasonable syntax sugar, and is easier to debug, since query is just a JavaScript call and provides all the power of the annotation set. Following example 5.1 presents example of rule, that can be written using this querying system. This example creates new annotation of type amount (line 4) by finding number (line 1) followed (line 2) by unit (line 3).

```
annotationSet.type('number').each(function(number){
    var nextWord = number.next('word');
    if (nextWord.features.major == 'unit')
        annotationSet.add(number.start,
                              nextWord.end, 'amount');
});
```

Listing 5.1: Annotation se querying example

5.1.3 Document

Library provides Document class, that represents plain text document and is responsible for loading, parsing and storing input document. It also provides access to documents annotations in the form of the annotation set described in previous paragraphs. Beside simple plain text document I've implemented HTML document which extends plain text document by overriding setter of its text property. Input document has to be converted into plain text to be accessible by tools like Tokenizer, thus HTML document preprocess HTML content and extracts all HTML tags which are further removed from document text and saved appropriately as annotations into annotation set. This is done by custom HTML preprocessor that works even on malformed documents.

5.2 HTML Document parsing

HTML documents contain a lot of additional pieces of information beside document text, which are used as functionality of the website (javascript), esthetic purposes (Cascading Style Sheets (CSS)) or developer's tags (comments). These bits of information are usually not relevant for text mining purposes and can be removed. Most of HTML tags are used just for the aesthetic purposes but some of them can serve as a line or block delimiters. Since blank characters like new line, tabs or multiple spaces are ignored in HTML and all of them are rendered as a single space, they become useless for text extraction and can be replaced with a single space. Instead of these blank characters I'm using some HTML tags. Most relevant tags seems to be paragraph (p), list item (li) and table row (tr). All of these tags renders inner text in most cases (on large enough screen) as a single line and don't share same line with other tags in this group. Other relevant tag is a break line (br) which can break content of mentioned tags into two lines. With missing new line delimiter, in HTML detection of these lines is crucial to correct sentence and non-sentence parts tagging.

Corpus

The final part of the core module is a corpus, which provides way of merging multiple documents of different type or even another corpora into one object that acts as a document. Corpus implements same interface as the document and thus all tools that work with the document work with the corpus as well. The main motivation for using the corpus is executing a statistical analysis on a large amount of documents from multiple sources.

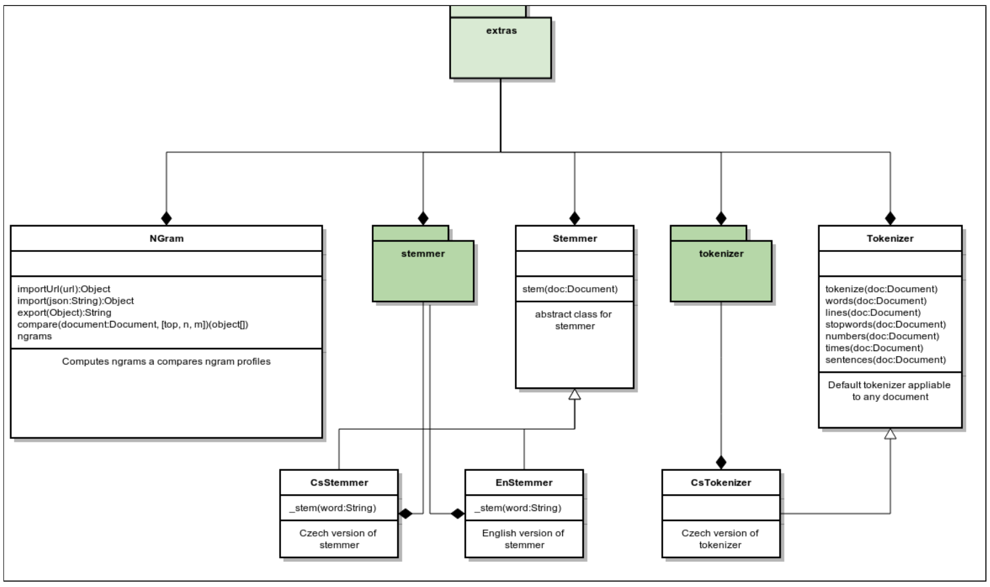


Figure 5.3: UML diagram of extras package

5.3 Tokenizer

Tokenizer is a part of the tokenizer module. It operates on any Document or Corpus objects and saves its results into their annotation sets. Tokenizer in root of extras module acts as a generic tokenizer for European languages. In order to create language specific version, this generic tokenizer should by extended and some of its methods or properties overwritten.

Tokenization is a process of extracting meaningful parts of text (tokens) from input string. Most common token used in text-mining is a word, which can be defined, for purpose of text-mining, as a string containing alphabetical symbols which are separated from other words by punctuation of white characters. In presented library, tokenizer class contains methods which take document text as primary source and generates annotations of appropriate type. Tokenization methods for word, number, time, sentence and line were implemented.

Word tokenizer generates tokens of type described above, but also identifies and tags stop word. Stop words are language specific words that usually have no meaning of their own (e.g. the, that, on, at), this process is called Stopping. Stopping is commonly included feature in nearly every text mining software package. The removal of stopwords is possible without loss of information because of the large majority of text mining tasks and algorithms, these words have little impact on the final results of the algorithm.[practical] Exception is a phrase detection and thus, implemented tokenizer doesn't automatically remove stop words, but only marks them.

Number tokenizer generates tokens that represent numbers, integers, floats or fraction.

Time tokenizer generates tokens containing either date or time or even duration. Type of the time token is also saved.

Line tokenizer is html specific. This tokenizer generates annotations for visual lines of text from some HTML block tags.

Sentence tokenizer generates sentences, which are defined in current implementation as a string that starts with capital letter and ends with interpunction. It also contains list of common abbreviations ending with a dot and uses this list to prevent incorrect sentence termination after abbreviations. In case of html document additional sentence correction is added, which takes advantages of line tokens. This fixes sentence by moving start of the first sentence of each line to the start of the line and moving start of the following sentences of that line to the end of the previous sentence. This fixing method prevents incorrect identification of the start of the sentence in case of an appearance of a capital letter within the sentence.

5.3.1 Czech tokenizer

Czech version of tokenizer is derived from the generic one and extends capability of number and date methods. It also provides Czech stop words dictionary list of abbreviations to improve sentence matching.

5.4 Stemmer

In computer linguistic exists a need for mapping words to their base form. Stemming is the process of normalizing related word tokens into a single form. Typically, the stemming process includes the identification and removal of prefixes, suffixes, and inappropriate pluralizations.[practical] Main motivation for this is to decrease the size of the necessary dictionary and simplifying rules. As humans we have gained ability to create base form of any word from our native language (Figure 5.4). In linguistics this form is called lemma, unfortunately learning computer this ability of ours, lemmatization, is very complex problem, which require extensive dictionaries and complex rules. In order to bypass the need for large dictionary, we can use stemmatization, which is a process of removing common suffixes and prefixes in order to get root of the word. This root doesn't have to match morphological root but usually should give same root for similar word.

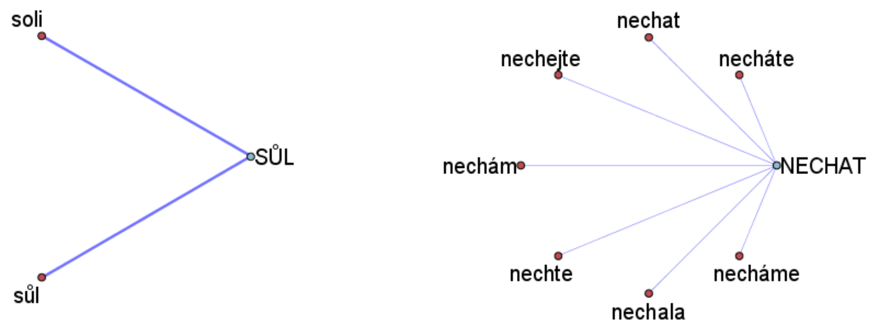


Figure 5.4: Word lemmatization

The stemmer class from the extras module represent a loader and abstract class which should be extended by a real stemmer. Existing JavaScript implementation of stemmer for Czech and English are ports of stemmers in snowball language. These implementations are not defined as modules and thus, they have to be wrapped and they become part of this library. Stemmer interface requires implementation of a single method, which returns stem of provided word. Interaction with a document like iterating over word tokens and creating and adding annotations is handled by abstract class.

5.5 Hunspell based stemmer

An addition to snowball ports is my implemention of stemmer, which is based on Hunspell spellchecker.

5.5.1 Hunspell

Hunspell is a tool for spellchecking which uses dictionary and rules. Each word in the dictionary is associated with a group of rules that can be applied

```
SFX P   y   iness   [^aeiou]y
SFX P   0   ness    [aeiou]y
SFX P   0   ness    [^y]
```

Listing 5.2: Sample from English hunspell aff file

on to the word to create new valid form of that word.

Rules are in format displayed in Listing 5.2. Each line can contain up to one rule, which is tab separated list:

1. Rule generates word suffix (SFX) or prefix (PFX)
2. Class of the rule
3. Suffix/prefix to remove
4. Suffix/prefix to add
5. Precondition

Class of the rule is character that is used in dictionary to map words to its class in ruleset. Each word can have more classes, but not all rules of its class can be applied to the word.

Substitution Rule aguments 3 and 4 are used to generate new word from dictionary form. First argument defines whether this substitution is applied to prefix or suffix.

Precodition Dictionary word affix is tested against regex which is 5th argument of the rule.

5.5.2 Rule based stemmer

Hunspell rules can be applied in reverse to create dictionary word from its expanded form. If we consider dictionary words as the base form of the word, which is how mosts hunspell dictionaries are created, we can create stemmer by applying inverted version of rules:

1. Select next rule
2. Create stem from word by removing rules addition parameter from the word
3. Add rules removal parameter to the stem
4. Test stem against precodition

5. Accept stem if precontion holds, else reset word and repeat

This rule based stemmer works well on languages that don't have many colliding rules like english. Unfortunatly it doesn't work well on languages with complex affix system like czech. Stemmer quality on these languages can be improved by reordering rules by strength of the precondition and length of affix that has to be removed.

5.5.3 Combining with dictionary

Previous rule based stemmer can be combined with dictionary provided along with the hunspell rules. By loading dictionary to the stemmer we can create stemmer which either is lemmatizer or something very similar; this depends on ruleset quality. The dictionary allows us to add two additional checks to the previous stemmer. To accept stem, it has to exits in the dictionary and the rule has to be applicable to the dictionary word. This additional check eliminates problems with wrongly applied rules.

Unfortunately now we have a problem with the size of the dictionary, the czech dictionary is 4MB file containing more than 300 000 words. To reduce its size we can test it against dataset of words we will be stemming. And by counting each word and rule that has been used, we can eliminate large part of the dictionary, leaving us with just few thousand even if we accept all words that appeared at lease once.

5.5.4 Summary

Implemented stemmer generator is capable of creating a rule based stemmer from a hunspell affix file, which is suffient for languages with a simple affix system. It is also capable of loading a hunspell dictionary file. It can compute model usage and using this statistic significantly reduce size of model or reorder rules.

5.6 N-Gram

N-gram is a sequence of n items from input. These items can be for example letters or words. In my implementation I'm using n-grams as a sequence of letters, while replacing punctuation with spaces, removing non-alphabetical symbols and trimming repeated white characters. This way I'm generating list of n-grams for desired values of n. By computing frequencies of n-grams we create Markov model of (n-1) order. Which can be used for text categorization, authorship detection or language recognition.

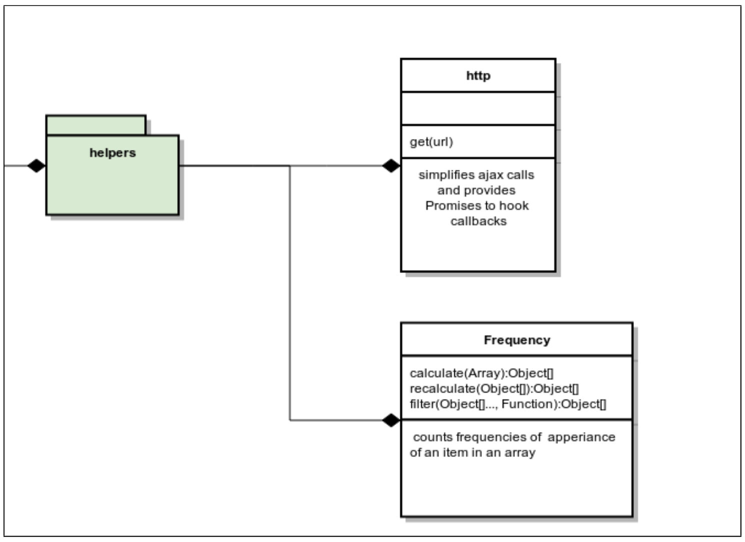

helpers
http
get(url)
simplifies ajax calls and provides Promises to hook callbacks
Frequency
calculate(Array):Object[]
recalculate(Object[]):Object[]
filter(Object[]..., Function):Object[]
counts frequencies of apperiance of an item in an array

Figure 5.5: UML diagram of helpers package

5.7 Bayesian classifier

Bayesian classifier is machine learning algorithm base on Bayes' theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

I've implemented naïve Bayes classifier with smoothing constant. This classifier can determine most probable class for given set of features. But first, the classifier must be trained. During training process, classifier is fed with classified features and number of occurrences of each feature in each class forms the classification model. This model can be stored as JavasSript Object Notation (JSON) and later reloaded.

During classification, classifier uses trained model and Bayes' theorem to calculate probability of feature set being given class. For solving following equation we need to know the probability of a feature set while it's given class, the probability of the class and the probability the feature set itself.

$$P(class|featureSet) = \frac{P(featureSet|class)P(class)}{P(featureSet)}$$

Probability of the feature set while it's given class can be estimated by using data from a training set. It's a number of occurrences of the features set, while classified as the class. But since occurrences of each feature is saves separately we need to calculate probability of the whole set. In order to do this, we need to assume that occurrences of features are independent.

With this assumption, we can multiply probabilities of each feature to form feature set probability.

Smoothing constant Calculating feature set probability by multiplying probability of each feature can very easily yield zero probability; it's caused by adding probability of feature that wasn't in the training dataset. To overcome this problem, we can assume that each feature can appear in any class. This can be done by adding smoothing constant to all feature occurrences, even those that wasn't seen before.

Probability of class can be estimated by counting trained documents of given class. It can be also set manually from other external observation.

Probability of feature set can be also estimated from number of occurrences of features in any class, but since the classifier selects the most probable class of the same feature set and it doesn't care about its actual probability, we can skip this computation and just use numerator of the probability.

5.7.1 Summary

I've implemented naïve Bayes classifier which can be used on both annotations and text, or even their combination. It can be used to determinate most probable classification of given feature set.

5.8 Other support classes

5.8.1 Frequency helper

Frequency is a simple object with methods for counting occurrences of some item or items key within an array and provides this information as an array. It also provides methods for filtering array by comparing its frequency profile with other arrays. These methods are useful for various applications of frequency analysis. It was for example used for generating dictionary from recipe database and filtering it using background corpus.

5.9 HTTP helper

Similar HTTP helper is present in most frameworks and in order to establish common access to resources on server like dictionaries and statistical model. This method wraps Asynchronous JavaScript and XML (AJAX) call and returns JavaScript Promise object that is later fulfilled with result data from the server. This helper is currently subject to change and will be probably replaced with external http module.

5.10 Distances

5.10.1 Hamming distance

Hamming distance is one of the most known string metrics, thus it's also part of the library. This distance is defined for strings of the same length as number positions with different characters. It's not defined for strings of a different length.

5.10.2 Levenshtein distance

Levenshtein distance also known as edit distance is string metric defined as minimum number of insertions, deletion and substitution required to transform one string to the other. Library contains weighted version of this distance, which allows set cost of each operation. Static method of Levenshtein object uses standard version where cost of all operation - insertion, deletion and substitution is set to 1. Instantiation allows to set these costs.

5.10.3 Gazetteer

Gazetteer is a list of words that belongs to some category, which are used for adding meaning to a words in a corpus; for example list of countries, cities, important names. Since most of these lists are formated as a text file with one term on each line, I've decided not to implement a parser (it's just spliting a string on the newlines).

Instead, I've implemented a class that applies a gazetteer provided as a list of words onto the corpus and enhancing the feature set of annotations in it by provided features. Word matching is done using both text and stem if available. To allow word deviations like misspelling or different affixes, I've added support for string metrics. Default metric is Levenshtein distance which can be overridden by providing it to the constructor; any object with distance method can be used.

5.10.4 SPARQL Wrapper

Simple Protocol and RDF Query Language (SPARQL) is a Resource Description Framework (RDF) query language, which querying RDF databases like dbpedia. Using this language and public SPARQL endpoints, the browser application can request additional information about processed content. Which can be very powerful tool for enhancing user experience.

Implemented wrapper provides simple interface to define SPARQL endpoint and query it using AJAX or JSON with padding (JSONP) (in case of missing Cross-origin resource sharing (CORS) headers). It also provides simple query wrapper, which at its current form simplifies prefix management. This allows creating enhanced endpoint specific versions of endpoint wrapper.

One of these is implemented dbpedia wrapper, which automatically sets connection parameters for any dbpedia language mutation and adds some of the most common prefixes into query.

5.11 Summary

In this chapter I've described implemented library that can be used in any corpus linguistic application in a browser or on a server. I've described basic functionality of core of this library (document and annotations), presented implemented annotation querying system, which was chosen instead of custom regex-like solution. I've also explained problems and possible solutions of HTML document in text-mining.

I've also described implemented tools that recipe extraction was build on - Tokenizer, Stemmer and N-Grams. And explained how these tools access document and how their results are stored.

CHAPTER 6

Recipe Parser

6.1 Language identification

Language identification is another common task in NLP. There are many approaches to this task, they often use some diacritics and special character sequences detection.

N-Gram document profile Most approaches use low order n-gram models. One possible approach is creating n-gram frequency profile from the training set. Frequency profile is a list of n-grams sorted in descending order by their frequency. This frequency profile is generated for each language we want to identify. In order to classify document, frequency profile of the document is created and compared to profiles generated from the training set. Comparison is performed by computing distance of each n-gram from classified document to the same n-gram in language profiles.

Profile distance Distance is defined as a difference in position (rank) in profiles. Distance of profiles is a sum of distances of all n-grams from input document. Language is identified by selecting language profile with the lowest distance. Size of language profiles can decreased by taking only top k n-grams. Dropping least frequent n-grams from profile have just a little impact on profile accuracy because of their low frequency. Speed of decrease of occurrence frequencies of n-grams according to their rank can be observed on Figure 6.1; for easier distinguish between language profiles is x axis in logarithmic scale. From total of 74,000, 108,000 and 98,000 n-grams extracted from English, Czech and Slovak corpora respectively, the frequency of first 300 1-5-grams occurred in 35-40%. Fast decrease of frequency can be explain by Zipf's law, which states for rank n and P_n probability of occurrence of the word at rank n and language specific a very close to 1:

```
{
  "cs" : "_|e|a|o|n|t|r|k|p|i|l|v|s|m|d|c|u|e_|y",
  "en" : "_|e|a|t|i|s|o|r|n|l|c|d|h|e_|p|s_|u|m|g",
  "sk" : "_|e|a|o|n|i|r|k|m|t|s|v|p|l|d|c|y|a_|u"
}
```

Listing 6.1: Language profile JSON

$$P_n \sim \frac{1}{n^a}$$

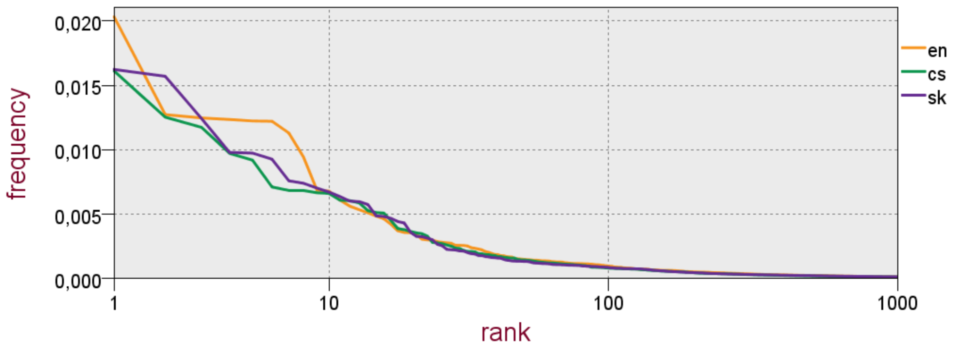


Figure 6.1: Frequencies of n-grams according to their rank

Generating language model According to experiment performed on Usenet posting by Cavnar using only 300 top n-grams (n = 1-5) can achieve 99.8% accuracy. [7]. Inspiring by cavnar's experiment I've created language profiles for czech, english and slovak using 300 most frequent n-grams (n = 1 - 5). These profiles are saved as a JSON file (example of these saved profiles containing 19 most common n-grams is shown in Listing 6.1) which require only about 1kB per language and even less if the profile is compressed with gunzip. Language profile was generated from 20 web pages for each language.

Identification process Before identifying document language a JSON containing language profiles is loaded and transformed into a map with n-gram as a key and its rank as a value (example available as Listing 6.2). This transformation speeds up lookup times, which can be assumed as constant (O(1)). And language profile for document is generated and trimmed to top 300 n-grams. For each n-gram in this profile distance is calculated from same

```
{
  cs : {
    "_" : 0,
    "e" : 1,
    "a" : 2,
    ...
  },
  en : {
    "_" : 0,
    "e" : 1,
    "a" : 2,
    ...
  sk : ...
}
```

Listing 6.2: Transformed language profile

the n-gram in each trained profiles, in case of missing n-gram in language profile, distance is considered maximal, which is equal to size of profile - 300.

Distances are summarized for each language and divided by number of n-grams in document profile, this way we receive information about average n-gram distance, which does not depend on a number of n-grams in profile. In standard n-gram language identification language with the lowest profile distance is considered a document language, because they expect to have profiles for all languages it would classify. I've added threshold and requirement to have average distance lower than this threshold. In case of larger average distance I consider identification unsuccessful and document language out of training set.

Estimating threshold To estimate threshold and size of the training set I've compiled collection of 100 web pages from various sources for each supported language and one set containing few unsupported languages, Polish, German and French. In order to estimate reasonable threshold value, I've tested all files in collection tested various values of threshold and counting success rate. Figure 6.2 dependency of success rate on threshold value. Language identification is most successful for values of threshold from 175 to 178 and classified 98% of documents in test suite correctly. For higher values more false positives appear and documents in unknown language are incorrectly classified as one of the supported languages. For lower values more documents of known language are incorrectly discarded as an unknown language.

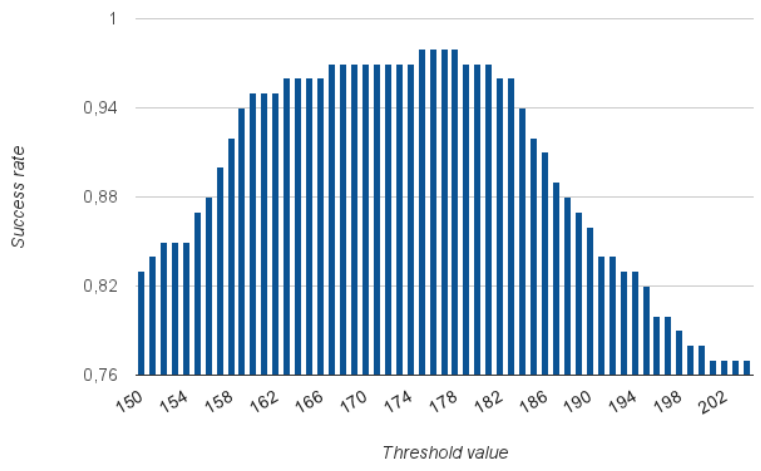


Figure 6.2: Estimation of threshold

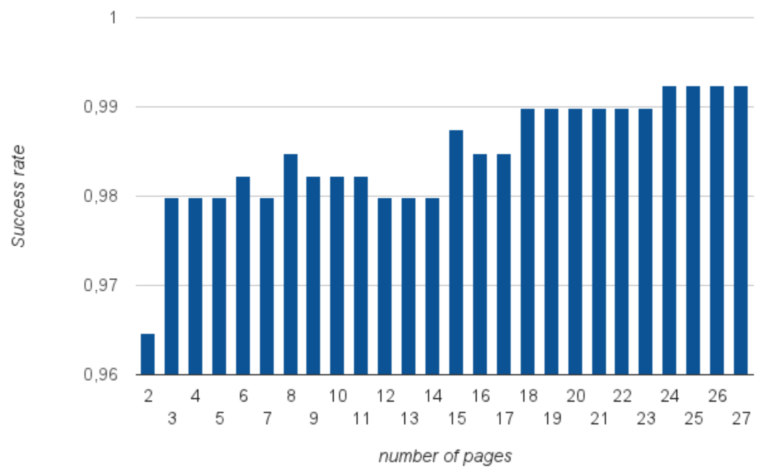


Figure 6.3: Estimation of size of training set

Estimating the size of the training set In order to estimate the size of the training set I've run similar test to the one for threshold estimation. This time I've fixed threshold at 178 and I was changing number of documents used for training. Average document in training set has around 500 words. In terms of n-grams more is always better, as display in Figure 6.3 language model starts stabilizing around 20 documents and with 24 documents model achieved more than 99% accuracy on testing set.

6.2 Recipe extraction

Extracting recipe is split into several steps.

1. Language identification
2. Tokenization and Stemming
3. Bayes classification
4. Grouping preparation and ingredients
5. Noise removal
6. Amounts extraction
7. Validating recipe quality

Language identification Web page language is identified by using n-gram ranks comparison. For supported languages appropriate gazetteer, stemmer and tokenizer are selected.

Tokenization and Stemming Language specific tokenizer is applied onto the document, which generates annotations for words, numbers, lines, sentences and time. Each word is processed by language specific stemmer, which adds word stem as annotation feature.

6.2.1 Bayes classification

Bayes classifier was trained on recipes from Czech recipe database sRecepty.cz. This site contains nearly 4000 recipes annotated with microdata, which served as good source of classified data. I've extracted these recipes and saved them as json files so I can rerun next steps without loading thousands of file html documents into memory and parsing them.

Background data was extracted from Czech dbpedia, which is linked data version of wikipedia. Thus, its data can be requested by SPAQL, which made easy to download abstracts of 500 pages linked from Food resource and it's children.

For generating bayesian model I've created generator class accepting recipe extractor, which serves as the middleware between the generator and documents. For the Czech model I've used json version of the extractor which provides generator with preparation and instructions from json files.

Creating stemmer In order to generate memory efficiently classifier model I had to stem the input. To do so, I've used the Hunspell stemmer and provided it with Czech affix file and dictionary used in OpenOffice. Then, I've optimized the stemmer by removing unused words and rules by testing it against sRecepty database.

Creating classifier The classifier classifies features into 3 classes - ingredients, preparation and background. Stemmed words of each block, excluding numbers, were used as the input features for the classifier. Classifier model and stemmer state were then saved as json file.

6.2.2 Classifying document parts

Classifier is executed on each line and sentence of the document to classify lines as ingredients and sentences as preparation step. Resulting probability numerators along with confidence are saved as annotation. The confidence is defined as probability numerator of positive class divided by sum of probability numerator of background and positive class. Background class was chosen as the negative class, because I'm interested whether the feature set should be a part of the recipe or not.

Preparation problems A lot of recipes contain instruction to do something with a lot of ingredients, like put them into bowl and mix them. To improve classification score of these sentences I'm adding part of the ingredient score to the preparation. To do so I'm allowing preparation to be maximum of preparation score and average of preparation and ingredient score.

6.2.3 Ingredient amounts

From each ingredient line is extracted amount, which can be either fully defined amount as number and unit or just unit or number. Value is set to 1 when no amount was found in the line.

6.2.4 Creating blocks

For completing recipe I'm bulking recipe parts into blocks - list of ingredients and preparation steps. These blocks are created bulking parts that are close to each other, which helps to remove noise (e.g. some sentences in comment section). Maximum distance between parts in block is three time size of the added part. This distance was determined by testing it on hundred recipes. After creating blocks, parts with confidence value lower than 0.99 are removed.

6.2.5 Pairing ingredients

The Final step of recipe extraction is pairing ingredients with preparation step. For each block from previous step is calculated overall weight, which represents a significance of the block in the page. This weight is calculated as the median confidence value multiplied by the logarithm of number of its parts. This weight express a need for both quality and amount of information. The Logarithm in the equation expresses quickly decreasing need for more information while confidence express the constant need for quality.

Each ingredient block is matched with each preparation block and for each pair is computed as an overall weight of the pair. This process is presuming that ingredients and preparation steps are usually close to each other on the page and favors these pairs. The pair weight is computed by multiplying the inverted value of a logarithm of their character distance and sum of the overall weight of ingredients and preparation. This final value represents the significance of the recipe in the page.

6.2.6 Validating recipe quality

Recipe parser doesn't use any pre-flight recipe detection algorithm. Instead, it tries to extract recipe from the page and then test if the extracted recipe is good enough. To accept extracted recipe as a valid one, the recipe must contain at least two ingredients and one preparation step, also confidence value of recipe parts has to be more than 0.99.

6.2.7 Summary

In this section, I've described steps of implemented recipe parsing algorithm. The algorithm identifies the language, uses optimized Hunspell stemmer and language specific tokenizer to prepare the document and classifies its segments by trained Bayes classifier. Finally, it aggregates classified parts and merges them into a single recipe, representing most significant recipe on the page.

CHAPTER 7

Application

The application is available on github.com/Ficik/recipes along with link to demo application and documentation, which are hosted on github pages: ficik.github.io/Recipes.

7.1 Using Application

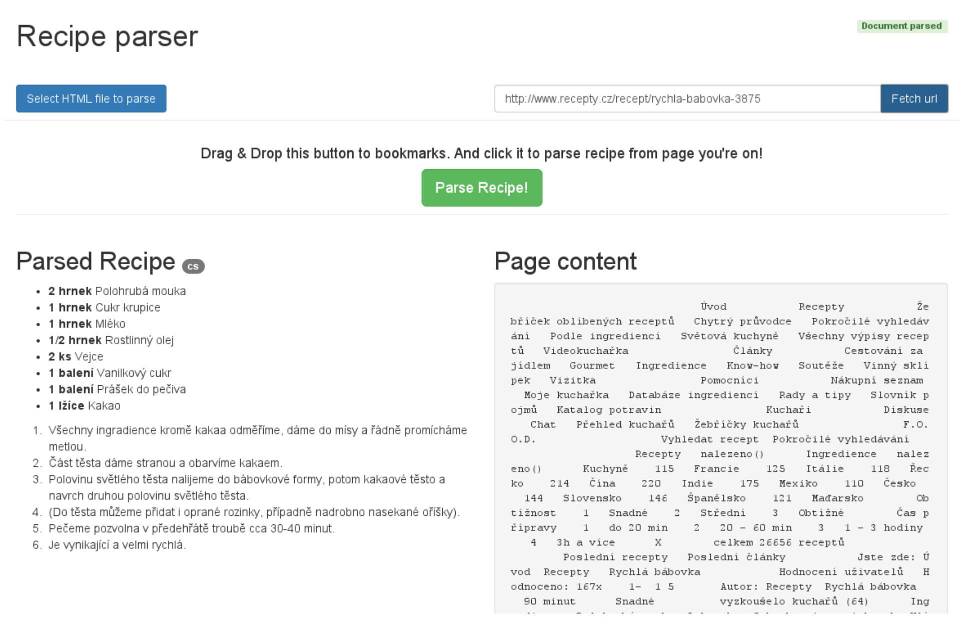
7.1.1 Interface of Demo Application

Web application allows user to upload an HTML or submit link to online page. Provided page is parsed and its textual form is displayed in the right panel while parsed recipe is displayed in the left panel tagged with schema.org/Recipe microdata. If no recipe is parsed, application reports this information in the top right notification space and leaves the left panel empty.

Currently, the only fully supported language is Czech, but the application is also capable of recognizing English and Slovak language. Language code of recognized language is displayed just next to the headline in the left panel.

Application displays notifications in top right corner, informing the user about current processing status and errors.

Browser support This demo application is not meant to be used by the end-user. The end user is expected to use the bookmarklet version of the parser. Thus, this application wasn't optimized for older browsers and was tested only in the latest versions of Google Chrome and Firefox. Because Internet Explorer and mobile browsers lack TextDecoder API, the upload feature is limited to utf-8 encoded documents only.


Recipe parser
Document parsed
Select HTML file to parse
http://www.recepty.cz/recept/rychla-babovka-3875
Fetch url
Drag & Drop this button to bookmarks. And click it to parse recipe from page you're on!
Parse Recipe!
Parsed Recipe
cs
Page content

- **2 hrnek** Polohrubá mouka
- **1 hrnek** Cukr krupice
- **1 hrnek** Mléko
- **1/2 hrnek** Rostlinný olej
- **2 ks** Vejce
- **1 balení** Vanilkový cukr
- **1 balení** Prášek do pečiva
- **1 lžíce** Kakao

1. Všechny ingradience kromě kakaa odměříme, dáme do mísy a řádně promícháme metlou.
2. Část těsta dáme stranou a obarvíme kakaem.
3. Polovinu světlého těsta nalijeme do bábovkové formy, potom kakaové těsto a navrch druhou polovinu světlého těsta.
4. (Do těsta můžeme přidat i oprané rozinky, případně nadrobno nasekané oříšky).
5. Pečeme pozvolna v předehřátě troubě cca 30-40 minut.
6. Je vynikající a velmi rychlá.

```
                                Úvod          Recepty          Že
bříček oblíbených receptů   Chytrý průvodce   Pokročilé vyhledáv
ání   Podle ingrediencí   Světová kuchyně   Všechny výpisy recep
tů   Videokuchařka                Články           Cestování za
jídlem   Gourmet   Ingredience   Know-how   Soutěže   Vinný skli
pek   Vizitka                Pomocníci           Nákupní seznam
  Moje kuchařka   Databáze ingrediencí   Rady a tipy   Slovník p
ojmů   Katalog potravin                Kuchaři          Diskuse
   Chat   Přehled kuchařů   Žebříčky kuchařů               F.O.
O.D.                  Vyhledat recept  Pokročilé vyhledávání
                   Recepty   nalezeno()      Ingredience   nalez
eno()      Kuchyně    115   Francie    125   Itálie    118   Řec
ko    214   Čína    220   Indie    175   Mexiko    110   Česko
  144   Slovensko    146   Španělsko    121   Maďarsko        Ob
tížnost    1   Snadné    2   Střední    3   Obtížné        Čas p
řípravy    1   do 20 min    2   20 - 60 min    3   1 - 3 hodiny
   4   3h a více      X        celkem 26656 receptů
        Poslední recepty   Poslední články           Jste zde: Ú
vod  Recepty   Rychlá bábovka            Hodnocení uživatelů   H
odnoceno: 167x    1-  1 5        Autor: Recepty  Rychlá bábovka
  90 minut      Snadné         vyzkoušelo kuchařů (64)       Ing
```

Figure 7.1: Demo application interface

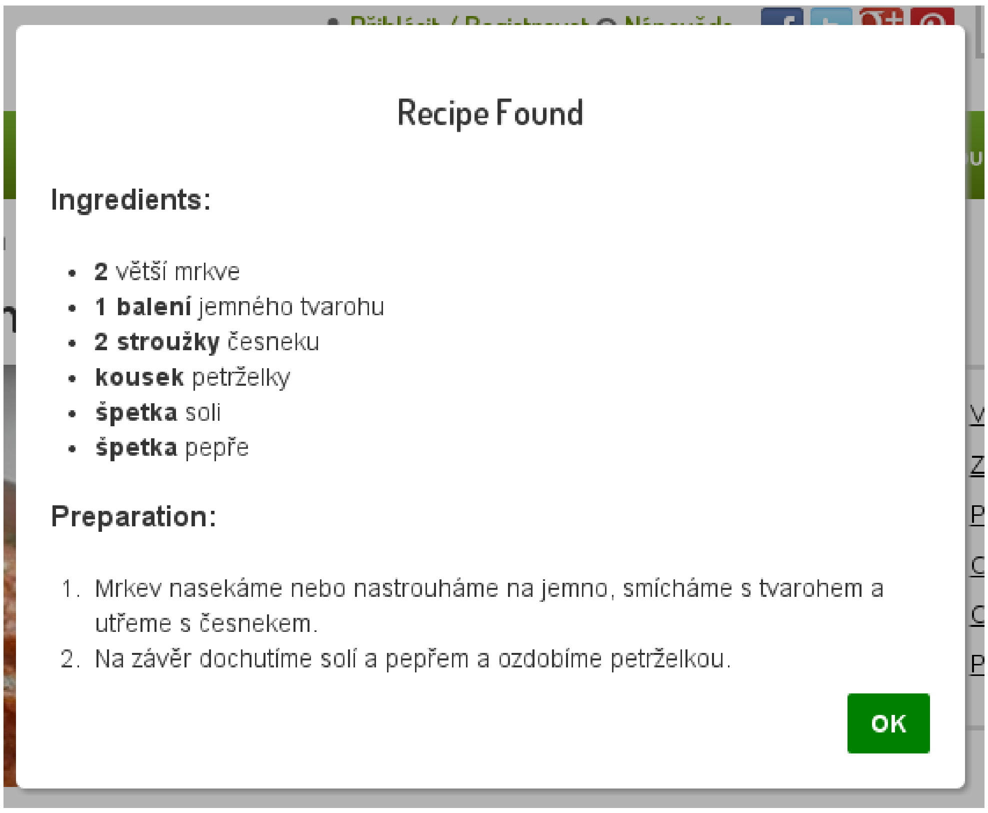
URL fetch support The application is pure client-side javascript application and due to CORS it cannot directly fetch pages from other domains. To bypass this limitation, application queries cors-enabled Yahoo API to fetch the webpage for it. Unfortunately some pages (for example those hosted on Amazons' AWS) block this Yahoo service.

Problems with encoding and URL fetching exists only in demo application and not bookmarklet and for that reason it's not necessary to solve them. The problem with encoding doesn't exist because the bookmarklet reads already by browser decoded HTML. The CORS problem also doesn't exists because the bookmarklet doesn't need access to 3rd party domain and its own code is loaded by a script tag to which the CORS restictions are not applied.

7.1.2 Bookmarklet

The demo application has green bookmarklet button which can be dragged into browser bookmarks. A bookmarklet is javascript code that can be saved in users bookmarks and execute on any page by clicking on it in the bookmarks. This is possible thanks to the javascript pseudo-protocol which allows interpreting a link starting with 'javascript:' as javascript code.

In this case, the bookmarklet code adds new script element, which loads recipe parser and the result presentation code. After loading, the parser code, the browser executes it and the application starts the parsing process. After a few hundred of milliseconds, a result window is displayed informing the user



Recipe Found

Ingredients:

- **2** větší mrkve
- **1 balení** jemného tvarohu
- **2 stroužky** česneku
- **kousek** petrželky
- **špetka** soli
- **špetka** pepře

Preparation:

1. Mrkev nasekáme nebo nastrouháme na jemno, smícháme s tvarohem a utřeme s česnekem.
2. Na závěr dochutíme solí a pepřem a ozdobíme petrželkou.

Figure 7.2: Bookmarklet overlay showing recipe

whether the page contains recipe and if so, it displays its ingredients and preparation steps.

After closing the result window, all globals and DOM elements that were added by the script are removed.

Browser extension Today, bookmarklet is a universal way to create browser extension, even though they require manual user invocation. Same code used in the bookmarklet, with some additional boilerplate can be used to create native Google Chrome or Firefox extension. These browsers allow user to write browser extensions in javascript and after packing them into zip package with defined structure and manifest, they can be uploaded to extension stores.

Main advantage of browser extension is automatic invocation on specified pages and that they don't require server to serve application code. The later is especially useful for storing larger models, since user is no longer limited by their bandwidth and only their memory.

The main disadvantage is requirement of having multiple codes to support multiple browser. The ES6 module system and build tools like gulp or grunt

```
1 npm install -g bower gulp
2 npm install
3 bower install
4 gulp build
5 gulp
```

Listing 7.1: Installation process

makes this task way easier than it was in the past, but it still doesn't solve the problem with browser without javascript extension support.

7.2 Installation

Node.js is required and git recommended for setting up own copy of application. Installation process listed on Listing 7.1 is quite standard in web development. Node.js installation package contains package managing tool - NPM. Using this tool install globally package manager for frontend - bower and task runner - gulp (line #1). Then install locally application build dependencies. Dependencies are listed in package.json (line #2). Then install runtime dependencies by bower. Finally, either build application into dist directory (line #4) or start local Hyper Text Transport Protocol (HTTP) server and watch for changes in application code to build it (line #5). The later rebuilds affected parts of code on each file change and reloads web page in browser after build.

Application root is in dist directory, It should be possible to run application just by opening index.html in this folder but not all features are supported on file protocol (ajax and bookmarklet), thus HTTP server for serving files from this directory is required. Using gulp for serving files is recommended, but not required.

7.3 Building your own model

The main part of the recipe parser is a model that contains parser, stemmer and classifiers configuration. This model can be generated by a script like build.js. This sample script loads HTML files of Czech recipes as training files into the model generator, provides it with a Hunspell dictionary and affix files and creates an instance of RecipeParser. Finally, configuration of this instance is serialized into a JSON file in the dist directory.

7.4 Summary

In this chapter, I've presented demo user interface and a bookmarklet, which represents a primary way the end-user will interact with the presented application. Described bookmarklet place in the browser extension world and noted its potential to be built as a browser extension. Finally, I've covered how to install the application and how to use it as a developer.

CHAPTER 8

Comparation with NLP toolkits

The application for extracting recipes is built on the top more general library - NLP.js; which is meant to provide web developers with NLP tools. Like other NLP toolkits provides tools like tokenizer, stemmer, classifiers.

8.1 Annotation oriented

Tools like python NLTK or nodejs Natural, which acts as toolkit and provides NLP tools operating with basic datatypes like stings, arrays and maps. On the other hand NLP.js is based on its Document and Annotation objects, which are heavily inspired by java NLP tool GATE, and most operations are recommended to be done on top of these abstractions. They were created to provide unified interface for querying and saving results. This allows developer creating more complex tools, that can take results of other tools in account.

The other javascript implementation of NLP tools is Natural. But unlike Natural, my implementation doesn't depend on any nodejs library.

Document and Annotation objects have small interface and they can be easily extended. And thanks to duck typing instead of strong type checking, it's even possible to use custom implementations.

8.2 Available tools

For now the NLP.js offers just a subset of tools that offer older libraries. Current list contains following tools:

- HTML parser

- Bayes classifier
- NGram classifier
- Adaboost classifier
- Czech and English snowball stemmer
- Huspell based stemmer generator
- Word frequency analyzer
- Generic and Czech tokenizer
- Accent removal tool
- String metrics

8.3 Platform limitations

This library is not intended to compete with desktop solutions, but should act as their companion, by provided way to use models generated from corpus of hundreds or even thousands of documents in client browser where they can be used on small set of documents.

Disadvantages Application is limited by its platform, where only one thread is available and all data has to be in memory and access to external resources is limited to HTTP requests with CORS restrictions. These limitations render this application unsuitable for processing megabytes of documents, because of no direct access to file system, and thus no effective way to text stream processing. It's also not suitable for application with extensive database lookups, because of limitations and low efficiency of AJAX calls.

Advantages On the other hand this solution is suitable for application which requires limited data exposure, because of data privacy policy. It can also help with decreasing required server processing power and so the production cost in case of large amount of simple text mining tasks.

Changes in future Some of the disadvantages might disappear in the future with new API provided by browsers. For example the connection to the external database for extensive entity querying might be more efficient with HTML5 WebSockets, which would keep open connection for querying. For frequently used applications a database of a few megabytes of data can be stored in HTML5 localStorage. Access to file system might be solved with a new HTML5 FileAPI.

The technologies rapidly implemented by browsers can solve problems with sizes of database and corpus, and add new possible use cases, but one problem won't be probably solved. The JavaScript is single threaded interpreted language and this will always limit its performance.

8.4 Summary

The proposed library is not a direct competitor for desktop NLP solutions, because of current platform limitations. Even though, the application provides a developer with a set of frequently used tools in NLP applications. Same tools that are provided by server-side solutions, like Native, NLPTK or GATE. It is inspired by GATE Document with Annotations structure and provides a similar interface.

CHAPTER 9

Results

9.1 Application

9.1.1 Library

Generic library is available for use and extension that provides stable and easy to understand API. This library covers basic needs of corpus linguistic and text mining purposes. It can be easily extended either by forking library on github[1] or creating external library listing this one as dependency. The library currently covers plain text and HTML document processing, tokenizing, stemming of Czech and English words and n-gram calculations.

9.1.2 Language identification

The application built on implemented library is capable of identifying language of a web page containing a recipe. During testing with 400 pages containing recipes in both supported and unsupported languages, application achieved 99% accuracy.

9.1.3 Recipe extraction testing

In order to test the success rate of extraction process I needed tagged data; Recepty.cz is another largest Czech recipe website, that tags its recipes with Microdata. I've downloaded 200 recipes from this page and created test suite by extracting ingredients and preparation steps using both implemented Microdata extractor and my application. Results of both methods were cleaned by trimming additional spaces and concatenated. Finally, the distance between ingredients and preparation blocks were computed using Levenshtein

[1]https://github.com/Ficik/nlpjs

distance. Levenshtein distance or minimal edit distance computes minimal number of edit operations, adding a character, removing a character, replacing a character, that are needed to transform one string to the other.

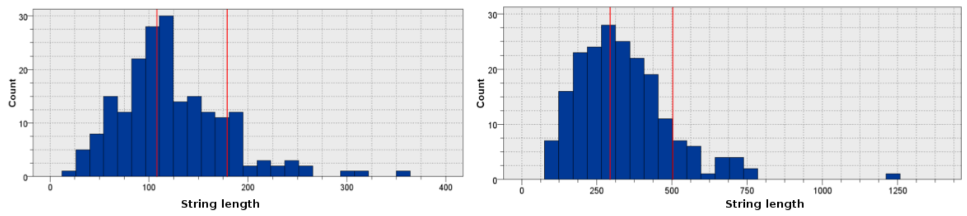


Figure 9.1: Distribution of lengths of blocks of ingredients (left) and preparation (right)

Accuracy Testing showed very good accuracy when classifying the ingredients (Figure 9.2). I've set a threshold for the correctly classified ingredient to 5 as I didn't manage to fully clean data returned from the extractor. Correctly classified were 169 recipes from 200 (84.5 %). Additionally, I've counted classifications that were extremely off; by at least one-third of the length of correct classification. These cases where parser extracted something else then required recipe was 6 %. This leaves 9.5 % of misclassifications where only some ingredients were extracted.

For the preparation steps, I've set the threshold for correct classification to 15 edits since average preparation is 3 times longer than average ingredients. This returned 82 % accuracy. I've also calculated classifications that were completely wrong, the same ways as I did for the ingredients. This resulted in only 12 completely wrong classification (6 %).

False positives Detection capability was tested using 200 random pages from Wikipedia, newspapers and blogs. Application successfully reported that page doesn't contain a recipe in 186 cases. Thus, its success rate on tested sample was 93 %.

Performance Figure 9.4 displays time necessary for extraction process with respect to raw document size. As we see average document size was around 29 kB and they were processed within 67 ms on average on a single thread of 2.4GHz processor. Loading resources vary on the Internet connection and thus loading times were not included in results, but since the worst time in was around 200 ms and resources needed to load were 200 kB, this application should process average document under a second with any nowadays Internet connection. The application is small enough to stay in the browser cache which can, with proper server setup, result in zero loading times.

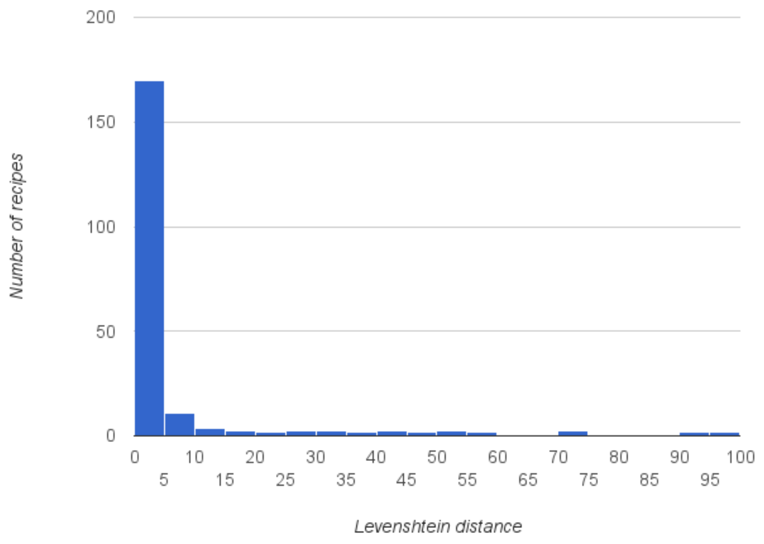


Figure 9.2: Histogram of Levenshtein distances of ingredients from desired result

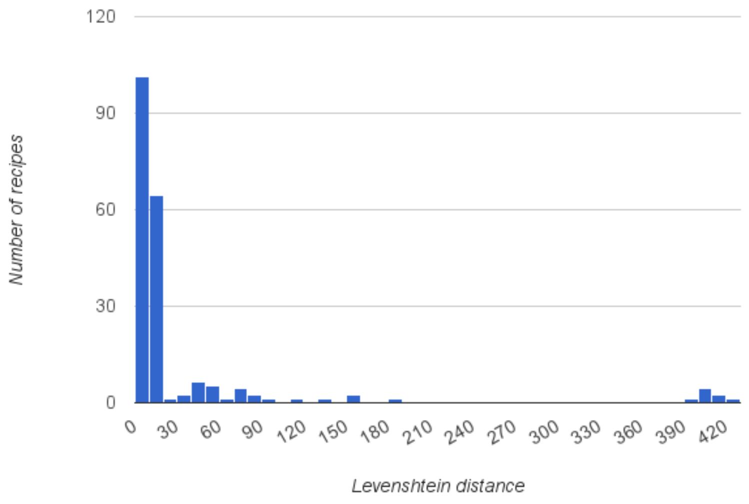


Figure 9.3: Histogram of Levenshtein distances of preparation from desired result

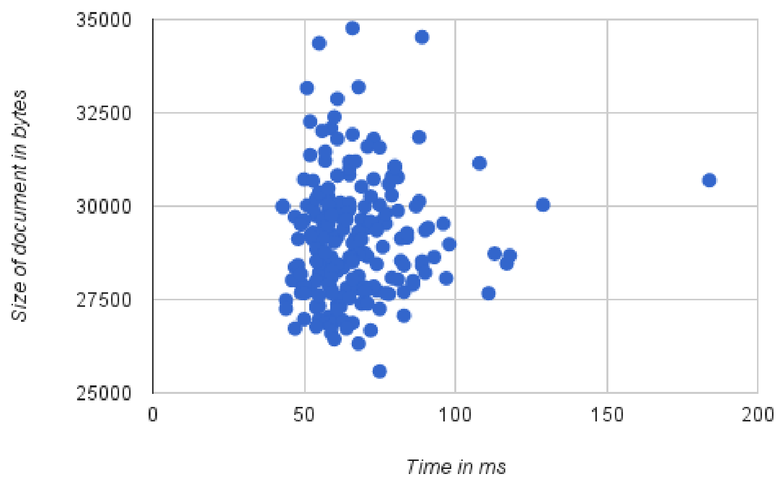


Figure 9.4: Processing times with respect to size of parsed document

Conclusion

I've successfully developed library capable of detecting and extracting recipes from the web page, with a decent success rate. Extracted recipe is presented to a user in modal window annotated with microdata. I was able to minimize the size of the application and necessary dictionaries, I was able to get it in 200kB using stemming, and available compression. This size allows reasonable loading times on any today's internet connection. The application is capable of processing average web page within hundreds of milliseconds which is, from a user perspective, almost instant. Along with recipe extraction the application I've developed a library which can be used for similar text mining applications. This library is distributed using nowadays standard distribution tools and modern module system. Library contains a subset of most important tools available in desktop toolkits like Natural.js and NLPTK - multiple classifiers, string metrics, stemmers, tokenizers and HTML parser. Due to the simplicity of gazetteer formats, there was no need to develop parser for them, instead I've implement tool for their application. Library contains third party stemmers, based on a snowball, for Czech and English as well as my implementation of Hunspell based stemmer.

Future extension

Missing tools Implemented library is still missing some important tools; one of those are POS tagger and language versions of existing tools. Next step is to add or integrate these features.

Understanding of the content The document is already annotated with lexical categories, next step would be building relations between words. The very first step would be forming a relation between subject and verb. This relation is one of the basic elements for rules creation; which is fundamental of semantic analysis.

Personal gain

By writing this thesis and implementing this application and library, I've extended my knowledge in NLP domain and gained experience in implementing complex text mining application. Additionally, I've created a library that will be implemented in near future into a social network and used daily by thousands of people.

Bibliography

[1] Arnold, R.; Bell, T.: A Corpus for the Evaluation of Lossless Compression Algorithms. In *DCC '97: Proceedings of the Conference on Data Compression*, Washington, DC, USA: IEEE Computer Society, 1997, str. 201.

[2] Banchs, R. E.: *Text Mining with MATLAB.* New York: Springer, 2012.

[3] Bell, T. C.; Cleary, J. G.; Witten, I. H.: *Text compression.* Englewood Cliffs: Prentice Hall, 1990.

[4] Bikel, D.; Zitouni, I.: *Multilingual Natural Language Processing Applications.* Indianapolis: IBM Press, 2012.

[5] Bird, S.; Klein, E.; Loper, E.: *Natural language processing with Python.* " O'Reilly Media, Inc.", 2009.

[6] Brisaboa, N.: An efficient compression code for text databases. In *25th European Conference on IR Research (ECIR 2003)*, 2003, s. 468–481.

[7] Cavnar, W.: Using an n-gram-based document representation with a vector processing retrieval model. *NIST SPECIAL PUBLICATION SP*, 1995: s. 269–269.

[8] Conway, D.; White, J. M.: *Machine Learning for Hackers.* Sebastopol: O'Reilly, 2012.

[9] Downey, A. B.: *Think Bayes.* Sebastopol: O'Reilly, 2013.

[10] Indurkhya, N.; Damerau, F. J.: *Handbook of Natural Language Processing.* Boca Raton, Florida: CRC Press, 2010.

[11] Ingersoll, G. S.; Morton, T. S.; Farris, A. L.: *Taming text: how to find, organize, and manipulate It.* Manning Publications Co., 2013.

[12] Jurafsky, D.; Martin, J. H.: *Speech and Language Processing.* Upper Saddle River, New Jersey: Pearson, 2008.

[13] Miner, G.; Elder, J.; Hill, T.; aj.: *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications*. Waltham, USA: Academic Press, 2012.

[14] Moffat, A.: Arithmetic coding revisited. In *ACM Trans. on Inf. Systems Vol 16*, July 1998, s. 256–294.

[15] Nather, P.: N-gram based Text Categorization. *Lomonosov Moscow State Univ*, 2005.

[16] Padro, M.; Padro, L.: Comparing methods for language identification. *Procesamiento del lenguaje natural*, 2004.

[17] Powel, M.: The Canterbury Corpus. November 2001, [Cited 2011-10-12]. Dostupné z WWW: <`http://corpus.canterbury.ac.nz/index.html`>

[18] Pustojovsky, J.; Stubbs, A.: *Natural Language Annotation for Machine Learning*. Sebastopol: O'Reilly, 2013.

[19] Russell, M. A.: *Mining the Social Web*. Sebastopol: O'Reilly, 2011.

[20] Wasserman, S.; Faust, K.: *Social Network Analysis: Methods and Applications*. New York: Cambridge University, 1994.

APPENDIX A

Glossary

JavaScript	Programming language used mostly in web browsers.
Microdata	Microdata provides semantic information about content of HTML tag by referencing to type of the tag content defined URI.

APPENDIX B

Acronyms

AJAX Asynchronous JavaScript and XML.

AMD Asynchronous Module Definition.

API Application Programming Interface.

BDD Behavior Driven Development.

CORS Cross-origin resource sharing.

CSS Cascading Style Sheets.

DOM Document Object Model.

HTML HyperText Markup Language.

HTTP Hyper Text Transport Protocol.

IDE Integrated Development Environment.

JSON JavasSript Object Notation.

JSONP JSON with padding.

NaCl Native Client.

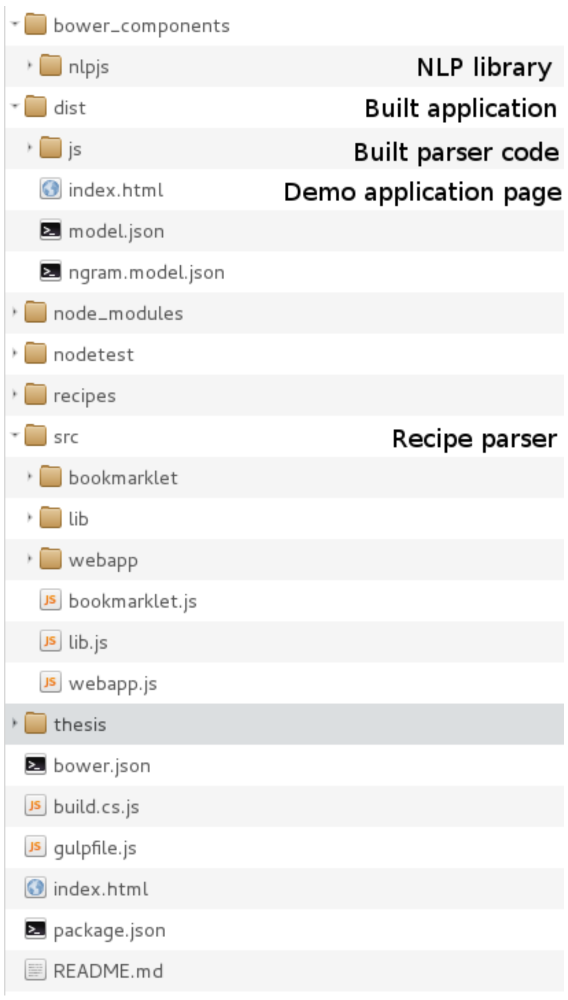
NLP Natural Language Processing.

NPM Node package manager.

POS	Part-Of-Speech.
RDF	Resource Description Framework.
RegExp	Regular Expression.
SPARQL	Simple Protocol and RDF Query Language.

APPENDIX C

CD content



```
bower_components
  nlpjs                NLP library
dist                   Built application
  js                   Built parser code
  index.html           Demo application page
  model.json
  ngram.model.json
node_modules
nodetest
recipes
src                    Recipe parser
  bookmarklet
  lib
  webapp
  bookmarklet.js
  lib.js
  webapp.js
thesis
bower.json
build.cs.js
gulpfile.js
index.html
package.json
README.md
```