UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Álvaro Emílio Prüsse

Uma Ferramenta para Auxiliar na Criação de Letras Musicais

Florianópolis
2023

Álvaro Emílio Prüsse

Uma Ferramenta para Auxiliar na Criação de Letras Musicais

Trabalho de Conclusão de Curso do Curso de Ciência da Computação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Raul Sidnei Wazlawick, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Prüsse, Álvaro Emílio
Uma Ferramenta para Auxiliar na Criação de Letras Musicais / Álvaro Emílio Prüsse ; orientador, Raul Sidnei Wazlawick, 2023.
73 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. ferramenta. 3. letras musicais. I. Wazlawick, Raul Sidnei. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Álvaro Emílio Prüsse

Uma Ferramenta para Auxiliar na Criação de Letras Musicais

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computaçãoe aprovado em sua forma final pelo Curso de Ciência da Computação.

Florianópolis, 13 de novembro de 2023.

Coordenação do Curso

Banca Examinadora:

Prof. Raul Sidnei Wazlawick, Dr.
Orientador

Prof. Roberto Willrich, Dr.
Universidade Federal de Santa Catarina

Felipe Nedel Mendes de Aguiar, Bel.
Universidade Federal de Santa Catarina

Florianópolis
2023

AGRADECIMENTOS

A Deus, na certeza de que, sem Ele, nada posso fazer. A Ele toda a honra e glória.

A minha família, em especial às mulheres da minha vida: minha mãe Tere, minha irmã Adelita e meu amor, Nane.

A todos meus amigos e colegas. Cada um de vocês contribuiu de maneira única para tornar essa caminhada mais agradável.

Ao Laboratório Bridge por todo o aprendizado, pelo ambiente inspirador e pelo apoio constante.

Agradeço a todo o corpo docente da UFSC pelo conhecimento e apoio. Em especial ao meu orientador, Prof. Raul, cuja bondade e humanidade transcendem sua excelência acadêmica.

Aos membros da banca, Prof. Roberto e Felipe, por aceitarem o convite e serem tão atenciosos na avaliação.

RESUMO

Há uma quantidade considerável materiais e ferramentas disponíveis que potencialmente podem contribuir no processo de criação letras musicais. Porém, esse conteúdo encontra-se distribuído em diferentes softwares, sites, livros etc. São raros softwares específicos para criar letras de músicas. A grande maioria das ferramentas existentes no mercado são destinados especificamente para língua inglesa. Este trabalho é destinado à criação de uma ferramenta web para reunir algumas funcionalidades que possam estimular e acelerar o processo de criação de letras musicas na língua portuguesa. As principais funcionalidades da ferramenta são: separação silábica/contador de sílabas, sugestão de palavras relacionadas, sugestão de palavras com a mesma classificação quanto à sílaba tônica. Todas as sugestões são construídas a partir do processamento de letras musicais de um mesmo gênero, o que pode produzir resultados melhores por estarem mais próximas do mesmo contexto. As letras foram extraídas com *web crawling* para formar um *corpus* linguístico. A partir do *corpus*, treinou-se um modelo com do Word2Vec, responsável por encontrar palavras similares. Ainda, foram propostos dois algoritmos baseados em regras linguísticas: um de separação silábica e um de para determinar a sílaba tônica.

Palavras-chave: letras de música, ferramenta, rimas, composição, ferramenta de suporte criativo.

ABSTRACT

There are a considerable amount of materials and tools available that can potentially contribute to the process of creating lyrics. However, this content is distributed in different software, websites, books, etc. Specific software for creating lyrics is rare. The vast majority of tools on the market are intended for the English language. This work is intended to create a web tool to gather some features that can stimulate and accelerate the process of creating lyrics in Portuguese. The main features of the tool are: syllable separation / syllable counter, suggestion of related words, suggestion of words with the same classification as the stressed syllable. All suggestions are constructed from processing lyrics of the same genre, which can produce better results because they are closer to the same context. The lyrics were extracted with web crawling to form a linguistic *corpus*. From the *corpus*, a model was trained with Word2Vec, responsible for finding similar words. Furthermore, two algorithms based on linguistic rules were proposed: one for syllabic separation and one for determining the stressed syllable.

Keywords: lyrics, tool, rhymes, composing, creativity support tool.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	*Application Programming Interface*
DOM	*Document Object Model*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
RF	Requisitos Funcionais
RNF	Requisitos Não-Funcionais
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured query language*
TCC	Trabalho de Conclusão de Curso
TDD	*Test-driven development*
TTS	*Text-to-Speech*
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*
XPATH	*XML Path Language*

SUMÁRIO

1 INTRODUÇÃO

À luz de a *História da Computação* (WAZLAWICK, 2016), observa-se que a computação evoluiu, na maioria das vezes, norteada pela busca da redução de esforço, tanto físico como intelectual, dos seres humanos.

Hoje, a computação está cada vez mais presente na vida das pessoas, capaz de realizar ou acelerar as mais diversas tarefas, das mais corriqueiras até as mais complexas. É nesse contexto, de que a computação pode tornar a vida das pessoas mais produtivas, que é proposto conceber uma ferramenta para auxiliar, seja na forma, seja no conteúdo, a criação de letras musicais na língua portuguesa.

1.1 MOTIVAÇÃO

Este trabalho é inspirado fortemente pelas ideias de Pat Pattinson, professor da *Berklee College of Music*. Ele acredita que não existem regras na esfera musical, mas que é possível estender o conceitos de *Tensão X Resolução*, existentes na música ocidental, às letras de músicas. Segundo ele, todos elementos que compõem uma música, incluindo a forma de como a letra da música é escrita (número de linhas, esquemas de rimas etc.), podem convergir para transmitir uma mesma sensação de estabilidade ou de instabilidade, tornando a canção mais interessante. Para ele, por exemplo, um refrão com um número ímpar de linhas tende à soar mais instável (ou inacabado) do que um refrão com um número par de linhas.(PATTISON, 2009)

Em (PATTISON, 2014) percebe-se que é bastante comum os compositores utilizarem dicionários de rimas e *thesaurus*[1] para auxiliar no processo de criação de letras de música. Esses dicionários, por muitas vezes, encontram-se de forma impressa. Ele encoraja o uso dessas ferramentas para estimular o processo criativo e evitar possíveis "bloqueios" na hora de compor.

Este trabalho propõe sintetizar algumas dessas ideias em uma só ferramenta para auxiliar no processo de criação de uma letra de música.

1.2 JUSTIFICATIVA

Há uma quantidade considerável de materiais e ferramentas disponíveis que potencialmente podem contribuir no processo de criação de letras musicais. Porém, esse conteúdo encontra-se distribuído em diferentes softwares, sites, livros etc. São raros os softwares específicos para criar letras de músicas. Os softwares mais semelhantes ao proposto por este trabalho são exclusivamente para letras em inglês e não são disponibilizados de forma livre.

[1] No Brasil, o termo é conhecido como dicionário analógico ou dicionário de ideias afins.

A realização deste trabalho depende de um esforço interdisciplinar. Disciplinas do curso de Ciência da Computação que contemplaram, em algum nível, conceitos como os de Engenharia de Software; Projetos de Sistemas; Programação; Banco de Dados; Programação Concorrente; Redes; *Datamining* e Inteligência Artificial foram importantes, ou até mesmo indispensáveis, para a construção deste Trabalho de Conclusão de Curso (TCC).

A título de exemplo, temos a tese de doutorado defendida na UFSC denominada *Escansão automática de versos em português* (MITTMANN, 2016). O tema é correlato à separação silábica (uma das funcionalidades propostas no produto deste trabalho). A existência da tese poderia sinalizar a relevância científica deste trabalho.

É esperado que a ferramenta proposta possa auxiliar na compreensão da estrutura da língua portuguesa, na composição de letras musicais, na educação musical, e que esse trabalho possa, de alguma forma, servir de inspiração para o desenvolvimento de outras ferramentas voltadas para a música e para a língua portuguesa.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Desenvolver um software para auxiliar na criação de letras musicais em língua portuguesa.

1.3.2 Objetivos Específicos

- Coletar uma grande quantidade de letras de músicas em português, de um mesmo gênero musical, disponíveis na internet para criação de um *corpus* linguístico;
- Encontrar uma técnica para identificar, de forma automática, palavras relacionadas entre si a partir do *corpus*;
- Encontrar uma técnica para realizar a separação silábica das palavras e identificar a sílaba tônica em cada uma delas;
- Organizar e estruturar as informações obtidas;
- Criar uma aplicação web (cliente/servidor) para tornar possível ao usuário final criar letras de músicas interagindo com as informações descritas nos itens supracitados;
- Analisar e discutir os resultados obtidos na construção da ferramenta proposta por este trabalho;
- Disponibilizar todo o código-fonte produzido de forma livre.

1.4 METODOLOGIA

A metodologia de pesquisa e desenvolvimento do trabalho dividiu-se em quatro principais etapas:

- **Etapa 1:** Levantamento do que já existe disponível em nível de software relacionado à ferramenta produzida neste trabalho. Procedeu-se com pesquisa em motores de busca como o da *Google*[2] e em lojas de aplicativos móveis *Play Store*[3] para *Android*, e *App Store*[4] para *iOS*;
- **Etapa 2:** Levantamento bibliográfico para dar suporte à solução desenvolvida durante a execução deste projeto. A pesquisa foi feita através de artigos, documentações e publicações relacionadas ao assunto. Utilizou-se principalmente o *Google Scholar*[5] e o *CiteSeerX*[6];
- **Etapa 3:** Análise de requisitos e desenvolvimento da ferramenta proposta por este trabalho.
- **Etapa 4:** Análise e discussão dos resultados obtidos na construção da ferramenta proposta por este trabalho.

1.5 ORGANIZAÇÃO DO TRABALHO

Capítulo 1 é um capítulo introdutório que busca apresentar o que é trabalho, esclarece o porquê de realizá-lo, quais são seus objetivos e apresenta a metodologia utilizada de pesquisa para a realização dele. Este primeiro apresenta a motivação, a justificativa, os objetivos e a metodologia deste TCC.

Capítulo 2 é um capítulo em que são apresentados os conceitos fundamentais para o entendimento e desenvolvimento da ferramenta proposta. Também traz uma análise de alguns trabalhos selecionados relacionados ao produto deste projeto

Capítulo 3 Apresenta o desenvolvimento da ferramenta proposta neste trabalho.

Capítulo 4 Apresenta as considerações finais sobre o trabalho e algumas sugestões para a continuidade do projeto.

[2] https://www.google.com
[3] https://play.google.com
[4] https://www.apple.com/ios/app-store/
[5] https://scholar.google.com
[6] https://citeseerx.ist.psu.edu

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo explora conceitos fundamentais que fornecem a base teórica essencial para uma melhor compreensão deste trabalho. Aqui, são detalhados conceitos relacionados ao desenvolvimento de software e de produto, além de algumas tecnologias utilizadas ao longo do trabalho.

2.1 FUNDAMENTOS DE FONÉTICA

Nesta seção são descritos alguns fundamentos da língua portuguesa que contribuíram para o desenvolvimento dos algoritmos de separação silábica e de determinação da sílaba tônica.

2.1.1 Vogal, Consoante e Sílaba

De acordo com (FERNANDES, 2023), vogal, semivogal e consoante são fonemas, os quais são unidades de som que distinguem as palavras. As vogais saem livremente pela boca, enquanto que o mesmo não ocorre com as consoantes. A vogal é o fonema mais importante, pois não existem sílabas sem vogais. Na língua portuguesa existem 5 letras vogais (a, e, i, o, u).

Sílaba é um fonema ou grupo de fonemas emitido num só impulso expiratório. Em português, o elemento essencial da sílaba é a vogal (BECHARA, 2012).

2.1.2 Sílaba tônica

Denomina-se sílaba tônica a sílaba pronunciada com maior força e intensidade numa palavra. Exemplos incluem: a-mi-**ZA**-de, ve-**RÃO** e **MÚ**-si-ca (NEVES, 2017) As demais sílabas, que não são tônicas, são denominadas sílabas **átonas**.

2.1.2.1 Classificação

As palavras podem ser classificadas quanto à sua tonicidade, ou seja, de acordo com a posição em que a sílaba tônica é encontrada. As palavras podem ser classificadas em:

1. **Oxítonas**: São ditas oxítonas as palavras em que a última sílaba é tônica. Exemplos incluem: ar-**ROZ**, ve-**RÃO** e tro-**FÉU**.
2. **Paroxítonas**: São ditas paroxítonas as palavras em que a penúltima sílaba da palavra é tônica. Como ocorre em: me-**NI**-no, perdido: per-**DI**-do e **VÍ**-rus.
3. **Proparoxítonas**: São ditas proparoxítonas (ou esdrúxulas) as palavras em que a antepenúltima sílaba da palavra é tônica. Exemplos incluem: **MÚ**-si-ca, **GRÁ**-fi-co e **DÚ**-vi-da.

2.1.3 Semivogais e Encontros Vocálicos

Os fonemas **i** e **u** são classificados como **semivogais** quando se juntam a uma vogal e são pronunciados com menos força. Os encontros vocálicos dão origem aos ditongos, tritongos e hiatos (BECHARA, 2012).

2.1.3.1 Ditongo

Segundo (BECHARA, 2012), ditongo é o encontro entre uma vogal e uma semivogal, ou vice-versa, na mesma silaba. Os ditongos podem ser:

1. **Crescentes**: quando a semivogal vem antes da vogal (SV + V). Exemplos: ág**ua**, cár**ie**, mág**oa**.

2. **Decrescentes**: quando vogal vem antes da semivogal (V + SV). Exemplos: p**ai**, r**ei**, m**ãe**, m**ão**, p**õe**. Esses últimos demonstram que os fonemas **e** e **o** tornam-se semivogais quando a vogal que os antecede é nasal.

2.1.3.2 Tritongo

De acordo com (BECHARA, 2012), tritongo é o encontro de uma vogal entre duas semivogais numa mesma sílaba (SV + V + SV). Os tritongos podem ser orais e nasais. Exemplos: q**uai**s, parag**uai**o, enxag**uei**, sag**uão**, sag**uões**.

2.1.3.3 Hiato

Segundo (BECHARA, 2012), hiato é o encontro de duas vogais em sílabas diferentes. Exemplos: s**a-í**-da, c**a-a**-ti-ga, mo-**i**-nho. Isso acontece porque passagem da primeira para a segunda se faz mediante um movimento brusco, com interrupção da voz. Em português, como em muitas outras línguas, nota-se uma tendência para evitar o hiato, através da ditongação ou da crase.

2.1.4 Divisão Silábica

De acordo com (ARAÚJO, 2023), a divisão silábica é feita de acordo com a quantidade de fonemas pronunciados em uma única emissão sonora.

1. Os dígrafos **ch**, **lh**, **nh**, **gu** e **qu** devem pertencer a uma única sílaba. Exemplos: a-**ch**a-do, **lh**a-ma, ra-i-**nh**a.

2. Os dígrafos **rr**, **ss**, **sc**, **sç**, **xs** e **xc** devem ser separados sílabas distintas. Exemplos: te**r-r**a, a**s-s**ar, na**s-c**eu, e**x-c**e-ção.

3. Ditongos e Tritongos devem pertencer a uma única sílaba. Exemplos: p**ai**, l**ou**-sa, pa-rag**uai**, u-ru-g**uai**-a.

4. Hiatos devem ser separados em sílabas distintas. Exemplos: s**a-a**-ra, p**a-í**s, d**i-a**.

5. Encontros consonantais devem ser separados em sílabas distintas, exceto aqueles cuja segunda consoante é **l** ou **r**. Exemplos: ca**s-p**a, a**c-n**e, **bl**u-sa, **cr**a-que.

6. Encontros consonantais que iniciam palavras são mantidos juntos na divisão silábica. Exemplos: **pn**eu-má-tico, **ps**i-có-lo-ga, **gn**o-mo.

2.1.5 Regras de Acentuação

A seguir, são apresentadas as regras de acentuação conforme descritas por (SENADO FEDERAL, 2023):

2.1.5.1 Monossílabos tônicos

Acentuam-se os vocábulos monossílabos tônicos terminados em **a/as**, **e/es**, **o/os**, como por exemplo: dá, pás, mês, só, pós, fé, lê, trás.

2.1.5.2 Oxítonas

Considera-se as seguinte regras para a acentuação das palavras oxítonas:

1. Acentua-se as que terminam em **a/as**, **e/es**, **o/os**, **em/ens**. Exemplos: carajás, café, invés, parabéns, porém, refém.

2. Acentua-se com acento agudo as que terminam com os ditongos tônicos abertos **éi**, **éu**, **ói**. Exemplos: papéis, chapéu, Ilhéus, rouxinóis, herói.

3. Acentua-se com acento agudo vogais tônicas **i(s)** e **u(s)** quando estiverem depois de um ditongo. Exemplos: tuiuiú, teiús, Piauí.

2.1.5.3 Paroxítonas

As palavras paroxítonas terminadas em **i/is**, **us**, **r**, **l**, **x**, **n**,**um/uns**, **ão/ãos**, **ã/ãs**, **ps**, **on/ons**. Exemplos incluem: júri/júris, vírus, caráter, têxtil, tórax, hífen (hifens não tem acento), fórum/fóruns, órgão/órgãos, ímã/ímãs, bíceps, próton/prótons.

2.1.5.4 Proparoxítonas

Todas as palavras proparoxítonas são acentuadas. Exemplos incluem: **lú**cido, **sé**culo, **pró**ximo, **câ**mara.

Há uma certa discussão de palavras chamadas de proparoxítonas aparentes que terminam em -ea, -eo, -ia, -ie, -io, -oa, -ua, e -uo. São palavras como “série”, “nódoa” e “tênue”. Antes do acordo ortográfico essas palavras eram ditas como paroxítonas e proparoxítonas, mas, com a adesão do novo acordo, foram aceitas como proparoxítonas.

No entanto, alguns gramáticos ainda consideram essas palavras paroxítonas terminadas em ditongos crescentes.

2.2 LETRAS DE MÚSICAS

Nesta seção, são introduzidos alguns conceitos fundamentais relacionados à escrita de letras de músicas.

2.2.1 Verso, Estrofe, Poesia e Canção

O **verso** tem origem da palavra em latim *versus*, que significa "linha de escrita". É o nome dado as linhas que compõe uma poesia. **Estrofe** é o agrupamento de versos. A **poesia** é formada por uma quantidade de versos, que apresentam ou não rimas, os quais são agrupados em estrofes. Por fim, pode-se dizer que **canção** é a união de uma poesia com uma música. (DIANA, 2015).

2.2.2 Letra de Música

Segundo (DAVID, 2010), letra de música é o conjunto de palavras que compõem a parte vocal de uma canção. Ela serve como meio para expressar emoções, contar histórias, transmitir mensagens e ideias, e é essencial para a identidade de uma canção. As letras podem variar em estilo, tema e complexidade, dependendo do gênero musical, das intenções do artista e do público-alvo.

2.2.3 Conceito Clássico de Rima

Segundo (DIANA, 2015), rima é um recurso literário que designa a aproximação sonora entre duas palavras que compõem o verso, oferecendo mais musicalidade à poesia. Os versos que não apresentam rima, são chamados de versos brancos. De acordo com o tipo de rima utilizada elas são classificadas em:

- **Alternadas**: se formam entre versos pares e os versos ímpares.
- **Opostas**: surgem entre o primeiro e o quarto verso e, entre o segundo e o terceiro verso.
- **Emparelhadas**: ocorrem entre o primeiro e o segundo verso e, entre o terceiro e o quarto verso.
- **Internas**: surgem no interior dos versos.

Segundo (BECHARA, 2012), se rimam vocábulos só com identidade das vogais tônicas. As rimas também podem ser classificadas em:

- **Consoantes**: A rima se diz consoante quando é perfeita, isto é, tem os mesmos fonemas a partir da última vogal tônica do verso. Por exemplo: vaga-lume/ciúme.
- **Assonantes (ou toante)**: é a rima imperfeita, em que há apenas identidade nas vogais tônicas. Por exemplo: calma/cada; terra/pedra.

2.2.4 Expansão do Conceito de Rima

A canção "Construção" de Chico Buarque é um exemplo notável da habilidade literária e musical do compositor brasileiro. Uma característica marcante da letra é o uso intenso de palavras proparoxítonas no final dos versos exemplificada na estrofe a seguir:

> Sentou pra descansar como se fosse **sá**bado
> Comeu feijão com arroz como se fosse um **prín**cipe
> Bebeu e soluçou como se fosse um **náu**frago
> Dançou e gargalhou como se ouvisse **mú**sica

(BUARQUE, 1971)

A rigor, as únicas rimas dessa estrofe estão no par (**sá**bado/**náu**frago). No entanto, ainda é possível identificar certa aproximação sonora/rítmica entre todas essas palavras do final de cada linha por terem a mesma classificação de tonicidade e o mesmo número de sílabas, mesmo nos casos em que as vogais tônicas não coincidam. Este trabalho considera que seria possível, por exemplo, substituir a palavra "**náu**frago" por "**bê**bado" sem afetar a musicalidade da estrofe como um todo, ainda que, em teoria, a estrofe perdesse todas as rimas.

Ainda, durante o andamento da canção "Construção", percebe-se que o autor realiza algumas substituições de alguns versos/palavras, como por exemplo, em:

> Morreu na contramão atrapalhando o **trá**fego
> ...
> Morreu na contramão atrapalhando o **pú**blico
> ...
> Morreu na contramão atrapalhando o **sá**bado

Para exemplificar a ideia de que seria possível intercambiar as palavras/versos, desconsiderando a intenção/semântica, poderia-se substituir os versos acima, sem prejuízo à musicalidade, por exemplo, por qualquer um desses:

> Morreu na contramão atrapalhando a **ví**tima
> Morreu na contramão atrapalhando o **bê**bado
> Morreu na contramão atrapalhando a **grá**vida
> Morreu na contramão atrapalhando o **sín**dico

Morreu na contramão, presenciou o ro**mân**tico

Este trabalho despretensiosamente estende o conceito clássico de rima apenas para contemplar a proximidade entre palavras que compartilham a mesma classificação de tonicidade, como se elas fizessem parte de uma categoria de "rima extremamente assonante".

2.2.5 Prosódia

É a arte de usar a entonação, o ritmo e a ênfase na fala ou na escrita, especialmente em poesia e na composição de letras de música. A prosódia é fundamental para garantir que a estrutura lírica da música esteja em harmonia com a sua melodia e ritmo. (PATTISON, 2009) destaca a importância de alinhar as palavras com a música de maneira que reforce a emoção e o significado pretendidos. Ele argumenta que a prosódia eficaz pode intensificar a experiência auditiva ao sincronizar a expressão lírica com os elementos musicais.

2.2.6 Importância da separação silábica

A separação silábica em composição musical é crucial para a **metrificação**, que é a arte de criar ritmo em poesia ou letras de músicas através da organização de sílabas e acentos. A metrificação depende da correta separação silábica para estabelecer o padrão rítmico de uma linha ou estrofe. Cada sílaba pode ser acentuada ou não, e a disposição dessas sílabas acentuadas define o metro da música, contribuindo para a sua musicalidade e fluidez. A habilidade de manipular a separação silábica permite ao compositor controlar o fluxo e o ritmo da letra, garantindo que ela se adapte harmoniosamente à melodia (SCOTT, 2006).

2.3 ASPECTOS TECNOLÓGICOS

Nesta seção são descritos conceitos, técnicas e ferramentas que foram utilizados durante o desenvolvimento deste trabalho.

2.3.1 Golang

Golang, conhecida como Go, é uma linguagem de programação de código aberto desenvolvida pela Google. É conhecida por sua simplicidade, eficiência e desempenho. Go é projetada para ser rápida, fácil de aprender e ideal para desenvolvimento de sistemas de rede e multicore. Sua sintaxe concisa e recursos modernos, como *garbage collection* (coleta de lixo) e suporte a concorrência, a tornam popular para desenvolvimento de aplicações *cloud*, servidores e ferramentas de linha de comando (DONOVAN; KERNIGHAN, 2015).

2.3.2 TypeScript

TypeScript é uma linguagem de programação desenvolvida pela Microsoft que adiciona tipagem estática opcional ao JavaScript, o que ajuda no desenvolvimento de aplicações grandes e complexas. Typescript é amplamente utilizado no desenvolvimento de aplicações *front-end* e *back-end*.

2.3.3 Python

Python é uma linguagem de programação de alto nível, interpretada, de propósito geral, conhecida por sua legibilidade e simplicidade. É amplamente utilizada para desenvolvimento web, análise de dados, inteligência artificial e computação científica, devido à sua vasta gama de bibliotecas e *frameworks* (LUTZ, 2013).

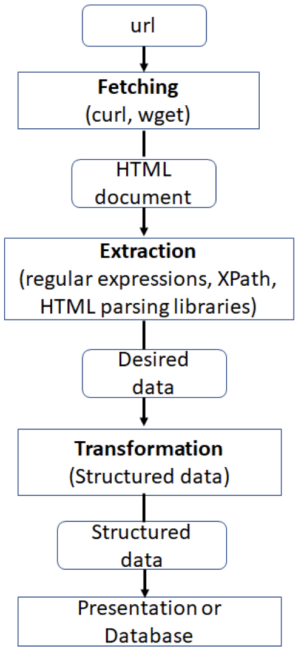
2.3.4 *WEB SCRAPING*

Segundo Khder (2021), *web scraping*, ou *web crawling*, refere-se à técnica de extração automatizada de dados de sites por meio do uso de software especializado. Este procedimento possui uma relevância particular nos dias atuais, especialmente nos campos de *Business Intelligence*, ao possibilitar a coleta de informações estruturadas a partir de documentos em formato de texto, como o *Hypertext Markup Language* (HTML). Seu valor é notável quando se trata de situações em que os dados não são disponibilizados em formatos de fácil processamento por máquinas, como *JavaScript Object Notation* (JSON) ou *Extensible Markup Language* (XML). A técnica permite coletar informações em tempo real como, por exemplo, preços de produtos em lojas virtuais, e até mesmo detalhes sobre atividades ilícitas online, fornecendo dados de grande importância tanto para autoridades quanto para pesquisadores. Comparado à entrada manual de dados, o *web scraping* se destaca pela sua capacidade de coletar informações abrangentes e precisas. O processo de *web scraping* é dividido em três etapas:

1. **Etapa de busca (*fetching*)**: Primeiramente, é necessário acessar o site desejado que contém as informações relevantes, o que é conhecido como fase de busca. Isso é feito por meio do protocolo *Hypertext Transfer Protocol* (HTTP), um protocolo da internet usado para enviar e receber solicitações de servidores web. Navegadores web utilizam métodos semelhantes para obter conteúdo de páginas da web. Nesta etapa, bibliotecas como curl e wget podem ser utilizadas para enviar uma solicitação HTTP GET para um endereço na web, também conhecido como *Uniform Resource Locator* (URL), e obter a página HTML como resposta.

2. **Etapa de extração (*extraction*)**: Após recuperar a página HTML, é necessário coletar os dados importantes. Para isso, são utilizadas expressões regulares, bibliotecas de análise HTML e consultas *XML Path Language* (XPATH), sendo essa etapa

Figura 1 – Etapas do processo de *web scraping*



Fonte: Khder (2021).

chamada de extração. A XPATH é uma ferramenta que auxilia na localização de informações em documentos.

3. **Etapa de transformação (*transformation*)**: Após os dados serem extraídos, é possível convertê-los em um formato estruturado para apresentação ou armazenamento. Com os dados armazenados, é possível obter informações que podem auxiliar em tomadas de decisões em *Business Intelligence* e em outras áreas.

2.3.5 Colly

Colly é um *framework* em Golang para a construção de *web crawlers*. Com ele, é possível criar desde um web *crawler* simples até web crawlers assíncronos complexos que processam milhões de páginas web. Além de oferecer uma *Application Programming Interface* (API) para realizar solicitações de rede, permite o tratamento do conteúdo recebido (GOCOLLY, 2023). Na Figura 2, temos um exemplo prático do uso do *framework*, demonstrando como localizar e acessar todos os links de uma página web específica.

2.3.6 Expressões regulares

Expressões regulares, ou *regex*, pode ser descrito como uma linguagem de padrões de texto compacta e altamente especializada, usada para identificar cadeias de caracteres de interesse. As expressões regulares são amplamente utilizadas em programação para

Figura 2 – Exemplo de uso do Colly

```go
func main() {
    c := colly.NewCollector()

    // Find and visit all links
    c.OnHTML("a", func(e *colly.HTMLElement) {
        e.Request.Visit(e.Attr("href"))
    })

    c.OnRequest(func(r *colly.Request) {
        fmt.Println("Visiting", r.URL)
    })

    c.Visit("http://go-colly.org/")
}
```

Fonte: GOCOLLY(2023).

tarefas como validação de dados, análise sintática e transformação de texto. A capacidade das expressões regulares de descrever e manipular estruturas de texto complexas as torna indispensáveis em muitos contextos de processamento de dados e desenvolvimento de software (FRIEDL, 2006)

2.3.7 SQLITE

SQLite é um Sistema de Gerenciamento de Banco de Dados (SGBD) relacional que se destaca pela sua leveza e eficiência, operando sem a necessidade de um servidor separado, diferentemente de sistemas como MySQL ou PostgreSQL. Sua arquitetura integrada permite leitura e escrita direta em arquivos de disco, tornando-o uma opção ideal para aplicações móveis, dispositivos embarcados e situações que requerem um banco de dados local sem a complexidade de um servidor completo. Além disso, é notável por seu tamanho reduzido e pela facilidade de uso, sendo uma escolha frequente em desenvolvimento e testes devido à sua natureza autocontida e poucas dependências externas. Destaca-se também por assegurar as propriedades Atomicidade, Consistência, Isolamento e Durabilidade (ACID), essenciais para a integridade e confiabilidade em transações de banco de dados. A atomicidade é garantida por meio de um sistema de *rollback*, que reverte todas as alterações em caso de falha em qualquer parte da transação. A consistência é mantida através das regras de integridade, assegurando que cada transação mova o banco de dados de um estado consistente para outro. Embora o SQLite ofereça diferentes níveis de isolamento de transações, ele pode não ser tão eficiente quanto outros sistemas em ambientes de alta concorrência. Por fim, a durabilidade é assegurada, pois uma vez que

uma transação é confirmada, todas as alterações são permanentemente gravadas no banco de dados, resistindo a falhas de sistema ou energia. Essas características fazem do SQLite uma solução confiável e eficiente para muitos cenários de aplicação (SQLITE, 2023).

2.3.8 Aplicação Web

Uma aplicação web é um software que é acessado através de um navegador web. Essas aplicações são hospedadas em servidores e acessadas via internet, utilizando tecnologias como HTML, CSS e JavaScript. Elas oferecem vantagens como acessibilidade em diversos dispositivos, facilidade de manutenção e atualização, e variam desde páginas estáticas simples até sistemas complexos e interativos (LAUDON; LAUDON, 2016).

2.3.9 API

Uma API, ou Interface de Programação de Aplicações, é um conjunto de definições e protocolos para construir e integrar software de aplicativos. Elas permitem que um produto ou serviço comunique-se com outros produtos e serviços sem a necessidade de conhecer a implementação desses sistemas. Isso simplifica o desenvolvimento de aplicativos, economizando tempo e dinheiro.

As APIs são frequentemente vistas como contratos, com documentação que representa um acordo entre as partes envolvidas: se uma parte envia uma solicitação remota estruturada de uma maneira específica, o software da outra parte responderá de acordo. Além disso, as APIs facilitam a integração de novos componentes de aplicativos em uma arquitetura existente, ajudando equipes de negócios e TI a colaborarem (REDHAT, 2023).

2.3.10 POSTMAN

Postman é uma plataforma popular para o desenvolvimento de APIs que oferece ferramentas para construir, testar e modificar APIs. Ela permite aos desenvolvedores enviar requisições HTTP para *endpoints* de API, receber respostas e examinar os resultados. A plataforma suporta várias tecnologias de API, incluindo REST, SOAP, GraphQL, entre outras, e é uma ferramenta essencial para desenvolvedores modernos de web e aplicativos móveis (POSTMAN, 2023)

2.3.11 TDD

Test-driven development (TDD) é uma metodologia de desenvolvimento de software que inverte a ordem tradicional de desenvolvimento. O processo iterativo começa com o desenvolvedor escrevendo um caso de teste automatizado que define uma melhoria desejada ou uma nova função. Em seguida, produz-se o código mínimo necessário para passar no teste e, por fim, realiza-se a refatoração do código para atender aos padrões de

qualidade. Este ciclo é repetido frequentemente, promovendo um desenvolvimento mais ágil e adaptável(JANZEN; SAIEDIAN, 2005).

2.3.12 FIBER

O Fiber é um *framework web* para a linguagem de programação Go, conhecido por sua simplicidade, eficiência e rapidez. Inspirado no popular *framework* JavaScript Express.js, o Fiber oferece uma maneira fácil e eficiente de desenvolver aplicações web e APIs em Go (FIBER, 2023).

2.3.13 SQLC

O SQLC é uma ferramenta que gera código Go a partir de consultas em *Structured query language* (SQL). Assim, a aplicação ganha em segurança de tipos e clareza, reduzindo erros comuns de SQL e permitindo que os desenvolvedores se concentrem mais na lógica de negócios do que nas preocupações com a camada de acesso a dados (SQLC, 2023).

2.3.14 SVELTE

Svelte é um *framework* moderno de JavaScript para a construção de interfaces de usuário. Ao contrário de outros *frameworks* populares como React ou Angular, o Svelte introduz uma abordagem diferente no processamento de componentes de interface de usuário. Em vez de utilizar um virtual *Document Object Model* (DOM) para atualizações de interface, o Svelte compila os componentes em código JavaScript otimizado durante a fase de build, resultando em atualizações diretas ao DOM real quando o estado dos componentes muda. Esta característica de compilação do Svelte oferece algumas vantagens, como uma redução no tamanho do código final e uma potencial melhoria no desempenho, já que há menos abstrações entre o código que os desenvolvedores escrevem e o que é executado no navegador (SVELTE, 2023).

2.3.15 *Levenshtein*

A distância de *Levenshtein*, também conhecida como distância de edição, é uma métrica para medir a diferença entre duas *strings*. A distância de *Levenshtein* entre duas palavras é o número mínimo de operações de edição (inserções, deleções ou substituições de um único caractere) necessárias para transformar uma palavra na outra (LEVENSHTEIN, 1966). É uma métrica é amplamente utilizada em aplicações de processamento de linguagem natural, como corretores ortográficos, comparação de textos e bioinformática.

2.3.16 *Embeddings*

Embeddings são representações vetoriais em um espaço n-dimensional que traduzem dados de alta dimensionalidade, preservando suas características essenciais. Esses

embeddings são amplamente empregados em métodos de aprendizado de máquina que processam entradas complexas, como vetores esparsos usados na representação de palavras. O objetivo principal de um *embedding* é refletir a semântica dos dados que representa, agrupando informações semelhantes próximas umas das outras dentro do mesmo espaço vetorial. É pretendido neste trabalho utilizar técnicas de criação de embeddings para construir automaticamente uma espécie de *thesaurus*. *Thesaurus* (ou dicionário de ideias afins, ou dicionário analógico) é uma ferramenta de referência que lista palavras agrupadas por similaridade de significado.

2.3.16.1 Word2Vec

O algoritmo Word2Vec é uma técnica importante para representar palavras como vetores em um espaço numérico, capturando seus significados semânticos. Isso é alcançado por meio de uma rede neural que treina esses vetores com base em palavras próximas em um texto. O Word2Vec é capaz de aprender relações semânticas e sintáticas entre palavras e é amplamente utilizado em aplicações de processamento de linguagem natural. Devido à sua capacidade de representar palavras em um espaço vetorial, o Word2Vec facilita o agrupamento de palavras com significados ou contextos semelhantes (DHARMA *et al.*, 2022).

A biblioteca Gensim para a linguagem Python oferece uma interface fácil de usar para treinar modelos Word2Vec com várias opções de personalização. Um exemplo mínimo de como treinar um modelo com Word2Vec com a Gensim:

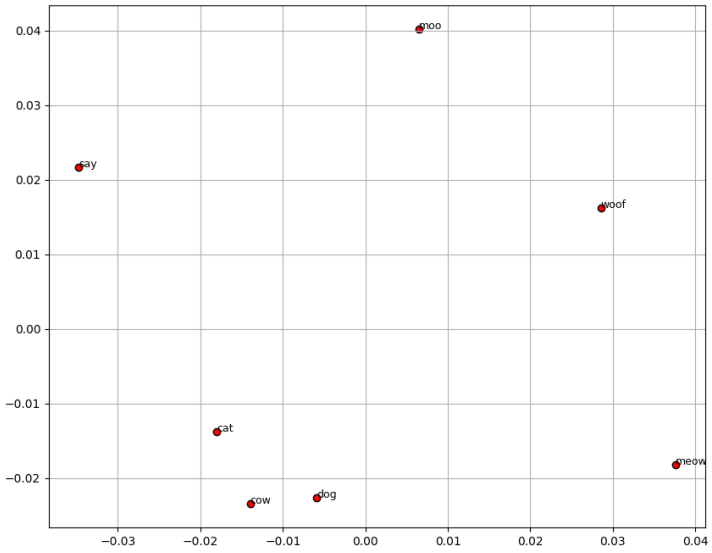
```
from gensim.models import Word2Vec
sentences = [
        ['cat', 'say', 'meow'],
        ['dog', 'say', 'woof'],
        ['cow', 'say', 'moo'],
];
model = Word2Vec(sentences, vector_size=100, window=5, min_count
   =1, workers=4)
```

A Análise de Componentes Principais (PCA, do inglês *Principal Component Analysis*) é uma técnica estatística de redução de dimensionalidade. É utilizada para simplificar a complexidade de espaços dimensionais altos, preservando o máximo de informação possível. Assim, é possível, por exemplo, visualizar a disposição das representações palavras em 2D como na Figura 3.

2.3.16.2 GloVe

GloVe (*Global Vectors*) é uma técnica de processamento de linguagem natural que combina abordagens de contagem e previsão direta para mapear palavras em vetores semânticos. Enquanto o Word2Vec foca em informações locais, o GloVe também considera

Figura 3 – Visualização 2D das Palavras com Word2Vec



Fonte: Elaboração Própria.

informações globais de co-ocorrência de palavras. Ele usa uma técnica de fatorização global de matrizes e é chamado de modelo log-bilinear, ao passo que o Word2Vec é um modelo de rede neural (DHARMA *et al.*, 2022). Geralmente, o GloVe é treinado usando um pacote específico e não diretamente em Python. No entanto, é possível carregar modelos pré-treinados usando o Gensim.

2.3.16.3 FastText

De acordo com (DHARMA *et al.*, 2022), FastText é uma técnica de incorporação de palavras desenvolvida pela equipe de pesquisa de IA do Facebook. Ela se destaca por sua velocidade e eficiência ao criar representações de palavras. O FastText estende o Word2Vec para considerar n-gramas de caracteres, o que permite que ele trate palavras que estejam fora do vocabulário e capte mais informações morfológicas.

Assim com o Word2Vec, modelos do FastText podem ser treinados em Python através da biblioteca fastText. Um exemplo mínimo para realizar o treinamento é mostrado a seguir:

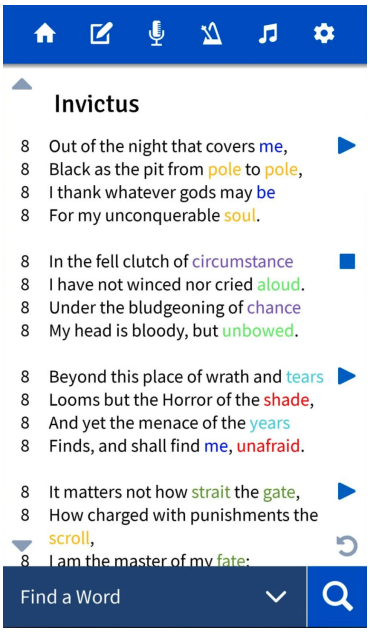
```
import fasttext
model = fasttext.train_unsupervised('sentences.txt', model='
    skipgram')
```

3 TRABALHOS RELACIONADOS

Apresenta-se aqui algumas ferramentas que serviram de referência para este trabalho.

3.1 LYRIC NOTEPAD

Figura 4 – Lyric Notepad



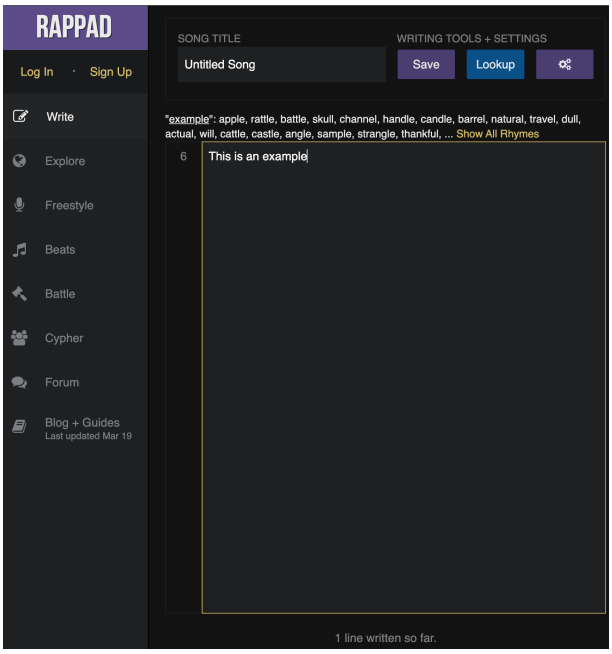
Fonte: Elaboração própria.

O Lyric Notepad (KENNY MC, 2023) é aparentemente o software mais similar da ferramenta proposta por esse trabalho. É um aplicativo móvel disponível nas plataformas *android* e *iOS* que tem um contador de sílabas; detecta as palavras que rimam entre si e as mostram na mesma cor; permite encontrar rimas e sinônimos para uma determinada palavra. A ferramenta conta ainda com: gravador, metrônomo e um bloco de notas. Disponível somente para a língua inglesa.

3.2 RAPPAD EDITOR

O RapPad (RAPPAD, 2023) é uma rede social voltada ao gênero *Rap*. Ela tem um editor on-line que permite o usuário escreva suas letras enquanto rimas vão sendo sugeridas para a última palavra digitada conforme a Figura 5. Essas funcionalidades também estão disponíveis somente em inglês.

Figura 5 – RapPad Editor



Fonte: Elaboração Própria.

Figura 6 – Dicionário de divisão silábica

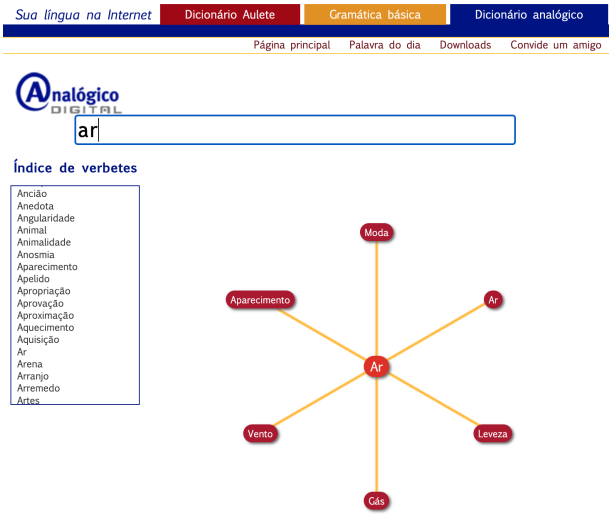
Fonte: Elaboração Própria.

3.3 PORTAL DA LÍNGUA PORTUGUESA

O Portal da Língua Portuguesa (PLB, 2023) é uma plataforma criada para a promoção, a difusão e a projeção da língua portuguesa. Dentre outros recursos, ela disponibiliza um dicionário que identifica a separação silábica, sílaba tônica como pode ser visto na Figura6. Esse *site* pode ser usado para obtenção de dados importantes para este a realização deste trabalho.

3.4 LEXIKON

Figura 7 – aulete.com.br



Fonte: Elaboração Própria.

O Lexikon (LEXICON, 2023) é um *thesaurus* disponível on-line, que representa, através de um grafo, o relacionamento (arestas) entre palavras (nodos) da língua portuguesa, como pode ser visualizado na Figura 7. Foi o único *thesaurus* encontrado online em língua portuguesa.

3.4.1 Trabalhos acadêmicos

Referente a um desafios técnicos deste projeto, pesquisou-se por palavras-chave como “divisão silábica", “algoritmo divisão silábica”, “*portuguese syllabification*” e “*portuguese syllabification algorithm*” em motores de busca acadêmicos, convencionais e repositórios de código-fonte de software. Os trabalhos encontrados com o maior grau de relevância para possivelmente suportar o desenvolvimento deste projeto foram os seguintes:

1. Separação das Sílabas e Determinação da Tonicidade no Português Brasileiro (SILVA; BRAGA; RESENDE, 2008);
2. *A Rule-based Syllabification Algorithm with Stress Determination for Brazilian Portuguese Natural Language Processing* (MONTE *et al.*, 2011);
3. *The Implementation of a Free Syllabification Algorithm Based on Linguistic Rules for Brazilian Portuguese* (ROCHA; NETO, 2013);
4. Escansão Automática de Versos em Português (MITTMANN, 2016).

Os autores (SILVA; BRAGA; RESENDE, 2008) desenvolveram dois algoritmos baseados em regras linguísticas para separação silábica e determinação da tonicidade em palavras do português brasileiro. Os algoritmos foram testados com uma amostra da base de dados CETEN-Folha. Os resultados mostraram uma taxa de erro de 0,71% na separação das sílabas e 1,1% na determinação da tonicidade. São descritas vinte regras linguísticas no trabalho, mas o processo de criação e motivação por trás de cada regra linguística não foi completamente descrito.

O trabalho de (MONTE *et al.*, 2011) propõe aprimorar o conjunto existente de regras linguísticas propostas por (SILVA; BRAGA; RESENDE, 2008) para a divisão silábica de palavras em português brasileiro. Os autores ressaltam a importância dos algoritmos para melhorar a eficiência de sistemas *Text-to-Speech* (TTS) e de outros sistemas inteligentes que processam textos em português brasileiro. Destacam também a vantagem do uso do algoritmo, em comparação ao uso de um dicionário, pois o algoritmo evita o uso de um arquivo grande para armazenar todas as palavras de um idioma e é capaz de lidar com palavras novas. O algoritmo atualizado demonstrou um número significativamente menor de erros em comparação com a versão anterior, principalmente no contexto de ditongos e hiatos. A implementação do algoritmo foi realizada na linguagem Java, e é *open-source*.

O trabalho de (ROCHA; NETO, 2013) também propõe melhorias ao conjunto de regras linguísticas do trabalho de (SILVA; BRAGA; RESENDE, 2008). As melhorias visam lidar melhor com casos de encontros vocálicos. O autor relata a escassez de recursos e pesquisas dedicados à divisão silábica do português brasileiro em comparação com outras línguas, especialmente no contexto de sistemas de TTS. O autor explica ainda que a divisão silábica é um pré-requisito importante para esses sistemas, pois influencia a transcrição fonética e a geração de prosódia. O trabalho produziu um separador silábico, *open-source* escrito na linguagem C#. Foram realizados testes de validação com um grande número de palavras. O algoritmo atualizado demonstrou melhor desempenho em comparação os trabalhos de (SILVA; BRAGA; RESENDE, 2008) e (MONTE *et al.*, 2011), especialmente no tratamento de hiatos e ditongos. O autor conclui que, apesar dos avanços, há espaço para mais refinamentos nas regras e na manipulação de estrangeirismos.

O trabalho de (MITTMANN, 2016) é uma tese de doutorado que apresenta o Aoidos, um sistema capaz de escandir versos em português. No que se diz respeito às funcionalidades pretendidas neste trabalho, primeiro, de acordo com um conjunto de regras propostas pelo autor, a posição da vogal tônica (ou a ausência dela) é determinada para cada palavra do verso de um poema. Em seguida, cada grafema é classificado em vogal, consoante ou semivogal, e a partindo da ideia de que a vogal é o núcleo cada sílaba, a divisão silábica da palavra é estabelecida. A taxa média de acerto do sistema foi de 99,0%, variando entre os corpora de 97,5 a 100,0%.

A pesquisa não encontrou nenhum código-fonte disponível em nenhuma das referên-

cias. Para atingir os objetivos propostos por este trabalho, optou-se por seguir na direção de criar dois novos algoritmos: um para separação silábica e outro para a identificação da sílaba tônica. A funcionalidade para buscar rimas depende desses dois algoritmos.

4 DESENVOLVIMENTO

A partir da etapa de pesquisa concluída, iniciou-se uma etapa de levantamento de requisitos.

4.1 LEVANTAMENTO DE REQUISITOS

Os requisitos podem classificados em Requisitos Funcionais (RF) ou Requisitos Não-Funcionais (RNF). Os RF definem funções específicas ou comportamentos do sistema. Eles descrevem o que o sistema deve fazer, as ações que deve executar em determinadas condições, as respostas a entradas específicas e como deve se comportar em determinadas situações. Por outro lado, os RNF são restrições sobre os serviços ou funções oferecidos pelo sistema. Eles incluem restrições de tempo, restrições sobre o processo de desenvolvimento e padrões. os RFs muitas vezes se referem à qualidade do sistema e abrangem áreas como desempenho, segurança e confiabilidade (SOMMERVILLE, 2011).

Diante disso, foram propostos os seguintes requisitos:

- RF01 - Inserção da letra de música pelo usuário.
- RF02 - Seleção de palavras para buscar palavras relacionadas ou rimas.
- RF03 - Exibição de palavras relacionadas ou rimas.
- RF04 - Visualização da letra com separação silábica com destaque da sílaba tônica

- RNF01 - O SGBD utilizado deve ser preferencialmente o SQLite.
- RNF02 - O sistema será desenvolvido especificamente para uso em navegadores de *desktop*, sem necessidade de adaptação para dispositivos móveis.

4.1.1 Diagrama de casos de uso

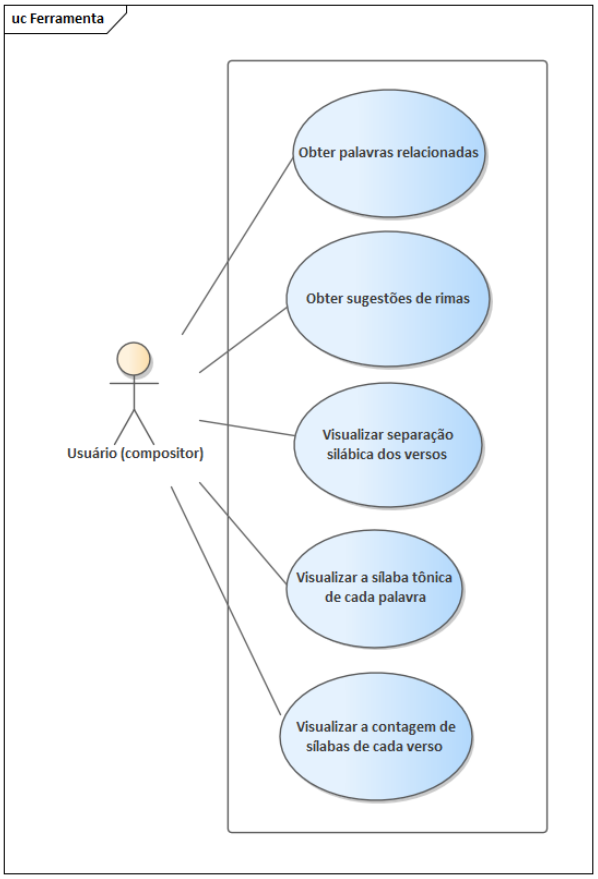
Segundo (SOMMERVILLE, 2011), um diagrama de casos de uso é uma representação gráfica que descreve como os usuários (ou atores) interagem com um sistema, mostrando a relação entre os usuários e as diferentes funções do sistema.

O diagrama dos casos de uso da ferramenta proposta neste trabalho que pode ser visto na Figura 8.

4.1.2 Casos de uso

Os casos de uso são descrições de como os usuários realizam tarefas com o sistema, cada um representando um conjunto de ações executadas pelo sistema para fornecer um

Figura 8 – Diagrama de casos de uso da ferramenta



Fonte: Elaboração Própria.

resultado útil para um ator. Os casos de uso são uma ferramenta valiosa para identificar e comunicar as funcionalidades que o sistema deve fornecer (SOMMERVILLE, 2011).

Os casos de uso da ferramenta proposta neste trabalho estão descritos a seguir:

4.1.2.1 Obter palavras relacionadas

- Entrada: o usuário seleciona uma palavra da letra da música.
- Saída: uma lista de palavras relacionadas baseada na similaridade semântica.

4.1.2.2 Obter sugestões de rimas

- Entrada: o usuário seleciona uma palavra da letra da música.
- Saída: uma lista de sugestões de palavras que tem a mesma classificação de tonicidade da palavra selecionada.

4.1.2.3 Visualizar separação silábica dos versos

- Entrada: o usuário insere um mais versos.
- Saída: a representação da separação silábica de cada verso.

4.1.2.4 Visualizar a sílaba tônica de cada palavra

- Entrada: o usuário insere um mais versos.
- Saída: a representação da sílaba tônica de cada palavra no verso em destaque.

4.1.2.5 Visualizar a contagem de sílabas de cada verso

- Entrada: o usuário insere um mais versos.
- Saída: Contagem total de sílabas para cada verso fornecido.

4.2 SIMILARIDADE ENTRE PALAVRAS

Esta seção dedica-se a documentar todas as atividades que foram realizadas para a construção do modelo de dados que permitiu que fosse criada a funcionalidade de busca por palavras relacionadas.

4.2.1 Extração dos dados

Foi implementado, em GoLang, um processo de *web crawling* com o intuito de coletar um extenso conjunto de letras de música de um site especializado [1]. Foram coletadas apenas letras de um único estilo musical (sertanejo) com a intenção de enriquecer a contextualização de palavras semelhantes. O estilo foi escolhido pela grande quantidade de letras de músicas disponíveis.

Utilizou-se o *framework* Colly para construir o *web crawler*. Ao final desse processo, foram contabilizados cerca de 25MB em texto em letras de músicas. Um único arquivo de texto foi criado, formando assim, um *corpus* de letras de músicas.

4.2.2 Pré-processamento

Essa etapa foi realizada na linguagem Python. Percebeu-se que o *corpus* possuía uma série de problemas, como erros gramaticais, comentários dentro das letras, marcações do tipo "refrão", "repete 2x", como pode ser visto na Figura 9. Então, durante o processo para transformar o texto em conjuntos de palavras (tokenização), verificou-se também se as palavras estavam presentes no dicionário do HunSpell (NÉMETH *et al.*, 2023). No fim desse processo, foram preparados um conjunto de sentenças. Foram consideradas apenas

[1] https://www.vagalume.com.br

Figura 9 – Exemplo de problema no *corpus*

```
REFRÃO: 2x
Eu vou pagar a conta
A conta da tristeza e nisso cai na mesa o seu retrato
Veja amigo este rosto é a razão do meu desgosto
É a moça dessa foto 3x4
Repete tudo
```

Fonte: Elaboração Própria.

sentenças que tinham, pelo menos, três palavras e que todas elas estavam presentes no dicionário.

4.2.3 Treinamento do modelo

Nessa etapa foram conduzidos alguns experimentos preliminares com algoritmos para orientar a seleção de um modelo. Foram realizados experimentos criando modelos com o GloVe, Word2Vec e FastText. O tempo de treinamento desses modelos é na ordem de dez minutos utilizando o *corpus* completo. A vantagem percebida com o FastText é que ele consegue trabalhar com uma palavra que não esteja presente no vocabulário do modelo. No entanto, ele deixa mais próximas as palavras que são morfologicamente semelhantes e essas palavras aparecem com mais prioridade na busca, como é mostrado na Figura 10

Devido à facilidade do treinamento em Python em relação ao GloVe e por gerar uma lista onde os itens similares são priorizados de uma forma mais interessante que o FastText, o Word2Vec foi o algoritmo escolhido para seguir com o treinamento para a obtenção do modelo final capaz de encontrar palavras similares.

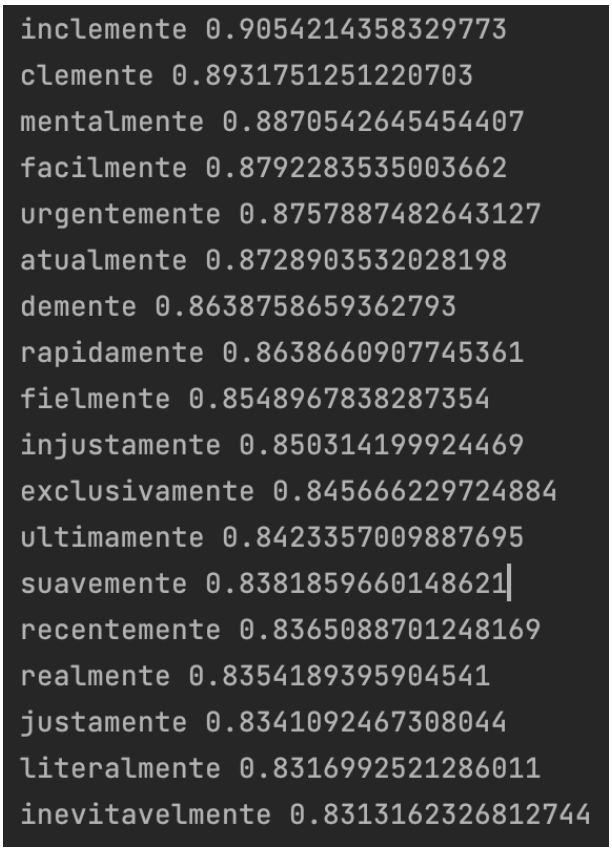
4.2.4 Criação do banco de dados

Tipicamente, o modelo gerado com o Word2Vec só pode ser utilizado em Python. Para eliminar a dependência do modelo, para acelerar a busca de palavras similares e para facilitar o consumo dessas informações em outras linguagens, optou-se por extrair as informações de similaridades entre as palavras do modelo salvá-las em um banco de dados. Criou-se um banco de dados com o *schema* que pode ser visto na Figura 11.

Foram salvas todas as palavras presentes no modelo. Foram criados os relacionamentos para as duzentas e cinquenta mais similares de cada palavra em um banco de dados (SQLite).

O modelo do banco de dados também pode ser representado pela Figura 12 .

Figura 10 – FastText priorizando palavras morfologicamente semelhantes



Fonte: Elaboração Própria.

4.3 ALGORITMOS DESENVOLVIDOS

Esses algoritmos foram construídos a partir da pesquisa realizada e de experimentação manual, busca por padrões em dicionários etc. Eles foram desenvolvidos com TDD em GoLang, sendo contextualizados a seguir.

4.3.1 Algoritmo de divisão silábica

Foram criadas algumas expressões regulares para identificar diferentes tipos de combinações de letras, como encontros consonantais seguidos de vogais, vogais nasais, etc. O algoritmo recebe uma palavra e retorna uma lista de *strings* (sílabas). As principais ideias para a criação do algoritmo foram as seguintes:

1. O *token* de consoante [C] representa um caractere de consoante ou um par de caracteres que o algoritmo assume como inseparáveis. Como, por exemplo, os pares em destaque em: **br**a, **bl**a, **cr**e, **nh**a, **ch**a, **qu**a, **gu**a.

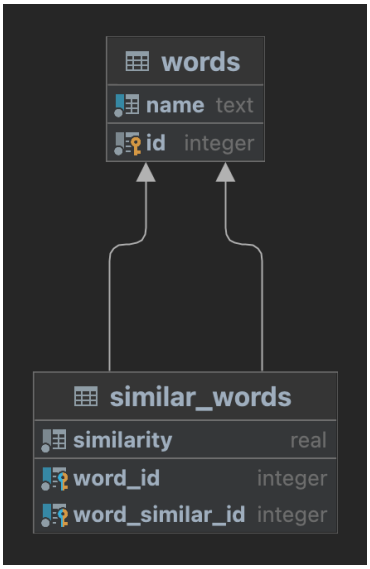
2. Quando encontrado um caractere consoante (ou par inseparável) seguido por um caractere vogal, são identificados automaticamente dois *tokens* [C] e [V]. Pode-se

Figura 11 – *Schema* do banco de dados

```sql
CREATE TABLE IF NOT EXISTS words
(
    id   INTEGER PRIMARY KEY NOT NULL,
    name TEXT UNIQUE         NOT NULL
);

CREATE TABLE IF NOT EXISTS similar_words
(
    word_id         INTEGER NOT NULL,
    word_similar_id INTEGER NOT NULL,
    similarity      REAL    NOT NULL,
    PRIMARY KEY (word_id, word_similar_id),
    FOREIGN KEY (word_id) REFERENCES words (id),
    FOREIGN KEY (word_similar_id) REFERENCES words (id)
);
```

Fonte: Elaboração Própria.

Figura 12 – Modelo do banco de dados



Fonte: Elaboração Própria.

adicionar um *token* de separação [SEP] antes de adicionar os novos se pelo menos uma vogal já foi adicionada na lista de *tokens*.

3. Ao encontrar um caractere de vogal e o *token* anterior for de vogal, se o caractere "representar" o fonema de 'i' ou 'u' ele é *token* de semivogal [SV], caso contrário, é um *token* vogal [V] e, portanto, deve ser separado.

4. Ao encontrar um caractere de vogal e o *token* anterior for de semivogal [SV], ele é considerado uma vogal [V] e, portanto, deve ser separado.

Por exemplo, ao final do processo, a lista de *tokens* criada para separação da palavra "boizinho" pode ser representada por: [C][V][SV][SEP][C][V][SEP][C][V]. A partir dessa lista, são recuperados os caracteres originais contidos em cada *token* e é criada a lista resultante contendo as apenas as sílabas separadas.

O algoritmo, assim como no dicionário da língua portuguesa, mantém na mesma sílaba apenas ditongos decrescentes.

4.3.2 Algoritmo de detecção da sílaba tônica

A construção desse algoritmo baseou-se essencialmente nas regras de acentuação de palavras descritas no capítulo anterior. Assim como no algoritmo anterior, criou-se algumas expressões regulares para detectar padrões que são mencionados a seguir.

A partir das sílabas obtidas através do algoritmo de divisão silábica Percorre-se cada sílaba da palavra de trás para frente (no máximo 3). Se alguma posição tiver uma vogal com acento agudo ou circunflexo o algoritmo retorna a posição encontrada. A palavra será oxítona se a última sílaba:

1. ter 'ã' ou 'õ'. Exemplos: co-ra-**ÇÃO**, emo-**ÇÃO**, co-ra-**ÇÕES**.
2. terminar em 'i' ou 'u', seguidos ou não por 's' 'm' ou 'ns'. Exemplos: ca-**QUI**, tu-**PI**, al-**GUNS**.
3. terminar em 'o' seguido por 'm' ou 'ns'. Exemplos: ba-**TOM**, ba-**TONS**.
4. não terminar em 'a' ou 'e' seguidos ou não por 's' 'm' ou 'ns' e não terminar em 'o' seguido ou não por 's'. Exemplos: pa-**PEL**, an-**DAR**, ra-**PAZ**.

Caso contrário, ela é paroxítona.

*As átonas são identificadas através de uma lista.

**A palavra "porque", sem acento circunflexo, foi identificada como exceção e é tratada separadamente.

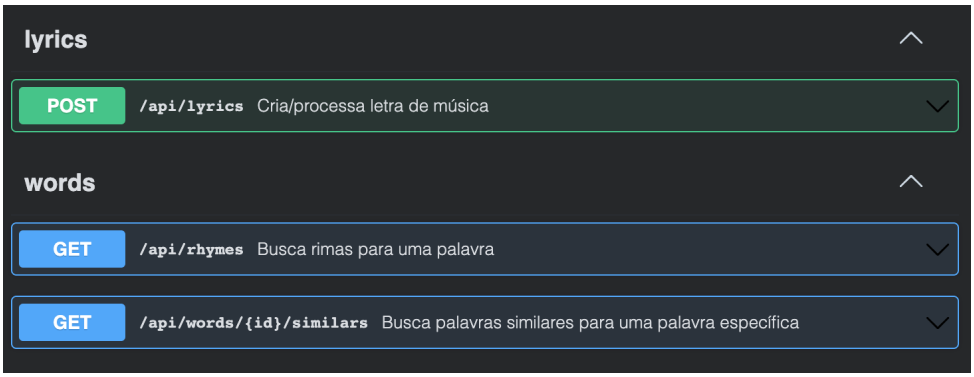
4.4 DESENVOLVIMENTO DA APLICAÇÃO SERVIDOR

Depois da criação do banco de dados (SQLite) gerado a partir do modelo do Word2Vec e da conclusão do desenvolvimento dos algoritmos, iniciou-se o desenvolvimento do *back-end* da ferramenta. Foi construída uma API REST utilizando a linguagem GO, Fiber como *framework web* e SQLC para realizar as consultas no banco de dados.

4.4.1 API

A API foi definida conforme é demonstrado na Figura 13. A API é detalhada a seguir.

Figura 13 – Rotas da API do sistema



Fonte: Elaboração Própria.

4.4.1.1 Modelo de dados

O modelo de dados utilizado na API é demonstrado na Figura 14.

Figura 14 – Modelo de dados utilizado na API

```
type InputLyric struct {  3 usages  Alvaro E
    Title string `json:"title"`
    Body  string `json:"body"`
}

type LyricLine struct {  3 usages  Alvaro E
    Words         []*word.Word `json:"words"`
    SyllableCount int          `json:"syllableCount"`
}

type Lyric struct {  3 usages  Alvaro E
    Lines []*LyricLine `json:"lines"`
}

type Word struct {  1 usage  new *
    ID         int64    `json:"id"`
    Name       string   `json:"name"`
    Syllables  []string `json:"syllables"`
    StressType int      `json:"stressType"`
}

type SimilarWord struct {  no usages  new *
    Word
    Similarity float64 `json:"similarity"`
}
```

Fonte: Elaboração Própria.

4.4.1.2 POST /api/lyrics

Rota para criar/processar uma letra de música e retorna a letra com separação sílabica, contagem de sílabas etc.

1. **Entrada**: Recebe um objeto do tipo InputLyric no corpo da mensagem;
2. **Saída**: Retorna um objeto do tipo Lyric em caso de sucesso.

4.4.1.3 GET /api/rhymes

Rota para buscar rimas para uma palavra fornecida.

1. **Entrada**: Recebe uma palavra (*string*) no valor do parâmetro de query *q*;
2. **Saída**: Retorna uma lista de objetos do tipo Word em caso de sucesso.

4.4.1.4 GET /api/words/:id/similars

Rota para buscar por palavras similares usando o ID de uma palavra.

1. **Entrada**: Espera um parâmetro id (inteiro) na rota, que é o ID da palavra para a qual palavras similares são buscadas.
2. **Saída**: Retorna uma lista de objetos do tipo SimilarWord em caso de sucesso

4.4.1.5 Detalhes do desenvolvimento

Na figura 15, temos um exemplo da construção da rota da API para retornar a lista de palavras relacionadas. Nesse exemplo, é esperado que a aplicação cliente realize uma

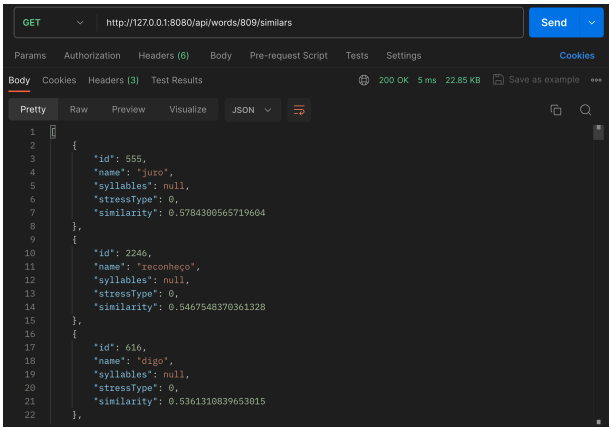
Figura 15 – Rota da API para obter palavras relacionadas

```
app.Get( path: "/api/words/:id/similars", func(c *fiber.Ctx) error {
    q := db.New(database)
    s := word.NewService(q)

    id, err := c.ParamsInt( key: "id")

    if err != nil { err }

    sw, err := s.FindSimilarWordsById(ctx, int64(id))

    if err != nil { err }

    return c.Status(fiber.StatusOK).JSON(sw)
})
```

Fonte: Elaboração Própria.

requisição GET passando o valor do identificador numérico da palavra, como no exemplo da Figura 16 em que é realizada uma requisição com o Postman para encontrar as palavras

Figura 16 – Requisição HTTP/GET com Postman para obter palavras similares



Fonte: Elaboração Própria.

similares a palavra “confesso” (id=809). Nesse exemplo, ainda, pode-se observar o retorno da API em JSON contendo uma lista de palavras relacionadas. Para gerar o código responsável por acessar o banco de dados com o SQLc são necessários três arquivos:

1. O *schema* do banco de dados (Figura 11);
2. Um arquivo yaml de configurações conforme o da Figura 17;

Figura 17 – Arquivo de definição sqlc.yml

```yaml
version: "2"
sql:
  - schema: "word/schema.sql"
    queries: "word/queries.sql"
    engine: "sqlite"
    gen:
      go:
        package: "db"
        out: "word/db"
```

Fonte: Elaboração Própria.

3. Um arquivo SQL contendo as consultas com alguns comentários que instruem o SQLc a gerar código (Figura 18).

Com o comando *sqlc generate*, o SQLc gera todo código necessário para acessar o banco de dados.

Para ordenar a lista de rimas, utilizou-se como critério a distância de *Levenshtein*, e o número de sílabas. Dessa forma, ocupam as primeiras posições na lista, as palavras que são mais parecidas com a palavra pesquisada.

Figura 18 – Definição do arquivo de *queries* do SQLc

```
-- name: FindWordById :one
SELECT *
FROM words
WHERE id = ?
LIMIT 1;
-- name: FindWordByName :one
SELECT *
FROM words
WHERE name = ?
LIMIT 1;
-- name: FindSimilarWordsById :many
SELECT w.id, w.name, sw.similarity
FROM similar_words sw
         JOIN words w on w.id = sw.word_similar_id
WHERE sw.word_id = ?
ORDER BY sw.similarity DESC;
-- name: FindAllWords :many
SELECT *
FROM words
```

Fonte: Elaboração Própria.

4.5 DESENVOLVIMENTO DA APLICAÇÃO CLIENTE

Para o desenvolvimento a aplicação cliente, empregou-se o framework Svelte, integrado com HTML, Tailwind CSS e TypeScript, e adotou-se o Skeleton.dev[2] como kit de interface do usuário (UI). A aplicação comunica-se com a API do servidor através requisições criadas com a função *fetch*. Um trecho de código que utiliza o comando pode ser visto na Figura 19. A aplicação permite que o usuário escreva a letra da música na lateral

Figura 19 – Método responsável por carregar as palavras similares do servidor

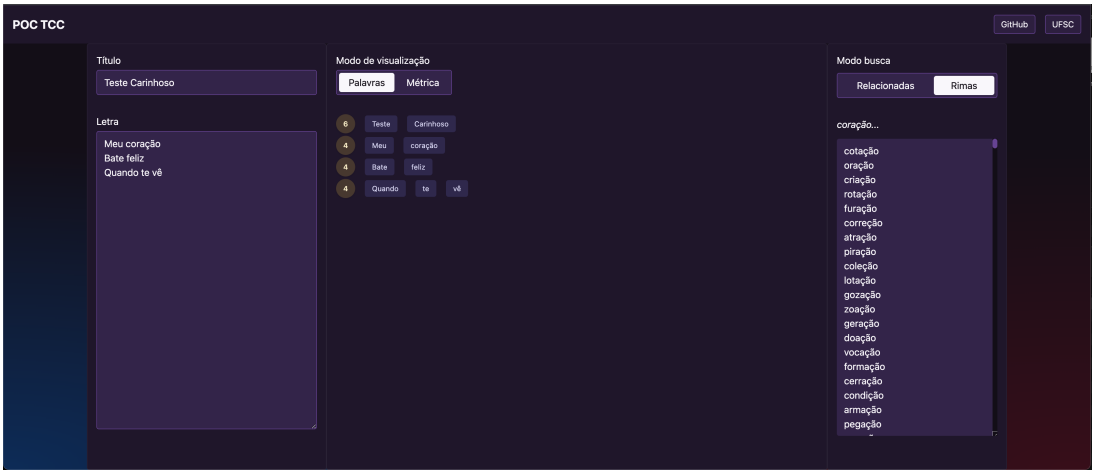
```
async function loadSimilarWords(word: Word) {
    if (!word || !browser){
        return;
    }
    const response = await fetch(`/api/words/${word.id}/similars`, {
        method: "GET"
    })
    similarWords = '';
    textArea.scrollTop = 0;
    const words = await response.json() as Word[];

    similarWords = words.map(w => w.name).join('\n');
}
```

Fonte: Elaboração Própria.

[2] `http://skeleton.dev`

esquerda, enquanto a parte central da tela vai sendo atualizada. A Figura 20 representa a ferramenta no modo de visualização de palavras. Nesse modo, as palavras de cada linha

Figura 20 – Interface gráfica da ferramenta no modo de palavras.

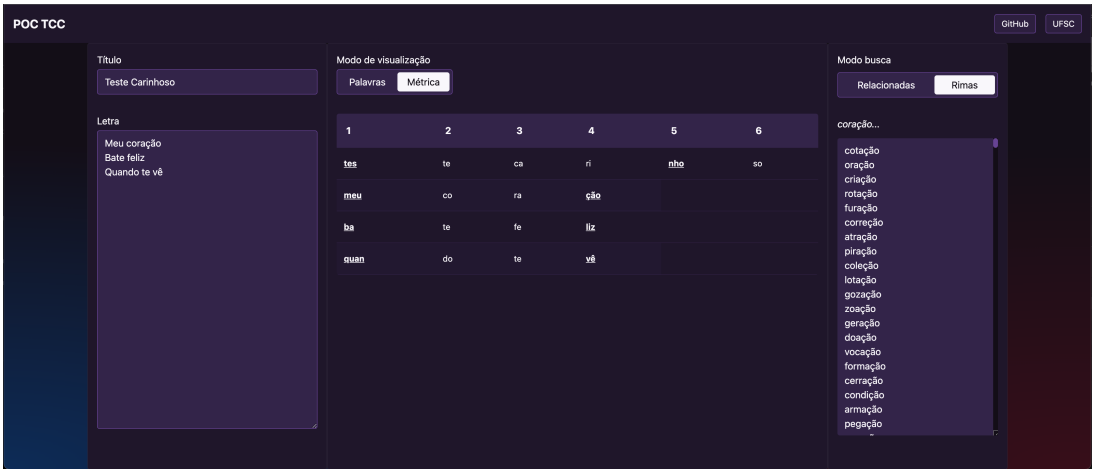


Fonte: Elaboração Própria.

vão sendo exibidas de modo que o usuário pode clicar para encontrar palavras relacionadas ou rimas (de acordo com o modo de busca selecionado no canto direito). No início de cada linha, é exibido um contador de sílabas. O resultado da busca é exibido na lateral direita.

A Figura 21 mostra a ferramenta no modo visualização métrica. A ferramenta exibe a separação silábica de cada linha, destacando cada sílaba tônica. As sílabas de cada linha ficam dispostas em colunas distintas.

Figura 21 – Interface da ferramenta no modo métrica.



Fonte: Elaboração Própria.

4.6 RESULTADOS OBTIDOS

Como forma de validar os algoritmos que foram produzidos durante este trabalho, foram realizados testes de integração. Mas optou-se por fazer um teste ainda maior, criando para isso uma base de referência a partir do dicionário de separação silábica do Portal da Língua Portuguesa (PLB, 2023) e comparando com a saída dos algoritmos que foram implementados. Essas palavras foram obtidas com *web crawling*. A saída do teste pode ser observada na Figura 22 e a síntese desses resultados pode ser observado na Tabela 2. A única falha produzida pelos algoritmos propostos foi com a palavra *baby* que é um

Figura 22 – Resultado do teste de mil palavras

```
=== RUN   TestTopWords
    top_words_test.go:32: PLB expected 'mas, we got mas
    top_words_test.go:32: PLB expected sa-u-'da-de, we got sau-'da-de
    top_words_test.go:32: PLB expected 'ao, we got ao
    top_words_test.go:32: PLB expected 'por-que, we got por-'que
    top_words_test.go:32: PLB expected 'gar-çom, we got gar-'çom
    top_words_test.go:32: PLB expected 'ba-by, we got 'baby
    top_words_test.go:32: PLB expected 'ba-tom, we got ba-'tom
--- FAIL: TestTopWords (0.01s)
```

Fonte: Elaboração Própria.

estrangeirismo. O resultado deste teste sinaliza que os algoritmos têm um bom nível de acerto.

Tabela 2 – Síntese dos resultados de validação

Descrição	Quantidade
Total de palavras validadas	1000
Total de divergências	7
Erros da implementação	1
Erros da base de referência (PLB)	6

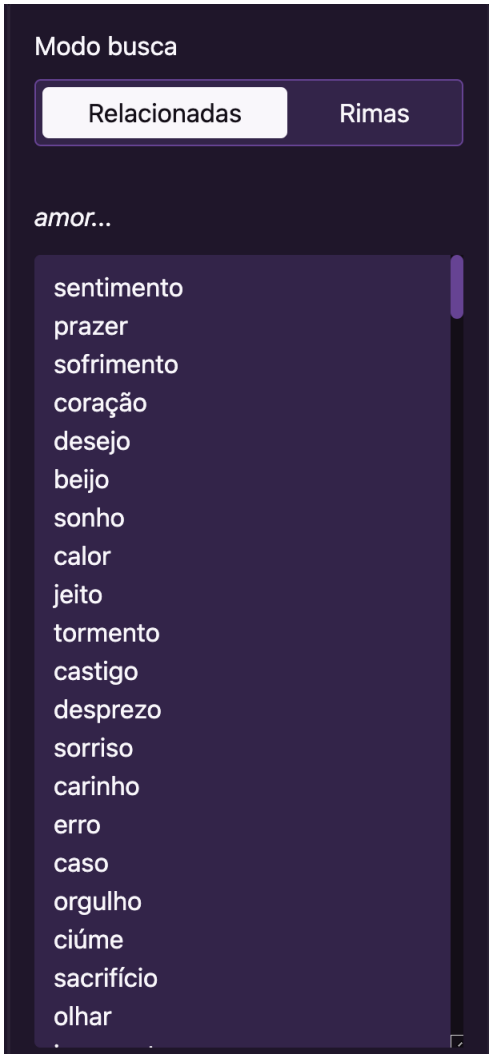
Fonte: Elaboração própria.

Apesar do resultado positivo, a língua portuguesa é repleta de exceções e os algoritmos desenvolvidos não são infalíveis. Por exemplo, durante o desenvolvimento foram detectados problemas na separação das palavras “sub-li-nhar” e “bo-cai-u-va”. Mas como essas são palavras raras em letras musicais e, mesmo com a separação incorreta, o algoritmo identifica corretamente a tonicidade dessas palavras. Optou-se então por não alterar o algoritmo de separação para contemplar esses casos.

A validação da acurácia da funcionalidade da busca por palavras relacionadas pode ser subjetiva. Mas foi possível observar que a ferramenta traz alguns bons resultados,

especialmente quando a palavra é uma palavra comum no *corpus*, como é o caso da palavra *amor*. Algumas palavras que a ferramenta relaciona com a palavra *amor* são mostradas na Figura 23.

Figura 23 – Palavras relacionadas à palavra *amor*



Fonte: Elaboração Própria.

É importante salientar que, ainda que uma palavra encontrada aparentemente não se relacione com a palavra pesquisada, ainda pode servir de inspiração para criação de uma letra de música. A ferramenta proposta atendeu a todos os requisitos funcionais e não-funcionais estabelecidos no inicio deste projeto, bem como foram atendidos todos os objetivos propostos neste trabalho.

4.6.1 Testes de desempenho

Foi realizado *benchmark* dos algoritmos desenvolvidos:. A máquina do ambiente de teste foi um Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz com 32gb de memória RAM. O

teste consistiu em medir o tempo de execução do algoritmo de silabificação mais o tempo de determinação da sílaba. O resultado é mostrado na Tabela 3.

Tabela 3 – Resultados de desempenho

Descrição	Tempo (ms)
Tempo total de execução / 1000 palavras	4,707918
Tempo total de execução / palavra	$\approx 0,00471$

Fonte: Elaboração própria.

5 CONCLUSÃO

Neste trabalho realizou-se o desenvolvimento de uma aplicação web para auxiliar na criação de letras de músicas. A ferramenta permite que o usuário pesquise por rimas e palavras similares, além de ter um modo em que usuário pode visualizar a divisão silábica de todas as linhas com as sílabas tônicas em destaque.

Com o desenvolvimento da ferramenta concluído com sucesso, tem-se que esse o trabalho cumpriu com todos os objetivos propostos desde da sua concepção, passando pela criação de um *corpus* linguístico de letras de música através de *web crawling*, pré-processamento do *corpus*, treinamento de modelos de inteligência artificial utilizando algoritmos de *embedding*, desenvolvimento de algoritmos de divisão silábica e determinação da sílaba tônica, criação de uma API para o back-end, desenvolvimento da interface gráfica da ferramenta, bem como a validação da ferramenta finalizada com todas as funcionalidades que foram pretendidas. Espera-se que esse trabalho possa contribuir e inspirar a realização de outros trabalhos acadêmicos relacionados à musica e a língua portuguesa. Todo o código-fonte produzido está disponível no repositório indicado no apêndice deste documento.

5.0.1 Trabalhos Futuros

São sugestões para a continuidade do projeto:

- Tornar o *front-end* da ferramenta mais acessível a dispositivos móveis;
- Otimizar e formalizar os algoritmos de divisão silábica e de determinação da sílaba tônica;
- Adicionar paginação nas listas de palavras sugeridas;
- Criar novos modelos a partir de letras de outros gêneros musicais e permitir que o usuário alterne entre os modelos;
- Adicionar funcionalidade para persistir/recuperar letras;
- Adicionar funcionalidade para buscar rimas de acordo com um tipo específico de rima;
- Adicionar funcionalidade para exportar/compartilhar a letra;
- Adicionar funcionalidade para adicionar/remover as relações existentes entre as palavras;
- Adicionar funcionalidade para geração de pequenos textos com inteligência artificial como, por exemplo, frases e títulos.

REFERÊNCIAS

ARAÚJO, Luciana Kuchenbecker. **Divisão Silábica - Português**. 2023. Disponível em: https://www.portugues.com.br/gramatica/silaba-divisao-silabica.html. Acesso em: 12 out. 2023.

BECHARA, Evanildo. **Moderna gramática portuguesa**. [*S.l.*]: Nova Fronteira, 2012.

BUARQUE, Chico. **Construção**. [*S.l.*: *s.n.*], 1971. Em: Construção. Philips Records.

DAVID, A. **Songwriting: Strategies for Musical Self-Expression and Creativity**. [*S.l.*]: Scarecrow Press, 2010.

DHARMA, Eddy Muntina *et al.* The accuracy comparison among word2vec, glove, and fasttext towards convolution neural network (cnn) text classification. **J Theor Appl Inf Technol**, v. 100, n. 2, p. 31, 2022.

DIANA, Daniela. **Verso, Estrofe e Rima**. [*S.l.*]: Toda Matéria, jul. 2015. Disponível em: https://www.todamateria.com.br/verso-estrofe-e-rima/. Acesso em: 27 nov. 2019.

DONOVAN, Alan A.A.; KERNIGHAN, Brian W. **The Go Programming Language**. [*S.l.*]: Addison-Wesley Professional, 2015.

FERNANDES, Márcia. **Vogal, Semivogal e Consoante**. 2023. Disponível em: https://www.todamateria.com.br/vogal-semivogal-e-consoante/. Acesso em: 10 set. 2023.

FIBER. **Fiber**. 2023. Disponível em: https://gofiber.io/. Acesso em: 10 out. 2023.

FRIEDL, Jeffrey EF. **Mastering Regular Expressions**. 3. ed. [*S.l.*]: O'Reilly Media, Inc., 2006.

GOCOLLY. **GoColly**. 2023. Disponível em: https://go-colly.org/docs/. Acesso em: 11 out. 2023.

JANZEN, David S; SAIEDIAN, Hossein. Test-driven development: Concepts, taxonomy, and future direction. **Computer**, IEEE, v. 38, n. 9, p. 43–50, 2005.

KENNY MC. **Lyric Notepad - Song Writing**. 2023. Disponível em: https://play.google.com/store/apps/details?id=kmcilvai.perfectpoet&hl=pt_BR. Acesso em: 9 set. 2023.

KHDER, Moaiad. Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application. v. 13, p. 145–168, dez. 2021. ISSN 27101274, 20748523. DOI:

10.15849/IJASCA.211128.11. Disponível em: http://ijasca.zuj.edu.jo/PapersUploaded/2021.3.11.pdf.

LAUDON, Kenneth C.; LAUDON, Jane P. **Management Information Systems: Managing the Digital Firm**. 14. ed. [*S.l.*]: Pearson, 2016.

LEVENSHTEIN, Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. **Soviet physics doklady**, v. 10, n. 8, p. 707–710, 1966.

LEXICON. **Lexicon - Dicionário Analógico**. 2023. Disponível em: https://aulete.com.br/analogico. Acesso em: 9 set. 2023.

LUTZ, Mark. **Learning Python**. 5. ed. [*S.l.*]: O'Reilly Media, 2013.

MITTMANN, Adiel. **Escansão Automática de Versos em Português**. 2016. Tese (Doutorado) – Universidade Federal de Santa Catarina.

MONTE, Anderson *et al.* A Rule-based Syllabification Algorithm with Stress Determination for Brazilian Portuguese Natural Language Processing. *In:* INTERNATIONAL Congress of Phonetic Sciences. [*S.l.*: *s.n.*], 2011. Disponível em: https://api.semanticscholar.org/CorpusID:42651034.

NÉMETH, László *et al.* **Hunspell**. 2023. Disponível em: https://hunspell.github.io/. Acesso em: 9 set. 2023.

NEVES, Flávia. **Sílaba Tônica: Oxítona, Paroxítona e Proparoxítona**. [*S.l.*]: Norma Culta, dez. 2017. Disponível em: https://www.normaculta.com.br/silaba-tonica-oxitona-paroxitona-e-proparoxitona/. Acesso em: 26 nov. 2019.

PATTISON, Pat. **Essential Guide to Rhyming**. Boston, MA: Berklee Press, 2014. ISBN 0876391501.

_____. **Writing Better Lyrics**. Cincinnati, OH: Writers Digest Books, 2009. ISBN 1582975779.

PLB. **Portal da Língua Portuguesa**. Disponível em: http://www.portaldalinguaportuguesa.org. Acesso em: 9 set. 2023.

POSTMAN. **Postman: The Collaboration Platform for API Development**. 2023. Disponível em: https://www.postman.com/. Acesso em: 9 set. 2023.

RAPPAD. **RapPad**. Disponível em: https://www.rappad.co/editor. Acesso em: 9 set. 2023.

REDHAT. **What is an API?** 2023. Disponível em: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces. Acesso em: 5 out. 2023.

ROCHA, Willian; NETO, Nelson. Implementação de um Separador Silábico Gratuito Baseado em Regras Linguísticas para o Português Brasileiro (The Implementation of a Free Syllabification Algorithm Based on Linguistic Rules for Brazilian Portuguese) [in Portuguese]. *In:* BRAZILIAN Symposium in Information and Human Language Technology. [*S.l.*: *s.n.*], 2013. Disponível em: https://api.semanticscholar.org/CorpusID:10446942.

SCOTT, C. **The Tools of Poetry**. [*S.l.*]: Allworth Press, 2006.

SENADO FEDERAL. **Manual de Comunicação do Senado**. [*S.l.*: *s.n.*], 2023. https://www12.senado.leg.br/manualdecomunicacao. Acesso em: 15 nov. 2023.

SILVA, Denilson; BRAGA, Daniela; RESENDE, Fernando. Separação das Sílabas e Determinação da Tonicidade no Português Brasileiro. *In:* XXVI Simpósio Brasileiro de Telecomunicações. [*S.l.*: *s.n.*], 2008. DOI: 10.14209/sbrt.2008.42892.

SOMMERVILLE, Ian. Software engineering (ed.) **America: Pearson Education Inc**, 2011.

SQLC. **SQLC**. 2023. Disponível em: https://docs.sqlc.dev/. Acesso em: 11 out. 2023.

SQLITE. **SQLite Documentation**. 2023. Disponível em: https://www.sqlite.org/docs.html. Acesso em: 11 out. 2023.

SVELTE. **Svelte**. 2023. Disponível em: https://svelte.dev/. Acesso em: 10 out. 2023.

WAZLAWICK, Raul. **História da Computação**. [*S.l.*]: Elsevier, 2016. ISBN 8535285458.

Apêndices

APÊNDICE A – CÓDIGO-FONTE

O código-fonte produzido durante o desenvolvimento deste projeto está disponível em: https://github.com/alvarofln/tcc/
A ferramenta estará disponível temporariamente em https://poc-tcc.up.railway.app/

APÊNDICE B – ARTIGO

Uma Ferramenta para Auxiliar na Criação de Letras Musicais

Álvaro E. Prüsse[1]

[1]Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

alvaro.e@grad.ufsc.br

Abstract. *There are a considerable amount of materials and tools available that can potentially contribute to the process of creating lyrics. However, this content is distributed in different software, websites, books, etc. Specific software for creating lyrics is rare. The vast majority of tools on the market are intended for the English language. This work is intended to create a web tool to gather some features that can stimulate and accelerate the process of creating lyrics in Portuguese. The main features of the tool are: syllable separation / syllable counter, suggestion of related words, suggestion of words with the same classification as the stressed syllable. All suggestions are constructed from processing lyrics of the same genre, which can produce better results because they are closer to the same context. The lyrics were extracted with web crawling to form a linguistic corpus. From the corpus, a model was trained with Word2Vec, responsible for finding similar words. Furthermore, two algorithms based on linguistic rules were proposed: one for syllabic separation and one for determining the stressed syllable.*

Resumo. *Há uma quantidade considerável materiais e ferramentas disponíveis que potencialmente podem contribuir no processo de criação letras musicais. Porém, esse conteúdo encontra-se distribuído em diferentes softwares, sites, livros etc. São raros softwares específicos para criar letras de músicas. A grande maioria das ferramentas existentes no mercado são destinados especificamente para língua inglesa. Este trabalho é destinado à criação de uma ferramenta web para reunir algumas funcionalidades que possam estimular e acelerar o processo de criação de letras musicas na língua portuguesa. As principais funcionalidades da ferramenta são: separação silábica/contador de sílabas, sugestão de palavras relacionadas, sugestão de palavras com a mesma classificação quanto à sílaba tônica. Todas as sugestões são construídas a partir do processamento de letras musicais de um mesmo gênero, o que pode produzir resultados melhores por estarem mais próximas do mesmo contexto. As letras foram extraídas com web crawling para formar um corpus linguístico. A partir do corpus, treinou-se um modelo com do Word2Vec, responsável por encontrar palavras similares. Ainda, foram propostos dois algoritmos baseados em regras linguísticas: um de separação silábica e um de para determinar a sílaba tônica.*

1. Introdução

À luz de a *História da Computação* [Wazlawick 2016], observa-se que a computação evoluiu, na maioria das vezes, norteada pela busca da redução de esforço, tanto físico como intelectual, dos seres humanos.

Hoje, a computação está cada vez mais presente na vida das pessoas, capaz de realizar ou acelerar as mais diversas tarefas, das mais corriqueiras até as mais complexas. É nesse contexto, de que a computação pode tornar a vida das pessoas mais produtivas, que é proposto conceber uma ferramenta para auxiliar, seja na forma, seja no conteúdo, a criação de letras musicais na língua portuguesa.

Este trabalho é inspirado fortemente pelas ideias de Pat Pattinson, professor da *Berklee College of Music*. Ele acredita que não existem regras na esfera musical, mas que é possível estender o conceitos de *Tensão X Resolução*, existentes na música ocidental, às letras de músicas. Segundo ele, todos elementos que compõem uma música, incluindo a forma de como a letra da música é escrita (número de linhas, esquemas de rimas etc.), podem convergir para transmitir uma mesma sensação de estabilidade ou de instabilidade, tornando a canção mais interessante. Para ele, por exemplo, um refrão com um número ímpar de linhas tende à soar mais instável (ou inacabado) do que um refrão com um número par de linhas.[Pattison 2009]

Em [Pattison 2014] percebe-se que é bastante comum os compositores utilizarem dicionários de rimas e *thesaurus*[1] para auxiliar no processo de criação de letras de música. Esses dicionários, por muitas vezes, encontram-se de forma impressa. Ele encoraja o uso dessas ferramentas para estimular o processo criativo e evitar possíveis "bloqueios" na hora de compor. Este trabalho propõe sintetizar algumas dessas ideias em uma só ferramenta para auxiliar no processo de criação de uma letra de música.

Há uma quantidade considerável de materiais e ferramentas disponíveis que potencialmente podem contribuir no processo de criação de letras musicais. Porém, esse conteúdo encontra-se distribuído em diferentes softwares, sites, livros etc. São raros os softwares específicos para criar letras de músicas. Os softwares mais semelhantes ao proposto por este trabalho são exclusivamente para letras em inglês e não são disponibilizados de forma livre. É esperado que a ferramenta proposta possa auxiliar na compreensão da estrutura da língua portuguesa, na composição de letras musicais, na educação musical, e que esse trabalho possa, de alguma forma, servir de inspiração para o desenvolvimento de outras ferramentas voltadas para a música e para a língua portuguesa.

1.1. Objetivos

1.1.1. Objetivo Geral

Desenvolver um software para auxiliar na criação de letras musicais em língua portuguesa.

1.1.2. Objetivos Específicos

- Coletar uma grande quantidade de letras de músicas em português, de um mesmo gênero musical, disponíveis na internet para criação de um *corpus* linguístico;
- Encontrar uma técnica para identificar, de forma automática, palavras relacionadas entre si a partir do *corpus*;
- Encontrar uma técnica para realizar a separação silábica das palavras e identificar a sílaba tônica em cada uma delas;

[1] No Brasil, o termo é conhecido como dicionário analógico ou dicionário de ideias afins.

- Organizar e estruturar as informações obtidas;
- Criar uma aplicação web (cliente/servidor) para tornar possível ao usuário final criar letras de músicas interagindo com as informações descritas nos itens supracitados;
- Analisar e discutir os resultados obtidos na construção da ferramenta proposta por este trabalho;
- Disponibilizar todo o código-fonte produzido de forma livre.

2. Metodologia

A metodologia de pesquisa e desenvolvimento do trabalho dividiu-se em quatro principais etapas:

- **Etapa 1:** Levantamento do que já existe disponível em nível de software relacionado à ferramenta produzida neste trabalho. Procedeu-se com pesquisa em motores de busca como o da *Google*[2] e em lojas de aplicativos móveis *Play Store*[3] para *Android*, e *App Store*[4] para *iOS*;
- **Etapa 2:** Levantamento bibliográfico para dar suporte à solução desenvolvida durante a execução deste projeto. A pesquisa foi feita através de artigos, documentações e publicações relacionadas ao assunto. Utilizou-se principalmente o *Google Scholar*[5] e o *CiteSeerX*[6];
- **Etapa 3:** Análise de requisitos e desenvolvimento da ferramenta proposta por este trabalho.
- **Etapa 4:** Análise e discussão dos resultados obtidos na construção da ferramenta proposta por este trabalho.

3. Trabalhos relacionados

Especialmente, referente a um desafios técnicos deste projeto, pesquisou-se por palavras-chave como "divisão silábica", "algoritmo divisão silábica", "*portuguese syllabification*" e "*portuguese syllabification algorithm*" em motores de busca acadêmicos, convencionais e repositórios de código-fonte de software. Os trabalhos encontrados com o maior grau de relevância para possivelmente suportar o desenvolvimento deste projeto foram os seguintes:

1. Separação das Sílabas e Determinação da Tonicidade no Português Brasileiro [Silva et al. 2008];
2. *A Rule-based Syllabification Algorithm with Stress Determination for Brazilian Portuguese Natural Language Processing* [Monte et al. 2011];
3. *The Implementation of a Free Syllabification Algorithm Based on Linguistic Rules for Brazilian Portuguese* [Rocha and Neto 2013];
4. Escansão Automática de Versos em Português [Mittmann 2016].

autores [Silva et al. 2008] desenvolveram dois algoritmos baseados em regras linguísticas para separação silábica e determinação da tonicidade em palavras do português brasileiro.

[2]https://www.google.com
[3]https://play.google.com
[4]https://www.apple.com/ios/app-store/
[5]https://scholar.google.com
[6]https://citeseerx.ist.psu.edu

Os algoritmos foram testados com uma amostra da base de dados CETEN-Folha. Os resultados mostraram uma taxa de erro de 0,71% na separação das sílabas e 1,1% na determinação da tonicidade. São descritas vinte regras linguísticas no trabalho, mas o processo de criação e motivação por trás de cada regra linguística não foi completamente descrito.

O trabalho de [Monte et al. 2011] propõe aprimorar o conjunto existente de regras linguísticas propostas por [Silva et al. 2008] para a divisão silábica de palavras em português brasileiro. Os autores ressaltam a importância dos algoritmos para melhorar a eficiência de sistemas TTS e de outros sistemas inteligentes que processam textos em português brasileiro. Destacam também a vantagem do uso do algoritmo, em comparação ao uso de um dicionário, pois o algoritmo evita o uso de um arquivo grande para armazenar todas as palavras de um idioma e é capaz de lidar com palavras novas. O algoritmo atualizado demonstrou um número significativamente menor de erros em comparação com a versão anterior, principalmente no contexto de ditongos e hiatos. A implementação do algoritmo foi realizada na linguagem Java, e é *open-source*.

O trabalho de [Rocha and Neto 2013] também propõe melhorias ao conjunto de regras linguísticas do trabalho de [Silva et al. 2008]. As melhorias visam lidar melhor com casos de encontros vocálicos. O autor relata a escassez de recursos e pesquisas dedicados à divisão silábica do português brasileiro em comparação com outras línguas, especialmente no contexto de sistemas de TTS. O autor explica ainda que a divisão silábica é um pré-requisito importante para esses sistemas, pois influencia a transcrição fonética e a geração de prosódia. O trabalho produziu um separador silábico, *open-source* escrito na linguagem C#. Foram realizados testes de validação com um grande número de palavras. O algoritmo atualizado demonstrou melhor desempenho em comparação os trabalhos de [Silva et al. 2008] e [Monte et al. 2011], especialmente no tratamento de hiatos e ditongos. O autor conclui que, apesar dos avanços, há espaço para mais refinamentos nas regras e na manipulação de estrangeirismos.

O trabalho de [Mittmann 2016] é uma tese de doutorado que apresenta o Aoidos, um sistema capaz de escandir versos em português. No que se diz respeito às funcionalidades pretendidas neste trabalho, primeiro, de acordo com um conjunto de regras propostas pelo autor, a posição da vogal tônica (ou a ausência dela) é determinada para cada palavra do verso de um poema. Em seguida, cada grafema é classificado em vogal, consoante ou semivogal, e a partindo da ideia de que a vogal é o núcleo cada sílaba, a divisão silábica da palavra é estabelecida. A taxa média de acerto do sistema foi de 99,0%, variando entre os corpora de 97,5 a 100,0%.

A pesquisa não encontrou nenhum código-fonte disponível em nenhuma das referências. Para atingir os objetivos propostos por este trabalho, optou-se por seguir na direção de criar dois novos algoritmos: um para separação silábica e outro para a identificação da sílaba tônica. A funcionalidade para buscar rimas depende desses dois algoritmos.

4. Desenvolvimento

A partir da etapa de pesquisa concluída, iniciou-se uma etapa de levantamento de requisitos.

4.1. Levantamento de requisitos

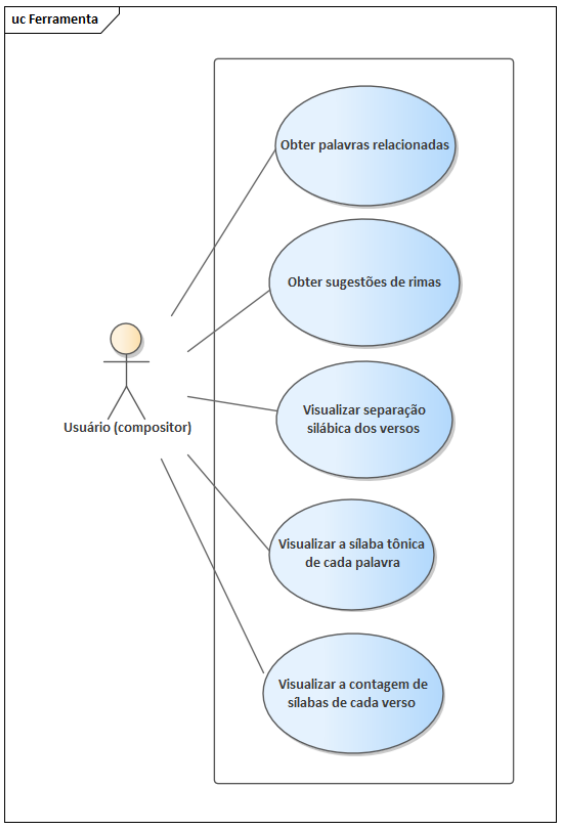
Os requisitos podem classificados em RF ou RNF. Os RF definem funções específicas ou comportamentos do sistema. Eles descrevem o que o sistema deve fazer, as ações que deve executar em determinadas condições, as respostas a entradas específicas e como deve se comportar em determinadas situações. Por outro lado, os RNF são restrições sobre os serviços ou funções oferecidos pelo sistema. Eles incluem restrições de tempo, restrições sobre o processo de desenvolvimento e padrões. os RFs muitas vezes se referem à qualidade do sistema e abrangem áreas como desempenho, segurança e confiabilidade [Sommerville 2011].

Diante disso, foram propostos os seguintes requisitos:

- RF01 - Inserção da letra de música pelo usuário.
- RF02 - Seleção de palavras para buscar palavras relacionadas ou rimas.
- RF03 - Exibição de palavras relacionadas ou rimas.
- RF04 - Visualização da letra com separação silábica com destaque da sílaba tônica

- RNF01 - O SGBD utilizado deve ser preferencialmente o SQLite.
- RNF02 - O sistema será desenvolvido especificamente para uso em navegadores de *desktop*, sem necessidade de adaptação para dispositivos móveis.

Figure 1. Diagrama de casos de uso da ferramenta



4.2. Diagrama de casos de uso

Segundo [Sommerville 2011], um diagrama de casos de uso é uma representação gráfica que descreve como os usuários (ou atores) interagem com um sistema, mostrando a relação entre os usuários e as diferentes funções do sistema.

O diagrama dos casos de uso da ferramenta proposta neste trabalho que pode ser visto na Figura 1.

4.3. Casos de uso

Os casos de uso são descrições de como os usuários realizam tarefas com o sistema, cada um representando um conjunto de ações executadas pelo sistema para fornecer um resultado útil para um ator. Os casos de uso são uma ferramenta valiosa para identificar e comunicar as funcionalidades que o sistema deve fornecer [Sommerville 2011].

Os casos de uso da ferramenta proposta neste trabalho estão descritos a seguir:

4.3.1. Obter palavras relacionadas

- Entrada: o usuário seleciona uma palavra da letra da música.
- Saída: uma lista de palavras relacionadas baseada na similaridade semântica.

4.3.2. Obter sugestões de rimas

- Entrada: o usuário seleciona uma palavra da letra da música.
- Saída: uma lista de sugestões de palavras que tem a mesma classificação de tonicidade da palavra selecionada.

4.3.3. Visualizar separação silábica dos versos

- Entrada: o usuário insere um mais versos.
- Saída: a representação da separação silábica de cada verso.

4.3.4. Visualizar a sílaba tônica de cada palavra

- Entrada: o usuário insere um mais versos.
- Saída: a representação da sílaba tônica de cada palavra no verso em destaque.

4.3.5. Visualizar a contagem de sílabas de cada verso

- Entrada: o usuário insere um mais versos.
- Saída: Contagem total de sílabas para cada verso fornecido.

4.4. Similaridade entre palavras

Esta seção dedica-se a documentar todas as atividades que foram realizadas para a construção do modelo de dados que permitiu que fosse criada a funcionalidade de busca por palavras relacionadas.

4.4.1. Extração dos dados

Foi implementado, em GoLang, um processo de *web crawling* com o intuito de coletar um extenso conjunto de letras de música de um site especializado [7]. Foram coletadas apenas letras de um único estilo musical (sertanejo) com a intenção de enriquecer a contextualização de palavras semelhantes. O estilo foi escolhido pela grande quantidade de letras de músicas disponíveis.

Utilizou-se o *framework* Colly para construir o *web crawler*. Ao final desse processo, foram contabilizados cerca de 25MB em texto em letras de músicas. Um único arquivo de texto foi criado, formando assim, um *corpus* de letras de músicas.

4.4.2. Pré-processamento

Essa etapa foi realizada na linguagem Python. Percebeu-se que o *corpus* possuía uma série de problemas, como erros gramaticais, comentários dentro das letras, marcações do tipo ”refrão”, ”repete 2x”, como pode ser visto na Figura 2. Então, durante o processo

Figure 2. Exemplo de problema no *corpus*

```
REFRÃO: 2x
Eu vou pagar a conta
A conta da tristeza e nisso cai na mesa o seu retrato
Veja amigo este rosto é a razão do meu desgosto
É a moça dessa foto 3x4
Repete tudo
```

para transformar o texto em conjuntos de palavras (tokenização), verificou-se também se as palavras estavam presentes no dicionário do HunSpell [Németh et al. 2023]. No fim desse processo, foram preparados um conjunto de sentenças. Foram consideradas apenas sentenças que tinham, pelo menos, três palavras e que todas elas estavam presentes no dicionário.

4.4.3. Treinamento do modelo

Nessa etapa foram conduzidos alguns experimentos preliminares com algoritmos para orientar a seleção de um modelo. Foram realizados experimentos criando modelos com o GloVe, Word2Vec e FastText. O tempo de treinamento desses modelos é na ordem de dez minutos utilizando o *corpus* completo. A vantagem percebida com o FastText é que ele consegue trabalhar com uma palavra que não esteja presente no vocabulário do modelo. No entanto, ele deixa mais próximas as palavras que são morfologicamente semelhantes e essas palavras aparecem com mais prioridade na busca, como é mostrado na Figura 3

Devido à facilidade do treinamento em Python em relação ao GloVe e por gerar uma lista onde os itens similares são priorizados de uma forma mais interessante que o

[7] https://www.vagalume.com.br

FastText, o Word2Vec foi o algoritmo escolhido para seguir com o treinamento para a obtenção do modelo final capaz de encontrar palavras similares.

Figure 3. FastText priorizando palavras morfologicamente semelhantes

4.4.4. Criação do banco de dados

Tipicamente, o modelo gerado com o Word2Vec só pode ser utilizado em Python. Para eliminar a dependência do modelo, para acelerar a busca de palavras similares e para facilitar o consumo dessas informações em outras linguagens, optou-se por extrair as informações de similaridades entre as palavras do modelo salvá-las em um banco de dados. Criou-se um banco de dados com o *schema* que pode ser visto na Figura 4.

Foram salvas todas as palavras presentes no modelo. Foram criados os relacionamentos para as duzentas e cinquenta mais similares de cada palavra em um banco de dados (SQLite).

O modelo do banco de dados também pode ser representado pela Figura 5 .

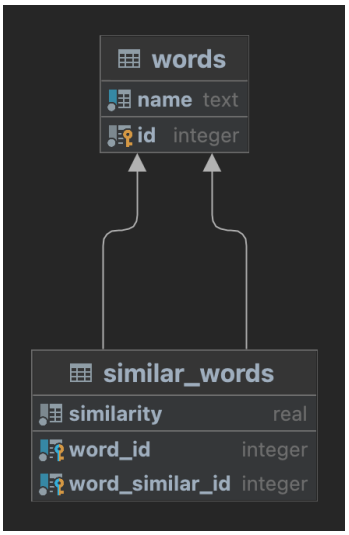
4.5. Algoritmos desenvolvidos

Esses algoritmos foram construídos a partir da pesquisa realizada e de experimentação manual, busca por padrões em dicionários etc. Eles foram desenvolvidos com TDD em GoLang, sendo contextualizados a seguir.

Figure 4. *Schema* do banco de dados

```sql
CREATE TABLE IF NOT EXISTS words
(
    id   INTEGER PRIMARY KEY NOT NULL,
    name TEXT UNIQUE         NOT NULL
);

CREATE TABLE IF NOT EXISTS similar_words
(
    word_id         INTEGER NOT NULL,
    word_similar_id INTEGER NOT NULL,
    similarity      REAL    NOT NULL,
    PRIMARY KEY (word_id, word_similar_id),
    FOREIGN KEY (word_id) REFERENCES words (id),
    FOREIGN KEY (word_similar_id) REFERENCES words (id)
);
```

Figure 5. Modelo do banco de dados



4.5.1. Algoritmo de divisão silábica

Foram criadas algumas expressões regulares para identificar diferentes tipos de combinações de letras, como encontros consonantais seguidos de vogais, vogais nasais, etc. O algoritmo recebe uma palavra e retorna uma lista de *strings* (sílabas). As principais ideias para a criação do algoritmo foram as seguintes:

1. O *token* de consoante [C] representa um caractere de consoante ou um par de caracteres que o algoritmo assume como inseparáveis. Como, por exemplo, os pares em destaque em: **br**a, **bl**a, **cr**e, **nh**a, **ch**a, **qu**a, **gu**a.
2. Quando encontrado um caractere consoante (ou par inseparável) seguido por um caractere vogal, são identificados automaticamente dois *tokens* [C] e [V]. Pode-se adicionar um *token* de separação [SEP] antes de adicionar os novos se pelo menos uma vogal já foi adicionada na lista de *tokens*.
3. Ao encontrar um caractere de vogal e o *token* anterior for de vogal, se o caractere "representar" o fonema de 'i' ou 'u' ele é *token* de semivogal [SV], caso contrário,

é um *token* vogal [V] e, portanto, deve ser separado.
4. Ao encontrar um caractere de vogal e o *token* anterior for de semivogal [SV], ele é considerado uma vogal [V] e, portanto, deve ser separado.

Por exemplo, ao final do processo, a lista de *tokens* criada para separação da palavra "boizinho" pode ser representada por: [C][V][SV][SEP][C][V][SEP][C][V]. A partir dessa lista, são recuperados os caracteres originais contidos em cada *token* e é criada a lista resultante contendo as apenas as sílabas separadas.

O algoritmo, assim como no dicionario do Portal da Língua Portuguesa, mantém na mesma sílaba apenas ditongos decrescentes.

4.5.2. Algoritmo de detecção da sílaba tônica

A construção desse algoritmo baseou-se essencialmente nas regras de acentuação de palavras descritas no [REGRASACENTUACAO]capítulo anterior. Assim como no algoritmo anterior, criou-se algumas expressões regulares para detectar padrões que são mencionados a seguir.

A partir das sílabas obtidas através do algoritmo de divisão silábica Percorre-se cada sílaba da palavra de trás para frente (no máximo 3). Se alguma posição tiver uma vogal com acento agudo ou circunflexo o algoritmo retorna a posição encontrada. A palavra será oxítona se a última sílaba:

1. ter 'ã' ou 'õ'. Exemplos: co-ra-**ÇÃO**, emo-**ÇÃO**, co-ra-**ÇÕES**.
2. terminar em 'i' ou 'u', seguidos ou não por 's' 'm' ou 'ns'. Exemplos: ca-**QUI**, tu-**PI**, al-**GUNS**.
3. terminar em 'o' seguido por 'm' ou 'ns'. Exemplos: ba-**TOM**, ba-**TONS**.
4. não terminar em 'a' ou 'e' seguidos ou não por 's' 'm' ou 'ns' e não terminar em 'o' seguido ou não por 's'. Exemplos: pa-**PEL**, an-**DAR**, ra-**PAZ**.

Caso contrário, ela é paroxítona.

*As átonas são identificadas através de uma lista.

**A palavra "porque", sem acento circunflexo, foi identificada como exceção e é tratada separadamente.

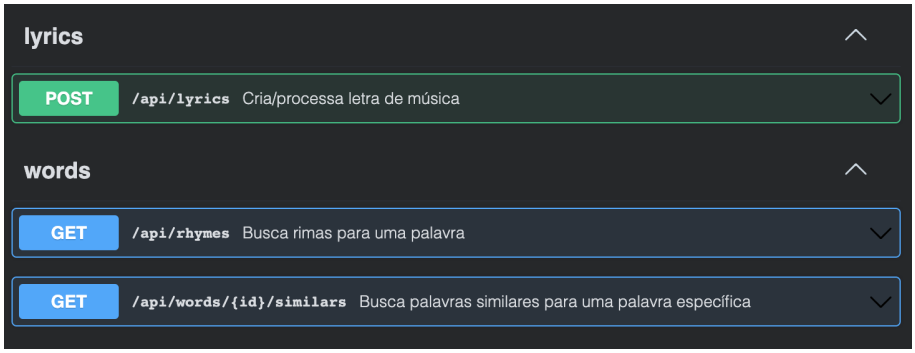
4.6. Desenvolvimento da aplicação servidor

Depois da criação do banco de dados (SQLite) gerado a partir do modelo do Word2Vec e da conclusão do desenvolvimento dos algoritmos, iniciou-se o desenvolvimento do *back-end* da ferramenta. Foi construída uma API REST utilizando a linguagem GO, Fiber como *framework web* e SQLC para realizar as consultas no banco de dados.

4.6.1. API

A API foi definida conforme é demonstrado na Figura 6. A API é detalhada a seguir.

Figure 6. Rotas da API do sistema



4.6.2. Modelo de dados

O modelo de dados utilizado na API é demonstrado na Figura 7.

Figure 7. Modelo de dados utilizado na API

```
type InputLyric struct {  3 usages  Alvaro E
    Title string `json:"title"`
    Body  string `json:"body"`
}

type LyricLine struct {  3 usages  Alvaro E
    Words         []*word.Word `json:"words"`
    SyllableCount int          `json:"syllableCount"`
}

type Lyric struct {  3 usages  Alvaro E
    Lines []*LyricLine `json:"lines"`
}

type Word struct {  1 usage  new *
    ID         int64    `json:"id"`
    Name       string   `json:"name"`
    Syllables  []string `json:"syllables"`
    StressType int      `json:"stressType"`
}

type SimilarWord struct {  no usages  new *
    Word
    Similarity float64 `json:"similarity"`
}
```

4.6.3. POST /api/lyrics

Rota para criar/processar uma letra de música e retorna a letra com separação sílabica, contagem de sílabas etc.

1. **Entrada**: Recebe um objeto do tipo InputLyric no corpo da mensagem;
2. **Saída**: Retorna um objeto do tipo Lyric em caso de sucesso.

4.6.4. GET /api/rhymes

Rota para buscar rimas para uma palavra fornecida.

1. **Entrada**: Recebe uma palavra (*string*) no valor do parâmetro de query *q*;
2. **Saída**: Retorna uma lista de objetos do tipo Word em caso de sucesso.

4.6.5. GET /api/words/:id/similars

Rota para buscar por palavras similares usando o ID de uma palavra.

1. **Entrada**: Espera um parâmetro id (inteiro) na rota, que é o ID da palavra para a qual palavras similares são buscadas.
2. **Saída**: Retorna uma lista de objetos do tipo SimilarWord em caso de sucesso

4.6.6. Detalhes do desenvolvimento

Na figura 8, temos um exemplo da construção da rota da API para retornar a lista de palavras relacionadas. Nesse exemplo, é esperado que a aplicação cliente realize uma

Figure 8. Rota da API para obter palavras relacionadas

```
app.Get( path: "/api/words/:id/similars", func(c *fiber.Ctx) error {
    q := db.New(database)
    s := word.NewService(q)

    id, err := c.ParamsInt( key: "id")

    if err != nil { err }

    sw, err := s.FindSimilarWordsById(ctx, int64(id))

    if err != nil { err }

    return c.Status(fiber.StatusOK).JSON(sw)
})
```

requisição GET passando o valor do identificador numérico da palavra, como no exemplo da Figura 9 em que é realizada uma requisição com o Postman para encontrar as palavras similares a palavra “confesso” (id=809). Nesse exemplo, ainda, pode-se observar o retorno da API em JSON contendo uma lista de palavras relacionadas. Para gerar o código responsável por acessar o banco de dados com o SQLc são necessários três arquivos:

1. O *schema* do banco de dados (Figura 4);
2. Um arquivo yaml de configurações conforme o da Figura 10;

Figure 9. Requisição HTTP/GET com Postman para obter palavras similares

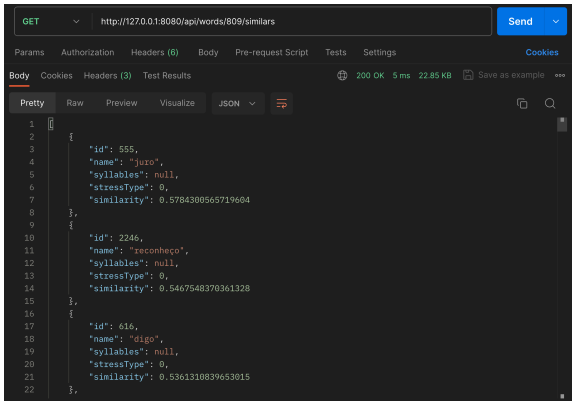
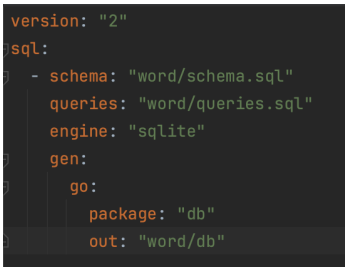


Figure 10. Arquivo de definição sqlc.yml



3. Um arquivo SQL contendo as consultas com alguns comentários que instruem o SQLc a gerar código (Figura 11).

Com o comando *sqlc generate*, o SQLc gera todo código necessário para acessar o banco de dados.

Para ordenar a lista de rimas, utilizou-se como critério a distância de *Levenshtein*, e o número de sílabas. Dessa forma, ocupam as primeiras posições na lista, as palavras que são mais parecidas com a palavra pesquisada.

4.7. Desenvolvimento da aplicação cliente

Para o desenvolvimento a aplicação cliente, empregou-se o framework Svelte, integrado com HTML, Tailwind CSS e TypeScript, e adotou-se o Skeleton.dev[8] como kit de interface do usuário (UI). A aplicação comunica-se com a API do servidor através requisições criadas com a função *fetch*. Um trecho de código que utiliza o comando pode ser visto na Figura 12. A aplicação permite que o usuário escreva a letra da música na lateral esquerda, enquanto a parte central da tela vai sendo atualizada. A Figura 13 representa a ferramenta no modo de visualização de palavras. Nesse modo, as palavras de cada linha vão sendo exibidas de modo que o usuário pode clicar para encontrar palavras relacionadas ou rimas (de acordo com o modo de busca selecionado no canto direito). No início de cada linha, é exibido um contador de sílabas. O resultado da busca é exibido na lateral direita.

A Figura 14 mostra a ferramenta no modo visualização métrica. A

[8] `http://skeleton.dev`

Figure 11. Definição do arquivo de *queries* do SQLc

```
-- name: FindWordById :one
SELECT *
FROM words
WHERE id = ?
LIMIT 1;
-- name: FindWordByName :one
SELECT *
FROM words
WHERE name = ?
LIMIT 1;
-- name: FindSimilarWordsById :many
SELECT w.id, w.name, sw.similarity
FROM similar_words sw
         JOIN words w on w.id = sw.word_similar_id
WHERE sw.word_id = ?
ORDER BY sw.similarity DESC;
-- name: FindAllWords :many
SELECT *
FROM words
```

Figure 12. Método responsável por carregar as palavras similares do servidor

```
async function loadSimilarWords(word: Word) {
    if (!word || !browser){
        return;
    }
    const response = await fetch(`/api/words/${word.id}/similars`, {
        method: "GET"
    })
    similarWords = '';
    textArea.scrollTop = 0;
    const words = await response.json() as Word[];

    similarWords = words.map(w => w.name).join('\n');
}
```

ferramenta exibe a separação silábica de cada linha, destacando cada sílaba tônica. As sílabas de cada linha ficam dispostas em colunas distintas.

4.8. Resultados Obtidos

Como forma de validar os algoritmos que foram produzidos durante este trabalho, foram realizados testes de integração. Mas optou-se por fazer um teste ainda maior, criando para isso uma base de referência a partir do dicionário de separação silábica do Portal da Língua Portuguesa [PLB] e comparando com a saída dos algoritmos que foram implementados. Essas palavras foram obtidas com *web crawling*. A saída do teste pode ser observada na Figura 15 e a síntese desses resultados pode ser observado na Tabela 1. A única falha produzida pelos algoritmos propostos foi com a palavra *baby* que é um estrangeirismo. O resultado deste teste sinaliza que os algoritmos têm um bom nível de acerto.

Apesar do resultado positivo, a língua portuguesa é repleta de exceções e os algo-

Figure 13. Interface gráfica da ferramenta no modo de palavras.

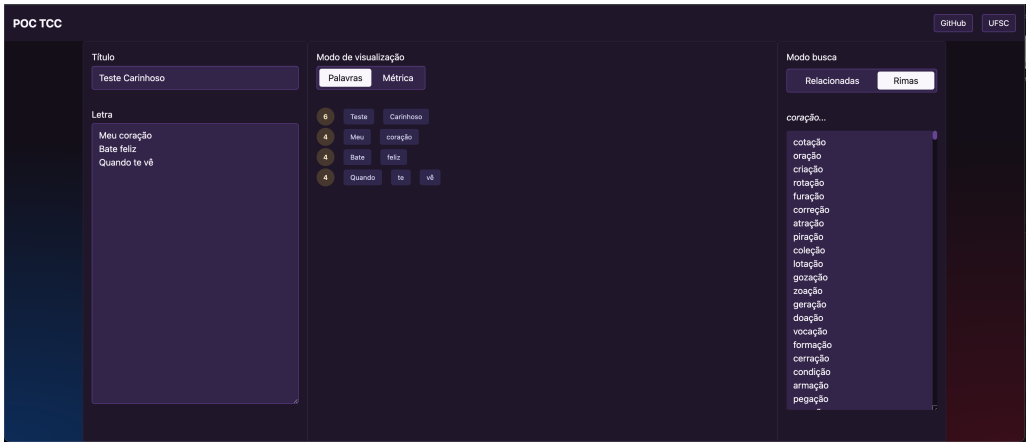
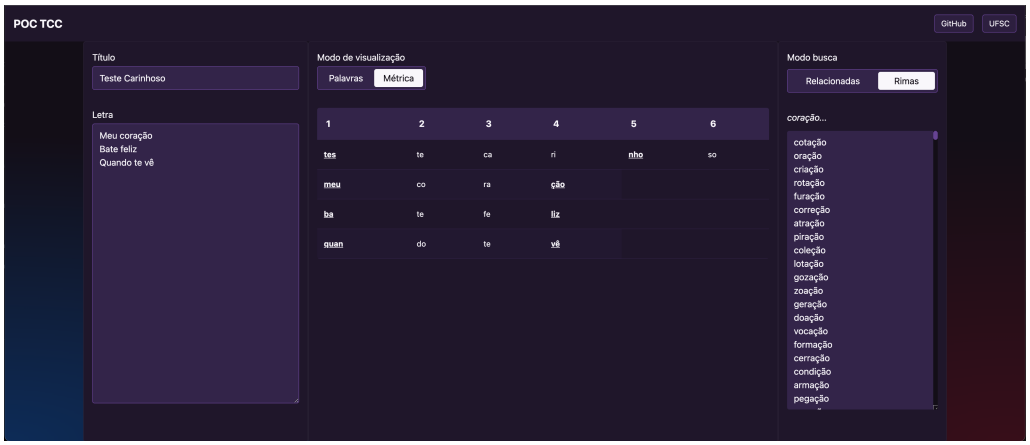


Figure 14. Interface da ferramenta no modo métrica.



ritmos desenvolvidos não são infalíveis. Por exemplo, durante o desenvolvimento foram detectados problemas na separação das palavras “sub-li-nhar” e “bo-cai-u-va”. Mas como essas são palavras raras em letras musicais e, mesmo com a separação incorreta, o algoritmo identifica corretamente a tonicidade dessas palavras. Optou-se então por não alterar o algoritmo de separação para contemplar esses casos.

A validação da acurácia da funcionalidade da busca por palavras relacionadas pode ser subjetiva. Mas foi possível observar que a ferramenta traz alguns bons resultados, especialmente quando a palavra é uma palavra comum no *corpus*, como é o caso da palavra *amor*. Algumas palavras que a ferramenta relaciona com a palavra *amor* são mostradas na Figura 16.

É importante salientar que, ainda que uma palavra encontrada aparentemente não se relacione com a palavra pesquisada, ainda pode servir de inspiração para criação de uma letra de música. A ferramenta proposta atendeu a todos os requisitos funcionais e não-funcionais estabelecidos no inicio deste projeto, bem como foram atendidos todos os objetivos propostos neste trabalho.

Figure 15. Resultado do teste de mil palavras

```
=== RUN   TestTopWords
    top_words_test.go:32: PLB expected 'mas, we got mas
    top_words_test.go:32: PLB expected sa-u-'da-de, we got sau-'da-de
    top_words_test.go:32: PLB expected 'ao, we got ao
    top_words_test.go:32: PLB expected 'por-que, we got por-'que
    top_words_test.go:32: PLB expected 'gar-çom, we got gar-'çom
    top_words_test.go:32: PLB expected 'ba-by, we got 'baby
    top_words_test.go:32: PLB expected 'ba-tom, we got ba-'tom
--- FAIL: TestTopWords (0.01s)
```

Table 1. Síntese dos resultados de validação

Descrição	Quantidade
Total de palavras validadas	1000
Total de divergências	7
Erros da implementação	1
Erros da base de referência (PLB)	6

4.9. Testes de desempenho

Foi realizado *benchmark* dos algoritmos desenvolvidos:. A máquina do ambiente de teste foi um Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz com 32gb de memória RAM. O teste consistiu em medir o tempo de execução do algoritmo de silabificação mais o tempo de determinação da sílaba. O resultado é mostrado na Tabela 2.

Table 2. Resultados de desempenho

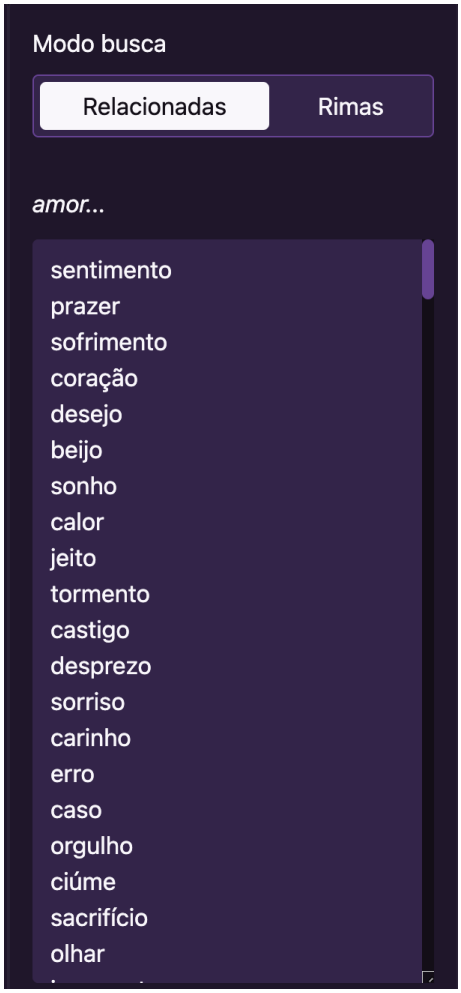
Descrição	Tempo (ms)
Tempo total de execução / 1000 palavras	4,707918
Tempo total de execução / palavra	$\approx 0,00471$

5. Conclusão

Neste trabalho realizou-se o desenvolvimento de uma aplicação web para auxiliar na criação de letras de músicas. A ferramenta permite que o usuário pesquise por rimas e palavras similares, além de ter um modo em que usuário pode visualizar a divisão silábica de todas as linhas com as sílabas tônicas em destaque.

Com o desenvolvimento da ferramenta concluído com sucesso, tem-se que esse o trabalho cumpriu com todos os objetivos propostos desde da sua concepção, passando pela criação de um *corpus* linguístico de letras de música através de *web crawling*, pré-processamento do *corpus*, treinamento de modelos de inteligência artificial utilizando algoritmos de *embedding*, desenvolvimento de algoritmos de divisão silábica e determinação da sílaba tônica, criação de uma API para o back-end, desenvolvimento da interface gráfica da ferramenta, bem como a validação da ferramenta finalizada com todas as funcionalidades que foram pretendidas.

Figure 16. Palavras relacionadas à palavra *amor*



Espera-se que esse trabalho possa contribuir e inspirar a realização de outros trabalhos acadêmicos relacionados à musica e a língua portuguesa. Todo o código-fonte produzido foi disponibilizado de forma livre.

5.1. Trabalhos Futuros

São sugestões para a continuidade do projeto:

- Tornar o *front-end* da ferramenta mais acessível a dispositivos móveis;
- Otimizar e formalizar os algoritmos de divisão silábica e de determinação da sílaba tônica;
- Adicionar paginação nas listas de palavras sugeridas;
- Criar novos modelos a partir de letras de outros gêneros musicais e permitir que o usuário alterne entre os modelos;
- Adicionar funcionalidade para persistir/recuperar letras;
- Adicionar funcionalidade para buscar rimas de acordo com um tipo específico de rima;
- Adicionar funcionalidade para exportar/compartilhar a letra;
- Adicionar funcionalidade para adicionar/remover as relações existentes entre as palavras;

- Adicionar funcionalidade para geração de pequenos textos com inteligência artificial como, por exemplo, frases e títulos.

References

Mittmann, A. (2016). *Escansão Automática de Versos em Português*. PhD thesis, Universidade Federal de Santa Catarina.

Monte, A., Ribeiro, D., Neto, N., Cruz, R., and Klautau, A. (2011). A rule-based syllabification algorithm with stress determination for brazilian portuguese natural language processing. In *International Congress of Phonetic Sciences*.

Németh, L. et al. (2023). Hunspell.

Pattison, P. (2009). *Writing Better Lyrics*. Writers Digest Books, Cincinnati, OH.

Pattison, P. (2014). *Essential Guide to Rhyming*. Berklee Press, Boston, MA.

PLB. Portal da língua portuguesa.

Rocha, W. and Neto, N. (2013). Implementação de um separador silábico gratuito baseado em regras linguísticas para o português brasileiro (the implementation of a free syllabification algorithm based on linguistic rules for brazilian portuguese) [in portuguese]. In *Brazilian Symposium in Information and Human Language Technology*.

Silva, D., Braga, D., and Resende, F. (2008). Separação das sílabas e determinação da tonicidade no português brasileiro. In *XXVI Simpósio Brasileiro de Telecomunicações*.

Sommerville, I. (2011). Software engineering (ed.). *America: Pearson Education Inc*.

Wazlawick, R. (2016). *Historia da Computacao*. Elsevier.