




Desenvolvimento de um corretor ortográfico

Developing a spell checker

Leonardo Carneiro de Araújo  *¹, Aline de Lima Benevides  †²
e João Pedro H. Sansão  ‡¹

¹Universidade Federal de São João del Rei, Brasil.

²Universidade de São Paulo, Brasil.

Resumo

Corretores ortográficos são ferramentas computacionais utilizadas cotidianamente na redação de textos e de mensagens ou, de forma oculta, na busca por informação e mineração de dados. Diante de sua relevância, o presente trabalho apresenta o percurso histórico de desenvolvimento dos corretores ortográficos e ilustra como, de forma simples, é possível criar um corretor ortográfico eficiente a partir da proposta de Norvig (2007). Salientam-se, também, algumas ferramentas e as estratégias empregadas na elaboração de corretores, como a remoção de afixos e a computação de n-gramas. Explicita-se, ainda, a implementação do corretor ortográfico de Norvig (2007) e verifica-se seu desempenho na tarefa de correção automática em diferentes conjuntos de dados de erros ortográficos. Expõe-se, também, uma comparação na performance de um corretor ortográfico que se vale da remoção de afixos em relação a um corretor que não adota semelhante estratégia.

Palavras-chave : Corretor Ortográfico. Ortografia. Afixos. Linguística Computacional.

Abstract

Spell checkers are ubiquitous computational tools that help us in correctly writing texts or messages and improving information inquiry and data mining. The present work presents the history of development of spell checkers and illustrates how, in a simple way, it is possible to create an efficient spell checker from Norvig's proposal. We also highlight some tools and how they are used in the development of spell checkers, such as affix removal and n-gram computation. Moreover, we present an implementation of Norvig's spell checker and its performance in automatic correction for different spelling error data sets. Also, in a comparison of spell checkers performance, we expose that it is worth removing affixes.

Keywords: Spell Checker. Spelling. Orthography. Affixes. Computational Linguistics.

1 Introdução

Corretores ortográficos são ferramentas conhecidas por usuários de computadores ou de celulares por terem um papel importante na elaboração de textos. Eles são utilizados para detectar erros ortográficos e sugerir correções - palavras ortográficas semelhantes à palavra escrita pelo usuário, que provavelmente corresponde àquela pretendida por quem escreve, sem qualquer erro ortográfico. Ainda que esta seja a sua função mais popular, sua utilidade vai além da mera sugestão de palavras. A tarefa de um corretor ortográfico é, primariamente, o reconhecimento e a classificação de padrões. Eles são muito úteis em tarefas de recuperação de informação e em ferramentas de busca (BRILL; CUCERZAN, 2005; SHAZEER, 2007; CAMERON; WILLIAMS, 2013), uma vez que tanto a pergunta de um usuário quanto a informação armazenada podem estar grafadas de maneira diversa da sequência gráfemica da palavra alvo, sendo necessário encontrar padrões que as aproximem, sem a necessidade de serem exatamente iguais.

Diante da relevância dos corretores ortográficos para os estudos computacionais, o presente trabalho apresenta uma breve revisão histórica a respeito dos corretores ortográficos, em especial, dos corretores de palavra isolada. Para tanto, expõem-se quais mecanismos embasam cada um desses corretores, ilustram-se os conceitos de implementação, bem como os problemas com os quais cada qual se depara. A implementação de um corretor, aqui exposta, busca revisar os exemplos apresentados por Norvig (2007) e por Robbins e Beebe (2005), a fim de verificar o desempenho do corretor.

Texto livre
Linguagem e Tecnologia

DOI: 10.35699/1983-3652.2021.26469

Autor Correspondente
Leonardo Araújo

Editado por
Daniervelin Pereira

Recebido em
27 de Novembro de 2020
Aceito em
4 de Janeiro de 2021
Publicado em
9 de Fevereiro de 2021

Essa obra tem a licença
"CC BY 4.0".



*Email: leolca@ufs.ju.edu.br

†Email: benevides.aline12@gmail.com

‡Email: joao@ufs.ju.edu.br

O presente trabalho, portanto, tem cunho eminentemente didático, buscando apresentar uma retrospectiva histórica e conceitos simples para a implementação de corretores ortográficos. Para tanto, serão incorporados ao texto pequenos trechos de códigos que ilustram a aplicação de certos conceitos e abordagens para uma melhor compreensão e contextualização do tema. O leitor que não estiver habituado poderá lê-los apenas superficialmente, sem maiores prejuízos ao entendimento dos principais conceitos e da retrospectiva apresentada.

2 O desenvolvimento histórico dos corretores ortográficos

Anteriormente ao surgimento dos corretores ortográficos, as ferramentas de busca desenvolvidas tinham a preocupação de encontrar registros (GLANTZ, 1956; DAVIDSON, 1962). Muitas vezes, para se encontrar um determinado registro em um banco de dados, era necessário fornecer uma lista de registros que se aproximavam da busca inquirida, tendo em vista que o registro poderia ter sido inserido com erros (palavras grafadas erroneamente) na base de dados. Um exemplo consiste na busca por um nome em uma lista de passageiros, na qual frequentemente se encontram registros com erros ortográficos (DAVIDSON, 1962). Se o objetivo for encontrar o passageiro *Nicholas*, tem-se que buscar por *Nicholas*, *Nicolas*, *Niccolas*, *Nicollas*, *Nikolas*, *Nichola*, dentre outros. Uma ferramenta de busca com um bom desempenho deve ser capaz de sugerir diferentes grafias e erros possíveis de ocorrer, seja no banco de dados, seja na digitação do usuário que realiza a busca.

O primeiro algoritmo foi disseminado nos anos 60, sendo proposto por Robert C. Russell e Margaret King Odell, chamado *Soundex* (RUSSELL, 1918, 1922; ODELL, 1956; KNUTH, 1973). O propósito desse algoritmo é codificar (quasi-)homófonos por meio de um mesmo código, isto é, simplifica-se a busca por registros que tenham representação fonética similar, quer representem nomes ou palavras, quer estejam grafados correta ou erroneamente. Embora o *Soundex* busque focar nas similaridades fonéticas, ele realiza apenas um mapeamento dos grafemas em um código composto por letras e números. A suposição do *Soundex* é de que as consoantes são mais importantes do que as vogais. Dessa forma, o algoritmo aglutina as vogais em um único grupo e as consoantes em seis, considerando, para tal, o modo de articulação. Outro aspecto destacado pelo *Soundex* é a importância da primeira letra, mantendo-a como o primeiro símbolo do código de cada palavra. A preservação da primeira letra deve-se ao fato de que, em geral, o usuário confere uma maior atenção às primeiras letras da palavra do que às demais. Por exemplo, em uma busca por nomes de pessoas, um código do *Soundex* neutraliza as diferentes grafias de um mesmo nome: E540¹ representa *Emily*, *Emely* e *Emilee*; D552², *Dominic*, *Dominick* e *Dominik*; e N242³, *Nicholas*, *Nicolas* e *Nikolas*. Note como esse sistema de codificação auxilia a encontrar um registro, mesmo que esteja grafado de forma distinta no banco de dados. É importante ressaltar, ainda, que o *Soundex* foi elaborado para o inglês, de forma que não se pode esperar desempenho igual em outra língua, sem que sejam realizadas as devidas adaptações (BEIDER; MORSE, 2008, 2010; PILAR ANGELES; ESPINO-GAMEZ; GIL-MONCADA, 2015).

Apesar de ter se tornado muito popular por auxiliar a busca de registros, os proponentes do *Soundex* não tinham o propósito de que ele atuasse como um corretor ortográfico. Uma das primeiras ferramentas a ter essa finalidade foi o *Spell* do sistema UNIX⁴. Ele foi desenvolvido por Stephen Curtis Johnson em 1975 e aperfeiçoado por McIlroy (1982). A ideia inicial de Johnson era bem simples: o verificador ortográfico deveria apenas buscar as palavras do texto em uma lista, previamente confeccionada, de palavras ortograficamente corretas (um dicionário). Na implementação do *Spell*, Johnson utilizou uma lista de palavras do Corpus Brown (KUČERA; FRANCIS, 1967). Essa proposta pode ser facilmente implementada por meio da criação de uma lista de palavras e de sua

1 Para formar o código E540, a primeira letra é mantida, as vogais são removidas, $m \rightarrow 5$, $l \rightarrow 4$ e, por fim, acrescenta-se um 0 ao final para compor o código com 3 números.

2 O código D552 também é gerado pelo *Soundex*. Mantém-se a primeira letra, remove-se as vogais, $m, n \rightarrow 5$ e $c, k \rightarrow 2$. Se necessário, trunca o código gerado para que este tenha comprimento 4 (a primeira letra seguida de 3 números).

3 De forma geral, podemos descrever o algoritmo *Soundex* como a composição dos seguintes passos: i) mantém a primeira letra; ii) remove todas as vogais; iii) realiza os seguintes mapeamentos: $b, f, p, v \rightarrow 1$; $c, g, j, k, q, s, x, z \rightarrow 2$; $d, t \rightarrow 3$; $l \rightarrow 4$; $m, n \rightarrow 5$ e $r \rightarrow 6$; iv) criar um código de comprimento 4 (composto por uma letra e 3 números), truncando a sequência, se maior, ou completando com zeros, se menor.

4 Aparentemente, o primeiro corretor ortográfico prático foi o *Spell* desenvolvido por Ralph Gorin em 1971 (GORIN, 1974; PETERSON, 1980).

comparação com um dicionário. Para criar um verificador ortográfico simples, semelhante ao *Spell*, pode-se utilizar comandos do *shell*⁵. A fim de realizar as verificações, é necessário criar um arquivo de dicionário, contendo uma lista de palavras ortograficamente corretas (ou utilizar uma lista de algum dicionário, como o arquivo `/usr/share/dict/american-english`⁶ no Linux). O verificador ortográfico proposto mostra-se vantajoso por ser de fácil implementação em qualquer língua, visto que requer apenas uma lista de palavras na língua desejada. Fazem-se necessários, para tanto, apenas três comandos do *shell*: *sed* (editor de fluxo para filtrar e transformar o texto), *sort* (ordenador de linhas de texto) e *comm* (comparador, linha a linha, de dois arquivos). Como entrada, utilizam-se um arquivo de texto, aqui chamado de `texto.txt`, e um arquivo de dicionário (ordenado) com uma palavra por linha, denominado de `dicionario.txt`.

```
sed -f spell < texto.txt |      # lê o arquivo de entrada pelo comando sed,
executa as edições prescritas no arquivo spell (veja abaixo) e
direciona a saída para o próximo comando
grep -v "^$" |                 # remove as linhas vazias
sort -u |                       # ordena as linhas por ocorrências únicas,
suprimindo as repetições
comm -13 dicionario.txt -      # compara dois arquivos, linha a linha -
neste caso compara a lista de palavras do arquivo texto.txt com as
palavras no dicionario.txt (o traço representa a saída padrão)
```

No exemplo apresentado, o comando *sed* foi utilizado para realizar três edições no texto: 1) eliminar caracteres não alfanuméricos, substituindo-os por espaço em branco (incluindo também, explicitamente, apóstrofe e traço como caracteres a serem aceitos); 2) converter maiúsculas em minúsculas (supõe-se que, na língua em questão, não haja diferenciação entre maiúsculas e minúsculas); e, por fim, 3) substituir espaços em branco por quebras de linha, deixando, assim, uma palavra por linha, forma apropriada para o comando seguinte. Essas três operações de edição são descritas abaixo no *script spell* que será executado pelo *sed*:

```
# spell sed script
s/[^[a-z0-9_-'-]/ /g      # substitui tudo que não for alfanumérico, apóstrofe
                           ou traço por espaço em branco
s/(.*)/\\L\\1/            # converte maiúsculas em minúsculas
s/\\s+/\\n/g               # substitui um ou mais espaços por quebra de linha,
                           de forma que cada palavra esteja disposta sozinha em uma linha
```

Como podem ter sido geradas linhas vazias ao final das edições realizadas com o *sed*, emprega-se o comando *grep* para removê-las. Em seguida, o *sort* irá ordenar as linhas e listar apenas as ocorrências únicas. Para verificar quais palavras não se encontram no dicionário, utiliza-se a função *comm*. Caso o dicionário contenha palavras com maiúsculas, deve-se também aplicar a conversão de maiúsculas em minúsculas sobre ele, uniformizando, assim, os dados antes de utilizar o comando *comm*⁷. Para interligar comandos diferentes, utiliza-se o *pipe* (`|`), cuja função é redirecionar a saída padrão de um programa para a entrada padrão do programa subsequente. A saída final, a ser visualizada pelo usuário, será a lista de palavras, as quais não foram encontradas no dicionário. Com essa lista em mãos, torna-se mais fácil realizar a busca no texto pelas palavras que contêm erros ortográficos.

Ao aplicar os *scripts* descritos acima no dicionário padrão do Linux e no *Soneto 18* de William Shakespeare, tem-se o seguinte resultado:

```
dimm'd
grow'st
ow'st
untrimm'd
wander'st
```

5 *Unix shell*, ou simplesmente *shell*, é o interpretador de comandos padrão dos sistemas Unix. O *shell* fornece uma interface de usuário em modo texto, a partir da qual o usuário é capaz de executar chamadas a programas, criar *scripts* e receber o resultado em texto das suas requisições.

6 Os arquivos de dicionários podem estar em local diferente, dependendo da distribuição.

7 Caso seja necessário converter o dicionário de maiúsculas em minúsculas e ordená-lo, bastaria substituir a linha do último comando da seguinte forma: `comm -13 <(sed 's/(.*)/\\L\\1/' < /usr/share/dict/american-english | sort)-`.

Para facilitar a localização dessas palavras, é interessante salvar o *script* no arquivo `spell.sh`, o que permite que seu resultado seja utilizado em conjunto com o `grep` para determinar em quais linhas aparecem as palavras desconhecidas. Ilustra-se, a seguir, esse resultado:

```
$ ./spell.sh | while read line; do grep -n "$line" texto.txt; done
6:And often is his gold complexion dimm'd;
12:When in eternal lines to time thou grow'st:
10:Nor lose possession of that fair thou ow'st;
12:When in eternal lines to time thou grow'st:
8:By chance or nature's changing course untrimm'd;
11:Nor shall death brag thou wander'st in his shade,
```

A mera comparação de listas (dicionário e corpus em análise) não é suficiente para contemplar diversos casos de derivação ou de afixação, que, por terem múltiplas possibilidades, podem não estar na lista do dicionário. Como uma forma de sanar tais casos e, consequentemente, de aumentar a abrangência do dicionário, Johnson optou por remover sufixos e prefixos utilizando regras simples. Basicamente, se uma determinada palavra, por exemplo, *usefulness*, não estiver no dicionário, ao remover o sufixo *-ness*, gerando a palavra *useful*, aumenta-se a possibilidade de que esta seja encontrada no dicionário. Essa abordagem, entretanto, faz com que alguns erros sejam aceitos, como a sequência *goed* seria aceita ao se encontrar a palavra *go* no dicionário. Observe que a funcionalidade do *Spell* restringia-se, até o momento, a apenas localizar erros ortográficos.

Uma outra ferramenta, o *Grope*, foi desenvolvida para o sistema UNIX com a finalidade de realizar a seguinte tarefa: fornecer sugestões de edição, de forma interativa, para os erros encontrados pelo *Spell* (TAYLOR, 1981 apud KUKICH, 1992b), a partir da busca de palavras com a menor distância de edição (TAYLOR, 1981 apud CASTRO, 2012). Pouco depois, em 1982, o *Typo* foi introduzido ao sistema UNIX (v3), por Morris e Cherry (1975) (MCMAHON; CHERRY; MORRIS, 1978; MAHONEY, 1998), com o propósito de encontrar erros de datilografia. Para tanto, lançava uso da frequência de ocorrência de bigramas e de trigramas do próprio documento a ser analisado. No inglês, por exemplo, em que há 28 letras ([a-z], espaço e apóstrofe), são possíveis $28^2 = 784$ bigramas e $28^3 = 21.952$ trigramas. Entretanto, apenas uma fração pequena deles realmente ocorre na língua, cerca de 70% dos bigramas e 25% dos trigramas. Se o objeto em análise for um texto muito grande, ou se avaliam-se estatísticas para um corpus da língua, usando posteriormente essa informação no corretor ortográfico, uma lista com menos de 6.000 bi- ou trigramas seria suficiente para o *Typo* averiguar a peculiaridade das sequências encontradas em um texto. Conforme descrito no manual do *Typo*, o programa procura no documento por palavras inusuais, erros de digitação e *hapax legomena*⁸. Para cada *token*, é calculado o seu índice de peculiaridade, que é utilizado para ordená-los em uma lista ao final. O índice de peculiaridade de um trigramma *xyz* é dado por

$$p_{\text{idx}}(xyz) = \frac{\log(f(xy) - 1) + \log(f(yz) - 1)}{2} - \log(f(xyz) - 1), \quad (1)$$

onde $f(xyz)$, $f(xy)$ e $f(yz)$ são as frequências de ocorrência dos tri- e bigramas (MORRIS; CHERRY, 1975; PETERSON, 1980).

Tanto o *Spell* quanto o *Typo* não efetuam correção, nem fornecem uma lista de candidatos para corrigir os possíveis erros. Evidentemente, a correção automática não é, em geral, desejável, uma vez que poderá criar erros de palavra real⁹ que dificilmente serão detectados de forma automática. Entretanto, um processo de correção interativo é conveniente, tendo se tornado o padrão nos corretores ortográficos.

Church e Gale (1991) propuseram o *Correct*, uma ferramenta complementar ao *Spell*, que gera uma lista de sugestões, ordenadas por probabilidade dos possíveis erros de digitação através de um modelo de canal ruidoso (KERNIGHAN; CHURCH; GALE, 1990; SHANNON, 1948). Neste modelo, supõe-se que, num processo de comunicação, uma sequência recebida é resultante da transmissão de uma mensagem produzida pela fonte através de um canal suscetível a ruído. O ruído presente

⁸ *Hapax legomena* são palavras que aparecem uma única vez no texto.

⁹ Erros de palavra real são erros ortográficos que resultam em uma palavra real, de forma que não são detectados por uma simples busca em dicionário.

no canal poderá distorcer a mensagem recebida. O objetivo, então, é, conhecendo as características desse canal, estimar qual seria a mensagem mais provável que poderia ter sido enviada pela fonte.

Ainda no sistema UNIX, em 1971, foi criado o *IsPELL*, com o objetivo de ser um corretor interativo capaz de sugerir possíveis correções para as palavras grafadas de forma errônea (GORIN; WILLISSON; KUENNING, 1971) a uma distância igual a 1 (um), isto é, considera-se a distância de Damerau-Levenshtein, também conhecida como estratégia do quase erro (em inglês *near miss strategy*, veja mais na Seção 3.2). O *IsPELL* vale-se, ainda, de uma lista de regras de afijos para abarcar uma quantidade maior de palavras, sem a necessidade de utilizar um dicionário extenso em que aparecem todas as variações criadas por processos de afixação.

A partir de 1998, foi desenvolvido por Kevin Atkinson o *GNU AsPELL* (ATKINSON, 1998) pertencente ao Projeto GNU. Ele surgiu como uma alternativa ao *IsPELL*, fornecendo suporte a UTF-8¹⁰ e a múltiplos dicionários. Por utilizar o algoritmo *Metaphone*¹¹ criado por Lawrence Philips (PHILIPS, 1990, 2000) e a estratégia do quase-erro do *IsPELL*, o *AsPELL* converte as palavras em equivalentes sonoros, encontra todas as palavras que estão a uma ou a duas distâncias de edição no domínio dos *metaphones* (ou seja, palavras que possuem sonoridade próxima) e, também, busca os candidatos com base na edição dos caracteres que compõem a sequência, assim como o *IsPELL*. A todos os candidatos gerados, é atribuído um valor probabilístico, que será utilizado para ordenar as sugestões.

Com uma abordagem semelhante ao *IsPELL*, foi criado o *MySPELL* em 2011, utilizando-se também da lista de palavras criada por Kevin Atkinson. As sugestões fornecidas pelo corretor ortográfico passaram a contar com sugestões provenientes de tabelas de substituição e de mecanismos de pontuação de n-gramas. O *MySPELL* foi incorporado ao OpenOffice.org e, posteriormente, deu origem ao *HunSPELL*, sendo este hoje usado no Mozilla Firefox, no Mozilla Thunderbird, no Google Chrome, no macOS, dentre outros. O *HunSPELL* incorporou, ainda, análise morfológica e informação de dicionários de pronúncia. Alguns verificadores ortográficos mais recentes valem-se de informações contextuais e de análises gramaticais para efetuar correções (GOLDING; SCHABES, 1996; VERBERNE, 2002; GUPTA, 2020). Entretanto, para realizar tal tarefa, é preciso usar dados de um corpus muito extenso, armazenar um volume muito grande de dados referente às frequências de ocorrência de uni-, bi- e trigramas de palavras da língua, bem como requerem um alto custo computacional.

Para uma revisão mais abrangente de algumas abordagens e técnicas utilizadas para criar corretores ortográficos, veja Kukich (1992b), Verberne (2002) e Mitton (2010).

3 Desenvolvimento de um corretor ortográfico

Ao abordar o desenvolvimento de um corretor ortográfico, devemos distinguir duas tarefas distintas: verificação e correção. A verificação consiste em analisar uma sequência de caracteres e sinalizá-la como uma palavra válida no léxico ou não. A tarefa de correção, por sua vez, irá propor candidatos para as sequências não encontradas no léxico. Tais candidatos serão selecionados dentre as palavras do léxico e, em alguns casos, o corretor realizará automaticamente a substituição da sequência malformada pelo candidato escolhido.

A tarefa de verificação pode ser implementada por uma busca no dicionário a partir de uma correspondência exata entre sequências, a investigada e as palavras do dicionário, assumindo, assim, que o léxico ou o corpus utilizado é morfológicamente completo. Podemos, também, assumir a incompletude morfológica e criar variações morfológicas de uma sequência, verificando se alguma delas encontra-se no dicionário. Outra forma de verificação consiste em realizar uma análise em n-gramas das sequências e calcular a probabilidade ou a peculiaridade de uma determinada sequência, o que é utilizado para decidir a aceitação ou não da sequência (PETERSON, 1980; ZAMORA; POLLOCK; ZAMORA, 1981). O método de n-gramas costuma também ser utilizado conjuntamente com outras técnicas, tais como: busca em dicionário, análise morfológica e representação fonética, para atingir uma melhor performance na tarefa de detecção. Autômatos finitos (*Finite-state automata*, FSA)

¹⁰ UTF-8 é uma forma de codificação binária, de comprimento variável, do Unicode (padrão universal utilizado para representar símbolos e caracteres).

¹¹ O algoritmo *Metaphone*, de forma similar ao *Soundex*, é utilizado para indexação de palavras pela sua pronúncia em inglês. A versão original foi melhorada, sendo substituída pelo *Double Metaphone* (2000) e, mais recentemente, pelo *Metaphone 3* (2009).

também podem ser utilizados na tarefa de verificação e correção. Os FSA começaram a ser utilizados na tarefa de busca por padrões nos anos 60, sendo aplicados em corretores ortográficos apenas mais tarde (LUCCHESI; KOWALTOWSKI, 1993; OFLAZER; GÜZEY, 1994; OFLAZER, 1996).

Os erros ortográficos podem ser classificados a partir dos seguintes domínios: ortográficos, cognitivos e fonéticos. A tarefa de correção ortográfica (ou geração de candidatos) pode utilizar diferentes técnicas para propor ou selecionar candidatos. A distância de mínima edição é comumente usada e será exposta na Seção 3.2. Ela é utilizada no algoritmo *SPROFF* (DUNLAVEY, 1981), assim como no *Grope* (TAYLOR, 1981 apud CASTRO, 2012) e será empregada, neste trabalho, para o desenvolvimento de um corretor ortográfico. Outra abordagem para buscar correções é a semelhança entre chaves. As sequências de caracteres são mapeadas em chaves e buscam-se chaves semelhantes considerando-se a posição, os elementos constituintes e o ordenamento na chave. Exemplos já mencionados anteriormente são o *Soundex* e o *Metaphone*. Ainda, utilizam-se abordagens baseadas em regras de geração de candidatos (YANNAKOUDAKIS; FAWTHROP, 1983a,b), n-gramas (KIM; SHAW-TAYLOR, 1992, 1994), probabilidade (KERNIGHAN; CHURCH; GALE, 1990; JURAFSKY; MARTIN, 2008; INGELS, 1996) e, recentemente, redes neurais artificiais (DEFFNER; EDER; GEIGER, 1990; HODGE; AUSTIN, 2002, 2003; YUNUS; MASUM, 2020).

A partir da exposição desses conceitos iniciais, apresenta-se, ao longo desta seção, a implementação de um corretor ortográfico simples e eficiente desenvolvido por Peter Norvig em Python (NORVIG, 2007). Sua implementação servirá de base para uma família de corretores ortográficos elaborados por Araujo (2020), cujos princípios básicos serão apresentados neste trabalho. Para tanto, expõem-se, na Seção 3.1, as ideias básicas do corretor ortográfico proposto por Norvig (2007) e, na Seção 3.2, revisita-se a definição de distância de mínima edição de Damerau-Levenshteine e vê-se como ela pode ser incorporada ao corretor ortográfico.

3.1 Organização e extração de frequência do corpus

O primeiro passo para o desenvolvimento e para o funcionamento de qualquer corretor ortográfico é definir quais são os marcadores que separam as palavras e o que são palavras. Em geral, na computação, define-se o espaço em branco como uma maneira útil de delimitar uma palavra da outra. Contudo, apenas esse mecanismo não é suficiente para definir o que é uma palavra, visto que se deve definir também se apenas os caracteres de [a-z] compõem palavras; ou se algarismos, apóstrofes e traços também fazem parte delas; ou ainda, se é preciso fazer diferenciação entre maiúsculas e minúsculas; etc. Essa delimitação baseia-se em decisões como considerar números e datas (1945), acrônimos (Gestapo¹² e Sudene¹³) e *initialisms*¹⁴ (OTAN¹⁵ e IBM¹⁶), palavras com hifens (pé-de-moleque), palavras compostas separadas por espaço (Cabo Verde) e palavras separadas por apóstrofes (pingo d'água) como palavras. Essas premissas podem variar de uma língua para outra e não se espera que qualquer abordagem adotada seja capaz de resolver todos os problemas.

O exemplo da Seção 2 considera que palavras podem ser compostas por letras, números, apóstrofes e traços, tendo como língua alvo o inglês. Norvig (2007) define uma função para separar o texto em palavras e, para tanto, faz uso de expressão regular¹⁷. Adapta-se sua ideia, utilizando a expressão `\w+(?:['-]\w+)*`, ao invés do simples `\w+` usado por Norvig (2007), para poder abarcar também palavras compostas com hifens e apóstrofes. A função, que separa as palavras de um texto e retorna uma lista de palavras, pode ser implementada da seguinte forma:

```
def words(text):
    import re
    return re.findall(r"\w+(?:['-]\w+)*", text.lower())
```

¹² Gestapo (*Geheime Staatspolizei*).

¹³ Sudene (Superintendência do Desenvolvimento do Nordeste).

¹⁴ *Initialism* é um acrônimo constituído apenas pelas iniciais.

¹⁵ OTAN (Organização do Tratado do Atlântico Norte).

¹⁶ IBM (*International Business Machines*).

¹⁷ Expressões regulares (ou *regex*) são uma forma concisa e flexível de representar padrões em sequências de caracteres. As expressões regulares são expressas através de uma linguagem formal, sendo interpretadas em um programa capaz de interpretá-las e realizar a busca em texto de padrões que são representados por determinada expressão.

A função `words`, ainda, converte todas as palavras em minúsculas, bem como remove qualquer espaço em branco no início ou no final das sequências de caracteres em cada linha.

Após a definição da função, carrega-se o dicionário contido no arquivo `texto.dicionario.txt` para um objeto da classe `set`, uma vez que este é uma tabela de dispersão (tabela *hash*)¹⁸, o que fará a consulta ser bem mais eficiente do que em uma lista.

```
dictionary = set(line.strip().lower() for line in open('dicionario.txt'))
```

Para verificar se uma palavra do texto está ortograficamente correta, averigua-se se ela encontra-se no dicionário. A função `unknown`, exposta abaixo, lista apenas aquelas que não foram encontradas.

```
def unknown(words, mydic):  
    return set(w for w in words if w not in mydic)
```

Em sequência, basta indicar o texto a ser lido para que a verificação ortográfica seja realizada. Neste caso, verifica-se o arquivo `texto.txt` através das funções `words` e `unknown`:

```
with open('texto.txt') as f:  
    print(unknown(words(f.read()), dictionary))
```

Como resposta, tem-se um conjunto de palavras que não foram encontradas no dicionário.

Ao utilizar o dicionário padrão do Linux e o texto do *Soneto 18* de William Shakespeare, obtém-se o seguinte resultado:

```
{"grow'st", "dimm'd", "ow'st", "wander'st", "untrimm'd"}
```

Norvig (2007) utiliza edições simples (veja Seção 3.2) sobre uma sequência de caracteres para gerar candidatos para correção da sequência errônea. Para escolher o melhor candidato, serve-se da frequência de ocorrência deles em um corpus grande, escolhendo aquele que for mais provável. Essa ideia não é nova, porém, mostra-se eficiente (PETERSON, 1980). Contabilizar a frequência de ocorrência de palavras em um corpus é uma ação que pode, facilmente, ser realizada em Python, por meio da classe `Counter` do módulo `collections`.

```
from collections import Counter  
corpusfile = "big.txt"  
with open(corpusfile) as f:  
    WORDS = Counter(words(f.read()))
```

A contagem do número de ocorrência das palavras na língua foi realizada a partir do arquivo `big.txt`. Norvig (2007), por sua vez, usou como corpus uma concatenação de trechos de livros do Projeto Gutenberg e uma lista das palavras mais frequentes fornecida pelo *Wiktionary* e pelo *British National Corpus*.

Adicionar todo o conteúdo de um corpus extenso ao dicionário pode não ser a melhor estratégia. Inevitavelmente, um corpus extenso que não tenha sido minuciosamente curado terá erros ortográficos em seu conteúdo. Os erros tipográficos tendem a ser poucos e a não se repetirem, gerando, assim, *tokens* com baixa frequência de ocorrência. Uma melhor abordagem para esse problema é excluir do dicionário aqueles com frequência de ocorrência muito baixa (PETERSON, 1980). A seguir, expõe-se uma função para retornar as palavras do dicionário, de acordo com a frequência de ocorrência desejada (sendo o comportamento padrão retornar os *hapax legomenon*). Com isso, obtém-se um dicionário mais limpo ao remover palavras muito raras, arcaicas, obsoletas e, também, palavras com erros ortográficos.

```
def get_hapaxlegomenon(n=None):  
    if n is None:  
        n = (1, 1)  
    if isinstance(n, int):  
        n = (1, n)  
    if isinstance(n, (tuple, list)) and len(n)==2:  
        return [w for w in WORDS if WORDS[w] < n[1]+1 and WORDS[w] > n[0]-1]
```

¹⁸ Uma tabela de dispersão, ou tabela *hash*, é uma estrutura de armazenamento que associa chaves de pesquisa (obtidas através de uma função *hash*) a valores, de forma que a recuperação de dados na tabela terá, tipicamente, complexidade $O(1)$ (em caso de colisões, no pior caso, a complexidade será $O(n)$).

A função apresentada acima retorna as palavras com determinada faixa de valor de frequência de ocorrência. Se o parâmetro for um valor n , retorna as palavras com frequência máxima igual a n . Se for uma lista ou ênuplo de tamanho 2, retorna as palavras com frequência de ocorrência entre os valores inicial e final dados pelo ênuplo de parâmetro. Para remover as palavras com frequência de ocorrência igual ou menor que 5, basta utilizar a função com parâmetro $n = 5$ e remover os itens na lista por ela retornada, por meio da função exposta abaixo.

```
removelist = get_hapaxlegomenon(5)
for key in removelist:
    del WORDS[key]
```

3.2 Distância de mínima edição

A distância de mínima edição é uma forma empregada pela Linguística Computacional e pela Ciência da Computação para quantificar a similaridade entre duas sequências de caracteres (ao longo do texto, utilizaremos também *strings* para designar uma cadeia de caracteres). As distâncias mais usuais são: distância de Levenshtein (permite inserção, apagamento e substituição de caracteres), distância de Damerau-Levenshtein (além das operações anteriores, permite também a transposição de caracteres adjacentes¹⁹), distância de Hamming (permite apenas substituições), mais longa sequência comum (LCS, *longest common subsequence*) e distância de Jaro e Jaro-Winkler (medida de similaridade entre duas *strings*). Ao considerar que todas as operações têm um mesmo peso, tem-se que as métricas mencionadas são, de fato, simétricas e, portanto, distâncias.

Essas distâncias envolvem 4 categorias de edição de um único caractere: adição, remoção, troca ou transposição de caracteres adjacentes, que, segundo Damerau (1964), correspondem a 80% dos erros ortográficos. Outros estudos apresentam percentuais diferentes. Mitton (1987), por exemplo, observou que 69% dos erros envolvem um único caractere; já Pollock e Zamora (1984) observaram um percentual bem maior, acima de 90%. Tais diferenças podem ser explicadas pelas características dos diferentes corpus em cada um dos estudos: Mitton (1987) utilizou um corpus escrito à mão, contendo textos redigidos por jovens de 15 anos; já Pollock e Zamora (1984) utilizaram textos científicos e acadêmicos. O grau de formalidade e de instrução dos autores dos textos justifica a maior percentagem de erros de apenas um caractere nestes do que naquele ²⁰.

Dada a frequência com que as 4 categorias ocorrem nos textos, a distância de mínima edição mostra-se relevante para a composição dos algoritmos de correção ortográfica. Sua implementação ocorre a partir do pressuposto de que, quando uma determinada *string* não é encontrada no dicionário, esta contém algum erro ortográfico. Para gerar possíveis candidatos almejados pelo escritor e ajudá-lo a corrigir a *string* malformada, utilizam-se as 4 operações básicas, que retornam os vizinhos mais próximos (DAMERAU, 1964), palavras essas que, possivelmente, são as pretendidas pelo escritor. As *strings* só são consideradas candidatas válidas se existirem no dicionário. Caso exista mais de uma candidata válida, é preciso estabelecer algum critério para elencá-las em ordem de preferência: frequência de ocorrência, índice de peculiaridade, distância de mínima edição (várias distâncias são possíveis), semelhanças sonoras, etc., bem como a combinação desses elementos.

Para gerar candidatos a partir de uma única operação de edição (apagamento, transposição, substituição ou inserção), usa-se como base a função `edits1` apresentada por Norvig (2007). Faz-se, entretanto, uma pequena customização para que essa função também seja funcional em línguas que têm outros caracteres, como o português que possui diacríticos.

```
import locale
from icu import LocaleData
language, encoding = locale.getdefaultlocale()
data = LocaleData(language)
alphabet = data.getExemplarSet()
```

¹⁹ Note que a transposição de dois caracteres adjacentes é equivalente a uma inserção e um apagamento. Dessa forma, o mesmo resultado de edição também é possível usando o método de Levenshtein, porém, o custo da operação é diferente, pois em um caso são duas operações, enquanto no outro há apenas uma operação.

²⁰ Infelizmente, não é possível saber qual o tipo de corpus utilizado por Damareau, visto que ele não o especifica em seu estudo.


```
def edits1(word):
    splits      = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes     = [L + R[1:]               for L, R in splits if R]
    transposes  = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces    = [L + c + R[1:]           for L, R in splits if R for c in
                    alphabet]
    inserts     = [L + c + R               for L, R in splits for c in
                    alphabet]
    return set(deletes + transposes + replaces + inserts)
```

A função `edits1` permite gerar todas as *strings* possíveis por uma simples operação de edição (DAMERAU, 1964), quer cada uma delas seja uma palavra real ou não. Tendo em vista que todas as operações possuem o mesmo peso, a distância entre a *string* passada como parâmetro e todas as demais candidatas geradas será a mesma. Como os pesos são iguais, será dispensável calcular a distância entre elas. Entretanto, como será exposto adiante, também é possível adotar uma outra abordagem, em que se considere pesos diferentes para cada uma das operações de edição.

Caso nenhuma palavra válida (isto é, uma palavra real) seja gerada ao realizar apenas uma única edição sobre a *string* de parâmetro, busca-se pelas *strings* geradas a partir de duas edições. Essa tarefa pode ser simplificada a partir da reaplicação da função `edits1` sobre os resultados gerados na primeira chamada. Com isso, a função `edits2` irá gerar candidatos com duas edições (NORVIG, 2007).

```
def edits2(word):
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))
```

A verificação da lista de candidatos válidos é realizada comparando-se as *strings* geradas com o dicionário. Caso um candidato dentre as *strings* de uma única edição seja encontrado, não será necessário gerar os candidatos provenientes de duas edições, reduzindo, dessa maneira, o custo computacional. A função `known` analisa quais *strings* de uma lista estão no dicionário (NORVIG, 2007):

```
def known(words):
    return set(w for w in words if w in WORDS)
```

A fim de que a função retorne os candidatos pela ordem de mínima edição, emprega-se a função `candidates` (NORVIG, 2007):

```
def candidates(word):
    return ( known([word]) or known(edits1(word)) or known(edits2(word)) )
```

Em operações de edição em que se deseja considerar pesos diferentes, deve-se calcular a distância de mínima edição a partir dos pesos definidos. A seguir, expõe-se uma implementação do algoritmo para calcular a distância de Damerau-Levenshtein (JURAFSKY; MARTIN, 2008) entre duas *strings*, na qual é possível especificar pesos diferentes para as operações de edição, através do parâmetro `cost`:

```
def damerau_levenshtein_distance(s1, s2, cost=(1, 1, 1, 1)):
    d = {}
    lenstr1 = len(s1)
    lenstr2 = len(s2)
    subscost, delcost, inscost, transcost = cost
    for i in range(-1, lenstr1+1):
        d[(i, -1)] = i+1
    for j in range(-1, lenstr2+1):
        d[(-1, j)] = j+1
    for i in range(lenstr1):
        for j in range(lenstr2):
            if s1[i] == s2[j]:
                d[(i, j)] = d[(i-1, j-1)]
            else:
                d[(i, j)] = min(
                    d[(i-1, j-1)] + subscost, # substituição
                    d[(i-1, j)] + delcost,   # apagamento
                    d[(i, j-1)] + inscost,   # inserção
                )
```

```

        if i and j and s1[i]==s2[j-1] and s1[i-1] == s2[j]:
            d[(i,j)] = min (d[(i,j)], d[i-2,j-2] + transcost) #
                transposição
    return d[lenstr1-1,lenstr2-1]

```

A função acima considera que os pesos possam ser diferentes de acordo com o tipo de edição, contudo, não leva em consideração os símbolos envolvidos em cada uma das edições. Pode-se supor que, ao digitar um texto, seja mais provável a troca ou a inserção de caracteres próximos (no teclado) do caractere alvo. Para que esse fator seja considerado, deve-se fazer a adaptação na implementação, atribuindo os pesos como funções dos símbolos envolvidos.

Os códigos vistos até esta seção são suficientes para implementar um corretor ortográfico básico, como aquele proposto por Norvig (2007). A implementação aqui utilizada encontra-se disponível em Araujo (2020). Como apresentado na Seção 1, existem ainda outras abordagens que podem ser aplicadas, algumas delas serão expostas nas seções subsequentes.

4 Afixos

Afixos são morfemas que se ligam ao radical de uma palavra, seja no início (prefixo), seja no meio (infixo), seja no final (sufixo), modificando seu significado e, portanto, criando uma nova palavra. Os prefixos e os sufixos podem ser facilmente detectados, visto que estão nas bordas das palavras, tendem a ser produtivos na língua e têm formas mais regulares, como *desfazer*, *desconfortável*, *desrespeito*, *plantação*, *citação*, *ambientação*. Os infixos, por sua vez, por ocorrerem no meio das palavras e terem formas mais variáveis, como *cafezal* e *cafeteira*, são mais difíceis de serem localizados computacionalmente, devido a esse fato não serão aqui considerados.

Os processos de afixação, quer derivacional, quer flexional, em línguas que permitem tais processos, são responsáveis pela ampliação do seu vocabulário, de forma que um dicionário, por maior que ele seja, pode não conter todas as palavras que poderiam ser geradas pelo acréscimo de afixos. Desse modo, uma maneira de estender virtualmente a quantidade de palavras de um dicionário é utilizar regras para detectar afixos e removê-los, preservando o radical da palavra. Verifica-se, então, se o radical encontra-se no dicionário e, em caso afirmativo, considera-se a palavra original como sendo ortograficamente correta.

Robbins e Beebe (2005) propõem um verificador ortográfico em awk que se vale do mecanismo de remoção de afixos descrito acima, assim como é proposto pelo *IsPELL*. O método empregado pelos autores para encontrar prefixos e sufixos é por meio de expressões regulares. A seguir, esboçam-se alguns exemplos de como regras de substituição podem ser criadas:

```

# suffix
's$           # it's -> it
ed$          "" e   # breaded -> bread; flamed -> flame
th$          ""     # tenth -> ten

# prefix
^anti-?      ""     # antiviral -> viral
^pre-?       ""     # predate -> date
^un-?        ""     # unwell -> well

```

As regras de substituição, que são previamente armazenadas em uma estrutura de dados, o dicionário *suffixes*, são aplicadas através da função *stripSuffix*.

```

def stripSuffix(word):
    import re
    words = []
    for regex in suffixes:
        m = re.search(regex, word)
        if m:
            replace = suffixes[regex].split()
            if not replace:
                words.append( word[0:m.start()] )
            else:
                for r in replace:
                    words.append( word[0:m.start()] + r )

```

```

        break
    return words

```

Com a remoção de afixos usando a função acima e um conjunto de regras, pode-se gerar palavras que podem ter originado uma determinada sequência desconhecida.

5 N-gramas

Uma forma de avaliar se uma determinada sequência de caracteres é válida em uma língua, sem utilizar um dicionário, consiste em verificar as características estatísticas de suas subsequências, denominadas de n-gramas. Os n-gramas são sequências contíguas de n itens (neste caso, de caracteres). A frequência de ocorrência de bi- e de trigramas pode ser estimada a partir da análise de um corpus representativo da língua em estudo. Ter conhecimento das sequências grafêmicas (ou fonêmicas) permite determinar se uma certa palavra pertence ou não ao léxico de uma língua e a sua peculiaridade, por exemplo. A Equação (1), apresentada na Seção 2, mostra uma forma de calcular o índice de peculiaridade de um trigramma, informação essa que permite inferir se uma palavra possui erros ortográficos ou não.

Para avaliar a estatística de n-gramas, primeiramente, cria-se uma função para gerar todos os n-gramas de uma determinada palavra:

```

def compute_word_ngrams(word, n):
    from collections import deque
    ngram = deque(maxlen=n)
    ngrams_list = []
    word = '^' + word + '$'
    for c in word:
        ngram.append(c)
        if len(ngram) == n:
            ngrams_list.append(''.join(ngram))
    return ngrams_list

```

Os n-gramas que estão no início ou no final de uma palavra são marcados com os caracteres \wedge e $\$$, respectivamente, como forma de indicar essas posições.

De forma semelhante à coleção WORDS criada para armazenar um dicionário com o número de ocorrências das palavras encontradas no corpus, definem-se, aqui, duas novas coleções: BIGRAMS e TRIGRAMS, para armazenar os bi- e trigramas com suas respectivas frequências de ocorrência. Como já é conhecida a frequência de ocorrência das palavras, parte-se dessa informação para obter a frequência de ocorrência dos n-gramas. A função, apresentada a seguir, retorna uma coleção com a frequência de ocorrência dos n-gramas para um dado valor n escolhido:

```

def compute_ngram_statistics(n=1):
    from collections import Counter
    NGRAM = Counter()
    for w in WORDS:
        ngrams = compute_word_ngrams(w, n)
        for ngram in ngrams:
            NGRAM[ngram] += WORDS[w]
    return NGRAM

```

Em seguida, obtém-se a estatística dos bigramas e dos trigramas da seguinte maneira:

```

BIGRAMS = compute_ngram_statistics(n=2)
TRIGRAMS = compute_ngram_statistics(n=3)

```

Com essas informações, é possível calcular o índice de peculiaridade de uma determinada palavra, conforme exposto na Equação (1), e usar essa informação para determinar se uma certa sequência de caracteres deve ou não ser aceita como uma palavra válida na língua. Ou ainda, pode-se utilizar esse índice para escolher os possíveis candidatos para a correção de uma sequência malformada.

A título de ilustração, a Figura 1 apresenta o histograma do índice de peculiaridade de palavras, pertencentes ao dicionário padrão do inglês no Linux. Para gerar tal gráfico, obteve-se, inicialmente, a frequência de ocorrência das palavras a partir de 100 livros do Projeto Gutenberg. Destaca-se, na

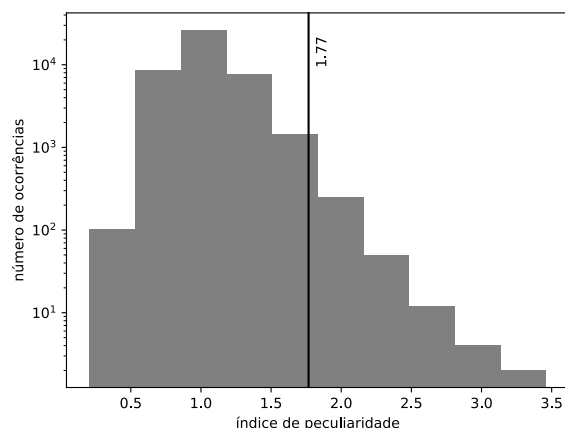


Figura 1. Histograma do índice de peculiaridade utilizando livros do Projeto Gutenberg como corpus.
 Fonte: Elaboração própria.

figura, com uma barra na vertical o valor de peculiaridade dos vocábulos, para os quais se observa que 99% das palavras têm índice de peculiaridade menor do que esse limiar.

O índice de peculiaridade pode ser utilizado como um bom indicativo de malformação de uma sequência de caracteres. A seguir, apresenta-se uma pequena lista de sequências encontradas no conjunto de livros extraídos do Projeto Gutenberg e que possuem limiar de peculiaridade maior do que o limiar determinado:

armfeldt, bio, ciitzens, dulcineae, dwellingplaces, emunctae, expoliuntur,
 fayn, fortyfoot, funkyish, goerz, heytesbury, highminded, jommlin,
 milkjug, pistoia, pocketsful, rohghah, strikebreaker, twentythree, unpav,
 valentinavyczia, wiolent, wyman

6 Outras abordagens

As abordagens até então expostas neste trabalho utilizam características estatísticas da língua para ordenar potenciais candidatos. Há, contudo, outras informações que podem ser incorporadas às análises a fim de auxiliar na organização e na definição dos candidatos mais prováveis, a depender do tipo de texto. Por exemplo, se o corretor ortográfico atuar no pós-processamento de um sistema de reconhecimento de caracteres (OCR, *Optical Character Recognition*), passa a ser importante considerar a proximidade ótica dos caracteres, de forma a corrigir erros gerados no processo de digitalização de documentos (LIU et al., 1991; TONG; EVANS, 1996; NAGATA, 1998; TAGHVA; STOFISKY, 2001; CACHO, 2012). No caso em que se deseja corrigir erros de datilografia, é intuitivo que se considere a proximidade das teclas (MIN; WILSON, 1995; SAMUELSSON, 2017) (ideia semelhante é utilizada pelo algoritmo de Needleman-Wunsch em bioinformática para alinhar proteínas e nucleotídeos). Ainda, se o objetivo é abarcar erros gerados pela falta de proficiência de indivíduos na tarefa de soletrar palavras, pode-se considerar a vizinhança fonológica de sequências em uma língua (AVANÇO; DURAN, 2014; PILÁN; VOLODINA, 2018). Embora essas abordagens não sejam tratadas neste texto, há uma família de corretores ortográficos que abarcam tais temas disponíveis no repositório de um dos autores deste artigo (ARAUJO, 2020).

7 Resultados

Apresentam-se, a seguir, os resultados da implementação do corretor ortográfico proposto por Norvig (2007) aplicados a um conjunto de arquivos de erros ortográficos de Birkbeck, criada por Roger Mitton e proveniente de diversas fontes reunidas no Arquivo de Textos de Oxford (*Oxford Text Archive*). Encontram-se entre eles os resultados de ditados para teste de ortografia e de erros em escrita livre, grande parte deles escritos à mão. Os dados foram obtidos de crianças em idade escolar e de alunos universitários ou adultos letrados. Para que a análise seja realizada, de início, verificam-se a acuracidade do corretor ortográfico no reconhecimento de palavras corretas ou incorretas do ponto de vista ortográfico e os efeitos do emprego de regras para a remoção dos afijos. Em sequência,

verifica-se a efetividade das sugestões geradas pelo corretor.

Como salientado, o intuito de remover afixos é expandir virtualmente o dicionário usado pelo corretor ortográfico e, assim, diminuir a taxa de palavras indicadas como ortograficamente erradas. Uma palavra não encontrada no dicionário pode passar a ser quando se remove dela os afixos, como demonstram os resultados apresentados na Tabela 1. Observa-se que, no corpus em questão, há um acréscimo de, aproximadamente, 5,2% de palavras corretas que são analisadas como ortograficamente adequadas. '*Accessibility*', '*choices*', '*unequaled*', por exemplo, que até então eram palavras desconhecidas para o corretor, passaram a ser consideradas ortograficamente corretas após a remoção dos afixos. A incorporação ocorreu porque as palavras '*accessible*', '*choice*' e '*unequal*', respectivamente, encontram-se no dicionário e, portanto, promoveram a incorporação de vocábulos derivados como palavras possíveis na língua. É interessante destacar que a percentagem pode ser ainda maior se a língua em estudo tiver uma riqueza morfológica maior do que o inglês.

Tabela 1. Matriz de confusão do corretor ortográfico na tarefa de reconhecimento de palavras (ortografia correta e com erro ortográfico).

(a) Sem remoção de afixos.		(b) Com remoção de afixos.	
corretor	categoria verdadeira	correta	erro
corretor	correta	430	14
	erro	52	648

corretor	categoria verdadeira	correta	erro
corretor	correta	455	55
	erro	27	607

Fonte: Elaboração própria.

Observe, entretanto, que, assim como há um maior índice de acerto do corretor no que se refere à inclusão de palavras corretas no banco de dados, há também um aumento na quantidade de palavras que contêm erros ortográficos que desacertadamente são consideradas corretas quando seus afixos são removidos, como exposto na Tabela 1. O número de palavras erradas que são analisadas pelo corretor como corretas sobe de 14 para 55, levando a um acréscimo de 6,2% de erros nessa categoria. Ao terem seus sufixos removidos, sequências como '*voteing*', '*timeing*' e '*lates*' passam a ser interpretadas como palavras corretas, visto que '*vote*', '*time*' e '*late*' são palavras existentes e estão no dicionário.

Embora, à primeira vista, a diferença entre o corretor sem remoção de afixo (Tabela 1a) e com remoção de afixo (Tabela 1b) seja pequena, em uma análise de 300 amostras aleatórias do corpus em estudo, calculou-se o valor preditivo positivo, dado por $CE/(CC + CE)$, e o valor preditivo negativo, dado por $EC/(EC + EE)$, onde CC, EC, EC e EE são os elementos da matriz de confusão, na ordem em que são apresentados na Tabela 1.

A Figura 2 apresenta o histograma, a partir do qual se verificou que a diferença entre os corretores é significativa. A barra em preto vertical expressa os valores preditivos obtidos pela Tabela 1, indicando valores próximos à média das distribuições. Um corretor ortográfico que inclua processos de remoção de afixos traz, portanto, ganhos significativos para a identificação de vocábulos corretos, ao custo de não identificar alguns vocábulos com possíveis erros ortográficos.

Ademais, é interessante destacar que o corretor ortográfico, apesar da remoção dos afixos, continua classificando palavras reais, como '*graphically*' e '*unequivocally*' como ortograficamente erradas. Destacam-se, para estes casos, duas imprecisões do corretor: a ausência de recursividade no processo de afixação e a limitação do corpus base para a formação do dicionário. O primeiro refere-se aos processos de afixação que ocorrem em sequência: '*graphic*' → '*graphical*' → '*graphically*' e, em alguns casos, com modificação no radical da palavra: '*unequivocally*' → '*unequivocal*' → '*equivocal*' → '*equivoque*'. Perceba que a reaplicação do processo de remoção dos afixos poderia ser eficiente para o primeiro exemplo, mas não para o segundo. Para este, faz-se necessária uma análise morfofonológica mais refinada. Os exemplos descritos, ainda, evidenciam as limitações impostas por qualquer banco de dados, sendo, assim, importante o uso de um dicionário ou de um corpus que seja, de fato, representativo da língua.

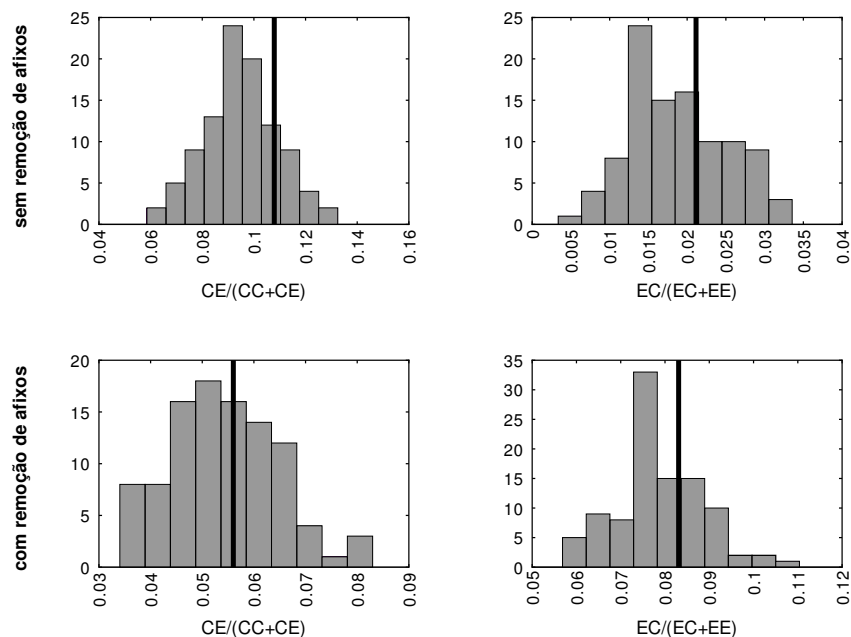


Figura 2. Resultado da distribuição dos valores preditivos positivo e negativo, gerados a partir de 300 amostras aleatórias. A barra preta vertical apresenta o valor obtido na Tabela 1.

Fonte: Elaboração própria.

No que se refere à efetividade das sugestões de candidatos para a correção de palavras com erro ortográfico, observa-se, conforme exposto na Tabela 2, que a taxa de acerto cresce ao se aumentar o número de candidatos. Os candidatos são gerados pelas regras de mínima edição e ordenados pelo número de edições e pela frequência de ocorrência no corpus das palavras geradas. A diferença entre os valores preditivos negativos para os corretores, com e sem remoção de afixos, é pequena, não impactando no desempenho da correção.

Tabela 2. Taxa de acerto na proposição de correção, em função do número de candidatos apresentados.

remoção de sufixo	número de candidatos			
	1	3	5	7
sem remoção	0.702	0.798	0.816	0.822
com remoção	0.702	0.798	0.816	0.822

Fonte: Elaboração própria.

Roger Mitton ainda disponibiliza em sua página pessoal²¹ outras bases de dados que serão aqui utilizadas: Holbrook (inclui erros ortográficos produzidos à mão por crianças no último ano escolar), *Aspell* (contém os dados coletados por Kevin Atkinson para testar o *Aspell*) e Wikipedia (contém erros de ortografia cometidos pelos contribuidores da Wikipedia). As duas últimas bases de dados, portanto, são obtidas de textos datilografados. É importante verificar a forma com que cada um dos dados foram obtidos, pois podem representar diferentes tipos de erros.

8 Conclusão

O presente trabalho apresentou uma revisão sobre o desenvolvimento de corretores ortográficos, especificando as estratégias e as ferramentas propostas por cada um deles. Mostrou-se, a partir de algumas ideias expostas por Norvig (2007) e Robbins e Beebe (2005), como é possível criar um corretor ortográfico simples e eficiente, o qual pode ser facilmente adaptado para diversas línguas.

Em resumo, para realizar a tarefa de detecção de erro, verificou-se se a sequência de busca se encontrava no léxico gerado por um grande corpus. Caso não fosse encontrada, observava-se se havia

²¹ <https://www.dcs.bbk.ac.uk/{~}ROGER/corpora.html>

Tabela 3. Bases de dados utilizadas para os testes de performance dos corretores ortográficos.

nome	erros ortográficos	palavras	forma
Birkbeck	36.133	6.136	escrito à mão*
Holbrook	1791	1200	escrito à mão
Aspell	531	450	datilografado
Wikipedia	2.455	1.922	datilografado

* em grande parte

Fonte: Elaboração própria.

Tabela 4. Matriz de confusão do corretor ortográfico na tarefa de reconhecimento de palavras sem a remoção de afixos.

(a) Birkbeck.				(b) Holbrook.			
corretor		categoria verdadeira		corretor		categoria verdadeira	
		correta	erro			correta	erro
	correta	5142	3129		correta	894	477
corretor	erro	824	32712	corretor	erro	224	1283
(c) Aspell.				(d) Wikipedia.			
corretor		categoria verdadeira		corretor		categoria verdadeira	
		correta	erro			correta	erro
	correta	337	15		correta	1395	12
corretor	erro	112	515	corretor	erro	516	2443

Fonte: Elaboração própria.

Tabela 5. Taxa de acerto na proposição de correção, em função do número de candidatos apresentados.

base de dados	número de candidatos			
	1	3	5	7
Birkbeck	0.315	0.377	0.391	0.397
Holbrook	0.260	0.322	0.341	0.353
Aspell	0.433	0.533	0.546	0.554
Wikipedia	0.607	0.692	0.702	0.705

Fonte: Elaboração própria.

nela a presença de afixo. Em caso afirmativo, aplicavam-se regras para remoção de afixos e buscava-se encontrar o radical da palavra no dicionário. Constatou-se que os corretores, com e sem remoção de afixo, apresentam resultados significativamente distintos, porém, o desempenho dos corretores na tarefa de corrigir erros ortográficos é essencialmente a mesma. Conforme previamente observado, a remoção de afixos, sem levar em consideração os aspectos gramaticais, costuma acarretar a aceitação de palavras grafadas de forma errada, não sendo, portanto, uma boa estratégia de correção ortográfica. Como abordagem alternativa, Elliott (1988) propôs a utilização de classes de equivalência de afixos para obter melhores resultados.

Embora os erros de palavras reais sejam relevantes, é importante ressaltar que o exemplo de corretor ortográfico proposto por Norvig (2007) não visa os corrigir, nem os erros gerados pela inserção nem pelo apagamento de espaços em branco em/entre palavras. Segundo Verberne (2002), os erros de palavra real compreendem de 25 a 40% dos erros encontrados (WING; BADDELEY, 1980; MITTON, 1987; YOUNG; EASTMAN; OAKMAN, 1991) e, segundo Kukich (1992a), os erros gerados pela inserção ou pelo apagamento de espaços abarcam cerca de 15%. Além disso, observou-se, no presente trabalho e em Kukich (1992b), que corretores ortográficos de palavras isoladas podem corrigir, em geral, aproximadamente, 78% dos erros ortográficos, desde que não sejam erros de palavra real. É preciso lembrar que, para gerar candidatos para corrigir os erros ortográficos, este trabalho utilizou operações de simples edição: adição, remoção, substituição e transposição de caracteres. Nele, a distância de edição e a frequência são utilizadas para ordenar e determinar quais são os melhores candidatos. Os resultados alcançaram uma taxa de 70% de êxito na correção e de, aproximadamente, 82% na exposição da palavra correta dentre os 5 primeiros candidatos. Não é possível afirmar se existe um limite para a performance de um corretor ortográfico de palavras isoladas, como o que foi apresentado neste trabalho. Entretanto, seres humanos desempenhando a mesma tarefa alcançam em média uma taxa de acerto na correção de 74% (KUKICH, 1992b), o que indica uma boa performance dos corretores aqui apresentados.

Ressalta-se, por fim, que o corretor ortográfico foi escrito em Python, usando orientação a objetos, tendo em vista que essa abordagem permite o reaproveitamento do código na elaboração de novos corretores ortográficos que considerem, por exemplo, novas formas de identificar erros, gerar candidatos e ordená-los conforme alguma métrica qualquer. Com a utilização de classes, é possível, inclusive, criar corretores ortográficos derivados de diferentes corretores, por meio de múltiplas estratégias e de métricas que geram e ranqueiam candidatos. Ademais, o *framework* criado para testar os corretores ortográficos poderá também ser prontamente aplicado a novos corretores. Desta forma, torna-se fácil gerar testes e estabelecer comparações entre diferentes abordagens.

Todos os códigos estão disponíveis no repositório no Github (ARAUJO, 2020), inclusive novos corretores que estão sendo desenvolvidos pelo autor.

Referências

- ARAUJO, Leonardo. *leolca/spellcheck v0.1-alpha*. [S.l.]: Zenodo, 2020. DOI: 10.5281/ZENODO.3235670. Disponível em: <https://zenodo.org/record/3235670>. Acesso em: 27 nov. 2020.
- ATKINSON, Kevin. *Aspell*. [S.l.: s.n.], 1998. Disponível em: <http://aspell.net/>. Acesso em: 8 jul. 2020.
- AVANÇO, Lucas Vinicius; DURAN, Magali Sanches. Towards a Phonetic Brazilian Portuguese Spell Checker. In: PROCEEDINGS of ToRPOrEsp Workshop PROPOR 2014. [S.l.: s.n.], 2014.
- BEIDER, Alexander; MORSE, Stephen P. *An Alternative to Soundex with Fewer False Hits*. 2008. Disponível em: <https://stevemorse.org/phonetics/bmpm.htm>. Acesso em: 8 jul. 2020.
- _____. *Phonetic Matching: A Better Soundex*. 2010. Disponível em: <http://stevemorse.org/phonetics/bmpm2.htm>. Acesso em: 8 jul. 2020.
- CACHO, Jorge Ramon Fonseca. *Improving OCR Post Processing with Machine Learning Tools*. Ago. 2012. Tese (Doutorado) – University of Nevada, Las Vegas.
- CASTRO, Daniel. *Métodos para la corrección ortográfica automática del español*. 2012. Tese (Doutorado) – Universidad de Oriente, Santiago de Cuba. Disponível em: http://www.cerpamid.co.cu/sitio/files/tesis_master_daniel.pdf.

- CHURCH, Kenneth W.; GALE, William A. Probability scoring for spelling correction. *Statistics and Computing*, Springer Science e Business Media LLC, v. 1, n. 2, p. 93–103, dez. 1991. DOI: 10.1007/bf01889984.
- CORPORATION, Microsoft. Michael Cameron; Hugh Williams. *Query suggestions for no result web searches*. US, set. 2013. US8583670B2, Depósito: 4 out. 2007. Concessão: 9 abr. 2009. Patent. Disponível em: <https://patents.google.com/patent/US8583670B2/en>.
- CORPORATION, Microsoft. Eric D. Brill; Silviu-Petru Cucerzan. *Systems and methods for spell checking*. EU, set. 2005. EP1577793A2, Depósito: 14 mar. 2005. Concessão: 21 nov. 2005. Patent. Disponível em: <https://patents.google.com/patent/EP1577793A2/en>.
- DAMERAU, Fred J. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 7, n. 3, p. 171–176, mar. 1964.
- DAVIDSON, Leon. Retrieval of misspelled names in an airlines passenger record system. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 5, n. 3, p. 169–171, mar. 1962. DOI: 10.1145/366862.366913. Disponível em: <https://doi.org/10.1145/366862.366913>.
- DEFFNER, Renate; EDER, Klaus; GEIGER, Hans. Word Recognition as a First Step Towards Natural Language Processing with Artificial Neural Networks. In: KONNEKTIONISMUS in Artificial Intelligence und Kognitionsforschung. [S.l.]: Springer Berlin Heidelberg, 1990. p. 221–225. DOI: 10.1007/978-3-642-76070-9_27. Disponível em: https://doi.org/10.1007/978-3-642-76070-9_27.
- DUNLAVEY, Michael R. Letter to the Editor: On Spelling Correction and Beyond. *ACM*, v. 24, n. 9, p. 608–608, set. 1981.
- ELLIOTT, R. J. *Annotating spelling list words with affixation classes*. [S.l.], dez. 1988.
- GLANTZ, Herbert T. On the recognition of information with a digital computer. In: PROCEEDINGS of the 1956 11th ACM national meeting. [S.l.]: ACM Press, 1956. DOI: 10.1145/800258.808966. Disponível em: <https://doi.org/10.1145/800258.808966>.
- GOLDING, Andrew R.; SCHABES, Yves. Combining Trigram-based and feature-based methods for context-sensitive spelling correction. In: PROCEEDINGS of the 34th annual meeting on Association for Computational Linguistics. [S.l.]: Association for Computational Linguistics, 1996. DOI: 10.3115/981863.981873. Disponível em: <https://doi.org/10.3115/981863.981873>.
- GORIN, Ralph E. *SPELL: Spelling Check and Correction Program*. [S.l.: s.n.], 1974. Disponível em: <https://www.saildart.org/allow/SPELL.REG%5C%5bUP,DOC%5C%5d>. Acesso em: 8 jul. 2020.
- GORIN, Ralph E.; WILLISSON, Pace; KUENNING, Geoff. *Ispell*. [S.l.: s.n.], 1971. Disponível em: <https://www.cs.hmc.edu/%7B%5C~%7Dgeoff/ispell.html>. Acesso em: 8 jul. 2020.
- GUPTA, Prabhakar. A Context-Sensitive Real-Time Spell Checker with Language Adaptability. In: 2020 IEEE 14th International Conference on Semantic Computing (ICSC). [S.l.]: IEEE, fev. 2020. DOI: 10.1109/icsc.2020.00023.
- HODGE, Victoria J.; AUSTIN, Jim. A comparison of a novel neural spell checker and standard spell checking algorithms. *Pattern Recognition*, Elsevier BV, v. 35, n. 11, p. 2571–2580, nov. 2002. DOI: 10.1016/s0031-3203(01)00174-1. Disponível em: [https://doi.org/10.1016/s0031-3203\(01\)00174-1](https://doi.org/10.1016/s0031-3203(01)00174-1).
- _____. A comparison of standard spell checking algorithms and a novel binary neural approach. *IEEE Transactions on Knowledge and Data Engineering*, Institute of Electrical e Electronics Engineers (IEEE), v. 15, n. 5, p. 1073–1081, set. 2003. DOI: 10.1109/tkde.2003.1232265. Disponível em: <https://doi.org/10.1109/tkde.2003.1232265>.
- INC., Google. Noam Shazeer. *Method of spell-checking search queries*. US, mar. 2007. US7194684B1, Depósito: 9 abr. 2002. Concessão: 20 mar. 2007. Patent. Disponível em: <https://patents.google.com/patent/US7194684B1/en>.
- INGELS, Peter. Connected Text Recognition Using Layered HMMs and Token Passing. *CoRR*, cmp-lg/9607036, 1996. Disponível em: <http://arxiv.org/abs/cmp-lg/9607036>.
- JURAFSKY, Daniel; MARTIN, James H. *Speech and Language Processing: An introduction to Natural Language Processing*. [S.l.]: Prentice Hall, jan. 2008. ISBN 0131873210.
- KERNIGHAN, Mark D.; CHURCH, Kenneth W.; GALE, William A. A spelling correction program based on a noisy channel model. In: PROCEEDINGS of the 13th conference on Computational linguistics. [S.l.]: Association for Computational Linguistics, 1990. DOI: 10.3115/997939.997975. Disponível em: <https://doi.org/10.3115/997939.997975>.

- KIM, Jong Yong; SHAW-TAYLOR, John. An approximate string-matching algorithm. *Theoretical Computer Science*, Elsevier BV, v. 92, n. 1, p. 107–117, jan. 1992. DOI: 10.1016/0304-3975(92)90138-6. Disponível em: [https://doi.org/10.1016/0304-3975\(92\)90138-6](https://doi.org/10.1016/0304-3975(92)90138-6).
- KIM, Jong Yong; SHAW-TAYLOR, John. Fast string matching using an n-gram algorithm. *Software: Practice and Experience*, Wiley, v. 24, n. 1, p. 79–88, jan. 1994. DOI: 10.1002/spe.4380240105. Disponível em: <https://doi.org/10.1002/spe.4380240105>.
- KNUTH, Donald Ervin. *The Art of Computer Programming*. [S.l.]: Addison-Wesley, 1973. v. 3.
- KUČERA, Henry; FRANCIS, Winthrop Nelson. *Computational analysis of present-day American English*. [S.l.]: Brown University Press, 1967.
- KUKICH, Karen. Spelling correction for the telecommunications network for the deaf. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 35, n. 5, p. 80–90, mai. 1992. DOI: 10.1145/129875.129882. Disponível em: <https://doi.org/10.1145/129875.129882>.
- _____. Technique for automatically correcting words in text. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 24, n. 4, p. 377–439, dez. 1992. DOI: 10.1145/146370.146380.
- LIU, Lon-Mu et al. Adaptive post-processing of OCR text via knowledge acquisition. In: PROCEEDINGS of the 19th annual conference on Computer Science - CSC '91. [S.l.]: ACM Press, 1991.
- LUCCHESI, Cláudio L.; KOWALTOWSKI, Tomasz. Applications of finite automata representing large vocabularies. *Software: Practice and Experience*, Wiley, v. 23, n. 1, p. 15–30, jan. 1993. DOI: 10.1002/spe.4380230103. Disponível em: <https://doi.org/10.1002/spe.4380230103>.
- MAHONEY, Michael Sean. *An Oral History of Unix*. [S.l.]: Zenodo, 1998. DOI: 10.5281/zenodo.2525530.
- MCILROY, Malcolm Douglas. Development of a Spelling List. *IEEE Transactions on Communications*, Institute of Electrical e Electronics Engineers (IEEE), v. 30, n. 1, p. 91–99, jan. 1982. DOI: 10.1109/tcom.1982.1095395. Disponível em: <https://doi.org/10.1109/tcom.1982.1095395>.
- MCMAHON, Lee E.; CHERRY, Lorinda L.; MORRIS, Robert. Statistical Text Processing. *The Bell System Technical Journal*, v. 57, n. 6, p. 2137–2154, 1978.
- MIN, Kyongho; WILSON, William. Syntactic Recovery and Spelling Correction of Ill-formed Sentences. In: 3RD Conference of the Australasian Cognitive Science (CogSci95). [S.l.: s.n.], mar. 1995. p. 1–10.
- MITTON, Roger. Fifty years of spellchecking. *Writing Systems Research*, Informa UK Limited, v. 2, n. 1, p. 1–7, jan. 2010. DOI: 10.1093/wsr/wsq004.
- _____. Spelling checkers, spelling correctors and the misspellings of poor spellers. *Information Processing & Management*, Elsevier BV, v. 23, n. 5, p. 495–505, jan. 1987.
- MORRIS, Robert; CHERRY, Lorinda L. Computer detection of typographical errors. *IEEE Transactions on Professional Communication*, Institute of Electrical e Electronics Engineers (IEEE), PC-18, n. 1, p. 54–56, mar. 1975. DOI: 10.1109/tpc.1975.6593963.
- NAGATA, Masaaki. Japanese OCR error correction using character shape similarity and statistical language model. In: PROCEEDINGS of the 36th annual meeting on Association for Computational Linguistics. [S.l.]: Association for Computational Linguistics, 1998. p. 922–928.
- NORVIG, Peter. *How to write a spelling corrector*. [S.l.: s.n.], 2007. <http://norvig.com/spell-correct.html>.
- ODELL, Margaret King. The profit in records management. *Systems*, New York, v. 20, 1956.
- OFLAZER, Kemal. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, v. 22, n. 1, p. 73–89, 1996.
- OFLAZER, Kemal; GÜZEY, Cemalettin. Spelling correction in agglutinative languages. In: PROCEEDINGS of the fourth conference on Applied natural language processing. [S.l.]: Association for Computational Linguistics, 1994. DOI: 10.3115/974358.974406. Disponível em: <https://doi.org/10.3115/974358.974406>.
- PETERSON, James L. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 23, n. 12, p. 676–687, dez. 1980. DOI: 10.1145/359038.359041. Disponível em: <https://doi.org/10.1145/359038.359041>.
- PHILIPS, Lawrence. Hanging on the metaphone. *Computer Language*, v. 7, n. 12, p. 39–43, 1990.
- _____. The Double Metaphone Search Algorithm. *C/C++ Users J.*, CMP Media, Inc., USA, v. 18, n. 6, p. 38–43, jun. 2000. ISSN 1075-2838.

- PILÁN, Ildikó; VOLODINA, Elena. Exploring word embeddings and phonological similarity for the unsupervised correction of language learner errors. In: LATECH@COLING. [S.l.: s.n.], 2018.
- PILAR ANGELES, Maria del; ESPINO-GAMEZ, Adrian; GIL-MONCADA, Jonathan. Comparison of a Modified Spanish Phonetic, Soundex, and Phonex coding functions during data matching process. In: 2015 International Conference on Informatics, Electronics & Vision (ICIEV). [S.l.]: IEEE, jun. 2015. DOI: 10.1109/iciev.2015.7334028. Disponível em: <https://doi.org/10.1109/iciev.2015.7334028>.
- POLLOCK, Joseph J.; ZAMORA, Antonio. Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 27, n. 4, p. 358–368, abr. 1984. DOI: 10.1145/358027.358048.
- ROBBINS, Arnold; BEEBE, Nelson H. F. *Classic Shell Scripting*. [S.l.]: O'Reilly Media, 2005. ISBN 0596555261.
- RUSSELL, Robert C. Robert C. Russell. *Index*. 1918. US 1261167A, Depósito: 25 out. 1917. Concessão: 02 abr. 1918. Disponível em: <https://patents.google.com/patent/US1261167A/en>.
- RUSSELL, Robert C. Robert C. Russell. *Index*. 1922. US 1435663A, Depósito: 28 nov. 1921. Concessão: 14 nov. 1922. Disponível em: <https://patents.google.com/patent/US1435663A/en>.
- SAMUELSSON, Axel. *Weighting Edit Distance to Improve Spelling Correction in Music Entity Search*. Jun. 2017. Diss. (Mestrado) – KTH Royal Institute of Technology, Stockholm.
- SHANNON, Claude E. A mathematical theory of communication. *Bell Syst. Tech. J.*, v. 27, n. 3, p. 379–423, 1948.
- TAGHVA, Kazem; STOFISKY, Eric. OCRSpell: an interactive spelling correction system for OCR errors in text. *International Journal on Document Analysis and Recognition*, Springer Science e Business Media LLC, v. 3, n. 3, p. 125–137, mar. 2001. DOI: 10.1007/pl00013558. Disponível em: <https://doi.org/10.1007/pl00013558>.
- TAYLOR, W. D. *Grope: A spelling error correction tool*. [S.l.], 1981.
- TONG, Xiang; EVANS, David A. A Statistical Approach to Automatic OCR Error Correction in Context. In: PROCEEDINGS of the Fourth Workshop on Very Large Corpora (WVLC-4). [S.l.: s.n.], 1996. p. 88–100.
- VERBERNE, Suzan. *Context-sensitive spell checking based on word trigram probabilities*. 2002. Diss. (Mestrado) – University of Nijmegen.
- WING, Alan M.; BADDELEY, Alan D. Spelling errors in handwriting: A corpus and distributional analysis. In: FRITH, Uta (Ed.). *Cognitive Processes in Spelling*. London: Academic Press, 1980. p. 251–285.
- YANNAKOUDAKIS, Emmanuel J.; FAWTHROP, David. An intelligent spelling error corrector. *Information Processing & Management*, Elsevier BV, v. 19, n. 2, p. 101–108, jan. 1983. DOI: 10.1016/0306-4573(83)90046-8. Disponível em: [https://doi.org/10.1016/0306-4573\(83\)90046-8](https://doi.org/10.1016/0306-4573(83)90046-8).
- _____. The rules of spelling errors. *Information Processing & Management*, Elsevier BV, v. 19, n. 2, p. 87–99, jan. 1983. DOI: 10.1016/0306-4573(83)90045-6. Disponível em: [https://doi.org/10.1016/0306-4573\(83\)90045-6](https://doi.org/10.1016/0306-4573(83)90045-6).
- YOUNG, Charlene W.; EASTMAN, Caroline M.; OAKMAN, Robert L. An analysis of ill-formed input in natural language queries to document retrieval systems. *Information Processing & Management*, Elsevier BV, v. 27, n. 6, p. 615–622, jan. 1991. DOI: 10.1016/0306-4573(91)90002-4. Disponível em: [https://doi.org/10.1016/0306-4573\(91\)90002-4](https://doi.org/10.1016/0306-4573(91)90002-4).
- YUNUS, Ahmed; MASUM, Md. A Context Free Spell Correction Method using Supervised Machine Learning Algorithms. *International Journal of Computer Applications*, Foundation of Computer Science, v. 176, n. 27, p. 36–41, jun. 2020. DOI: 10.5120/ijca2020920288. Disponível em: <https://doi.org/10.5120/ijca2020920288>.
- ZAMORA, Elena M.; POLLOCK, Joseph J.; ZAMORA, Antonio. The use of trigram analysis for spelling error detection. *Information Processing & Management*, Elsevier BV, v. 17, n. 6, p. 305–316, jan. 1981. DOI: 10.1016/0306-4573(81)90044-3. Disponível em: [https://doi.org/10.1016/0306-4573\(81\)90044-3](https://doi.org/10.1016/0306-4573(81)90044-3).