BACHELOR THESIS

Jan Výkruta

Automatic Expressive Reading

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: Mgr. Jan Hajič, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2019
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ........ date .......... signature of the author
Title: Automatic Expressive Reading

Author: Jan Výkruta

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Jan Hajič, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Expressive reading is one of possible oral presentations. The text being read is usually prose or poetry. Little has been done in research of what affects expressiveness and whether it can be generated by computers. LibriSpeech, a large scale corpus of read prose and poetry allows us to test generation of expressive reading using machine learning methods. We have focused on poetry as it is generally more expressive. We have prepared methods, that can be used to train more models as well as to prepare different data that could be fed in our learning methods. Moreover, we have developed an extendable application that takes a poem, predicts the reading, visualizes it and plays an audio record generated from the reading using a TTS system.

Keywords: generative modeling prosody speech recognition text to speech
# Contents

**Introduction** 3

1 Related Work 8

2 Technologies used 10
   2.1 Phones .............................. 10
   2.2 Automatic Speech Recognition and Forced Alignment 10
   2.3 Machine learning ..................... 10
   2.4 Recurrent Neural Networks .......... 11
   2.5 LSTM ................................ 11
   2.6 Normalization ........................ 13
   2.7 Technology and Formats ............. 13
      2.7.1 Kaldi .......................... 15
      2.7.2 Keras and Tensorflow .......... 16
      2.7.3 MaryTTS ........................ 16
      2.7.4 SubStation Alpha subtitles ... 16

3 Dataset 18

4 Prosody Learning Setup 20
   4.1 Data points description ............. 20
   4.2 Data points creation ................. 21
      4.2.1 Temporal features ................ 21
      4.2.2 Prosodic features ................. 22
      4.2.3 Words-to-Phones information .... 23
      4.2.4 Final structure of the dataset ... 24
      4.2.5 Data points creation process .... 24
   4.3 Data points usage .................... 26

5 Models 28
   5.1 Baseline base-0 model ............... 30

6 Visualization 34
   6.1 SubStation Alpha subtitles .......... 34
   6.2 HTML table .......................... 35

7 Software 36
   7.1 Software structure .................... 36
   7.2 Project structure ...................... 37
   7.3 Database ................................ 38
   7.4 Back-end ................................ 38
   7.5 Prediction handling ................... 39
      7.5.1 Structural segmentation .......... 39
      7.5.2 G2P ............................. 41
      7.5.3 Prediction module ................. 43
      7.5.4 TTS module ....................... 45
7.5.5 Loading of precomputed values .................................. 47
7.6 Training pipeline utilities ......................................... 50
7.6.1 Corpus preparation and Computation of FA .................... 51
7.6.2 Alignment adjustments ......................................... 52
7.6.3 Alignment distribution ......................................... 54
7.6.4 Prosody features computation ................................ 55
7.6.5 Data points preparation ....................................... 56
7.6.6 Training .......................................................... 56
7.6.7 Incorporation of the model into the software .................. 56
7.6.8 Metadata format ................................................. 56
7.6.9 Data preparation methods ..................................... 57
7.6.10 Training methods ................................................ 58
7.6.11 Prediction methods ............................................ 59
7.7 Front-end and user interface ....................................... 60
7.7.1 Index page ......................................................... 61
7.7.2 Result page ....................................................... 62
7.7.3 Transformations module ....................................... 63
7.7.4 Data transformers ............................................... 64
7.7.5 Index script ....................................................... 66
7.7.6 Result script ..................................................... 66
7.7.7 Server .......................................................... 68
7.8 The Communication API between front-end and back-end ........ 68
7.8.1 front_end_utils.py ............................................. 69
8 User guide ........................................................................ 70
8.1 Dependencies .......................................................... 70
8.1.1 Dataset phase dependencies .................................... 70
8.1.2 Dependencies of other phases ................................. 71
8.2 Setup ........................................................................ 71
8.3 Settings .................................................................... 72
8.4 Dataset Phase .......................................................... 72
8.5 Before other phases ................................................... 73
8.6 Data points Phase ..................................................... 73
8.7 Training phase .......................................................... 74
8.8 Prediction phase ........................................................ 74

Conclusion ........................................................................ 76

Bibliography ...................................................................... 78

List of Figures .................................................................... 80

List of Tables ...................................................................... 82

A Attachments .................................................................... 83
A.1 Standard phones in English .......................................... 83
A.2 Custom phones .......................................................... 83
A.3 Global phone dictionary ............................................ 83
A.4 Provided files ............................................................ 83
Introduction

When I was in elementary school we would get assignments in which we would have to memorize a poem and then recite it in front of the classroom. Most of the students had problems with memorizing the poem, therefore, their recitation was not very good, and they got bad marks. As for me, however, I had no problem to memorize the poem. The woe of me was that my recitation was very monotonous, almost as if I was reciting prose. None of the teachers was satisfied with my performance, and I always had to plead to get a good mark. My recitation was not monotonous on purpose or because of laziness. I simply could not tell when to change the pitch, energy or tempo, which I recognized as the factors affecting expressiveness of recitation. I had the same problem with singing even though one is guided by a melody. To this day I cannot do it. At that time I thought: “What if there was something that would tell you how to recite a poem.” Therefore, I decided to try to create a software that can help me or anyone else affected with the same shortcoming.

In this work we would like to explore factors affecting expressiveness of read text, especially poetry. We will mostly focus on visual demonstration of certain features of speech expressiveness, which is a crucial part of reciting. We have chosen poems as they tend to have almost exaggerated expressiveness, and so they present an ideal target of our study. There is a vast collection of poetry records read by volunteers, as well as a corpus partially consisting of poems, which offer examples of expressive reading, and allows us to conduct this experiment.

Notations, definitions

We first introduce the terminology connected to expressive reading.

- Oral reading session corresponds to the sound of a text read out loud.
- Expressiveness is a quality of conveying a feeling or thought.
- An expressive reading is any oral reading session of a text, when the intonation and timing are not monotonous, and instead are changing in a manner suggesting that the fluctuations convey a certain thought or feeling.
- Meter is used to describe a rhythmic structure established for a verse (e.g. iambic pentameter).
- We use the term prosody to describe auditory measures, namely pitch and energy or loudness of the voice.
- We will use the term prosodic features to describe pitch and energy values of an entity.
- Temporal features or data is used to describe the duration of an entity and the position of the entity in time (for example the time when a word started to being said).
- We will use the term verse to describe one line of a poem, and the term stanza to describe a group of verses delimited by more than one newline character (in other words delimited by an empty line) from another group.
- A Text-To-Speech (TTS) system is a system capable of artificial production of human speech based on input of normal language text or a symbolic representation of text.
We will use the term reading as shorthand for an oral reading session.

*Forced alignment* (FA) is a process by which text transcription is aligned to an audio recording to generate phone level segmentation, determining where in time is each phone said, and what is its duration.

*Machine learning* (ML) is a process of utilizing statistical patterns in data by a computer system, to perform tasks, that cannot be described by explicit instructions.

*Automatic speech recognition* (ASR) is utilization of computer systems to recognize spoken language, and turn it into textual representation.

A *phone* or phoneme (we will use phone) is a unit of sound in a language.

A *grapheme* is the smallest unit in a writing system.

**Goals**

The task that we attempt is to have a computer generate an expressive reading of a given poem. We focused our work on prosody and temporal features as they are the primary factors of the expressive qualities of a reading [Cowie et al., 1999]. Two readings from two different people of the same text could both be expressive while having different both prosodic and temporal features (sounding differently). Therefore, there is no one correct reading that would be expressive. We would like to explore whether a set of prosodic and temporal features can be generated for a poem, creating an expressive reading of the poem. As the quality of expressiveness of a reading cannot be described formally (and we do not treat it that way anyway in this work), it is left to the user to assess how well can the generated reading express the poem.

We decided to use machine learning to predict the prosodic and temporal features. It is a “cheap” approach to solving problems (meaning the solution using machine learning, does not require deep understanding of problem’s domain). Moreover, a suitable collection of data exists, which reinforces the decision to use machine learning.

Our goal is to create an application which could be fed a poem and the application would try to generate an expressive reading of the poem. Partially, we would like to try whether is it possible to do this using machine learning. Partially, we would like the resulting application to be used as a toy, therefore, it should produce varying results and the results should be visualized so as to make the experience of using the application enjoyable. The reading should be visualized in several ways as well as transformed using a TTS system into an audio record, which would be played. Moreover, we would like the application to be easily extendable, each part of the application to be replaceable and allowing anyone to pick up the work where we left off. Finally, we would like to allow the user to train their own model, which would be easy to integrate into the application.

**Solution overview**

Our software solution consists of a front-end component, a back-end component, and within the back-end the prosody prediction component. The front-end is re-
Figure 1: The overview of architecture of this software. The arrow symbolizes the data flow. The dotted line is the border between front-end and back-end.

Figure 2: The overview of the learning pipeline.
sponsible for accepting user requests and communicating the results to the user, both through a prosody-conscious visualization of the text, a synthesized audio of the generated expressive reading, and a detailed table of the prosody values themselves. The back-end takes care of generating these results from a given text (either user-given, or from a dataset), by running the prosody prediction component and collecting the materials (graphics, audio) that the front-end presents to the user. The architecture of the system is given in Figure 1. In order to get a prosody prediction module via machine learning, a separate pipeline is necessary to obtain training data and train a model that is then used by the software application. This pipeline is shown in Figure 2.

To build the prediction module we start with data preparation. We use LibriSpeech corpus [Panayotov et al., 2015] that is based on LibriVox (available at https://librivox.org/) which hosts audio recordings of read text by volunteers, and is under public domain. Using metadata distributed with the corpus we filter out all non poetry records. Then, using Kaldi ASR toolkit [Povey et al., 2011], we compute prosodic features and time alignments for these poems. Alignment and features are mapped to poem’s text associated with the source recording. Higher level features are computed for each phone of each poem. These features are training data for our machine learning methods. Models based on recurrent neural networks are trained on the data and the prediction module runs on these models.

Whenever a user inputs a poem, the poem is processed by several modules. First the poem is segmented into a structure, for each word its phonetic transcription is either looked-up or predicted by a Grapheme-To-Phoneme tool [Park and Kim, 2019]. For each phone its time alignment and prosodic features are predicted. An audio record of the poem is generated using a TTS system MaryTTS.

We have prepared a web page based front-end capable of prosody-conscious visualization of poem’s text. In the first part of the visualization phones are grouped into words and displayed using SubStationAlpha subtitles. In the second part words of the poem are visualized using tables and plots of computed features. Finally, an audio record of the poem is played. To accommodate these visualization methods, the front-end has a module which transforms the poem reading into all required formats (SubStationAlpha subtitles, HTML table and an audio record). An example of the visualization is given in Figure 3.

**Thesis structure**

We start with related work overview in Chapter 1. We review papers studying expressiveness and fluency of reading, which suggest what features we should focus on. Moreover, we describe a similar system with a completely different approach. In Chapter 2 we introduce technologies used in this project, and describe concepts which should the reader be familiar with to understand this text. We start with theoretical concepts, we describe phones, forced alignment and machine learning. We continue with description of used technologies, we describe Python packages, Kaldi toolkit, MaryTTS and subtitles format. In Chapter 3 we describe the dataset we used, and what steps were necessary to create it from already existing sources. In Chapter 4 we describe our experiment. We focus on definition and creation of data points, which will be used during the train-
Figure 3: An example showing the visualization of a poem. At the top are rendered the subtitles. On the bottom is the table with plotted values.

Various formats of data created during this stage are described as well as certain data structures. In Chapter 5 we describe the training phase of machine learning process. Different models are described as well as subjectively evaluated. In Chapter 6 we describe our methods of visualization of the generated reading. We describe the SubStation Alpha subtitles, structure of our table and audio generation using a TTS. In Chapter 7 we elaborately describe the whole software as well as the scripts used in the data preparation and training. Finally, in Chapter 8 we provide a user guide. The guide should help anyone acquainted with computer science use, modify and expand the software.
1. Related Work

Little has been done in automatic expressive reading research. Only similar research being SPARSAR [Delmonte and Bacalu, 2013, Delmonte and Prati, 2014, Delmonte, 2015], which also tries to generate expressive reading of poetry. However, SPARSAR approaches this task differently. It utilizes a complex system of layers of deep structural poem analysis spanning various domains of linguistics: syntax, semantics, grammar, rhythm, sentiment etc. as well as several previously developed systems at the same institute. Their work has not been made open-source. We would like to present a different, simpler approach avoiding poem analyses on structural level. We believe a good approach is to analyze how actual people conceive expressiveness of poems. This can be done by study of audio records with read poetry, which are readily available. Moreover, this approach is much less demanding on knowledge of mathematical linguistics, therefore, even inexperienced user can understand the software and develop their own models.

Other research aims to study fluency. Research on correlates of expressiveness and fluency [Cowie et al., 1999] states that expressiveness is broadly pitch related, however, fluency and expressiveness are interrelated; and that textual features play only a part in affecting prosody, suggesting other global mechanisms at work. Other work focuses on fluency classification [Bolanos et al., 2013, Bolaños et al., 2013]. A short story is read by children and a classifier is created assessing their fluency by scoring them on a fluency scale. The research suggests again that expressiveness and fluency are closely interrelated, and that females are generally more expressive when reading.

Research aiming to analyze rhythmic structure of poems and possible applications of syllable-stress to poem generation and translation [Greene et al., 2010]. For a given poem in a known meter, syllable-stress is assigned to each word. One of the possible applications is generating new poetry in certain meter. The stress pattern is used to prune the space of possible lines. This creates good sounding albeit nonsensical poetry. Another application presented by the authors is translation of poetry. For each line of a poem the stress pattern is used to prune the translation hypothesis space. This is demonstrated on translation of Divine Comedy from Italian — in which the meter is hendecasyllabic — into English targeting iambic pentameter. In contrast to [Greene et al., 2010], we would like our expressive reading predictor not to analyze or depend on the meter of a given poem.

Other research conducted by Baumann and Meyer-Sickendiek [2016] focuses to analyze spoken free-verse poetry. Free-verse poetry is much closer to everyday language, as it does not adhere to a strict meter as with traditional poetry, a different approach is needed to analyze it. The authors collaborate with the Lyrikline project (available at http://www.lyrikline.org), from which they obtain poem texts with records by the authors. They force align the text with records, and extract wide range of features. They plan to use machine learning to learn poet styles and cluster poets by similarity of their style. The results will be presented to an annotator that will assess performance of the models. The assessment would then be fed back into the machine learning methods. The authors study differences between poet’s reading of the poem and reading by a
naive reader. This is done to pinpoint which features are essential. Reading by a naive reader is generated using a Text-To-Speech system. The authors have implemented and tested only a small portion of their goals.
2. Technologies used

In this chapter we will describe technologies used in our software, as well as theoretical concepts the reader should be acquainted with. We have defined several terms in Section , we will describe some of them in more detail.

2.1 Phones

A phone is a unit of sound in a language that distinguishes one word from another. There are 39 phones in the English language. Each phone is represented by a group of symbols from the International Phonetic Alphabet. As these symbols are not practical to use in software, their transcription to Latin alphabet can be used instead. The phones are listed in Section A.1.

2.2 Automatic Speech Recognition and Forced Alignment

Automatic Speech Recognition (ASR) is a process of utilizing computer systems to recognize spoken language, and transform it into a textual representation. Forced Alignment (FA) is a subtask of ASR. FA is a process by which text transcription is aligned to an audio recording to generate phone level segmentation, determining where in time is each phone said, and what is its duration. As we only use the FA, we will not go into details about other processes in ASR.

FA is applied to generate training data used to train models in ASR. Instead of ASR, we use the resulting alignment between phones and audio as our temporal features as well as to extract the prosodic features from the correct temporal segment of the audio. We use the Kaldi ASR toolkit to compute FA.

2.3 Machine learning

Machine learning (ML) is a process of utilizing statistical patterns in data by a computer system to perform tasks that cannot be described by explicit instructions. Machine learning algorithms build a mathematical model based on observed data, which is called training data or set. This model can then be used to predict some target values for data not yet observed. ML approach to solving a problem is applicable — to a certain degree — whenever there are sufficiently large collections of data describing the problem. There are many techniques and algorithms used in ML, however, we focus only on recurrent neural networks (RNN), which can be subsumed under the term Deep learning. Deep learning resources we used, and would suggest to the reader, are the book by Goodfellow et al. [2016], a series of lectures by Straka [2018] based on the book and an exceptional blog by Olah [2015]. In this thesis we expect the reader to be familiar with neural networks and techniques used in ML.
2.4 Recurrent Neural Networks

Recurrent neural networks are a special case of neural networks. There are certain types of data, where the history of observed data can be of great importance to data currently being observed. For example if we would like to check whether a verb is written correctly, we need to know whether the subject was singular or plural. RNN are designed to take into account the history of data, therefore they are ideal for processing of time series data. The term time in this situation does not only refer to time as we perceive it, but also to, for example, positions of words in a sentence. For a neural network to be able to forward information from a previous state to some of the following states, it is necessary to introduce loops in the network. Instead of the loops the network can be copied several times (unrolled). The copied networks make up building blocks for the recurrent neural network. The blocks can be connected to pass information from previous states to following. The unrolling is given in Figure 2.1. The blocks are connected with one edge, take one input, take into account state of the previous block, output one result and send its state to the next block. This architecture can be reviewed in Figure 2.2. Therefore, the RNN can use information about previous states in the current state. However, the farther the states are apart the more difficult it is for the network to learn a dependency on previous data. This is caused by vanishing or explosion of back propagated gradients used during the training. One of the architectures designed to mitigate this is called Long Short-Term Memory (LSTM).

![Figure 2.1: Overview of a recurrent neural network. The x's denote input, h's output and A's the unrolled networks. The picture is taken from Olah [2015].](image)

2.5 LSTM

Long Short-Term Memory networks are a special kind of RNN networks, designed to learn long term dependencies. We remind that RNN consist of several repeating blocks. While in basic RNN one block has a simple structure (for example one layer, such a network can be seen in Figure 2.2), in LSTM the building blocks are more complex.

There are several possible structures of the building blocks, we will describe one of the common ones. In LSTM the blocks are connected by two edges. Over one edge the state of the previous block is sent, the other is called the cell state. The cell state is not only an edge from one block to another, it is more of a line running through the blocks with only linear transformations, therefore, it is easy for the network to send information from previous states to distant following states. Each building block has the ability to forget information flowing on the
cell state as well as to add its own information. The architecture of a LSTM network can be reviewed in Figure 2.3.

We will describe the LSTM building block in steps. The block first decides what portion of the data flowing on the cell state it is going to forget. A sigmoid layer is used to generate values between 0 and 1, the cell state is masked using these numbers. This step can be seen in Figure 2.4.

Then the block decides, which portion of data it is going to add to the cell state. The decision is made by another sigmoid layer. Candidate data (data that could be added) are created by some kind of regular layer (in our case the tanh layer). Using the decision from the sigmoid layer, the candidate data are masked, and then they are merged into the cell state. This step given in Figure 2.5.

Finally, the building block has to decide what portions of the cell state will be the output. The decision is made by another sigmoid layer. The cell state is pushed through another regular layer (in our case another tanh layer), output of this layer is then masked with the decision. The result is the state of the module, and is sent to output and to a following block. This step can be seen in Figure 2.6.

At no point in any step the sigmoid layers look at the cell state directly. There are several variants of LSTM building block architecture. One of them lets all the sigmoid layers to look at the cell state, therefore, the layers can adjust their decisions. This variant is given in Figure 2.7.
2.6 Normalization

As many machine learning methods do not perform well when the data does not look like Gaussian distributed values with zero mean and unit variance, normalization of the data is used. The data is centered by removal of mean value of each feature and then scaled by division of features by their standard deviation. We use a modification, that is not prone to be affected by outlier values. We refer to this as robust scaling. Robust scaling instead of removing mean removes the median, and scales according to the quantile range between the 25th quantile and 75th quantile.

2.7 Technology and Formats

Several third-party components are used in our software as well as non-standard formats to operate the software. In this section we will describe some of the
third-party components and all formats. Software we are not going to describe in too much detail are several Python packages, JavaScript libraries and other utilities, we list them and summarize their functionality. We start with Python packages (all can be found in PyPI, available at https://pypi.org/):

- **numpy** is a library used for efficient mathematical computations as well as machine learning,
- **pydub** is a library to manipulate and transform audio files,
- **httplib2** is a small HTTP library,
- **scikit-learn** is a library used for ML data preprocessing as well as basic ML itself,
- **joblib** is a pipe-lining library, it is only used to save and load scalers,
- **nltk** is a toolkit used for natural language processing, we only use a dictionary it provides,
- **jsonpickle** is a library for serialization and deserialization of Python objects into the JSON format,
- **django** is a web framework used to run a significant portion of our software,
- **g2p_en** is a module for grapheme-to-phoneme conversion of English language,
- **python-Levenshtein** is a module to compute the Levenshtein distance between two strings,
- **mutagen** is a library to handle audio files.

JavaScript libraries and other utilities:

- **Chart.js** is a library for creation of charts in a web-page, available at [https://github.com/chartjs/Chart.js](https://github.com/chartjs/Chart.js),
- **SubtitlesOctopus** is a library to render SubStation Alpha subtitles in a web-page using background workers and no DOM manipulations, available at [https://github.com/Dador/JavaScriptSubtitlesOctopus](https://github.com/Dador/JavaScriptSubtitlesOctopus),
- **ffmpeg** is a solution to convert audio and video, it is required by the pydub package, available at [https://ffmpeg.org/](https://ffmpeg.org/),
- **sox** is a utility for audio processing, available at [http://sox.sourceforge.net/](http://sox.sourceforge.net/),
- **rsync** is a utility for synchronization of files between directories, available at [https://rsync.samba.org/](https://rsync.samba.org/).

### 2.7.1 Kaldi

Kaldi [Povey et al., 2011] is a toolkit for ASR. It offers wide functionality, however, we have only used a small subset. We use Kaldi to compute forced alignment. The inner workings are based on Hidden Markov Models [Gales et al., 2008]. We will describe the directories and files required to compute forced alignment.

Kaldi utilities are in the `src` directory and there are many projects that can serve as a base for a new project in the `egs` directory. We have adopted the LibriSpeech [Panayotov et al., 2015] project. There is a `run.sh` script in each project, which controls computations of the project (we will refer to this script as a recipe). We use our own recipe based on the original. There is also a `cmd.sh` script configuring commands used in the recipe. The `data` directory is used to describe jobs, which should be run by the recipe. The jobs are generated automatically using scripts provided by the LibriSpeech authors. Each job is contained in its own directory (if the reader would be interested in their structure we suggest tutorial available at [https://www.eleanorchodroff.com/tutorial/kaldi/index.html](https://www.eleanorchodroff.com/tutorial/kaldi/index.html)). There are also two directories with additional information. A `data/lang` directory describing the language and a `data/local` directory that
would be normally used to build `data/lang`, but it is also used later. Both these directories would be generated automatically using the original recipe, however, we changed the recipe, therefore, we download these directories from LibriSpeech repository. The results of the recipe are located in the `exp` directory.

To compute FA a speech recognition model is needed, we use one trained by the LibriSpeech authors, it is downloaded into `exp/tri` directory. Apart from the model the `data/lang` directory is needed. The `data/lang/phones.txt` contains mapping between phones and their IDs, the `data/local/dict/lexicon.txt` contains a lexicon of all words of the language present in used dataset with their phonetic transcription. The `exp/filtered.ali/final.mdl` is the model with which was the data aligned. All these files are needed during the alignment adjustment.

2.7.2 Keras and Tensorflow

Keras [Chollet et al., 2015] is an API for training and running neural networks. It offers high-level approach, simplicity of use and is written in Python. It can use several back-ends to perform the actual computation, among them TensorFlow [Abadi et al., 2016]. Keras is available at [https://keras.io/](https://keras.io/), TensorFlow at [https://www.tensorflow.org/](https://www.tensorflow.org/).

2.7.3 MaryTTS

MaryTTS is an open-source multilingual TTS system. It can process input and generate output in several formats. Apart from plain text it can process several markup languages. The output can be in plain text, in one of the markup languages or an audio file. The SSML language based on XML can be used to describe prosody, therefore, it is the one we use to encode the input. We have chosen an audio file as output. The complete standard for the SSML language is available at [https://www.w3.org/TR/speech-synthesis11/](https://www.w3.org/TR/speech-synthesis11/). We will focus on the `prosody` element, which is heavily utilized in our software. The `prosody` element offers several useful attributes. The `contour` attribute is used to describe the pitch contour of contents of the element. The contour is specified by a list of pairs, where each pair consists of time and pitch value. The `volume` attribute should mark a change in volume of the contents, however, MaryTTS seems to ignore this argument. The `duration` attribute is used to specify the duration of the contents.

MaryTTS allows the user to pick from several voices, which will be used to generate the resulting audio. We have tested all English voices, and found the `cmu-bdl-hsmm` and `cmu-rms-hsmm` to be the best, therefore, these are installed automatically, others have to be installed manually. MaryTTS is available at [http://mary.dfki.de/index.html](http://mary.dfki.de/index.html).

2.7.4 SubStation Alpha subtitles

SubStation Alpha (SSA) is a format of subtitles that allows complex visual transformations during rendering of the subtitles using tags. Each file conforming to SSA format has to include a header in which metadata and styles of subtitles are
defined. Metadata are in a section [Script Info], styles are in a section [V4+ Styles]. The styles allow configuration of font, font size, colors, positions and margins. Then follows a section with the subtitles itself called [Events]. The subtitles consist of lines, each line contains starting time, ending time, name of the style applied to the line, effects that should be applied to the line, the text that should be shown and margins settings. There are several effects available: there can be none, an effect described by a function or a karaoke\(^1\) effect.

The text on each line that should be shown can contain several tags, that will alter its look. We will describe the tags we used, however, there are many more. The \texttt{\textbackslash fs} tag followed by an integer sets the font size of the rest of the line or until another such a tag is encountered. The \texttt{\textbackslash fad} tag takes two arguments, it controls a fade-in and a fade-out animation, the parameters signify duration of the animation. The \texttt{\textbackslash 1c} tag sets the color for rest of the line, or until another such a tag is encountered. It expects one argument in format: \texttt{&HBBGRR&}, where \texttt{BB} should be a hexadecimal number for blue color and analogically \texttt{GGRR} for green and red. The \texttt{\textbackslash k} tag followed by an integer, is used to create the karaoke effect for rest of the line or until another such a tag is encountered. The argument specifies in milliseconds how long it should take to change the color of the corresponding text. When the text is rendered, it is colored from left to right, signifying how long it should take to read the text and when to start. A complete guide can be found at \url{http://docs.aegisub.org/3.2/ASS_Tags/}.

\footnote{Karaoke is a type of entertainment, when text of a song is shown on a screen and an amateur singer tries to sing the words as they change color. A part of text should be sang when the text changes color.}
3. Dataset

Since we want to use machine learning methods a dataset is needed for development. To build an expressive reader, our dataset must contain texts and prosody for each text. There are no existing datasets containing prosody; however, the prosody can be calculated from audio recordings. Therefore, data for this task are audio recordings of read poetry. A vast collection of texts under public domain called Project Gutenberg available at [https://www.gutenberg.org/](https://www.gutenberg.org/) allowed volunteers to create LibriVox, a collection of audio recordings based on these texts hosted at [https://librivox.org/](https://librivox.org/) which are also under public domain, making them an ideal source. Most of the texts and audio recordings are older works for which copyright has expired and a significant portion of them consists of poetry.

To build a dataset from LibriVox each of the audio recordings would have to be downloaded separately as they are not structured in any easily processable format. Moreover, each audio recording would have to be cleaned separately as they begin with a prologue generally stating who is reading this text, that it is being read for LibriVox and where to find it; and end with an epilogue stating practically the same. These prologue and epilogue messages are undesirable as they are not a part of the poem, and they would confuse the statistical methods, so they need to be cut off. For each volunteer the messages are different making it difficult to automate such a task. Therefore, we decided to avoid using the unprocessed audio recordings with the messages, as well as cutting off the messages ourselves. Instead, we used parts of an already existing corpus as our dataset.

A large scale 1000 hours long corpus called LibriSpeech [Panayotov et al., 2015] available at [http://www.openslr.org/12/](http://www.openslr.org/12/) has been created in order to build ASR models on a large dataset. It consists of cleaned up — as described in the previous paragraph — LibriVox audio recordings, organized by reader identification number (ID) and recording ID. Each audio recording is split into several shorter parts making it less demanding to process by ASR systems. For each text only separate portions of its audio recording are used, less intelligible parts are cut out, so the original structure of the text is lost. A transcription of the work being read is associated with each audio recording. The transcription is stripped of all punctuation and all sentence structure. This is required by the ASR system we use. Structure of transcription files is given in Listing 3.1.

Listing 3.1: Structure of the transcription files. The hyphen separated numbers at the beginning of each line are IDs of parts the audio was split into. The words following the IDs are said in the corresponding part of the audio recording. The three dots signify the line continues with more words.

<table>
<thead>
<tr>
<th>ID</th>
<th>ID</th>
<th>Line continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7909</td>
<td>106370</td>
<td>AND STRAIGHTWAY THEY CEASED . . .</td>
</tr>
<tr>
<td>7909</td>
<td>106370</td>
<td>AND BENEATH THE BREEZE THE . . .</td>
</tr>
<tr>
<td>7909</td>
<td>106370</td>
<td>AND BENEATH IT SMOOTH ROCKS . . .</td>
</tr>
<tr>
<td>7909</td>
<td>106370</td>
<td>FROM HERE AN ICY BREATH . . .</td>
</tr>
<tr>
<td>7909</td>
<td>106370</td>
<td>BUT THERE IS A CONTINUAL . . .</td>
</tr>
<tr>
<td>7909</td>
<td>106370</td>
<td>AND A HOLLOW RAVINE BRINGS . . .</td>
</tr>
</tbody>
</table>

The corpus is distributed split into several parts, based on quality of audio recordings. A database is also distributed with the corpus describing each reader, each audio recording and other metadata used by the authors to create ASR
models. Each recording is associated with a project and each project is associated with a genre, using this information we filtered out all non poetry which left us with approximately 15 hours of read poetry. These audio recordings served as our dataset. Structure of an example directory containing one audio recording with its transcription is given in Figure 3.1. An overview of dataset statistics is provided in Table 3.1.

As we do not create ASR models, the data for training can be of much smaller magnitude. The difficulty of cutting off prologue and epilogue messages, splitting the audio recordings at word boundaries and computational capacity restrictions during training phase made us not pursue creation of additional data.
4. Prosody Learning Setup

The experimental part of this thesis consists of training a prosody generation machine learning model. Before we move to training, the training data must be constructed.

4.1 Data points description

The basic data point fed to most of our learning methods in our experiments is a phone represented by a vector of length four consisting of:

- `phone_id`, there are 54 phones, each phone is internally represented as a string of characters, and a global phone dictionary is used to obtain associated IDs
- `duration`, duration of the phone in seconds, represented as a floating point number,
- `average_pitch`, average pitch of the phone in Hertz represented by a floating point number,
- `average_energy`, average energy of the phone in decibels represented as a floating point number.

We will call this data point a basic data point. A more advanced data point is a phone represented by a vector of length six consisting of:

- `previous_phone_id`, the ID of the phone that directly preceded the current phone,
- `current_phone_id`, the ID of the phone for which this is a data point, the current phone,
- `following_phone_id`, the ID of the phone directly following the current one,
- `current_phone_duration`, the duration of the current phone in seconds, represented as a floating point number,
- `current_phone_average_pitch`, the average pitch of the current phone,
- `current_phone_average_energy`, the average energy of the current phone.

We will call this data point a trigram. A global dictionary contains all phones, and the position of a phone in the dictionary corresponds to the ID of the phone. There are two types of phones: standard phones and custom phones.

The standard phones are used to describe a sound that should be made, and a collection of such phones forms the sound of a word. These phones are listed in Section A.1 Each representing a specific sound apart from `SPN` which stands for spoken noise. These phones are generated by the Kaldi toolkit as well as the g2p system.
We created the custom phones to describe the structure of a poem. The custom phones are listed in Section A.2. The `<COMMA>`, `<SEMICOLON>`, `<THREEDOTS>`, `<COLON>` phones are a finer alternative to `<PUNCTUATION-MIDDLE-OF-SENTENCE>` in a case where would be switching between finer and more robust approach beneficial. In this work we use only the finer option. The `<DOT>`, `<QUESTION-MARK>`, `<EXCLAMATION-MARK>` phones are a finer alternative to ’<PUNCTUATION-END-OF-SENTENCE>’, in analogy to the previous sentence. The `<WHITESPACE>` phone is used at the end of a word. The `<START-POEM>`, `<END-POEM>` phones are used to delimit the beginning and ending of a poem.

The contents of the global dictionary are listed in Section A.3. There are 54 phones. The data points passed to a learning method are usually transformed in some way (for example the values are normalized), the transformation used is dependent on the method and will be described with its method.

### 4.2 Data points creation

To create described data points a long process awaits our dataset described in the previous chapter. Preparation of data points requires five types of information:

- when is each phone being said, and its duration (temporal features),
- pitch values for each phone (part of prosodic features),
- energy values for each phone (part of prosodic features),
- which groups of phones form which word (words to phones information),
- the poem text.

We start with the poem text as it can be very easily extracted from the transcription file (described in Chapter 3). The transcription file has the format: each line contains `record.partition_name poem text in the partition...`. From each line the partition name is stripped and the rest is saved. This creates a file with poem’s text.

#### 4.2.1 Temporal features

We need to compute the starting time and duration of each phone. This can be achieved using forced alignment. The FA is computed during creation of speech recognition models, to test the quality of a model. Therefore, the Kaldi ASR toolkit provides utilities to compute the FA of a record with its transcription. The corpus described in the previous chapter is already prepared for this task, hence the structure of the extracted dataset was not modified.

To perform the alignment Kaldi needs a data directory describing the jobs to perform on the dataset (Kaldi directories are described in Subsection 2.7.1). This is prepared from the dataset using utilities provided by the corpus authors. The alignment also needs a language directory, and a speech recognition model, we use those created by the corpus authors. The forced alignment is computed, however, during the process a problem occurs that was ignored by the LibriSpeech authors. They might have not even known about it as the error message is only logged
during the process but not written to standard output. According to [Ivanov et al. 2015] the argument called `beams` of the handling script serves as a trade of between alignment speed and accuracy. What is not documented anywhere is that it also serves as a kind of limitation to how long a record can be aligned. If this limitation is not satisfied for a certain record, this record is omitted and the fact is logged. Such a violation occurred for several records from the corpus, hence we increased the `beams` argument value to 100.

Results of FA are saved in Kaldi’s own binary format. The alignment is then transformed in several rather technical steps (therefore, they are described in Chapter 7) resulting in a file with alignment for each poem. The format of the file is as follows: each line corresponds to one phone from the poem, the order of the phones is the same as in the poem, each line consists of `start_time`, `duration` and `phone_string_representation`. The `start_time` field is the time, when the phone started to being said in seconds. The `duration` field is the duration of the phone in seconds. The `phone_string_representation` field is the identifier of the phone, it is one of the standard phones described in this chapter with a positional suffix. The positional suffix is one of the letters: B, I, E, S. It denotes the position of the phone in a word it is part of. The letters mean: at the beginning, in the middle, at the end and a single phone in the word. These will be used soon to create another structure necessary for further processing. An example of the format can be seen in Listing 4.1.

Listing 4.1: An example of the alignment file format.

```
. . .
0.33 0.16 SH_B
0.49 0.1 AE1_I
0.59 0.05 D_I
0.64 0.09 OW2_I
0.73 0.09 Z_E
0.82 0.07 TH_B
0.89 0.07 R_I
0.96 0.08 IY1_E
1.04 0.08 T_B
1.12 0.04 AH0_I
. . .
```

4.2.2 Prosodic features

The prosodic features constitute of pitch and energy values. Neither can be computed directly for a phone. However, Kaldi allows us to compute both at repeating time steps each progressing a certain amount of time. The default value of the progression is 10ms, which is what we used. The values of the progression is called `feature granularity` or just granularity throughout this text. Pitch and energy are computed each by a different script. They are both output in the same format. It is Kaldi’s own dictionary format. The data is then transformed in a rather technical process, therefore it is described in Chapter 7. After the transformation there are files with pitch and energy for each poem calculated at time steps defined by the granularity. Each line of the files corresponds to the values computed at that time step. Each line of the pitch file has the format: `ignored_value` and `pitch` separated by a space. The `ignored_value` field was
left in case it would be useful. The **pitch** field is the pitch in Hertz (Hz). Each line of the energy file has the format: **energy** and 12 ignored fields. The ignored fields were left in a case they would be useful. The **energy** field is the energy in decibel. An example of pitch and energy file format can be seen in Listing 4.2 and Listing 4.3 respectively.

Listing 4.2: An example of the pitch file format.

```plaintext
0.6892858 196.0257
0.6089537 187.4211
0.7241628 177.4156
0.9521322 167.9442
0.9633806 163.8079
0.9063125 159.7734
0.9029101 157.4006
0.9520133 155.8383
```

Listing 4.3: An example of the energy file format.

```plaintext
15.15943 -19.6783 -2.917067 -33.50447
16.18159 -18.83723 1.376625 -34.92297
17.48585 -14.89442 11.8613 -30.39478
20.02316 -2.43429 27.17081 -24.98362
20.91581 5.671484 24.82432 -16.31271
```

### 4.2.3 Words-to-Phones information

To properly assign the aligned phones to words from the poem, additional information is needed. The phones need to be grouped together according to what word they form. This creates a dictionary where the key is a word and the value are pointers to phones representing the word. The pointers point to phones in the alignment file. We will call this the **Words-to-Phones** information, file or just Words-to-Phones throughout the text. To construct the dictionary the positional suffixes are used to aggregate phones representing a word. The suffixes are stripped and space separated phones for each word saved. To progress a lexicon containing each word in the dataset with its phone representation is needed. Such a lexicon is contained in Kaldi’s dictionary directory, which was created by the LibriSpeech authors. Using the saved lists of phones aggregated from the alignment, the corresponding word for each list is looked up. Finally, the corresponding word and IDs of its phones are saved in the Words-to-Phones file. The whole process is described in greater detail in Subsection 7.6.3. Format of the file is given in Listing 4.4.

Listing 4.4: An example of the Words-to-Phones file.

```plaintext
SHARP 72 73 74 75
MARTYRDOM 76 77 78 79 80 81 82 83
TOWARDS 85 86 87 88 89
```
4.2.4 Final structure of the dataset

Each poem is in its own directory. Contents of a poem directory are the file with text of the poem, the transcription file of the poem (as described in Chapter 3), the alignment file, the pitch values file, the energy values file, the Words-to-Phones file and the original audio recording split into several flac files. An example can be seen in Figure 4.1.

4.2.5 Data points creation process

The dataset is split into training, validation and testing datasets. This is done on a poem basis, each poem is picked at random into one of the sets. The training set consists of 79 poems, the validation of 4, and the testing of 5 poems. Now the datasets are transformed into collections of data points.

For each poem the poem’s text is segmented into stanzas, verses and words. While the actual structure of the poem is not preserved in the source corpus, there is a reason to do this. In the transcription of the poem each line corresponds to everything said in the associated audio recording partition. The recording is split in places where there was a longer pause in speech, which corresponds to the expectation that there should be pauses at the end of each verse. Therefore, we can look at the lines in the transcription file as if they were actual verses. As the transcription file contains always just one block of lines, there is always only one stanza. However, the segmentation is prepared if there were better data, which preserved the original structure. We will use the term structure to refer to a poem segmented into stanzas, verses and words.

To each word from a structure are assigned the corresponding phones from the alignment. This is done utilizing the Words-to-Phones file. To each phone are assigned its prosodic features (pitch and energy values), which were measured in the time interval of the phone. All the values measured in the interval are assigned to the phone (for example if the duration of the phone is 0.04s and the feature...
granularity is 10, then there are 4 pitch and 4 energy values acting as prosodic features). The process is described in the software chapter in Subsection 7.5.5.

Custom phones for each word are generated, based on the structure. If there is any punctuation in a word, the last punctuation is taken and a punctuation phone is created, otherwise a \textit{WHITESPACE} phone is generated. If the word is at the end of a line the \textit{END-OF-VERSE} phone is generated. If the word is at the end of a stanza the \textit{END-OF-STANZA} phone is generated. If more than one of the custom phones meet up (for example at the end of the poem), all of them are generated. The custom phones for each word are all created at once.

As these phones need both temporal and prosodic features, we cannot apply the same process as for the standard phones. The time between the last standard phone of the word and the first standard phone of the next word is taken as duration. If it is zero the prosodic values are interpolated from the same phones, otherwise the values measured in the interval given by the duration are used. For start time of custom phones the ending time of the word is used (the custom phones are located at the end of the word, therefore ending of the word is the beginning of the phones). The batch is constructed from these values. The duration is assigned only to the first custom phone, others are assigned zero. Prosodic values are assigned to all custom phones. Start time is assigned only to the first custom phone, the others are assigned the start time plus the duration.

As the prosodic features are made of all pitch and energy values measured in the interval of the phone, we can calculate several higher level features. Supported higher level features are:

- maximal energy, the maximal value from energy values,
- average energy, the average of energy values,
- average pitch, the average of pitch values,
- distance between maximal and minimal pitch value,
- direction of pitch, whether the pitch increases, decreases or remains stable.

Both standard and custom phones are extracted from the structure in the order in which they appear in the poem. Moreover, the \textit{START-POEM} phone is put in front of phones from a poem and the \textit{END-POEM} phone at the end of the phones. This way phones belonging to a poem are delimited from phones of other poems.

For each phone certain features are picked. When creating the basic data points we picked: phone ID (ID of the phone in the phone dictionary described in Section 4.1), duration, average pitch and average energy. When creating the trigram data points we picked: previous phone ID, current phone ID, following phone ID, current phone duration, current phone average pitch and current phone average energy. These are saved into a file. Each row in the file corresponds to one phone and contains the picked features. The same process is applied to all poems in all the datasets, where poems from one dataset are appended to the same file. This results in three files each containing one dataset. This concludes data points creation. Examples of the files containing datasets can be reviewed in Listing 4.5 and Listing 4.6. Statistics of individual sets are given in Table 4.2.5.
<table>
<thead>
<tr>
<th>dataset</th>
<th>size</th>
<th>avg. pitch</th>
<th>avg. energy</th>
<th>avg. duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic train</td>
<td>475816</td>
<td>155.52</td>
<td>19.56</td>
<td>0.074</td>
</tr>
<tr>
<td>basic validate</td>
<td>38027</td>
<td>124.53</td>
<td>19.56</td>
<td>0.068</td>
</tr>
<tr>
<td>basic test</td>
<td>28275</td>
<td>190.95</td>
<td>19.15</td>
<td>0.08</td>
</tr>
<tr>
<td>trigram train</td>
<td>475658</td>
<td>155.57</td>
<td>19.56</td>
<td>0.074</td>
</tr>
<tr>
<td>trigram validate</td>
<td>38019</td>
<td>124.55</td>
<td>19.56</td>
<td>0.068</td>
</tr>
<tr>
<td>trigram test</td>
<td>28265</td>
<td>191.02</td>
<td>19.15</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 4.1: An overview of training sets statistics.

the trigram sets were created using the same seed as basic sets, therefore, their statistics are almost identical.

Listing 4.5: Example of a file containing basic data points.

```
"40", "0.0", "0.0", "0.0"
"16", "0.08", "237.02945", "19.0075275"
"22", "0.03", "198.99570000000003", "22.45085666666667"
"47", "0.0", "196.0257", "22.45886"
"15", "0.04", "197.7619", "22.42243"
"16", "0.05", "207.7205399999997", "22.438346"
```

Listing 4.6: Examples of a file containing trigram data points.

```
"40", "16", "22", "0.08", "237.02945", "19.0075275"
"16", "22", "47", "0.03", "198.99570000000003", "22.45085666666667"
"22", "47", "15", "0.0", "196.0257", "22.45886"
"47", "15", "16", "0.04", "197.7619", "22.42243"
"15", "16", "37", "0.05", "207.7205399999997", "22.438346"
```

4.3 Data points usage

The data points are fed into the training methods creating several models. The models and their methods are described in the next chapter. The training methods usually apply certain transformations to the data points before they are used in training.

We will use the term **labels** to describe the duration, average pitch and average energy. We will also use the term **input features** to describe the phone ID in the case of a basic data point, and to describe all three phone IDs in the case of a trigram data point. The labels are real numbers. The input features are categorical values.

The labels are either used as they are, or are normalized. Each label is normalized independently using the robust scaling (described in Section 2.6). The input features are either left as they are, normalized using the robust scaling or encoded using one-hot encoding. The first approach is used as a baseline, regardless of the fact, that such an approach is unusual in machine learning. The second approach, while also unusual and initially used because of a mistake, yields surprisingly good results. The last approach is a standard in machine learning to encode categorical values. Which method is used where is described with the corresponding model in next chapter.
The training and validation datasets are used during model training. The test dataset is used only to evaluate the model. When the model is later used to predict labels for new input features, the features are transformed in the same way the training features were.
5. Models

Several models were created and tested. Temporal and prosodic features of a phone are dependent on the context of the phone. Therefore, it is desirable during training and prediction to take into account the history of already seen phones, making recurrent neural networks the main structure used. We only used smaller networks, to allow quick development cycle, and because of restrictions on computational capacities.

As a baseline model a dense network is used. All further development was carried out using recurrent networks. We will use the terms input features and labels as defined in the previous chapter. The dense network consists of three layers (we use the term dense 1x1 in our tables):

- a dense layer with 100 neurons,
- a dense layer with 100 neurons,
- a dense layer with 3 neurons (number of predicted labels).

There are four increasingly complex recurrent networks. The simplest one consists of two layers (we use the term LSTM 1 in our tables):

- a LSTM layer with 100 neurons,
- a LSTM layer with 3 neurons (number of predicted labels).

A more complex network consists of four layers (we use the term LSTM 1x1 in our tables):

- a LSTM layer with 100 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 100 neurons,
- a LSTM layer with 3 neurons (number of predicted labels).

Even more complex network consists of six layers (we use the term LSTM 2x2x2 in our tables):

- a LSTM layer with 200 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 200 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 200 neurons,
- a LSTM layer with 3 neurons (number of predicted labels).

Finally, the most complex network consists of eight layers (we use the term LSTM 5x5x5x5 in our tables):

28
<table>
<thead>
<tr>
<th>model ID</th>
<th>transformation</th>
<th>w. size</th>
<th>w. stride</th>
<th>epochs</th>
<th>network</th>
</tr>
</thead>
<tbody>
<tr>
<td>base-0</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>10</td>
<td>Dense 1x1</td>
</tr>
<tr>
<td>mm-idnorm</td>
<td>IN + FN</td>
<td>33</td>
<td>33</td>
<td>5</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>mm-0</td>
<td>OH + FN</td>
<td>33</td>
<td>33</td>
<td>5</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>mm-1</td>
<td>OH + FN</td>
<td>33</td>
<td>33</td>
<td>5</td>
<td>LSTM 1x1</td>
</tr>
<tr>
<td>mm-2</td>
<td>OH + FN</td>
<td>33</td>
<td>33</td>
<td>16</td>
<td>LSTM 1x1</td>
</tr>
<tr>
<td>mm-3</td>
<td>OH + FN</td>
<td>33</td>
<td>33</td>
<td>30</td>
<td>LSTM 1x1</td>
</tr>
<tr>
<td>ss-0</td>
<td>OH + FN</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>ss-1</td>
<td>OH + FN</td>
<td>5</td>
<td>5</td>
<td>30</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>xs-0</td>
<td>OH + FN</td>
<td>100</td>
<td>5</td>
<td>5</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>xx-0</td>
<td>OH + FN</td>
<td>100</td>
<td>100</td>
<td>5</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>ls-0</td>
<td>OH + FN</td>
<td>50</td>
<td>5</td>
<td>5</td>
<td>LSTM 1x1</td>
</tr>
<tr>
<td>ls-1</td>
<td>OH + FN</td>
<td>50</td>
<td>5</td>
<td>3</td>
<td>LSTM 2x2x2</td>
</tr>
<tr>
<td>ls-2</td>
<td>OH + FN</td>
<td>50</td>
<td>5</td>
<td>3</td>
<td>LSTM 5x5x5x5</td>
</tr>
<tr>
<td>ls-3</td>
<td>OH + FN</td>
<td>50</td>
<td>5</td>
<td>1</td>
<td>LSTM 5x5x5x5</td>
</tr>
<tr>
<td>ls-4</td>
<td>OH + FN</td>
<td>50</td>
<td>5</td>
<td>50</td>
<td>LSTM 1</td>
</tr>
<tr>
<td>xx-1</td>
<td>OH + FN</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>LSTM 2x2x2</td>
</tr>
<tr>
<td>xx-2</td>
<td>OH + FN</td>
<td>100</td>
<td>80</td>
<td>15</td>
<td>LSTM 2x2x2</td>
</tr>
<tr>
<td>xx-3</td>
<td>OH + FN</td>
<td>100</td>
<td>80</td>
<td>5</td>
<td>LSTM 2x2x2</td>
</tr>
<tr>
<td>xx-4</td>
<td>OH + FN</td>
<td>100</td>
<td>80</td>
<td>5</td>
<td>LSTM 5x5x5x5</td>
</tr>
<tr>
<td>xx-5</td>
<td>OH + FN</td>
<td>100</td>
<td>85</td>
<td>7</td>
<td>LSTM 5x5x5x5</td>
</tr>
<tr>
<td>xx-6</td>
<td>OH + FN</td>
<td>100</td>
<td>80</td>
<td>2</td>
<td>LSTM 5x5x5x5</td>
</tr>
<tr>
<td>ll-0</td>
<td>OH + FN</td>
<td>50</td>
<td>40</td>
<td>5</td>
<td>LSTM 2x2x2</td>
</tr>
</tbody>
</table>

Table 5.1: The overview of all models trained and what parameters were used. Values in model ID column are used to refer to models in the text. The transformation column is used to describe which transformations were applied. IN stands for ID normalization, FN for feature normalization and OH for one-hot encoding. The w. size column holds the size of the rolling window used, w. stride holds the step used to move the window. The epochs column describes the number of epochs used. Finally, the network column describes the network used (the values are defined in description of each network).

- a LSTM layer with 500 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 500 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 500 neurons,
- a dropout layer with the rate of 0.1,
- a LSTM layer with 500 neurons,
- a LSTM layer with 3 neurons (number of predicted labels).

Parameters, with which each model was trained, can be seen in Table 5. We will describe each model going from top to bottom, therefore, we suggest that
the reader first goes through the table and then returns to reading. For each model we will mention both its strong and weak points, as well as our hypothesis what affects its performance. Finally, we will give our subjective opinion on its performance. The opinion is based on model’s performance on the first stanza of Divine Comedy. Evaluation using the test set can be seen in `test_results.txt`, however, these results should not be considered decisive, as the nature of the problem makes it difficult to measure, and we did not aim to formally evaluate the results. Therefore, we will not review the results here.

## 5.1 Baseline base-0 model

The base-0 model served as our baseline. It did not utilize a recurrent network, instead a basic dense network was used. This is the only model using trigram data points. Neither the input features nor the labels were transformed in any way. The model predicts very long duration of phones, it heavily prefers U-shaped pitch contour (meaning the beginning and end of a word have higher pitch than the middle), and the energy values have little variation and each value is close to its neighbors. Neither of these properties suggest that the result would be expressive in any way.

### Improvements

From now on we will only describe models using recurrent networks. Because of the nature of recurrent networks, we needed to make sequences out of the data points. To do this we used a sliding window over the data points. We tested several window sizes, as well as several step sizes changing by how much the
window moved. If the number of data points did not allow to exactly create the windows with given size and stride, the data points were padded (this is described in Subsection 7.6.10). The windows can be reviewed in Figure 5.1. We noticed a trend: with increasing number of epochs the results generally worsened even though the performance on validation set was improving. We were not too surprised by this. The validation and testing data are not reliable as the expressiveness of a reading cannot be formally measured, and two very different readings can both still be expressive.

**mm-idnorm**

The mm-idnorm model used normalization of categorical input features instead of one-hot encoding. The model predicted diverse pitch contours and moderately flat energy contours. We were a little lucky that our first model predicted interesting pitch values. The model, however, predicts rather long duration for certain short words.

**mm-0**

The mm-0 model is the first in a series of models using one-hot encoded input features. We could say this model is the previous model done right. The models differ quite a bit. The mm-0 model improved in predicting shorter duration times for correct short words. The energy contour is less flat, and we would consider the energy values to be good. However, the pitch contour considerably flattened and lost its appeal.

**mm-1**

As the mm-0 model performed reasonably well, we decided to use the same parameters with a more complex network. The resulting mm-1 model predicted practically identical energy contour and very similar duration times. However, the pitch contour was much less flat, although keeping a similar shape.

**mm-2, mm-3**

We were under the impression, that the model did not have enough time to learn, therefore, we increased the number of epochs. We used an early stopping condition to stop the training, when the model has not improved in the last 10 epochs. The resulting mm-2 model was trained for 26 epochs and then reverted to the state it was in after the epoch 16. The model predicted very similar energy values to those of other mm series models. It still predicts short duration times for correct short words. The pitch contour shape corresponds to that of mm-1, however, it is in some segments more flat and in other less flat.

We let the training run with the same parameters for 30 epochs just to see, what would the results look like. The resulting model mm-3 clearly prefers flat pitch contours. Energy contours remain practically the same, and the model assigns a little shorter duration times to short words.
**ss-0, ss-1**

As duration and pitch values prediction did not change much, when modifying the network and epochs, we decided to change the window size and stride. Two models with a very short window, and stride of the size of the window were created. One was trained for 5 epochs (ss-0), the other for 30 epochs (ss-1). The ss-0 model predicted similar energy contours as the previous models. Prediction of duration times slightly worsened compared to mm-3, however, the worst performance was on pitch values. The resulting pitch contours were flat and all values were close to 160 Hz. Training the model for more epochs yielded little improvement, the resulting ss-1 model performed practically the same.

**xs-0**

We came to the conclusion that a small window degrades the performance, therefore, all other models use long windows. A xs-0 model with very long windows and short stride was trained first. The performance of the model was similar to that of ss-0 and ss-1, it predicted reasonable energy values and duration times, however, the pitch values resulted in a flat contour. Moreover, the pitch values were around 220 Hz in certain verses. We suspect that this occurred because all the windows were similar, therefore universal values were predicted.

**xx-0**

To mitigate the similarity of windows, we tested a longer stride. The resulting model xx-0 uses very long windows, and stride of the size of the windows. Predicted duration is practically the same, the energy contour is slightly flatter and the pitch contour is much more diverse, in fact it is similar to the contour predicted by mm-2.

**ls-0, ls-1, ls-2, ls-3 and ls-4**

It seems that long windows with long stride is a good direction to follow, however, we still decided to test more cases with long windows and short strides. A series of these models uses the same window and stride size, were trained for similar amount of epochs and use increasingly complex networks. The models ls-0, ls-1, ls-2, ls-3, ls-4 universally preferred flat pitch contours, decent energy contours and duration times. ls-2 an ls-3 predicted the best pitch contours in this series, however, energy contour considerably flattened with ls-3. The results worsened with higher amount of epochs than used in this series.

**xx-1, xx-2**

The ls series confirmed our hypothesis that short stride is detrimental to performance, therefore, all remaining models were trained with long window size and stride. The xx-1 and xx-2 models which were similar to xx-0, predicted similar energy and duration values, however, the pitch contour flattened, which is no surprise as we let the model train for longer than necessary to test the effects. Moreover, the xx-2 also flattened the energy contour.
xx-3

Higher number of epochs proved to be detrimental to performance of models (we remind that during training the performance on validation set was still improving, however, the set is not reliable). Therefore, we tested several short training sessions. The xx-3 model predicts similar values to those predicted by xx-0. The duration times differ slightly, the pitch contour has the same shape, although it is flatter and the pitch contour has very similar (flat) shape.

xx-4

The xx-4 model was trained with the same parameters only on the most complex network. Duration prediction improved, correct short words have short times, long complex words have longer duration than before. The energy and pitch contours are less flat, similar to those predicted by the first models, which we considered good.

xx-5, xx-6

The xx-5 model was trained for too long. While the energy contour remained the same, the model assigned shorter duration to complex words and the pitch contour flattened. The xx-6 model predicted similar duration, the energy contour flattened and the model clearly prefers a J-shaped pitch contour (meaning the end of a word has higher pitch than the rest).

ll-0

The last model trained was the ll-0 model, in which we tested shorter window sizes. The model predicted good duration times, similar energy contour to those predicted by first models, which we considered good, and reasonable pitch contour.

Favorites

Among our favorite models are mm-idnorm, mm-1, mm-2, xx-4 and ll-0, therefore, we suggest the reader to test them on their own poem. Other models are included as well.
6. Visualization

All visualization is controlled by the same system, each visualization type complements the others. There are three types of visualization:

- rendering of the reading using SubStation Alpha subtitles,
- detailed overview of the reading in a HTML table with plots,
- an audio record is played.

As the audio record cannot be seen, and instead has to be heard, we will not describe it here, its generation is documented in Subsection 7.5.4. The TTS system used to generate the record is not as configurable as we would like, therefore, the audio record and the visual representation can be impaired. If the record would seem distracting a mute button is provided. We leave to the reader to listen to some of the examples provided with this work. We suggest records located in the recordings.zip.

6.1 SubStation Alpha subtitles

The poem reading is visualized using subtitles. Each verse is rendered at the time when it is supposed to be read. When the verse is rendered it consists of two segments of text. The first segment in green color is a phonetic transcription of the verse. The second segment in white is the text of the verse. The phonetic segment suggests how to read the words in the text segment. It is made of space separated groups of phones, each group corresponding to one word. The phones are separated between themselves with hyphens. Each phone is of different size, the size signifies the energy of the phone. The bigger it is the higher the energy. As the playback of the reading progresses, the phones start to change color to a spectrum of red colors. When a phone changes its color, it signifies the phone should be read at that time. The red colors signifies the pitch of the phone. The richer the color the higher the pitch. An example is given in Figure 6.1.

The SSA format allows specifying styles of the subtitles. We use two styles, Animate and Static for the phonetic segment and text segment respectively. Both styles use short fading transitions using the \fad tags. Different size of phones are achieved using the \fs tag. The color is changed using the \lc tag. And finally, the change of color at certain time is achieved using the \k tag. Because of this tag, the style of the phonetic segment marks the whole segment as a karaoke.

Such sleepy dulness in that instant weigh'd

![Figure 6.1: An example of the SSA subtitles.](image)
Figure 6.2: An example of the HTML table showing segment of reading of the first stanza of the Divine Comedy.

6.2 HTML table

Stanzas in the table are delimited by an empty row. Each row of the table corresponds to a verse. Each cell corresponds to a word, which is shown in the cell. In each cell the temporal and prosodic features of the word can be seen. me stands for maximal energy; ae for average energy, ap for average pitch, dist for distance between maximal and minimal pitch, start for start time of the word, end for end time of the word and dur for duration of the word. All pitch values are in Hertz, all energy values are in decibel and duration is in seconds. Moreover, for each word plots of pitch and energy values are shown. Each point on these plots corresponds to a phone forming the word. Blue plots show average pitch of phones forming the word and red plots show average energy of the phones. Finally, the currently read line is highlighted in yellow.
7. Software

The software allows a user to input a poem and pick several settings using a webpage. The user will then be redirected to another web-page containing the visualization of reading of the poem. The visualization has several formats, which are generated by the software. To generate the reading of a poem, the poem is structured, and prosodic values are predicted for the poem. The reading is then transformed into the visualization formats. The software is designed with modularity in mind. Each part of the process is easily substitutable.

7.1 Software structure

Figure 7.1: Overview of software structure. The arrow signifies the data flow and the dotted line separates front-end from back-end.

The software consists of two main components: a web-based Django application serving as project’s front-end and a prediction application serving as the back-end. Both applications are encapsulated in a Django project containing settings for both applications and database configuration. Front-end and back-end are completely interchangeable given both adhere to the communication API. There are also scripts used in the training pipeline which are only necessary during training and the software is independent of them.
7.2 Project structure

A fully set up project consists of these directories:

- **dataset_preparation** contains scripts used to prepare the dataset and create data points used in training,

- **examples** contains examples of poems used by the demo, these are the examples an user can pick from,

- **expressive_poetry_reader** contains the Django project with settings and database configuration,

- **learning** contains scripts used in training,

- **marytts** is home to the TTS system MaryTTS,

- **models** contains all available models and their metadata,

- **reading_predictor** contains the back-end,

- **manual_setup** contains files used during manual setup,

- **staticfiles** would contain all static files if the application would be deployed on a server,

- **temp** serves as a destination where all temporary files are stored,

- **web_demo** contains the front-end.

And these files in the root of the project:

- **activate.sh** is a script used to set correct path variables,

- **db.sqlite3** is a database (described in section Section 7.3) used to store information used by the Django project, and to store paths to all examples available to the front-end,

- **deactivate.sh** is a script used to reset the original path variables,

- **manage.py** is a script used to manage the whole software project,

- **requirements.txt** is a file containing all python package dependencies,

- **runtime.txt** contains the version of CPython interpreter with which the project is guaranteed to run,

- **setup.py** is a script used to set up the Django application,

- **setup.sh** is a script used to set up the whole project,

- **test_results.txt** is a file with performance of the models on the testing set.
7.3 Database

The database contains several tables created automatically by the Django project used to store web-page administrator’s credentials, database’s migrations etc. Only reading predictor example is created by our software and serves mostly as a repository of files related to the reading of a poem. It has the following fields:

- **poem_name** the name of the poem, this is shown to a user when they pick one of the examples,
- **poem_text_file** the path to the file containing poem’s text,
- **poem_phones_with_alignment_file** the path to the file containing computed forced alignment (described in Subsection 4.2.1),
- **poem_pitch_file** the path to the file containing the computed pitch values (described in Subsection 4.2.2),
- **poem_energy_file** the path to the file containing the computed energy values (described in Subsection 4.2.2),
- **poem_words_to_phones_file** the path to the file containing words-to-phones mapping (described in Subsection 4.2.3),
- **poem_record** the path to the file containing the poem’s audio recording in mp3 format,
- **feature_granularity** integer number describing with what frequency were pitch and energy computed (described in Subsection 4.2.2).

Poem name and feature granularity are a characters field and a positive integer field respectively. All other are file path fields.

7.4 Back-end

The back-end constitutes of several modules, all bound together by a back-end controller, an instance of class PredictorController. This controller decides, based on the user input supplied by the front-end, whether to just read precomputed values for an example poem, or whether to predict values for a custom poem or an example poem. Functionality of the controller can be seen in Figure 7.3. At this point the computation takes two different paths. One being the prediction of values for user’s own or an example poem (prediction path), the other being reading of precomputed values for an example poem (loading path). The later is practically the same as data point preparation differing only in post-processing of the constructed PoemReadingValue instance.

Decisions which direction to take:

- if the user poem option is chosen then the prediction path is taken,
- if an example poem option is chosen and the model to just read the precomputed values is chosen, then the loading path is taken,
• if an example poem option is chosen and a different model than the one in
the previous item is chosen, then example’s ID is used to look up the text
of the example in the database, and using this text the prediction path is
taken.

In the two following sections we will describe both paths: the prediction
handling and the loading of precomputed values respectively. Lastly, we describe
the training pipeline utilities in their own section.

7.5 Prediction handling

This section describes the first path of the computation, which is the prediction
values of prosodic and temporal features. The prediction segment is controlled
by an instance of the CustomPoemPredictor class. This predictor only passes
the poem text and model name to an instance of the FromTextPoemReading-
Constructor class (this class is the Predictor, that is shown in all the pictures).
This constructor processes the received poem text and model name and returns
an instance of PoemReadingValue. The poem is fed into the module responsible
for structural segmentation.

7.5.1 Structural segmentation

The poem is split into lines. Lines separated by one newline are considered
a block, blocks separated by two or more newlines are considered stanzas. The
module first creates these blocks and passes them to the constructors of Stanza
objects creating a list of these objects. In Stanza constructor each line in the
block is passed to the constructor of Line object creating a list of Line objects.
In Line constructor the line is separated by whitespace creating words (each word
retains any punctuation not separated by whitespace). Each word is directed to
the constructor of Word object creating a list of Word objects. In Word constructor
Input ID, poem text and model name from front-end

Should the reading be predicted or read from original values

- Read from original
- Predicted

Is the poem an example poem or a new one.

- New one
- Example

Process input ID using FromPrecomputedValuesPredictor

Process poem text and model name using CustomPoemPredictor

Load text of the poem associated with the input pick ID using the database

Return the constructed PoemReadingValue instance to front-end.

Process loaded poem text and model name using CustomPoemPredictor

Figure 7.3: The functionality of the PredictorController class.

Structural segmentation of text

Grapheme to Phoneme system

Reading prediction

Generating audio using TTS

Predictor

TTS generation

Figure 7.4: Back-end segment responsible for the prediction. The predictor is responsible for the first three modules and the TTS generation for the last.
the passed word is only saved as the original word, leaving other fields empty and ready for other modules. These other fields will eventually hold lists of phones. The list of stanzas created in the most upper level is then returned by the module. The whole architecture can be seen in Figure 7.6. This module is also used by FromExamplePoemReadingConstructor described in Subsection 7.5.5. This module returns a list of Stanza objects, we will call this a poem structure. From now on we will also use terms word, line and stanza to refer to an instance of Word, Line and Stanza classes respectively.

7.5.2 G2P

Words are extracted from the structure, for each word the phones representing the word are generated using the grapheme-to-phoneme (g2p) module. A word is passed to the module and then to the python package g2p.en [Park and Kim].

Figure 7.6: Architecture of the poem structure segmentation module.
Figure 7.7: The data flow between the g2p module and the constructor.

2019], which returns the phone representation. The package first tries to lookup
the word in the CMUdict [Weide, 2005] and only if it is unsuccessful it uses
a neural network model. The phones are returned as a list of strings.

The returned list must be filtered as the g2p_en package produces certain
undesirable phones, for example punctuation (the package returns punctuation
phones in format, that is undesirable, therefore we later create our own punctua-
tion phones) or even empty phones. All phones that are not in our global-
phones_dictionary dictionary are filtered. This dictionary is used all throughout
the software and all encoding of phones to phone IDs and vice versa are done using
this dictionary. This dictionary contains all standard phones as well as our custom
phones, both described in Section 4.1. When the standard phones representing
each word are generated, they are transformed from the string format into object
format, more specifically into instances of PhoneWithAlignmentWithFeatures
class.

The PhoneWithAlignmentWithFeatures class is used to represent a phone. The
class just encapsulates instances of two other classes: the PhoneWithAlign-
ment class and the PitchAndEnergyValue class. The first class handles all tem-
poral data, and phone identification. Temporal data consist of phone’s starting
time and duration, identification is just a string representing the phone, one that
is used in the global_phones_dictionary. The second class handles all prosodic
data. It holds energy and pitch values associated with the phone, and methods
to calculate higher level features like average energy etc. Both pitch and energy
values are collections of predicted (in this path) or measured (in the other path,
described further on) values in the time interval of the phone. The structure can
be reviewed in Figure 7.8.

From now on we will use the term phone object to describe an instance of
class PhoneWithAlignmentWithFeatures.

During the transformation of phone string format to phone object, described
above, only the identifier of the phone is known. Therefore, all other values
are left empty, ready for further processing. For each word our custom phones
representing the structure of the poem are generated. This is done by looking
at the end character of each word, and if there is a punctuation a custom phone
is generated. Moreover, the phones for <END-OF-WORD>, <END-OF-VERSE> and
<END-OF-STANZA> are generated by poem structure analysis (more closely de-
scribed in Paragraph 4.2.5). If two or more custom phones meet up (for example
PhoneWithAlignmentWithFeatures

PhoneWithAlignment
+ start_time: float
+ duration: float
+ phone: string
+ methods

PitchAndEnergyValue
+ energy_values: list of floats
+ pitch_values: list of floats
+ methods computing high level features

Figure 7.8: Overview of the structure of the PhoneWithAlignmentWithFeatures class.

at the end of the poem) all of them are generated at once. Phone objects for both standard and custom phones are assigned to their corresponding fields in each Word instance.

7.5.3 Prediction module

Reading prediction

List of PhoneWithAlignmentWithFeatures instances representing all phones in the poem

List of Prediction instances, each representing values predicted for one phone of the poem

Predictor

Figure 7.9: The data flow between the prediction module and the predictor.

From each word both standard and custom phone objects are extracted. The class PhoneValuesPredictor is responsible for the predictions. It is constructed with the model name supplied by the front-end. Using the name the correspond-
ing model is loaded. The model is loaded using metadata read from the metadata file, the filename has the prefix `meta_` followed by the name of the model. These metadata files are located in project’s model directory, and based on the model name the correct one is read. More on metadata structure in Subsection 7.6.8. The `PhoneValuesPredictor` class offers several loading and prediction methods, the correct ones are chosen based on the metadata. While the load method is rather universal, for each model a custom predict method is usually required. The metadata also contain several variables, which are saved and used during the prediction. Phone strings are extracted from phone objects passed to the predictor and then encoded into integer IDs using the `global.phones.dictionary`. These phone IDs are then passed to the correct prediction method. Overview of prediction methods can be seen in Subsection 7.6.11. The prediction method returns a list of prediction objects, each object corresponding to one phone passed to the method. The prediction objects are realized using the `Prediction` class. It serves only as an encapsulation of predicted duration, pitch and energy values. The module can be reviewed in Figure 7.10 and Figure 7.11.

![Figure 7.10: The overview of PhoneValuesPredictor construction.](image)

After prediction, each phone loads the predicted values from the corresponding prediction object. As mentioned in Paragraph 7.5.2 temporal data consist of start time and duration, but only duration is predicted. Therefore, the starting times have to be computed from the duration. To do this, phones are extracted from the words and their temporal data is adjusted. Phones are looped over, accumulated time so far is used as the starting time of the current phone and the accumulated time is increased by the duration of the current phone, then next phone is processed.

At this point the poem reading is constructed. An instance of `PoemReadingValue` class represents the poem reading. From now on we will use the term `poem reading` to refer to an instance of the `PoemReadingValue` class. The poem reading consists of the poem structure gradually constructed in the above steps, path to the associated audio, granularity and model name. The path to the audio is either the path itself or empty if there is no audio. The empty case occurs when the values should be predicted for the poem. The existing path case occurs when reading precomputed values for the poem (this case is described in Subsection 7.5.5). The poem reading offers a method that provides the audio. If the path
is not empty, the audio is just read, otherwise it is generated, more on this in the next paragraph. The prediction constructor is responsible for creation of the poem reading object, which is returned as a response to the request to predict a reading for a poem. The whole process handled by the prediction constructor can be reviewed in Figure 7.13.

7.5.4 TTS module

Lastly, one more module is considered part of the back-end. A module to create an audio recording from predicted values using a TTS system. The TTS system used is MaryTTS. This is the last back-end module, however, this module is not controlled by the prediction constructor. The recording is not always needed, therefore, it is generated on demand by an instance of the PoemReadingValue class associated with the poem. The recording is usually demanded by the front-end. To generate the recording, the predicted values have to be transformed into the SSML format used by the TTS system. The TextToSpeechRecordGenerator class is responsible for the generation of the audio. A poem reading is passed to the class. The reading is transformed into the SSML format using the SSMLTransformer class. To transform the poem, the SSML standard conforming header is generated. The whole poem is delimited using a speak element. Each stanza is delimited using a paragraph element. For each word in a stanza a prosody element is generated. The prosody element describes word’s duration,

\footnote{Standard available at \url{https://www.w3.org/TR/speech-synthesis11/}}
a pitch contour of the word and the word itself. The contour is described by a list of pairs consisting of time position and pitch. The contour is computed from the number of phones used to represent the word. For example, there are three phones in the word “then”, therefore, there are three pairs describing the contour. Each pair corresponds to one phone. A code example can be seen in Listing 7.1.

The TTS system is run separately as a server. Therefore, the generated SSML description of the poem reading, among several other variables, is encoded into an HTTP request using the `httplib2` package. The other variables are: what language should be used, which voice should be used and what should be the format of the output. The request is then sent to the server. If the response of the server is a success the returned content is an audio in the `wav` format. This recording is encoded into the `mp3` format. This must be done using several temporary files as the package `pydub` does not allow conversions in memory. Bytes of the final `mp3` audio are then returned in a `bytearray`. If the server cannot be accessed or any other connection error occurred, an empty `bytearray` is returned.

The process can be reviewed in Figure 7.16 and Figure 7.17.

Listing 7.1: An example of the SSML format encoding the first verse in the Divine Comedy.

```
<?xml version="1.0" encoding="UTF-8"?>
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/10/synthesis
http://www.w3.org/TR/speech-synthesis/synthesis.xsd"
xml:lang="en-US">
<p><prosody duration="0.14s" contour="(0%,157Hz)→(50%,150Hz)">IN</prosody>
<prosody duration="0.10s" contour="(0%,146Hz)→(50%,140Hz)">the</prosody>
<prosody duration="0.39s" contour="(0%,154Hz)→(20%,153Hz)→(40%,134Hz)
→(60%,138Hz)→(80%,147Hz)">midway</prosody>
<prosody duration="0.16s" contour="(0%,165Hz)→(50%,177Hz)">of</prosody>
<prosody duration="0.24s" contour="(0%,204Hz)→(33%,204Hz)→(66%,207Hz)
→(99%,215Hz)">this</prosody>
<prosody duration="0.25s" contour="(0%,209Hz)→(50%,197Hz)">our</prosody>
<prosody duration="0.51s" contour="(0%,204Hz)→(16%,209Hz)→(32%,215Hz)
→(66%,212Hz)">of</prosody>
```

Figure 7.12: The overview of an instance of the `PoemReadingValue` class.
7.5.5 Loading of precomputed values

This subsection describes the second path of computation, which is the loading of precomputed values. This is used to display example poems and a significant subsection of this process is essential to creation of data points. Even though the audio recordings in this path are already prepared, we would like to visualize them, therefore, it is still necessary to construct complex structures from extracted prosody values. If the back-end controller picks this path, it passes the input ID received from the front-end to an instance of the FromPrecomputedValuesPredictor class. This predictor uses the input ID as an example poem ID and look ups by the ID in the database. The database is described in Section 7.3. Paths from the database lookup are used to read the corresponding files. Readers of these files are passed to an instance of the FromExamplePoemReadingConstructor class. The files and their formats are described in Section 4.2.

The constructor reads the pitch and energy files into lists of floating-point numbers, we will refer to them as pitch respectively energy values. The poem text file is read line by line, we will refer to it as poem text. The alignment file is loaded into a list of instances of PhoneWithAlignment class (this class is described in the previous section), we will refer to it using the term alignment. The words-
Generating audio using TTS

Array of bytes representing the generated audio, in mp3 format

Figure 7.14: The overview of data flow between TTS module and mostly the front-end controller.

to-phones file is loaded into a list of instances of WordToPhones class, we will refer to it using the term words-to-phones data. This class encapsulates a word and IDs of phones associated with the word as specified in the words-to-phones file. The IDs point into the list of PhoneWithAlignment instances constructed from the alignment file. The files are now all loaded.

The reading is constructed in three steps.

1. The poem text is structured using the module for structural segmentation.

2. The phones read from the alignment file are mapped to the words of the poem.

3. The custom phones are calculated.

In the step one the poem text is passed to the module responsible for the structural segmentation of the poem. This process is the same as in the first path described in the previous section.

To map the phones from the alignment file to the words of the poem in step two, the words to phones data (this contains the recognized words with phones IDs pointing to the alignment file) is needed. Each word of the poem corresponds to a word from the words-to-phones data. The contents of the words-to-phones data are not fully reliable. Kaldi can sometimes recognize a different word, instead of the one actually being said. Moreover, we feared it could sometimes make
up a word, or rather split one actual word into two words it recognized. The occurrence of a made up word would mismatch the mapping of recognized words to the actual words. Therefore, we tried to use a heuristic. Instead of mapping the words one-to-one, we used the levenshtein distance to optionally skip an unsuitable word from recognized words. This approach, however, not only did not prevent the mismatch, it actually caused the mismatch. Therefore, we used the one-to-one mapping.

Each word of the poem (word A) is mapped to a recognized word (word B). We use the phone IDs associated with the word B to assign phones to the word A. The phone IDs are used to retrieve the PhoneWithAlignment instances from the alignment. These instances will be used to create phones for the word A. At this point the temporal and identification data of the phones are ready (they are in the PhoneWithAlignment instances). Only the prosodic data must still be generated. For each phone the start time and end time are used to retrieve the pitch and energy values in this interval. The feature granularity of the poem has to be taken into account during the extraction. A PitchAndEnergyValue instance is constructed from the extracted values. Now both the PhoneWithAlignment instance and PitchAndEnergyValue instance are ready to be combined into a PhoneWithAlignmentWithFeatures instance forming a phone. These phones are then assigned to the word A. This step can be reviewed in the Figure 7.18.

Finally, in the step three the custom phones are created. First, the punctuation phones are generated looking at the last character of each word. Then, end-of-word, end-of-verse and end-of-stanza phones are generated using poem analysis (more closely described in Section 4.1).

If two or more custom phones meet up (for example at the end of the poem is the punctuation, end-of-verse and end-of-stanza phone) all of them are generated at once (in a batch). The process is the same as in Subsection 7.5.2 However, one more step is needed to assign the correct temporal and prosodic data to
the custom phones. For each custom phone batch the preceding and following standard phone is taken. If the interval between the end time of the preceding phone and the start time of the following phone is not empty, then the length of the interval is used as the duration, and the pitch and energy values in this interval are used as the prosodic data. Otherwise, the duration is zero. The prosodic values are computed as a arithmetic mean of the last prosodic value of the preceding phone and first prosodic value of the following phone.

Prepared temporal and prosodic data are used to create the custom phone batch. The duration is only used for the first custom phone, others have duration set to zero. The prosodic data is used for each custom phone. To accomplish this step for the end of the poem, which is not followed by any phones, a mock word with one phone is constructed. The phone has all values set to zero. The word is only used in this step. This step can be seen in Figure 7.19.

The constructed structure, the received audio path and granularity are used to construct an instance of the `PoemReadingValue` class. The reading from pre-computed values has been constructed and is returned. The whole process can be reviewed in Figure 7.20.

### 7.6 Training pipeline utilities

The training pipeline is completely stand-alone, it is not used in the main application, is is not required to run the software. It merely serves to produce models. Significant portion of the loading-from-precomputed-values process is used during the data preparation. In fact, everything apart from the database lookup at the very beginning. The reused segment in yellow can be seen in Figure 7.20. The data preparation process can be seen in Figure 7.21. The process itself (with higher abstraction) is described in Chapter 4. Here we will focus on the scripts used during the process and technical side of training.

For this process a directory with working Kaldi distribution is required.
7.6.1 Corpus preparation and Computation of FA

The corpus preparation and computation of forced alignment is done using the run.sh script which is saved in the root of the LibriSpeech project recipe in a Kaldi distribution directory. First the corpus is downloaded and extracted from archives using a script provided by LibriSpeech authors. The corpus is split into several parts, each corresponds to a directory after the extraction, however, we need the whole corpus in the same directory. The directory with the largest portion is used as the directory where the whole corpus will reside (we will call it the corpus directory). Using the rsync utility other directories are merged into the corpus directory. The archives and other directories are erased. The corpus is now filtered using the filter_data.py script. All poems are extracted from the corpus and saved in a different directory (we will call it the dataset directory, as the poems are our dataset). The filtering is done based on the IDs of the audio recordings in the corpus. The IDs of poems were manually extracted from a database describing the corpus and saved in a file.

A model used for the forced alignment, a directory describing the language, and a directory with the lexicon are downloaded from the LibriSpeech repository and saved in exp/tri, data/lang and data/local directories of the LibriSpeech project respectively. Kaldi data directory, which is used during the alignment,

---

Words of the poem with assigned phones from step two

Create mock word at the end of the poem, with one phone, with all fields set to zero

For each word flag which custom phones should be created

Assign an arithmetic mean of the values to the custom phones, use zero as duration

Get the prosodic values in the interval between the words

Assign the values to the custom phones, use the interval as duration

Get the last prosodic value of the preceding phone and first of the following phone

If the current word is immediately followed by the following word, with no time between them

true

false

Figure 7.19: Overview of the last step of loading.

7.6.2 Alignment adjustments

As the alignment is split into multiple jobs, there are several files with the alignment. The alignment data is saved in Kaldi’s own binary format. We transform this format into a text format using Kaldi’s ali-to-phones utility. The text format is called CTM and has the following format on each line: record_id, channel_number, start_time, phone_duration, phone_id. Each line represents one phone. The record_id is the name of the audio file used to compute the alignment of this phone, channel_number describes the channel of the audio file and is ignored, start_time signifies the time when the phone was said in seconds, phone_duration is the duration of the phone in seconds and phone_id is the ID of the phone. There are as many of these files as there were jobs during the computation, the files are merged into one to allow simpler processing.

The audio of each poem was split by the authors of the corpus into several smaller files (partitions), which is done to lower the load during the computation.

is created from the dataset using the data_prep.sh utility and metadata files both provided by the LibriSpeech authors. The data is aligned using Kaldi’s align_fmllr.sh script. The downloaded model is used to do this. A description of mentioned directories can be reviewed in Subsection 2.7.1.
Kaldi computes alignment according to each partition, therefore, the times in the alignment are relative to the audio partition they were computed from. This means that if a poem is split into five partitions, then there will be five segments in the alignment starting at time zero, each segment containing start times relative to its partition, while all the segments are part of the same poem. Therefore, the times have to be adjusted to be relative to the beginning of the poem, not to the beginning of one of the audio partitions. The adjust_times.py script is used to do this. Examples can be seen in Listing 7.2 and Listing 7.3.

Listing 7.2: Example of alignment before adjustment.

<table>
<thead>
<tr>
<th>Input ID</th>
<th>Start Time 1</th>
<th>Duration 1</th>
<th>End Time 1</th>
<th>Start Time 2</th>
<th>Duration 2</th>
<th>End Time 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001-134707</td>
<td>15.44</td>
<td>0.100</td>
<td>15.54</td>
<td>0.150</td>
<td>15.69</td>
<td>0.120</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.69</td>
<td>0.120</td>
<td>15.81</td>
<td>0.120</td>
<td>15.93</td>
<td>0.060</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.93</td>
<td>0.060</td>
<td>15.99</td>
<td>0.060</td>
<td>233</td>
<td>320</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.44</td>
<td>0.100</td>
<td>15.54</td>
<td>0.150</td>
<td>15.76</td>
<td>0.140</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.76</td>
<td>0.140</td>
<td>15.90</td>
<td>0.140</td>
<td>233</td>
<td>320</td>
</tr>
</tbody>
</table>

Listing 7.3: Example of alignment after adjustment.

<table>
<thead>
<tr>
<th>Input ID</th>
<th>Start Time 1</th>
<th>Duration 1</th>
<th>End Time 1</th>
<th>Start Time 2</th>
<th>Duration 2</th>
<th>End Time 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001-134707</td>
<td>15.44</td>
<td>0.100</td>
<td>15.54</td>
<td>0.150</td>
<td>15.69</td>
<td>0.120</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.69</td>
<td>0.120</td>
<td>15.81</td>
<td>0.120</td>
<td>15.93</td>
<td>0.060</td>
</tr>
<tr>
<td>1001-134707</td>
<td>15.93</td>
<td>0.060</td>
<td>15.99</td>
<td>0.060</td>
<td>233</td>
<td>320</td>
</tr>
</tbody>
</table>
Figure 7.21: Overview of the data preparation, we will follow the arrows in our text.

Now we use the phone IDs to lookup the actual phones. The mapping dictionary between phones and phone IDs is created by the prepare_lang.sh script supplied by LibriSpeech authors. The dictionary is saved in the lang directory in the phones.txt file. The id_to_phone.R script is used to perform the lookup. The script performs a database merge not preserving the order. Therefore, the output is sorted primarily by the record_id and secondarily by the start_time. At this point the alignment is adjusted.

### 7.6.3 Alignment distribution

The alignment file is split by record_id into files named by the ID, each containing alignment of phones from the audio with the ID used. These files are then distributed from the exp directory, in which they were computed, to corresponding poems in the dataset directory. Now each alignment is in the same directory as the poem text file and audio recording partitions of the corresponding poem. During this step the poem transcription in format: record_name text in the partition... is stripped of the record name and saved in a different file as poem’s text.

The words-to-phones information (described in Subsection 4.2.3) is computed from each alignment file. An example can be seen in Listing 7.4. The phone representations from an alignment file are merged into a space separated list of phones forming a word. This is done, using the positional suffix of the phones, by the phones_to_pron.py script. Each phone also obtains an ID pointing to itself in the alignment file. This creates a file where each line is the created list of phones for a word (an example can be seen in Listing 7.5). Now these phone lists are mapped to words they represent. This is done by the pron_to_words.py script. To do this a lexicon file is needed (an example can be seen in Listing 7.6), it is provided in the dict directory, in the lexicon.txt file by the LibriSpeech authors. An example can be seen in Listing 7.7. The words to phones file is now constructed and saved in the directory containing the corresponding poem.

Listing 7.4: An example of word-to-phones data.

```
0.25 0.28 W_B
0.53 0.1 AH1_I
0.63 0.22 N_E
```
### 7.6.4 Prosody features computation

Pitch is computed using a Kaldi utility `compute-kaldi-pitch-feats`. The output is saved in Kaldi’s own format. As with alignment the pitch is computed for each poem audio partition separately. The format is as follows: the file is composed of segments each corresponding to one audio partition, each segment delimited by square brackets and labeled with the audio partition name. Therefore, the segments of the same poem have to be merged together. Moreover, each segment contains two columns, values in the first column are ignored, values in the second column are pitch values in Hertz (Hz). Each row corresponds to values computed at a certain time step (described in Subsection 4.2.2). To merge the segments and distribute the result the universal `process.pitch_energy.py` script is used. The constructed pitch file is saved in the corresponding poem’s directory.

Energy is computed using a Kaldi utility `compute-mfcc-feats`. The output is saved in the same format as pitch only each segment has thirteen columns. Where the values in first column is energy, other columns are ignored. The same script `process.pitch_energy.py` is used to merge the segments and distribute the result. The constructed energy file is saved in the corresponding poem’s directory.
7.6.5 Data points preparation

The data_preparation.py script is used to prepare the data points (we remind, that a basic datapoint consists of a phone ID, its duration, average pitch and average energy, the whole process is described in Section 4.2). For each poem the path to the containing directory is constructed. The paths are sampled, based on user’s arguments, into train, validation and testing data segment. For each path in each segment the poem reading is constructed by loading of the precomputed values (described in Subsection 7.5.5, shown in yellow in Figure 7.20). The readings are accumulated into three lists, the training, validation and testing data.

Each of the lists is processed by a data preparation method, which is picked according to user’s input. The format of training data is described in Section 4.1 and the data preparation methods are described in Subsection 7.6.9. Finally, the results of the data preparation method for each of training, validation and testing data, are saved in csv format files. The data points are now prepared.

7.6.6 Training

All training is done using the Keras API running on top of the TensorFlow framework. Data transformation is done using the scikit-learn package. The train.py script is responsible for training and data point transformation for each training method. Each training method is unique, but all follow the same structure. First, several possible neural networks are defined, one of them is used in the training process based on user’s input. An unique identifier of the model that will be trained is constructed. All of the metadata is constructed. The train data is loaded from the file constructed in the previous step. The train data is transformed. The validation data is also loaded and transformed. Optional methods, which are called during the training (Keras callbacks) are created. The training and validation data are used to train the network. The trained network is saved, all objects used during the data transformation are saved (they will be needed during prediction as well). All metadata is saved in a file. Specific methods used are described in Subsection 7.6.10.

7.6.7 Incorporation of the model into the software

The software has a designated directory for models. The metadata file is put into this directory along with any other necessary files (data transformers, network weights etc.). The metadata file is read during prediction and handles everything automatically.

7.6.8 Metadata format

Each model is described by metadata, which are created during the training of the model. Metadata are used to allow easy integration into the software (the files only have to be copied into the models directory, everything else is automatic). They are needed to pick correct model loading and handling methods during prediction. Moreover, each training method can save any data it deems necessary during the prediction. The metadata is saved, each variable
and its values on separate line in the format: `variable_name=variable_value`. An example can be seen in Listing 7.8. The metadata is parsed using the `parse_metafile_from_lines` method that accepts lines of the metadata file. It constructs a dictionary with the metadata variables. The `scaler` and `encoder` variables are treated differently. These variables are names of files containing special objects used during the training and prediction. The objects are loaded from the files and used as the value of the variables in the dictionary. The metadata can contain anything an author of new training and predicting methods would deem necessary, given that it can be saved in described format. However, there are several variables that are mandatory for every metadata file to allow integration into the application:

- **model_loading_method** is used to load the model, must be a method from `PhoneValuesPredictor` class,
- **predict_method** is used to make predictions using the model, must be a method from `PhoneValuesPredictor` class,
- **model_name** is used to show options to an user, and to identify the model,
- **description** is used to describe the model, shown in the front-end.

Listing 7.8: An example of a metadata file.

```python
model_loading_method=keras_rnn_load
predict_method=keras_rnn_one_hot_encoding_predict
model_json_file_name=keras_rnn_one_hot_33_window_no_overlap.json
model_weights_file_name=keras_rnn_one_hot_33_window_no_overlap.h5
scaler=keras_rnn_one_hot_33_window_no_overlap_scaler
encoder=keras_rnn_one_hot_33_window_no_overlap_encoder
seq_len=33
optimizer=adam
loss=mse
model_name=keras_rnn_31_epochs_one_hot_33_window_no_overlap
description=keras_rnn_31_epochs_one_hot_33_window_no_overlap
```

### 7.6.9 Data preparation methods

There are two data preparation methods. One called `trigram_data_prep` creates the trigram data points. It receives a poem reading, from which the phones are extracted. Triplets of phones are taken, they are looked up in the `global_phones_dictionary` and their IDs are used as output, moreover, prosodic features of the middle one are also used as output. The method returns a list of data points, each of length six.

The other data preparation method is the `single_phone_data_prep`. It receives a poem reading, extracts phones from the reading. Each phone is looked up in the `global_phones_dictionary` dictionary. It’s ID, and prosodic features are used as output. The method returns a list of data points, each of length four.
7.6.10 Training methods

Each training method receives all command line arguments. All the methods are contained in the train.py file. At the beginning of the file several networks are created (described in Chapter 5). The context of these methods is described in Subsection 7.6.6.

**keras_dense method**

First, a network is created. The identifier of the model is constructed from all the arguments. Metadata are created: all arguments are saved to metadata, names of future files containing the model are created and saved, the loading and predicting methods are saved. The trigram train set is read into a numpy array. The first three columns are used as input features. The other three as labels. If specified in the arguments, several callbacks are created. Namely, a logger callback, a callback reducing learning rate, a callback implementing an early stopping criteria and a callback saving the best model. The model is fitted using the loaded set. The network structure is saved, the weights of the network edges are saved using the HDF5 format and the metadata are saved.

**keras_rnn_id_normalized method**

First, a network is created. The identifier of the model is constructed from all the arguments. Metadata are created: all arguments are saved to metadata, names of future files containing the model and scaler are created and saved, the loading and predicting methods are saved. The basic train set is read into a numpy array. All four columns in the train set are used to fit a normalizing scaler (using RobustScaler from scikit-learn) and subsequently normalized. The first columns is used as input features, others as labels. Input features and labels are each transformed into a list of sequences of length specified in arguments. Sequences are overlapping a certain length specified in the arguments. If the train set cannot be divided into these segments without a reminder, the train set is padded. The end-of-poem phone is used as padding. The to_strided_sequences, from_strided_sequences and get_padded methods implement the segmentation, padding and desegmentation, which will be used during prediction (the sequences are implemented using a rolling window, more in Section 5.1). The same process is carried out for the validation set, only the scaler is not fit again.

If specified in the arguments several callbacks are created. Namely, a logger callback, a callback reducing learning rate, a callback implementing an early stopping criteria and a callback saving the best model. The model is fitted using the loaded sets.

The network structure is saved, the weights of the network edges are saved using the HDF5 format. The metadata is saved and the scaler used to transform the sets is saved.

**keras_rnn_one_hot method**

The process is the same, as in keras_rnn_id_normalized method, apart from certain differences. Instead of normalizing all four columns only the three later (corresponding to labels) are normalized. The first one (corresponding to input...
feature phone ID) is encoded in one hot encoding, using as many classes as there are phones (described in Section 4.1). Then both input features and labels are transformed into sequences as in the previous method. The same is done with the validation set. At the end the encoder is also saved as metadata.

7.6.11 Prediction methods

Prediction methods directly correspond to training methods and knowledge of those methods is essential to understand this section. We would like to emphasize that prediction methods are not part of the training pipeline, instead they are part of the main software. Their usage is to understand the trained models, in what format should the input be etc. We list them here, because they are tightly bound to the training methods. Each training method needs its equivalent prediction method, therefore they are discussed close to each other; however, we will would like to remind, that they are not part of the pipeline.

The correct prediction method is picked based on the metadata file associated with the model name. Each prediction method receives a list of phone IDs and returns a list of Prediction instances. Each instance corresponds to an ID. Each prediction method must return the same number of predictions as the number of phone IDs it received. The context of these methods is described in Paragraph 7.5.3.

**keras_dense_predict method**

This method is paired with the keras_dense method. Phone IDs are passed to the method, the IDs are transformed and serve as input features. The phones are aggregated into triplets, in each triplet the left and right ID overlaps with the left and right neighboring triplet respectively. The middle ID overlaps with both. The format is described in Subsection 4.2.5. The triplets are used as input features and are fed into the model. During triplets construction the very first and last phone IDs do not have their own triplet (they are never the middle ID), therefore, there are two predictions missing, a prediction for the first and last phone ID. These phone IDs correspond to the <START-POEM> and <END-POEM> custom phones. For these phones neither temporal nor prosodic features matter, therefore two more Prediction instances with zero values are generated. The predictions are then returned.

**keras_rnn_ids_normalized_predict method**

This method is paired with the keras_rnn_id_normalized method. Phone IDs are passed to the method and act as input features. The phone IDs are normalized using the same scaler as in training. The phone IDs are structured into segments as described in the corresponding training method. If the number of phone IDs cannot be divided into segments of the same length and with the same overlapping as were the input features during training of the model, then the phone IDs are padded. The <END-POEM> phone is used as padding. Segmented input features are fed to the model. The predicted labels are desegmented. The predicted labels are denormalized using the same scaler. Prediction objects carrying the prediction
are constructed from the labels and accumulated in a list. Predictions for padded input features are cut off and the predictions are returned.

**`keras_rnn_one_hot_encoding_predict` method**

This method is paired with the `keras_rnn_one_hot` method. Phone IDs are passed to the method and act as input features. The IDs are encoded into one hot encoding using the same encoder as during the training. The encoded IDs are structured into segments as during training. If the number of phone IDs cannot be divided into segments of the same length and with the same overlapping as were the input features during training of the model, then the phone IDs are padded. The `<end-of-poem>` phone is used as padding. Segmented encoded IDs are fed into the model. The predicted labels are desegmented. The labels are denormalized using the same scaler that was used on labels during the training. Prediction objects carrying the prediction are constructed from the labels and accumulated in a list. Predictions for padded input features are cut off and the predictions are returned.

### 7.7 Front-end and user interface

![Diagram](image.png)

**Figure 7.22: Overview of the front-end architecture. We describe the components in the direction of the arrow.**

The front-end is realized by a web application with one module. The application is controlled by a `ReadingTransformationController` controller. The front-end provides a user interface using a web-page. The web-page is generated and data are retrieved from it using a collection of views. Communication between
the views and the user interface is completely handled by a server. The data flow between the front-end controller and the views is given in Figure 7.23. The front-end controls the software. It calls methods from the `front_end_utils.py` file described in Subsection 7.8.1. These methods initiate all available computations returning three types of data:

- all available models in the format of a list of pairs, each pair consists of a model’s name and description,
- all available examples in the format of a list of pairs, each pair consists of an example’s ID and name,
- a reading value fully describing the reading of the custom or example poem, an instance of the `PoemReadingValue` class (described in Section 7.5).

The whole architecture can be seen in Figure 7.22.

### 7.7.1 Index page

A user can request only the index page. The index page consists of a form which the user can use to request generation of reading for a certain poem. To construct the page a universal HTML header and instructions are included. Then the form is rendered. The `InputForm` class is used to construct the form. This form is made of three fields. The `input_pick` field is a choice field populated with all available example poems and one special choice, the choice to input user’s own poem. This field is represented by an HTML `select` element. The `model` field is a choice field populated with all available models and one special option for example poems, an option to just read the pre-computed values instead of predicting them. This field is represented by an HTML `radio` select element. Lastly, the `poem_text` field is a character field used to input user’s own poem. This field is represented by an HTML `textarea` element. This form has a submit button. When the submit button is pressed the form is submitted using the `POST` method. Lastly, the universal HTML footer is included. There is a JavaScript script controlling the page, this script is described in Subsection 7.7.5.

The index page corresponds to an URL address with empty path segment (if the server is run on a local machine with default settings, then the index page
can be requested at \url{http://127.0.0.1:8000/}). If the page was not requested using the \texttt{POST} method, a new form is constructed and rendered. If the page was requested using the \texttt{POST} method it means that the form’s submit button was used. If the data in the form is not valid the form is rendered again with its fields filled up the same as they were when submitted. Otherwise, if the form is valid, the input pick ID, model’s name and poem’s text are encoded into a query segment of an URL address. This query is then appended to a result URL and the index page redirects to the constructed URL address. The index page is implemented in the \texttt{index} method representing the index view.

### 7.7.2 Result page

The result page corresponds to an URL with a path segment containing the word result and any valid query segment (if the server is run on a local machine with default settings, then the result page can be requested at \url{http://127.0.0.1:8000/result?input_pick=...&model_name=...&poem_text=...}). If the query does not contain all needed fields the page redirects to the index page. If the query is valid the input pick ID, model name and poem text are extracted. The result page is implemented in the \texttt{result} method representing the result view.

The extracted data is passed to an instance of \texttt{ReadingTransformationController} to construct a poem reading (an instance of \texttt{PoemReadingValue}, described in Paragraph 7.5.3) and to create a dictionary of reading transformations. If this process is not successful for any reason the result view redirects to an error page. The code called from the controller accesses \texttt{Tensorflow}'s computation graph, which cannot be used in parallel. Therefore, the access to \texttt{ReadingTransformationController} is synchronized. A lock is acquired before the access to the controller and released when the reading and its transformations are constructed. The constructed dictionary is sent to a \texttt{result} template, and the template is rendered as the result page. The whole process can be seen in Figure 7.24.

The \texttt{result} page visualizes the reading of the poem. First the universal HTML header is included, then a styling section is defined. JavaScript libraries are included. Two third-party libraries are used as well as our own script. First library is the \texttt{SubtitlesOctopus} capable of subtitles rendering. The other is the \texttt{chart.js} library capable of chart plotting. Our own script is described in Subsection 7.7.6. The subtitles are rendered on a \texttt{canvas} element. The page allows to playback the reading of the poem by visualizing it in several ways. The control of the playback is managed by our script. A \texttt{range} input element is used as the playback seeker (similar to seekers present in all media players), the maximum value is length of the poem, the step size is the granularity of the poem. Buttons to start, pause and stop the playback are defined, and functions are registered to run when the buttons are clicked. An option to change the rate of the playback is realized using a \texttt{select} element. Visualizations of the reading:

- karaoke style subtitles (described in Section 6.1, generated in Subsection 7.7.4 and rendered using the \texttt{SubtitlesOctopus} library) showing how and when to read which word of the poem,
- a table showing prosody values for each word of the poem (described in
Section 6.2 and Subsection 7.7.4 and plots of pitch and energy (plotted using the chart.js library when the page is loaded),

- an audio record of the poem (either generated by a TTS system or read from the original record) is played.

The mute button allows to mute the generated audio. The audio is sent in the base64 string encoding, which the HTML audio element can natively play. The download button allows the user to download the record. All three visualizations are controlled by the same script.

Input pick ID, model name and poem text from the user
input is valid
Wait to acquire the lock
A poem reading, an instance of PoemReadingValue
A dictionary of poem reading transformations
Reading transformations module
Create the transformations dictionary
Get poem reading from the back-end
Pass the data to the front-end controller
Render the result page
Release the lock

Figure 7.24: Overview of the front-end process.

The Django HTML templates are used to render the web-pages. Django allows substituting of specific statements in the templates with an item from a dictionary, which is passed to the template. This is primarily used to construct the result page from all reading transformations.

The ReadingTransformationController class is responsible for dispatch of user’s input to the back-end and subsequent communication of the returned poem reading to the transformation module. The dispatch of user’s input is done using the compute_reading_from_input method which is part the communication API (described in Section 7.8). The method returns a poem reading.

### 7.7.3 Transformations module

The poem reading is passed to the transformations module. The data flow between the front-end controller and the transformations module can be seen in Figure 7.25. The reading_transformers list is used to register transformers that will be applied to the poem reading. The application of the transformers creates a dictionary of transformed readings where keys are supplied by transformers as well as the values. The individual transformers are described in Subsection 7.7.4. The dictionary is then returned.
A dictionary of poem reading transformations

Figure 7.25: Overview of the data flow between the front-end controller and the poem reading transformations module.

### 7.7.4 Data transformers

A collection of transformers used to transform a poem reading into a different format. These transformations are then used to render the reading by various technologies on the result web-page.

#### class Transformer

An abstract class used to describe the transformers API. The `get_name` method supplies the name of the transformed reading. The names are used as keys in the dictionary constructed by the controller. The `transform` method takes a poem reading and transforms it into a different format. This method can also serve to extract features of the reading.

#### class StepTransformer

The `StepTransformer` class extracts the feature granularity, this is used as the size of step by the web-page player.

#### class RecordTransformer

The `RecordTransformer` class transforms the poem’s recording from binary format into a base64 string encoding.

#### class ModelNameTransformer

Extracts the name of the model used to predict the poem reading being transformed.

#### class LengthTransformer

Extracts the length of the poem from the poem reading.

#### class RecordLengthTransformer

Extracts the length of the poem audio recording.
class SortedTimesTransformer

Creates a list of verse ending times and transforms it into JSON format. This is used by the web-page to highlight the currently read verse.

class PlotsArgumentListTransformer

Creates a structure used to lookup pitch and energy values for each word. This is used by the web-page to visualize the poem reading using plots. The format of the structure is that of a list (list A) of pairs of lists. Each item in the list A represents values for one word. Each of these items is a pair (represented by a list of length two). The first item in the pair is a list of pitch values for the word. The second item in the pair is a list of energy values for the word. This structure is transformed into the JSON format and returned.

class TableTransformer

An HTML table is created representing the whole poem. Each line in the table corresponds to one verse in the poem. Each cell in a row corresponds to one word in the verse associated with the row. In each cell word’s maximum energy, average energy, average pitch, distance between maximum and minimum pitch, starting time, ending time and duration are displayed. Moreover, canvases for pitch and energy plots are created, these are filled up later by the result web-page script.

class SubtitleTransformer

This class creates SubStation Alpha subtitles used to visualize the poem. The header is loaded from a file. Verses of the poem are aggregated, for each verse two lines of subtitles (corresponding to the two segments described in Chapter 6) are generated. Times for each verse are generated using the transform_time method. The transform_time method takes time in seconds and transforms it into a string representation in format: hours:minutes:seconds:milliseconds. The color of separate phones (describing the pitch) is generated by the build_color method. It creates a color descriptor in SSA format for any object providing its average pitch. Based on the input, one of the red, green or blue fields is FF and other two are the average pitch. The pitch is first normalized using the normalize_pitch method and transformed into hexadecimal by the into_hex method. The size of separate phones (describing the energy) is generated using the normalize_energy method.

Both normalizing methods use the same principle. They map the value of pitch or energy which lies in a smaller interval to a value in a bigger interval. The values of pitch are usually in \([120, 210]\) interval, the values of energy are usually in \([15, 25]\) interval. If any value is not from its interval then the minimal or maximal value from the interval is taken. The bigger interval used for pitch is \([0, 255]\), values from this interval correspond to a byte segment of color. The bigger interval used for energy is \([30, 65]\), values from this interval correspond to sizes of a font. The function

\[
f(t) = c + \left( \frac{d - c}{b - a} \right) (t - a)
\]
is used to map the interval \([a, b]\) onto the interval \([c, d]\). This function satisfies 
\(f(a) = c\) and \(f(b) = d\). In our case for pitch: \(a = 120, b = 210, c = 0\) and \(d = 255\); and for energy: \(a = 15, b = 25, c = 30\) and \(d = 65\).

### 7.7.5 Index script

Knowledge of the form structure rendered on the index page is needed to understand this description. The only functionality of the script is to disable and enable input fields based on currently picked options. Cases in which an action is needed are:

- when the submit button is pressed it is checked whether the poem_text field is empty while the own poem option in input_pick field is chosen, that would mean the user is trying to submit an empty poem, this is prevented,
- the poem_text input field is disabled if an example poem ID is chosen in the input_pick field (because the text of the poem is read from the database),
- if the option in the input_pick field to use an own poem is selected, the radio button in the model field associated with the option to just-read-the-precomputed-values for the poem is disabled (because the values are only precomputed for the example poems), otherwise it is enabled.

### 7.7.6 Result script

Knowledge of the result page (described in Subsection 7.7.2) is required to understand this section. The result script is responsible for plotting of the values and for control and setup of the playback. First it plots the pitch and energy values using the chart.js library. Canvases for the plots are already generated in the table. The createPlots function loops over all canvases. These canvases are implicitly numbered based on their position when read from left to right and from the top to the bottom, the numbering starts at zero. These positional indices are used to lookup in the plotArgumentLookup structure which is build using the same type of loop in Subsection 7.7.4. For each word lists of pitch and energy values are retrieved. The canvas and the values are then passed to the makePlot function. The makePlot function first creates two unique identifiers using a number generator. It creates empty labels for the plots as we do not need any description of the values, but without the labels the plots would break. It uses the labels and given values to build datasets, that can be fed into a library function which plots the values. The values are plotted.

The download function is called when the download button is pressed, it creates a new a element which it uses to download the record.

The options variable is used to setup the SubtitlesOctopus subtitles renderer. This is done only once.

The control of the playback is managed by numerous variables and methods. Variables used in the script:

- sortedTimes are used to lookup the currently read line so it can be highlighted in the table, this is done by the search function,
plotArgumentLookup is used to lookup arguments for functions that will plot the prosodic values for each word, the structure is described in Subsection 7.7.4.

now represents the current time of playback,

interval holds the JavaScript interval object, this object calls the forward function in intervals lasting the amount of milliseconds stored in the forwardBy variable (for example every 10ms),

poemLength holds length of the poem in milliseconds,

poemRecordLength holds length of poem’s audio,

playbackLength holds the maximum out of poemLength and poemRecordLength, this is to mitigate the possible impairment of audio with the visualization,

poemLengthStr holds the length of the poem in the format hours:minutes:seconds:milliseconds,

baseForwardBy holds the base rate of playback from which all slow downs and speed ups are computed,

forwardBy holds the actual computed playback rate,

seeker is a reference to the playback seeker,

record is a reference to the audio element,

playbackTimer is a reference to a field showing the time of the playback,

currentLine is the number of the currently read line, lines are numbered from zero,

defaultColor is the color used to mark lines that are not currently read,

highlightColor is the color used to highlight the currently read line.

The variables are global and are used by these functions:

start function creates the interval object if it does not exist, and starts the playback of the audio file,

pause destroys the interval object, and pauses the playback of the audio file,

init sets the now variable to zero, effectively setting the playback to beginning, calls the update function,

stop calls the pause and init functions, then rewinds the audio record to the beginning,

forward forwards the time by the amount of milliseconds stored in the forwardBy variable, then calls update,
• **update** updates all elements used in the playback loop, it sets the current time to the canvas, sets the current time on the playback seeker, sets the current time in the playback timer, and highlights the current line in the table,

• **setPlaybackTimer** sets the playback timer’s value to the string representation of current time produced by the `parseTime` function,

• **parseTime** is used to parse time in milliseconds to the string format: hours:minutes:seconds:milliseconds,

• **seek** is run when the playback seeker is manually changed (when the user seeks in the playback), it sets the current time to the value of the seeker,

• **mute** is used to mute and unmute the audio playback,

• **search** implements a linear search used to lookup the currently read line,

• **setLine** searches which line is being read, and highlights it accordingly.

### 7.7.7 Server

A Django development server is used to communicate between the user interface and the views. While this server should not be used under heavy load in production, it is more than sufficient to handle the load of a desktop application. While the deployment of the application on an actual production server should be straightforward, no setup script to do this is provided as there was no production server for testing of the setup. The application is independent of the server it runs on.

The application uses a structure to dispatch user’s requests to correct views based on the path segment of an URL address. Each view corresponds to an unique path segment of an URL address, and each view corresponds to a unique page. The path segment is provided by the server. The structure is a list of Django path objects (`urlpatterns` variable in `expressive_poetry_reader/urls.py`).

### 7.8 The Communication API between front-end and back-end

Communication is possible only in one direction because the front-end is dependent on the back-end while the back-end cannot be dependent on the front-end. While the front-end can technically access any part of the back-end as the privacy of access is not easily enforceable, only a small subset of back-end methods is to be used by the front-end to obtain information from the back-end. The back-end is not allowed to communicate with the front-end in any way. All back-end methods that can be called from the front-end are located in the `front_end_utils.py` file. The API can be reviewed in Figure 7.26.
7.8.1 front_end_utils.py

The methods in the `front_end_utils.py` file initiate all available computations and return three types of data:

- all available models in the format of a list of pairs, each pair consists of a model’s name and description,

- all available examples in the format of a list of pairs, each pair consists of an example’s ID and name,

- a reading value fully describing the reading of a custom or an example poem, an instance of the `PoemReadingValue` class (described in Section 7.5).

All queries from the front-end must be directed at methods from this file. There are two methods used to list which models respectively examples are available: `get_available_models` and `get_available_examples`. The `compute_reading_from_input` method passes the input to the back-end controller and gets the poem reading from it. Finally, the `get_id_of_own_poem_input` method is used to inform the front-end which `input_pick ID` is associated with the option to input an own poem (this is used on the index page given in Subsection 7.7.5).
8. User guide

First we would like to clarify whom we imagine as a user of this software. This software is not aimed at any specific end user, instead we would like to imagine a user-developer to take interest in this work. We should define what can such a user-developer do. From now on we will use just the term user. The user of the main application should be able to use command line, and be able to install certain software and make it accessible in the \texttt{PATH} variable. If the user would like to recreate the whole experiment, non-trivial knowledge of software compilation and installation is required. This is caused by the dependency on Kaldi ASR toolkit.

The main use-case of this software is prediction of a reading for a poem and subsequent visualization of the reading. However, the user can easily train their own models on our data. The user can also train on their own data. Depending on the data, this might be very complex.

The usage of the software can be described in several phases. A phase in which dataset is prepared (dataset phase). A phase in which data points are prepared (data points phase). A training phase. And finally, a phase in which reading is predicted for poems supplied by the user and visualized (prediction phase, this is the main use-case). The phases will be described in detail later, now we will review dependencies for each phase.

This chapter formally describes the software use. For a more tutorial style guide the user should refer to the \texttt{README} file in project’s directory.

8.1 Dependencies

This software should be run on a Linux system because there are several shell scripts used, and it is very difficult to make the third-party software used run on a different platform. However, it is probable that it can be made to run under Cygwin or MinGW.

8.1.1 Dataset phase dependencies

The dataset phase is the most demanding. The main dependency is Kaldi ASR toolkit, which must be installed and configured. This must be done by the user as it cannot be automated in any reasonable way. We suggest following the official guide (available at \url{http://www.kaldi-asr.org/doc/install.html}) and instructions in \texttt{INSTALL} files in the repository. During the compilation one will most likely run into problems with the \texttt{ATLAS} library used for linear algebra, either there will be a problem with linking or there will definitely be a problem during compilation. It will require the user to turn off the CPU throttling. We suggest using the \texttt{OpenBLAS} library instead as it does not cause any of these problems, the guide to do so is available at \url{https://kaldi-asr.org/doc/matrixwrap.html}. The compilation will also require a C++ compiler and other usual development tools available on most systems.

While running Kaldi scripts, sometimes they can abort with a message that certain software is not installed, the message also includes instructions how to
install the missing software. The user should follow these messages. The process
cannot be fully described as each system is different and different problems can
occur. During this phase several utilities available on all standard systems are
used, therefore, we will not describe them here. The rsync utility is used to
synchronize directories. The wget utility is used to download the corpus. The
CPython implementation of the Python language is required, the version 3.6.5
is the lowest tested, but any 3.* version should work. A standard R language
interpreter is required, the lowest tested version is 3.6.0, but any version should
work. Finally, if the user would like to create their own examples to put in the
prediction application, the sox and ffmpeg tools are required, any version should
suffice. The user should install all described software in steps specific for their
system.

8.1.2 Dependencies of other phases

All remaining phases have the same dependencies. The CPython implementa-
tion of the Python language is required, the version 3.6.5 is the lowest tested,
but any 3.* version should work. pydub, httplib2, scikit-learn, joblib,
nltk, keras, numpy, moviepy,jsonpickle, django, g2p_en, tensorflow and
python-Levenshtein Python packages are required. These are listed in the
requirements.txt file in the root of the project and can be installed using a
pip install -r requirements.txt command. The user should take into ac-
count that on their system the command can be pip3 and that it might require
--user option following the word install, but one should only take care of this
themselves if they decide not to use the provided setup script, which will do this
for them.

The prediction phase in addition needs the MaryTTS system to be present
on the system, it does not require any installation but it must be downloaded.
It is available at [http://mary.dfki.de/download/index.html](http://mary.dfki.de/download/index.html) however, if one
uses the setup script it will be downloaded for them and put into the marytts
directory in project’s root. The TTS system needs a JavaVirtualMachine to run.
Finally, the ffmpeg tool is required.

8.2 Setup

Git can be used to clone the software repository. The command git clone
https://gitlab.com/JanVykruta/expressive_poetry_reader.git will create
software’s directory. If the user already has the software directory (can be ob-
tained from the software.zip archive), there is no need to run this command.
We expect that the user has put the software directory into their home directory
(~/), and we believe that the user will be able to adjust all instructions if they
chose to save it elsewhere. We expect all dependencies from the previous section
to be fulfilled unless it is mentioned that they can be installed using a setup
script.

There is no setup for the dataset phase (apart from the dependencies). All
other phases require further setup. To set up the other phases the setup.sh
script located in the root of the project should be run. The script will prepare
everything one may need during any of the phases. Only several voices for the

71
TTS system are installed, if the user would like a different voice, it has to be installed using the marytts/bin/marytts-component-installer script.

8.3 Settings

All settings used during the prediction phase are located in /expressive_poetry_reader/expressive_poetry_reader/settings.py. The BASE_DIR variable is used throughout the project to refer to the directory containing the project. The MODELS_PATH_DIR contains all models and their metadata files. The TEMP_DIR is used for all temporary files. The MARYTTS_HOSTNAME and MARYTTS_PORT specify where should be the requests to TTS sent. The MARYTTS_VOICE variable is used to pick the voice used by the TTS system. DEBUG can be set to True if the user would like to modify the project. ALLOWED_HOSTS contains all URL addresses where this application can be hosted. All other variables are part of the Django toolkit, we expect that anyone who would need to change them is an experienced Django user.

8.4 Dataset Phase

The dataset phase is the most demanding on resources as well as user’s expertise. We strongly suggest avoiding this phase if at all possible. It requires up to 250GB of free space and is very time-consuming. Especially Kaldi installation and corpus acquisition.

We will assume Kaldi is located in ~/kaldi. To recreate this phase one should copy the run.sh script located in manual_setup in project’s directory to ~/kaldi/egs/librispeech/s5 and overwrite the run.sh located there. Then, the user should move to the directory and run the copied script with ./run.sh. There might be some errors, which must be resolved by the user, and the script rerun. The run.sh script can be run with --stage n argument, where n is a number, the script will not run steps with lower numbers than n. Functionality of each step can be reviewed in the script itself. Reruns of the script should only be necessary if there was an error during a previous run.

The script also creates the dataset. It is by default saved in ~/kaldi/egs/librispeech/s5/dataset/LibriSpeech/filtered, this can be changed in the script itself (the user should look for the data variable).

The alignment is computed, now it has to be transformed and distributed to the dataset. Before a handling script can be run, it must be properly configured. The paths.sh script in dataset_preparation in project’s directory contains all the settings. Only the KALDI_ROOT and DATA_ROOT should be changed. The first should be the path to a Kaldi directory, the second should point to the dataset downloaded in the previous step. Other options should be changed only with understanding of the consequences.

When the settings are configured, the adjust_forced_alignment.sh script can be run. It will transform the alignment and distribute it to the dataset. Afterwards the dataset can be moved at will and the Kaldi directory deleted. The user can use their own data which are to be aligned, if the data is in the
correct format as described in Chapter 3, we cannot suggest this as we did not go through this step.

8.5 Before other phases

The rest of the phases are less demanding. We provide the output of the Dataset Phase in `after_dataset_phase.zip`, which was handed in with this work. If the user uses the repository, the instructions to obtain this file are described in the README file. We expect that the project directory has been set up as described in the Setup section 8.2. Before any of the phases, the `activate.sh` script has to be sourced. The `sh ./activate.sh` command can be used to achieve this. The script loads needed paths. Any actions must be done in the same terminal in which the script was sourced. When the user is done, the `deactivate.sh` script should be sourced, which restores the original paths, or the terminal can just be closed.

8.6 Data points Phase

We will use the word `set` to describe a collection of data points. The term `dataset` is used to describe the dataset created in the Dataset Phase. The `data_preparation.py` script located in the `learning` directory is used to create the training, validation and testing sets. The script allows several modifications of resulting sets (for example the sizes or the format), all options can be reviewed with the `-h` argument. The user can either use the prepared dataset from the previous phase (including the one from `after_dataset_phase.zip`), or supply their own dataset, given it has the same format (described in Chapter 4).

The sets we used can be recreated using the commands listed in Listing 8.1. First the `~/work` directory should be created. These commands will create the same training, validation and testing sets we used in the following steps. If the dataset was moved the commands have to be adjusted. The user is free to created a different type of data points. All they need to do is create a new method in the script. The method should take an instance of `PoemReadingValue` class and return a list of data points. Each data point has the format of a list of values that represent the data point. Then the method has to be registered in the `data_point_formats` dictionary with a string that would describe the format and that will be used as the value of the command line argument.

We provide the created sets in `training_sets.zip`. The git user should refer to the README file.

```
Listing 8.1: The commands we used to create the sets.

```

```bash
python3 data_preparation.py \codetilde/kaldi/egs/librispeech/s5/dataset/LibriSpeech/filtered \codetilde/work basic --stable_seed
python3 data_preparation.py \codetilde/kaldi/egs/librispeech/s5/dataset/LibriSpeech/filtered \codetilde/work trigram --
stable_seed
```
8.7 Training phase

For now we will assume, the user uses the same data point format as we did. The script `train.py` located in the `learning` directory is used to train all the models. The `train_set_path` argument is the path to the training set created in the previous phase. Analogically for validation set. All the models and their metadata files will be saved in the directory specified by the `output_dir` argument. The name of the learning method has to be specified using the `learning_method` argument. Other training parameters can also be specified. The available training methods are specified in the `learning_methods` dictionary.

We have trained many models, the corresponding commands can be seen in and run with the `recreate_models.py` script in the `learning` directory. The user can create their own method, the method has to be implemented and registered in the dictionary of learning methods. The model trained using a custom method needs to have its metadata file to be integrable into the prediction software. There are several mandatory metadata variables describing the model, they can be reviewed in Subsection 7.6.8. The training method is responsible for creation of the metadata.

The user can also create new network structures. The network structure should be returned by a method, and the method registered with a name of the structure in the `models` dictionary in all applicable learning methods.

If the user decided to use a different data point format they will have to implement their own training, prediction and potentially testing methods as well as design their own network structures. While the training method can do practically anything, one has to take into account the prediction method which imposes a limitation. The prediction method can only receive phones of a poem, therefore, the training method is limited to use only phones during training, otherwise the resulting model would not be usable during prediction.

Eventually, all created models with all their metadata files should be moved to the directory specified by the `MODELS_PATH_DIR` variable in settings.

8.8 Prediction phase

We will assume the application is run on a local machine. The `python3 manage.py runserver` command will run the server. The application can be accessed locally and used following shown instructions.

If the user did not set up any non-local TTS server in the settings, a local TTS server has to be run. This can be achieved by running the `~/expressive-poetry_reader/marytts/bin/marytts-server` command in a separate terminal session. The local server will now serve all requests from the application given the default settings were not changed.

If the user would like to add their own example poems, the corresponding files should be moved to the `examples` directory. The examples have to be registered in the database, the code doing this can be seen in the `setup.py` file.

If the user would like to create another visualization, they need to create a class implementing the `Transformer` interface. Then they need to register an instance of the class in the `reading_transformers` list of transformers in the `data_transformers.py` file. Finally, they need to use the transformation in the
result.html template. The transformation can be accessed in the template using the name returned by the `get_name` method of a transformer. We suggest studying the Django tutorial available at https://docs.djangoproject.com/en/2.2/topics/templates/ as well as the code in the `web_demo/data_transformers.py` file. All templates are located in the `web_demo/templates` directory and can be modified in any desired way.

If the user decided to create their own training methods, the user also has to provide the corresponding prediction methods. These should be implemented in the `PhoneValuesPredictor` class in the `learning/predict.py` file. A prediction method should take a list of phone IDs and return a list of `Prediction` instances, the `Prediction` instances should correspond to the IDs. The prediction method is almost entirely dependent on the corresponding training method. The prediction phase is the most common case of usage. A quick-start example can be seen in Listing 8.2.

Listing 8.2: An example of the simplest and quickest usage.

```bash
# install dependencies:
# sudo pacman -S python3 ffmpeg git jre11-openjdk
# sudo apt-get install python3 ffmpeg git default-jre

# run setup (this should be run only once):
# cd ~
git clone https://gitlab.com/JanVykruta/expresive_poetry_reader.git
cd expresive_poetry_reader
./setup.sh

# this should be run every time the application is run:
# in different terminal session run:
~/expresive_poetry_reader/marytts/bin/marytts-server

# back in the initial terminal session (or in a new session if there was no setup):
# . ./activate.sh
python3 manage.py runserver
# wait for a little bit
# now the application can be accessed at http://127.0.0.1:8000/
```
Conclusion

We have successfully developed an application that receives a poem from a user, generates its reading and visualizes it. The application is easily accessible once it is set up using a web-page. The reading is visualized in two different ways: by subtitles that show when each phone should be read and what should be its pitch and energy, and by an HTML table that shows plots of pitch and energy of each word. The subtitles are intended to visualize the temporal features more, and the table to visualize the prosodic features more. Moreover, an audio record is generated, which uses the temporal features and pitch values, we, however, did not manage to use the energy values in audio generation.

The application is easily extendable and allows a more advanced user to modify or replace sections of the application. To communicate with the back-end a defined API is used, therefore, the front-end can be completely replaced. An user interested in the whole process needed to build prediction models can recreate and modify all experiments. The whole process is automatic. However, to recreate the very first phase of the process non-trivial dependency has to be resolved manually, which requires a user familiar with software compilation. This dependency is caused by the Kaldi toolkit, which cannot be easily replaced. Therefore, to avoid the first phase, data output by the first phase are provided for the user, allowing them to continue with much simpler phases.

We used a machine learning approach to generate expressive reading of poems. We utilized LSTM neural network architecture, which is designed for usage with time series data. As there is no definitive measure of how expressive a reading is, we left the evaluation to the reader. We will, however, provide our subjective evaluation. In our opinion the models managed to learn how to predict energy values well. We also believe that predicted duration times are reasonable. The models differ mostly in prediction of pitch values. Some of the models failed to learn anything and predicted very monotonous values. Others managed to predict interesting, albeit not very accurate values. Few models predicted reasonable pitch values. We believe it is due to small dataset and or due to the quality of the dataset.

We have fulfilled our goals, we created a functional application, that can be used as a toy, and we tested whether we can use machine learning to generate expressive reading of a poem. Our conclusion is that expressive reading can be generated using machine learning, however, a better dataset is needed for better results. We believe that proper punctuation and original poem structure would greatly improve model performance.

While we fulfilled our goals, there is more work to be done. The application should be deployed to a server, this would make its usage very user-friendly, however, we did not have a suitable server. The playback on the result page of the website could be improved, currently there is a problem when the seeker is dragged. While the problem does not affect functionality it degrades the performance. A better dataset could be prepared and used with our data point creation as well as with our training methods. We have our eyes on Lyrikline project available at [https://www.lyrikline.org/](https://www.lyrikline.org/) The project hosts free-verse poetry and audio records of poems read by the authors themselves. The data
are not freely available, but according to the authors are willing to cooperate and would allow usage of their databases. Finally, different front-ends could be used with our back-end. For example one that would produce images instead of a webpage. Or a user interface, that would work as a desktop application.

We hope that our work will be interesting and stimulating for (not only) poetry fans in computer science and beyond. We would also like to express a wish that someone would find pieces of this work useful in their own project.
Bibliography


François Chollet et al. Keras. https://keras.io, 2015.


78


Alexei V. Ivanov, Patrick L. Lange, David Suendermann-Oeft, Vikram Ramnarayan, Yao Qian, Zhou Hao Yu, and Jidong Tao. Speed vs. accuracy: Designing an optimal asr system for spontaneous non-native speech in a real-time application. 2015.


Kyubyong Park and Jongseok Kim. g2pe. [https://github.com/Kyubyong/g2p](https://github.com/Kyubyong/g2p), 2019.


Robert Weide. The carnegie mellon pronouncing dictionary [cmudict. 0.6], 2005.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The overview of architecture of this software. The arrow symbolizes the data flow. The dotted line is the border between front-end and back-end.</td>
</tr>
<tr>
<td>2</td>
<td>The overview of the learning pipeline.</td>
</tr>
<tr>
<td>3</td>
<td>An example showing the visualization of a poem. At the top are rendered the subtitles. On the bottom is the table with plotted values.</td>
</tr>
<tr>
<td>2.1</td>
<td>Overview of a recurrent neural network. The x’s denote input, h’s output and A’s the unrolled networks. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of a basic RNN structure. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.3</td>
<td>An overview of a LSTM network. The C’s denote the cell state, the h’s the previous and current states. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.4</td>
<td>The first step of LSTM building block operation. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.5</td>
<td>The second step of LSTM building block operation. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.6</td>
<td>The third step of LSTM module operation. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>2.7</td>
<td>A LSTM building block variation allowing sigmoid layers to look at the cell state. The picture is taken from Olah [2015].</td>
</tr>
<tr>
<td>3.1</td>
<td>An overview of files contained in a poem directory. Files ending with .flac are parts of the split audio recording. The file with trans in it is the transcription file.</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of a directory with one poem. The file with ali in name is the alignment file, the file with trans in name is the transcription file and the file with wtop in name is the Words-to-Phones file.</td>
</tr>
<tr>
<td>5.1</td>
<td>An overview of windows and window steps (strides) used to create data point sequences fed into recurrent networks. Available data points are in green, padding is in gray. The blue braces are windows of size 4 with stride of size 2. The red braces are windows of size 3 with stride of size 3. The yellow braces are windows of size 7 with stride of size 5.</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of the SSA subtitles.</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of the HTML table showing segment of reading of the first stanza of the Divine Comedy.</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview of software structure. The arrow signifies the data flow and the dotted line separates front-end from back-end.</td>
</tr>
</tbody>
</table>
7.2 The overview of back-end segment. We will describe the structure in order following the arrow. The dotted line separates the front-end from back-end.

7.3 The functionality of the PredictorController class.

7.4 Back-end segment responsible for the prediction. The predictor is responsible for the first three modules and the TTS generation for the last.

7.5 The data flow between the constructor and the structural segmentation module.

7.6 Architecture of the poem structure segmentation module.

7.7 The data flow between the g2p module and the constructor.

7.8 Overview of the structure of the PhoneWithAlignmentWithFeatures class.

7.9 The data flow between the prediction module and the predictor.

7.10 The overview of PhoneValuesPredictor construction.

7.11 The overview of the prediction process itself.

7.12 The overview of an instance of the PoemReadingValue class.

7.13 The overview of the whole prediction process.

7.14 The overview of data flow between TTS module and mostly the front-end controller.

7.15 An overview of the audio generation.

7.16 The overview of the TTS module.

7.17 The SSML encoding algorithm.

7.18 Overview of the second step of loading.

7.19 Overview of the last step of loading.

7.20 Overview of the loading process. The yellow segment is reused by the training pipeline.

7.21 Overview of the data preparation, we will follow the arrows in our text.

7.22 Overview of the front-end architecture. We describe the components in the direction of the arrow.

7.23 Overview of the data flow between the front-end controller and the views.

7.24 Overview of the front-end process.

7.25 Overview of the data flow between the front-end controller and the poem reading transformations module.

7.26 Overview of the Communication API.
List of Tables

3.1 An overview of dataset statistics........................................... 19
4.1 An overview of training sets statistics.................................... 26

5.1 The overview of all models trained and what parameters were used. Values in model ID column are used to refer to models in the text. The transformation column is used to describe which transformations were applied. IN stands for ID normalization, FN for feature normalization and OH for one-hot encoding. The w. size column holds the size of the rolling window used, w. stride holds the step used to move the window. The epochs column describes the number of epochs used. Finally, the network column describes the network used (the values are defined in description of each network). 29
A. Attachments

A.1 Standard phones in English


A.2 Custom phones

The custom phones are: <START-POEM>, <END-POEM>, <DOT>, <COMMA>, <SEMICOLON>, <THREEDOTS>, <COLON>, <WHITESPACE>, <END-OF-VERSE>, <END-OF-STANZA>, <QUESTION-MARK>, <EXCLAMATION-MARK>, <PUNCTUATION-MIDDLE-OF-SENTENCE>, <PUNCTUATION-END-OF-SENTENCE>

A.3 Global phone dictionary


A.4 Provided files

The software.zip archive contains all source files, their structure is described in Section 7.2. The recordings.zip archive contains several generated readings, we found interesting and we suggest the reader to listen to them. The after_dataset_phase.zip contains the output of the Dataset Phase (described in Section 8.4). The training_sets.zip archive provides the sets we used to train our models. These archives can be either downloaded (if the user follows the README file) or they are contained in the archive, which is handed in to the SIS. We admit that it is not a good practice to put archives into another archive; however, we would like the README following user to be able to pick which files they need, while keeping the instructions uniform.