University of Wrocław
Department of Mathematics and Computer Science
Institute of Computer Science

Michał Barciś

Regular expressions on audio data

Master's thesis
under supervision of
Jan Chorowski, PhD

Wrocław 2016

Abstract

Nowadays, it is very easy to gather a huge amount of audio recordings, but we are still unable to analyse them efficiently. We could make this data more useful if we had the ability to search for phrases in it. This thesis presents a solution to this problem based on regular expressions. We use them to adjust the weights in a language model and make searched phrases more probable. That way we are not only able to run regular expressions on audio recordings, but also to smoothly change the threshold of similarity between found occurrences and the query.

Acknowledgements

First of all, I would like to thank my supervisor, *Jan Chorowski, PhD*, for his dedication and support. Dr Chorowski was always there to patiently answer my questions and provide suggestions how to improve the work. At the same time, he valued my own ideas and encouraged me to express them in this thesis.

I would also like to thank all members of the *AudioScope* team, whose passion and knowledge always motivated me to do more research; especially *Paweł Rychlikowski, PhD*, who believed in me two years ago and gave me a chance to join the team.

Finally, I would like to thank *my girlfriend and my parents* for supporting me even in the hardest moments.

Thank you.

Michał Barciś

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays a magnitude of digitized audio data exists and is easily available. The first time a person was able to record his voice was in 1857, when a Parisian inventor Édouard-Lon Scott de Martinville patented his invention — the phonautograph (*FirstSounds.org* n.d.). Since then, a massive development in the area of audio recording has been made. Currently almost everyone carries a smartphone on himself, capable of recording sound with exceptional quality. Additionally, the size of recording devices and microphones was reduced so much, that it is possible to record almost everything, everywhere and at all times.

In fact, there is a lot of places where hundreds of hours of speech recordings are being produced every day. Below we will present some examples of such places.

- In **courtrooms**, the testimonies are usually recorded. Sometimes a court stenographer takes part in trials and his role is to transcribe spoken or recorded speech into a written form. However, he is rarely present in the lower court.
- In **call centres** the calls are recorded and archived for future inspection.
- In some companies it is common to **record meetings** in order to keep notes of the decisions that have been made.
- **Detectives** and **law enforcement** sometimes use concealed microphones or wire phones to spy on people.

In all the above cases hundreds, or even thousands of hours of recordings are being gathered. This makes manual inspection of this data highly inefficient and thus impractical. As a result, the true potential of those recordings is not being

exploited. Just think how many options we would have if it was possible to search for any information hidden in this data automatically.

- It would be possible to search for example for a name of a suspect or for all mentions of the word "robbery" in the testimony. Evidence would be harder to miss and, most importantly, people would spend much less time analysing the data.
- In call centres the ability to search for some predefined phrases would make analysis of data much simpler. It would be possible for example to create statistics on how often people asked about particular features, etc.
- Companies could analyse recordings from their meetings and decide almost instantly whether some topic has been discussed and what decisions were made.
- Wiretapped data could be analysed automatically and suspicious recordings could be tagged — for example the ones that mention bombs or guns.

1.2 Thesis objectives

This work is a part of a project named *AudiosScope*, described in the Section 3.2. The main goals for this thesis are:

1. present a problem of searching for patterns in audio data,
2. introduce a solution to this problem, that is based on speech recognition,
3. gather in one place all information essential to understand this solution,
4. provide ideas about additional work that may be done.

1.3 Thesis organization

This thesis is divided into chapters. Each one of them provides some additional knowledge and utilises concepts introduced earlier in this study.

In the rest of the Chapter 1 we introduce the problem and briefly state the goals of this thesis. Chapters 2 and 3 explain theoretical knowledge needed to understand concepts utilised later and describe programs useful during the work on this project.

The main idea of this thesis is presented in Chapter 4. Then, in Chapter 5 some experiments using this idea are outlined together with practical usage.

1.4 Problem formulation

We are trying to solve the problem of searching for a pattern in an audio file. Below we will try to roughly specify it.

Definition 1.4.1
Input:

- *An audio file containing a fragment of speech data, usually a sentence.*
- *A text file containing a pattern to search for.*

Output:
A decision whether the given pattern was pronounced in the audio file or not.

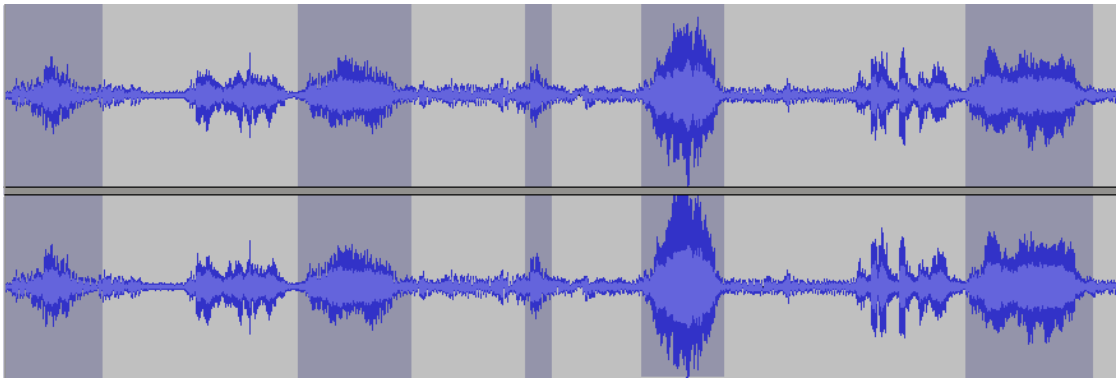

Figure 1.1: A waveform representing an example of speech recording

The problem is not easy to solve. In the figure 1.1 an example of audio recording is given, with areas marked with a darker background. They represent the same phrase pronounced in different manners. Even on this waveform we can see how distinct they are. Some are short, others are very long. Some have small amplitude, some have very high amplitude. The dynamics also change a lot in each pronunciation.

This is only an example, but it shows how complex the problem is. Therefore, in order to solve it, we decided to use the most modern approaches from the field of *natural language processing* and *machine learning* (described in Chapters 2 and 3) and modify an existing speech analysis software.

Chapter 2

Scientific background

2.1 Regular expressions

In the following section we are going to introduce *regular expressions* as an easy way to define a search query. Nowadays, they are very popular and used in various places — ranging from lexical analysers and text editors even to web crawlers.

The aim of this section is to formalize those regular expressions and define a simple notation. In the beginning, we will show some operations on languages that can be represented by them. Then, we will introduce the inductive definition of these expressions. In the end, we will show how to convert them to non-deterministic automata and weighted finite state transducers.

2.1.1 Regular languages

Regular expressions are used to represent *regular languages*. But what exactly is a *regular language*? First, we need to define a *language* (or, more precisely, a *formal language*). It is a (perhaps infinite) set of *words* over a given *alphabet* Σ. For example, we can define a language $L = \{abc, cba\}$ over the alphabet $\Sigma = \{a, b, c\}$. Often we will not state the alphabet explicitly. This knowledge should be sufficient for understanding the rest of this chapter. For more information about this exciting field of knowledge, please refer to the handbook about formal languages, for example (Hopcroft and Ullman 1979).

We will now define some operations on the languages that are going to be later represented with regular expressions. In the following examples both L and R are any languages. All the following operations return a new language.

concatenation The concatenation of R and L contains all strings that can be formed by taking a string from L and appending to it a string from R.

> **Example 2.1.1** *Let $R = \{a, b\}$, $L = \{c, d\}$. Then the concatenation of R and L equals $\{ac, ad, bc, bd\}$.*

union The union of L and R contains all strings from L and all strings from R.

> **Example 2.1.2** *Let $R = \{a, b\}$, $L = \{c, d\}$. Then the union of R and L equals $\{a, b, c, d\}$.*

> **Example 2.1.3** *Let $R = \{a, b\}$, $L = \{a, d\}$. Then the union of R and L equals $\{a, b, d\}$.*

Kleene star The Kleene star of L contains an empty string and all strings from L concatenated with the Kleene star of L. In other words, it contains all variable-length variations with repetitions of all strings from L.

> **Example 2.1.4** *Let $L = \{a, b\}$. The Kleene star of L equals to an infinite language $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb \dots\}$.*

2.1.2 The structure of regular expressions

Regular expressions define languages. Let r be an expression. Notation $L(r)$ denotes a language represented by r.

Now it is time to introduce the inductive definition of regular expressions. We will start with the **basis**:

> **Definition 2.1.5 (regular expressions)**
>
> - *Both ε and $\varnothing$ are regular expressions. The first one denotes the language with only one word of length 0. The second one denotes an empty language.*
>
> > ***Example 2.1.6*** *$L(\varepsilon) = \{\varepsilon\}$, $L(\varnothing) = \varnothing$.*
>
> - *Any symbol from the alphabet Σ is a regular expression. It denotes a language containing only this symbol.*
>
> > ***Example 2.1.7*** *$L(a) = \{a\}$.*
>
> *Let us proceed with the inductive definition. It has four constructors. The first three correspond to the operations on languages defined in Section 2.1.1. The fourth one introduces parentheses.*

- *If A and B are regular expressions, then AB is a regular expression denoting the concatenation of $L(A)$ and $L(B)$.*

  Example 2.1.8 *Regular expression* `abc` *denotes language $\{abc\}$.*

- *If A and B are regular expressions, then $A|B$ is a regular expression denoting the union of $L(A)$ and $L(B)$.*

  Example 2.1.9 *Regular expression $a|b$ denotes language $\{a, b\}$.*

- *If A is a regular expression, then A^* is a regular expression denoting the Kleene star of A.*

  Example 2.1.10 *Regular expression a^* denotes an infinite language $\{\varepsilon, a, aa, aaa, aaaa \dots\}$.*

- *If A is a regular expression, then (A) is also a regular expression denoting the same language. $L(A) = L((A))$.*

2.1.3 Extensions to regular expressions

The definition 2.1.5 gives us the complete tool to denote regular languages. Unfortunately, in practice, this basic set is not very expressive. In order to make it more useful we do need to introduce some syntactic sugar.

Since we will use the regular expressions as a tool to specify search queries, we may often say that the regular expression *matches* some text.

Definition 2.1.11 *Regular expression A matches string $T \iff T \in L(A)$.*

Example 2.1.12 *Regular expression eve|adam matches adam. This expression also matches eve and these are the only two strings that it matches.*

- `.` is a regular expression that denotes a language with all symbols from Σ.

  Example 2.1.13 *If $\Sigma = \{a, b, c\}$, then $L(.) = L(a|b|c) = \{a, b, c\}$.*

- `[A]`, where A is a string with different symbols, is a regular expression that denotes a language with all symbols from A.

> **Example 2.1.14** $L([xy]) = L(x|y) = \{x, y\}$.

- If A is a regular expression, then `A+` is a regular expression that equals `AA*`, i.e. it denotes almost the same language as `A*`, but without the empty string ε.

> **Example 2.1.15** `http://.+/[ABC][123]` *matches strings* `http://t/A1`, `http://example.com/C3`*, etc., but it does not match* `/C3` *or* `http:///C3`.

> **Example 2.1.16** `(Pawn|Queen) on [ABC][0123]+` *matches strings* `Pawn on B0`, `Queen on A12`*, etc., but it does not match* `Pawn on A` *or* `Pawn on 0`.

Regular expressions are a perfect tool to provide the search queries one may look for in the audio files. They are not only very expressive but also easy to use.

2.2 Weighted Finite State Transducers

In the following section we are going to introduce the *Weighted Finite State Transducers (WFSTs)*, which are widely used in the state-of-the-art automatic speech recognition systems (Mohri 1997; Mohri, Pereira, and Riley 2008) as well as in our work.

In Section 2.2.1 we will introduce a *finite state automata* (or *finite state machines*) with non-determinism in Section 2.2.2. Then, Section 2.2.3 will modify them in order to define *finite state transducers*, which will finally be expanded in Section 2.2.4 to *weighted finite state transducers*.

2.2.1 Finite state automata

A finite state automaton is an abstract device that accepts or rejects finite strings. It usually operates on a predefined alphabet Σ. One can think of such automaton as a machine that is in some state. It can read symbols from the input tape (string), which can change the state it is in. This abstract machine also has an abstract light that turns on when it is in an accepting state or off when it is not.

More formally, the *finite state automaton* is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of all states,
- Σ is a finite alphabet, i.e. a set of input symbols,
- δ ($\delta : Q \times \Sigma \to Q$) is a transition function,

- q_0 is an initial state,
- F is a set of accepting states.

The machine works as follows: it starts in the initial state q_0. It reads a symbol s from the input tape and changes the state to $\delta(q_0, s)$. Those operations are being repeated until there is nothing new to read. The machine is then in some state Q_l. If $Q_l \in F$, the string is being accepted. Otherwise, it is rejected.

Such automaton is usually presented as a graph. States (Q) are depicted as nodes (circles), transition function (δ) is drawn as arcs with the input symbol written above them. Initial state (g_0) is marked with a thicker border or an arrow and each final state (from the set F) with a double circle. An example of such a graph is presented in the figure 2.1.

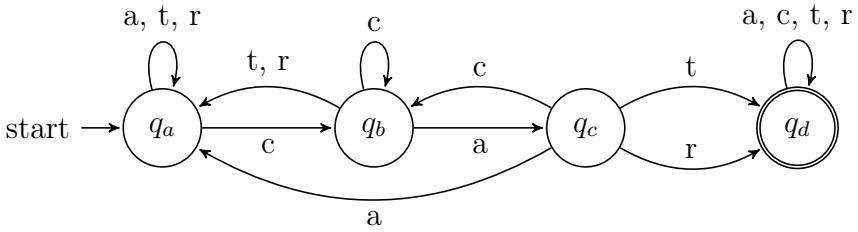

Figure 2.1: An example of a finite state automaton over the alphabet $\{a, c, t, r\}$ that accepts all strings containing a word "cat" or "car".

The δ transition function might also be partial, i.e. there might exist some state and symbol for which δ is undefined. On a graph we will just not include such arcs. If the machine needs to use such transition, it just rejects the word instantly and finishes computations. In the figure 2.2 an example of such situation is presented.

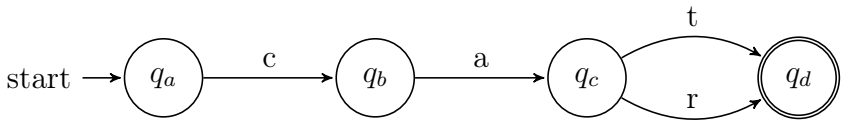

Figure 2.2: An example of a finite state automaton that accepts only two strings: "cat" and "car".

In the following sections we will build upon this idea and expand it with some additional features.

2.2.2 Non-determinism

The first modification of the finite automata, we will like to introduce, is non-determinism. The change affects only the transition function δ. In fact, it would

not be a function any more — it is going to be a relation. What does that actually change? From one state there can be two outgoing symbols. Therefore, there may also be two outcomes of reading symbol s in a state q. If that happens, we would take both paths and if any of them finishes in an accepting state, the whole string is accepted. We could also think about this in another way: instead of taking two paths we just take the one that will lead to the success.

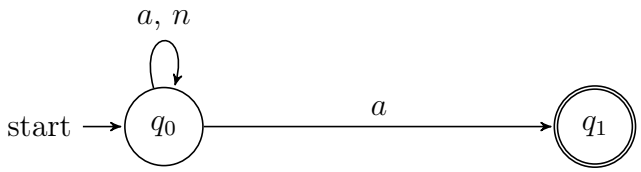

Figure 2.3: An example of a non-deterministic finite state automaton over the alphabet $\{a, n\}$.

An example of such machine is presented in the figure 2.3. It accepts any string that starts with some variable-length variation of a and n symbols, but it has to end with an a symbol. For example it would accept words *nana*, *aanaana*, *nananaaaa*, *aaaaaa*, but would not accept *an*, *aaaaaan*, *nanannan*, etc.

In order to make the automata easier to use, we add one more type of transition to the δ relation — ε-transitions. They can be traversed without reading any symbols.

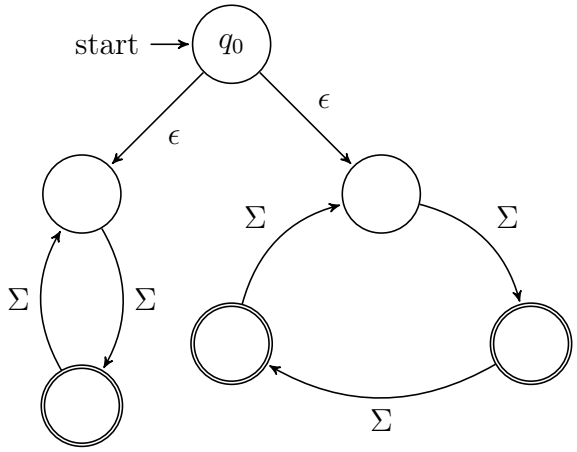

Figure 2.4: Non-deterministic final state automaton accepting all words of a length not divisible by six ($2 \cdot 3 = 6$).

Each non-deterministic automaton can be transformed to deterministic one, that accepts the same language. We call such process *determinisation*. Unfortunately, sometimes this is very costly. Examine the figure 2.4. It presents an

automaton that accepts all words of length not divisible by 6. We could construct a similar machine A_k that accepts all words of length not divisible by a product of all elements of $\mathcal{P}_k$, where $\mathcal{P}_k$ is a set of first k prime numbers. It would have $\sum \mathcal{P}_k$ states. After determinisation, its size would grow to $\prod \mathcal{P}_k$. It is easy to prove by contradiction using the pigeonhole principle. Therefore, the deterministic automaton recognizing the same language as a non-deterministic one, can be even exponentially bigger.

2.2.3 Finite state transducers

The finite state transducer, instead of having just one input tape, has two tapes: input and output. During a change of a state, some symbols (even from different alphabet) might be written to the second one.

More formally, the *finite state transducer* is a 6-tuple, $(Q, \Sigma_1, \Sigma_2, \delta, q_0, F)$, where:

- Q is a finite set of all states,
- Σ_1 is a finite alphabet, a set of **input** symbols,
- Σ_2 is also a finite alphabet, a set of **output** symbols,
- δ ($\delta : Q \times \Sigma_1 \to Q \times \Sigma_2$) is a transition relation,
- q_0 is an initial state,
- F is a set of accepting states.

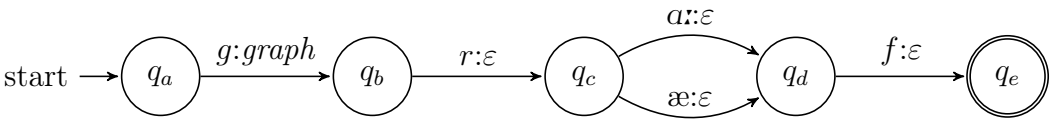

Figure 2.5: An example of a finite state transducer.

In the figure 2.5 an example of a finite state transducer is given. Its input alphabet (Σ_1) is a set of all phonemes, and the output alphabet (Σ_2) is a set of all words. On this example we can see the first practical usage of FSTs in speech recognition — the automaton that reads phonemes and outputs words formed by them.

2.2.4 Weighted finite state transducers

The final modification we are going to apply to our automata is the addition of weights. Each transition and each final state would have some value associated with it. This values, in our case, would usually be a floating point numbers that represent probabilities, but generally the weights might represent a variety of things, e.g. distances, costs, etc. The only property that is required from them is that, together with two binary operations, $\oplus$ and $\otimes$, they form a semiring.

When we traverse a path, weights of each arc are multiplied (using $\otimes$ operation). After reaching the final state, the accumulated value is first multiplied with the state's weight and then the result is added ($\oplus$) to the results from all other accepting nodes that the machine may finish in.

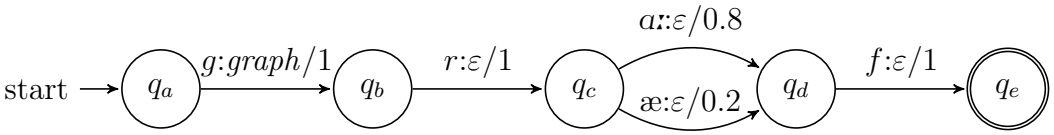

Figure 2.6: An example of a weighted finite state transducer.

In the figure 2.6 an example of a weighted finite state transducer is given. It does not only provide a word given its pronunciation, it also tells us how probable it is.

It turns out that the WFSTs are a very useful tool in the speech recognition. Up to now, we have been only using them as acceptors, but in fact they are much more powerful. For example, given a sequence of phonemes, they can generate a list of all possible words together with associated probabilities.

2.2.5 WFST composition

It is possible to define multiple operations on transducers: concatenation, union, intersection, negation, etc. For us, the most interesting would be the *composition*.

It is a binary operation working on transducers A and B. If the first one transduces string s_1 to z_1 with weight w_1 and the second one transduces s_2 to z_2 with weight w_2 then A composed with B transduces s_1 to z_2 with weight $w_1 \otimes w_2$.

Intuitively, one can think that in the beginning the input word is transduced by the first WFST and then, the outpt is provided as an input for the second one. The final outcome is the result of second transducer's computation.

In the figures 2.7 and 2.8 an example of transducers' composition is given. The first figure presents two WFSTs and the second one illustrates the composition of them.

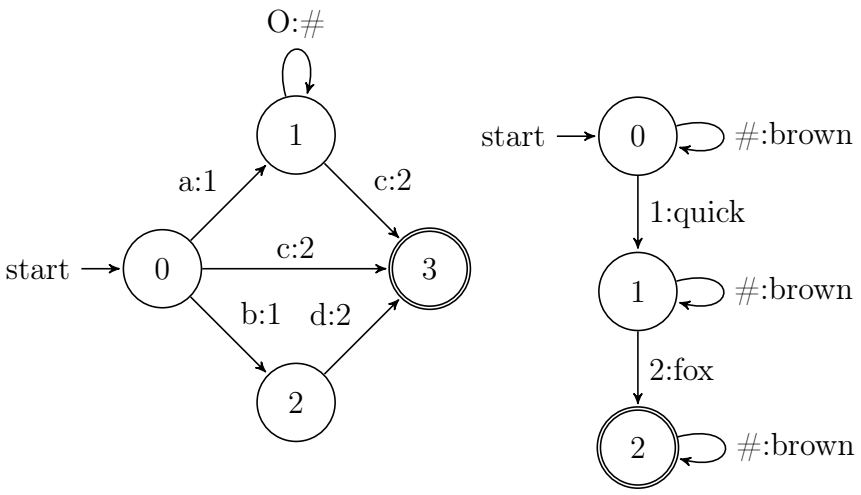

Figure 2.7: An example of two weighted finite state transducers.

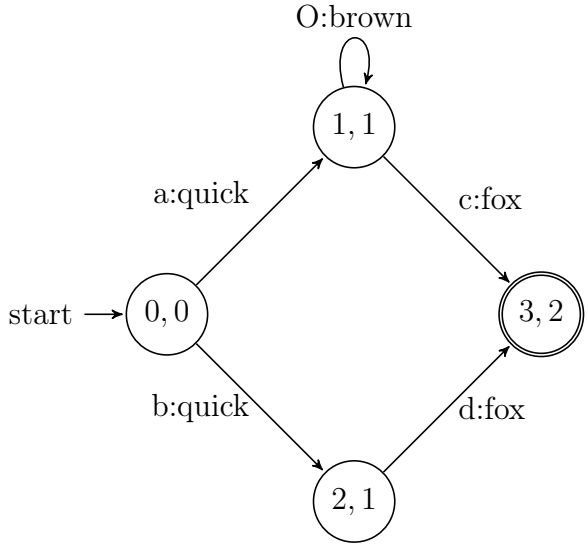

Figure 2.8: A composition of transducers from the figure 2.7.

2.3 WFSTs and regular expressions

A connection between the *WFSTs* and speech recognition has already been outlined. We will elaborate on it in Section 2.4. But what does it have to do with *regular expressions*? In fact, a lot. Both, regular expressions and *finite state automata*, recognize the same set of languages. Even more, for any expression r, it is possible to construct an automaton that identifies $L(r)$. In this section we will show how to do it.

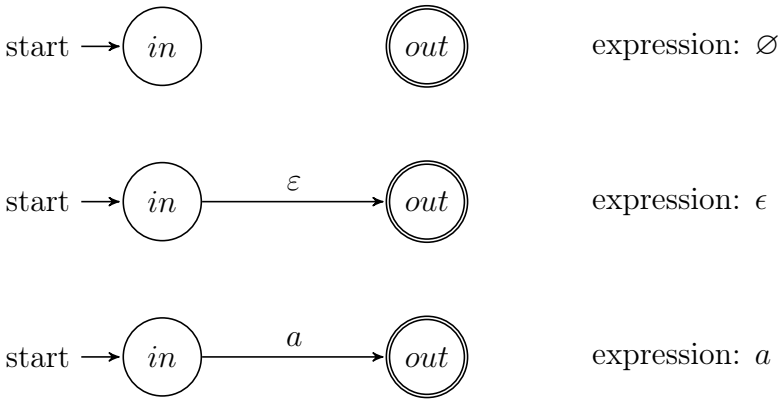

Figure 2.9: How to construct an automaton representing base regular expression.

Recall the definition of the regular expressions. It was inductive, with three base symbols and four recursive constructions. The method we will introduce, defines transducers for base symbols, and then describes a way to connect those transducers, in order to represent any expression. It is called a *Thompson's construction algorithm* (Ken Thompson 1968).

In the figure 2.9 transducers representing all base regular expressions are presented. Note how evident the distinction between $\varnothing$ and ε is. It is also important that all of those transducers have one starting and one accepting state. We will use this property in the inductive construction. In the description we will assume that both A and B are WFSTs that represent some regular expressions.

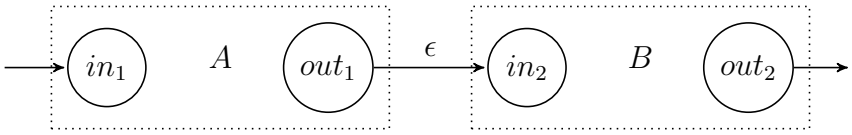

Figure 2.10: How to represent a concatenation of two regular expressions as an automaton.

The figure 2.10 depicts a contacenation of A and B. We just connect the *out*-node of A with the *in*-node of B with an ε-transition.

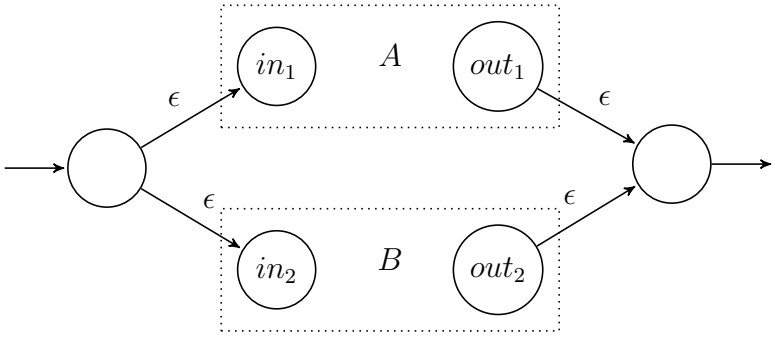

Figure 2.11: How to represent a sum of two regular expressions as an automaton.

The figure 2.11 presents a sum of A and B. We need to introduce two new states and connect the first one with both *in*-states and the second one with two *out*-states using an ε-transitions. That way a decoder might choose any of A and B transducers.

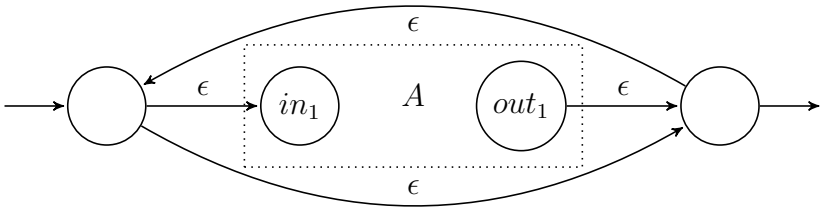

Figure 2.12: How to represent a Kleene star of a regular expression as an automaton.

The figure 2.12 presents a Kleene star of A. Even thought it is an unary operation, the resulting transducer is the most complicated one, because it generates multiple paths. Besides adding two new states, input and output, we also need to connect them properly with A, so the decoder can traverse the A graph multiple or zero times.

2.4 Speech recognition

In the following section we will describe some basic concepts of speech recognition that are essential to know in order to fully understand this work. It is not aiming to be a complete study of this subject. More information can be found in any natural language processing textbook, e.g. Jurafsky and Martin 2000.

The approach to speech recognition we have chosen is based on Mohri, Pereira, and Riley 2008. We split the process into four parts: `H`, `C`, `L` and `G`. Each of them

represents a different state of transformed data. The whole system takes as an input features extracted from audio data. They may for example consist of *MFCC* and *PLP* features. Each consecutive part would take as an input the result from its predecessor.

H part processes the extracted features and outputs context-dependent phonemes (one can think about them as phonemes with their surrounding).

C represents the context-dependency and given context-dependent phonemes outputs context-independent ones.

L is a *lexicon*. Provided a list of phonemes, it returns a list of words.

G is a *language model*. It accepts only sentences that make sense in the language in use, with weights representing probabilities of such sequence of words.

Each of those parts can be represented as a *WFST*. It is a very useful property. Thanks to it, almost the whole process of speech recognition can be implemented as a weighted finite state transducer.

The part that is especially important for us is a *language model* (G). It tells us how probable the given sequence of words is. Usually it is constructed from a *corpus*, i.e. a large set of texts used to statistically analyse a language.

There are multiple ways to create such models. One of them is an *n-gram model*. It assigns a probability to each combination of n words. So a *1-gram* would just be a set of all words with a probability of occurrence of each of them in the corpus. Analogically, a *2-gram* would be a set of all combinations of two words, etc. *1-grams* are often called *unigrams* and *2-grams* — *bigrams*.

2.5 Machine learning

Machine learning is a huge field of science that is evolving rapidly. In utilizes statistics in order to teach computers to achieve things they were not explicitly programmed to do. For the purpose of this work we will need to look only at one specific problem of this field — *classification*. It tries to assign one of some predefined categories to each test sample.

> **Example 2.5.1 (An example of classification problem)** *Given a set of audio recordings and a search pattern, decide if it occurs in the data.*
>
> *In this problem we have two classes: positive and negative. If a search pattern is in the recording, then it should belong to the positive class, otherwise — to the negative.*

A method to solve this problem would be proposed in Chapter 4. Currently, we will focus on some general techniques to evaluate classifiers. We could have just considered general effectiveness of it (how many positive samples were recognized as positive), but it would not be very informative for us. A classifier that always returns positive answer would score 100%. So we need to use something more robust, that would take into the account not only the number of correctly classified samples, but also the number of incorrectly reported ones.

In order to do that, let us first consider some options regarding good and bad classifications. There are actually four options.

True positives (TP) — samples, that were correctly identified as positive.

False positives (FP) — samples, that were incorrectly identified as positive.

True negatives (TN) — samples, that were correctly identified as negative.

False negatives (FN) — samples, that were incorrectly identified as negative.

Actually, it is pretty hard to come up with a method to objectively score a classifier. It heavily depends on the problem at hand. For example, if we want to detect cancer, we would rather not have any false negatives, but some false positives are acceptable. On the other hand, if we have written an algorithm that controls a rifle and shoots bad people, we would want it to be perfectly sure that a person is not good before killing them. Therefore, false positives are not acceptable.

Fortunately, classifiers often have some adjustable parameter, that let us gradually exchange some false positives by false negatives and vice versa. But first, we will introduce more definitions.

Precision is a ratio of true positives to all samples that were classified as positives. Precision $= \frac{TP}{TP+FP}$.

Recall (also Sensitivity, true positive rate (TPR)) is a ratio of true positives to all samples, that should be positive. Recall $= \frac{TP}{TP+FN}$.

Fall-out (also False positive rate (FPR)) is a ratio of false positives to all samples, that should be negative. Fall-out $= \frac{FP}{TN+FP}$.

We might use the parameter mentioned in the previous paragraph to run the classifier multiple times and compare those executions. One of a ways to do that is a diagram called a *receiver operating characteristic* (ROC) diagram. On the y axis it has a true positive rate and on the x axis a false positive rate. A classifier that just randomly assigns classes to samples with some probability, that can be

adjusted, would always end up on the $y = x$ line. If some point is below this line, it means that inverting program's decisions would actually give us better results. We would like our classifier to be as far to the left and to the top as possible. In the figure 2.13 an example of such ROC diagram is given with the results of four different classifiers marked on it.

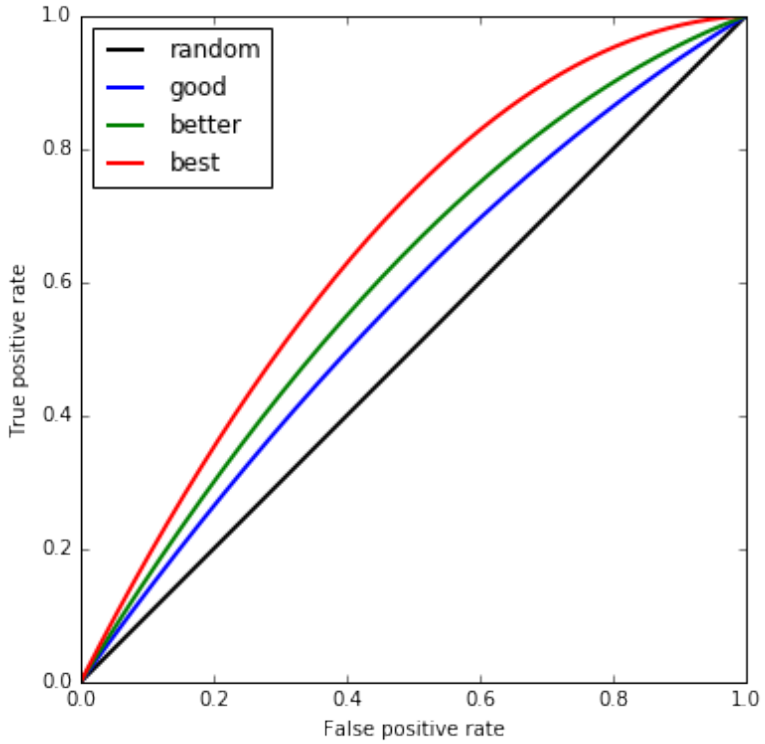

Figure 2.13: An example of a ROC diagram with four classifiers marked with different colors.

Chapter 3

Technical background

In the following chapter we will introduce software that was used in order to create this thesis. Section 3.1 describes the *Kaldi* toolkit and *OpenFST*. Both of them are used by us in our speech recognition pipeline. In the next section (3.2) a bigger project that this thesis is a part of, is introduced. Section 3.3 describes the application made in order to gather more data. At the end, in Section 3.4, programs and libraries used by us are presented.

3.1 Kaldi and OpenFST

Kaldi (Povey et al. 2011) is a speech recognition toolkit written mainly in *C++* and *Bash*. It is licensed under the *Apache License v2.0*, which means that it is free for use and edit, as long as the original work is cited. *Kaldi* is intended to be used by researchers, therefore it is easily modifiable and extendable. In practice, it is a set of small programs and *Bash* scripts that allow us to construct a working speech recognition pipeline.

Kaldi is strongly integrated with the *OpenFst* toolkit (Allauzen et al. 2007). This library implements a way to construct the *weighted finite-state transducers* together with basic operations on them. It is designed with the efficiency in mind. Additionally, some utility programs are provided in order to analyse and process transducers, which makes the work with this library much more pleasurable.

3.2 Project AudioScope

This thesis is a part of a larger project, named AudioScope. The aim of this project is to create an automatic search system, working on audio files. It is developed by a science consortium formed by the *University of Wrocław*, the *Wrocław University*

of Science and Technology and the *Neurosoft* company. The project started at the 1st of April 2015 and is due to finish at the 31st of March 2017.

The project is funded by the *Applied Research Programme* of the *National Centre for Research and Development*. It supports various science institutions and industries in their practical studies.

The system receives as an input a set of speech recordings in Polish. It should mark all fragments that contain a searched phrase. In order to achieve the goals, we have decided to use the *Kaldi* toolkit, with various parts modified by different teams and people. The team at the *University of Wrocław* mainly focuses its work on the *Language Models* and data acquisition.

Essential datasets in Polish are very hard to get. We have struggled to acquire both, good corpora and audio recordings with transcriptions. We did not limit ourselves to just one corpus or one dataset. The decision has been made to find as any different sources as possible. Later, we merge collected data together and run tests on different combinations.

In terms of corpora, one of the best sets that we were able to find, is the National Corpus of Polish (NKJP, `http://nkjp.pl`) dataset. It was created at the *University of Łódź* and published for everyone to use. We have also made a couple of corpora on our own. Among others, one was created from the transcriptions of audio files that we acquired, another from *Wikipedia*.

One of the recurring problems, that was often mentioned on our meetings, was where to find datasets of transcribed audio files. In the beginning, we have started with a collection of audiobooks. It was a great starter, because hundreds of hours of data could be easily found. Unfortunately, the way speakers were reading the text was far from natural. We have also used some smaller sets of audio files and even tried to utilise some recordings in different languages.

3.3 Audioup

Because it is so hard to acquire transcribed audio recordings in Polish, we decided to make an application that will allow us to create such kind of data. It is called *Audioup*. It allows people to easily record themselves reading some predefined texts. We have chosen fragments from the *Winnie-the-Pooh* book. It is easy to read, interesting for children as well as adults and does not contain difficult language. The application is working as a web page and utilizes the newest technologies in order to capture voice, encode it and send it to the server, without the use of any plug-ins or additional programs. It was written in *HTML5* and *JavaScript*. It uses technologies such as *application cache*, *local storage*, *audiocontext*.

It was designed with the ease of use as the priority. Only three clicks of the mouse are needed to start recording. The text on the site was also reduced to

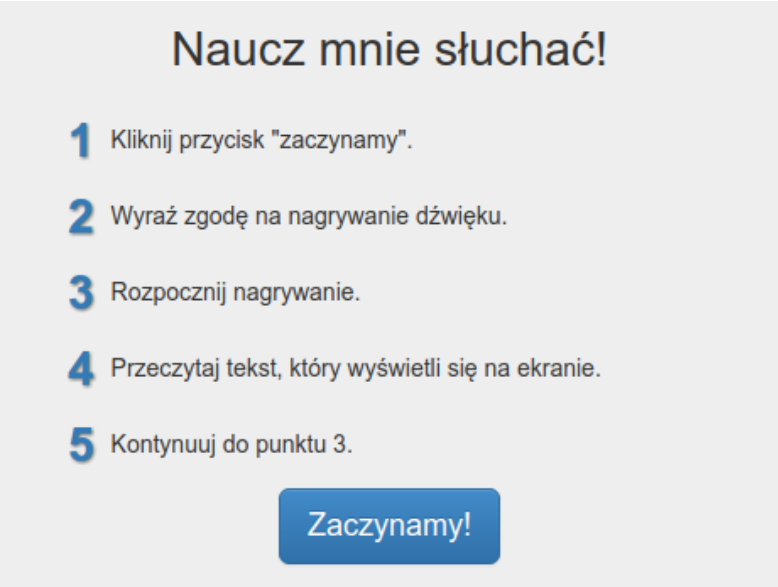

Figure 3.1: A screenshot of the *Audioup* program showing the introduction screen.

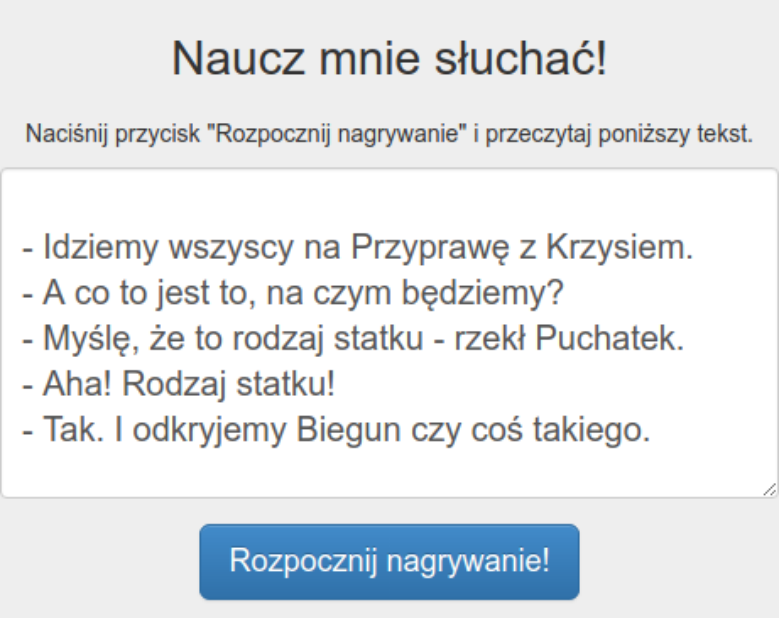

Figure 3.2: A screenshot of the *Audioup* program showing the screen displayed just before starting recording.

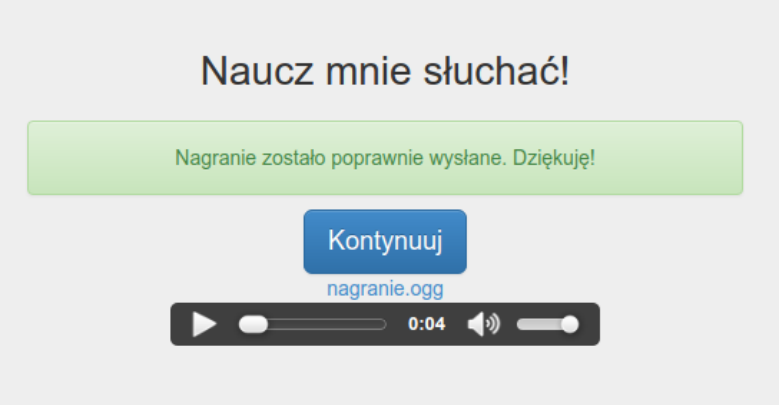

Figure 3.3: A screenshot of the *Audioup* program showing the screen displayed after sending the recording to the server.

Figure 3.4: A form to input additional details in the *Audioup* program.

the minimum. Additionally, the web page is very accessible. Its responsive design makes it usable on a variety of devices like smartphones, tablets and computers.

In order to make the recording possible almost everywhere, the first time the web page is visited, it stores itself on a guest's computer. After that, it is available to use and record audio even when there is no Internet connection. The data is temporarily stored in the browser until it could be sent to the server.

In the figures 3.1, 3.2 an 3.3 a basic usage of the application is presented. At the beginning a user is greeted with instructions how to use the application. After clicking the "start" button they are presented a text to read aloud. They start the recording and read the text. Then the gathered data would be automatically sent to the server. The user can then listen to it or download it and start the process again.

The figure 3.4 presents a form used to give some additional information about the speaker. They are stored together with the recorded data and the transcriptions.

3.4 Utilities

The project also utilised a lot of small scripts and simple programs in order to achieve its goal. Most of them were written in *Python* and *Bash*. The results of experiments were gathered and processed using *Jupyter notebook* (Pérez and Granger 2007). Diagrams were potted with *Matplotlib* (Hunter 2007). Graphs were created using *TikZ* LaTeX library (Tantau 2013) and the *Linux dot* utility.

Chapter 4

Searching phrases in audio files

The main goal of this thesis is to introduce a way to search for phrases in audio files. Recall Section 1.4, where the problem has been formulated. In the following chapter we are going to describe our solution to this problem.

In Section 4.1 we are going to describe a naive approach that seems obvious after getting in touch with the problem and speech recognition for the first time. We will also argue why this solution does not work well. Next, in Section 4.2, our solution to the problem will be described. At the end, in Section 4.3, some difficulties that came up during the work are outlined.

4.1 Naive approach

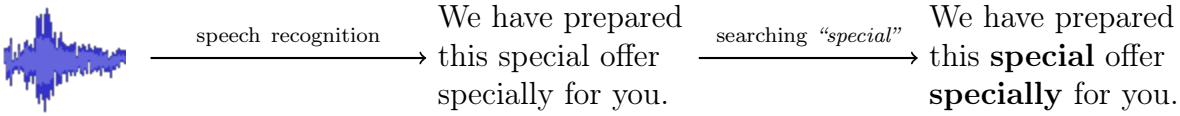

Figure 4.1: A naive approach to searching in audio data using speech recognition.

A naive approach that one might think about when given the above problem might be similar to the one presented in the figure 4.1. We get the input audio file and process it using some speech recognition software in order to obtain text output. Then, we just search for a given phrase in this text using any standard algorithm. At the end we obtain the same text with occurrences of the searched phrase tagged in it. But is it really that simple?

In reality speech recognition software often makes mistakes and recognize different words than actually spoken. In fact even the best programs struggle in

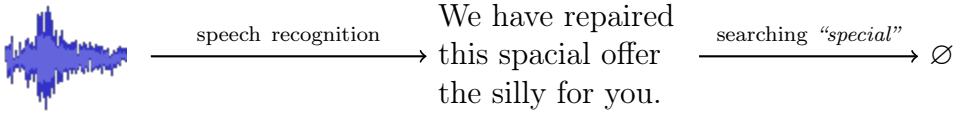

Figure 4.2: Why the naive solution is not working.

order to get the word error rate (percentage of incorrectly recognized words) below 10% (Geoffrey Hinton et al. 2012). Because of that the situation might as well look as in the figure 4.2. Both words we were looking for were incorrectly recognized as words that sound very similar. At the end, the search algorithm did not return any results.

4.2 Our solution

A solution proposed by us, works in a completely different manner. We decided to examine a way in which speech recognition algorithms work and modify them, such that we will search for a phrase during speech recognition, not after that. This effectively merges the two stages (speech recognition and searching) together.

Our idea is based on weighting the *language model* in order to make all occurrences of the searched pattern more probable. In the example 4.2.1 the intuition how such a model could work is presented.

> **Example 4.2.1** *We are given a problem consisting of audio data and a text pattern to find in this data. Let's assume there is some speech recognition software that uses both language and acoustic models. For the sake of simplicity let us assume the language model is uniform, i.e. all words are equally probable.*
>
> *We expect, that the given pattern really exists in audio data, but the speech recognition software does not recognize it properly, i.e. the acoustic model assigns greater probability to some other, similar pattern in place where the searched one should occur.*
>
> *But the pattern we are searching for must have also been given high probability by the acoustic model, because we assume that acoustic model must give high probability to sentences that were actually spoken.*
>
> *Now let's consider the same speech recognition software, but with a language model that assigns a slightly higher probability to the text pattern we want to find. That way even though the phrase we were searching for was not the most probable choice of the acoustic model, it will be included in the final result.*

There are however two things to consider. First, how to implement weighting

so it is efficient and easy to use. Second, how to choose the value by which phrases should be weighted.

4.2.1 How to implement weighting?

At the beginning of explaining how we have implemented weighting we need to state the assumptions we have made.

1. We do not have any knowledge about the structure of a language model. The solution should work with any models.

2. The goal is to be able to search for multiple phrases at the same time. The reason behind this requirement is that we are often looking for a base form of some word and we want to find all occurrences of this expression. For example, when searching for *"kill"*, we would also like to get results for *"killing"*, *"kills"*, etc.

3. Additionally, the phrases we are going to search for, might be of variable length, like *"going to school"* or *"Can I have three ounces of water please"*. It would be an advantage, if we could put placeholders in the middle of sentences, e.g. *"Suspect <name> is guilty"*.

Our solution is built upon the idea of representing each step of speech recognition as a *WFST* (described in the Section 2.2) introduced in Mohri, Pereira, and Riley 2008 and the regular expressions as *WFSTs*.

As we know from Section 2.3, any regular expression can be represented as a *finite state automaton*, which is also a *WFST*. So we introduce a transducer `R` that is going to act as the regular expression in the decoding process. We are going to compose `R` with the `HCLG` graph (described in Section 2.4).

There are two reasonable ways to do that: `HCRLG` and `HCLGR`. The result of both of those compositions is the same, but the input and output symbols of the regular expression (`R`) graph are different. In the first variant, regular expressions would work on phonemes and output phonemes. This is a pretty intuitive way to define them. However, the second variant (with `R` at the end) has a useful property — it lets us define the output symbols any way we want. Finally, we decided to use the `HCLGR` solution. Thanks to that we were able to mark the found occurrences with special symbols that let us easily localize the place where the regular expression was found.

In the figures 4.3 and 4.4 two kinds of `R` graphs implemented by us have been presented. The first one is used to find all occurrences of the given phrase. It reduces the weight of associated regular expression, so it is more probable to find it in the audio data. The second one, on the other hand, ensures that the phrase

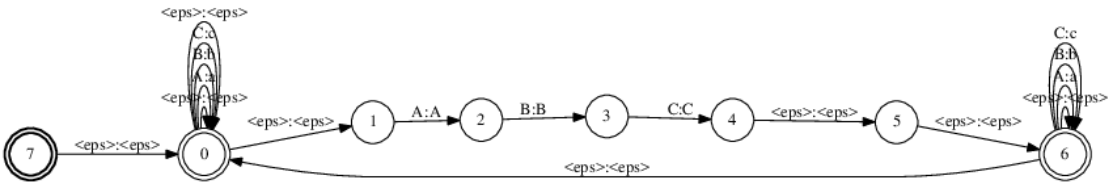

Figure 4.3: An example of WFST that is used to find all occurrences of phrase "ABC" in a text over the alphabet {A, B, C}. In order to make it more readable it is not minimized and thus contains some unnecessary *<eps>* arcs. Note the arc from state 6 to state 0 that may be used in order to traverse the path with regular expression multiple times.

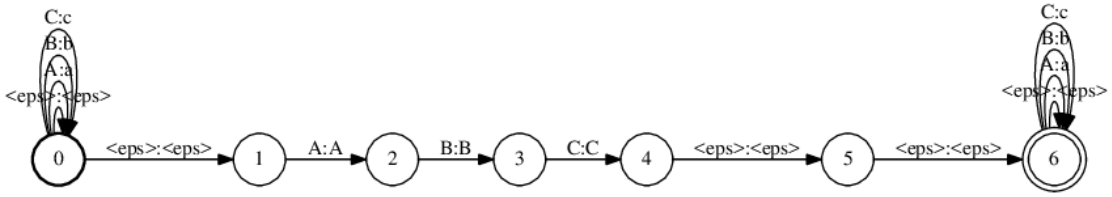

Figure 4.4: An example of WFST that is used to find just one occurrence of the regular expression "ABC" in a text over the alphabet {A, B, C}. In order to make it more readable it is not minimized and thus contains some unnecessary *<eps>* arcs.

is found somewhere in the recording. It might be useful, for example, when we know that something has been said and we want to precisely locate it in the input data.

4.2.2 How to choose the regular expression's weight value?

The last thing that is left, is to choose the weight value, i.e. the cost of the arcs that represent the search phrase. After the regular expression is composed with an `HCLG` graph, this value would be multiplied ($\otimes$) by appropriate arc's weights in the language model, effectively reducing the traversal cost for those words.

Example 4.2.2 (Decoding of the same recording with different weights)

transcription: *kolejne wartości kolejne ideały które miejmy nadzieję że będą przyświecać poszczególnym pokoleniom*

weight 0**:** *p a l e j n e w a r a t o si ci i k o l e ni a i d e ng a ll e k t u l e ni e j n e n a dzi e rz e w e n a m p sz y si f j e c a ci p o cz e b o l e m p o k o l e ni o*

weight −2**:** *p a l e j n e w a r a t o si ci i k o l e ni a I D E A <re_found> ll e p ll n e m j e n e n a dzi e rz e w e n a m p sz y si f j e c a ci p o c z e b o l e m o k o l e ni o*

In the example 4.2.2 we have presented an output of the speech recognition algorithm for the same input, but with different regular expression weights. The phrases to look for in both cases are identical and represent all forms of the word "idea" in Polish. All occurrences of the searched phrase are marked with upper-case letters and "<re_found>" symbol appended just after them. The search pattern was present in the test sentence, but it was not found when the weight of the regular expression was 0. Changing it to −2 allowed locating it in the correct place, but in a slightly different form.

Why is this parameter so significant and how should it be set? It depends on the situation. Sometimes, when it is not important to find each occurrence of the pattern, but we do not want to tag some data incorrectly, it would be better to use a greater weight value. On the other hand, when it is essential to locate the phrase each time it appeared and some additional incorrect detections are tolerable the value of this parameter should be small. Later it might be possible to further filter out the false results using different algorithms or sometimes even manually.

Because this parameter has such a big impact on results and it depends so much on a use-case, we decided to let the user adjust it. If we needed to construct a general algorithm and specify a default value for this weight, we would probably pick the one with the greatest *F1 score* on a training data.

4.3 Difficulties

During the development of this project we have often found ourselves in a place, where theory seems to work very well, the input data looks good, but the algorithm does not compute any answer at all or produces unsatisfactory results. Often those problems were caused by the *FSTs* growing rapidly. For example, during determinization, automata usually grow quadratically fast.

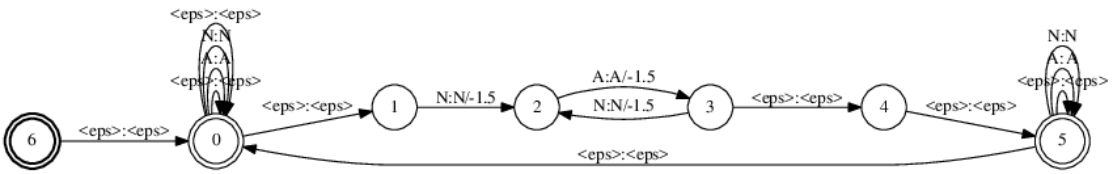

Figure 4.5: An example of *WFST* that represents regular expression "`.*((NA)+)*.*`".

In fact, some transducers cannot be determinized at all. In the figure 4.5 an example of such WFST is given. It represents a very easy regular expression: "`.*((NA)+)*.*`". Any equal determenistic transducer cannot exist, because the same input word can generate two different outputs. The word `NA` can be transformed to `NA` with weight -3 (using all states), or 0 (by looping in the state 0).

Fortunately, the determinization of `HCLGR` graph is only needed to speed up the process of decoding. However, we did not want to skip it completely. The final solution determinizes the regular expression before adding `.*` parts around it. The examples of graphs obtained using this technique will be presented in the next chapter.

Chapter 5

Experiments

In this chapter we will introduce experiments undertaken during the work on this thesis. Section 5.1 is devoted to describe the datasets that our solution was tested on. Section 5.2 outlines our approach to the experiments, their detailed implementation and results.

5.1 Datasets

During the work on this thesis we have conducted multiple practical experiments in order to test whether our assumptions are correct. At the beginning they mostly involved testing different versions of R graphs and manually examining the outcome on different datasets.

5.1.1 Digits dataset

We have even prepared a special dataset, named "Digits". It contained over 7 minutes of recordings created by Paweł Florczuk and Michał Barciś. Both speakers were pronouncing random digits in two ways: fast and slow. Resulting sets were tagged with "hard" and "easy" labels. This data has two important properties that made it ideal to start the research with.

- It is relatively small. Thanks to that all experiments were finished quickly and the results were easy to analyse.
- There are only ten words. It makes the data easier to decode and thus provides better results, which are desirable at the early stages.

After achieving satisfactory outcomes on this dataset with a language model containing ten words we decided to make it a bit harder. We did not want to jump

straight into a big dataset, but wanted to test how our solution would work with larger graphs. Therefore, we decided to use a bigger language model. Instead of recognizing actual words, they have been replaced by phonemes, i.e. now we will receive results in a form of "d w a j e d e n p j e ni c" in place of "dwa jeden pięć". That way, the much bigger graph has been constructed, but the test data was still pretty simple. In order to make the experiment more realistic, instead of assigning uniform probabilities to phonemes, we constructed a bigram language model on them. This way we have faced a slightly harder problem, but it was still possible to assess it manually. Each experiment also run relatively fast, it took around ten minutes to decode the whole dataset. It made working with this data much quicker and efficient.

5.1.2 Experts

The second dataset we have used is named "Experts". It is one of the most often used test sets in the *Audioscope* project. It consists of 2 hours and 22 minutes of audio recordings made probably in a school. Speakers usually explain some secondary-level subjects. Recordings are divided into 1857 utterances, each of them are a couple of seconds long and correspond to around one sentence.

It is not possible to analyse such big dataset manually. Therefore, we have designed a test that allows us to easily asses the effectiveness of our solution. It is not tailored in any way to our approach. In fact, it was proposed, even before this project started, as a generic way to rate solutions to a problem of searching phrases in audio data.

The test contains 1863 search patterns represented as a base form together with a set of related words. For example, base form *bohater* ("hero" in Polish) is related to 10 words: *bohater*, *bohatera*, *bohaterami*, *bohaterom*, *bohaterowi*, *bohaterowie*, *bohaterów*, *bohaterem*, *bohaterze* and *bohaterzy*. For each of the records in the "Experts" dataset it is marked whether it contains each of the search patterns or not. The information is only boolean, the place where the phrase is found is not marked. The goal of this test is just to tag each recording with all found patterns.

5.2 Experiments description and results

Our approach to the test introduced in the previous section, is based on the idea described in Chapter 4. We will try to find each phrase individually during decoding and mark them in the output data with a unique symbol *<re_found>*. The algorithm would be realised by lowering the weights of the searched patterns in the language model. We achieve it by composing regular expressions with the *HCLG* graph.

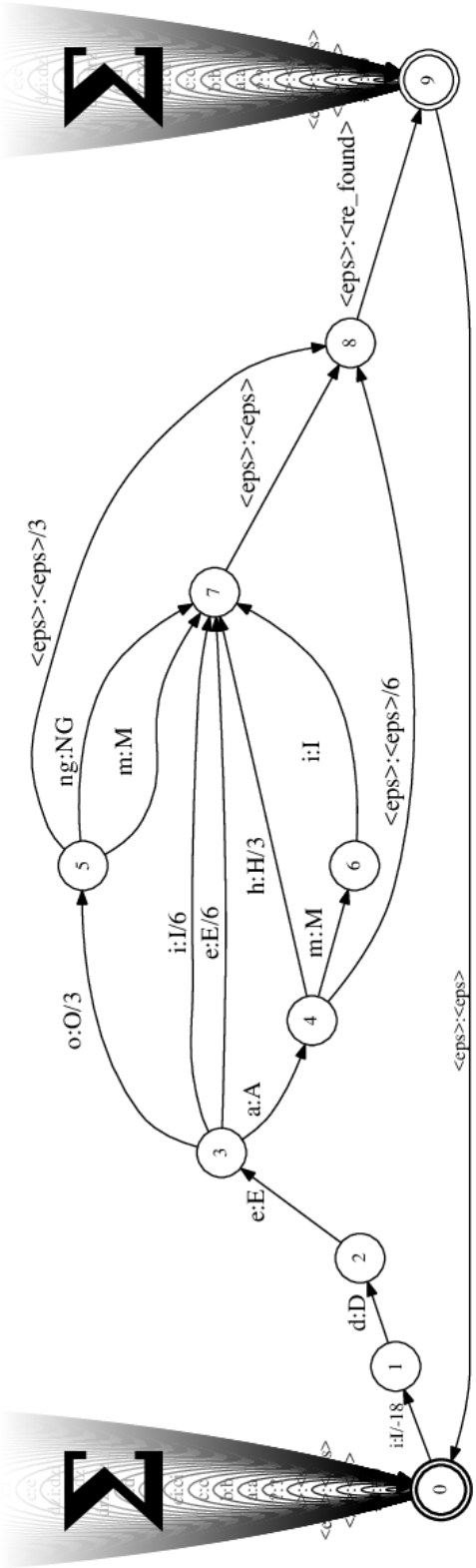

Figure 5.1: An example of an *FST* representing a regular expression used in process of speech recognition.

In the figure 5.1 an example of transducer used as a search pattern for base form "idea" is presented. The weights of arcs in the phrase are set to -3, but some pre computations have been made in order to push weights as close as possible to the starting node, on all paths. All found occurrences of the expression are marked with upper-case letters.

> **Example 5.2.1**
> *A decoding of the recording transcribed as*
>
> *"kolejne wartości kolejne* ***idea****ły które miejmy nadzieję że będą przy świecać poszczególnym pokoleniom"*
> *with the* `R` *graph presented in the figure 5.1 is:*
>
> *"p a l e j n e w a r a t o si ci i k o l e ni a I D E A <re_found> ll e p ll n e m j e n a dzi e rz e w e n a m p sz y si f j e c a ci p o c z e b o l e m p o k o l e ni o".*

In the example 5.2.1 an actual decoding is presented. As we can see, the searched pattern ("idea") was found. The result might not have been exactly what we have been looking for, because it is a part of a different word. But it gives a good intuition of the construction of the whole system. In fact, in the current setup, it is not possible to distinguish such false positives, because we have no notion of a word. In the Chapter 6 we will give some ideas how this approach could be improved.

Our goal is to plot the *receiver operating characteristic* (ROC) diagram of the classifier. It illustrates the true positive rate against the false positive rate on the test dataset for different `R` graph's weights. For the reasons described in Section 4.2.2, this diagram is suitable to present the efficiency of our solution. However, in order to compare it with other programs, we computed the *F1 score* for each point in the ROC diagram and chose the highest value.

The "Experts" dataset is huge. Specifically, there is a lot of patterns we have to search for. Our solution was not designed to serve this purpose. We mainly aimed at the highest effectiveness and the ability to easily control the ratios of false and true positives. Therefore, this test was impractical for our solution. It would take more than two months on the available hardware to produce all needed results. We decided not to run the whole test, but just approximate its result.

In order to draw the ROC diagram we need two parameters: *false positive rate* (FPR) and *true positive rate* (TPR). Due to the nature of data, where most of the results are negative, it is fairly easy to approximate the false positive rate precisely. In order to estimate both rates we have determined their confidence intervals. In order to achieve that we have used the *Wilson* method (Wallis 2013) for binomial distribution.

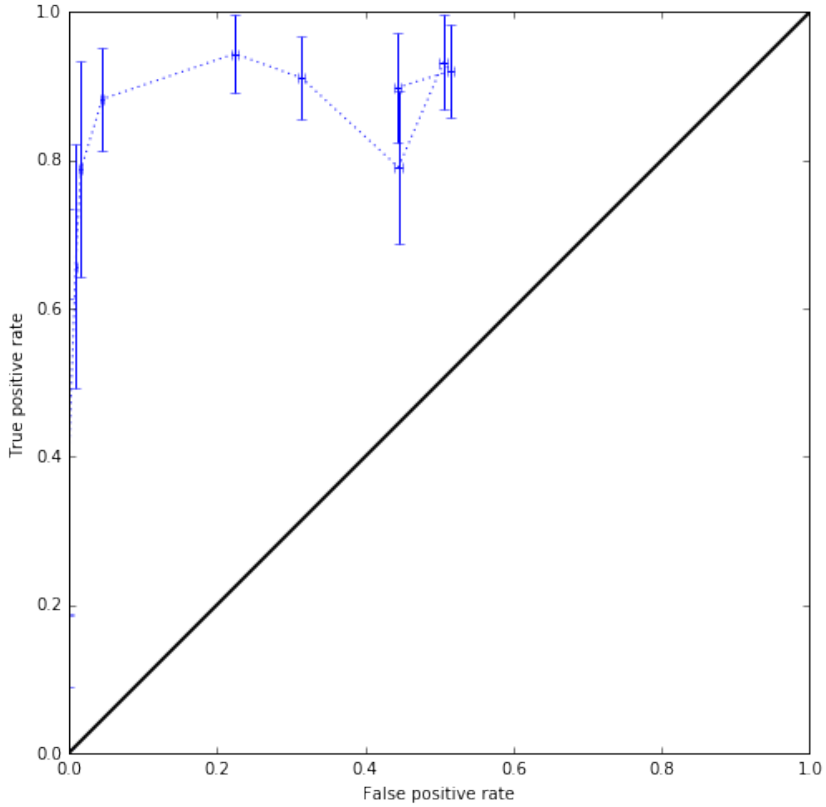

Figure 5.2: ROC diagram of the classifier implemented using presented concepts.

In the figure 5.2 the *ROC* diagram of our classifier is presented. The right part of it stands out from the rest. For a false positive rate greater than 0.2 the true positive rate often drops for a lower weight value and overall the results seem random. This is due to the way the decoder in *Kaldi* works.

> **Example 5.2.2** *Let us consider a case when we want to find all occurrences of the phoneme sequence "*`P O SZ CZ E G U L N Y M`*" in the test data, with a very low weight (e.g.* -5*). The decoder in Kaldi does not always compute the exact result, because it is very costly. It just greedily considers some number of the best options. Because the cost of our search phrase is so low, it tries hard to fit it to the data. It does not "know" how long each phoneme is, so it might for example consume even a couple of original words and recognize them as the single phoneme* `P`*. Then, the same thing can happen with the phoneme* `O`*, etc. At some point, the recording finishes, but all the options that the decoder is considering, might be in states that are somewhere in the middle of the search phrase.*
>
> *None of them is a final state, so no result should be returned. However, in practice, if the decoder cannot find a path that ends with a final state, it just returns its best guess. Unfortunately, because only a fraction of the search pattern was found, it is not marked as a positive classification.*

Fortunately, it is not a big problem for us. The value of false positive rate greater than 0.2 means, that in each fifth phrase, the search pattern would be found. In reality, they occur much less frequently. So a program that has such high `FPR` would be impractical.

The most interesting portion of the ROC diagram is presented in the figure 5.3. It shows only results with false positive rate lower than 10%. The first classifier (with weight set to 0) has a `TPR` of about 5% with `FPR` close to 0 (0.01%). Then, each subsequent program with a lower weight value achieves greater true positive rates, for a cost of a false positive rate.

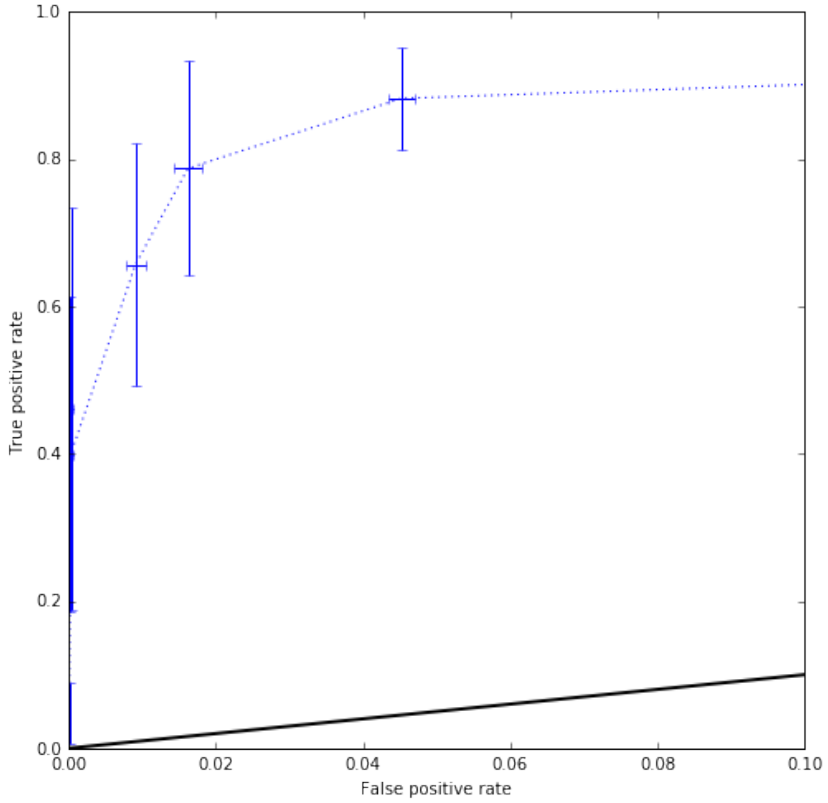

Figure 5.3: ROC diagram for values of FPR between 0 and 0.1.

Chapter 6

Conclusions

At the begining of this Chapter, in the Section 6.1 we will give some ideas for future work based on this project. The thesis is finished (Section 6.2) with a short summary.

6.1 Future work

Even though this thesis presented a final solution, that is robust enough to be used in different projects, it still leaves some field for improvements. Therefore, in this section we will outline some ideas for future work.

6.1.1 Language models based on words

Up to now, experiments have only been conducted with the language model, where phonemes were used as words. The solution with actual words being used instead, might provide better results. This approach has not been tried by us, because a phoneme-based language model is easier to train and test regular expressions based on it.

6.1.2 Out-of-vocabulary symbols in regular expressions

The idea described in the Section 6.1.1 might be sometimes problematic, because we could recognise and search only for the words that are present in the dictionary. In practical use cases, we often want to look for some unique names, that we have never seen before. In the current implementation it is problematic, because regular expressions operate on symbols from the alphabet. One possible solution is to introduce additional symbol $\mathcal{S}$ for such names. Then, before composing such expression with the L graph, a phoneme-based language model would be put in

place of $\mathcal{S}$. Such model could even be trained, for example, to detect names of cities, people, companies, etc.

Such solution would be very intuitive for end users. One could just define expressions like "`<person_name> is considered guilty`", or "`passenger train from <city> to <city> via (<city>)+`".

6.1.3 Performance

In this thesis we focused on achieving the highest scores, not on the efficiency. In fact, our solution is relatively slow. It is mainly due to the slowness of *Kaldi* itself and the need to run the whole process of audio recognition for each searched phrase. For a big dataset, or a huge number of different queries, this might be impractical.

A potential solution to this problem might involve data indexing. We could pre-process the data and for each fixed-length fragment save some number of the most probable words, that occurred in it. Then, when the user asks for some query, we will not use our solution on the whole dataset, but only on parts, where some words from the query are present.

Another approach to deal with performance issues might involve tweaking *Kaldi*'s decoder to work more efficiently. Of course, the results might get worse because of that and some experiments will be necessary to find the optimal parameters.

6.1.4 Changing the weight of regular expression depending on its content

The last idea of further studies, we want to present in this thesis, involves the weight of the regular expression. Currently, each word in the regular expression is being given the same cost. But the words are very different. Some are essential to understand the sense of the sentence, some are often skipped, some can be substituted by different ones. Because of that the classification results could be improved by changing the weight of regular expressions cleverly. We could for example increase the cost of short words, or the not popular ones.

6.2 Final words

Natural language processing has always been a fascinating and popular field of science. People wanted the computers to be able to understand them in the most natural way — using voice. This study has tackled one of many problems of NLP,

searching for phrases in speech recordings. This problem is still far from being solved, as well as the more general problem of the speech recognition.

However, this study has proven that the situation is not tragic. Even though we are still far from implementing a perfect speech recognition software, it gives us enough knowledge to be able to search for phrases efficiently. Moreover, the proposed solution gives the ability to easily change its sensitivity, making it robust and adjustable for various purposes.

Contributions

This thesis is not only built upon some existing software, but it is also a part of the *AudioScope* project. Therefore, it might sometimes be hard to distinguish what was made by me, what by my colleagues from the team, and what was used as-is. Because of that, in this section, I will enumerate my contributions.

1. *AudioUp* — the web application made to collect large number of samples of speech recordings. Described in the Section 3.3. I have designed and implemented the whole program and set up the server to run it.

2. A program generating FSTs from regular expressions. It implements the *Thompson's construction algorithm*(Ken Thompson 1968) and takes as an input a regular expression and a list of allowed symbols. It can output an FST representing this expression.

3. Modified *Kaldi* scripts used in *AudioScope* in order to compose the language model with previously genereted FSTs representing regular expresions. The idea of those modifications is described in the Chapter 4.

4. Scripts needed to run the test and analyse its efficiency. It included data preparation, running the *Kaldi* scripts and generating diagrams based on the results of computiations. The experiment is described in the Chapter 5.

Bibliography

Allauzen, Cyril et al. (2007). "OpenFst: A General and Efficient Weighted Finite-State Transducer Library". In: *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*. Vol. 4783. Lecture Notes in Computer Science. `http://www.openfst.org`. Springer, pp. 11–23.

FirstSounds.org. URL: `http://www.firstsounds.org/`.

Geoffrey Hinton et al. (2012). "Deep Neural Networks for Acoustic Modeling in Speech Recognition". In:

Hopcroft, John E. and Jeffrey D. Ullman (1979). "Introduction to Automata Theory, Languages and Computation". In: pp. 83–113.

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3, pp. 90–95.

Jurafsky, Daniel and James H. Martin (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0130950696.

Ken Thompson (1968). "Programming Techniques: Regular expression search algorithm". In: Communications of the ACM.11, pp. 419–422. DOI: `10.1145/363347.363387`.

Mohri, Mehryar (1997). "Finite-state Transducers in Language and Speech Processing". In: *Comput. Linguist.* 23.2, pp. 269–311. ISSN: 0891-2017. URL: `http://dl.acm.org/citation.cfm?id=972695.972698`.

Mohri, Mehryar, Fernando Pereira, and Michael Riley (2008). "Speech recognition with weighted finite-state transducers". In: *Springer Handbook of Speech Processing*. Springer, pp. 559–584.

Pérez, Fernando and Brian E. Granger (2007). "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3, pp. 21–29. ISSN: 1521-9615. DOI: `10.1109/MCSE.2007.53`. URL: `http://ipython.org`.

Povey, Daniel et al. (2011). "The Kaldi Speech Recognition Toolkit". In: *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Catalog

No.: CFP11SRW-USB. Hilton Waikoloa Village, Big Island, Hawaii, US: IEEE Signal Processing Society.

Tantau, Till (2013). *The TikZ and PGF Packages. Manual for version 3.0.0.* URL: http://sourceforge.net/projects/pgf/.

Wallis, Sean A. (2013). "Binomial confidence intervals and contingency tests: mathematical fundamentals and the evaluation of alternative methods". In: Journal of Quantitative Linguistics.20, pp. 178–208. DOI: 10.1080/09296174.2013.799918.