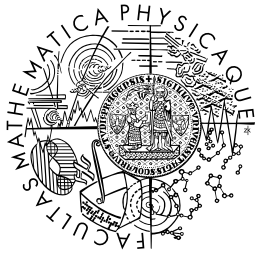

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Patrícia Březinová

Computational analysis and synthesis of song lyrics

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Mgr. Martin Popel, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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 ....

Author's signature

I would like to thank my supervisors Mr. Popel, and his predecessor Mr. Hajič, for their guidance, input, and a great amount of time for weekly consultations. I also want to thank Mr. Delmonte for his collaboration and numerous modifications of SPARSAR. Most of all, I want to thank my husband and family, for their support and help with my little son – I would not be able to finish this without them.

Title: Computational analysis and synthesis of song lyrics

Author: Patrícia Březinová

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Martin Popel, Ph.D., Institute of Formal and Applied Linguistics

Abstract: We explore a dataset of almost half a million English song lyrics through three different processes – automatic evaluation, visualization, and generation. We create our own rhyme detector, using EM algorithm with several improvements and adjustable parameters. This may, in some cases, replace human evaluators that cannot be used, for example, after each iteration of lyrics generator to evaluate its improvement. By creating a web-page visualization of the results with interesting matrix rhyme highlighting, we make our evaluation accessible to the public. We discuss interesting genre differences discovered by applying our automatic evaluation on the entire dataset. Finally, we explore lyrics generation using state-of-the-art GPT-2.

Keywords: song lyrics, automatic evaluation, rhyme detection, lyrics generation, GPT-2

Contents

Introduction

As artificial intelligence keeps catching up with humans, even despite numerous attempts, in artistic fields people still prefer human-made art. For computers, it is hard to create art, and even harder to understand and analyze it.

A piece of art in everyday life of almost everyone is music. It is a complex form, where many aspects influence the audience, i.e. melody, rhythm, lyrics, performance, etc. Although we do realize they are interconnected and may affect each other, in this thesis, we will more deeply explore only one of these aspects – song lyrics.

We have a large crowd-sourced dataset of almost half a million English song lyrics. At first, this sounded as a good base for learning a lyrics generator. However, as we explored rhymes and automatic analysis, we realized it is a much more interesting path to pursue. Every attempt at lyrics or poetry generation that we encountered used humans for their final evaluation. This proves, and Greene et al. [2010] agree, that automatic evaluation of poetry is hard.

Unfortunately, there was no sufficient rhyme detector that we could use for our case. In this thesis, we will dive more deeply into the problem and create one ourselves. It will give us the ability to analyze our dataset and draw interesting conclusions about the data.

Additionally, we will create a web-page that demonstrates detector's capabilities and visualizes rhymes in an innovative way. With this tool, we hope to give artists, authors of poems and songs, or even amateurs a new way to explore their texts.

At the end, we will focus on lyrics generation and explore current state-of-the-art pre-trained GPT-2 model and its capabilities in this field.

This work may include some literary or technical terms that the reader is not familiar with. For their definition, please see the "Glossary of literary and technical terms" section at the end of this thesis.

Outline

In Chapter 1, we will make we will explain the literary background such as rhyme, its types, and other literary devices. We will also describe approaches and review existing tools for rhyme detection, visualization, and lyrics generation.

Chapter 2 introduces data that we will be working with, their structure and statistics, and the steps we took to pre-process them.

The most complex part of this thesis is explained in Chapter 3, which specifies the details of how we perform rhyme detection in song lyrics.

Chapter 4 evaluates our rhyme detector and shows the statistics when we run it on our dataset.

How the output of our detector is brought to life by visualization is illustrated in Chapter 5.

In Chapter 6, we describe and review the results of lyrics generation experiment.

Lastly, the results are summed up in Conclusion (Chapter 6), including suggestions for future work.

List of changes

1. Related work

This chapter gives a basic overview of all relevant tools and background information researched during work on this thesis. First, it gives literary background necessary to make the reader familiar with rhyme and its different types, to know what to look for before we start detecting them. Second, it describes existing tools for rhyme detection and visualization, and the different approaches they took. Last, it explains current state-of-art tools for lyrics generation.

1.1 Rhyme types and literary devices

In English, there are many different definitions for what a rhyme is. It is described as "a word that has the same last sound as another word" by Cambridge Dictionary (Walter [2008]) or a "literary device, featured particularly in poetry, in which identical or similar concluding syllables in different words are repeated" by LiteraryDevices Editors [2020]. The definition of what a good rhyme is even changes for different languages and time periods (Zhirmunsky and Hoffmann [2013]). For example, full identity in sound is highly valued in French (*rime riche*), but less valued in English (perfect rhyme requires leading consonant sounds to differ). Some authors refrain from giving an exact definition and instead leave it to reader's intuition (Plecháč [2018]). We will define rhyme through its different types, which will be helpful for detection in Chapter 3.

1.1.1 Basic rhyme types

Perfect rhyme (also true rhyme, or sometimes just "rhyme") is the most common and superior type of rhyme. It requires last stressed vowel and all following sounds to be identical. Some authors (e.g. Bain [1867], van der Schelde [2020], Bergman [2017]) additionally require immediately preceding sounds to differ. The consequence of this condition is the exclusion of identity from perfect rhymes. However, in this thesis, we will use the definition without the additional condition.

Perfect rhyme can be further distinguished depending on how many syllables are involved:

- **Masculine** (also single, monosyllabic) – "the commonest kind of rhyme, between single stressed syllables at the ends of verse" (Baldick [2008]). Examples:

  fly /flaɪ/ – sky /skaɪ/

  before /bi.fɔːr/ – explore /ɪks.plɔːr/ [1] [2]

- **Feminine** (also double) – "a rhyme on two syllables, the first stressed and the second unstressed" (Baldick [2008]). Examples:

[1] For the examples, we are using IPA transcriptions because it is more comfortable for human readers. See Appendix A.1 for pronunciation tables.

[2] Stressed syllables are underlined. Syllables are separated with a dot.

bitten /bɪ.tən/ – written /rɪ.tən/

lazy /leɪ.zi/ – crazy /kreɪ.zi/

- **Dactylic** (also triple) – "a rhyme on three syllables, the first stressed and the others unstressed"(Baldick [2008]). Examples:

  amorous /æ.mər.əs/ – glamorous /glæ.mər.əs/

  vanity /væ.nɪ.ti/ – humanity /hjuː-mæ.nɪ.ti)/

Identical rhyme (also rime riche) is "a kind of rhyme in which the rhyming elements include matching consonants before the stressed vowel sounds." This includes "rhyming of two words with the same sound and sometimes the same spelling but different meanings e.g.:

seen /siːn/ – scene /siːn/

The term also covers word-endings where the consonant preceding the stressed vowel sound is the same:

compare /kəm.pɛr/ – despair /dɪs.pɛr/." (Baldick [2008])

It is generally considered not as good as perfect rhyme because it is too predictable for the listener.[3] However, all rhyme detection tools as well as gold data that we will be using (annotated by professionals) include identity in perfect rhymes. To make the comparison and evaluation with our tool easier, we will do so as well.

Imperfect rhyme (also slant or half rhyme) rhymes "the stressed syllable of one word with the unstressed syllable of another word" (Bergman [2017]). Examples:

cabbage /kæ.bɪdʒ/ – ridge /rɪdʒ/

painting /peɪn.tɪŋ/ – ring /rɪŋ/

In other sources, definitions differ – for example LiteraryDevices Editors [2020] calls this effect "feminine rhyme". On the other hand, Baldick [2008] and The Editors of Encyclopaedia Britannica [2014] use the term "imperfect rhyme" for end-line consonance (see definition below) and van der Schelde [2020] uses it for end-line assonance (see definition below). For the purpose of this thesis, we would like to keep rhyme types disjoint. Therefore we will require the sounds in the imperfect rhyme to be identical, except for the stress. This will differentiate it from *forced rhyme* (see below).

Unaccented rhyme (also weakened rhyme) "occurs when the relevant syllable of the rhyming word is unstressed" (The Editors of Encyclopaedia Britannica [2014]). Examples:

hammer /hæ.mər/ – carpenter /kɑːr.pən.tər/

The difference opposed to imperfect rhyme is that here rhyming parts of both words are unstressed. However, for simplicity, in the scope of this thesis we will include this category under *imperfect rhymes*.

[3] https://literaryterms.net/rhyme/

Forced rhyme (also near rhyme) "includes words with a close but imperfect match in sound in the final syllables" Bergman [2017]. Examples:

green /gri:n/ – fiend /fi:nd/

hide /haɪd/ – mind /maɪnd/

This includes the case when spelling is changed in order to make the rhyme work, e.g.:

truth /truθ/ – endu'th /en.duθ/ (a contraction of "endureth")

It can also refer to using unnatural word order to get the rhyming word at the end of the line (Bergman [2017]) but we will not make use of this interpretation in this thesis.

1.1.2 Other literary devices

This is a short overview of other literary devices that closely correlate with forced rhyme and may, according so some sources, be considered a rhyme. We will conservatively exclude these from our classification and focus solely on rhymes occurring at the end of verse.

Assonance is "repetition of stressed vowel sounds within words with different end consonants" (The Editors of Encyclopaedia Britannica [2014]). Examples:

quite /kwaɪt/ – like /laɪk/

free /fri:/ – breeze /bri:z/

When used at the end of verse with ending consonants having a similar sound, it is equal to forced rhyme. However, the term itself defines a literary device applicable anywhere in the poem, even in the middle of the verse. Some sources classify it as rhyme, giving it various names (van der Schelde [2020], Bergman [2017], and others).

Consonance is "the recurrence or repetition of identical or similar consonants" (The Editors of Encyclopaedia Britannica [2014]). Examples:

country /kən.tri/ – contra /kɑ:n.trə/

hickory dickory dock /hɪ.kə.ri dɪ.kə.ri dɑ:k/

Similarly as assonance, it applies to repetition of consonants in any part of the verse. When seen at the end of verse, it can be considered a rhyme and again, various terms are used – perhaps the most common is "pararhyme" (The Editors of Encyclopaedia Britannica [2014], Baldick [2008]).

The last two terms may seem as more of a tool for poets than songwriters. Surprisingly, they have found their way into song lyrics and have become a standard in genres like hip hop according to van der Schelde [2020]. From the creative point of view, it is not less sophisticated rather it enriches rhyme as we know it (Brogan [2016]).

Other rhyme types exist e.g. eye rhyme where "the spellings of the rhyming elements match, but the sounds do not, e.g. love /ləv/ – prove /pru:v/" (Baldick [2008]). We do not consider them relevant for song lyrics or the purpose of this thesis.

1.2 Rhyme detection tools

We have defined what to look for, and now we will focus on how to do it. According to Plecháč [2017], there are 4 methods for rhyme detection. We will describe each one and evaluate existing tools. For our use case, it is important that we can use the tool for automatic evaluation – there must be a way to run it with code whether it would be a web service, an executable script, or as a library/module.

Another requirement is to be able to run on a block of text and generate rhyme scheme as a result (for rhyme scheme definition see Section 3.6.2). Lastly, it has to be free and preferably open-source.

Notably, English orthography is highly nonphonemic, thus a pronunciation dictionary is needed for converting graphemes to phonemes. Additionally word stress in English is irregular, so the dictionary must contain markers of stress to detect the rhyme types as we defined them.

1.2.1 Naive rule-based approach

The simplest approach is to compare for identity of phonemes at the end of lines. Noticeably, this only detects perfect rhymes. Nevertheless the result will seem decent because it has almost 100% precision, i.e. notices all "obvious" rhymes. Another downside of this approach is its limitation to the size of the dictionary.

There are many rhyming dictionaries (of various quality and size) to choose from but the vast majority uses or enhances CMU dictionary (CMUdict).[4] For more details about CMUdict see Section 3.3.2.

Pronouncing and CMU dictionary

Pronouncing[5] is a Python library providing an interface for CMU Pronouncing Dictionary. One possibility is to install CMUdict directly, search for pronunciations for both words and compare then. *Pronouncing* searches the dictionary automatically for a given word and returns a list of rhyming words. However, the list is truncated and probably better suited for writer's inspiration only.

1.2.2 Advanced rule-based approach

Enhancing the naive approach with various similarity measures or other features allows us to find more subtle rhymes like *imperfect* or *forced*.

Rhyme Genie

Although this tool is not free, it is the most popular, so it is worth mentioning. Rhyme Genie[6] is a desktop application for Windows and MacOS that suggests 30 different rhyme types for a given word. Additionally, it includes sayings, clichés, idioms, and a very unique feature – adjustable rhyme similarity. However, its use case is the reverse of what we are looking for – rhymes are not found, only suggested.

[4] http://www.speech.cs.cmu.edu/cgi-bin/cmudict
[5] https://pypi.org/project/pronouncing/
[6] https://www.rhymegenie.com/rhyme-genie.html

SPARSAR

SPARSAR (Delmonte and Prati [2014]) is also a very interesting tool for poetry analysis and expressive Text-to-speech conversion. It is originally designed for a thorough examination of a very strictly structured Shakespeare's sonnets. To achieve this, it has to run analyses on many levels – and these results can be used to analyze any poem. It looks at the poem on three levels: phonetic (pronunciation, consonant and vowel tongue position, assonance, etc.), poetic (metrical structure, rhyme schemes, acoustic length, etc.), and semantic (sentiment, metaphorically linked words, anaphora, etc.).

User can choose between a window application with graphs and diagrams or a headless mode with .xml output files. Its main disadvantage for our use case is that it is written in Prolog and therefore is very strict on the input format and runs only under a specific older version of Ubuntu.

Datamuse

Datamuse API[7] combines the advantages of a rhyming dictionary and semantic analysis. It uses CMUdict for phonetic transcription, analyzes CommonCrawl[8] web data repository for forced rhymes, Google Books Ngrams (Weiss [2015]) for building language model, and WordNet 3.0 (Pearson et al. [2005]) for semantic relations. Users can send complex queries, e.g. "words that rhyme with *grape* that are related to *breakfast*". Similarly to Rhyme Genie, it focuses more on rhyme suggestion rather than rhyme detection.

1.2.3 Similarity (distance) based approach

Generally, substituting arbitrary consonant in perfect rhyme does not necessary create forced rhyme – corresponding phonemes must have similar sounds. Objective criteria to measure this similarity can be phoneme's features e.g. plosive, nasal, fricative, voiced, etc. Rhyme detectors can use these features and other specific sound rules to calculate sound distance between two words. We found no specific tool focusing on this approach, but some listed tools like Rhyme Genie incorporated it into their algorithm.

However, not all phonetic features contribute to similar sound the same. Furthermore, speakers from different areas perceive sound differently and may have distinct boundaries for what is similar and what is not. Using AI, we can tailor the similarity based on the data.

1.2.4 Machine learning

The vast majority of recent tools used AI to solve this problem. Unsupervised machine-learning methods for rhyme detection require a large amount of data containing rhymes. The main advantage is that this data does not need to be annotated – rhyming can be learned directly from the data thanks to its large quantity. Consequentially, by learning from the data, detectors can adapt to different genres, rhyme types, or even languages.

[7] https://www.datamuse.com/api/
[8] https://commoncrawl.org/

When words commonly occur next to each other in text, it is referred to as *collocation*. For rhyme detection, we will define a more useful term – *vertical collocation* – as the co-occurrence of words, syllables, or phonemes at the end of lines in close proximity. Notably, commonly vertically co-occurring words have a higher probability of being rhymes. One possibility how to take advantage of this fact is using the EM algorithm (described further in Section 3.5.2).

Reddy & Knight's EM algorithm

Reddy and Knight [2011] proposed a language-independent model for finding rhyme schemes in poetry. They created an unsupervised model based on EM algorithm that assigns the most probable rhyme scheme for each sequence of line-final words. It achieved good results when tested on annotated English and French corpus with poetry from 15$^{\text{th}}$ to 20$^{\text{th}}$ century. However its big pitfall lies in the fact that it is biased towards the rhyme schemes from golden data. It has a predefined set of all rhyme schemes found in tested data and those are the only ones it chooses from. For illustration, according to Plecháč [2017], in a 14-line stanza it can choose from 90 schemes which is only 0.00005% of all possible options. In 29% of cases from French corpus it has only one choice.

RhymeTagger

As an improvement on top of Reddy and Knight [2011], Plecháč [2018] came up with an open-source collocation-driven tool named RhymeTagger.

It uses the same dataset as the previous approach with addition of a larger Czech poetry corpus.[9] Each line-final word is transcribed into phonetic transcription and split into two types of components – syllable peak for each syllable and consonant cluster in between. In the *expectation* step, probabilities for each component pair are calculated based on their co-occurrence in line-final words, e.g. the conditional probability of the rhyme based on peak component pair əʊːəʊ will be very high but for consonant component pair kːr quite low. These statistics for component pairs are then used in the *maximization* step to calculate the probability for line-final word pair as a combined probability of all their components (paired according to their position from the end). If the probability of two words is above a given threshold they are considered a rhyme. After all such pairs are classified, probabilities are iteratively recalculated in the EM cycle.

For words that were not successfully classified with this method, there is a fallback. The author observed that some words are now pronounced differently than during the Shakespearean era, therefore using current pronunciation dictionaries may ruin the original rhyme, e.g. original near /nɛːr/ – there /ðɛːr/ vs. contemporary near /nɪːr/ – there /ðɛːr/. The original pronunciation can be therefore inferred from words with similar orthography. Plecháč calculated rhyme probability given final character trigrams, which helped achieve higher recall. Although other methods may have better precision, collocation-driven approach wins in recall as seen in Figure 1.1.

For evaluation, they used precision, recall and F-score calculated as follows:

[9] https://github.com/versotym/corpusCzechVerse

$$PRECISION = \frac{true\ positives}{true\ positives + false\ positives}$$

$$RECALL = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$F\text{-}SCORE = \frac{2 * PRECISION * RECALL}{PRECISION + RECALL}$$

For an intuitive view, the reader can imagine precision as how many of algorithm-detected rhymes were actually rhymes, and recall as how many rhymes were discovered. F-score describes the trade-off between the two.

Souhlasím, že je zajímavé uvést zde Figure 1.1 a že k tomu je vhodné zadefinovat precision/recall/f-score. Jinak by se to ale hodilo spíš do Kapitoly 4, kde definujete vlastní scores (Last-index, ARI). Aspoň byste zde měla na odkázat na Kapitolu 4, že tam budou evaluační metriky lépe vysvětleny. A tam byste zase měla aspoň zmínit Precision/Recall/F-score a diskutovat, jak se vlastně liší třeba od Last-index score a co je k čemu vhodnější. Tím posledním si sám nejsem jist, ale přijde mi férové aspoň přiznat, že to jsou alternativní metriky pro měření kvality detekce rýmů.

Deep-speare

As a part of their sonnet quatrain generating model, Lau et al. [2018] have implemented a Rhyme component that identifies and generates rhymes. It is a unidirectional forward LSTM (Hochreiter and Schmidhuber [1997]) that learns to separate rhyming word pairs from non-rhyming. They generate input by pairing one line-final word with the other three from the same quatrain. Since the rhyme scheme of a sonnet quatrain is always *abab*, this will result in one rhyming pair and two non-rhyming. Additional non-rhyming pairs are generated with random word sampling. Then the model with margin-based loss learns the margin separating the best pair from all the others. It returns a cosine similarity score that estimates how well do two words rhyme.

To evaluate this model, authors used phoneme matching with CMUdict and the EM model from Reddy and Knight [2011] trained on their own data and they were able to outperform both according to F-score.

1.3 Visualization tools

In the following section, we will describe existing visualization tools for poetry. Software mentioned below focuses on poems, however song lyrics can be considered just a more structurally relaxed version of a regular poem.

Poem Viewer

Quite complex and comprehensive visualization tool is Poem Viewer [Abdul-Rahman et al., 2013]. With no need for complicated installations it is easily available for the writers as a web-based application as shown in Figure 1.2. Unfortunately, at the time of writing this thesis the upload of custom text was not

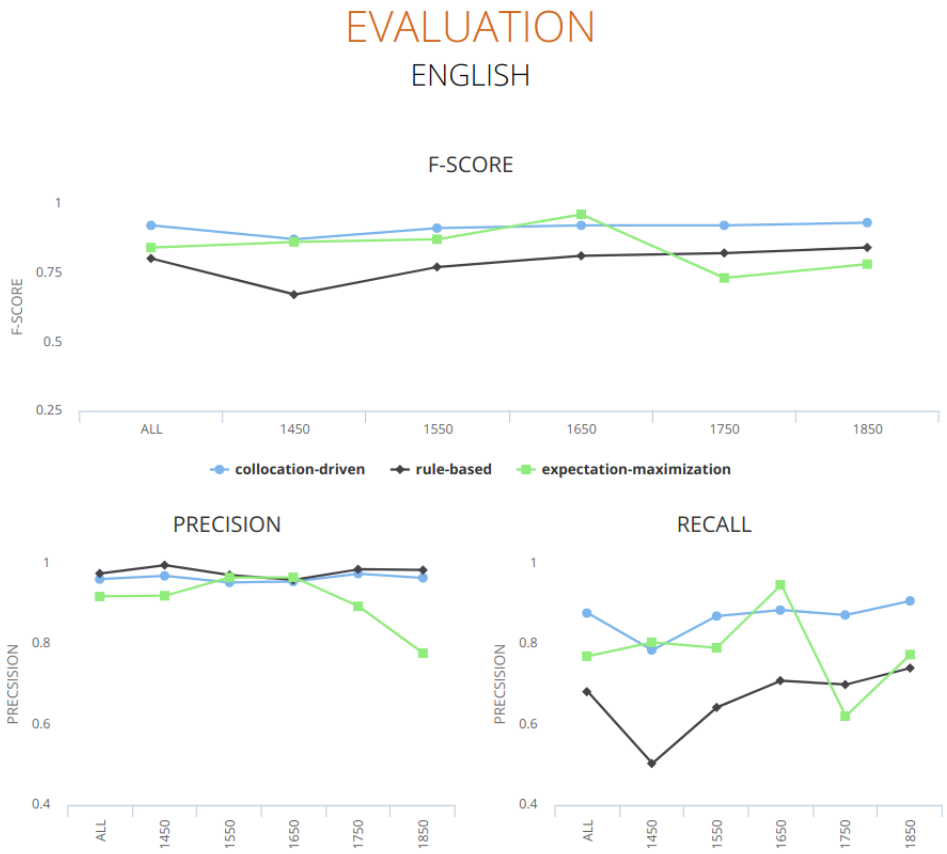

Figure 1.1: Evaluation of RhymeTagger on English corpus in comparison with the EM algorithm and simple rule-based approach. The x axis is the year when the tested poem was written, the y axis are the evaluation scores as described above. Reproduced from Plecháč [2017].

working. Luckily, this is still an ongoing project so this might be just a temporary issue. Nevertheless there are some default poems available to demonstrate this software's capabilities.

Most of the analyzed features (shown in Figure 1.3) focus on the phonetic aspects of the poem. After phonetic transcription to IPA, users can analyze consonant features, vowel length and position, stress, syllables, word classes and sentiment using color codes and markers. A second layout offers six different graphs/animations of tongue positions during each verse. Arcs are used to mark end rhyme, alliteration, assonance, consonance, their particular frequencies and repeating words.

Overall this software, although very elaborate, feels overwhelming and confusing for an inexperienced user. Moreover, it is perhaps better suited for its original use case – a well-structured poem – than less regular song lyrics.

ProseVis

This Java desktop visualization tool by Clement et al. [2013] analyzes text through parts-of-speech, phonemes, stress, tone, and break index. These features are

Figure 1.2: Screenshot from Poem Viewer tool – visualizing Love by Elizabeth Barrett Browning.

extracted using OpenMary Text-to-speech system (Schröder et al. [2006]) and predictive classification. The authors believe their visualization will present the features to user in a more human readable form (ProseVis [2014]).

Poemage and RhymeDesign

Poemage (McCurdy et al. [2015a]) and RhymeDesign (McCurdy et al. [2015b]) are both open-source applications with focus on analysis of sonic devices and sonic topology in poetry. Poemage[10] focuses on complex structures of words connected through sonic or linguistic resemblance across the space of the poem. It is available for MacOS or Windows with a web version currently under development. In MacOS application RhymeDesign – which also provides the backend for Poemage – users can enter their poem and query for one of the default rhyme types or choose a custom rhyme type.

Ambiances

This software is unique in the fact that the analysis is integrated in the process of writing. As described in the paper Meneses and Furuta [2015], writers input the poem, receive a visualization and can control this visualization with body and hand gestures which in turn influence the poem. By such interconnection the authors aim to make Ambiances a part of the writing process and give it a chance to influence the final result. However, the actual software does not seem to be publicly available.

[10] http://www.sci.utah.edu/~nmccurdy/Poemage/

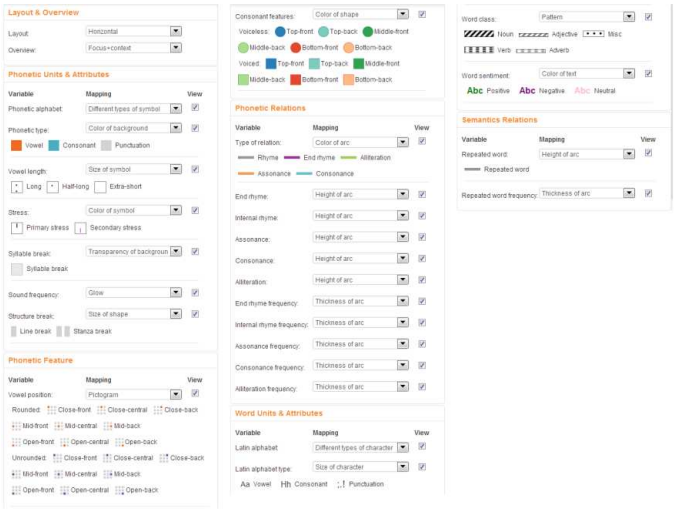

Figure 1.3: Available options and their default mappings in Poem Viewer.

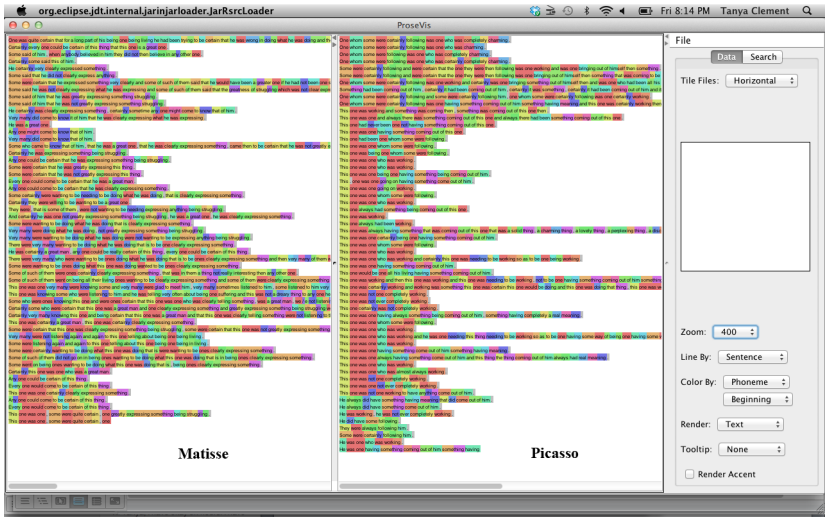

Figure 1.4: Comparison of two poems in ProseVis. Reproduced from ProseVis [2014].

1.4 Generation tools

Generation of text, especially artistic, is a very challenging task for a machine. As we observed the outputs of exiting tools, we found that the most common flaws include lack of creativity, unnatural and frequent change of the subject, and incoherence. Generation of song lyrics is basically a more specific branch of text generation. Similarly to rhyme detection, it can be distinguished into two main types of methods – rule-based and machine learning.

1.4.1 Rule-based generating

Rule-based tools are inherently very complex and the output is often limited to certain structure or topic. They usually require the user to input starting configuration, whether it is genre, topic, time period, amount and type of rhymes, phrases, etc. They tend to be less creative but better at rhyming and following

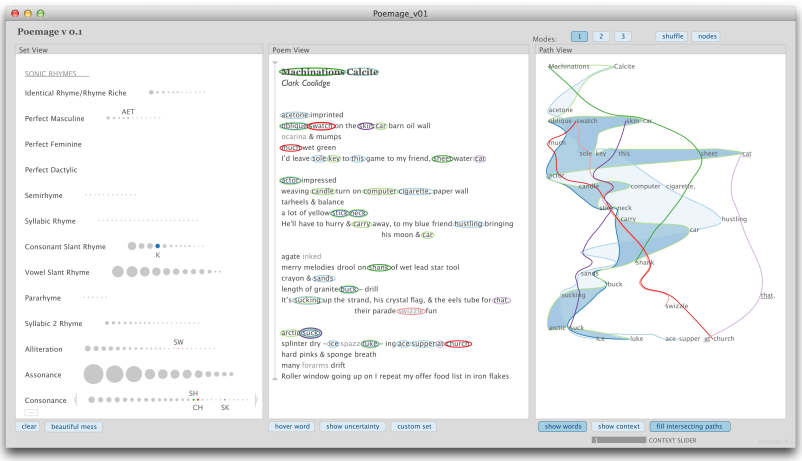

Figure 1.5: An example analysis in Poemage.

the form and structure. To achieve that, they may use rhyming dictionaries (see Section 1.2). Simpler ones just use a large set of pre-written templates, e.g. MasterPiece Generator.[11] In this thesis, we will focus on generation using machine learning.

1.4.2 Generating using machine learning

The popular approach for generating song lyrics is certainly machine learning in its many forms. It can be proved by the large number of lyrics generators created by enthusiasts, e.g. *These Lyrics Do Not Exist*[12], *Freshbots Lyrics Generator*[13], *Random Lyrics Generator*[14], *DeepBeat – Rap Lyrics Generating AI*[15], *BoredHumans Generator*[16], *RapPad Lyrics Generator*[17], and many more. We will further describe Deep-speare (mentioned earlier in Section 1.2.4) and GPT models, which are considered state-of-the-art in text generation.

Deep-speare

Deep-speare (Lau et al. [2018]) is a joint neural network architecture that generates only a specific type of poems with strict form and meter – sonnet quatrains. It consists of three models: language model generates one word at a time, pentameter model samples meter-conforming sentences, rhyme model enforces rhyme, and they are all trained together in multi-task learning setting. They present very good results – generated poems are mostly indistinguishable from human-written ones, apart from expert evaluation, where they report lack of emotion and worse readability.

[11] https://www.song-lyrics-generator.org.uk/
[12] https://theselyricsdonotexist.com/
[13] https://www.freshbots.org/lyrics-generator
[14] http://www.anticulture.net/RandomLyrics.php
[15] https://deepbeat.org/
[16] https://boredhumans.com/lyrics_generator.php
[17] https://www.rappad.co/songs-about/

GPT-2

Generative pre-trained Transformer version 2 (GPT-2) (Radford et al. [2019]) is an unsupervised transformer model capable of various text-processing and generating tasks such as answering questions, translating, summarizing, writing coherent paragraphs, etc. It was created by AI-based research laboratory named OpenAI.

This model was trained on and evaluated against WebText, a dataset consisting of the text contents of 45 million links on sites like Google, Blogspot, GitHub, NYTimes, BBC, eBay, etc. It offers 4 models of different sizes increasing in the number of parameters: 124 million (small), 355 million (medium), 774 million (large), and 1.5 billion (XL) parameter models.

Although this model was only trained for the general task of predicting the next word, given all of the previous words within some text, it can be further fine-tuned for a more specific task to suit user's needs. One example of lyrics generated by fine-tuned GPT-2 is *Keywords To Lyrics*[18]. However, even without fine-tuning, it can quickly adapt to the style of the input and continue in the same manner, as we show later in Chapter 6.

GPT-3

During writing of this thesis, an even larger model GPT-3 (Brown et al. [2020]) with 175 billion parameters was officially introduced. It is considered the largest artificial neural network in May 2020. It was trained on five different corpora: Common Crawl, WebText2, Books1, Books2 and Wikipedia. The architecture is the same as in GPT-2, only number of layers and other parameters increased. It is capable of writing articles indistinguishable from human-written ones (Romero [2021]), even produce functional JavaScript code for natural-language formulated task.

Realizing the power of this tool, the authors did not want to make it available to broad public, fearing it might be misused with bad intentions. Instead they created a form to sign up for access, reviewed individual requests, granting access only to a small portion of them.

Originally, this thesis intended to focus more on lyrics generation. However, as our research of works in this field shows, there is an abundance of tools for that purpose. Users can just choose a tool that fits their needs. Creating one tool that would be better or more versatile would either require more computing power or literary knowledge, and is above the scope of a master thesis. Therefore, we shifted our main focus to automatic rhyme detection and evaluation, which seems to be far less explored and more interesting topic.

[18] https://lyrics.mathigatti.com/

2. Data

A crucial part of every analysis are the data. To be able to conduct an analysis with results that can reasonably represent the domain, we need to have enough of them – the more the better.

Our dataset consist of 658,460 song lyrics scraped from the crowd-sourced website Genius.[1] Sadly, the original author of the dataset is unknown, it has been passed on to us by a colleague as a potentially interesting source for research. However, all song lyrics are publicly available on the Genius website and can be linked with the corresponding item of the dataset via the *url* attribute.

We apologize for any strong language that may be used in song lyrics or their excerpts in this thesis. Due to its various forms and the size of the dataset, it would be extremely difficult to remove them, and because they occur in pop culture naturally, we chose to portray them faithfully.

2.1 Preprocessing

In most areas it is very hard to find a dataset of good quality and large quantity. Usually at least one of the two suffers. It is not any different with our data – although the dataset is large, the contents were created by ordinary people and intended for human readers so they are not well suited for automated processing. It is necessary to look closely at the data, remove faulty or redundant items, and clean the rest with preprocessing.

To assess what are the problems in the data and how to address them, we created a very small dataset of only about 10 songs which we cleaned manually. To select these songs, we looked at about 100 random songs and chose the ones that contained the most common faults. We also tried to contain a broad spectrum of errors by focusing on the diversity in the selected dataset. We then iteratively implemented an automated solution for each type of data corruption, comparing the automatically and manually cleaned data, until they matched. We also extracted statistical information that further showed the weak points that needed addressing.

We received the dataset in JSON format, with each song as a separate item, each containing the following features:

- *title* – the name of the song
- *lyrics* – the text of the song's lyrics
- *album* – song's album (or null)
- *genre* – one of the following: rap, pop, rock, r-b (rhythm & blues), country
- *artist* – song's performer
- *url* – the URL of the lyrics page on Genius website
- *year* – the year the song was produced

[1] https://genius.com/

- *is_music* – boolean flag distinguishing song lyrics from other texts
- other song details: *producer, featured artist, recording location, charts, writer, samples, sampled in, has featured video, has featured annotation*
- other website specific information: *rg artist id, rg type, rg tag id, rg song id, rg album id, rg created, has verified callout*

The features from the last two points were *null* for all or most of the items. That, and the fact that they do not give us much more information that would contribute to the lyrics analysis, made them useless and so we decided not to keep them. We removed all songs for which the attribute *is_music* was False, indicating it was not a song (often a poem or prose) or they did not contain lyrics – 34,259 songs in total. We further removed one song with invalid (incomplete) JSON. After comparing the lyrics to each other, we found and removed 32,551 duplicates.

Upon further inspection, we found out that our dataset also contains lyrics in different languages. We used the neural network model for language identification by Google (CLD3)[2] for the classification. It showed that our dataset contains exactly 100 different languages, most of them represented only marginally. Since other languages did not have enough data to support a good analysis and implementing them would be above the scope of this thesis, we kept only English lyrics. We further removed 832 items with language detection errors. All of them were under 10 lines long so they would not be a valuable addition anyway. At this point our dataset contained 438,037 items.

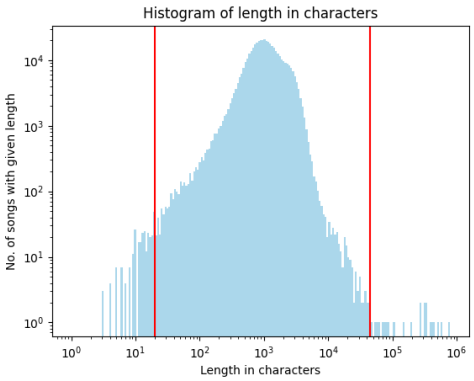

Figure 2.1: Histogram of number of characters in songs of our dataset.

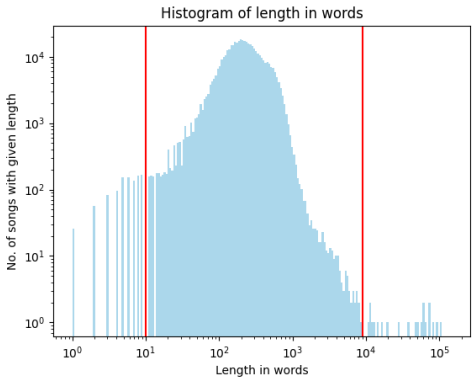

Figure 2.2: Histogram of number of words in songs of our dataset.

To learn more about our data, we created histograms with song's length in characters, words, and lines (see Figures 2.1, 2.2, 2.3). Knowing the common issues of extreme values, we manually examined a few of the shortest and a few of the longest items. Confirming our expectations, we found out that they were not valid songs either. The long ones were usually book excerpts or rap improvisation battles, while the short ones were often links to advertisements or motivational quotes. We removed 14 long and 1,838 short lyrics. Although it may not seem as much, it could have strong negative influence mainly during the

[2] https://github.com/google/cld3

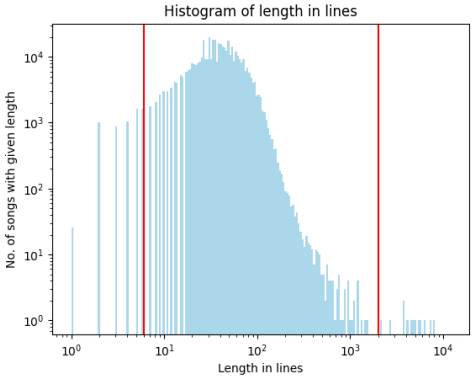

Figure 2.3: Histogram of number of lines in songs of our dataset.

generation phase. In Figures 2.1, 2.2, and 2.3, red lines mark the borders for removal – only the items in between them were kept.

2.2 Structure of the data

This section gives statistical information about the dataset after preprocessing. Table 2.1 sums up basic statistics about the data overall and for each genre specifically. The pie chart in Figure 2.4 shows the portions of the data belonging to each genre. All the attributes are listed in Table 2.2. For some songs, not all attributes are available. Number of items with non-empty values is given as well.

Genre	Songs	Avg. lines per song	Avg. words per line
Pop	293,679	36.73	5.50
Rap	99,189	64.32	6.88
Rock	34,372	38.73	5.30
R&B	5,126	52.82	5.51
Country	3,819	38.43	5.85
Total	436,185	43.36	5.96

Table 2.1: Basic statistics about the dataset.

2.3 Annotated subset

From the dataset described above, we separated a subset of 50 songs, 10 song for each genre, and annotated them with rhyme schemes ourselves. We then separated them into a development and test set, so that both groups would have an approximately equal number of non-empty lines per genre (plus or minus 2 lines). We reserved the development set for iterative testing and improvement of our algorithm. The test set was kept unseen and used only for the final evaluation, as described in Chapter 4.

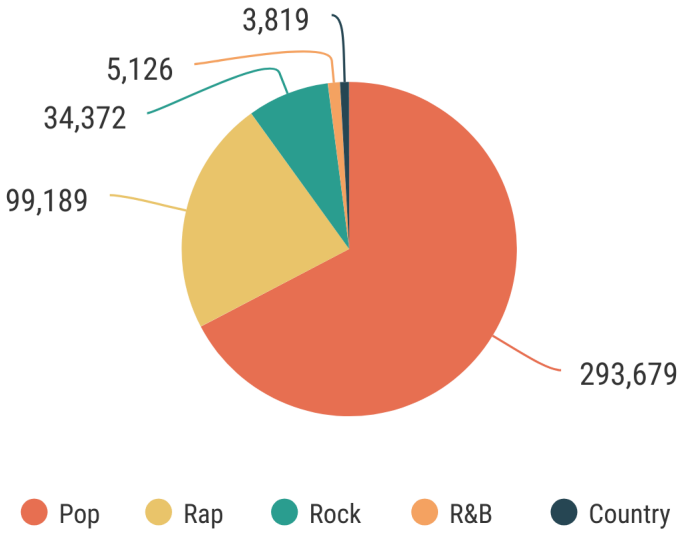

Figure 2.4: Distribution of genres in the dataset.

Attribute	Non-empty values
lyrics	436,185
title	436,179
album	112,060
genre	436,185
artist	436,184
url	436,185
year	96,491
lang	436,185
id	436,185
word_count	436,185

Table 2.2: Attributes and their counts of non-empty values.

2.4 Chicago Rhyming Poetry Corpus

As an additional dataset for evaluation, we included one from an outside source – the Chicago Rhyming Poetry Corpus[3], annotated with rhyme schemes by professionals. It contains 1,321 poems from 32 authors written in 14$^{\text{th}}$ to 19$^{\text{th}}$ century. It has a total of 93,045 lines with an average of 70.44 lines per song. It is the same dataset that was used for training and evaluation of Rhyme Tagger.

[3] https://github.com/sravanareddy/rhymedata

3. Rhyme detection

In this Chapter, we will first consider using available tools for rhyme detection. After not finding the right tool we will reconsider to make the detection ourselves.

Detecting rhymes may seem like a simple task at first, but looking into details one discovers many problems that need to be addressed. As we have seen in Chapter 1, it is not a well-defined task so we need to set the requirements.

Then we progress through individual steps needed to find the rhymes and design a rating that would evaluate song's rhymes:

1. phonetic transcription with additional data preprocessing
2. syllabification and extraction of phonemes after last stressed syllable
3. comparison of two lines and calculating their rhyme rating
4. finding all rhymes and assigning a scheme
5. calculating song rating

3.1 Using available tools for detection

The simplest approach would be to use one of the tools described in section 1.2. We want a detector that is free, strong (detects more than perfect rhymes), and offers headless mode (we could run it automatically from code). Ruling out unsuitable tools, we are left with Rhyme Tagger and SPARSAR. Rhyme Tagger is easy-to-use but only outputs rhyme scheme. To be able to automatically evaluate the rhyming quality of a song, we would need more information like stress or rhyme type.

SPARSAR

SPARSAR, on the other hand, has a very rich and detailed output. Although it is lacking documentation, the *xml* output format is quite descriptive to understand what most of the values represent. It seemed promising so we attempted to pursue this path.

To bridge the outdated system requirements, we contacted the authors for a newer build (for Ubuntu 19.1). They were very helpful and soon we were able to run it on our computer. Nevertheless, we encountered several issues, mostly stemming from the fact that SPARSAR was written in Prolog. Firstly, the xml output was difficult to parse automatically, as the values were written in Prolog syntax, using the same delimiter for different levels of separation. Secondly, SPARSAR parses the text in sentences but lyrics are usually written without punctuation. We added punctuation using *punctuator2* (Tilk and Alumäe [2016]) which also added space for error.

Lastly, when SPARSAR encountered an unknown word, it failed for the entire song. Since our data were crowd-sourced, it contained many unusual words, so in the beginning it failed on 80% of our data. We iteratively worked with the authors for months on fixing bugs and adding words (mainly contractions, such

as I'mma, y'all, yo', 'em, etc.) into their dictionary. In the end, we were able to successfully run it on 95% of our data.

However, we were still not very satisfied with the result. As you can see in the example in Table 3.1, it failed to detect the perfect rhyme between 2nd and 4th line, and instead marked a rhyme on line 1 and 3, where there was none. The example shows the first four lines of the song, but the scheme letter *h* does repeat only once more on line 34. Such errors were not sparse and led us to believe that the encoded inclination of this tool to look for sonnet-shaped schemes caused it to make errors in the diverse schemes of song lyrics. It may be a great tool for sonnets, but we have found it insufficient for our purpose, so we proceeded to create our own rhyme detector.

Scheme		Line	Last word's pronunciation
correct	SPARSAR		as assigned by SPARSAR
A	a	Pulled out from the station	s-t-ey-sh-ah-n
B	h	fifteen after two	t-uw
C	a	300 miles away from Vegas	v-ey-g-ah-s
B	b	We had nothin better to do	d-uw

Table 3.1: Example of incorrect scheme assignment by SPARSAR. Excerpt if the beginning of the song *Good Life*. The correct scheme is marked in capital letters, the scheme assigned by SPARSAR is marked in lowercase letters.

3.2 Defining the requirements

Before we dive into algorithm selection and implementation details of our detector, let's define what exactly do we want our detector to do. Additionally, we also establish terms for common cases in Table 3.2, to keep our further explanations short and clear.

component	a vowel or a consonant cluster in a syllable
rhyme candidates	two lines that are being compared for rhyme
rhyming fellows	two lines that rhyme together
rhyming part	the exact components that participate in rhyme (i.e. are equal or similar in sound)
rhyme group	a group of lines that all rhyme together (i.e. have identical scheme letter)
rhyme rating	rating of the quality of one rhyme between two lines
song rating	rating of the rhyming quality of the entire song

Table 3.2: Establishing terms.

Input Although we realize sound can affect rhymes, for the purpose of this thesis we will focus solely on text. As List [2020] points out, most works focus on rhyme schemes in well structured poetry instead of common rhymes in text.

We do not have standard rhyme scheme patterns nor fixed syllable counts – song authors use rhymes arbitrarily. Although in some genres, mainly rap, rhymes in the middle of the line (internal rhymes) can occur, as we decided in Chapter 1 we will only focus on end rhymes in this thesis. For our input, we expect song lyrics in English, formatted in lines with rhyme always at the end of line. Optional empty lines between stanzas or chorus will be preserved but skipped.

Output The main element of our output is rhyme scheme. As a single element, it gives the best overview of the song and more importantly, it allows us to compare our detector with others or with the gold data. Additionally, we want more information to assess rhyme quality: rhyme rating for each individual rhyme, song rating, rhyme type, pronunciation of the rhyming part, optional modification of stress, etc.

3.3 Pronunciation

3.3.1 Phonetic alphabets overview

Unlike many other languages, English does not have a straightforward pronunciation rules. Therefore to be able to assess rhymes, we need to transcribe our text into a phonetic alphabet first. There are two commonly used alphabets to choose from – IPA and ARPAbet. The original International Phonetic Alphabet (IPA) used since 1888 uses one UNICODE character to encode each phoneme and it is commonly used for example in dictionaries. Since it uses non-ASCII characters, ARPAbet was developed as an equivalent for computers. It has two versions: 1-character that uses upper-case and lower-case letters and (the more common) 2-character version where each phoneme is represented by one or more upper-case ASCII characters (Lea [1980])(see Table 3.3 for comparison). We will be using the 2-character ARPAbet because it is used by the CMUdict.

Example word	IPA	1-character ARPAbet	2-character ARPAbet
st**o**ry	ɔ	c	AO
bu**tt**er	ɾ	F	DX

Table 3.3: Short comparison of different pronunciation alphabets.

3.3.2 CMUdict

Carnegie Mellon University Pronouncing Dictionary (CMUdict) is an open-source pronunciation dictionary.[1] Currently it contains 134,373 words (including their inflections) and their pronunciations in 2-character ARPAbet. For each word, there is one or several possible pronunciations in North American English including stress markers for primary, secondary or no stress. For the implementation, we used its Python wrapper package *cmudict*.[2] To use this we had to strip the input of punctuation and convert it to lower case.

[1] http://www.speech.cs.cmu.edu/cgi-bin/cmudict

[2] https://pypi.org/project/cmudict/

CMUdict is a large dictionary and it includes also slang words so it should cover most of our input. To test this, we looked at all last words on each line of our data (since those are the important ones for rhyme analysis) and we found out that 5.52% of them are not included in CMU dictionary. We manually inspected a small portion of them and found out that they can be mostly split into the following 5 categories:

- uncommon words, e.g. superglue, redundantly
- misspelled words, e.g. decsion, girlfren
- numbers
- foreign words, e.g. revoluccion, ecolli
- interjections and onomatopoeia, e.g. shoooshooo, woahwoah

3.3.3 Dealing with out-of-dictionary words

In an attempt to decrease the amount of out-of-dictionary words, we replaced the closing single quotation mark "’" (U+2019) with the typewriter apostrophe "'" (U+0027) since only the second variant of apostrophe is accepted by CMUdict. However, this only decreased the percentage of unrecognized words to 5.47%.

Clearly, we needed a way to estimate the pronunciation for the rest of the words, so we used grapheme-to-phoneme library *g2p* by Park and Kim [2019]. It predicts the pronunciation for out-of-dictionary words using deep learning seq2seq model by TensorFlow (Abadi et al. [2015]).

Even having the pronunciation for every word will not ensure we find every rhyme intended. Song artists may take their liberty in modifying or skewing the pronunciation to make the rhyme work. Sometimes they can also sing two syllables in one beat or use an unusual pronunciation from different culture to convey a message. As we established in the beginning, we will focus only on information retrievable from the text and ignore these possible deviations in pronunciation.

3.4 Syllabification

3.4.1 Alignment

When comparing lines for rhymes, we have to establish a system of alignment so that we analyze only relevant pairs of phonemes. Initially, we created a simple rhyme detector that just traversed both verses backwards phoneme by phoneme[3] and compared them. However, rhyming words do not have to have an equal number of phonemes. For example words in the Table 3.4 have a 2-syllable rhyme. If we compared all phonemes one by one they get misaligned on consonant clusters S-T-R and P-L and we will miss the penultimate syllable rhyme. For forced rhymes this inherently becomes a very frequent issue.

[3]For simplification, we use the term phoneme for one symbol in ARPAbet.

Word	ARPAbet transcription
constrain	K AH N – S T R EY N
complain	K AH M – P L EY N

Table 3.4: Example of misalignment when aligning by phonemes.

We need to make sure that we are comparing corresponding parts of verses otherwise we will miss the rhyme. A better approach would be to compare corresponding syllables because they naturally create an alignment in the rhythm of the song. Each syllable can be further split into 3 groups ("CVC") – leading consonant cluster (*onset*), vowel (*nucleus*), and trailing consonant cluster (*coda*). Consonant clusters can sometimes be empty. For syllabification we used Python library *syllabify*,[4] which requires input in ARPAbet (we use the output of CMUdict) and conveniently returns syllables in CVC triplets as described above.

3.4.2 Extracting relevant components

Since we established that rhymes are located at the end of each line, there is no need to analyze the entire verse. How far should we look? The first choice would be to look at the last word only. However rhymes can extend over more words as we see in example in Figure 3.1.

I was the man the Duke **spoke to**;
I helped the Duchess to cast off his **yoke, too**;

Figure 3.1: An example of two-word rhyme from *The Flight of the Duchess* by Robert Browning.

When we look at rhyme types, they do not go further then the first stressed syllable (looking at the line backwards). Notably, even if the rhyme does extend further we can ignore the rest because it cannot increase the rating. For example, if there are more rhyming syllables preceding the perfect rhyme, they cannot make the score better. Similarly, if the rhyme is not perfect, syllables preceding the final stress would already be considered an internal rhyme – which is also used (mainly in rap lyrics) but less valued than the classical end rhyme and as stated in Section 3.2, not a target of analysis in this thesis. We will therefore limit our window to the minimal number of syllables needed to include the stressed syllable in both rhyme fellows. If one rhyme fellow needs less syllables than the other (i.e. it is an imperfect rhyme), we will move stress to the right to match the shorter syllable span (and later decrease the rhyme rating by a *stress penalty, cf. Section 3.5.3*).

[4] https://github.com/kylebgorman/syllabify

3.5 Rhyme analysis

3.5.1 Finding similar phonemes

Finding perfect rhymes is easy – phonemes after the last stress need to agree exactly. Imperfect rhymes can be found similarly, using the trick of moving the stress described in Section 3.4.2. To set ourselves apart from simple rhyme detectors, we need to detect more than just perfect and imperfect rhymes – we need to find forced rhymes as well. To determine forced rhymes we have to assess the match in sound of individual phonemes.

We will refer to an occurrence of a given component after the last stress simply as *occurence*. Similarly, *co-occurrence* of a component will refer to the occurrence of the same component on the same position in the rhyme fellow (as detected by the algorithm described in Section 3.5.2).

For assessing the match in sound, we took inspiration from Plecháč [2018]. He used the fact that rhymes tend to reoccur and, having enough data, commonly co-occurring pairs are most probably rhymes even without the knowledge of their pronunciation. He calculated the probabilities of phoneme co-occurrences in line-end words from their frequencies in data, used this co-occurrence matrix to predict rhymes, and then iteratively recalculated the matrix using the EM algorithm.

However, our system of alignment is different so our matrix components will look differently. The differences are shown in Table 3.5. The first difference is that Plecháč only uses vowels and consonant clusters without acknowledging the syllable separation. We do separate the consonant cluster into two on the border of syllables (e.g. "rp" in carpenter is in one cluster for Plecháč, but in two separate clusters for us). The second difference is that Plecháč recognized the position of the component and paired only the components on the same position (i.e. both "ə" in carpenter are in separate groups because they occur on different positions in the word). We do not believe that position has an effect on phoneme similarity and therefore we add it together for all positions, e.g. if once two phonemes co-occur on 3rd position and once on the 5th we will say that this pair of phonemes has 2 co-occurrences.

Plecháč	k	ɑː	r.p		ə	n.t		ə r
Us	k	ɑː	r	.p	ə	n	.t	ə r

Table 3.5: Comparison of our components for word *carpenter* with those of Plecháč [2018]. Different color signifies different component group.

Another difference is that Plecháč only looks at the last word and its 1 or 2 last syllables while we look at everything after the last stress up to 4 syllables.

Knowing that perfect match in sound always means perfect rhyme (which should intuitively get the highest possible rhyme rating of 1.0), we froze the values on matrix diagonal to reflect it. In Plecháč [2018], the diagonal was being recalculated as the rest of the matrix, and although it seemed to converge to high numbers we felt it did not reflect the superiority of the perfect match.

To calculate the probabilities in the matrix, we used the formula from Plecháč [2018] adding the adjustments described above. The formula for a the value in the matrix on coordinates i, j is as follows:

$$p(i,j) = \begin{cases} 1.0, & \text{if i=j} \\ \dfrac{f(i,j)}{f(i,j) + f(i)f(j)}, & \text{otherwise} \end{cases}$$

where *f(i,j)* represents the relative frequency of the co-occurrence of the pair of components i and j, and similarly *f(i)* represents the relative frequency of the component i, i.e. $f(i) = count(i)/total\_count$. In both relative frequencies, only the occurrences from the co-occurrences of two different components we used to reflect the frozen diagonal.

Technically, the matrix is only triangular because the order of i and j does not matter.

3.5.2 Training the detector

What we have just described is basically step 3 of learning the matrix using the EM algorithm. The whole outline can be summed up in 4 steps:

1. Initialize the matrix of vertical collocation probabilities.
2. Find rhymes using this matrix.
3. Adjust the matrix based on detected rhymes.
4. Repeat steps 2 and 3 until convergence, i.e. when matrix reaches a state it already had.

To improve the matrix in step 3, in each iteration we will count the vertical collocations only from the pairs that were marked as rhyming in the previous iteration.

For the matrix initialization, Plecháč calculates the vertical collocations in the data and preserves only pairs with number of vertical collocations above a set threshold. We instead initialized it with the default value of 0.2.

Finding rhymes (step 2) is further described in Section 3.6.1.

3.5.3 Rhyme rating

We will first calculate rhyme ratings for pairs of verses and then use them to calculate an overall song rating. Not only do these rhyme ratings help us evaluate the rhyming quality of the song but they might also be an interesting feedback for the users (e.g. authors of song lyrics) – we thus show the rhyme ratings in our web visualization as described in Section 5.2.2.

For each rhyme, we would like to assign a rating between 0 and 1. Since perfect rhyme is often in literature described as superior because it represents the perfect match in sound it will logically receive the highest rating of 1.

For other cases, such as imperfect rhymes or some forced rhymes, where it was necessary to move the stress to the rhyming part, we will give a penalty of -0.1 for moving the stress. This includes both the case when stress on one line had to be moved to match the other, and the case when stress on both lines had to be moved to exclude non rhyming syllables from the rhyme.

When the phoneme sounds are similar, we will assign rhyme rating as a simple multiplication of probabilities for the individual components from the matrix.

This is the final formula

$$rhyme\ rating = stress\_penalty + \prod_{i,j\ \epsilon\ rhyming\ part} p(i,j)$$

where *stress_penalty* is 0 if stress was not moved, -0.1 otherwise. Indices *i, j* iterate over components in rhyming parts of the both words.

3.6 Scheme

3.6.1 Finding all rhymes

To search for rhymes in the full lyrics, we need to decide which verse pairs to check. The most straight-forward approach would be "brute force" – try each line with all the other lines. Besides its obvious disadvantage of increased time requirements it also detects rhymes that span across tens of lines. It is not strictly defined how many lines apart can the rhyme fellows be to still be considered a rhyme – the author can even make it a part of his artistic expression, e.g. in *Author's Prologue* by Thomas [1952] the 1st line rhymes with 102th, 2nd with 101th and so on. Realistically, a rhyme between a line in the beginning of the lyrics and 20 lines later, would hardly be noticed at all by song's listeners – it requires a close proximity of rhyme fellows within the poem. We decided to set the default window size to 5, to include support for rhyme propagation within stanzas.

We traversed the song lyrics from beginning to the end, line by line, for each line trying all rhyme candidates in the given window.

Some words may have multiple possible pronunciations – in that case we evaluate each possible combination of pronunciations for all words in the given pair. For every such combination we assign rhyme rating and create a list of candidates.

As List [2020] states, it is difficult to know where to draw the line between intended rhyme and accidental word similarity. However, a threshold must be set to draw a boundary between the two. By default, we have set this threshold to 0.8 because we found it to work quite well in our data. However, it can be adjusted by users to their value of choice; similarly with other parameters such as window size. The list of rhyme candidates can now be reduced to the ones with rhyme rating above the selected threshold.

From this list of candidates for each line, we select only the candidate with the highest rhyme rating. When the ratings are identical, we select the closest line. Other candidates are saved, in case changes need to be made later in scheme adjustment phase, cf. Section 3.6.3. When the line does not have any suitable rhyme candidates, it is assigned rating 0. If the line rhymes with a candidate that is already a part of a rhyme, they join together into a rhyme group of 3 (or more) lines.

1	Packs of Backwoods and Dutches, leave the Swishers for the **sweeties**
2	Only roaches in the dishes we be ripping up your **beedies**
3	We be ripping up your treaties, I ain't ripping if it's **seedy**
4	I ain't riffing, I ain't raffing, I'm just rapping on a **CD**

Table 3.6: Example of scheme adjustment from *aaaa* to *aabb*. Excerpt from the song *Whooping Cough* from our dataset.

3.6.2 Assigning rhyme scheme

Rhymes in songs or poems are typically marked using a rhyme scheme. That means each verse gets assigned a letter – lines that share the same letter rhyme and those with different letters do not. We also decided to adapt this common notation. In case the song needs more letters than there are in the alphabet, we will add another letter and continue alphabetically – a, b, c, ..., aa, ab, ac, ..., ba, bb, bc, ..., ca, etc.

There are two options for representing non-rhyming lines. The first is to assign every non-rhyming line the same default character. We chose this option as the default (using character "–"), because it is easier to read for the user. The second option is to assign each non-rhyming line a unique rhyme scheme letter. This approach is more suitable for metrics that look at rhyme scheme letters as clusters of rhyming words. We support both options and can convert one into the other when needed.

3.6.3 Scheme adjustment

In some cases, the algorithm of selecting the best rhyme for each line does not yield the best possible scheme. Consider, for illustration, the example in Table 3.6. There is a perfect rhyme between lines 1-2 and 3-4, and a forced rhyme between lines 2-3. With the algorithm as is, it would receive the *aaaa* scheme. However, that is not what a human, for example a gold data annotator, would assign. He would see that similarity inside the first and the second tuple is greater so they logically form two rhyme groups and the scheme is *aabb*.

Additionally, song rating (as defined in the following section 3.7) for scheme *aaaa* would be less than 1 (because of the lower rating of the forced rhyme between lines 2 and 3) while the song rating for *aabb* would be equal to 1. Loosing rating by marking weaker rhymes does not make sense, so we must add an exception to only keep the better score. We can see that this problem is similar to the problem of maximizing song rating.

To address this issue, we have to do one more iteration over the resulting rhyme groups. We focused only on rhyme groups of size 4 and larger because for smaller group such changes would not increase the score. For a larger group, we iterate over the rhymes, starting from the ones with the lowest rating, and try how would removing this rhyme affect the song rating. If it increased the song rating, we keep the change and split the rhyme group as necessary. If the rating did not increase, we try removing the next rhyme, and if we ran out of rhymes to remove, we keep the group as is.

3.7 Calculating song rating

The next step is to combine these rhyme ratings into one final rating for the entire song. We will use the straight-forward approach of averaging the assigned rhyme ratings. The dilemma here is where to store the rhyme rating since rhyme is a property of two lines. The first logical idea would be to store it with each line participating in the rhyme. However, then we would add it to the final song rating twice. Moreover, for larger rhyme groups, it would be disproportionate, because third and all the following lines would be added only once.

Therefore, we decided to store the rating only with the second line of the rhyme. That means the first line in each rhyme group will always be assigned the default value "-". It cannot be assigned rhyme rating 0 because that would mean it is a non-rhyming line and would lower the final average. In summary, song rating is the average of rhyme ratings for all rhymes except the lines with unassigned rating "-".

4. Evaluation

In this chapter, we will evaluate our rhyme detector. In the beginning, we will compare its performance with RhymeTagger. Then we will use it to analyze our dataset and calculate statistics about song lyrics.

4.1 Performance evaluation on schemes

To evaluate the quality of our Rhyme Detector, we will compare its rhyme scheme predictions with gold data. We have two annotated datasets we can use – Chicago Rhyming Poetry Corpus (ChicagoRPC) and the annotated test subset of our dataset (Genius). We will also compare our performance with that of RhymeTagger.

4.1.1 Taggers

We will be comparing different variants of the taggers to also evaluate the influence some factors have on the performance. The variants are following:

- **RhymeTagger (ChicagoRPC)** – the original version of RhymeTagger, as it was trained by Plecháč.
- **RhymeTagger (Genius$\frac{1}{3}$)** – RhymeTagger trained on one third of our data.[1]
- **Rhyme Detector** – our detector, as described in Chapter 3, with the default settings, trained until the EM algorithm reached convergence in 4$^{\text{th}}$ iteration.
- **Rhyme Detector – experiment** – an experimental version of our detector that was not trained with the EM algorithm, but instead the matrix was created by counting the co-occurrences in the data and using their probabilities. In this experiment, we counted the co-occurrences for all the lines; in case the line did not have a rhyme within the window, we took the line immediately preceding it instead.
- **Rhyme Detector – 1 iteration** – our detector after the 1$^{\text{st}}$ iteration of the EM algorithm.
- **Rhyme Detector – perfect** – our detector configured to only finding perfect rhymes – the co-occurrence matrix is an identity matrix.

4.1.2 Scores

To evaluate the performance, we need to find an appropriate measure that could compare the gold scheme with the prediction. This task may seem easy at first, but we need to be careful because the straight-forward approach of comparing

[1] Training of RhymeTagger was significantly more time consuming than training of our detector so we did not manage to train in on a larger portion.

the schemes letter by letter would not work. If the prediction made an error in the beginning it would alphabetically shift the rest of the scheme and it would no longer match with the gold data.

Last Index Score (LI) To solve this problem, we propose Last Index Score (LI score). The idea is to convert the scheme from letter representation to last rhyme's index representation. This means for each line, we will use the index of the last line that rhymed with it. If the line does not have a rhyme, we will use index -1. With such representation, we can compare these indices directly, independently from scheme letters, and the proportion of matching indices will give us a score between 0 and 1. For an illustrative example, see Table 4.1.

line	Straight-forward		Last Index	
index	Gold	Prediction	Gold	Prediction
0	a	a	-1	-1
1	a	b	0	-1
2	b	c	-1	-1
3	b	c	2	2
4	c	d	-1	-1
5	c	d	4	4
6	b	c	3	3
7	b	c	6	6
	SCORE: 0.125		SCORE: 0.875	

Table 4.1: Comparison of the straight-forward approach and LI score.

ARI score If we look at the rhyme scheme, we can notice that scheme letters can equally be represented as line cluster. All lines that share a scheme letter form one cluster (a rhyme group), and every line that does not have a rhyme is a cluster of its own. Adjusted Rand Index (ARI) Score[2] is a corrected-for-chance statistical measure of similarity between two data clusterings. We can therefore convert the schemes to use a unique letter for each non-rhyming line and use this score to evaluate the similarity of the schemes.

For these two scores, we include both micro and macro average on the dataset. Macro average is the average of all song scores, while micro average is the average of song scores weighted by the number of lines in each song.

4.1.3 Comparison of the results

We calculated the aforementioned scores for all variants of taggers and summed up the results in Tables 4.2 and 4.3.

They both performed better on the Chicago Poetry Corpus (see Table 4.2). Surprisingly, our detector performed the best after 1st iteration, but the difference is not substantial.

[2] https://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

	ARI		LI	
	macro	micro	macro	micro
RhymeTagger (ChicagoRPC)	**0.9463**	**0.9384**	**0.9635**	**0.9534**
RhymeTagger (Genius)	0.8819	0.8822	0.9282	0.9202
Rhyme Detector	0.9040	0.8915	0.9413	0.9272
Rhyme Detector – experiment	0.9025	0.8906	0.9406	0.9272
Rhyme Detector – 1 iteration	**0.9066**	**0.8926**	**0.9426**	**0.9277**
Rhyme Detector – perfect	0.8824	0.8732	0.9293	0.9149

Table 4.2: Evaluation of taggers on Chicago Rhyming Poetry Corpus.

Surprisingly, as we can see in Table 4.3, RhymeTagger (Genius) performed worse on the *Genius* dataset than the original pre-trained version. The possible cause is the nature of our data – song lyrics have a significantly smaller percentage of rhymes than poems, so the collocations approach might not work as well for our dataset. This might also be the reason why our detector, trained only on our data, does not perform as well as RhymeTagger.

	ARI		LI	
	macro	micro	macro	micro
RhymeTagger (ChicagoRPC)	**0.6519**	**0.6627**	**0.8021**	**0.8139**
RhymeTagger (Genius)	0.6309	0.6443	0.7883	0.8040
Rhyme Detector	0.6157	0.6306	0.7716	0.7861
Rhyme detector – experiment	**0.6359**	**0.6571**	**0.7866**	0.8020
Rhyme detector – 1 iteration	0.6096	0.6256	0.7682	0.7832
Rhyme Detector – perfect	0.6224	0.6410	0.7838	**0.8030**

Table 4.3: Evaluation of taggers on test subset of Genius.

4.2 Statistical analysis of the dataset

Although our detector was trained on our dataset, it was unsupervised so we can still use our detector to evaluate this dataset and give us new statistical information about a large number of song lyrics. We ran rhyme detection for nearly half a million songs and summed up the results in Tables 4.4, 4.5, 4.7, 4.8, and 4.6. In the rest of this section, we will look at them more closely and discuss the outcome that might be surprising, or the opposite, confirms the specific characteristics of a particular genre. Extreme values are emphasized in the tables. Keep in mind, that the lyrics and their classification to genres is crowd-sourced and might be biased.

In Table 4.4, we sum up the basic information for all genres including the portion of lines that rhyme and we can already see some interesting results. Surprisingly, the highest portion of rhyming lines is in the R&B genre. We do not see any characteristic of this genre that could cause this. However, it is not a big difference and maybe having more examples from this genre would make it less significant.

Genre	Songs	Lines			
		total	avg per song	rhyming	(%)
Pop	293,679	9,104,273	31.001	4,536,554	49.8
Rap	99,185	5,661,603	**57.081**	2,849,905	50.3
Rock	34,372	1,087,245	31.632	523,879	48.2
R&B	5,125	225,344	43.970	117,862	**52.3**
Country	3,816	121,207	31.763	61,142	50.4

Table 4.4: General statistics about dataset and rhymes, per genre.

We can see that throughout genres typically only about half of the lines rhyme. There are more reasons that could cause this. One possibility is that rhyming in songs is not as essential as perhaps in poems and there are in fact no more rhymes to be detected. A more likely possibility is that there are more rhymes but were not detected, either because our detector did not see them or because of the imperfect formatting of our dataset. For example, in Figure 5.1, lines 4 and 5 could have a rhyme *kiss you – miss you*, but on the 5$^{\text{th}}$ line *miss you* is followed by ", babe" what causes this rhyme to go unnoticed by our detector.

Predictably, rap has a significantly higher average number of lines per song, which confirms the fact that this genre is more talkative. What may be unexpected is that it is nearly two times more than for the other genres – only R&B slightly stands out but that can be explained by the fact that these two genres are known to have influenced each other throughout history.

Rhyme type	Genre				
	Pop	Rap	Rock	R&B	Country
Perfect	81.1	**67.1**	80.9	79.2	80.0
— masculine	72.5	58.2	72.3	70.2	73.5
— feminine	7.9	8.4	7.7	8.5	6.2
— dactylic	0.7	0.5	0.9	0.5	0.3
Imperfect	12.0	**22.3**	12.1	13.5	12.2
Forced	6.9	**10.6**	7.0	7.3	7.8

Table 4.5: Percentage of different rhyme types from all rhymes in the dataset, per genre.

Next, Table 4.5 shows distribution of different rhyme types. It did not come as a surprise that the most common type, by a long shot, is perfect masculine. The reasons behind this might be several – not only has perfect match the strongest effect melodically, it is also the easiest to come up with, and makes the lyrics easy to remember. The multi-syllable perfect rhymes have a lower percentage as longer matching word pairs are rather rare. The amount of forced rhymes might be higher in reality because their detection is inherently the hardest and they might be missed more often.

Concerning rhyme types, we see that genres are generally not very different, except for rap. Rap is very unique with rhymes, its artists are known for playing

with them more creatively, using internal rhymes, consonance, and assonance more often. They frequently play with emphasis what can be seen as a rapid increase in imperfect rhymes. There are more forced rhymes as well and perfect rhymes are decreased as a result.

Rhyme category	Genre				
	Pop	Rap	Rock	R&B	Country
1-syllable (2-component) rhymes	91.1	90.3	91.1	90.6	93.3
2-syllable (5-component) rhymes	8.2	9.2	8.0	8.9	6.4
3-syllable (8-component) rhymes	0.7	0.5	0.9	0.5	0.3
Perfect sound match	93.1	**89.4**	93.0	92.7	92.2
Stress moved	14.5	**28.3**	14.5	16.5	14.8

Table 4.6: Statistics about rhyme properties in general, disregarding rhyme types, in percentage from total rhymes.

Table 4.6 is quite similar to the previous table, but by counting syllables regardless of rhyme type, and evaluating sound match and stress separately, it offers us a little bit different angle. The low percentage of multi-syllable rhymes may be caused by the fact that we only look at the components after the last stress and stress is more often on the last or penultimate syllable – even if more syllables rhyme, only shorter rhyme will be detected.

The decreased match in sound and increased moving of stress in rap confirm the unique properties of rap we have seen previously.

A slightly increased percentage of 1-syllable rhymes in country may be noteworthy but we see no significant properties of country that could support this as a general claim.

Genre	groups per song		groups per	group size	
	avg	max	100 lines	avg	max
Pop	6.13	169	19.79	2.52	159
Rap	**11.48**	224	20.12	2.50	98
Rock	6.09	81	19.26	2.50	68
R&B	8.62	48	19.61	2.67	42
Country	6.68	98	**21.02**	**2.40**	**24**

Table 4.7: Statistics about rhyme group size and counts per genre.

Table 4.7 summarizes statistics concerning size of rhyme groups. We can observe nearly double average size for rap compared to other genres, which directly corresponds to nearly double average song length, as we have seen in Table 2.1.

An interesting observation can be made about country – average number of rhyme groups per 100 lines is slightly higher than for other genres. This corresponds with average group size being lower – obviously country tends to contain more and smaller rhymes groups. It would be interesting to know whether this is only a property of our dataset or a property of country music in general. Although the maximum group size does not tell us any general information about

the group because it may only be an outlier, it is still interesting to see that this number is again the smallest for country.

	Genre				
	Pop	Rap	Rock	R&B	Country
Average song rating	0.432	**0.599**	0.420	0.520	0.456
Median song rating	**0.521**	0.380	0.357	0.235	0.269

Table 4.8: Song rating per genre.

Looking at average and median song ratings in Table 4.8, we can observe two curious extremes – rap having the highest average rating and pop with the highest median rating. So there must be some extremely highly rated rap songs that pulled up the average.

Although we did not predict this result, it can be a sign that some artists took the importance of rhyme in rap very seriously and elaborately incorporated it densely into their lyrics.

The highest median in pop shows that many pop songs are filled with more rhymes, which can be explained by their strong tendency to be memorable. However, it seems that there are some low extremes that pulled the average rating down.

5. Visualization

To make the results more approachable for a common user, it is always better to visualize them in some way. Therefore, to demonstrate our detector's capabilities, we created a website that visualizes rhymes and their quality, shows statistics, and allows users to experiment with the parameters. This way, it can be used by anyone without any programming background.

5.1 Input

The input page consists of a text-box for song lyrics, a card with parameters, and an *Analyze* button, as seen in Figure 5.1.

The text-box expects text input of song lyrics, separated into verses with newlines such that rhymes are at the end of the line. Once text is entered, *Analyze* button will be enabled.

For analysis, the default parameters are pre-filled, but users can choose to change them. Selecting the checkbox *Perfect rhymes only* will trigger the detector to only detect perfect rhymes. The *Window* size specifies the maximum number of lines between rhyme fellows (window=1 means checking the previous line only). ~~Changing the size of the window will affect how many lines apart can rhyme be.~~ Smaller window is better for creating rhyme schemes, while longer window (e.g. equal to the song length) will give better overview of rhyme repetition throughout the entire song and give a more interesting matrix visualization. *Rhyme threshold* parameter sets the minimal rhyme rating – rhymes with lower rating will be discarded.

Pressing the *Analyze* button will start the analysis and minimize the input page. For the duration of rhyme detection, a loader box~~loader~~ is shown to inform the users their request is being processed. When the back-end returns the results, an *Analyze*~~analyze~~page is expanded to show the visualizations. If desired, users can expand the input page, edit the input, and re-submit for analysis.

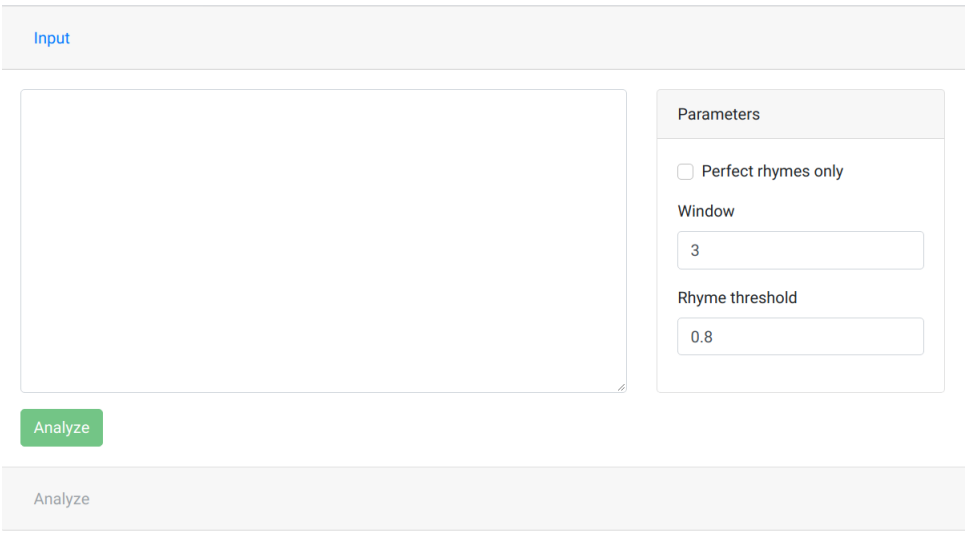

Figure 5.1: Website's form for entering the lyrics and setting the parameters.

spíš bych tu page nazval Analysis či Results, ale na předělání screenshotů asi není čas, tak to nechte.

5.2 Visualization of the results

The visualizationpage contains lyrics with scheme, matrix visualization of rhymes, and short statistics. It is primarily designed for songs of short or moderate length, longer lyrics may not fit on the screen with the analysis side-by-side, and will have to be rearranged in a column, which makes the results less comfortable to read.

nebo The *Analyze* page

5.2.1 Lyrics and statistics

Lyrics with their assigned scheme letters and line number are shown on the left, as we can see in Figure 5.2. Rhyming lines are highlighted, each with a color corresponding to its rhyme type. For the sub-types of perfect rhyme, we selected similar colors to indicate that they are more closely related – namely red for *masculine*, orange for *feminine*, and yellow for *dactylic* rhymes. *Imperfect* rhymes are highlighted in blue and *forced* in green color. When the user hovers over a rhyming line, this line and all lines rhyming with it are highlighted.

Statistics areis shown on the right under the matrix. They contain theIt contains song rating and percentages of different rhyme types in the song.

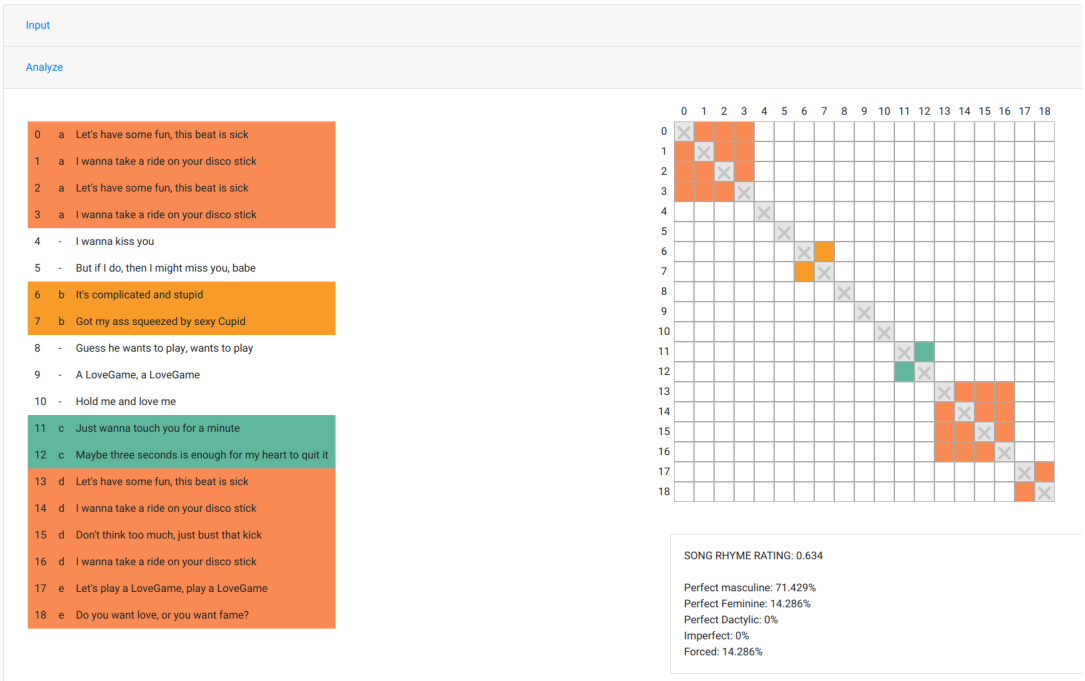

Figure 5.2: Screenshot from an analysis with theanalysis with default window size. Example from *Love Game* by Lady Gaga.

5.2.2 Matrix

To make the visualization more creativea creative visualization, we took inspiration from Colin Morris.[1] He came up with an idea to represent the repetitiveness of lyrics by self-similarity matrix and he demonstrated it in his project SongSim.[2]

[1] https://github.com/colinmorris
[2] https://colinmorris.github.io/SongSim/#/

In his matrix, there is one row and one column for each word of the song. For each cell, if the word in given row and column are identical, the cell is colored, otherwise it stays white (Figure 5.3).

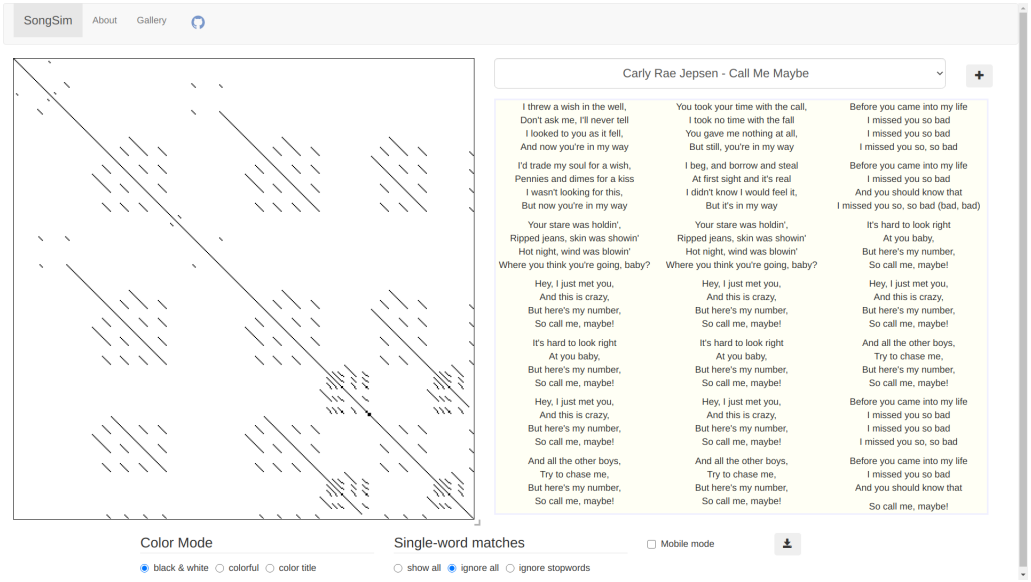

Figure 5.3: Screenshot of Collin's SongSim visualization of song's repetitiveness.

Instead of words, in our matrix, we compare lines and highlight rhymes~~rhymes~~. For rows and columns, we use rhyme scheme letters for the corresponding line, and when they agree, the matrix cell will receive the color of this rhyme's type, as described in Section 5.2.1 (Figure 5.2). For comparison with the default window, in Figure 5.4, we include a screenshot with a longer window to better demonstrate the matrix.

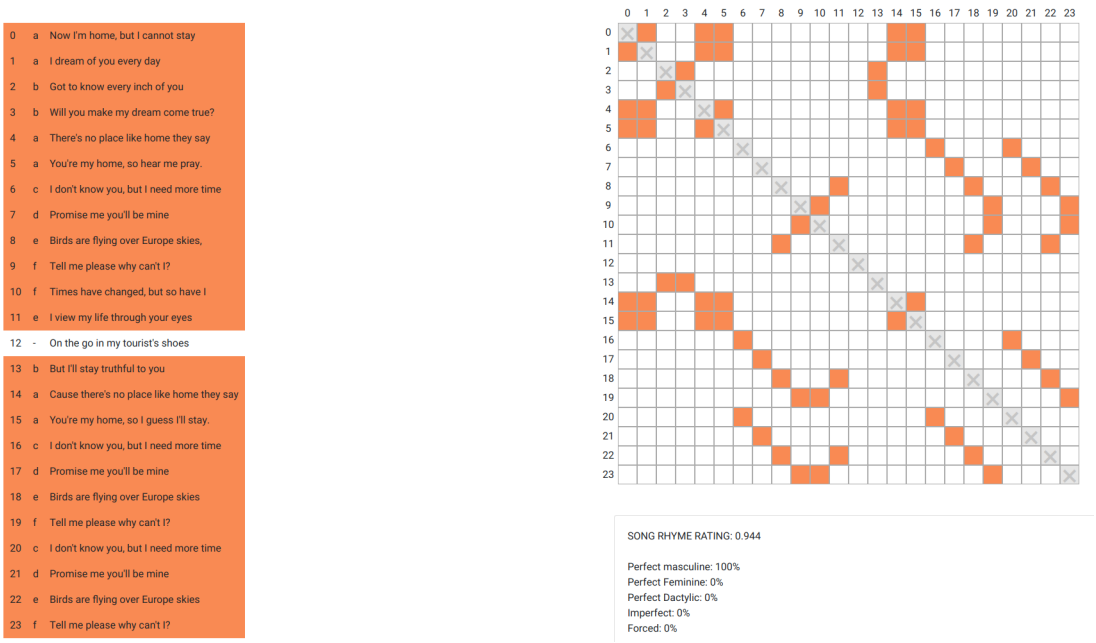

Figure 5.4: Screenshot from an analysis with a window size matching the song length. Example from *Europe's Skies* by Alexander Rybak.

So far, users haveuser has been given a large-picture overview of the entire song. To explore the details, users can view rhyme's properties by hovering over the particular matrix cell. An pop-over boxPopover will display more details as shown in Figure 5.5. *Rhyming phonemes* for both rhyming lines display only phonemes participating in the rhyme – meaning from the last stressed phoneme (or where the stress was moved) onward. Lines, that correspond to this rhyme, will also be highlighted in the text on the left.

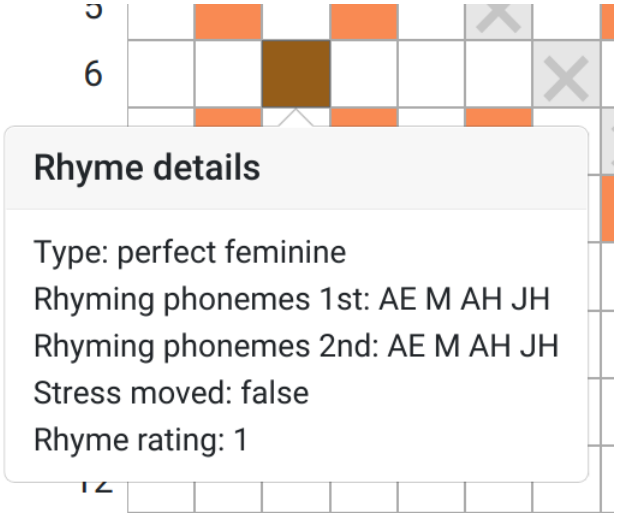

Figure 5.5: Detail of a pop-over box for a givenpopover over one matrix tile.

5.3 Technologies

For the front-end of the web page, we used a TypeScript-based open-source web application framework —Angular (Google [2021]). As a design library, we used Bootstrap,[3] and ported version

Buď and a ported version, pokud to portoval někdo jiný. Nebo and ported a version, pokud jste to portovala Vy.

of standard Bootstrap's components to Angular – *ngx-bootstrap*.[4] The back-end simple REST API is written in Python using a micro-web framework *Flask*.[5] It calls our detector, in a classic variant without any modifications, as it was used for the evaluation in Chapter 4.

We host it at a public url `https://rhyme-detector.brezinovi.sk/` from our personal computer, so some short-term unavailability is possible. In case of any difficulties, please do not hesitate to contact us at *patricia@brezinovi.sk*.

[3] `https://getbootstrap.com/`
[4] `https://valor-software.com/ngx-bootstrap/`
[5] `https://flask.palletsprojects.com/en/2.0.x/`

6. Generation

Writing song lyrics is a task that requires creativity. This is one of a few areas where AI has not become better than humans yet. While it is generally capable of keeping similar structure and topic, it still fails in using rhymes and other literary devices.

As described in Section 1.4, the current state-of-the-art tool for generation is GPT-3 [Brown et al., 2020]. We have sent several requests for their API but have not received a reply to this day.

The second-best option was its earlier version GPT-2 (Radford et al. [2019]). It offers pre-trained models of sizes small, medium, large, and XL that can be further fine-tuned to the specific task. Since training models of such sizes has very high memory requirements, we would only be able to fine-tune the small model on available school clusters.

However, our colleagues from ÚFAL at the Charles University agreed to let us generate on GPT-2 XL they are using in their ongoing research project THEaiTRE (Rosa et al. [2021]). It is a vanilla model, meaning that it has not been fine-tuned to any specific task and we do not have resources to fine-tune it. But as their research confirms, it is not necessary because it adapts very well when given a relevant primer, i.e. a text snippet used as a start of the text to be generated. Knowing it has been trained on 8 million web pages, it is highly probable that it has already seen some lyrics data.

Without fine-tuning, the quality of the result depends on the selection of the primer. To choose what works the best, we decided to compare the results using primers of different lengths. First, we tried one-line primers. As expected, it was not enough for GPT-2 to recognize it as poetry, and it continued with full paragraph as you can see in Figure 6.1.

> A chair is still a chair
> In most other aspects, I think the two camps in this debate can agree. We're in good company, for instance, as far as the law is concerned. When it comes to the "should" part of the debate, the fact that some states have no-fault insurance mandates, and others have them, is not a matter of opinion, or of opinion alone.

Figure 6.1: Result of generation (in blue) using a 1-line primer (in black).

Subsequently, we iteratively increased the length of the primer exponentially. We conducted experiments with several song lyrics, but the results did not differ ~~very~~ much, so we only show one example in Table 6.1.

Except for the 4-line primer, it was able to successfully replicate the poem-like structure. It also used a lot of repetition, even when the original text did not contain any (e.g. the 2-line primer). We assume~~,~~ it must have already learned that repetition is very common for this kind of text structures. However, it was not successful in creating rhymes of its own.

Although longer primer seems to be a safer path to good results, they do~~it~~ ~~does~~ not seem to be significantly better than the result from a 2-line primer.

Occasionally, it appears to miss the clue and continue with regular text, as it did with the 4-line primer here.

Original	2-line primer
A chair is still a chair Even when there's no one sittin' there But a chair is not a house And a house is not a home When there's no one there to hold you tight And no one there you can kiss goodnight Woah girl	A chair is still a chair Even when there's no one sittin' there I still think there are people here A chair is still a chair Even when there's no one sittin' thereI still think there are people here A chair is still a chair
4-line primer	**8-line primer**
A chair is still a chair Even when there's no one sittin' there But a chair is not a house And a house is not a home So why do we make up the terms? It's not just us who insist on the word house, we use it in all sorts of other contexts too. For example, if you want to use a single word to refer to a group of people, you might say: If you want to invite everyone to an event, you might say: It's not hard to see why we seem to love using a term like 'house' to refer to a group of people – it's the easiest word for us to remember.	A chair is still a chair Even when there's no one sittin' there But a chair is not a house And a house is not a home When there's no one there to hold you tight And no one there you can kiss goodnight Woah girl A room is a still a room And a still a room is still a home When there's no one there to hold you tight And no one there you can kiss goodnight I'm a little man on a little island And a lonely little bird on a lonely little island But a lonely little bird is not a castle And a lonely little bird is not a home And a lonely little bird is not a world I'm a little man in a little house And a lonely little bird in a lonely little house

Table 6.1: Beginning of "A House Is Not A Home" lyrics by Luther Vandross. Comparison with results generated using 2, 4, and 8-line primers.

Overall, GPT-2 succeeded in replicating the poetic form and structure. It creatively generated meaningful content that was ~~very~~ close to human-written. A slight give-away is excessive repetition, but for an individual example, it could be mistaken for author's style.

Osobně bych GPT-2 u básní nehodnotil tak pozitivně, ale snad je jasné, že je to spíš Váš názor (žádná exaktní evaluace). Aspoň jsem ale smazal to “very” close.

Conclusion

We started this thesis with a thorough research about rhyme and tools used for its detection, visualization, and generation. We explored our dataset and cleared it from impurities to improve our chances for a better analysis. After overcoming many obstacles of creating a rhyme detector, we evaluated it and performed an analysis over our entire dataset. At the end, we visualized the results and explored the generation using GPT-2.

Designing the detector was not easy, we rebuilt it several times as unpredictable exceptions came our way. The biggest difficulty was working with crowdsourced data – although we did what we could in the pre-processing phase, still in a dataset this large, there were words we had to deal with almost individually.

Another problem was the ambiguity of the resulting scheme. Often, there is no single correct scheme to be assigned. It is possible that different people would assign different schemes to the same song because someone would group all rhyming lines together under one letter, but another person may create separate groups by stanzas or other rules. Clearly, for evaluation we only have the gold scheme that was assigned by the annotator. We needed to compensate for this by adjusting the rhyme scheme in Section 3.6.3 so that it is more reminiscent of common human annotation.

Although, in the comparison test, our detector did not outperform Rhyme Tagger, it was still a powerful detector and we believe it was a contribution to this research field. We tried new methods and approaches, and we were able to calculate statistics on almost half a million songs, which confirmed what we suspected about genre differences, but also gave evidence for new interesting findings. Our automated evaluation could, for some use-cases, replace human evaluators.

On top of that, we created an online web visualization that made this tool accessible for public. We implemented an innovative representation of rhymes using a self-similarity matrix.

Finally, we generated lyrics using GPT-2 and experimented with different primers, trying to achieve the best result. The generator was capable of replicating the form of the lyrics and even generating meaningful content.

Future work

In future, it would be worth to consider a more advanced data pre-processing or a cleaner dataset. Although cleaning such a big dataset has to be done automatically, every mistake can contribute to worse performance of both the detector and the generator.

Alternatively, this pre-processing could be included in a more robust detector, that would take care of typing errors and automatically separate text into verses by rhymes.

For more comparisons with RhymeTagger, it could be interesting to train our detector on ChicagoRPC dataset (the same that RhymeTagger was trained on) and compare the results whether the poetry data help the detector learn rhymes better.

For more evaluation statistics, it could be interesting to design a metric that would evaluate the structure of the text (e.g. metre, rhythm, syllable count, and higher-level structure). Not only would this create a measure that would quantify how GPT-2 generated lyrics resemble human-written ones, but it would probably yield more interesting statistical differences between genres.

Another metric could be designed to evaluate repetitiveness in lyrics. Some repetition is common in lyrics but there is no implicit way to quantify how much is normal. This could also be used, in combination with other metrics, to automatically recognize machine-generated lyrics.

An interesting experiment would be to combine the detector and the generator to get better generated results. After generation of new lyrics, it could be evaluated and regenerated until the score reached a desired threshold.

Having access to GPT-3, we believe more impressing results in generation could be achieved.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

A. Abdul-Rahman, J. Lein, K. Coles, E. Maguire, M. Meyer, M. Wynne, C.R. Johnson, A. Trefethen, and M. Chen. Rule-based visual mappings – with a case study on poetry visualization. *Computer Graphics Forum*, 32(3):381–390, 2013. doi: 10.1111/cgf.12125. URL http://dx.doi.org/10.1111/cgf.12125.

Alexander Bain. *English composition and rhetoric, a manual*. New York, Appleton, 1867.

Chris Baldick. *The Oxford Dictionary of Literary Terms*, volume 3. Oxford University PressPrint, 2008. ISBN 9780199208272.

Bennet Bergman. Rhyme. 2017. URL https://www.litcharts.com/literary-devices-and-terms/rhyme.

T. V. F. Brogan. *The Princeton Handbook of Poetic Terms*, volume 3. Princeton University Press, 2016. ISBN 9781400880645.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

Tanya Clement, David Tcheng, Loretta Auvil, Boris Capitanu, and Megan Monroe. Sounding for meaning: Using theories of knowledge representation to analyze aural patterns in texts. *DHQ: Digital Humanities Quarterly*, 7(1), 2013.

R. Delmonte and A. M. Prati. SPARSAR: An expressive poetry reader. pages 73–76, apr 2014. doi: 10.3115/v1/E14-2019. URL /https://www.aclweb.org/anthology/E14-2019.

Arthur M Eastman, Alexander Ward Allison, Herbert Barrows, et al. *The Norton Anthology of Poetry*. Norton New York, 1970.

Google. Angular, 2021.

Erica Greene, Tugba Bodrumlu, and Kevin Knight. Automatic analysis of rhythmic poetry with applications to generation and translation. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 524–533, 2010.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Jey Han Lau, Trevor Cohn, Timothy Baldwin, Julian Brooke, and Adam Hammond. Deep-speare: A joint neural model of poetic language, meter and rhyme. *arXiv preprint arXiv:1807.03491*, 2018.

Wayne A Lea. *Trends in speech recognition*. Prentice Hall PTR, 1980.

Johann-Mattis List. Automated detection of rhymes in texts (from rhymes to networks 4), 2020. URL http://phylonetworks.blogspot.com/2020/07/automated-detection-of-rhymes-in-texts.html.

LiteraryDevices Editors. Rhyme - examples and definition of rhyme as a literary device, Dec 2020. URL https://literarydevices.net/rhyme/.

Nina McCurdy, Julie Lein, Katharine Coles, and Miriah Meyer. Poemage: Visualizing the sonic topology of a poem. *IEEE transactions on visualization and computer graphics*, 22(1):439–448, 2015a.

Nina McCurdy, Vivek Srikumar, and Miriah Meyer. Rhymedesign: A tool for analyzing sonic devices in poetry. In *Proceedings of the Fourth Workshop on Computational Linguistics for Literature*, pages 12–22, 2015b.

Luis Meneses and Richard Furuta. Visualizing poetry: Tools for critical analysis. *paj: The Journal of the Initiative for Digital Humanities, Media, and Culture*, 3, 2015.

Kyubyong Park and Jongseok Kim. g2pe, 2019. URL https://github.com/Kyubyong/g2p.

Barbara Zurer Pearson, PA de Villiers, K Brown, and E Lieven. Encyclopedia of language and linguistics, 2005.

Petr Plecháč. A collocation-driven method of discovering rhymes (in czech, english, and french poetry). In *Taming the Corpus*, pages 79–95. Springer, 2018.

Petr Plecháč. Collocation-driven method of discovering rhymes in a corpus of czech, english, and french poetic texts. 2017. URL http://versologie.cz/talks/2017basel/.

ProseVis. Prosevis. 2014. URL https://sourceforge.net/projects/prosevis/.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Sravana Reddy and Kevin Knight. Unsupervised discovery of rhyme schemes. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 77–82, 2011.

Alberto Romero. A complete overview of GPT-3 — the largest neural network ever created, 2021. URL https://towardsdatascience.com/gpt-3-a-complete-overview-190232eb25fd.

Rudolf Rosa, Tomáš Musil, Ondřej Dušek, Dominik Jurko, Patrícia Schmidtová, David Mareček, Ondřej Bojar, Tom Kocmi, Daniel Hrbek, David Košťák, et al. Theaitre 1.0: Interactive generation of theatre play scripts. *arXiv preprint arXiv:2102.08892*, 2021.

Marc Schröder, Anna Hunecke, and Sacha Krstulovic. Openmary–open source unit selection as the basis for research on expressive synthesis. In *Blizzard Workshop*, 2006.

The Editors of Encyclopaedia Britannica. Encyclopedia britannica. 2014. URL https://www.britannica.com/.

Dylan Thomas. Author's prologue. *Collected Poems 1934-1952*, 1952.

Ottokar Tilk and Tanel Alumäe. Bidirectional recurrent neural network with attention mechanism for punctuation restoration. In *Interspeech*, pages 3047–3051, 2016.

Jona van der Schelde. Phonological and phonetic similarity as underlying principles of imperfect rhyme. 2020.

Elizabeth Walter. *Cambridge advanced learner's dictionary*. Cambridge university press, 2008.

Andrew Weiss. Google n-gram viewer. *The Complete Guide to Using Google in Libraries: Instruction, Administration, and Staff Productivity*, 1:183, 2015.

Viktor Zhirmunsky and John Hoffmann. Introduction to rhyme: Its" history and theory". *Chicago Review*, 57(3/4):121–128, 2013.

List of Figures

List of Tables

Glossary of literary and technical terms

consonant cluster A sequence of syllables without a vowel. 9

end rhyme Rhyme at the end of line. 21

gold data In data classification, it is the dataset with correct labels already assigned. It can be used for supervised learning or an evaluation of unsupervised learning. 22, 30

headless mode A mode in which software runs on hardware without a graphic user interface, e.g. a script in terminal. 8

internal rhyme A rhyme that occurs in the middle of lines of poetry, instead of at the ends of lines.. 21, 24, 34

LSTM Long-Sort Term Memory – a type of recurrent neural network. 10

quatrain A type of stanza consisting of four lines. 10, 14

rhyming part A part of word (or multiple words) that rhymes (is identical or similar in sound) with other word/words. 5

rime riche A rhyme produced by agreement in sound not only of the last accented vowel and any succeeding sounds but also of the consonant preceding this rhyming vowel. 4, 5

sonnet A poetic form traditionally containing 14 lines written in iambic pentameter with rhyme scheme *abab cdcd efef gg*. 10, 14

syllable peak A nucleus of a syllable – either a vowel or a syllabic consonant. 9

transformer model A deep learning model that adopts the mechanism of attention, differentially weighing the significance of each part of the input data. 14

A. Attachments

A.1 IPA and ARPAbet transcription table

Following tables show the transcription between IPA and ARPAbet for consonants (Figure A.1) and vowels (Figure A.2). The ARPAbet phoneme set used by CMUdict is shown, as described on their website[1]. Note, that IPA diphthongs are not transcribed separately but as one two-character ARPAbet symbol.

ARPAbet	IPA	Example
B	b	be
CH	tʃ	cheese
D	d	dee
DH	ð	thee
F	f	fee
G	g	green
HH	h	he
JH	dʒ	gee
K	k	key
L	l	lee
M	m	me
N	n	knee
NG	ŋ	ping
P	p	pee
R	r	read
S	s	sea
SH	ʃ	she
T	t	tea
TH	θ	theta
V	v	vee
W	w	we
Y	j	yield
Z	z	zee
ZH	ʒ	seizure

Table A.1: Consonant phonemes – transcription between IPA and ARPAbet.

[1] http://www.speech.cs.cmu.edu/cgi-bin/cmudict

ARPAbet	IPA	Example
AA	ɑ	odd
AE	æ	at
AH	ʌ	hut
AO	ɔ	ought
AW	aʊ	cow
AY	aɪ	hide
EH	ɛ	Ed
ER	ɜr	hurt
EY	eɪ	ate
IH	ɪ	it
IY	i	eat
OW	oʊ	oat
OY	ɔɪ	toy
UH	ʊ	hood
UW	u	two

Table A.2: Vowel phonemes – transcription between IPA and ARPAbet.