MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6

CH-1007 Lausanne

Master of Science HES-SO in Engineering

Information and Communication Technologies (TIC)

SwigSpot - Creation of a Swiss German Dataset

Supported by the School of Engineering and Architecture of Fribourg

Lucy Linder

Supervised by
Prof. Dr. Jean Hennebert
Prof. Dr. Andreas Fischer
At iCoSys, *Institute of Complex Systems*

Expert
Dr. Claudiu Musat, Research Director
Artificial Intelligence & Machine Learning Group
At *Swisscom*

Fribourg, HES-SO//Master, 2018

Abstract

In the past years, Swiss German has become more and more pregnant in written contexts. However, there are still few natural language processing (NLP) studies, corpora or tools available. As a result, support for Swiss German dialects is non-existent in our day-to-day interactions with technology. To automate the treatment of Swiss German and foster its adoption in online services, the SwigSpot project aimed at creating a large corpus of Swiss German sentences available to researchers.

Using Machine Learning techniques, we first created a model able to discriminate between French, English, Italian, German and Swiss German languages using training material from available corpora. We then made the assumption that the Web was the most likely source of unseen sentences. In a first attempt, we crawled more than one million landing pages from the Swiss .ch domain. It yielded very poor results, fewer than 1'000 new Swiss German sentences, suggesting that Swiss German is mostly used in more informal contexts such as blogs or social media. In a second attempt, we used a search engine and manual "seeds" to gather URLs likely to have Swiss German content. Crawling those URLs yielded far better results: using only 5 seeds, 211 URLs and 3 minutes of processing time, we gathered about 8'000 unseen Swiss German sentences.

This project is a Master's Deepening Project proposed by Swisscom's new Artificial Intelligence group.

Keywords: language identification, Web scraping, corpus creation, machine learning.

About Lucy Linder

Fascinated by literature from an early age, I hold a Bachelor of Arts in Sociology, Communication Sciences and Philosophy. Just before starting my Master in the same area of study, I attended an introductory course in Java, which was a revelation for me.

I was immediately seduced by the logical rigor of the languages, the beauty of the code and the magic of the programs. The freedom entailed by the control of computers illuminated my evenings for six months, until I decided to leave everything to enter the Haute école d'ingénierie et d'architecture de Fribourg. After a little more than five years, my passion is still strong and my thirst for learning seems inexhaustible.

This deepening project was a great opportunity to discover new aspects of Machine Learning, Web scraping and project management. Working in partnership with a company such as Swisscom and discovering Swiss German in a new light made the experience even more valuable.

Contents

1 Introduction **1**

1.1 Introduction . . . 1
1.2 Project goals . . . 2
1.2.1 Swisscom needs . . . 2
1.2.2 Project statement . . . 2
1.3 Project overview . . . 2
1.3.1 Steps . . . 2
1.3.2 Project outcomes . . . 2
1.4 Schedule and methodology . . . 3
1.5 Organization of the report . . . 4

2 State of the Art **5**

2.1 About Swiss German . . . 5
2.2 Available corpora . . . 6
2.2.1 ArchiMob . . . 6
2.2.2 Swiss SMS corpus . . . 6
2.2.3 NOAH . . . 6
2.2.4 SB-CH Corpus . . . 7
2.2.5 Leipzig . . . 7
2.2.6 An Crúbadán . . . 8
2.3 Summary . . . 8

3 Quickstart Dataset **9**

3.1 What do we need ? . . . 9
3.1.1 Granularity of the data . . . 9
3.1.2 Usage of the dataset . . . 9
3.2 Extracting sentences from available corpora . . . 10
3.2.1 Sentences from the NOAH corpus . . . 10
3.2.2 Sentences from the ArchiMob corpus . . . 11
3.2.3 Sentences from the Swiss SMS corpus . . . 11
3.3 Final dataset . . . 12
3.3.1 Composition . . . 12
3.3.2 Quality . . . 13
3.4 Summary . . . 14

4 Language identification: theory **15**

4.1 Introduction . . . 15
4.2 LID as a supervised classification task . . . 16
4.2.1 Character N-grams . . . 16
4.2.2 Preprocessing . . . 17

4.2.3 Feature extraction . . . 17
4.2.4 Classification . . . 18
4.3 Summary and selected approaches . . . 20

5 Language identification: implementation **22**

5.1 Technologies . . . 22
5.2 Timeline . . . 23
5.3 Structure of the code . . . 23
5.4 Data handling . . . 24
5.4.1 Sets . . . 24
5.4.2 Sanitization . . . 24
5.4.3 WrappedVectorizer . . . 25
5.5 Naive implementations . . . 25
5.5.1 Naive Vectorizer . . . 25
5.5.2 Naive Identifier . . . 26
5.6 Sklearn . . . 26
5.7 Neural Networks . . . 28
5.8 A complete example . . . 29
5.9 Summary . . . 29

6 Language identification: results **31**

6.1 Final models . . . 31
6.1.1 Logistic regression . . . 31
6.1.2 SVM . . . 32
6.1.3 MultinomialNB . . . 32
6.1.4 Naive identifier . . . 32
6.1.5 Neural Networks . . . 32
6.2 Results . . . 33
6.3 Tests on real data . . . 34
6.4 Summary . . . 34

7 Data gathering **37**

7.1 Crawling the Web: a challenge . . . 37
7.2 Boilerpipe . . . 38
7.3 Swiss German Extractor, a first attempt . . . 38
7.3.1 Interfaces . . . 38
7.3.2 Implementation . . . 38
7.3.3 Extractors . . . 40
7.4 Summary . . . 40

8 The *.ch domain* approach **41**

8.1 The idea . . . 41
8.2 Implementation . . . 41
8.2.1 General overview . . . 41
8.2.2 The langid-microservice . . . 42
8.2.3 Database . . . 44
8.2.4 Spark crawler . . . 44
8.3 Execution . . . 47
8.4 Results . . . 48
8.5 Conclusion . . . 49

9 The *search Google* approach **50**

9.1 The idea . . . 50
9.2 Proof of concept . . . 51
9.3 Existing work . . . 53
9.3.1 CorpusBuilder . . . 53
9.3.2 BootCaT . . . 53
9.3.3 Leipzig . . . 55
9.4 Conclusion . . . 55

10 Conclusion **56**

10.1 Summary . . . 56
10.2 Review of the objectives . . . 57
10.3 Conclusion and perspectives . . . 57
10.4 Final word . . . 58
10.5 Acknowledgements . . . 58

Bibliography **59**

1 Introduction

Chapter's content

1.1 Introduction 1
1.2 Project goals 2
1.3 Project overview 2
1.4 Schedule and methodology 3
1.5 Organization of the report 4

1.1 Introduction

"*If you talk to a man in a language he understands, that goes to his head. If you talk to him in his language, that goes to his heart.*"

– Nelson Mandela

Swiss German ("Schwyzerdütsch" or "Schwiizertüütsch", abbreviated "SG") is the name of a large continuum of dialects attached to the Germanic language tree spoken by more than 60% of the Swiss population [1]. Used every day from colloquial conversations to business meetings, Swiss German in its written form has become more and more popular in recent years with the rise of blogs, SMS and social media. Authors and companies also begin to adopt their dialects in their work, for example in books or company reports [2].

Even though Swiss German is widely spread in Switzerland, there are still few natural language processing (NLP) studies, corpora or tools available [3]. As a result, support for Swiss German dialects is non-existent in our day-to-day interactions with technology.

"*If we could communicate with all machines as easily as with other people, our daily lives would be very much easier.*"

– Marc Steffen, Swisscom [4]

To automate the treatment of Swiss German and foster its adoption in online services, the first step is to gather resources into a usable dataset. Organizations such as the University of Zürich have already undergone this process, but at a very small scale. Given the increasing occurrences of Swiss German in online resources, we believe that a larger dataset can be produced.

1.2 Project goals

> ***tl;dr*** The goal of the SwigSpot project is to gather Swiss German resources into a well-designed corpus available to researchers. This corpus should contain unseen sentences *in context* and act as a complement to already available corpora.

1.2.1 Swisscom needs

Swiss German is a hot topic for the Swisscom team. Noemi Aepli, a computational linguist from the University of Zürich, attained more than 93% accuracy with an automated POS-tagger and currently works on word embeddings. Automatic normalization of Swiss German is a "holy grail", but is too big a challenge to tackle within this project. The most interesting area to work on, according to Swisscom needs, is thus the gathering of more Swiss German data samples. Ideally, those samples would be sentences *in context* and *tagged with dialects*.

The University of Zürich already scanned a lot of sources for Swiss German, including newspapers, blogs, novels and online resources. One of the challenges is thus to discover new sources.

1.2.2 Project statement

The project statement as given at the beginning of the project is the following:

> Existing text corpora of Swiss German texts are currently too small or too specific to enable the creation of more advanced NLP and machine learning services. The goal of the SwigSpot project is to gather Swiss German resources into a well designed corpus available to researchers. The use of machine learning and NLP methods is envisaged to support the text collection. This project is proposed by Swisscom's new Artificial Intelligence group and will include synchronization with this group.

1.3 Project overview

This section lists the steps and project outcomes determined after a first discussion with Swisscom and one week of work.

1.3.1 Steps

As shown in Figure 1.1, the creation of a corpus can be divided into two intertwined cycles, or phases: *data gathering* and *data consolidation*. In the first phase, we develop a model able to identify Swiss German. Once available, this model can be used to gather new resources that can in turn be used to improve the identification model. In a second phase, analyzes of the data may help refine the corpus and make new features available, for example dialect tags.

1.3.2 Project outcomes

According to the steps presented in Figure 1.1, this project focuses mainly on the first cycle. Thus, expecting outcomes are:

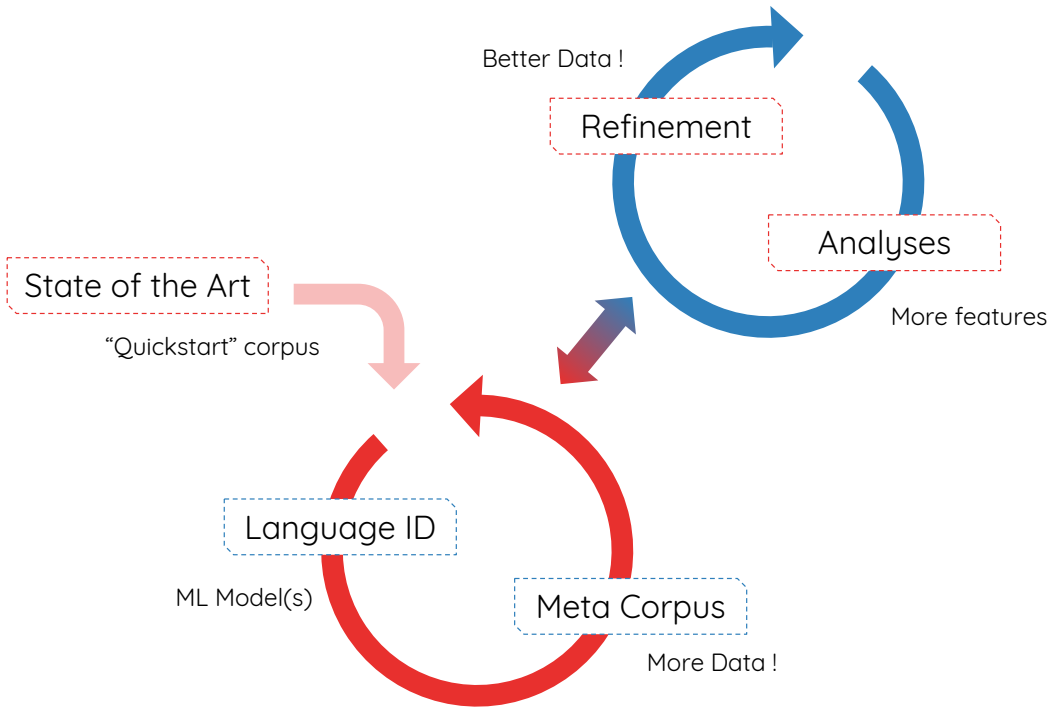


Figure 1.1: Phases of the project: data gathering and data consolidation.

1. language identification
   (a) ability to detect Swiss German;
   (b) (*nice-to-have*) ability to label sentences with dialects.
2. data gathering
   (a) programs and tools to find new Swiss German resources;
   (b) (*nice-to-have*) a corpus of new Swiss German sentences.

Note: point 2b is marked as *nice-to-have*, because we ignore if data exist that have not yet been processed by the University of Zürich or the Swisscom team.

1.4 Schedule and methodology

This project follows an *agile methodology* [5] with small iterations of 1-2 weeks. The backlog is updated after each iteration during a meeting with the supervisors. We defined five major milestones presented in Figure 1.2:

1. *State of the art*: analyses have been performed about available corpora and language identification; one or more ideas on how to collect new data have been defined;
2. *Quickstart dataset*: a minimal dataset for language identification based on existing resources has been prepared;
3. *Language ID*: a first language identification model is ready to use;
4. *Data Gathering*: a period of time is dedicated to gathering new data using various approaches

yet to be defined;

5. *Project end*: end of the project, the report has been produced.

1.5 Organization of the report

This report follows the flow of the project. In Chapter 2, we quickly introduce the Swiss German language and discuss the existing corpora. In Chapter 3, we focus on the creation of a quickstart dataset. We define our needs, review the available corpora on this point of view and present the dataset retained for further processing. Chapters 4 to 6 center around LID - *Language IDentification* - techniques using Machine Learning, from the state of the art to the development and evaluation of LID models for Swiss German. Chapter 7 to 9 present different approaches to crawling the Web and discovering Swiss German sentences.

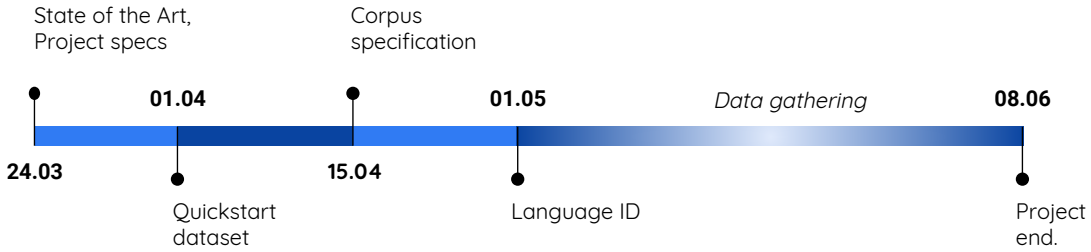


Figure 1.2: Major milestones

2 State of the Art

Chapter's content

2.1 About Swiss German 5
2.2 Available corpora 6
2.3 Summary 8

2.1 About Swiss German

Swiss German is best described as a *dialect continuum* with huge variations and no strict borders. As illustrated in Figure 2.1, it varies enormously between the regions without a clearly delimited scope. Some of those differences come from the influence of nearby languages: regions close to the Italian part of Switzerland will have more Italian components, etc [3]. While every region develops its own form of the dialect, Swiss German speakers are usually able to understand each other [6].

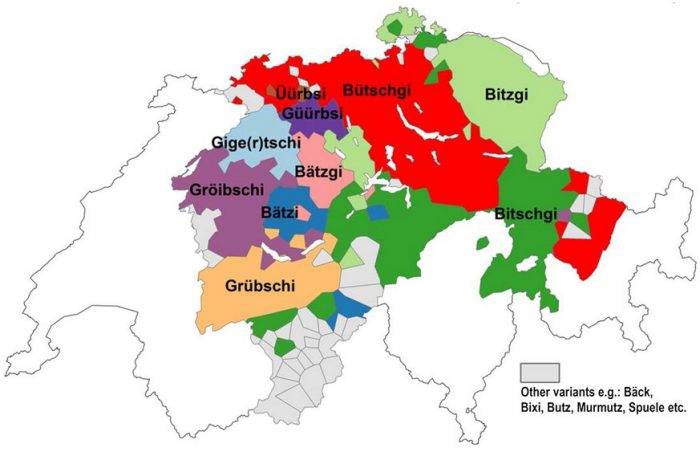


Figure 2.1: Swiss German word for "apple core" depending on the regions [7].

In the past, the concept of *diglossia* applied to Swiss German: speakers used their dialect in most oral situations, but systematically switched to Standard German in written contexts. However, this distinction is slowly disappearing. For the younger generation, writing in Swiss German has become natural. Noëmi Aepli is even talking about "*a cult status to write and publish in Swiss German*" [3].

Written Swiss German is very difficult to address in NLP as there is no writing standard. The orthography can vary a lot even inside a regional dialect and every speaker can develop its own

style. Another difficulty is the sparsity of resources, since 60% of the Swiss population only accounts for less than 0.066% of the world population…

2.2 Available corpora

Currently, few corpora of Swiss German resources are available, most of them developed by the University of Zürich.

2.2.1 ArchiMob

ArchiMob is described as

> a general-purpose corpus of spoken Swiss German suitable for studying linguistic microvariation and spatial diffusion with quantitative approaches [8].

The first version of the ArchiMob corpus, released in 2016, contains 34 transcripts (528'381 tokens) of semi-directive interviews with Swiss German speakers who witnessed the Second World War. The data can be downloaded in the form of XML documents.

Aside from transcription and text alignment, further analysis has been carried using both manual and automated tools. The final release contains normalization, dialect and Part-of-Speech (POS) annotations. *Normalization* is the process of mapping different but equivalent Swiss German words into a standard form (generally a Standard German word), while *POS-tagging* is the process of assigning a syntactic category, or role, to a word. When it comes to Swiss German, both processes are very challenging. For more information, we invite the reader to consult the ArchiMob documentation [9].

2.2.2 Swiss SMS corpus

The Swiss SMS corpus [10] consists of 25'947 SMS (650'000 tokens) sent by the Swiss public in 2009/2010. 41% of the SMS are labeled Swiss German (10'800 SMS, 288'400 tokens) with only a few misclassifications.

The corpus is available in two forms, both of which requiring a login and a web browser. The *SMS Navigator* lets us browse the data (cleaned and anonymised) by languages. The *ANNIS interface* is more complex and contains normalization and POS annotations as well. More information on the processing is available on their website [11].

2.2.3 NOAH

The NOAH's Corpus of Swiss German Dialects [3] is a compilation of sentences from various text genres (see Table 2.1) annotated with Part-of-Speech tags and dialect (if known). The initial version contains 73'616 tokens and has been extended a short time after with 41'000 new tokens, reaching a total of 115'000 tokens. The corpus can be downloaded as a zip archive containing five XML files, one for each type of source.

Text source	No. of tokens
Alemannic Wikipedia	20,135
Swatch Annual Report 2012	13,386
Novels from Viktor Schobinger	11,165
Newspaper articles	11,259
Blogs	17,671
Total	73,616

Table 2.1: NOAH Corpus composition, release 1.0

2.2.4 SB-CH Corpus

The *SpinningBytes Swiss German Corpus (SB-CH)* [12] is a corpus created by SpinningBytes in collaboration with the Zürich University of Applied Sciences (ZHAW). Published on Github, it references 165'916 sentences, 2'799 of which are labeled with sentiments.

For our purposes, this dataset is the least interesting. First, only 70% of the sentences are actually Swiss German, but no tag makes it possible to know which. Second, their dataset is composed of sentences gathered from NOAH, the Swiss SMS corpus, chatmania (a Swiss online chat) and Facebook. The first two are already known, while the Facebook sentences require an API key to be accessed (and are not guaranteed to still be available). Only the last source, chatmania, offers 90'898 new sentences. However, most of them are very short (one or two words) and/or contain foreign words, usernames, symbols and emojis, thus requiring cleaning before use.

Listing 1: Extract of `chatmania.csv`, SB-CH Corpus

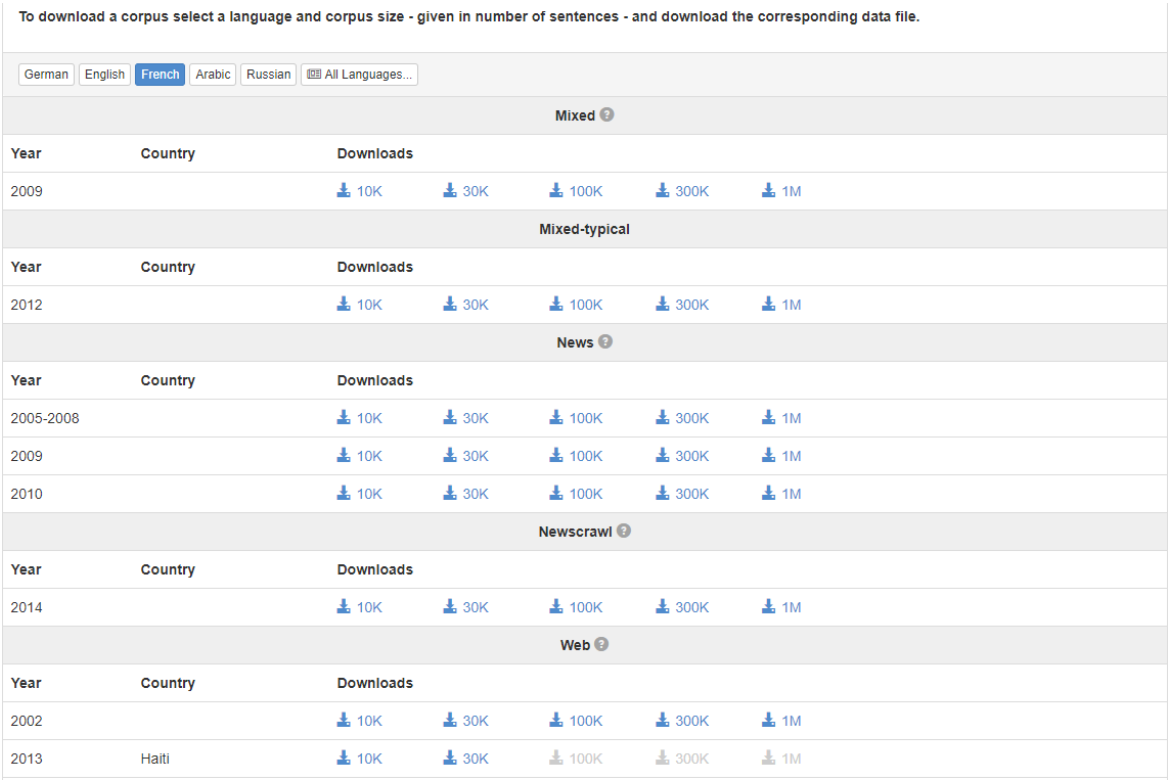
```
10731,"_nick_: ihm si <censored>`?"
10735,"hm *g"
10741,"gggg _nick_ muess lache hüt obe"
10743,"klopf klopf klopf"
10744,"_nick_ hoi du"
10747,"häppere = kartoffel"
10751,"fifty sh..s of grey"
10784,"g, danke"
10805,"_nick_: https://www.youtube.com/watch?v=yvljhtzks2m"
10806,"*ggg* _nick_"
10843,"irrelevant ... kompetend ... ja"
11126,"id stadt.....shoppen<<<<<"
11915,"That was very kind :))))))"
14157,"salü ***_-_-_-_ _nick_ _-_-_-_***"
```

2.2.5 Leipzig

The *Leipzig Corpora Collection* [13] is a collection of datasets similar in size and content for 136 languages. Each dataset consists of randomly selected sentences along with statistical information, with non-sentence and foreign-language material removed. The sources are either newspaper texts, Wikipedia texts, or texts randomly collected from the web (see Section 9.3).

The *Leipzig Corpora Collection* includes 2 datasets of Swiss German, each of 100'000 sentences. This discovery has been made at the end of the project, so it will be ignored in the next chapters. See Section 9.3 for details.

As shown in Figure 2.2, their interface lets us choose the size, the year and the source of the data we want to download. The zip archive consists of multiple files, including the sentences as a text file, the list of the sources and a MySQL dump of the collection and the statistics.



To download a corpus select a language and corpus size - given in number of sentences - and download the corresponding data file.

German | English | French | Arabic | Russian | All Languages...

Mixed

Year	Country	Downloads				
2009		10K	30K	100K	300K	1M

Mixed-typical

Year	Country	Downloads				
2012		10K	30K	100K	300K	1M

News

Year	Country	Downloads				
2005-2008		10K	30K	100K	300K	1M
2009		10K	30K	100K	300K	1M
2010		10K	30K	100K	300K	1M

Newscrawl

Year	Country	Downloads				
2014		10K	30K	100K	300K	1M

Web

Year	Country	Downloads				
2002		10K	30K	100K	300K	1M
2013	Haiti	10K	30K	100K	300K	1M

Figure 2.2: Screenshot of the Leipzig interface for French datasets.

2.2.6 An Crúbadán

An Crúbadán - Corpus Building for Minority Languages is a project aiming at the creation of text corpora for under-resourced languages by crawling the web [14]. They have collected 3'220'781 Swiss German words from 110 URLs of the Swiss German Wikipedia. The format of the dataset is a zip file containing four text files listing the words, the word bigrams, the character trigrams and the URLs used. No full sentences are readily available.

2.3 Summary

In this chapter, we have introduced Swiss German and the main difficulties it introduces for automatic natural language processing. We have then described the different corpora available for Swiss German.

3 Quickstart Dataset

Chapter's content

3.1 What do we need ? 9
3.2 Extracting sentences from available corpora 10
3.3 Final dataset 12
3.4 Summary 14

In this chapter, we describe the creation of a "quickstart dataset", i.e. a collection of labeled data that can be used to create Machine Learning (ML) models for language identification.

3.1 What do we need ?

Our dataset should be representative of the data we will have to categorize: sentences from the web, possibly written in French, German, Italian, English and Swiss German.

3.1.1 Granularity of the data

The gathering of Swiss German samples can be done at various granularity levels: words, sentences, paragraphs. Words don't convey information about context, thus limiting the knowledge we can derive from it. Paragraphs are ideal, since they convey a lot of information and can be split into sentences if needed. However, the difficulties are (a) we don't have paragraphs at our disposal in existing corpora, and (b) adding the constraint of finding new paragraphs makes the data gathering task more difficult. For those reasons, we chose the sentence as the unit of work.

3.1.2 Usage of the dataset

According to *ethonologue.com*[1], there are currently more than 7'000 living languages in the world. Moreover, the way people express themselves in writing is highly dependent on the context: one writes differently in an SMS, an academic report or a love letter. Creating a language identifier working in any context and for any language is beyond the scope of this project.

To reduce the scope, we make the following assumptions:

[1] https://www.ethnologue.com/

1. The Web is the largest source of text and our best shot in finding new Swiss German sentences;
2. Swiss German is likely to be found in multilingual contexts;
3. Swiss German speakers often mix Swiss German and German when writing;
4. English is omnipotent on the Web.

Assumption 1 suggests that the sentences we will have to categorize come from two contexts: formal (blogs, online Wikipedia, ...) and informal (forums, social networks). Assumptions 2 to 4 suggest that our language identifier should at least be able to discriminate between the four national languages, English and Swiss German.

3.2 Extracting sentences from available corpora

From the corpora presented in Section 2.2, only the SB-CH and the Leipzig Collection make sentences in text format directly available. This section discusses if and how sentences could be extracted from NOAH, ArchiMob and the Swiss SMS corpus.

3.2.1 Sentences from the NOAH corpus

The NOAH corpus contains five different XML files using the following format: a `<document>` contains multiple `<article>` composed of sentences `<s>` broken into words `<w>`. Part-of-Speech (POS) tags let us know the syntactic role of each word. A POS-tag beginning with `$` identifies punctuation, such as dots or commas. Listing 2 shows an extract of such document.

```
<document dialect="various" title="?">
    <article n="a0" dialect="Aarauerdütsch" title="wiki aarau">
        <s n="a0-s0">
            <w n="a0-s0-w0" pos="APPR">Mit</w>
            <w n="a0-s0-w1" pos="ART">de</w>
            <w n="a0-s0-w2" pos="NN">Eroberig</w>
            <w n="a0-s0-w3" pos="APPRART">vom</w>
            <!-- ... -->
            <w n="a0-s0-w28" pos="VVPP">schtattgfunde</w>
            <w n="a0-s0-w29" pos="$.">.</w>
        </s>
        <s n="a0-s1">
            <w n="a0-s1-w0" pos="ART">d</w>
            <!-- ... -->
            <w n="a4-s202-w14" pos="$.">.</w>
        </s>
        <s n="a4-s203"/>
    </article>
</document>
```

Listing 2: Extract of an XML file from the NOAH corpus.

To recreate the sentences, we wrote a script based on the pseudo-code presented in Listing 3. This lets us extract 7'430 sentences. After filtering duplicates and sentences without at least one letter, 7'140 sentences remained.

```
doc = document.root
sentences = []
for article in doc:
    for sentence in article:
        s = ""
        for word in sentence:
            s = s + word
            if not word.pos.startswith("$"):
                s = s + " "
        s.trim()
        if s is not empty:
            sentences.append(s)
return sentences
```

Listing 3: Pseudo-code: recreation of sentences from the NOAH corpus.

3.2.2 Sentences from the ArchiMob corpus

The ArchiMob corpus is composed of 43 transcripts of interviews. Each XML file has a <text> element whose <body> is broken into utterances <u>. An utterance can contain both words <w> and non-linguistic units such as <vocal>, <pause>, <gap> etc. Finally, part of speech that were unintelligible for some reason are enclosed in <unclear> tags (see Listing 4).

This structure makes the recreation of sentences difficult, because we lack information on where they should start and end. Indeed, by using utterances as delimiter, we end up with samples like "*uund und so isch das*". On the other hand, treating each document as a sentence clearly misses the point. After some trials, we thus decided not to use the ArchiMob corpus as a possible seed.

3.2.3 Sentences from the Swiss SMS corpus

The data from the Swiss SMS corpus only available through online search engines requiring login.

To get all sentences from a language, we can select the targeted language and leave the query field empty. Results are then displayed in an HTML table paginated using maximum 200 results per page. Since the corpus contains more than 10'000 sentences, or 54 pages, we found it easier to automate the export using a Python script. Listing 5 shows the different options available.

By default, the script looks at the language tag and downloads only SMS without any *borrowings*. From the Swiss SMS corpus documentation:

> Each SMS was tagged for the languages contained within. There are three possible tags:
>
> — Main language: For each SMS, a main language was defined as the dominant language, i.e. the language which provides most words to the SMS.
>
> — Borrowing: Words from a language other than the main language. The words in the foreign language, however, have to be an established part of the main language's vocabulary.
>
> — Nonce Borrowing: a word from another language than the main language of the utterance, which has not become an established part of this language (yet).

```
<u start="media_pointers#d1007-T25" xml:id="d1007-u14" who="person_db#EJos1007">
    <w normalised="und" tag="KON" xml:id="d1007-u14-w1">und</w>
    <vocal>
        <desc xml:id="d1007-u14-w2">eh</desc>
    </vocal>
</u>
<!-- ... -->
<u start="media_pointers#d1007-T39" xml:id="d1007-u22" who="person_db#EJos1007">
    <w normalised="und" tag="KON" xml:id="d1007-u22-w1">und</w>
    <w normalised="das" tag="PDS" xml:id="d1007-u22-w2">das</w>
    <unclear>
        <w normalised="haben" tag="VAFIN" xml:id="d1007-u22-w3">häm</w>
        <w normalised="wir" tag="PPER" xml:id="d1007-u22-w4">mir</w>
    </unclear>
    <w normalised="versprochen" tag="VVPP" xml:id="d1007-u22-w5">verschproche</w>
    <w normalised="und" tag="KON" xml:id="d1007-u22-w6">und</w>
</u>
<!-- ... -->
<u start="media_pointers#d1007-T57" xml:id="d1007-u31" who="person_db#EJos1007">
    <w normalised="und" tag="KON" xml:id="d1007-u31-w1">uund</w>
    <pause xml:id="d1007-u31-w2"/>
    <w normalised="und" tag="KON" xml:id="d1007-u31-w3">und</w>
    <w normalised="so" tag="ADV" xml:id="d1007-u31-w4">so</w>
    <w normalised="ist" tag="VAFIN" xml:id="d1007-u31-w5">isch</w>
    <w normalised="das" tag="PDS" xml:id="d1007-u31-w6">das</w>
</u>
```

Listing 4: Extract of an XML file from the ArchiMob corpus.

If the language is *any*, the option `-y` can be used to print the language tag at the beginning of each sentence followed by a semicolon.

3.3 Final dataset

3.3.1 Composition

Training/test set Given the Swiss German sentences available (see Table 3.1) and the requirements (see Section 3.1) we decided to use the NOAH corpus as the main source for the training/test set. While scarce and not ideal, they are the more "generic". Indeed, SMS and ChatMania extracts use a very specific language that is both less likely to be used in other online contexts and of poor value for most Swiss German NLP applications.

Source	Count	Avg words	Median words
NOAH	7'145	13.6	11
Swiss SMS	10'692	20.4	18
chatmania (CH-SB)	90'898	7	6

Table 3.1: Available Swiss German sentences

```
Usage: get_sms4science_any.py [OPTIONS]

Download SMS from the sms4science's SMS Navigator into a text file.

Options:
-u, --username TEXT             Your SMS Navigator username.
-p, --password TEXT             Your SMS Navigator password.
-l, --lang [de|fr|en|it|sg|any] The language to download.
-n, --num INTEGER               Maximum number of results. Use -1 for no
limit.
-y, --labels                    Print language labels at the beginning of
each sentence.
-m, --multi                     Take any SMS, not only monolingual ones.
-w, --words INTEGER             Min number of words for an SMS to be
retained.
-o, --output TEXT               The output file.
--help                          Show this message and exit.
```

Listing 5: Usage of the Swiss SMS corpus export script.

To have a balanced corpus, we need about 7'000 sentences in French, German, Italian and English that are similar in their content to the ones found in the NOAH corpus. To simplify the work, we decided to use random sentences from the Leipzig collection. The advantage is that most of the parsing and cleaning work has already been done. As the Leipzig collection offers multiple datasets for each language, we decided to use data extracted from random Wikipedia articles between 2010 and 2016.

Validation set While not ideal for training, SMS can still be used for validation purpose. We thus created two datasets based on SMS. `SMS-SG` contains all Swiss German SMS available and can help assert the recall capabilities of a model. `SMS-any` contains SMS annotated with language information. The latter is too different from the data used for training, so there is a high probability that a model trained with NOAH and Leipzig sentences will perform poorly. Anyway, it can still give insights on how a model behaves with edge cases.

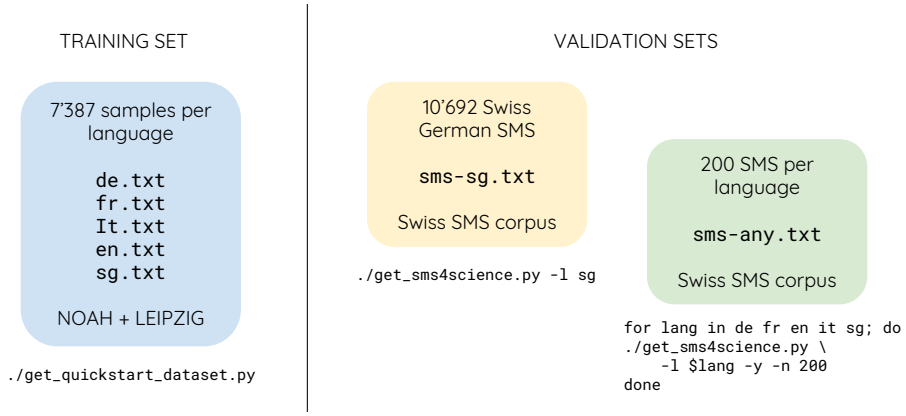


Figure 3.1: Overview of the datasets: contents, files and scripts used to create them.

3.3.2 Quality

This dataset is not ideal in many ways. First, it is very small, especially for advanced machine learning techniques such as neural networks. Second, the samples are very similar between lan-

guages, except for Swiss German. This is true both for the kind of content and the structure of the sentences. For example, Swiss German sentences are shorter in average (see Figure 3.2) and contain significantly more non-letter characters. Those biases are problematic, because a model can learn to generalize using those clues instead of the true characteristics of a language.

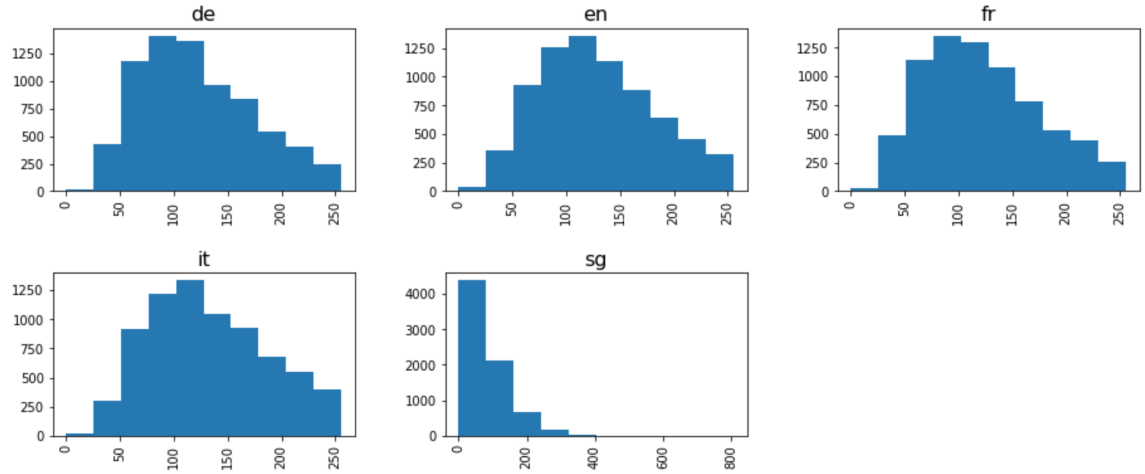


Figure 3.2: Number of characters per sentence for each language included in the training set.

3.4 Summary

In this section, we explained the creation of a quickstart dataset. The training data consists of 36'935 sentences in five different languages coming from the NOAH and the Leipzig corpora. Validation sets are based on SMS from the Swiss SMS corpus. Differences between the samples for each language makes the corpus sub-ideal. However, this dataset is meant to serve as a starting point only. After a first iteration, we hope to have discovered new sentences to form a stronger training set.

4 Language identification: theory

Chapter's content

4.1 Introduction 15
4.2 LID as a supervised classification task 16
4.3 Summary and selected approaches 20

In this chapter, we describe the creation of a LID - *Language IDentification* - model. From the requirements, this model should use Machine Learning algorithms and discriminate Swiss German out of four other languages: French, Italian, German and English.

4.1 Introduction

The general task of Language Identification (LID) has been around for more than two decades. As a result, many different approaches have been proposed.

For a long time, studies have focused their efforts on long monolingual documents from major languages of the world such as English and the main European languages. In this context, simple approaches such as *n-gram-based rank ordering* introduced by Cavnar and Tankle [15] (see Section 4.2.1) or *Naive Bayes* (see Section 4.2.4) are able to attain nearly perfect accuracies, prompting some researchers to label the task "a solved problem" [16]. Those techniques are still used by some of the most popular libraries for automatic language identification such as langid [17] or LangDetect [18].

However, those models are based on assumptions that do not hold anymore against real life data. First, the development of micro-blogging platforms and social media such as Twitter raised the need for models able to perform well on very small text samples. Second, texts found online often mix different languages, making the assumption of monolinguality obsolete. Third, while most models perform well on "usual" languages, there is still progress to be made in order to support under-resourced languages or languages based on similar phylogenetic families [19] [20].

Nowadays, LID is usually formulated as a special case of text categorization. In this perspective, LID becomes a supervised classification task to which generic machine learning techniques can be applied. The set of features used remain mostly variations of N-grams, but the range of classifiers applied has broadened to include more complex models and pattern-recognition algorithms, including support vector machines (SVMs) [21], decision trees [22], logistic regressions [23] [24] and neural networks [25].

4.2 LID as a supervised classification task

In this project, we consider language identification as a multi-class supervised ML classification task using N-grams as feature set. This section describes briefly how N-grams work, then review the questions, techniques and choices to make in each of the common steps involved in creating a machine learning model, namely *preprocessing*, *feature extraction* and *classification*.

4.2.1 Character N-grams

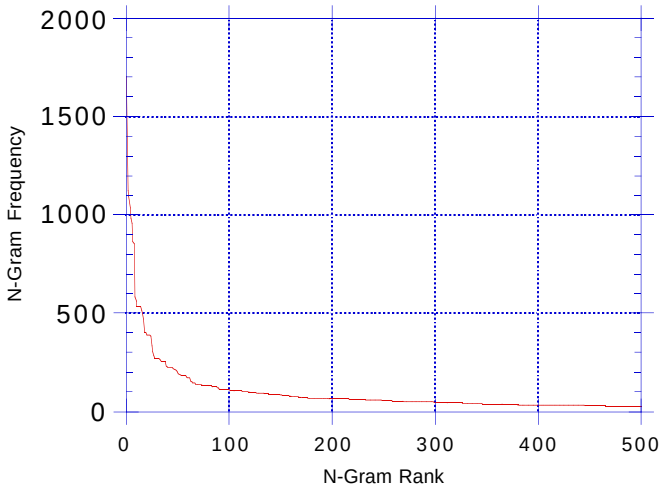


Figure 4.1: N-Gram frequencies by rank in a technical document [15]

The use of character N-grams as features set for language identification has been introduced by Cavnar and Tankle [15]. An *N-gram* is "*a contiguous sequence of n items from a given sample of text or speech*" (Wikipedia). For example, the sentence "Hello World" contains 12 bigrams, 11 trigrams, 9 4-grams, etc. as illustrated below:

```
Hello World

    bigrams:    _h he el ll lo o_ _w wo or rl ld d_
    trigrams:   _he hel ell llo lo_ o_w _wo wor orl rld ld_
    4-grams:    _ell ello llo_ lo_w o_wo _wor worl orld rld_
```

In text identification, N-grams are especially interesting because of what we commonly know as the Zipf's law [26]:

> The r*th* most common word in a human language text occurs with a frequency inversely proportional to r.

The implication is that for any given language, there is a set of words that clearly dominates in terms of frequency. Furthermore, there is a smooth continuum of dominance from most frequent to least. Of course, this also holds for N-grams, as illustrated in Figure 4.1. As Cavnar and Trenkle summarized [15]:

> Zipf's Law implies that classifying documents with N-gram frequency statistics will not be very sensitive to cutting off the distributions at a particular rank. It also implies that if we are comparing documents from the same N-gram frequency distributions.

But, why using N-grams and not words ? First, N-grams are easier to generate, as one needs only to split text at regular intervals. On the contrary, detecting the boundaries of words is hard and yields a lot of edge cases. Second, the number of possible N-grams is always smaller than the number of possible words. In other words, N-grams split sentences into smaller chunks of fixed size, resulting in a feature space that is greatly reduced and very easy to generate.

4.2.2 Preprocessing

Data preprocessing is the transformation of raw data into a format more suitable for feature extraction. In N-gram-based approaches, possible preprocessing steps include sanitization and tokenisation.

Sanitization The question of what characters should be kept or discarded before generating N-grams has no straight answer. For example, Cavnar and Trenkle [15] kept only letters and apostrophes, discarding other punctuation marks and digits. On the other hand, `langid` [17] doesn't apply any kind of preprocessing, so digits or other special characters may appear in the list of features. Others like [21] used only letters and space.

Tokenization Another question is whether or not N-grams should be limited to or span across word boundaries. In their study, Mathur and al. [27] reported that the latter showed a marginal performance boost, even if it increased the number of possible features and the training time. In this project, we thus generate N-grams across word boundaries, mostly because it simplifies the work: no assumptions have to be made on what a word is and no tokenizer needs to be trained.

4.2.3 Feature extraction

Feature extraction is the transformation of data, in our case sentences, into vectors of characteristics (i.e. N-grams) that can be fed into a classifier.

N-gram length The first question is the length of N-grams to use. Studies have shown that unigrams convey little information. The performance increases as n increases, until saturation around $n = 8$ [27]. In the literature, both fixed size and variable size N-grams have been used: Cavnar and Trenkle use N-grams with $1 \leq n \leq 5$, `langid` uses uni-, bi- and trigrams [17], Grefenstette [28] and Botha [21] use fixed-size trigrams, Devenhage 5-grams [29]. But this is also dependent on the languages and the kind of data we have. For smaller sentences, smaller N-grams seem more relevant.

Number of features Another question is the number of N-grams to keep as features. Indeed, by using only ASCII letters and spaces, the number of possible trigrams is only $(26+1)^3 = 729$, so no selection has to be made. But this number explodes when we consider language-dependent diacritics and longer N-grams. For example, French contains 16 diacritics (excluding special ligatures); with $n = 3$, we have $(26 + 16 + 1)^3 = 79'507$ possibilities and $3'418'801$ for $n = 4$...

In general, we want to limit the size of the feature vector to a minimum, both for performance and time/memory considerations. Zipf's law also tends to show that few N-grams are sufficient to create a signature of a language (see Figure 4.1). Indeed, Cavnar and Trenkle kept only the 400 more frequent N-grams for each language profile, giving a total of $14 \times 400 = 5'600$ features and `langid` uses 7'480 features for their pre-trained model to support 91 languages.

Weighting scheme A final area of discussion is the ponderation, or the *weight*, of each feature in our vector space. The simplest approach is to use raw frequencies [15]. However, in case we use only one feature set for each language, raw frequencies can give too much importance to common N-grams found in all language, thus having poor discriminative power.

In text categorization, a common weighting factor is the *TF-IDF*, *term frequency - inverse document frequency*, with a logarithmic scaled frequency.

The basic idea behind *logarithmic TF* (see equation 4.1) is that the importance of a word is not simply proportional to its frequency. Using log frequencies for N-grams features has shown to yield drastic improvements and is at the basis of the LIGA algorithm [19].

$$tf(t,d) = log(1 + f_{t,d}) \tag{4.1}$$

The *inverse document frequency* is mostly used in the area of *information retrieval* and, to our knowledge, has rarely been used in LID. The basic idea is that the rarer a term appears in a corpus, the more information it provides (see equation 4.2).

$$idf(t,d) = log\Big(\frac{N}{1 + |\{d \in D : t \in d\}|}\Big) \tag{4.2}$$

4.2.4 Classification

As mentioned in Section 4.1, a number of classifiers have been applied for LID. Given the time at our disposal, it would be impossible to describe them all. We discuss here some of the most common approaches.

Naive Bayes

Naive Bayesian classifiers (NB) are probabilistic models based on the *Bayes' theorem*. The term *naive* is used when we assume a complete independence between the features. Many studies now use this kind of classifier as a baseline. It is very easy to apply, scales very well (the number of parameters increases linearly with the number of features) and often yields decent results.

The **Bayes' theorem** is stated as:

$$P(L_i|X) = \frac{P(L_i) \cdot P(X|L_i)}{P(X))}, posterior = \frac{prior \times likelihood}{evidence} \tag{4.3}$$

Applied to language identification, L_i is a language model, X a given n-gram and $P(X|L_i)$ the probability to observe the n-gram X given a language L_i. The prior $P(L_i)$ denotes the initial

degree of belief that the n-gram belongs to a certain model. In many cases, no prior knowledge is assumed and it is set to $1/\#languages$.

Equation 4.3 let us compute the probability of one single n-gram to belong to a certain language. Now, to get the probability of a whole text to belong to a language, given the assumption of independence, we can multiply the probabilities:

$$P(L_i|T) \approx \prod_{j=1}^{J} P(L_i|X_j) \tag{4.4}$$

In other words, the probability of a text T to be in language L_i is the product of the probabilities of each of its N-grams to be in that language.

A **Bayesian classifier** uses the equation 4.4 to compute probabilities for each category and then applies a *decision rule* to make a final decision. A common rule, known as *MAP - maximum a posteriori*, is to choose the category with the highest probability:

$$L_i = \underset{i}{argmax} P(L_i|T) = \underset{i}{argmax} \prod_{j}^{x} P(L_i|X_j) \tag{4.5}$$

The bayesian classifier is a *generative* model, meaning that we need to generate language models and infer likelihoods prior to using the model. On the other hand, this also means that to add support for new languages, the classifier itself doesn't need to be recreated or changed.

In our case, language models are constituted using the most common N-grams found for each language in the training set. One question is thus: what do we do when we encounter new N-grams after the training ? There are different solutions to this, but the simplest one is to ignore it, thus giving this N-grams a probability of appearance of $P(L_i|X_{unknown}) = 1$.

The bayesian classifier is thus very interesting for LID, thanks to its simplicity, performance and flexibility.

Logistic regression

Like Naive Bayesian, logistic regression (LR) can estimate probabilities. Unlike NB, it is a *discriminative* model: LR "learns" directly from a feature set common to each language. As a result, there is no need for priors or multiple language models, but the classifier must be retrained in case we add or remove a language.

The core of logistic regression is the assumption that the feature space can be nicely separated into regions, one for each category. In *linear logistic regression*, the boundaries are drawn using a linear function (e.g. a straight line in case 2-dimensional space, a plane in 3-dimensional space, etc.), while other kernels use more complicated boundaries.

Once regions have been discovered, generating posterior probabilities becomes easy: samples clearly lying inside a region have a high probability of being part of the given category, while samples falling at the boundary of multiple regions will have lower probabilities. The final decision can use the same mechanism as the Naive Bayes, i.e. select the category with the highest probability.

The real challenge of logistic regression is to find the boundaries, i.e. to split the feature space into regions. In practice, this is done in an iterative manner using an algorithm known as *gradient descent*. Put in simple words, the idea is to draw boundaries at random, then iteratively moving them in order to minimize the errors in prediction.

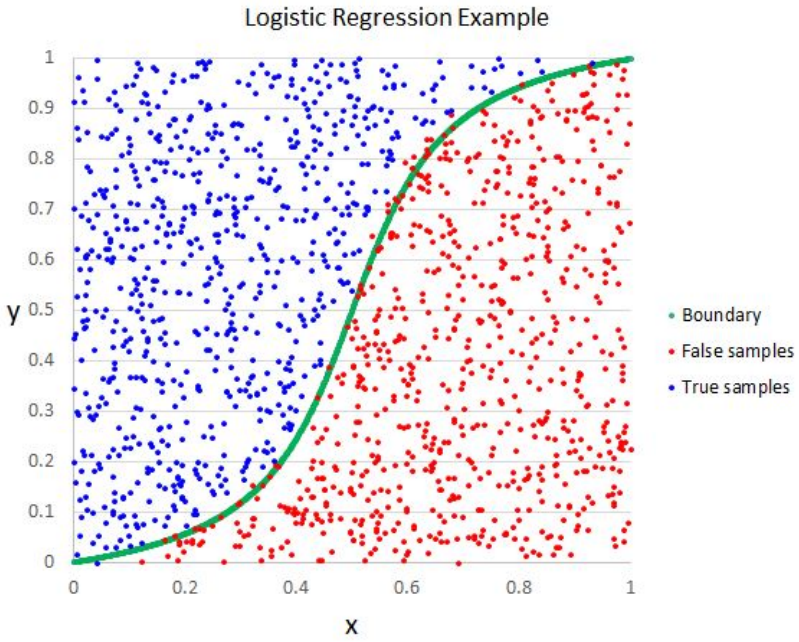


Figure 4.2: Example of logistic regression applied to two classes using two independent features, x and y [1].

Support Vector Machines

Support Vector Machines (SVM) is a classification algorithm that seeks to find the best hyperplane to divide the feature space into classes [30] [31]. As such, the core idea is similar to logistic regression, but the way boundaries are shaped and generated is very different. However, the details of how SVMs work are too complex to be summarized here.

Compared to LR, SVMs have some advantages. First, they can work well even if the data are not separable in the base feature space. Second, they yield strong theoretical guarantees and are less subject to *overfitting*, as they are based on mathematical transformations only (vs. an iterative process). On the other hand, SVMs are memory-intensive, harder to tune and the results may be difficult to interpret.

4.3 Summary and selected approaches

Table 4.1 presents the questions, variations and models that we found the most relevant to implement for a first language identification model using character-based N-grams.

[1] Source: https://helloacm.com/a-short-introduction-logistic-regression-algorithm/.

Preprocessing	N-grams generation on the raw sentences. N-grams generation after keeping only letters and spaces.
Feature extraction	Fixed-size N-grams ($3 \leq n \leq 7$) and variable-size N-grams. Number of features: from 1'000 to 10'000 Features extracted from Swiss German samples only. Features extracted from all samples. Feature weights using frequencies, log-frequencies, TF-IDF.
Classification	Naive Bayes (baseline). Logistic Regression. SVMs. Neural Networks (as a bonus).

Table 4.1: Summary of the approaches/variations selected for implementation.

5 Language identification: implementation

Chapter's content

5.1 Technologies 22
5.2 Timeline 23
5.3 Structure of the code 23
5.4 Data handling 24
5.5 Naive implementations 25
5.6 Sklearn 26
5.7 Neural Networks 28
5.8 A complete example 29
5.9 Summary 29

5.1 Technologies

The language models have been developed in Python 3 using Jupyter Notebooks. To simplify the installation and the management of the various dependencies, we used Anaconda, a Python distribution especially made for data science.

Aside from neural networks, all the models have used classes from scikit-learn, a simple yet efficient machine learning library built on NumPy and SciPy. Graphs and visualizations have been created using matplotlib and, in some cases, seaborn.

The neural networks experiment has been created using Keras, a high-level neural networks API, using TensorFlow as a backend engine.

Software	version
Python	3.6.5
Anaconda	5.1.0
Keras	2.1.5
TensorFlow	1.7.0

Table 5.1: Versions of the major tools used.

5.2 Timeline

The implementation phase has been organized into short *sprints* described in Table 5.2.

Sprint name	Content
Getting Started	⋆ implementation of a Naive Bayes classifier from scratch ⋆ discovery of the tools offered by scikit-learn
Scikit-Learn 101	⋆ discovery of the tools offered by scikit-learn ⋆ first models using simple examples
Utilities	⋆ implementation of a `sanitization` function ⋆ implementation of a "garbage model" using binary classes ⋆ creation of reusable scripts for data loading, visualization, etc.
Vectorizer	⋆ understanding the `TfidfVectorizer` ⋆ grid search, selection of the best parameters
Classifiers	⋆ understanding the scikit-learn classes for logistic regression and SVM ⋆ grid search to fine-tune the classifiers
Neural Networks	⋆ study of deep neural networks in an LID perspective ⋆ setup of keras / tensorflow ⋆ various attempts
Wrapping Up	⋆ comparison of all the results gathered so far ⋆ export of the best models found

Table 5.2: Agile *sprints* during LID implementation

5.3 Structure of the code

The code is organized into folders:

— `data` contains text files created using scripts presented in Chapter 3.
— `langid` is a Python package containing custom classes and reusable code such as the `NaiveIdentifier` and the sanitization functions.
— `Notebooks` contains all the notebooks and scripts specific to the notebook environment.

Jupyter notebooks are very flexible, making it easy to import usual Python packages and/or to run files containing Python snippets. As a rule, each notebook imports the utilities it needs at the top, then redefines the font size of plots (impossible to do from a script). An example is shown in Listing 6.

```
# run scripts specific to the notebook environment
%run notebook_utils.py
%run gridsearch_utils.py

# load python package
cd ..
from langid import np_sanitize
cd -

# set big font in plots
import matplotlib
SMALL_SIZE = 20
matplotlib.rc('font', size=SMALL_SIZE)
matplotlib.rc('axes', titlesize=SMALL_SIZE)

# other imports
from sklearn.pipeline import Pipeline
```

Listing 6: First cells in a notebook used to import packages and setup the environment.

5.4 Data handling

5.4.1 Sets

The data used for cross-validation are the ones from NOAH and Leipzig (see Chapter 3). They are divided randomly into a train and a test using 80% and 20% of the samples respectively:

```
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(
    X, y, test_size=.2, random_state=0)
```

Other datasets are used for validation: (1) the Swiss German SMS tests recall, (2) a sample of SMS gives some insight on how our model performs with an especially hard data and (3) a sample of data taken from the Leipzig corpus serves as our main validation set. Note that the latter was added afterwards because of the late discovery of this resource.

The script `notebook_utils.py` defines functions to load the sets and to display performance metrics.

5.4.2 Sanitization

The selected sanitization approach is to remove all characters except letters and spaces. In practice, a first *regex* is used to replace all non-letters by spaces. A second one is then applied to normalize spaces, i.e. to replace a sequence of spaces by a single character. This is shown in Listing 7.

```
import re
import numpy as np

reg_nonletters = re.compile("[^\w ]|\d|_")

def remove_nonletters(text: str) -> str: return re.sub(reg_nonletters, " ", text)
def remove_manyspaces(text: str) -> str: return re.sub(r'\s+', ' ', text)

def sanitize(text: str) -> str:
    text = remove_nonletters(text)
    text = remove_manyspaces(text)
    return text.strip()

np_sanitize = np.vectorize(sanitize)
```

Listing 7: Implementation of the sanitization function.

```
tfidfvectorizer_arguments = dict(analyzer='char')

vec = WrappedVectorizer(
    sanitizer=np_sanitize,
    sg_only=True,
    **tfidfvectorizer_arguments);

vec.fit(X_train, y_train);
```

Listing 8: An example on how to create and train the `WrappedVectorizer`.

5.4.3 WrappedVectorizer

The `WrappedVectorizer` encapsulates an sklearn's `TfidfVectorizer` and adds two options: `sanitizer` and `sg_only`.

`sanitizer` is an optional sanitization function to be applied, while `sg_only` determines how the vectorizer is trained: if set to `True`, only Swiss German samples will be passed to the vectorizer's `fit` method, meaning that only Swiss German N-grams will be used as features. If set to `False`, the vectorizer is trained on all samples.

5.5 Naive implementations

5.5.1 Naive Vectorizer

To understand how feature extraction works, we began by creating our own *vectorizer* class, `NaiveVectorizer`.

During training, the `NaiveVectorizer` should "learn" the most common N-grams in order to create a *vocabulary* of length l that is specified in the constructor. The method used is the following:

1. For each sentence S in the dataset, extract the N-grams;

2. Count the frequency f of each N-gram;
3. Sort the N-grams by frequency f, from most common to least;
4. Keep the l most common N-grams as *vocabulary*.

Once the vocabulary is known, we have everything to transform a sentence into a feature vector. Note that in our implementation, we decided to use *log-frequencies* and to normalize the feature vectors. This is especially useful as the length of the sentences vary a lot between samples. Those steps are summarized below:

1. Initialize a vector v with the size of the vocabulary;
2. For each N-gram in the vocabulary, count how often it appears in the sentence s and store it into v: $v[i] = f_i$;
3. Use log-frequencies: $v = 1 + log(v)$;
4. Normalize the features to account for the length of the sentence s: $v = \frac{v}{|v|}$;
5. Return v.

The final code for the `NaiveVectorizer` is about 100 lines long and has the same behavior as scikit-learn's `TfidfVectorizer` using `sublinear_tf=True, use_idf=False`. Thanks to NumPy's `vectorize` method, the performance is also quite similar[1].

In our implementation, we also added an option, `ignore_non_words`, that exclude from the vocabulary any N-grams that doesn't contain at least one letter. This is especially useful when we work on raw sentences. This way, we ensure that years, dates or smileys won't be used to predict a language. However, this type of trick doesn't replace proper sanitization on sentences.

5.5.2 Naive Identifier

`NaiveIdentifier` is a simple bayesian classifier that implements the naive bayes approach presented in Section 4.2.4. During training (the `fit` method), we create and train one vectorizer per language. These will be our language models. The vectorizer class to use can be specified as a parameter to the constructor.

To classify a new sentence, we transform it using each of the vectorizers, sum the resultant vectors and keep the one with the highest sum. In a way, this is equivalent to looking at a sentence and counting the number of French words, English words, etc. we know of. If we recognize only 1 French word but 10 English words, English is probably the correct language. The complete code is available in Listing 9.

5.6 Sklearn

sklearn contains classes for all of the most popular machine learning algorithms, as well as many utilities to handle datasets, run benchmarks and display results.

The principal classes used are listed below:

— CountVectorizer: a vectorizer using raw frequencies;

[1] In many circumstances, our implementation is even a bit faster than scikit-learn's.

```python
class NaiveIdentifier:

    def __init__(self, klass=TfidfVectorizer, **vectorizer_options):
        """
        Create a naive identifier.
        :param klass: the vectorizer class to use.
        :param vectorizer_options: the vectorizer's init arguments.
        """
        # ... memorize the arguments for later ...

    def fit(self, X_train: List[int], y_train: List[int]):
        """
        Create and fit one vectorizer per class.
        :param X_train: the training set
        :param y_train: the class of each sample in the training set
        """
        self.vectorizers_ = []
        X_train, y_train = np.array(X_train), np.array(y_train)

        for i in range(np.unique(y_train).size):
            # train one vectorizer using samples of class i
            v = # ... instantiate a vectorizer ... #
            v.fit(X_train[y_train == i])
            self.vectorizers_.append(v)

    def predict(self, X: List[int]) -> csr_matrix:
        """
        :param X: the dataset
        :return: the most probable class as a column vector
        """
        # transform X and sum the resulting vector using each available vectorizer
        mat = np.hstack([v.transform(X).sum(axis=1) for v in self.vectorizers_])
        # keep the highest one
        return mat.argmax(axis=1).A1
```

Listing 9: NaiveIdentifier: implementation

— TfidfVectorizer: similar to our `NaiveVectorizer`, but with more options and the ability to apply IDF as well. It uses `CountVectorizer` under the hood;
— LogisticRegression: implementation of a regularized logistic regression using a multiclass *one-versus-all* approach by default;
— SCV: an SVM implementation using a multiclass *one-vs-one* scheme by default;
— MultinomialNB: a Naive Bayes classifier for multinomial models.

There are also a couple of utility classes that we judge worth mentioning:

— Pipeline: in general, a model sequentially applies a list of transforms and a final estimator. This "pipeline" can be encapsulated into an sklearn `Pipeline` object, so the training and testing of a model can be done quickly as exhibited in Listing 11.
— GridSearchCV and LogisticRegressionCV: simplify *hyperparameters optimization* using exhaustive searching through a manually specified subset of parameters and values. By default, both use the 3-fold cross validation scheme and a scoring system based on accuracy. See Listing 10 for an example of usage.

```
# The parameters to test by cross-validation
svc_params = [
    {'kernel': ['rbf'],
        'gamma': [1e-3, 1e-4],
        'C': [1, 10, 100, 1000]
    },
    {'kernel': ['linear'],
        'C': [1, 10, 100, 1000]
    }
]

# instantiate a grid search
gsvm = GridSearchCV(
    SVC(C=1, max_iter=1000), # C and max_iter are fixed
    svc_params,              # the grid
    n_jobs=-1, verbose=1)

# run the tests
gsvm.fit(Xvec, y)

# get the best model
best = gsvm.best_estimator_
```

Listing 10: Hyperparameter optimization of an SVM model using the `GridSearchCV` utility class.

```
# step 1:
# create and train the vectorizer
vec = TfidfVectorizer(**params)
X_fit = vec.fit_transform(X_train,
↪  y_train)

# step 2:
# create and train the classifier
clf = LogisticRegression()
clf.fit(X_fit, y_train)

# predict
X_fit = vec.transform(X_test)
y_predicted = clf.predict(X_fit)
```

```
# create a pipeline:
# the n-1th steps must define the
# method 'transform',
# the nth step must define 'predict'
pipe = Pipeline(steps=[
('vec', TfidfVectorizer(**params)),
('clf', LogisticRegression())
])

# train
pipe.fit(X_train, y_train)

# predict
y_predicted = pipe.predict(X_test)
```

Listing 11: Training and testing of a model, without and with the `Pipeline` utility.

5.7 Neural Networks

The neural networks we implemented use the same kind of features as the other models. They are composed of one or two hidden layers with dense connections and *rectified linear units - relu* - activation functions. The output layer uses a *sigmoid* activation as inspired by [25]. A *stochastic gradient descent* is used for training.

Using Keras, creating a neural network can be done in a few lines. For example, the code presented in Listing 12 trains and evaluates a three-layer NN with a hidden layer of 500 neurons.

```
## == create a model using keras == ##
hidden_size, output_size = 500, len(langs)

model = Sequential()
model.add(Dense(hidden_size, input_dim=num_features, activation='relu'))
model.add(Dense(output_size, activation='sigmoid'))
model.add(Activation("softmax"))

## == train the model == ##
model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=SGD(lr=0.01), metrics=["accuracy"])
model.fit(
    vectorizer.transform(X_train), y_train,
    epochs=50, batch_size=128, verbose=1)

## == evaluate the model == ##
(loss, accuracy) = model.evaluate(
    vectorizer.transform(X_test), y_test, batch_size=128, verbose=1)
```

Listing 12: Definition and training of a 3-layer Neural Network using Keras.

5.8 A complete example

To quote Linus Torvalds, "*talk is cheap, show me the code*". The code presented in Listing 13 shows a typical notebook: imports, data loading, pipeline creation, training and evaluation.

5.9 Summary

In this section, we presented the libraries and codes used to create and test machine learning models for language identification. The performances of those models are discussed in Chapter 6.

```
# import our utility functions
%run notebook_utils.py

# various imports
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

# load the data
X_train, X_test, y_train, y_test = load_split_data()
X_any, y_any = load_sms_any()
X_valid, y_valid = load_validation_data()

# train and fit our classifier
pipe = Pipeline([
    ('vect', TfidfVectorizer(analyzer='char',
                ngram_range=(3,3), max_features=5000)),
    ('clf', LogisticRegression())
])
pipe.fit(X_train, y_train)

# print the results

print("TEST SET")
print("========")
y_pred = pipe.predict(X_test)
print_results(y_test, y_pred)
plot_confusion_matrix(y_test, y_pred, normalised=True)

print("SMS RECALL")
print("==========")
test_recall_with_sms(pipe)

print("\n\nVALIDATION SET")
print("==============")
y_valid_pred = pipe.predict(X_valid)
print_results(y_valid, y_valid_pred)
plot_confusion_matrix(y_valid, y_valid_pred, normalised=True)

# export the model
import pickle
with open('model.pickle', 'wb') as f:
    pickle.save(pipe, f)
```

Listing 13: A full notebook: create, train and evaluate a pipeline.

6 Language identification: results

Chapter's content

6.1 Final models 31
6.2 Results 33
6.3 Tests on real data 34
6.4 Summary 34

6.1 Final models

Table 6.1 describes the features used for each of the tested classification algorithms on the final models. The data have always been sanitized beforehand, keeping only spaces and letters. This choice has been made in order to cope better with noisy data coming from sources like the Web.

Model	Weights	N-grams	Type	#features
LogisticRegression	log-TF, IDF scaling.	trigrams	char.	10'000.
SVM	log-TF, IDF scaling.	trigrams	char.	10'000
Multinomial NB	Raw frequencies.	1-3 grams	char.	10'000
NaiveIdentifier	Raw frequencies.	1-3 grams	words	3'000/lang
NeuralNetwork	Raw frequencies.	1-3 grams	char.	3'000

Table 6.1: Features used for each model.

Each model has been tuned to find the best hyperparameters. For most of them, however, the default yielded the best results. Furthermore, we observed that *hyperparameter tuning* on the classifier had little impact, while little changes in the behavior of the vectorizer could drastically change the overall performance of the models. Thus, we concluded that feature extraction is the most critical step of the pipeline.

6.1.1 Logistic regression

Using the `LogisticRegression` classifier, the accuracy augments with the size of the features. After 6'000 features, the improvements are, however, less significant. More than 10'000 features don't yield any more real improvements. Concerning the type of N-grams, we get very similar

results with N-grams between 1 and 3 and with trigrams. We finally opted for trigrams for the simplicity of fixed-size N-grams. Interestingly, using *tfidf* instead of raw frequencies improved the performance by 0.5% on all the datasets.

6.1.2 SVM

The SVM model behaves similarly to the LogisticRegression concerning the features and yields very similar results, with a very slight improvement on accuracy (less than 0.5% difference). One thing to note is that during training, the SVM model never converged. Since it is also very slow (especially compared to logistic regression), we had to stop training after 1'000 iterations. We don't know if this is due to the data itself or to the sklearn implementation.

6.1.3 MultinomialNB

The multinomial naive bayesian classifier implementation from sklearn yields better results when using raw frequencies. Contrary to the models presented above, the accuracy drops significantly when *tfidf* scaling is applied. As a result, we used the `CountVectorizer` class to vectorize our sentences.

We were surprised to see that this classifier works well with word N-grams as well: the performances are very similar on both the test set and the Leipzig validation set. On SMS, however, the character N-grams have an advantage: the overall accuracy is 95.22% for character-based N-grams, against 91.44% for word N-grams.

6.1.4 Naive identifier

Our simple implementation of a bayesian classifier is surprisingly efficient: the performances are similar to the other models, yet it is simpler in design. Contrary to the `MultinomialNB` from sklearn, the `NaiveIdentifier` works clearly better with word N-grams, as shown in Table 6.2.

	Accuracy			**SG (Valid.)**			**SG SMS err.**	
	Test	SMS	Valid.	Prec.	Recall	F1	Count	%
Characters	98.16	92.64	98.38	99.75	92.61	96.05	460	4.30
Words	95.56	89.55	96.18	97.28	87.72	92.25	1348	12.61

Table 6.2: `NaiveIdentifier`: comparison between word and character N-grams.

6.1.5 Neural Networks

We tested very simple neural network architectures with one to two hidden layers. The best results were obtained using the architecture illustrated in listing 14. The size of the hidden layer is the same as the size of the input, following the tracks of [25]:

> Regarding the number of units in the hidden layers, there are some rules of thumb: use the same number of units in all hidden layers, and use at least the same number of units as the maximum between the number of classes and the number of features. But

there can be up to three times that value. Given the high number of features we opted to keep that same number of units in the hidden layer.

This final network was trained using minibatches of size 128 and 50 epochs. The end results are interesting, but not as good as the results obtained with simpler approaches (see Table 6.3).

The same network was trained both on trigrams using *tfidf* scaling and with N-grams between 1 and 3 using raw frequencies. The results are slightly better using the latter (about 0.5%), except for SMS. In this case, the accuracy is of 88.46% for trigrams against 85.67% for variable N-grams. Since this difference is more significant, we decided to retain the trigrams approach in this report.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 3000)              9003000
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 15005
_________________________________________________________________
activation_1 (Activation)    (None, 5)                 0
=================================================================
Total params: 9,018,005
Trainable params: 9,018,005
Non-trainable params: 0
_________________________________________________________________
```

Listing 14: Final Neural Network architecture

6.2 Results

We were able to reach an accuracy greater than 98% and an F1-score for Swiss German sentences greater than 96% on the validation set for all the algorithms tested except for Neural Networks, were the Swiss German F1-score drops to 93%.

Classifier	Accuracy			SG (Valid.)			SG SMS err.	
	Test	SMS	Valid.	Prec.	Recall	F1	Count	%
LogisticRegression	99.40	85.57	98.55	98.37	94.57	96.43	63	0.59
SVM	99.45	87.06	98.61	99.11	94.07	96.52	85	0.79
MultinomialNB	98.55	95.22	98.29	99.47	93.03	96.14	386	3.61
NaiveIdentifier	98.16	92.64	98.38	99.75	92.61	96.05	460	4.30
Neural Network	98.01	88.46	97.20	96.96	89.21	92.92	389	3.64

Table 6.3: Best results. Accuracy on each dataset; precision, recall and F1-score obtained for Swiss German on the validation set; classification errors on Swiss German SMS only.

As shown in Table 6.3, the variations in accuracy on the test set is $\pm 0.5\%$ and even less on the Leipzig validation set: $\pm 0.3\%$. The performances on the Swiss SMS set, however, are more variable.

The SMS corpus is very challenging, as many SMS contain *slang*, foreign expressions or acronyms. The vocabulary is also quite different from the test set. It seems that classifiers based

on trigrams and *tf-idf* scaling methods are less efficient: both logistic regression and SVMs have an accuracy below 88%. On the other hand, naive bayes classifiers based on raw frequencies and variable character/word N-grams are able to keep an accuracy over 92%.

Another difference is the precision/recall obtained on the Swiss German sentences. Logistic regression and SVM tend to favor recall, while the other classifiers favor precision. In most cases, the errors happen between High German and Swiss German. We suspect this is more due to the kind of features than the classifiers themselves: trigram features with *tf-idf* confuse German for Swiss German more easily, while variable size N-grams using raw frequencies favor the inverse mistake. This tendency becomes clearer when looking at the confusion matrices illustrated in Figure 6.1.

6.3 Tests on real data

Using the webapp described in Chapter 7, we compared the behavior of the classifiers on sentences from several websites. Below are some general conclusions from those experiments.

- Short sentences of less than 10 words are likely to be misclassified. This is especially true for English sentences using many single "I".
- Except for the `NaiveIdentifier`, empty sentences or sentences using an unknown vocabulary are classified as Swiss German[1]. However, the confidence differs between the models: SVMs tend to give strong probabilities, while logistic regression and Multinomial Bayes are less categoric.
- Entities such as titles, names or addresses are problematic. For example, adding the city name "Zürich" to a sentence can completely change the outcome.
- Models trained with a high number of features (10'0000+) are likely to output strong confidence probabilities, even on errors. On the one hand, strong confidence is good for recall. On the other hand, we can no longer rely on 95%+ confidence filters to ensure the correctness of our results. By decreasing the number of features to 6'000 N-grams, we were able to get more contrasted confidence levels.
- After sanitization, sentences might lose their meaning completely and/or seem like Swiss German. This is especially true for sentences containing a lot of digits, dates and symbols.

Table 6.4 shows an extract of problematic sentences. As we can see, each classifier has its strengths and weaknesses. It is interesting to note that the `NaiveIdentifier` is the only one "detecting" unknown languages by returning a zero confidence.

6.4 Summary

This chapter discussed the results obtained using different algorithms for language identification. On validation data, all models scored well, with an accuracy above 97%.

The choice of features is the most significant parameter. Character N-grams with $1 \leq N \leq 3$ or $N = 3$ seem to be the best choice. Surprisingly, word N-grams are also an interesting alternative. Above 6'000 features, the improvements are very small, while using more than 10'000 features can impact negatively the efficiency of the classifier. Indeed, the more the features, the more confident the model when it makes an error.

[1] This can also be due to the biases in the quickstart dataset or the fact that Swiss German is the label with the highest value.

It is very difficult to tell which model is the best. From experiments on SMS and real data, we could see that each of them has its strengths and weaknesses.

Sentence	Label	Naive	NB	LogReg	SVM
don't let me go	en	en 0.33	fr 1.00	sg 0.70	sg 0.62
i'm in pain from missing you	en	en 0.38	en 1.00	sg 0.97	en 0.53
...	?	de 0.00	sg 0.20	sg 0.89	sg 0.99
© Foto: Ana-Marija Bilandzija für ZEIT ONLINE	de	de 0.50	sg 1.00	sg 0.97	sg 0.93
Karl, pk tu lâches pas ton Iphone ?	fr	de 0.33	fr 0.89	sg 0.82	sg 0.76
J'aime beaucoup Zürich au printemps	fr	fr 0.80	sg 1.00	fr 0.60	sg 0.53
для развития дзюдо	?	de 0.00	de 0.48	sg 0.91	sg 0.99
nieprzyjemny zapach ciała drażniący	?	de 0.00	sg 1.00	sg 0.99	sg 1.00

Table 6.4: Example of problematic sentences.

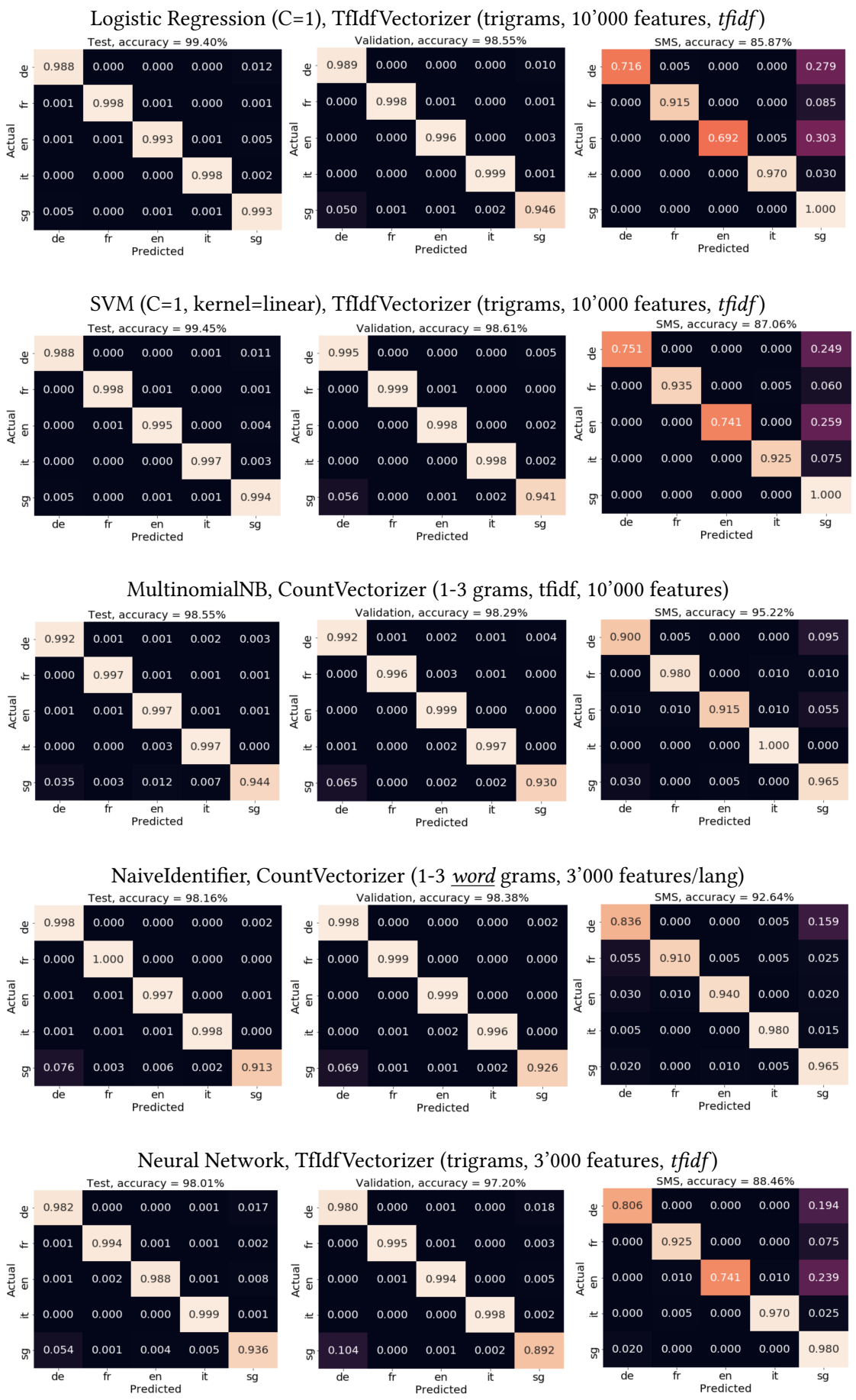


Figure 6.1: Confusion matrices for the training SMS and validation set for each estimator.

7 Data gathering

Chapter's content

7.1 Crawling the Web: a challenge 37
7.2 Boilerpipe 38
7.3 Swiss German Extractor, a first attempt 38
7.4 Summary 40

Having a language identifier ready, the next step is to find new Swiss German sentences. Our hypothesis is that our best pool of unseen data is the Web. In this chapter, we discuss the challenges and the techniques used to extract Swiss German sentences from online websites.

7.1 Crawling the Web: a challenge

Crawling the Web in search of Swiss German sentences has many challenges. The most obvious one is to find websites *actually containing Swiss German*. But aside from this, there is also the more basic challenge of extracting text from Web pages.

Lack of uniformity and fast ever-changing paradigms are the middle names of the Internet. The HTML markup language contains little semantic information and has a long history of misuse and misunderstanding. Even if this becomes less true with the new HTML5 standard, we still have no bullet-proof techniques to detect "real sentences" in a Web page.

Getting hold on the content of a Web page in itself can be problematic. The use of AJAX and interactive Web components, which is more and more pregnant on modern websites, makes the crawling task more difficult: without JavaScript support, we can miss important part of websites[1].

Another common category of problems is server-side errors and/or misconfigurations: a wrong encoding information in the header or a PHP error can make the crawler fail. Other challenges include the ability to find related pages and to scale to millions of webpages.

[1] In older websites, the *Flash* technology was often used, resulting in the same kinds of challenges.

7.2 Boilerpipe

We found a lot of alternatives and solutions for Web crawling. However, most of them are either paid service, slow, limited or heavy/difficult to use. After some digging, we settled on the boilerpipe library, which seemed to offer the best compromise.

Boilerpipe is a free open-source Java library for extracting text content from a Web page. It is fast, light and easy to use. Based on shallow text features, it uses a combination of quantitative linguistics (average word and sentence length, absolute number of words, etc.), local context (e.g. position of text blocks), densitometric information (e.g. text density in HTML blocks) and heuristics to extract relevant text [32].

The library lets us choose the kind of extractor to use based on our specific need. Extractors include `ArticleExtractor`, `ArticleSentenceExtractor`, `KeepEverythingExtractor`, etc. It is available on Maven and weighs less than 135 KB when packaged in a jar.

7.3 Swiss German Extractor, a first attempt

To test the *boilerpipe* library as well as the integration with our language identifiers, we created a simple webapp.

7.3.1 Interfaces

The webapp contains two pages.

A first page lets the user enter a URL and choose the kind of *boilerpipe* extractor to use, the minimum number of words for a sentence to be processed and the kind of language identification model to apply. This is illustrated in Figure 7.1. Results are displayed in a table, with one row per text block found. This row can contain multiple sentences, depending on the extractor. An example is shown in Figure 7.2.

To easily grasp the results, a specific color is assigned to each language and the opacity of the text reflects the probability: the higher, the blacker. Moreover, various filters are available: colors can be hidden using the "Show colors" checkbox, a filter "min proba" lets you hide all sentences with a probability less than a given threshold and the "SG only" checkbox hides all non-Swiss German sentences. Finally, the exact probabilities are shown in a pop-over when the mouse is placed over a sentence.

The second page is similar, but features a textarea instead of a URL field. This is useful to test the capabilities of a model on specific sentences.

7.3.2 Implementation

This webapp is written in Python 3 using the Flask microframework.

As *boilerpipe* is only available in Java, we used JPype to interface Python and Java virtual machines at the native level. A port of *boilerpipe* to Python was already available on GitHub, but it had several bugs and limitations. We refined it and published the forked version on our GitHub: `https://github.com/derlin/boilerpipe3`.

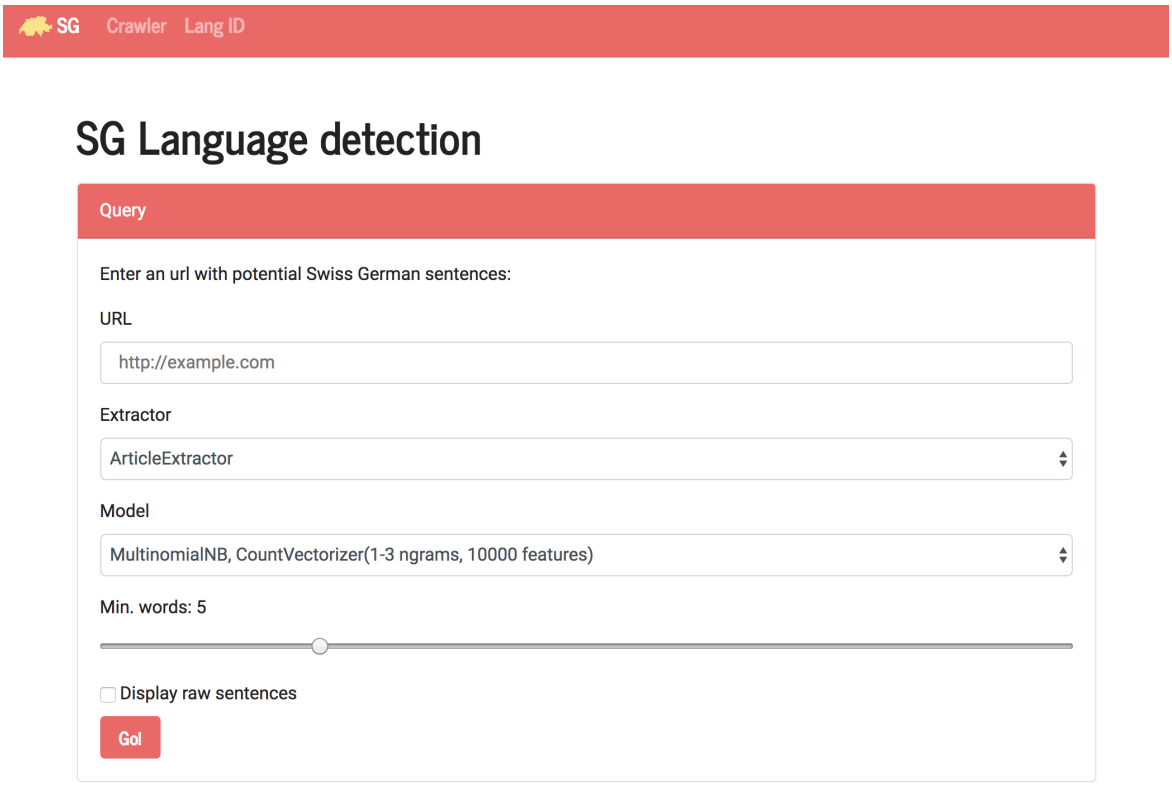


Figure 7.1: Webapp: "crawler" page.

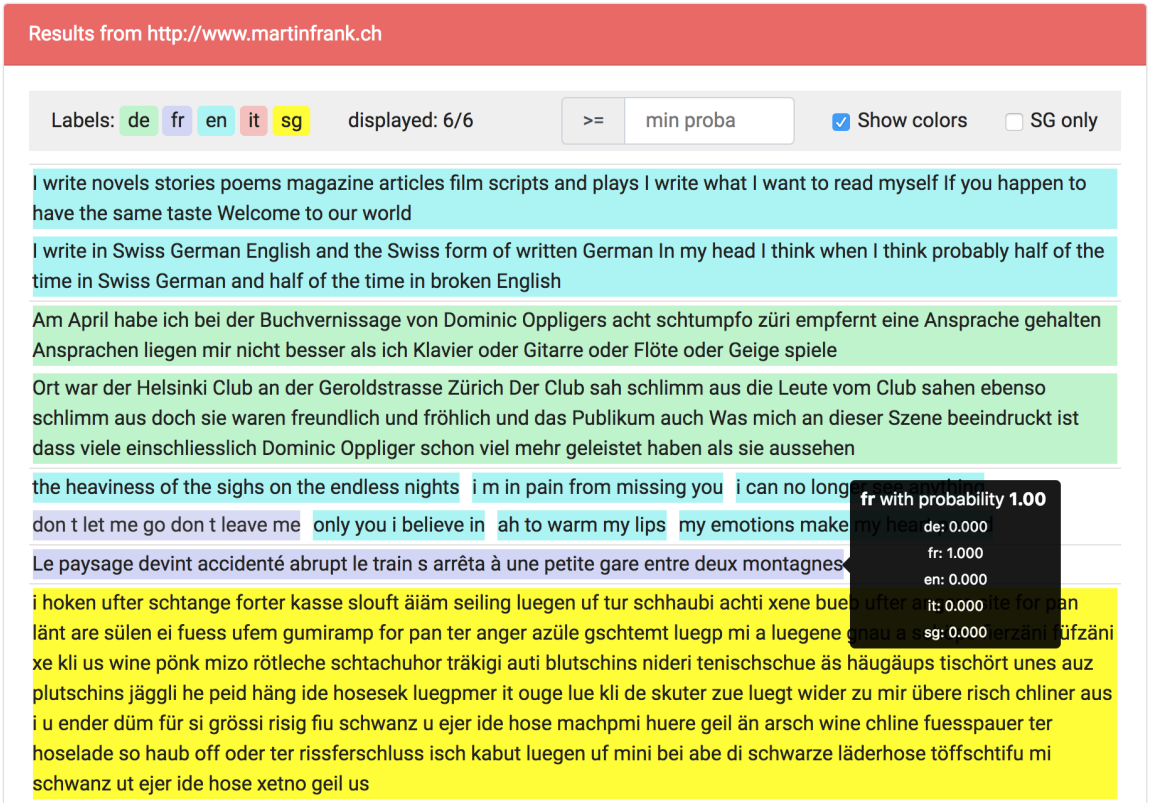


Figure 7.2: Webapp: example of crawl results.

```python
def get_sentences(url: str, extractor_name=EXTRACTORS[0], model=MODELS[0],
    min_words=0, with_proba=False, return_raw=False):
    """
    Get sentences and language predictions from an URL.
    :return: [[(sentence, lang, proba)]]
    """
    # use boilerpipe to extract text blocks
    extractor = Extractor(extractor=extractor_name)
    model = models[model]
    extracted_text = extractor.getTextBlocks(url=url)
    # don't call the model if we don't have at least one text block
    if len(extracted_text) > 0:
        func = model.predict_proba if with_proba else model.predict
        return [
            preds for preds in(
                func(ss.split("\n"),
                        min_words=min_words, return_raw=return_raw)
                for ss in extracted_text
            )
            if preds # ensure we don't return empty results
        ]
    return []
```

Listing 15: Implementation of `get_sentences`.

The language identifiers described in Chapter 5 have been serialized using the pickle module. We wrote two class wrappers, `Model` and `Models`, to ease the loading and the use of those classifiers inside the webapp.

A single method takes care of the whole process of text extraction and language prediction. It is shown in listing 15. Note that depending on the extractor, the text returned by *boilerpipe* can contain multiple sentences, separated by a newline character (\n). This is why we first split the text blocks before calling our language classifier.

7.3.3 Extractors

We were surprised by the capabilities of *boilerpipe*. In most cases, the text returned is on-point and complete. However, the extractors are very different and choosing one is a matter of compromise. For example, `KeepEverythingExtractor` guarantees that we don't miss a relevant text, but also returns results such as "click here to register" or "written on Sept. 2012". Being very short, those kinds of text are often misclassified, leading to false positives. On the other hand, `ArticleExtractor` increases our chances to miss a relevant sentence. Indeed, the *boilerpipe* heuristics are not perfect and can potentially miss important areas of a page. After some tests, we settled on the `ArticleExtractor` for its simplicity.

7.4 Summary

Extracting text from URLs raises many challenges. *boilerpipe* is a Java library well-suited for the task, being efficient, fast and easy to use. To get acquainted with sentence extraction from the Web and test our language identifiers on real data, we created a simple Python 3 webapp.

8 The .ch domain approach

Chapter's content

8.1 The idea 41
8.2 Implementation 41
8.3 Execution 47
8.4 Results 48
8.5 Conclusion 49

This chapter describes our first try on finding Swiss German sentences online: crawling the whole .ch domain.

8.1 The idea

As Swiss German is only spoken in Switzerland, we figured the most likely place to find new sentence is inside the .ch domain.

According to domainlists.io, there are currently more than one million registered .ch domains. As it is likely that each domain contains more than one page, we are talking of several million pages to parse...

Due to the short timespan of the project, we decided to limit the search to the landing page of each domain. While this is not ideal, it can help us determine the viability of the approach. Moreover, we assume that a website containing Swiss German sentences is likely to feature at least one Swiss German sentence somewhere in the landing page, or at least German sentences. Thus, a first run can give us insight on where to look on another iteration.

8.2 Implementation

8.2.1 General overview

To be able to parse more than one million pages, we need an efficient pipeline. We settled for a distributed approach using Apache Spark running on an Apache™ Hadoop® cluster. The main program runs on the DAPLAB - *Data Analysis and Processing Lab* - a Hadoop® cluster powered by the *Haute école d'ingénierie et d'architecture de Fribourg*. The general architecture of the system is illustrated in Figure 8.1.

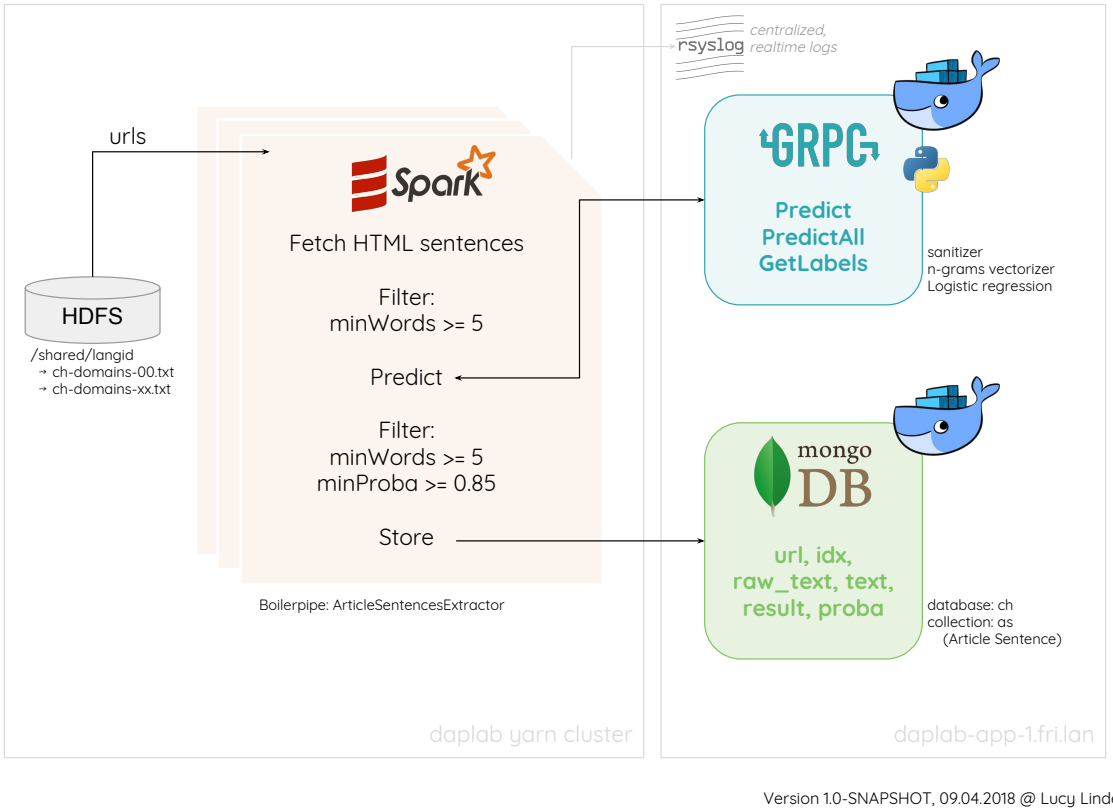


Figure 8.1: .ch domain crawler: system overview.

To make the system more flexible and to cope with different programming languages, we split the pipeline into three main components:

1. *langid-microservice*: a Python 3 microservice for language identification;
2. *database*: a MongoDB database to store the results;
3. *crawler*: the main program.

Both the *langid-microservice* and the database are running inside a Docker container, making them portable and easy to install/upgrade. The main program is written in Scala, while the microservice is implemented in Python 3 (see Chapter 5).

8.2.2 The langid-microservice

The langid-microservice is a standalone RPC server for language prediction with four endpoints that are described in Table 8.1.

The language identification model used is a logistic regression classifier using 6'000 features based on variable size N-grams between 1 and 3 and a *tf-idf* scaling. The sentences are sanitized by keeping only letters, dots and commas[1].

The microservice is implemented using gRPC, a modern high-performance RPC framework ini-

[1]This choice has been made in the middle of the project. We didn't have all the final models available and made a decision based on the best results available at the time.

Endpoint	Description
`GetVersion`	Get the version and description of the language identifier used.
`Predict`	Predict the language of a sentence. The response includes the sanitized text and the list of probabilities.
`PredictAll`	Same as `Predict`, but for multiple sentences.
`Labels`	Get the list of labels, as text.

Table 8.1: Endpoints of the langid-microservice.

tially developed by Google[2] that uses Protocol Buffers (*protobuf*) to define the service and serialize the data. It is packaged in a Python 3 module that includes both the server and an interactive command-line client for quick testing (see listing 16). By default, the server runs on `localhost` using the usual gRPC port, 50051. The setup is explained in detail in a `README`. To simplify the deployment, the project directory also includes a `Dockerfile`.

```
$ python -m langrpc.client
LANGID version 1: TfidfVectorizer_ngrams3-5_f6000_logreg

> bonjour, en quel langage suis-je entrain de m'exprimer s'il-vous-plait ?
bonjour, en quel langage suisje entrain de mexprimer silvousplait

--> fr

    fr: 0.857799
    sg: 0.109292
    de: 0.023057
    it: 0.00539807
    en: 0.00445334

> Ich wünsch Ihne e schöne Daa
ich wünsch ihne e schöne daa

--> sg

    sg: 0.943013
    it: 0.0301972
    de: 0.013073
    fr: 0.0110014
    en: 0.00271568

> exit
Bye!
```

Listing 16: Example of an interactive session using the gRPC client.

There are multiple advantages of using gRPC instead of a regular JSON REST API. First, it uses a binary serialization format, resulting in smaller payloads and faster transmission. Second, it comes with a powerful set of tools to bootstrap both the server and the client code in more than eight languages, including Python and Scala. Third, the resulting server code is self-contained, performant and able to scale to millions of RPCs per second without any modification on our part. Finally,

[2]Contrary to popular beliefs, the "g" in gRPC does not stand for "google". See https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md for more information.

gRPC also supports authentication and bidirectional streaming, features that could be exploited in another release.

8.2.3 Database

We decided to use MongoDB as a database engine.

MongoDB is a free open-source NoSQL document-oriented database. Its main advantage is its *schema-less* design. It stores data in flexible JSON-like documents, meaning that the structure can vary between documents and change over time. This is especially useful for a first implementation, as it gives us the opportunity to test variable formats and change our mind anytime without hassle. Other advantages include a powerful aggregation framework supporting complex queries, an official driver for Scala and a Docker image available on Docker Hub.

The final database contains two collections: `as` and `log`. `as` stands for "Article Sentences" and contains the actual Swiss German sentences found. For each sentence, we keep the following type of information:

- *source*: the domain and URL of the page where the sentence was found, as well as its index in the collection of sentences returned by *boilerpipe*. The latter is useful to get an idea of the relative location of the sentence on the page.
- *sentence*: the original sentence;
- *result*: the sentence after sanitization, the label of the language predicted and the array of probabilities as returned by the langid-microservice `Predict` endpoint;
- *code version*: the initials of the *boilerpipe* extractor used and the classifier information returned by the langid-microservice `GetLabel` endpoint. This information is useful for reproducibility purposes.
- *time*: the date and time of the crawl.

To avoid duplicates, the *primary key* is derived from the language identifier version, the *boilerpipe* extractor, the URL and the index of the sentence in the page using the following pattern:

```
<version number>-<extractor initials>|<url hash>-<idx>
```

The `log` collection is meant to keep a trace of each URL crawled, whether it contained Swiss German or not. One record contains the same code version fields as in the `sg` collection, the URL, the timestamp, the number of sentences found, the number of Swiss German sentences and an extract of the exception message, if any. Since logs should be generated on each run, the *primary key* is randomly generated.

Listing 17 gives an example of documents for each collection.

8.2.4 Spark crawler

The Spark Crawler application contains the main logic. As illustrated in Figure 8.1, it takes as input one or more text files containing URLs and does the following tasks:

1. Download the HTML content of the URL using the built-in `URLConnection`;

```
{
    "_id": "1-ASE|1700875192-0",
    "domain": "0713.ch",
    "url": "http://0713.ch",
    "idx": 0,
    "raw_text": "Wenn zom Fänschter use luegsch :)",
    "text": "wenn zom fänschter use luegsch",
    "result": "sg",
    "proba": [ 0, 0, 0, 1 ],
    "extractor_name": "ASE",
    "version_number": 1,
    "version_description": "ng3-5_sg_f6k_lreg",
    "when" : ISODate("2018-04-20T13:23:34Z")
}
```

```
{
    "_id": ObjectId("..."),
    "url": "http://0-1.ch",
    "sg": 14,
    "count": 200,
    "ex": "",
    "when": ISODate("2018-04-20T13:23:19Z"),
    "model_version": 1,
    "model_version_descr": "ng3-5_sg_f6k_lreg",
    "extractor": "ASE"
}
```

Listing 17: Example of MongoDB records: an `sg` document and a `log` document.

2. Extract sentences using the *boilerpipe*'s `ArticleSentenceExtractor`;
3. Discard sentences with less than w words;
4. Predict the languages by calling the langid-microservice;
5. Discard sanitized sentences with less than w words;
6. Discard sentences with an *sg* probability less than p;
7. Insert the remaining sentences to MongoDB (if any) as well as a *log report*.

The constants w and p are configurable at launch, but we always used the default $w = 5$ and $p = 0.8$. Filtering the sentences before calling `Predict` doesn't change the results, but can help limiting the RPC calls and thus improve performances.

8.2.4.1 Configuration

Various aspects of the program can be configured using a `.properties` file passed as an argument or through the `spark-submit`'s `--properties-file` option. This includes the path to the input file(s), the connection information for the gRPC and the MongoDB services, the name of the Mongo database/collections and the filters to apply on the results. Except for the input filepath, all options have a default value[3]. Internally, the properties are encapsulated into a `Serializable` object and passed to all the nodes in the cluster.

[3] See `src/main/resources/default_config.properties`.

8.2.4.2 Fetching HTML

We wrote our own `HTMLFetcher` instead of using the one shipped with *boilerpipe* in order to:

add support for invalid SSL certificates: by default, Java's `URLConnection` throws an exception when trying to connect to an SSH endpoint with an invalid or missing certificate. To get past this limitation, we can either change the `CACERT` file located in the JRE, or override the default `TrustManager` through code. We decided upon the latter for obvious reasons.

improve the parsing of the charset *attribute in the server's response*: *boilerpipe*'s implementation doesn't work with charset attributes surrounded by quotes.

better deal with encoding errors: by default, charset decoders use a *replace* strategy when encountering malformed or unmappable character. This means dropping the erroneous input, appending the coder's replacement value to the output buffer and resuming the operation. We discovered that using an *ignore* strategy, that is dropping the invalid input and resuming the operation, yields better results in most circumstances. Here is an example:

```
replace: Egau was, Ã¤r ziÃ¤hts - Dr Urs MÃ¼ller us SpiÃ¤z!
ignore:  Egau was, är ziähts - Dr Urs Müller us Spiäz!
```

Add timeouts: we configured the `URLConnection` so it will abort the operation after a timeout of 10 minutes.

Despite those improvements, there are still issues that need to be addressed. The most problematic is when the HTTP server has a bug and returns a never-ending stream of PHP errors. This occurred multiple times in the ch domain, making our program fail with an `OutOfMemoryEror` exception. Unfortunately, there are no simple ways to detect such situations. Another recurrent problem is a server that declares a different *charset* in the header than the one used in the HTML page.

8.2.4.3 Logging

When the program executes on a Yarn cluster, logs are written to temporary files on each node and collected when the program finishes. This is problematic when we need to monitor the program *during* execution... Our work-around is to use rsyslog.

rsyslog has been configured on one machine on the cluster in order to accept TCP connections and to redirect all message tagged with the `langid` prefix to a separate file, `/var/log/langid.log`. This file is also rotated every day using the Linux `logrotate` system utility, so it won't consume all the available disk space.

The last step is to configure the application to use `rsyslog`. Spark uses `log4j` as a default logging utility, which already provides an `rsyslog` handler[4]. The only difficulty is to override the default `log4j.properties` file shipped with Spark... After some trials and errors, we were able to find the proper options to use with `spark-submit`, which are shown in listing 18.

[4] See the file `log4j-rsyslog.properties` at the root of the project for an example.

```
# the file log4j.properties contains the custom configuration
# it should be present in the current folder
absolute_path=$(pwd)
filename=log4j.properties

spark-submit \
 --files "$absolute_path/$filename" \
 --driver-java-options "-Dlog4j.configuration=$filename" \
 --conf "spark.executor.extraJavaOptions=-Dlog4j.configuration=$filename" \
    application.jar
```

Listing 18: Options used with `spark-submit` to specify a custom `log4j configuration`

8.2.4.4 Other notes

To avoid instantiating a new gRPC or MongDB connection for each URL, we applied the *Singleton* pattern. This ensures that only one connection per node is alive.

To package the application and deal with the dependencies, we use the scala-sbt manager, as it is commonly used across Scala open source projects. We were able to setup the project quickly through our *IntelliJ* IDE. However, we encountered a lot of difficulties when we had to configure the `assembly` task (i.e. the creation of a *fat jar* including both our code and the dependencies). It happened that for some libraries, we used versions different from the ones included in the Spark distribution on the DAPLAB. As a result, we had to *shade* some libraries (i.e. rename the packages inside the jar) in order to have a working jar.

8.3 Execution

The scraping of the whole .ch domain took about two weeks to complete.

Jean Hennebert provided a text file with **1'367'215** URLs in the .ch domain. This file has been split into chunks of 1'000 URLs and uploaded into a folder on HDFS, resulting in 273 files to process.

At first, we created one instance of the application for every 10 files, then augmented the load up to 30 files per application instance. In theory, we could just launch one instance for all the URLs, but this means that if the program fails at the penultimate URL, we would have to relaunch everything from the start[5].

The number of executors to use depends on the number of URLs to process and the number of instances running. After some trials, we settled for 3 executors and 20-30 files per program, with a maximum of 4 programs running at the same time.

The time necessary to process one file is difficult to estimate, as it depends on a lot of factors: the hardware present on the nodes, the payload of the cluster, the network, the number of unreachable URLs... In our first tries, we obtained an average of **45 minutes** per 1'000 URLs (about 30 seconds per URL) and were able to speed up the processes a bit more by changing the priority of the jobs on the cluster.

[5]We could also find a way to tell exactly what URL triggered the error and which URLs were already processed, but in practice this is more difficult than it sounds.

About the Daplab

The DAPLAB is a Big Data infrastructure located at the *Haute école d'ingénierie et d'architecture de Fribourg*. Running on Hadoop® , it currently features 1'068 cores, 5'688 GB of RAM and a storage capacity of 575 TB.

In total, about 20 machines are configured as working nodes and can potentially be used by *Spark* applications. In practice, this number can vary depending on the workload on the cluster. Furthermore, the machines vary greatly in terms of hardwares, which can have an influence on the execution time. Older machines have *Xeon 5XXX* processors and 32 GB of RAM, while newer ones feature *dual E5-XXXX v3* CPUs and 128 GB of RAM.

More information is available on daplab.ch as well as in the DAPLAB documentation.

During the execution, we faced several difficulties, including:

- *cluster instability*: the cluster is a bit unstable, resulting in lost nodes and connectivity issues.
- *failed jobs*: some URLs can make the job fail, particularly when they have an error in their PHP code (see Section 8.2.4.2).
- *never-ending jobs*: on multiple occasions, a job stops doing anything without any error or apparent reasons. We are still not sure on what can cause this issue, and whether the cause is Spark, the cluster or our application. As we write, a job is still "running" after 554 hours...

8.4 Results

In total, we processed **1'150'975 URLs**. 227'684 of them were *unreachable* and 321'046 triggered encoding or server errors, leaving **829'929 HTML pages** with potential Swiss German.

Using a logistic regression classifier based on 6'000 Swiss German N-grams between 1 and 3 characters and a *tf-idf* scaling (see Chapter 4), we gathered $30'452$ sentences having a Swiss German probability $\geq 80\%$. If we restrict the results to more than 95% probability, this number drops to 1'517.

Threshold	≥ 0.85	≥ 0.90	≥ 0.95
Number of results	$30'452$	$7'969$	$1'517$

The Swiss German probabilities cannot be blindly trusted. As discussed in Section 6.3, the logistic regression classifier tends to predict Swiss German with a high confidence when encountering a zero feature vector. As a result, unknown languages such as Armenian or Arabic, ASCII art or part of CSS code are a good proportion of the 1'517 sentences. Addresses or copyrights are also likely to be labeled Swiss German. Table 8.2 shows some typical examples of such instances.

By scrolling through the results, we were able to detect about 40 sites *actually* written in Swiss German, about two thirds of which having little text content. They are usually local music or poetry groups, personal pages or city's special event pages. The most promising site is `http://as-we-travel.ch`, a travel blog featuring more than 150 articles written in Swiss German.

Correct sentences:	
97%	s gliche isch mitem stromnetz und de wasserversorgig i new york. all die leitige und versorgigsinfrastruktur isch extrem alt, und drum isches nid sälte dass es mal n komplette stromuusfall git. glaubs im summer isch de letschti riesä shutdown xi, [...]
95%	än wichtigä teil vo dä päge isch di umfangriichi galerie
96%	merci thömu, jetzt isch zzwänzgi abe gheit
90%	Itz si mer o über Facebook derbi...
Incorrect sentences:	
93%	a..... ad..... adj..... ADJ..... ADJ..... [...]
86%	ŋŋ դարի երրորդ միւռոնը եւ կոչուեցաւ յաղթութեան միւռոն
92%	CD sRössli Hü bim König CD sRössli Hü bim König
91%	o u r m i s s i o n
91%	Zürich Zürich Zürich Zürich Zürich Zürich Zürich Zürich Zürich Zürich
85%	Telefonisch +49 1805 2455269 oder per Mail: welcome@billbox.com
86%	saved from urlhttpinternet.email htmlhead titlemeine homepagetitle head hintergr. weiss, body bgcolorffffff linkdee vlinkdede bild über die ganze seitenbreite p aligncenterimg [...]

Table 8.2: Example of sentences found in the .ch domain.

8.5 Conclusion

In this chapter, we described a distributed pipeline to extract Swiss German sentences from URLs and how we used it to scrape the homepages of 1'150'975 URLs from the .ch domain.

The *.ch domain approach* is not a good alternative to find new Swiss German sentences. Out of more than one million URLs, we were able to track down 50 potentially interesting websites and about 600 new sentences. Said bluntly, those results are not worth the pain.

Looking at homepages only might not be a good strategy, as they are usually developed to reach the maximum number of people, hence the usage of English or German instead of dialects.

This lack of results doesn't mean there is not Swiss German online. Instead, we suppose dialects are more likely to be used in informal contexts such as golden books, commentary sections and forums.

9 The search Google approach

Chapter's content

9.1 The idea 50
9.2 Proof of concept 51
9.3 Existing work 53
9.4 Conclusion 55

9.1 The idea

As our *.ch domain approach* described in Chapter 8 suggests, Swiss German is not likely to be found on landing pages. This leads us to make the following assumption:

Swiss German is mostly used in *informal contexts*,
such as forums, golden books, etc.

The question is *how to access those pages* ? Where are those informal online platforms ?

We discovered later that this approach has already been described in a few papers. It is discussed further in Section 9.3.

As humans, we are accustomed to use various search engines to navigate the Web. We have an idea in mind, type some keywords and quickly spot interesting URLs among the results. Our idea is to use the same kind of logic for Swiss German, that is:

1. Gather a list of common Swiss German keywords or short sentences;
2. Produce several search queries using the list above;
3. Send the queries to a search engine and collect the top X URLs returned;
4. Filter the URLs and ensure their uniqueness;
5. Use a scrape engine, for example the pipeline presented in 8 to gather new sentences.

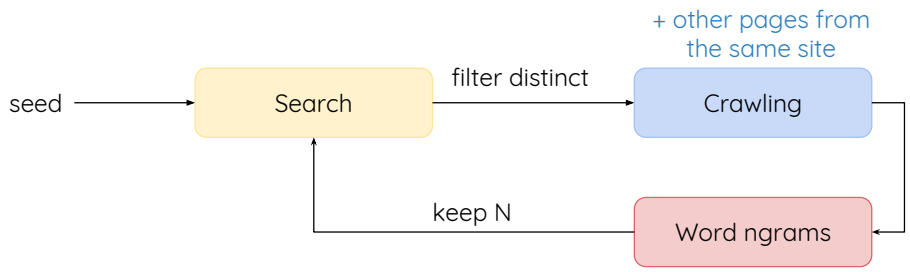


Figure 9.1: An example of reinforcing loop based on the *Search Google approach*.

In the first iteration, we need to provide a *seed*, i.e. a first search query. Afterwards, we can devise a system to extract new seeds from the results, thus creating a *positive reinforcing loop*.

Another improvement to the system would be the ability to scrape other related URLs, either internal or external to the current domain, when we find a good proportion of Swiss German in a page.

This general idea is presented in Figure 9.1.

Inception

The story behind the *Search Google approach* is a nice anecdote.

This idea came up during a meeting with my supervisors. I was explaining how the *.ch domain approach* was not going as expected and asked Andreas to provide me a list of URLs with actual Swiss German in order to ensure the problem didn't come from the detection step of the pipeline.

Andreas opened a new tab, typed "*das isch sone seich*" (translating bluntly as "*this is bullshit*", to fit the mood) on Google and started copy-pasting the results into a text file. My programmer's mind didn't need more... I asked him to provide me with other search terms and went straight home to devise a first prototype.

9.2 Proof of concept

A Swiss German speaker provided us with five seeds likely to emerge in informal contexts such as forums:

- "*das isch sone seich*": that's bullshit;
- "*das isch super*": that's great;
- "*weiss öpper*": does somebody know...
- "*het super*": does somebody have...
- "*wär chamer*": who can help/give me ...

To automatize the retrieval of search engine results, we tried two approaches: the use of an existing API, and the scraping of an online search engine.

Google Custom Search is originally conceived to provide custom search for one or a collection of websites. It is, however, possible to parameterize an engine to search the whole internet. Google also provides a Custom Search JSON API that we can use to make query and retrieve a list of URLs as a result. This API matches exactly our needs, but it requires an associated Google cloud project and its free version is limited to 100 queries a day. Furthermore, the API limits the number of results to 10 URLs/query, meaning that retrieving the first 100 results for a given seed already accounts for 10 queries...

Using an existing engine by script is another possibility, but is not disregarded by site owners. Google for example has measures to detect and block such "robot" accesses[1]. We tried anyway using a lesser known engine, startpage.com. It is slower than popular search engines, but the results seem to be of good quality nonetheless.

We wrote two command line scripts in Python 3, `googlesearch.py` and `startpage.py`. Both take a query and a number of desired results as parameters and output the retrieved URLs into a text file. Note that in case the scripts are called multiple times, it is possible to retrieve the same URL twice.

Using those scripts, we retrieved **211 unique URLs** from the five seeds listed above and fed them to the crawler described in Chapter 8. It took **less than three minutes** to run and gathered more than **10'000 Swiss German sentences**, 8'500 after removing duplicates. The average Swiss German probability is a little above 90%. Listing 19 shows more statistics about the results.

```
URLs                      Number of sentences
---------------           --------------------
crawled:    211           Proba >= 80%: 10'122  (uniques:  8'557)
errors:      14           Proba >= 85%:  8'816  (uniques:  7'479)
no SG found: 54           Proba >= 90%:  6'470  (uniques:  5'528)
                          Proba >= 95%:  2'189  (uniques:  1'903)

avg proba: 0.91


Sentences per URL         Characters in sentences
--------------------      -----------------------
mean :      66.59                          raw        san
std  :     153.87                          ---        ---
min  :       1.00          count:        10122.00   10122.00
25%  :       3.75          mean :          150.16     154.04
50%  :      12.00          std  :         5501.09    5578.15
75%  :      51.00          min  :           15.00      16.00
max  :    1487.00          25%  :           33.00      34.00
                          50%  :           47.00      50.00
                          75%  :           79.00      83.00
                          max  :       553307.00  561059.00 (next 3533)
```

Listing 19: Statistics about the Swiss German sentences found during the proof of concept.

[1] We discovered that by direct experience: it took Google about one hour to detect and block our IP address...

9.3 Existing work

As we later discovered, using search engines to create language corpora is not a novel approach. We recommend Sharoff's paper "*Creating General-Purpose Corpora Using Automated Search Engine Queries*" (2006) [33] for a more in-depth presentation of the technique.

9.3.1 CorpusBuilder

The idea first originated from Ghani et al. in 2001 [34], who developed *CorpusBuilder*, an approach for automatically collecting documents in a minority language using Web queries. The general algorithm is as follows:

1. Initialization
2. Generate query terms from relevant and non-relevant documents;
3. Retrieve next most relevant document for the query;
4. Language Filter, assign document to relevant or non-relevant set;
5. Update frequencies and scores based on relevant and non-relevant documents;
6. Return to step 2.

They experimented with various methods for initialization and query term selection. The starting-point can be a small set of documents, three in most of the experiments, or user-supplied keywords. The queries are a conjunction of k required and k forbidden words selected from the list of existing documents, using one of six methods: *uniform random selection*, *term-frequency*, *probabilistic term-frequency*, *rtfidf*, *odds-ratio* or *probabilistic odds-ratio*. *TextCat* [15] was used as a language filter to discriminate between relevant and non-relevant documents[2] and *AltaVista* as a search engine, a very popular engine at the time later purchased by *Yahoo!*. Most of the experiments were conducted with Slovenian, then partly repeated on Tagalog, Croatian and Czech, suggesting that this approach is generalizable to dialects such as Swiss German.

They obtained the best results using 3 inclusion and 3 exclusion terms in queries ($k = 3$) selected using the *odds-ratio* techniques. Using user-supplied keywords as seeds performed as well as using whole document, but not significantly better. Overall, the queries returned a new document about 80% of the time. To estimate the performance of the system, they asked a native speaker to evaluate a small number of pages randomly selected from a pool of several thousand results. Out of 100 pages judged pertinent by the system, 99 were true positive. The rate of true negative is a bit smaller, between 90 and 95%.

As we can see, the algorithm is very similar to the one we applied, except for two things: the use of forbidden words in queries and the collection of a single document at each iteration.

9.3.2 BootCaT

Following [34], Baroni et al. developed the *BootCaT toolkit*, a suite of Perl programs to bootstrap specialized corpora and terms from the Web [35]. As shown in Figure 9.2, their approach is similar to the one introduced by Ghani et al. For specialized domains, they found that a very small list of seeds, ranging from 2 to 15, was sufficient to get decent results. Contrary to [34], they kept the top 10 results for each query, which was generated using a random combination of words. In their

[2] They also experimented with constructing a filter on-the-fly, which yielded encouraging results when applied to the Tagalog language.

experiment, they never had to repeat the entire process more than two or three times to gather enough data. In [36], they also discuss the use of this tool to gather Japanese specialized terms and corpora from the Web.

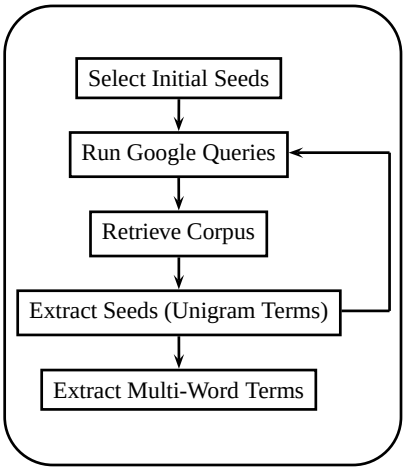


Figure 9.2: The BootCaT flow [35].

Since their publication in 2004, the BootCaT toolkit has evolved to become a standalone, multi-platforms application for more generic usages. From the download page,

> BootCaT automates the process of finding reference texts on the Web and collating them in a single corpus [...] one can build a relatively large quick-and-dirty corpus (typically of about 80 texts, with default parameters and no manual quality checks) in less than half an hour.

We tried the software on Windows. Using one single Swiss German seed, "das isch sone seich", one iteration, a German filter and the default options, we were surprised to gather as much as 21'434 tokens from 10 URLs. Contrary to our crawler, *BootCaT* handles PDFs as well as regular HTML pages.

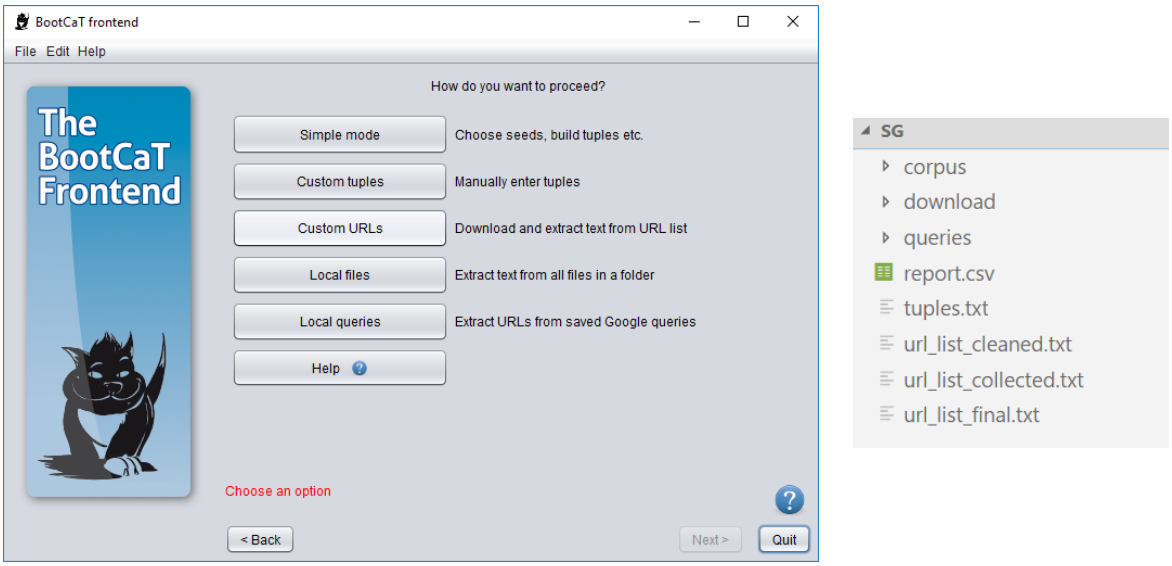


Figure 9.3: The BootCaT frontend on Windows and the generated corpus' folder structure.

As shown in Figure 9.3, the resulting corpus contains many useful information, including the

original files, the list of URLs crawled and a csv file with overall statistics about the corpus. The only drawback is that this tool requires a lot of human interaction. Indeed, the wizard is composed of more than 10 steps that need to be manually tuned. Furthermore, to retrieve the Google Search results, there is no automation involved, so that one needs to open each query page in a browser and manually save the HTML page into a `queries` folder...

9.3.3 Leipzig

The *Leipzig Corpora Collection* presented in 2.2 uses different techniques to find relevant URLs to crawl, one of which being based on search engine queries [13]:

> Frequent terms of a language are combined to form Google search queries and retrieve the resulting URLs as a basis for the default download system. As a small set of frequent terms is needed for each language, the *Universal Declaration of Human Rights* was used as a resource. [...] Based on the lists of seeds tuples of three to five high frequent words are generated. These tuples are then used to query Bing and to collect the retrieved URLs. In a next step these websites are downloaded and further preprocessed.

The Leipzig pipeline is thus fully automated and all the seeds come from the UDHR, meaning that the accuracy (and diversity) of the results is not guaranteed.

9.4 Conclusion

In this chapter, we discussed the idea of using search engines to retrieve relevant URLs as the basis for crawling. Our proof of concept proved the technique to be worth investigating. Upon further analysis, we discovered that this approach has already been documented and incorporated in different tools, which we described briefly.

We have gathered a corpus of about 10'000 Swiss German sentences in one iteration using the most basic approach. Possible next steps include: to improve the query generation step, to automate the pipeline and to implement an iterative loop. We also believe that incorporating humans in the pipeline can help improve the results, both in terms of performance and in terms of relevance of the resulting corpus.

10 Conclusion

Chapter's content

10.1 Summary 56
10.2 Review of the objectives 57
10.3 Conclusion and perspectives 57
10.4 Final word 58
10.5 Acknowledgements 58

10.1 Summary

In this project, we explored different techniques to sustain the creation of a Swiss German corpus of unseen, *in context* sentences. Using an agile approach, we realized several pieces of software which assemble the necessary tools for a full prototype. One assumption that guided many of the decisions made throughout this project is that the Web is the best source for new Swiss German material.

To get started, we defined the steps of a new corpus creation as two intertwined cycles and decided to focus on the first one, the *data gathering loop* (Chapter 1): from a *quickstart dataset*, create a language identifier, use it to gather new material, then use this new material to improve the language model, etc.

In Chapter 2, we identified six existing Swiss German corpora or online sources of Swiss German sentences. We used two of them, NOAH and the Leipzig collection, to create a training set of 7'000 sentences for each of five languages: English, Italian, French, German and Swiss German. We also created two other datasets based on the Swiss SMS corpus, for validation purpose. This is described in Chapter 3.

LID - *Language IDentification* can be viewed as a supervised classification task (Chapter 4). We settled for character-based, bag-of-words approaches using N-grams as features, and experimented with four main families of classifiers: Naive Bayes, linear regression, SVMs, and Neural Network, in Anaconda notebooks (Chapter 5). As presented in Chapter 6, all those classifiers scored well, with an accuracy greater than 98%. However, they behave quite differently on real life data, though none of them is really stepping out.

The last step of the cycle is data gathering. In a first attempt (Chapter 8), we crawled more than one million landing pages from the Swiss `.ch` domain using a distributed Apache Spark application. This yielded very poor results, less than 1'000 new Swiss German sentences, suggesting that Swiss

German is mostly used in more informal contexts such as blogs or social media. In a second attempt (Chapter 9), we used a Web search engine and manual "seeds" to gather URLs likely to have Swiss German content. Crawling those URLs yielded far better results: using only 5 seeds, 211 URLs and 3 minutes of processing time, we gathered more than 8'000 unseen Swiss German sentences.

10.2 Review of the objectives

We consider to have met all of the core objectives presented in Section 1.3.2.

1. language identification
   (a) ability to detect Swiss German ✔
   (b) (*nice-to-have*) ability to label sentences with dialects ✖
2. data gathering
   (a) programs and tools to find new Swiss German resources ✔
   (b) (*nice-to-have*) a corpus of new Swiss German sentences ✔

The timespan of the project didn't leave us time to investigate the dialect idenfitication (point 1(b)) further. However, the optional objective 2(b) is at least partially met: the code repository contains multiple text files with new Swiss German sentences as well as a *dump* of the MongoDB databases containing the results of the crawl.

10.3 Conclusion and perspectives

If there is one thing to draw from this project, it is that:

There are a lot of Swiss German resources still to be discovered

More specifically, we want to highlight the following:

(1) Swiss German language identification is possible using simple techniques such as Naive Bayes or logistic regression;
(2) Scraping the Web is a challenging task: sanitization, character encoding and text extraction are especially difficult;
(3) Using search engines to direct the search of Swiss German on the Web is a promising approach.

Language identification already yielded good results. However, we believe it can be further improved by using data of higher quality. If we want to support dialects as well, we might consider more complex classifiers such as Neural Networks along with *hybrid* architectures or *co-training*. For example, we could train several models on different feature sets and combine their output using a logistic regression. *Online* models are also worth investigating, meaning that the models are constantly updated as new data is found.

As discussed in Chapter 8, the language identification might fail when confronted to languages that are not part of the training domain. This is especially true for languages using a different character set, such as Russian or Japanese. Thus, one way to improve the scraping pipeline is to detect the character set early and discard data that are not using the ISO-8859-1 (Latin 1) encoding.

Another improvement would be the ability to scrape related pages. Related pages can be either pages from the same domain, or external pages referenced by hyperlinks in the text. However, this raises several technical challenges, especially if the scraper is a distributed application: how can we ensure we don't scrape the same page multiple times ? How to avoid infinite loops ? This requires additional tools and synchronization protocols.

In order to find relevant websites to scrape, the *search engine approach* seems promising and is worth investigating further. In the lineage of *CorpusBuilder* and *BootCaT* (Chapter 9), our proof of concept could be refined in order to support iterations. In this perspective, we use the output of a first run to generate new queries that give birth to a new iteration.

We believe that the *search engine approach* can be adapted to support the creation of *dialect-oriented* corpora as well. Provided the lack of existing corpora tagged with dialects, we would suggest a *human-in-the-loop* approach: the system relies on human inputs to provide very specific seeds, for example words existing only in *Bärndütsch*, and to validate/label the results. Ideally, as the time passes and more data is found, the system would become less and less dependent on those inputs.

10.4 Final word

On a personal note, this project was very interesting. It helped me sharpen my knowledge in many domains such as Machine Learning, Web crawling and distributed programming and helped me develop new skills, especially in agile project management. Working with Swisscom was also a valuable experience. Finally, I was pleased to discover Swiss German more intimately. Being from the French part of Switzerland, I somehow disregarded this strange language. Working on it for three months radically changed my opinion and I am now proud to say I am quite fascinated by it.

10.5 Acknowledgements

> For good ideas and true innovation, you need human interaction, conflict, argument, debate.
>
> – Margaret Heffernan

I would first like to express my gratitude to my two thesis advisors, Dr. Jean Hennebert and Dr. Andreas Fischer. Thank you for letting me take the lead on this project, while always being here to steer me in the right direction whenever I needed it. Thank you for supporting me and help me build confidence even when the outcomes didn't meet my expectations. Thank you for making the weekly meeting such a friendly and stimulating place and for taking the time to share your technical knowledge with me.

I would also like to thank Claudiu Musat and all the Swisscom team for this great opportunity, and also for their insights and involvement in this project.

A special thank you to Frédéric Bapst for his encouragement and for proof-reading this report in such a rigorous way.

I felt privileged to be part of such an adventure, so thank you to all those that made it possible.

Bibliography

[1] Office fédéral de la statistique. *Suisse allemand et allemand standard en Suisse: Analyse des données de l'Enquête sur la langue, la religion et la culture 2014*. Tech. rep. 2017.

[2] The Swatch Group AG. *Swatch Group Geschaftsbericht 2012*. URL: http://www.swatchgroup.com/de/investor_relations/jahres_und_halbjahresberichte/fruehere_jahres_und_halbjahresberichte.

[3] Nora Hollenstein and Noëmi Aepli. "Compilation of a Swiss German dialect corpus and its application to PoS tagging". In: *Proceedings of the First Workshop on Applying NLP Tools to Similar Languages, Varieties and Dialects*. 2014, pp. 85–94.

[4] Swisscom. *AI learning Swiss German*. [Online; accessed 2018/03/01]. URL: https://www.swisscom.ch/en/business/enterprise/themen/digital-business/language-recognition-swiss-german.html.

[5] Kent Beck et al. *Manifesto for agile software development*. 2001.

[6] Rudolf Ernst Keller. *German dialects: phonology and morphology, with selected texts*. Manchester University Press, 1961.

[7] watson.ch. *Mundart-Forschung: "Dialäkt Äpp" gibt Hinweise auf Sprachentwicklung*. [Online; accessed 2018/03/06]. URL: https://www.watson.ch/Wissen/Schweiz/616850530-Mundart-Forschung----Dialaekt---pp---gibt-Hinweise-auf-Sprachentwicklung.

[8] Tanja Samardzic, Yves Scherrer, and Elvira Glaser. *Archimob - a corpus of spoken Swiss German*. 2016. URL: https://archive-ouverte.unige.ch/unige:91722.

[9] Heath Gordon et al. "Archimob Corpus Release 1.0 documentation". In: (2016).

[10] Elisabeth Stark, Simone Ueberwasser, and Beni Ruef. *Swiss SMS Corpus*. 2009.

[11] sms4science. *Swiss SMS corpus*. URL: http://www.sms4science.ch.

[12] Ralf Grubenmann et al. "Towards a Corpus of Swiss German Annotated with Sentiment". In: *Proceedings of the 11th Language Resources and Evaluation Conference (LREC)*. (to appear). 2018.

[13] Dirk Goldhahn, Thomas Eckart, and Uwe Quasthoff. "Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages." In: *LREC*. Vol. 29. 2012, pp. 31–43.

[14] Kevin P Scannell. "The Crúbadán Project: Corpus building for under-resourced languages". In: *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop*. Vol. 4. 2007, pp. 5–15.

[15] William B Cavnar, John M Trenkle, et al. "N-gram-based text categorization". In: *Ann arbor mi* 48113.2 (1994), pp. 161–175.

[16] Paul McNamee. "Language identification: a solved problem suitable for undergraduate instruction". In: *Journal of Computing Sciences in Colleges* 20.3 (2005), pp. 94–101.

[17] Marco Lui and Timothy Baldwin. "langid.py: An off-the-shelf language identification tool". In: *Proceedings of the ACL 2012 system demonstrations*. Association for Computational Linguistics. 2012, pp. 25–30.

[18] Nakatani Shuyo. *Language Detection Library for Java*. 2010. URL: http://code.google.com/p/language-detection/.

[19] Leonid Panich. *Comparison of Language Identification Techniques*. Bachelor's Thesis, Heinrich Heine Universität Düsseldorf. 2015.

[20] Ali Selamat and Nicholas Akosu. "Word-length algorithm for language identification of under-resourced languages". In: *Journal of King Saud University-Computer and Information Sciences* 28.4 (2016), pp. 457–469.

[21] Gerrit Botha, Victor Zimu, and Etienne Barnard. "Text-based language identification for the South African languages". In: (2006).

[22] Juha Hakkinen and Jilei Tian. "N-gram and decision tree based language identification for written words". In: *Automatic Speech Recognition and Understanding, 2001. ASRU'01. IEEE Workshop on*. IEEE. 2001, pp. 335–338.

[23] Alexander Genkin, David D Lewis, and David Madigan. "Large-scale Bayesian logistic regression for text categorization". In: *Technometrics* 49.3 (2007), pp. 291–304.

[24] Kavi Narayana Murthy and G Bharadwaja Kumar. "Language identification from small text samples". In: *Journal of Quantitative Linguistics* 13.01 (2006), pp. 57–80.

[25] Alberto Simões, José João Almeida, and Simon D Byers. "Language identification: a neural network approach". In: *OASIcs-OpenAccess Series in Informatics*. Vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014.

[26] Logan Wilson. *Human Behavior and the Principle of Least Effort*. JSTOR. 1949.

[27] Priyank Mathur, Arkajyoti Misra, and Emrah Budur. "LIDE: Language Identification from Text Documents". In: *arXiv preprint arXiv:1701.03682* (2017).

[28] Gregory Grefenstette. *Comparing two language identification schemes*. 1995.

[29] Bernardt Duvenhage, Mfundo Ntini, and Phala Ramonyai. "Improved Text Language Identification for the South African Languages". In: *arXiv preprint arXiv:1711.00247* (2017).

[30] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM. 1992, pp. 144–152.

[31] Thorsten Joachims. "Text categorization with support vector machines: Learning with many relevant features". In: *European conference on machine learning*. Springer. 1998, pp. 137–142.

[32] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. "Boilerplate detection using shallow text features". In: *Proceedings of the third ACM international conference on Web search and data mining*. ACM. 2010, pp. 441–450.

[33] Serge Sharoff. "Creating general-purpose corpora using automated search engine queries". In: *WaCky* (2006), pp. 63–98.

[34] Rayid Ghani, Rosie Jones, and Dunja Mladenić. "Mining the web to create minority language corpora". In: *Proceedings of the tenth international conference on Information and knowledge management*. ACM. 2001, pp. 279–286.

[35] Marco Baroni and Silvia Bernardini. "BootCaT: Bootstrapping Corpora and Terms from the Web." In: *LREC*. 2004, p. 1313.

[36] Marco Baroni and Motoko Ueyama. "Retrieving Japanese specialized terms and corpora from the World Wide Web". In: *Proceedings of KONVENS*. 2004, pp. 13–16.

[37] Gerrit Reinier Botha and Etienne Barnard. "Factors that affect the accuracy of text-based language identification". In: *Computer Speech & Language* 26.5 (2012), pp. 307–320.

[38] Rong-En Fan et al. "LIBLINEAR: A library for large linear classification". In: *Journal of machine learning research* 9.Aug (2008), pp. 1871–1874.

I, Lucy Linder, declare under the penalty of perjury that the work performed for this project is my own. I did not copy or use anyone else's published or unpublished results other than those that are clearly stated and attributed to their rightful owners.