PROGRAMAÇÃO GENÉTICA DE ÁRVORES DE REGRAS PARA NORMALIZAÇÃO DE TEXTOS

Fabio Ferman

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador(es): Geraldo Bonorino Xexéo

Rio de Janeiro

Janeiro de 2016

PROGRAMAÇÃO GENÉTICA DE ÁRVORES DE REGRAS PARA NORMALIZAÇÃO DE TEXTOS

Fabio Ferman

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____
Prof. Geraldo Bonorino Xexéo, D. Sc.

_____
Prof. Geraldo Zimbrão da Silva, D. Sc.

_____
Prof. Carla Amor Divino Moreira Delgado, D. Sc.

RIO DE JANEIRO, RJ - BRASIL
JANEIRO DE 2016

Ferman, Fabio

Programação Genética de Árvores de Regras para Normalização de Textos / Fabio Ferman. – Rio de Janeiro: UFRJ/COPPE, 2016

XI, 129 p.: il.; 29,7 cm.

Orientador: Geraldo Bonorino Xexéo

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2016.

Referências Bibliográficas: p. 105 - 111.

1. Normalização de texto. 2. Programação Genética. 3. Solução baseada em Regras. I. Xexéo, Geraldo Bonorino. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Agradeço primeiramente ao meu orientador Geraldo Bonorino Xexéo por todo o apoio e orientação dados na execução deste trabalho, que certamente exigiram horas de dedicação e paciência.

Aos professores Geraldo Zimbrão da Silva e Carla Amor Divido Moreira Delgado por terem aceitado o convite para compor a banca avaliadora do meu trabalho.

Agradeço à CAPES, COPPETEC e CNPq pelo apoio financeiro no formato de bolsas e no financiamento das viagens para congressos, me proporcionando desfrutar de forma mais prazerosa e intensa a minha pós-graduação.

À toda a equipe do CAPGov por todo o suporte dado durante os mais de dois anos que estive no mestrado, me ajudando a evoluir academicamente e profissionalmente.

Agradeço ao Tiago Santos da Silva e Luan Barbosa Garrido por todas as discussões e sugestões a respeito do meu trabalho. À Daiane Evangelista Ferreira por toda a ajuda e gentileza de me auxiliar principalmente em dúvidas técnicas quanto à formatação.

Ao Adriano Gomes Sabino, Daiane Evangelista Ferreira, Fernanda Ribeiro, Luan Barbosa Garrido, Sérgio Assis Rodrigues e Tiago Santos da Silva pelas contribuições que fizeram durante todo o curso.

Agradeço aos meus amigos que sempre perguntaram sobre como estava o desenvolvimento da minha dissertação e quando iria terminá-la, além de me proporcionarem momentos de lazer e diversão.

À minha namorada, Ilana Roitman, pelo suporte emocional e por todo o companheirismo oferecido, além de me entender quando não pudemos nos encontrar para que eu concluísse meu trabalho.

Agradeço aos meus pais e minha irmã por me ajudarem em todos os tipos de dificuldades que apareceram antes e durante o mestrado, fazendo eu chegar até o dia de hoje em que consigo entregar meu trabalho de conclusão do curso.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROGRAMAÇÃO GENÉTICA DE ÁRVORES DE REGRAS PARA NORMALIZAÇÃO DE TEXTOS

Fabio Ferman

Janeiro/2016

Orientador: Geraldo Bonorino Xexéo

Programa: Engenharia de Sistemas e Computação

Erros de grafia consistem de adversidades que devem ser tratadas em diversos cenários, dentre os quais se destacam: a necessidade de uma escrita correta para documentos importantes no intuito de mostrar seriedade e clareza, possibilidade de uso de sistemas *Text-to-Speech* e para aplicação de técnicas de processamento de linguagem natural. Até algum tempo atrás esses erros podiam ser considerados falhas de digitação ou desconhecimento quanto à forma correta de escrita, porém atualmente estes podem ser cometidos de forma proposital, existentes em maior escala com o surgimento do internetês, muito presente nos microblogs e aplicativos de mensagens instantâneas. Dentre as técnicas de normalização de texto existentes destacam-se as baseadas em regras, que são muito precisas porém com baixa recuperação, e as que utilizam do aprendizado de máquina, que em geral apresentam maior recuperação e menor precisão. Esse trabalho visa a criação de uma técnica híbrida entre as abordagens para normalização de texto, tal que esta desfrute dos benefícios presentes em ambas as técnicas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

GENETIC PROGRAMMING OF RULES TREES FOR TEXT NORMALIZATION

Fabio Ferman

Janeiro/2016

Advisor: Geraldo Bonorino Xexéo

Department: Systems Engineering and Computer Science

Spelling deviations are adversities, which must be addressed in many scenarios, for example: the necessity of a well-written text in important documents in order to show seriousness and clarity, the possibility to use Text-to-Speech systems, and to apply natural language processing techniques. Until some time ago, these errors could be considered an unintentional consequence of cognitive or interface problems, but nowadays spelling deviations can be committed intentionally, increasing the diversity of types of deviations with the creation of the netspeak, found at microblogs and instant message applications. Among the normalization techniques there are the rule-based approaches, which usually favor precision over recall, and the machine learning approaches, which in general favor recall over precision. This work propose the development of a hybrid technique between both approaches for text normalization, in order to enjoy the best of each technique.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 Introdução

O avanço da tecnologia vem promovendo muitas evoluções no campo da comunicação, desde a criação do telégrafo por Samuel Finley Breese Morse em 1838, passando pela invenção do celular pelo laboratório Bell em 1947, do SMS (inicialmente limitado a 160 caracteres) em 1985, do ICQ em 1996, das redes sociais em 2004 com o Facebook e Orkut e em 2006 com o Twitter, até chegar em 2009 a um aplicativo de mensagens instantâneas utilizado por mais de 700 milhões de usuários e que contabiliza uma média de 30 bilhões de mensagens enviadas por dia (G1-SP, 2015), o Whatsapp. O surgimento de novas plataformas de comunicação vem alterando, além da velocidade da informação, do dia a dia de cada indivíduo e das formas de relacionamento entre as pessoas, também tem alterado o formato da escrita.

Ao redor do mundo estima-se a existência de mais de 7 mil idiomas (LEWIS *et al.*, 2015), que utilizam diferentes tipos de alfabetos para sua representação, além de serem regidos por um grupo específico de regras ortográficas e gramaticais. A complexidade da grafia para cada língua pode variar de acordo com as suas normas de escrita, resultando em múltiplas espécies de discordâncias acerca da norma culta do idioma. A recente entrada dos microblogs e aplicativos de mensagens instantâneas, tendo a informalidade e o desleixo com a escrita como características notáveis, incrementam ainda mais a quantidade de discordâncias.

Textos com erros de escrita podem não ser um problema quando estão sendo usados para a comunicação informal entre amigos, contudo para alguns contextos é importante que estejam seguindo à norma culta da língua. Livros, revistas, jornais, contratos, e outros tipos de textos necessitam ter sua grafia sem incorreções de modo que possam transparecer sua credibilidade, importância e clareza.

Uma vez que ninguém está imune de cometer erros, em diversas circunstâncias existe a necessidade da despesa adicional com a contratação de revisores para diminuir a probabilidade de algo errôneo ser publicado. O uso de corretores automáticos também ajuda no processo de correção e, à medida que estes evoluem e capturam mais falhas, menos gastos extras passam a ser necessários para garantir um texto de qualidade (ASONOV, 2010).

Outra importância dos textos corretamente escritos consiste da possibilidade de integração dos deficientes visuais na sociedade, que em um primeiro momento pode parecer contra intuitivo, à medida que esses não conseguem ou têm dificuldade de

enxergar e consequentemente ler. Entretanto, a existência de sistemas *Text-to-Speech* (TTS), que realizam a leitura de textos de forma automática, permite levar a esse grupo a compreensão dos textos (PENNELL & LIU, 2011, 2010). Escritas com erros de grafia, consequentemente, geram uma leitura errada que pode comprometer o seu entendimento. As abreviações também apresentam problemas para sistemas TTS, uma vez que esses não sabem como devem pronunciar as palavras abreviadas e acabam por apenas emitir um som estranho na tentativa da leitura, que no geral impede o seu entendimento.

Para a aplicação de inúmeras técnicas desenvolvidas de processamento de linguagem natural é mais propício dispor de um texto escrito corretamente (GADDE *et al.*, 2011), aumentando assim a qualidade da aplicação das técnicas, como na identificação de entidades nomeadas (BANDYOPADHYAY *et al.*, 2014, CORNEY *et al.*, 2014, LIU *et al.*, 2012), geolocalização (FERMAN *et al.*, 2015, SULTANIK & FINK, 2012), detecção de eventos em tempo real (WANG *et al.*, 2014), análise de sentimentos (BAHRAINIAN & DENGEL, 2013, MAEDA *et al.*, 2012), algoritmos de recomendação (SILVA *et al.*, 2010), e outras.

Com o intuito de realizar a correção de um texto é necessário desenvolver uma técnica capaz de detectar onde estão as incorreções, eleger possíveis candidatos para conserto e em seguida selecionar o mais apropriado. Com a evolução da computação diversas técnicas de correção (também conhecidas como técnicas de normalização) de texto já estão disponíveis para uso na Web. Essas variam entre a complexidade da solução, o conjunto de idiomas tratados, os tipos de erros abordados, recursos utilizados e outras variáveis que definem uma técnica.

Devido a sua amplitude de atuação e alta variabilidade de situações a tratar, a normalização de textos ainda é um problema em aberto, mesmo que para determinados cenários já existem soluções satisfatórias, pois para outros ainda se dispõe apenas de técnicas rudimentares.

A qualidade das soluções, que é identificada a partir da análise da quantidade de acertos e erros obtidos nos variados cenários que são aplicadas, tem relação direta com o total de tratamentos realizados pela técnica ao lidar com os textos. Soluções com regras fixas, que não realizam qualquer tipo de aprendizado em cima de uma base de treinamento, estão dependentes da capacidade do desenvolvedor antever os tipos de incorreções e portanto, se restringem a corrigir um conjunto limitado e pré-estabelecido de erros. As soluções com aprendizado de máquina por sua vez têm maior potencial,

pois permitem aprender automaticamente de um corpus anotado o conhecimento nele presente, adquirindo uma maior mestria dos dados que a simples observação humana não permite conceber, contudo dependem da cobertura do corpus em relação a língua.

1.1 Objetivos

Este trabalho tem como propósito desenvolver uma técnica robusta de geração de programas normalizadores de textos, alternativa às soluções já existentes na literatura. A criação de diferentes possibilidades de solução para um determinado problema faz-se vantajosa uma vez que cada técnica pode obter resultados mais qualificados em contextos específicos, além de necessitar de um conjunto diferente de recursos para seu funcionamento, o qual pode ser a peça chave na escolha da solução a ser aplicada.

O método a ser proposto tem como objetivo apresentar uma arquitetura híbrida no que tange a fonte de conhecimento, advinda de duas procedências: (1) do desenvolvedor e (2) do aprendizado de máquina, tirando proveito do melhor que cada abordagem provê. A primeira permite inserir de forma clara e direta o conhecimento que o desenvolvedor tem sobre o tema em sua solução, enquanto a segunda possibilita o manuseio de uma grande quantidade de informações (a qual está além da capacidade humana de realizar manualmente) para se chegar em uma solução mais qualificada.

1.2 Resultados

Como o objetivo da pesquisa consiste no desenvolvimento de uma técnica de geração de programas normalizadores, a arquitetura criada é genérica o suficiente tal que possa ser aplicada a qualquer contexto de normalização e independente do idioma. Assim, a proposta não faz nenhuma pressuposição sobre as regras de escrita e solicita como entrada externa apenas componentes fáceis de se adquirir ou desenvolver.

Para validação da proposta, a arquitetura é submetida a sua aplicação no contexto de normalização de tweets em português. Devido à não existência de determinados componentes e/ou falta de compatibilidade dos demais, todas as entradas externas solicitadas são criadas, mostrando assim a plausibilidade de gerar os recursos que por ventura não existam.

O resultado encontrado equipara-se com o valor médio presente na literatura, atingindo um total de 41,45% de aumento da métrica BLEU (PAPINENI *et al.*, 2002) do estado inicial (não normalizado) para o estado final (após realizada a normalização) dos textos. A velocidade de execução do programa deve ser compreendida como dois

momentos de execução, onde o primeiro consiste de uma etapa mais demorada mas que deve ser executada apenas uma vez, onde é concebido o programa normalizador de textos adaptado ao contexto (levando na média um total de três horas e meia). O segundo momento refere-se ao tempo que o programa normalizador demora para corrigir um determinado texto, que no cenário aplicado dos tweets leva em média 6,14 milissegundos.

Para que a arquitetura pudesse englobar os tratamentos necessários tal que os erros de escrita sejam solucionados, foi realizado um estudo em cima dos desvios de escrita em relação a norma culta para o português, o qual gerou uma categorização dos erros com maior nível de detalhes.

1.3 Organização do trabalho

O trabalho é dividido em sete capítulos, no qual o primeiro foi responsável por introduzir o assunto da pesquisa, dissertando suas motivações, objetivos e principais resultados obtidos. O capítulo 2 faz uma discussão mais aprofundada sobre os erros de escrita, enfatizando os motivos que levam a sua ocorrência, as possibilidades de agrupamento pelas características dos erros e uma seção especial abordando-os quando cometidos nos microblogs e aplicativos de mensagens instantâneas.

O capítulo 3 traz a revisão da área de pesquisa em correção de textos, apresentando desde soluções mais simples até as mais complexas existentes na literatura. Entre as soluções de correção, explora aquelas específicas para textos clássicos, editores de texto e conteúdo de redes sociais. Por fim, são apresentados variados tipos de métricas de validação para o problema, tal que cada uma deve ser aplicada a um determinado contexto.

A explicação do funcionamento da programação genética é trazida pelo capítulo 4, em que explana todas as suas etapas e componentes, assim como seu funcionamento, operações genéticas, função de avaliação e critérios de parada.

O capítulo 5 faz a formulação da proposta do trabalho, na qual consiste em criar uma arquitetura capaz de gerar um programa de normalização de textos a partir de uma abordagem híbrida entre aplicação de regras e aprendizado de máquina. Para isso, inicia o capítulo realizando a elaboração da técnica de representar um programa no formato de árvore sintática e segue pela definição dos componentes da arquitetura.

Em seguida (capítulo 6), é criado um cenário completo para aplicação da arquitetura proposta, ilustrando a codificação dos seus principais componentes, as

funções utilizadas e os passos realizados para criação das bases de dados necessárias. De posse de todos os elementos obrigatórios e de um programa que simule as etapas da arquitetura, esta é posta em análise com a sua execução em diferentes conjunturas.

Por fim, o último capítulo sumariza a ideia apresentada pela arquitetura, seguida pelas contribuições realizadas pela pesquisa e os trabalhos futuros que devem ser realizados tais que a arquitetura continue a evoluir.

2 Revisão da área de aplicação

Neste capítulo são apresentados os estudos feitos acerca do cometimento de erros de escrita. Assim, são analisados os diversos motivos que podem levar a ocorrência dos desvios de grafia em relação à norma culta da língua, as diferentes categorizações em que os erros podem ser classificados, e um estudo específico em cima dos textos de microblogs e aplicativos de mensagens instantâneas.

2.1 Motivos de ocorrência dos desvios de escrita

Os erros de escrita são caracterizados como qualquer discrepância entre a grafia de uma palavra unitária, ou um conjunto destas, em relação às regras ortográficas e gramaticais de um idioma. Essas divergências podem ocorrer em diversos formatos e, com o objetivo de realizar as devidas correções, é importante primeiramente entender porque elas ocorrem. Segundo Kukich (1992), há duas razões para as incorreções acontecerem: (1) o indivíduo sabe escrever a palavra, mas ao grafar no papel ou realizar a digitação (em um computador, smartphone, ou outro aparelho) comete um ou mais erros devido à distração, pressa ao escrever, ou ao simples ato de esbarrar em outras teclas; e (2) o indivíduo não sabe, ou tem dúvidas, de como determinada palavra deve ser escrita, e por conseguinte acaba optando pelo seu modo errôneo.

O desconhecimento do idioma e das regras ortográficas e gramaticais é uma característica presente em todos ao nascer, e à medida que a vida passa somos ensinados a ler e escrever corretamente. Conforme o contato com a língua formal aumenta, a quantidade de erros ocasionados pelo desconhecimento deveria diminuir, sobrando apenas desacertos mais específicos em relação à norma culta. A existência de pessoas que permanecem exibindo uma diversidade de erros com alta frequência podem ser resultantes de um sistema de ensino de má qualidade, ou indivíduos portadores de distúrbios ou limitações (ZORZI & CIASCA, 2009).

Os erros de escrita mais comuns acontecem devido às diversas possibilidades de formar um fonema, e assim representar o som que compõe uma determinada palavra. As correspondências entre o som e as letras são divididas em três grupos (SEABRA *et al.*, 2011): biunívoca, poligâmica e poliandríaca (descritos na Tabela 1). Os dois últimos grupos, por possibilitarem a geração de dúvidas, são os principais responsáveis pelo cometimento de erros.

Tabela 1 – Grupos de correspondência entre som e letra (SEABRA et al.*, 2011)*

Grupo	**Explicação**
Biunívoca	Corresponde as letras que geram apenas um tipo de som e tem este som gerado apenas por esta letra. Exemplos de letras biunívocas para o português: ***p** (som de /p/), **t** (som de /t/), **d** (som de /d/), **f** (som de /f/), **v** (som de /v/)*
Poligâmica	Corresponde as letras que representam diferentes sons de acordo com a posição em que aparecem na palavra. Exemplo de letra poligâmica para o português: ***s** com som de /s/: quando inicia uma palavra ou é antecedida por outra consoante* ***s** com som de /z/: quando aparece em situações intervocálicas*
Poliandríaca	Corresponde aos sons que podem ser gerados por diferentes letras. Exemplo de som poliandríaco para o português: *som /k/ gerado pela letra **c**: quando sucedida pelas vogais **a**, **o** ou **u*** *som /k/ gerado pela sequência de letras **qu**: quando sucedidas pelas vogais **e** ou **i***

Um novo motivo para os erros de escrita ocorrerem começou a aparecer nos últimos vinte anos, com o surgimento dos microblogs (Facebook, Orkut, Google+, Twitter, etc.) e aplicativos de mensagens instantâneas (Whatsapp, Skype, ZapZap, ICQ, Telegram, etc.). Nesses dois novos tipos de plataforma de comunicação os textos têm apresentado uma gramática informal e não estruturada, com uso intenso de abreviações, alto volume de erros de digitação e escrita, busca na economia de caracteres[1], além dos textos se apresentarem em uma grande quantidade de vezes em um tom informal de conversa (ELLEN, 2011).

Elementos não textuais, conhecidos como *emoticons* (CUI *et al.*, 2011), também aparecem entrelaçados nas frases com objetivo de expressar sentimentos, feições e reações, no qual tem-se como exemplo: expressões de alegria (=D, =]), amor (S2), tristeza (:(, =/), entre outros. O descaso com a grafia correta faz com que a escrita em muitos casos se assemelhe à oralidade, na qual as palavras são grafadas do mesmo modo em que são pronunciadas (“não” como “naum”, “com” como “cum”, “então” como “intão”, etc.). Estes erros característicos das duas últimas décadas no ambiente Web são conhecidos como internetês[2] (ANDRADE *et al.*, 2012, GONZALEZ, 2007).

[1] A busca na economia de caracteres faz as palavras serem codificadas em um formato reduzido, tal que possam ser escritas com um número menor de letras, por exemplo, “obrigado” se transforma em “obg”, “saudades” em “sdds”, “você” em “vc”.

[2] Existem outros nomes para o internetês, como “miguxês” por exemplo. Em inglês, o termo usado é “netspeak”.

2.2 Categorização dos desvios de escrita

Uma vez entendidos os motivos que levam a geração de erros nos textos, chega o momento em que é oportuno categorizá-los segundo algum tipo de critério para ajudar na investigação de sua correção. Os tipos de categorização que são mostrados nesse capítulo vão desde agrupamentos relacionados ao motivo da geração do erro, até agrupamentos em que estão vinculados a características da grafia e portanto mais relacionados a técnica necessária a ser aplicada para realizar a sua correção. A divisão dos erros por grupos permite elaborar algoritmos que definem com exatidão a qual classe de problemas que pretende resolver, por exemplo, suponha a existência de seis classes de erros **A**, **B**, **C**, **D**, **E** e **F**; um novo trabalho pode mostrar que a literatura apenas resolve os problemas **A**, **B**, **C** e **D** e que sua solução resolve os problemas **E** e **F**, deixando explícita sua contribuição e permitindo o autor focar exclusivamente nos dois grupos que estão bem definidos devido à categorização.

Seabra *et al.* (2011) propõem uma divisão em três categorias (Tabela 2) as quais são aplicadas no contexto de crianças em processo de aprendizagem, visando analisar o desenvolvimento da alfabetização, além da identificação e controle de distúrbios e limitações. Quando as crianças apresentam apenas as falhas de terceira ordem, elas são consideradas alfabetizadas.

Tabela 2 – Categorização segundo Seabra et al. *(2011)*

Categoria	**Explicação**
Falhas de primeira ordem	Falhas cometidas pela leitura lenta, com soletração de cada sílaba. A escrita é baseada na correspondência linear entre as sequências dos sons e das letras, tendendo a repetição (“ppai” ao invés de “pai”), omissão (“trs” ao invés de “três”), inversão (“parto” ao invés de “prato”), falhas decorrentes do conhecimento ainda inseguro do formato de cada letra (“rano” ao invés de “ramo”) e a incapacidade de identificar algum som (“sabo” ao invés de “sapo”).
Falhas de segunda ordem	Falhas mais propícias à arbitrariedade, na qual a escrita é tomada como uma transcrição fonética da fala, partindo do princípio de que na leitura cada letra pronunciada tem a sua representação central o que leva, por exemplo, as seguintes falhas: “matu” ao invés de “mato”, “tenpo” ao invés de “tempo”, “genrro” ao invés de “genro”.

Falhas de terceira ordem	Falhas limitadas as trocas entre letras concorrentes, isto é, com semelhanças de som, cujos erros são gradativamente superados. Alguns exemplos de erros são: “assúcar” ao invés de “açúcar”, “sau” ao invés de “sal”, “xinelo” ao invés de “chinelo”.

Uma outra categorização proposta tem relação com a origem do erro (Tabela 3). Esta também é dividida em três categorias, sendo as duas primeiras erros que ocorrem desde os primórdios da escrita (KUKICH, 1992), enquanto que a última surgiu há mais ou menos vinte anos com os microblogs e aplicativos de mensagens instantâneas: o internetês (ANDRADE *et al.*, 2012, GONZALEZ, 2007).

Tabela 3 – Categorização guiada pela origem do erro

Categoria	**Explicação**
Erros cognitivos	Erros decorrentes do desconhecimento do escritor quanto à forma correta de escrita da palavra, por exemplo, o indivíduo escreve “cassador” ao invés de “caçador” por não saber que a palavra é grafada com **ç** e não com **ss**.
Erros tipográficos	Erros relacionados à desatenção e/ou pressa na digitação dos caracteres. Se analisadas as grafias decorrentes deste tipo de erro percebe-se uma forte relação com a disposição das teclas, por exemplo, a troca da letra **o** pela letra **i** ao se escrever “pote”, devido a ambas as letras estarem localizadas lado a lado nos teclados QWERTY. A omissão de letras, assim como a presença de letras extras, também são características dos erros tipográficos, devido a uma tecla não ser pressionada ou um esbarrão em uma tecla adicional.
Erros propositais	Erros de escrita cometidos de forma intencional. Criam-se variações de escrita para as palavras, tais que essas não seguem as regras ortográficas da língua com intuito de acelerar o processo de digitação, diminuir o tamanho das frases em virtude da limitação de seu tamanho (no caso do Twitter há a limitação de 140 caracteres por publicação) e devido a características da informalidade e desleixo quanto às normas cultas da língua.

Assim como as categorizações anteriores são guiadas pelo erro, é possível também realizar a categorização guiada pelo conserto (Tabela 4). Neste caso tem-se quatro categorias em que todos os possíveis erros de escrita podem estar classificados em uma ou mais destas, as quais mostram os tipos de passos a seguir para se corrigir a grafia (AGARWAL *et al.*, 2007).

Tabela 4 – Categorização guiada pelo conserto

Categoria	**Explicação**
Inserção	Pertencem a esta categoria as palavras que para atingirem sua forma correta necessitam fazer uma ou mais inserções de caracteres, por exemplo, para corrigir "pogramação" necessita-se inserir a letra **r**, transformando em "programação".
Deleção	Pertencem a esta categoria as palavras que para atingirem sua forma correta necessitam fazer uma ou mais deleções de caracteres, por exemplo, para corrigir "opição" necessita-se deletar a letra **i**, transformando em "opção".
Substituição	Pertencem a esta categoria as palavras que para atingirem sua forma correta necessitam fazer uma ou mais substituições de caracteres, por exemplo, para corrigir "caixite" necessita-se substituir a letra **i** por **o**, transformando em "caixote".
Transposição	Pertencem a esta categoria as palavras que para atingirem sua forma correta necessitam fazer a inversão entre um ou mais pares de caracteres na palavra, por exemplo, para corrigir "compuatdor" necessita-se inverter as letras **a** e **t**, transformando em "computador".

Kinoshita *et al.* (2005) ampliam o conceito de escrita, focando não apenas na ortografia mas também em outras questões não apresentadas nas categorias anteriores. Para tanto, criam quatro divisões como mostrado na Tabela 5.

Tabela 5 – Categorização segundo Kinoshita et al. *(2005)*

Categoria	**Explicação**
Erros ortográficos	Os erros ortográficos olham individualmente para cada palavra. São colocados nesta categoria as palavras que, da forma como estão grafadas, não pertencem ao idioma em análise. Os tipos de erros ortográficos podem ser encontrados na Tabela 4.
Erros gramaticais	Os erros gramaticais necessitam da análise em cima da sentença completa. São colocados nesta categoria as sentenças que não seguem as normas gramaticais da língua. Um exemplo de erro gramatical é apresentado na frase "Nós vai andar de carro", na qual o verbo está flexionado na terceira pessoa do singular, enquanto deveria estar na primeira pessoa do plural, apresentando assim um erro de concordância verbal.
Erros semânticos	Os erros semânticos necessitam da análise em cima da sentença completa. É classificado como um erro semântico a sentença que não apresenta uma ideia coerente, tornando assim esse grupo diretamente relacionado ao contexto da frase. Um exemplo de erro é apresentado na frase "O caminhão come bananas", na qual mostra uma ideia absurda de um caminhão realizando o ato de comer.

Erros de estilo	Os erros de estilo necessitam da análise em cima da sentença e, por vezes, em cima do texto por completo. É classificado como um erro de estilo a sentença que apresente palavras que destoem do restante do texto com relação ao uso de estruturas sintáticas complexas e utilização de palavras incomuns, causando assim prejuízos ao entendimento do leitor. Um exemplo deste tipo de erro seria no uso de palavras do hino nacional brasileiro como "plácido" e "retumbante" em contos infantis.

Por fim, tem-se a categorização que divide os erros que são identificados olhando unicamente para a palavra (erros de palavras não reais) e os erros de contexto (erros de palavras reais) que só são identificados se analisadas também as palavras ao redor (MISHRA & KAUR, 2013), a qual é apresentada na Tabela 6.

Tabela 6 – Categorização quanto a erros unitários e erros de contexto

Categoria	**Explicação**
Erros de palavras reais	Consiste de palavras válidas no idioma, todavia no contexto em que aparecem estão erradas. Esse tipo de erro necessita da análise do contexto para sua identificação.
Erros de palavras não reais	Consiste de palavras inválidas no idioma, as quais não são encontradas nos dicionários. Para detecção deste tipo de erro necessita-se apenas a análise individual de cada palavra.

As categorizações anteriormente listadas correspondem à divisões em alto nível dos problemas na escrita. Selecionando algum dos grupos de divisões, já é possível definir melhor a quais problemas determinada solução pretende atacar. Contudo, pode-se ainda realizar uma divisão mais granular dos tipos de erros, deixando mais claros os problemas de escrita e permitindo criar formas mais apropriadas de correção para cada caso.

2.3 Proposta de categorização dos erros

Neste trabalho levantaram-se os erros que aparecem na literatura, criando uma categorização com maior nível de detalhes apresentada na Tabela 7[3].

Algumas das categorias concebidas, dependo do contexto em que se encontram, podem não ser consideradas um erro de escrita, contudo para processamentos automáticos por vezes é necessário a sua identificação para que recebam um pré-

[3] No APÊNDICE A é possível encontrar os trabalhos na literatura que estão relacionados a cada categoria criada.

processamento. No caso do aparecimento de URLs no texto por exemplo, estas precisam ser identificadas e tratadas antes da aplicação de um algoritmo de separação de sentenças, isto porque separar frases pelo aparecimento do ponto final quebraria a URL em sentenças diferentes, onde na realidade o ponto final não indica o fim de uma frase quando este se apresenta dentro de uma URL.

Tabela 7 – Proposta de categorização dos tipos de erros com alto nível de detalhes

Categoria		**Explicação**
Necessitam de identificação	*Mentions*	As *mentions*, as quais aparecem principalmente nas redes sociais, é a forma como um usuário da rede consegue citar outro usuário. São reconhecidas por consistirem de um token iniciado por **@**. Exemplo: *@maria me liga?*
	Hashtags	As *hashtags*, as quais aparecem principalmente nas redes sociais, é a forma como incluir o seu comentário em um grupo no qual compartilham o mesmo assunto. São reconhecidas por consistirem de um token iniciado por **#**. Exemplo: *#mengão campeão!*
	URLs	As URLs aparecem principalmente em textos informais ou explicativos, no qual a utilizam para fazer uma referência na Web. Exemplo: *Acessa aí http://www.cos.ufrj.br*
	Emoticons	Os *emoticons* aparecem principalmente nas redes sociais e aplicativos de mensagens instantâneas. São formados por um conjunto de caracteres que assumem a representação de uma imagem conhecida, como feições, coração, entre outras. Exemplos: *Me dei bem na prova =]* *Te amo S2*
	Onomatopeias	As onomatopeias são palavras utilizadas para representar sons. Em diversos casos podem necessitar sofrer uma normalização e/ou identificação. Exemplo: *Aaaiiii, doeu muito!* *Forma normalizada: Ai, doeu muito!*
Capitalização		Os erros de capitalização ocorrem devido ao uso incorreto da caixa alta ou caixa baixa em uma ou mais letras dentro de uma palavra. Exemplos: *JORGE, cadê você?* *Vai para o brasil ou França?*

Repetição	Caracteres	A repetição de caracteres aparece principalmente em textos informais, e são utilizadas para expressar emoção. Exemplo: *Goooooooolll do flamengo!*
	Pontuação	A repetição de pontuação funciona igualmente a repetição de caracteres. Servem para expressar emoção, contudo além de aparecerem em textos informais, algumas vezes também ocorrem em textos de história. Exemplo: *Achei dinheiro na rua!!!!!!*
Abreviação	Acrônimo reconhecido	Os acrônimos reconhecidos consistem de uma forma compacta de representação de uma ou mais palavras. São caracterizados por já serem conhecidos e aceitos pelo idioma em sua norma culta. Exemplos: *Eu sou do RJ, e você?* *Hoje vou me consultar com o Dr. Sérgio.*
	Truncamento	Os truncamentos consistem de uma forma compacta de representação de uma palavra. É caracterizado como um truncamento quando a palavra é subitamente cortada (em seu início ou no seu final). Exemplos: *Vai para a facul hoje?* *Ontem eu tava cansada.*
	Acrônimo não reconhecido	Os acrônimos não reconhecidos consistem de uma forma compacta de representação de uma ou mais palavras. Aparecem em textos informais, principalmente em redes sociais e aplicativos de mensagens instantâneas, e são caracterizados por não serem reconhecidos na norma culta da língua. Exemplos: *Chegou minha msg?* *Vc vai hoje tbm?*
Junção		O erro de junção de palavras ocorre devido a mais de uma palavra aparecerem juntas. Exemplo: *Meudeus! Que tédio!*
Separação		O erro de separação de palavras ocorre devido a uma palavra aparecer separada. Exemplo: *Estou mega a n i m a d a.*
Omissão de pontuação	Separação de sentença	A omissão de pontuação na separação de sentenças é caracterizada quando duas frases que deveriam estar separadas não estão por conta da falta da pontuação (na maioria dos casos pela falta do ponto final). Exemplo: *Chegou bem em casa Sim.* *Forma correta: Chegou bem em casa? Sim.*

	Composição da palavra	Algumas palavras são compostas por pontuação. Essa categoria de erros é caracterizada quando esta pontuação não aparece na palavra. Esse conjunto de palavras é mais comum de ocorrer na língua inglesa, porém também ocorrem em português. Exemplos: *Me dá um copo dágua* *Forma correta: Me dá um copo d'água* *Im hungry!* *Forma correta: I'm hungry!*
Hífen	Inserção	Essa categoria é identificada pela inserção indevida do hífen, ou seja, usá-lo em um local inapropriado. Exemplos: *Quero comprar um novo micro-ondas*
	Omissão	Essa categoria é identificada pela omissão do hífen, ou seja, deixá-lo de usar no local apropriado. Exemplo: *Estou procurando meu guardachuva* *Forma correta: Estou procurando meu guarda-chuva*
Acentuação	Inserção	Essa categoria é identificada pela inserção indevida de um acento. Exemplo: *Ólha aqui!*
	Omissão	Essa categoria é identificada pela omissão de um acento. Exemplo: *Olha o meu relogio!*
	Substituição	Essa categoria é identificada pela substituição do acento correto a se utilizar. Exemplo: *Você vem?*
Representação fonética	Letra	Essa categoria é identificada pelo uso do fonema ou nome da letra para representar a escrita da palavra. Exemplos: *Estou esperando sua xamada!* *Já estou chegando em kza!*
	Número	Essa categoria é identificada pelo uso do nome do número para representar a escrita da palavra. Exemplo: *Tem 9dades?*
	Símbolo	Essa categoria é identificada pelo uso do nome do símbolo para representar a escrita da palavra. Exemplo: *Você está d+!*
Erro de contexto		Essa categoria se caracteriza pela análise em cima de sentenças e não em cima de cada palavra individualmente. Neste caso, todas as palavras se verificadas individualmente existem, contudo ao analisá-las juntas se constata algum erro (como erros de regência, concordância, etc.). Exemplo: *O menina está com fome.*

Erro de digitação	Inserção	Essa categoria é identificada pela digitação adicional errada de um ou mais caracteres. Exemplo: *Olha a minha nova canketa.* *Forma correta: Olha a minha nova caneta*
	Deleção	Essa categoria é identificada pela digitação errada com a omissão de um ou mais caracteres. Exemplo: *Cadê meu relógo?* *Forma correta: Cadê meu relógio:*
	Substituição	Essa categoria é identificada pela digitação errada com a substituição de um ou mais caracteres corretos. Exemplo: *Esse é meu caixite!* *Forma correta: Esse é meu caixote!*
	Transposição	Essa categoria é identificada pela digitação errada com transposição entre um ou mais pares de caracteres: Exemplo: *Vou mexer no compuatdor!* *Forma correta: Vou mexer no computador!*
Dialeto informal		O dialeto informal é caracterizado quando em um texto se encontra palavras usadas na linguagem informal, muitas vezes caracterizado pelo regionalismo. Exemplo: *Hoje quero 2 cacetinhos e 1 vara.* *(Na Bahia, o pão francês é conhecido como cacetinho e a bisnaga como vara)*
Erros específicos da língua	am versus ão	Essa categoria é identificada pelo uso errado da terminação **ão** e **am** nas palavras em português. Nos dois casos os fonemas são parecidos, porém apresentam significados diferentes (o primeiro correspondendo a uma ação futura e o segundo uma ação no passado). Exemplos: *Vocês esperarão muito ontem minha mãe chegar?* *Forma correta: Vocês esperaram muito ontem minha mãe chegar?* *Amanhã vocês esperaram na portaria do hotel.* *Forma correta: Amanhã vocês esperarão na portaria do hotel.*
	Porque / Porquê / Por que / Por quê	Essa categoria é identificada pelo uso errado das variações da palavra "porque". Esta, em todas as suas formas, têm o mesmo significado, porém dependendo do tipo de frase (interrogativa ou não) e da sua posição dentro da frase (início, meio ou fim), deve-se utilizar um determinado tipo de grafia. Exemplos: *Porquê hoje acabou tarde?* *Forma correta: Por que hoje acabou tarde?* *Não sei o por que.* *Forma correta: Não sei o porquê.*

Censo	Essa categoria é caracterizada quando um conjunto de palavras são disfarçadas no texto por serem de baixo calão. Exemplo: *Me cortei! Que m****.*

2.4 Características dos microblogs e aplicativos de mensagens instantâneas

A seção anterior apresentou variados formatos de categorização para os erros de escrita. Em várias das categorias a probabilidade de cometer o erro está diretamente vinculada ao desconhecimento do usuário em como escrever a palavra ou a problemas de digitação, por exemplo, em um computador o erro pode estar vinculado à disposição espacial das letras no teclado. Todavia, os erros que correspondem ao internetês seguem outra distribuição de probabilidade que é desconhecida, uma vez que são cometidos de forma proposital e não dispõem de regras que a norteiem. Devido a sua imprevisibilidade, o internetês para a língua portuguesa é explorado com maior nível de detalhes nessa seção.

O internetês começou a aparecer nas duas últimas décadas no ambiente Web principalmente utilizado pela geração jovem nos blogs, flogs, redes sociais e aplicativos de mensagens instantâneas. Para Bisognin (2008) a criação desse novo tipo de escrita é fruto da influência de vários fatores como a fala, escrita, necessidade de ênfase, acentuação, cifras e criptografia, pontuação e norma culta. Em seu trabalho, realiza a exploração de textos em português presentes no Orkut e, dentre os resultados obtidos, destaca-se a análise do tamanho médio das palavras (apresentado na Figura 1). Ao observar os dados gerados, nota-se uma maior concentração de palavras formadas por duas e três letras, decrescendo sua frequência à medida que o tamanho da palavra aumenta. A existência de palavras pequenas aparece como principal característica do internetês, onde se busca a economia de caracteres possibilitando a digitação mais veloz, além de ajudar a suprir restrições de limitação do tamanho da mensagem, encaixando mais informações em uma frase com menos letras.

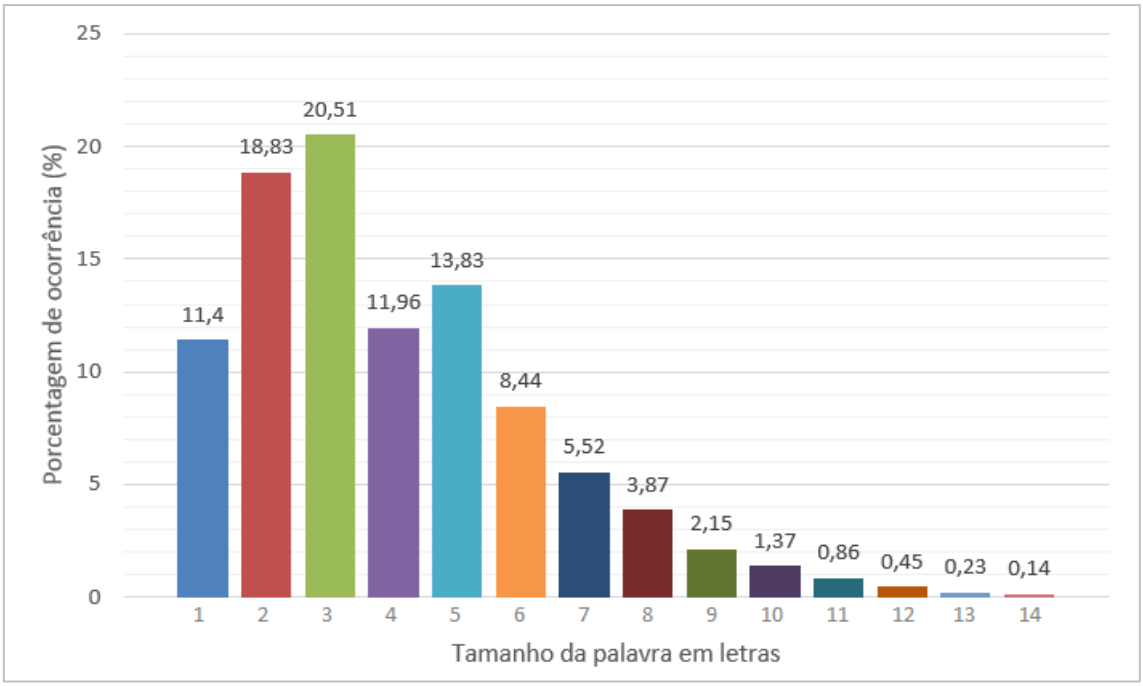

Figura 1 - Frequência do aparecimento das palavras, agrupadas pelo seu tamanho, no trabalho de Bisognin (2008)

Gonzalez (2007) por sua vez realiza análises em cima de textos, também em português, retirados de 98 blogs e, como mostrado na Figura 2, uma proporção similar é encontrada, onde a maior parte das palavras são compostas por duas e três letras. A análise realizada em dois trabalhos, com duas bases de fontes diferentes (Orkut e blogs), mostram um certo padrão de escrita no internetês, o qual pode ser explorado.

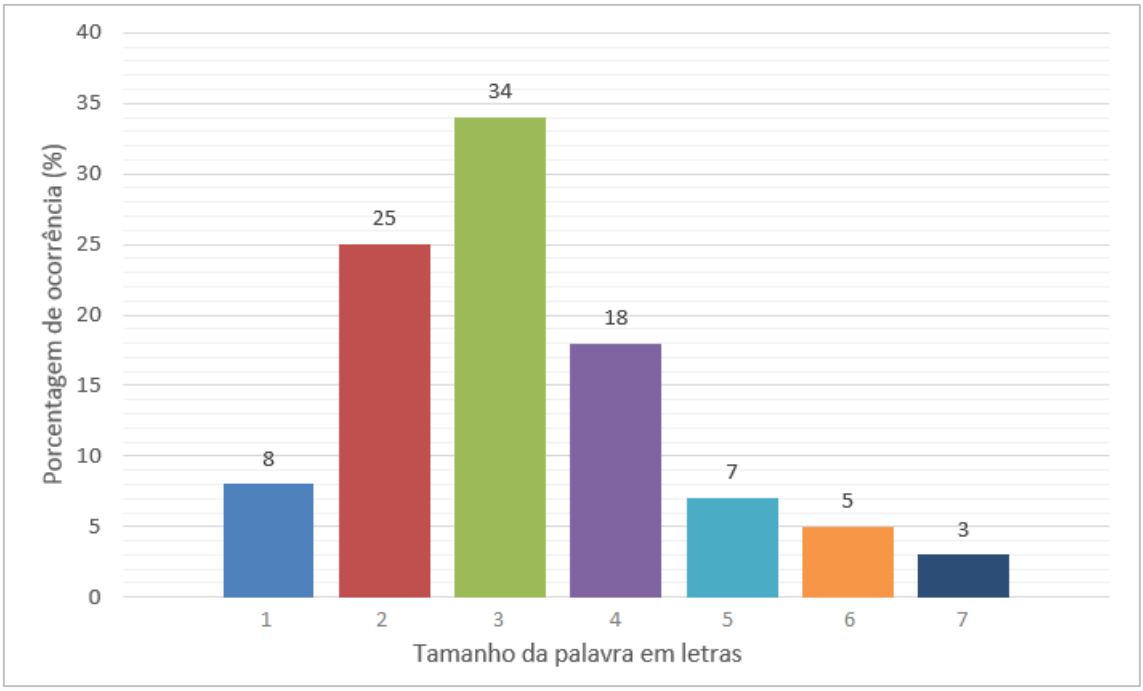

Figura 2 - Frequência do aparecimento das palavras, agrupadas pelo seu tamanho, no trabalho de Gonzalez (2007)

Em uma análise em cima da classe gramatical das palavras, investigou-se a frequência de aparecimento destas para o internetês e norma culta, como mostrado na Figura 3, onde para cada classe gramatical é apresentado a porcentagem da frequência dos termos em internetês e na norma culta. Os advérbios, preposições, pronomes e interjeições aparecem como as classes mais modificadas na grafia do internetês, pois mesmo não correspondendo a classe com maior frequência de aparecimento, são as que têm sua frequência no internetês superior a quantidade de aparições na norma culta (GONZALEZ, 2007).

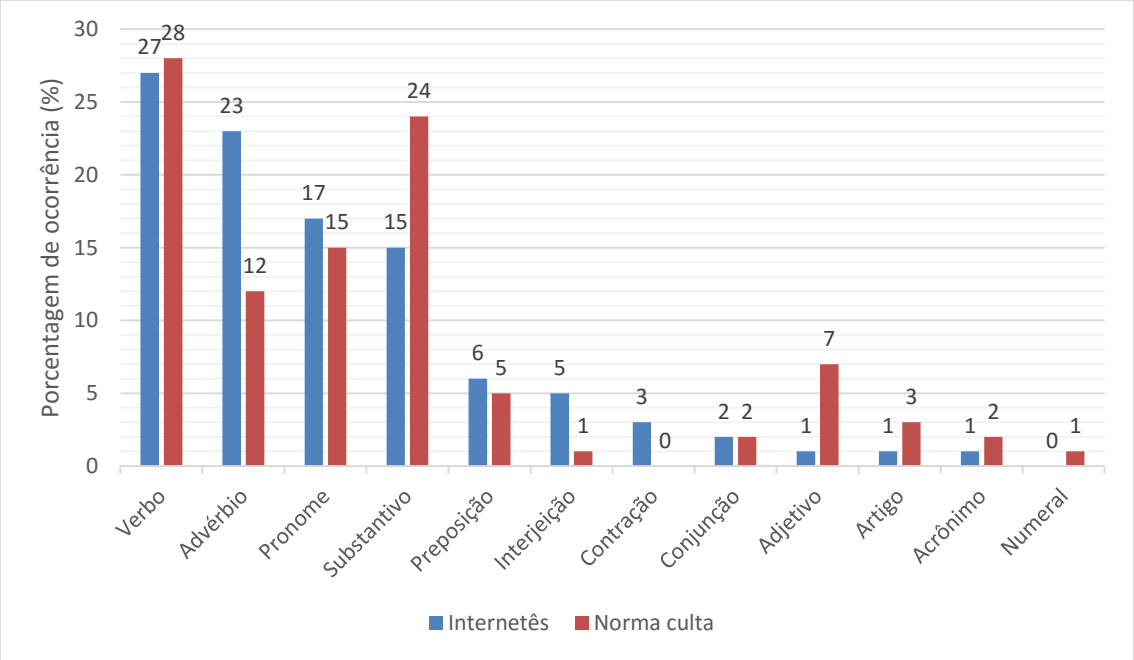

Figura 3 - Relação entre a frequência de aparecimento dos tokens em internetês e segundo a norma culta quanto à sua classe gramatical (GONZALEZ, 2007)

Os verbos por sua vez não tem um percentual de modificações alto, contudo sua frequência é a maior de todas e portanto em números absolutos são os mais alterados, merecendo assim uma atenção especial. Em muitos casos de modificação em verbos, a alteração sofrida ocorre quando esse está no infinitivo e a letra **r** que termina o token é eliminada devido a influência da oralidade na escrita da palavra (GONZALEZ, 2007).

Para escrever uma palavra com um número menor de letras é necessário realizar a supressão destas. Dentre as omissões das vogais destacou-se a porcentagem elevada de eliminação da vogal **e** em relação as demais, a qual ocorre devido ao fato de muitas letras do alfabeto (como o **b**, **c**, **d**, **g**, **p**, etc.) sonorizarem a vogal **e** ao pronunciar seu nome. Acerca da exclusão de consoantes destaca-se a supressão de **r** e **s**, que por influência da oralidade são eliminadas principalmente em finais de verbos e palavras no plural; **m** e **n**, tipicamente eliminadas quando estão exercendo a função de nasalização

(como por exemplo em “tb” como “também” e “qd” como “quando”); e **h**, em geral omitidos do início das palavras, uma vez que não mudam o som na leitura destas (GONZALEZ, 2007).

Além da eliminação das letras, por vezes é comum acontecer a substituição destas, que ocorre por motivos da influência da oralidade, ou diminuição da quantidade de caracteres por token. As substituições mais frequentes encontradas correspondem a letra **o** por **u** no final das palavras, **qu** por **k**, **ch** por **x**, **ca** ou **c** por **k** e **e** por **i** (GONZALEZ, 2007).

Aliado à busca pela redução do número de caracteres, existe a tentativa em diminuir a quantidade de toques necessários para realizar a escrita, fazendo assim diferenciar o caractere **a** de **á**, onde o primeiro necessita de um toque enquanto o segundo de dois toques, um para o acento e outro para a letra. Na análise de Gonzalez (2007) identificou-se que em 84,5% dos casos as palavras em internetês apresentam economia de toques, enquanto que 14,5% não apontam mudança na quantidade de toques e 1% apresentam aumento. Das palavras com economia de caracteres constatou-se que 45% das palavras em internetês se diferenciam em um em quantidade de toques necessários para digitar a palavra em internetês e na norma culta, 24% em dois toques, 9% em três toques e 1% em quatro ou mais toques (caso correspondente as palavras “aniversário” representada por “niver”, e “fim de semana” representada como “fds”).

As modificações sofridas pelas palavras na norma culta resultam em tokens em internetês com significados diferentes e escritas iguais, causando assim possíveis ambiguidades no entendimento e necessidade de contexto para compreensão (GONZALEZ, 2007). Após estudos, Bisognin (2008) cria dezessete categorias em que os termos em internetês podem se apresentar, mostradas na Tabela 8.

Tabela 8 – Categorização do internetês segundo Bisognin (2008)

Categoria	**Exemplos (internetês → norma culta)**
Indicação de monossílabos por uma simples letra	q→que / d→de / t→te / p→pra / m→me / c→com.
Substituição do acento agudo pela letra h no final da palavra	eh→é / neh→né / tah→tá / lah→lá / jah→já / poh→pó / feh→fé / voh→vó
Reprodução da fala	u→o / ki→que / aki→aqui / du→do / cum→com / gent→gente / loka→louca
Nasalização indicada por UM ou UN em final de palavra	taum→tão / naum→não / naun→não / bjaum→beijão / paixaum→paixão

Sequência de consoantes representando uma palavra, sem uso de vogais	pq→porque / cmg→comigo / vc→você / ngm→ninguém / qlqr→qualquer
Várias formas para um mesmo vocábulo	Mto,mtu,mt,mtoo→muito / tbm,tbm,tbem→ também / bjos,bjus,bjxx,bjs,beeeeijos→beijos
Registro sem acentuação	vo→avó / so→só / pa→pá
Palavra com ausência de uma letra	fla→fala / pod→pode / nunk→nunca / kra→cara / tmpo→tempo / plas→pelas
Onomatopeias para riso e choro	Hehe, hahaha, rsrsrs, kkkk, huaauauahau, heieheiheie, shushaushua
Repetição de letra para indicar intensidade	muitoooooo→muito / nadaaaaaa→nada / disculpaaaaa→desculpa
Redução do nome de pessoas	Biel→Gabriel / Lau→Laura / Pri→Prscila / Ro→Roberto
Criações especiais, as quais só são entendidas no contexto	pah (eh mtu kiridu i companheru pah todas as horas...) → para (É muito querido e companheiro para todas as horas...) txi (bjuxxx negááááá!!! Txi amooo) → Te (Beijos nega! Te amo)
Repetição de sinais de pontuação para enfatizar sentimento	To bem e tuh??? / ihhh ta velho hein?!??!? / ok????????
Supressão de sinais de pontuação que marcam fronteiras oracionais	Eii resenhaa hahahahahah vejo isso é o tira fosto língua!!! → Eii resenhaa! hahahahahah! Vejo isso é o tira fosto língua!!!
Substituição de palavras e expressões por símbolos ou algarismos	T+→até mais / d+→demais / 9dades→novidades / v6→vocês
Transformação de expressão ou fraseologia em siglas	Tdb→tudo de bom / fds→fim de semana / rdtr→rolando de dar risada
Uso de caracteres ou *emoticons*	Desculpa >.< bjos / to tow beem \o/

2.5 Considerações finais

Neste capítulo foi apresentado o enredo que norteia erros de escrita, no qual explorou-se suas características e tendências. Mostrou-se as categorizações de erros encontrados na literatura, uma proposta de categorização com maior nível de detalhes, e um estudo aprofundado dos textos em internetês, no qual embora não seja regido por regras, apresenta algum tipo de padrão e que pode ser explorado.

3 Técnicas de solução de detecção e correção automática

Para o problema de detecção e correção automática de textos, múltiplas técnicas têm sido desenvolvidas. Estas variam de abordagens mais simples até soluções mais elaboradas. A diversidade de complexidades e limitações que as técnicas possuem tem como uma de suas explicações o objetivo da correção e o tipo de texto a ser tratado, pois em certas situações é necessário apenas que uma parcela dos erros sejam corrigidos, assim como dependendo do texto em análise pode-se fazer algumas suposições para elaboração da solução.

Neste capítulo são mostrados os diversos tipos de técnicas que existem na área de detecção e correção automática de textos, abordando soluções de textos em linguagem culta e coloquial (a qual engloba o internetês).

3.1 Grupos de técnica de normalização de textos

As soluções existentes na literatura pertencem a dois tipos de abordagens, as quais correspondem a técnicas com: (1) estrutura e parâmetros fixos e (2) estrutura e parâmetros variáveis. Ambos os grupos dispõem de uma grande quantidade de soluções já desenvolvidas que diferem em desempenho e qualidade.

Sabendo apenas o tipo de técnica que duas soluções distintas utilizam não é possível determinar qual destas tem maior qualidade, entretanto as que utilizam de estrutura e parâmetros variáveis têm maior potencial de ser uma solução robusta que trata de uma maior quantidade de casos. Este grupo de técnicas utiliza o aprendizado de máquina, no qual sua solução estabelece parâmetros ou estruturas variáveis que, utilizando de uma base de treinamento, parte de uma solução inicial e a aperfeiçoa à medida do tempo, chegando a uma solução melhor adaptada ao problema. Já o primeiro grupo consiste de soluções fixas que desde sua concepção apresentam a mesma estrutura e parâmetros de execução, não tendo a necessidade de uma fase de treinamento e nem se aperfeiçoando à medida do tempo.

Uma vez que o problema de correção de textos deve englobar uma grande quantidade de casos de tratamento, conceber uma estrutura fixa e determinar o valor dos parâmetros de maneira manual pode limitar o alcance da solução. Em contrapartida, é possível desenvolver técnicas que repassem ao computador a tarefa de encontrar a melhor estrutura, fluxo e parâmetros para determinado problema. Acerca do conhecimento necessário para se realizar a normalização de textos, enquanto o primeiro grupo o adquire através dos passos pré-estabelecidos embutidos na solução criada pelo

desenvolvedor, no segundo grupo esse conhecimento é adquirido na fase de treinamento pela máquina.

3.1.1 Técnicas sem aprendizado de máquina

Inúmeras técnicas já existentes na literatura não fazem uso do aprendizado de máquina e por este motivo consistem de soluções fechadas e que tendem a utilizar conceitos mais simples. Contudo, para determinados cenários a sua aplicação já é o suficiente para resolução do problema, não sendo necessário gastar esforços em métodos mais elaborados de normalização.

Asonov (2010) desenvolve uma solução de normalização de textos escritos em linguagem culta que tem por objetivo identificar erros de contexto, ou seja, verificar se duas palavras juntas seguem as regras gramaticais da língua. Deste modo, conta com um léxico no qual contabiliza a frequência de aparecimento de cada vocábulo do idioma individualmente e para cada par de palavras, onde a ordem destas tem papel importante. De posse desses valores, para cada duas palavras no texto calcula-se a probabilidade destas consistirem de um erro gramatical, onde este é reconhecido quando há uma combinação de palavras pouco frequente em relação ao aparecimento individual de cada um dos vocábulos.

Huang *et al.* (2013), por sua vez, normalizam textos que tratam de um domínio específico, o diagnóstico de conserto de carros. Em casos como este, termos técnicos são utilizados, fazendo-se necessário o conhecimento adicional do domínio para sua identificação e correção. Assim, a solução conta com um léxico para o idioma e um adicional com os vocábulos específicos do domínio, o qual é consultado para identificar os termos técnicos e abreviações. Huang *et al.* (2013) ainda usam uma tabela adicional contendo os principais erros cometidos na escrita e sua respectiva correção, inserindo de forma direta o conhecimento necessário para realizar a normalização dos textos.

Enquanto as soluções anteriores focam na normalização de textos onde se assume principalmente a existência de erros cognitivos e tipográficos, a correção de textos com a presença do internetês precisa de um tratamento diferente e/ou adicional, uma vez que as características de escritas são modificadas e a quantidade de erros é superior. Os sistemas HASCH (***H****igh Performance* ***A****utomatic* ***S****pell* ***C****hecker*) (ANDRADE *et al.*, 2012) e CECS (***C****asual* ***E****nglish* ***C****onversation* ***S****ystem*) (CLARK & ARAKI, 2011) correspondem a soluções de normalização de textos em internetês para o português e inglês respectivamente. Estes contam com dois léxicos, nos quais o primeiro contém as palavras do idioma segundo a norma culta enquanto o segundo aos

termos em internetês mais frequentes nos textos e seu respectivo mapeamento para o vocábulo correto. Ambas as técnicas buscam primeiramente o termo no conjunto correto de palavras e, caso não encontrado, o procura entre os termos em internetês.

O HASCH, por se tratar de um corretor de português, realiza um tratamento extra para a acentuação. Na ocasião em que não se obtiver nenhum resultado quando realizada a busca nos dois léxicos, acrescenta-se os acentos disponíveis do idioma na palavra para que a busca seja novamente feita, porém agora com as palavras alteradas. No caso da nova busca também não encontrar nenhum candidato para normalização da palavra, é calculada a distância de edição para fazer a correção e encerrar o algoritmo. O CECS por sua vez apresenta dois outros tratamentos: (1) a substituição dos números por letras, quando estiverem desempenhando o papel fonético de uma sequência de caracteres (por exemplo "*4get*" como "*forget*") e (2) tratamento de ambiguidade na correção, onde para uma determinada palavra não há como saber se ela está certa ou errada a menos que se analise o contexto, apresentando assim uma abordagem com análise de n-gramas. Este último sistema é utilizado como uma fase de pré-processamento do internetês nos textos, utilizando em seguida de sistemas de correção de erros clássicos para complementar a normalização.

O sistema TwitIE (BONTCHEVA *et al.*, 2013) consiste de uma aplicação especializada em textos de microblogs para extração de conteúdo, que conta com uma fase intermediária de normalização de textos. Para tal, faz uso de outra heurística que tem relação direta com o internetês: a influência da oralidade na escrita. Esta influência é principalmente encontrada com as letras representando foneticamente uma sequência de caracteres (GONZALEZ, 2007), e por esse motivo utiliza das distâncias de edição de Levenshtein nos vocábulos em seu formato original e foneticamente codificado para eleger o melhor candidato a correção.

Avanço *et al.* (2014) por sua vez criam um codificador fonético especialista para a língua portuguesa (o PB-Rules), o qual é mais restritivo que os demais, podendo assim oferecer como sugestão de correção um conjunto menor de possibilidades que facilite o processo de escolha do melhor candidato à normalização. Em seu algoritmo de correção, utiliza a combinação de três técnicas (com a respectiva prioridade): PB-Rules, Soundex e distância de edição de Levenshtein. Para tratar com mais exatidão as características presentes nos textos em internetês, um segundo trabalho (DURAN *et al.*, 2015) propõe técnicas adicionais como o tratamento da separação de tokens que deve

levar em consideração o aparecimento de *emoticons*, e uma alternativa de como tratar sugestões que diferenciem apenas na acentuação sem fazer uso do contexto.

Com o objetivo de se gerar a lista de candidatos a correção para um determinado vocábulo, além de poder realizar a aplicação de alguma das opções supracitadas, é possível fazer uso de ferramentas já existentes. Duarte (2013) faz uso do Hunspell[4], o qual consiste de um corretor ortográfico que é utilizado atualmente nas plataformas LibreOffice, OpenOffice.org, Mozilla Firefox 3 e Thunderbird (HUNSPELL, 2014). Hu (2013) por sua vez elenca os candidatos através do uso do Aspell[5], no qual consiste de uma aplicação concorrente do Hunspell que apresenta resultados superiores com base no estudo de Aspell.net (2004).

Com as listas de candidatos a correção geradas, Hu (2013) faz a seleção do melhor postulante utilizando da frequência de aparecimento deste em uma base de trigramas. As tuplas $<p_1\ p_2\ p_3>$ que são consultas na base são formadas por p_2 contendo o candidato a correção do vocábulo **v** em questão, e p_1 e p_3 correspondendo a palavra anterior e posterior a **v**. Os trigramas mais frequentes são escolhidos para formar a correção do texto, utilizando assim diretamente do contexto para a realização da normalização.

Ao invés do uso das palavras como menor unidade a ser manipulada, Macedo e De Moraes (2010), para fazerem a seleção de um token candidato, recorrem ao manuseio dos trigramas[6] formados pelas letras dos vocábulos. Assim, selecionam o aspirante que contiver a maior quantidade de trigramas (formados por suas letras) em comum com os trigramas do token original.

A utilização de n-gramas não apenas está restrita a escolha do candidato, mas também é usada para evitar que determinado vocábulo inicialmente correto seja erroneamente substituído. Como existem tokens corretos que dificilmente constam nos léxicos, como por exemplo nome de pessoas e lugares, Sidarenka *et al.* (2013) propõem uma abordagem que utilizam da frequência de unigramas e bigramas para determinar se o vocábulo deve permanecer intacto ou deve ser corrigido.

3.1.2 Técnicas com aprendizado de máquina

A utilização do aprendizado de máquina nas soluções de normalização de textos permite capturar características dos erros de grafia onde a percepção humana pode

[4] http://hunspell.sourceforge.net/
[5] http://aspell.net/
[6] É importante notar que na literatura os termos n-grama, bigrama e trigrama, são usados indistintamente para identificar os agrupamentos quando feitos com letras e com palavras.

apresentar dificuldade de identificar um padrão. Para a sua utilização é possível fazer uso de diversas soluções as quais são apresentadas a seguir.

Choudhury *et al.* (2007) desenvolvem uma solução usando cadeias de Markov, utilizando-a para modelar diversos tipos de erros de escrita, como digitação trocada, omissão de um caractere, uso da representação fonética e presença de letras extras. Para isso, em uma primeira etapa realiza a criação da estrutura da cadeia e posteriormente assinala as devidas probabilidades, onde faz uso de um corpus de treinamento. De posse da cadeia criada, é possível estimar a probabilidade de ocorrência de qualquer tipo de erro e portanto utilizá-la no auxílio de sua correção.

O sistema PENN (***P**ersonalized **E**rror Correction Using **N**eural **N**etwork*) desenvolvido por Garaas *et al.* (2012), por sua vez, faz uso das redes neurais para aprender a corrigir os textos. A técnica desenvolvida consiste em capturar o que os usuários digitam em um editor de texto e identificar as ocorrências de deleção e substituição de caracteres por outros. Quando essas ocorrências acontecem são então armazenadas o estado anterior as substituições ou deleções e o estado posterior, representando respectivamente assim ao vocábulo errado e sua respectiva correção. Com esse sistema, a solução se especializa e aprende à medida que os usuários utilizam o editor onde a solução está acoplada para captura das palavras. Contudo, o sistema pode aprender conhecimento errado, como por exemplo para palavras parecidas ("*their*" e "*there*") que por vezes são confundidas na digitação, mapeando uma palavra válida em outra palavra válida.

Han e Baldwin (2011) realizam a correção de textos em internetês em que, para elegerem os candidatos a correção de um token, processa a eliminação de caracteres repetidos, para em seguida calcular as distâncias de edição fonética e ortográfica visando a identificação da correção. Com o candidato escolhido, utiliza do aprendizado de máquina através da aplicação do classificador SVM (utilizando uma base limpa do Twitter para treinamento) para confirmar se o vocábulo de fato apresenta incorreção ou se trata de uma palavra válida, mesmo que não presente no léxico (como os nomes de pessoas por exemplo).

Coetsee (2014) modela o problema de normalização através dos *Conditional Random Fields* (CRF), que usa um *dataset* de tweets para treinar o modelo, aprendendo a correção de textos em internetês. Para o funcionamento de sua técnica, utiliza quatro módulos: (1) um tokenizador, (2) um gerador de candidatos para vocábulos com erros,

(3) um classificador para identificar os tokens que precisam de correção e (4) um modelo de linguagem.

Uma das principais características do internetês é o uso indevido da capitalização, onde em diversos casos são usados para enfatizar, expressar emoção e sentimento, não sendo assim fiel às regras de escrita da língua. Nebhi *et al.* (2015) fazem uso de um classificador SVM para detectar capitalizações erradas e cadeias de Markov para identificar a sua forma correta de escrita, facilitando assim posteriormente a normalização do texto no que tange aos demais erros.

3.1.3 Técnicas utilizando SMTs

Uma abordagem amplamente utilizada nas soluções de normalização de textos, e que por este motivo é analisada separadamente das demais soluções, consiste na aplicação das máquinas estatísticas de tradução (do inglês *Statistical Machine Translation* - SMT) (BROWN *et al.*, 1993). As SMTs têm como objetivo transformar uma frase f_1 (em um determinado idioma) em uma frase f_2 (em um outro idioma), ou seja, realizar a tradução automática de um texto.

Com uma simples analogia, mostra-se que as SMTs também podem ser utilizadas para realizar a normalização de textos, onde a frase f_1 estaria regida por um léxico L_1 e um conjunto de regras ortográficas e gramaticais R_1, enquanto que f_2 estaria regida respectivamente por L_2 e R_2. Uma vez que um idioma é definido por um conjunto de palavras e regras, a frase errada pode ser entendida como escrita em um idioma com os vocábulos de L_1 (os quais correspondem a todas as palavras corretas e as possíveis incorreções) e regido pelas regras R_1 (extremamente flexíveis que permitem a existência de erros ortográficos e gramaticais), enquanto que a frase normalizada escrita em um idioma com L_2 (exclusivamente consistido de palavras pertencentes ao idioma) e R_2 (correspondente as regras ortográficas e gramaticais do idioma).

A solução do problema de normalização de textos através do uso das SMTs é formalizada matematicamente por Shannon (2001) com a metáfora do canal de ruídos, o qual assume um processo de comunicação entre um autor e um receptor. O primeiro emite uma mensagem $\boldsymbol{c}$ (limpa[7]) por um canal de comunicação imperfeito, enquanto que o segundo a recebe com ruídos (a qual chamamos a mensagem $\boldsymbol{c}$ com ruídos de $\boldsymbol{n}$ - suja[8]). O objetivo é descobrir a mensagem $\hat{\boldsymbol{c}}$ com maior probabilidade de ser a real mensagem $\boldsymbol{c}$. Assim, o problema é expressado matematicamente como:

[7] Limpa do inglês *clean*.
[8] Suja do inglês *noisy*.

$$\hat{c} = arg\max_{c}\big(\Pr(c|n)\big)$$

Em que aplicando o teorema de Bayes encontra-se:

$$\hat{c} = arg\max_{c}(\Pr(n|c) \times Pr(c))$$

As duas probabilidades presentes na fórmula apresentam os seguintes significados:

- *Pr*(n|c): conhecida como canal com ruído (*noisy channel*), ela diz a probabilidade de se gerar a frase suja ***n*** dado que se tem a frase limpa ***c***;
- *Pr*(c): conhecida como modelo fonte (*source model*), ela diz a probabilidade de ocorrência de cada frase limpa ***c***.

Entretanto, para as SMTs serem aplicadas ao contexto de normalização, precisam superar um problema intrínseco à sua definição: conseguir obter a probabilidade *Pr*(n|c), necessitando para isso conhecer as infinitas possibilidades de erros de cada possível frase correta, além da sua frequência de ocorrência. Assim, inúmeras técnicas utilizam de diferentes abordagens para tratar a normalização com o conceito das SMTs, as quais se dividem em dois tipos (GADDE *et al.*, 2011): (1) supervisionadas, que utilizam um corpus para treinamento contendo frases sujas e suas respectivas correções e (2) não supervisionadas, que aprendem as probabilidades de tradução de maneira alternativa. Enquanto o primeiro tipo sofre com a necessidade de um corpus de treinamento, o segundo padece de não tratar diversos tipos de erros de escrita devido a forma de simulação da probabilidade *Pr*(n|c).

Contractor *et al.* (2010) utilizam uma abordagem não supervisionada que faz uso de medidas de similaridade para simular a probabilidade *Pr*(n|c). Essas medidas são desenvolvidas pelos autores a partir da observação da ocorrência dos erros, e são usadas para criar uma lista ponderada com os candidatos a normalização para cada um dos vocábulos. Com todas as palavras contendo seus respectivos candidatos, e auxílio de um modelo de linguagem e um *decoder*, supõe-se a melhor frase limpa $\hat{\boldsymbol{c}}$.

Um modelo de linguagem é responsável por conter um léxico pertencente ao idioma, além das frequências de ocorrência dos tokens na base. Este por sua vez pode ser um modelo baseado em palavras (*word-based translation model*) (KNIGHT & AL-ONAIZAN, 1998), o qual contêm instâncias de palavras e as suas respectivas frequências de aparecimento em textos cultos; do mesmo modo que também pode ser um modelo mais rebuscado, como o modelo baseado em frases (*phrase-based model*) (KOEHN *et al.*, 2003), o qual se baseia em armazenar a frequência de segmentos de

textos como n-gramas. Já um *decoder* serve para, que de maneira otimizada, se encontre as traduções de maiores probabilidades entre um número exponencial de possibilidades, no caso de Contractor *et al.* (2010) é utilizado o Moses[9] como *decoder*.

Assim como Contractor *et al.* (2010), Saloot *et al.* (2015) geram as listas de candidatos à correção de um vocábulo. Contudo, para fazer uso de diversas medidas de similaridade simultaneamente, assinala diferentes probabilidades para um mesmo candidato baseado em critérios diferentes, e a partir da entropia máxima as agrupa em uma única medida.

Gadde *et al.* (2011) por sua vez desenvolvem uma técnica que utiliza ambas as abordagens (supervisionada e não supervisionada). Para superar o problema da aquisição da probabilidade *Pr*(n|c) desenvolve um algoritmo de geração de erro a partir de textos limpos, podendo assim criar um corpus para treinamento e também forçar de forma artificial os mais variados tipos de erros que possam interessar. Com a base de treinamento criada, roda-se então a SMT com o *decoder* Moses para encontrar as melhores correções.

Os trabalhos de Brill e Moore (2000) e Toutanova e Moore (2002) aplicam a probabilidade *Pr*(n|c) condicionando esta à posição dentro da palavra em que o usuário está digitando. A motivação desse condicionando está atrelada ao fato dos autores perceberem que existem determinados erros mais comuns de ocorrerem do que outros, por exemplo o termo "*ant*" nas palavras de língua inglesa é grafado de forma errônea com mais frequência quando este é um sufixo de "*relutant*" do que quando é um prefixo de "*antler*", podendo o fato estar ligado a atenção do escritor quando inicia e quando termina de escrever uma palavra. Por este motivo, o autor transforma a probabilidade *Pr*(n|c) no produto das probabilidades $Pr(t_n|t_c,pos)$, onde $\mathbf{t_n}$ corresponde ao trecho dentro da palavra errada, $\mathbf{t_c}$ a forma correta de $\mathbf{t_n}$, e **pos** a indicação da posição do trecho na palavra.

Kaufmann e Kalita (2010) utilizam as SMTs para realizar a correção de textos em internetês. Sua técnica consiste em que a probabilidade *Pr*(n|c) seja recuperada a partir de uma base de treinamento criada de forma manual. Para que o resultado tenha qualidade, mesmo que restrito pelo tamanho reduzido da base de treinamento, os autores (1) realizam uma fase de pré-processamento para normalização ortográfica, consistindo de um mapeamento **de-para** no qual termos que remetem a apenas uma possibilidade de

[9] http://www.statmt.org/moses/

correção são trocados pela sua respectiva forma correta, e realização da remoção de letras consecutivas repetidas; e (2) desambiguação sintática, na qual tem como objetivo identificar as *mentions* e *hashtags* que devem ser descartadas do texto e as que têm papel semântico na frase e portanto tem seu token inicial - **@** e **#** - eliminados para receberem os devidos tratamentos.

Quando se está em um problema de tradução, após o uso das SMTs ainda se utiliza um modelo de reordenação, visto que a estrutura linguista varia entre os idiomas, e portanto diversas vezes é necessário fazer a inversão na ordem das palavras. Contudo, para o problema de correção de textos, a reordenação na maior parte dos casos se torna dispensável.

3.2 Técnicas de normalização de texto para editores de texto[10]

A tarefa de normalização em editores de textos apresenta dificuldades diferentes das demais. Nesse caso, precisam atuar de forma dinâmica, detectando erros à medida que o usuário faz a digitação, e por vezes conta com limitações como não ter o contexto completo da frase, pois não se pode utilizar a parte da frase que ainda nem foi digitada para corrigir o que já está escrito. Todavia, editores de textos se beneficiam por usarem uma abordagem iterativa de correção, na qual utilizam a ajuda do usuário para selecionar a palavra mais apropriada, ou até optar por não substituir a palavra detectada pelo algoritmo; enquanto que os demais algoritmos anteriormente apresentados utilizam abordagens não iterativas, as quais são totalmente automáticas, verificando e corrigindo as palavras mediante a aplicação de heurísticas (CHURCH & GALE, 1991).

Dentre as aplicações existentes para o português encontram-se o CoGroo (KINOSHITA *et al.*, 2005), Curupira (MARTINS *et al.*, 2002) e o ReGra (**Re**visor **Gra**matical) (NUNES & OLIVEIRA JR, 2000, RINO *et al.*, 2002). Este último foi considerado o principal corretor automático para o português (MARTINS *et al.*, 2002) e é atualmente utilizado pelo pacote Office (REGRA, 1993). Seus funcionamentos seguem princípios semelhantes em que aplicam uma sequência de procedimentos para identificar e sugerir correções.

Para compreender os procedimentos existentes são explicadas as fases do ReGra, escolhido como exemplo por ser a principal aplicação brasileira. Ele é constituído por três módulos: (1) estatístico, o qual fornece parâmetros físicos de um texto como o número total de parágrafos, sentenças, palavras, caracteres, assim como o

[10] No inglês são conhecidos como algoritmos de *Spell Checkers*, que detectam vocábulos que podem não estar grafados de forma correta.

grau de legibilidade (MARTINS *et al.*, 1996) (que indica o nível de dificuldade de leitura do texto); (2) mecânico, que detecta erros facilmente identificáveis como palavras e símbolos de pontuação repetidos ou isolados, uso não balanceado de símbolos delimitadores (como parênteses e aspas), capitalização inadequada e ausência de pontuação no fim da sentença e; (3) gramatical, o qual é composto por um conjunto de regras heurísticas implementadas na forma de redes de transição estendidas (*Augmented Transition Networks*).

O módulo gramatical, o qual se apresenta como a etapa mais difícil do processo, é dividida em três tarefas, que são rodadas de forma sequencial. A primeira tarefa executada corresponde ao pré-processamento da cadeia de entrada, com a etiquetação dos itens lexicais e suas possíveis desambiguações. A tarefa seguinte diz respeito ao processamento gramatical propriamente dito, com verificação das escolhas feitas pelo usuário e suas possíveis correções se necessário. Para isso, faz a verificação da boa formação sintática da sentença e verificação da consistência das relações de dependência entre os mesmos.

As regras cadastradas no sistema que realizam a correção gramatical são categorizadas em pontuais e genéricas. As regras pontuais correspondem a maior parte das regras, porém com as características de terem um alcance limitado de correção, serem pouco dependentes do etiquetador sintático, e se restringem a problemas de adequação lexical e uso de padrões linguísticos já registrados nas gramáticas normativas da língua (como por exemplo o não uso de crase diante de palavras masculinas), enquanto que as regras genéricas são pouco numerosas, operam diretamente sobre a saída do etiquetador, muito abrangentes, e investigam as relações de concordância e regência entre os itens lexicais etiquetados sintaticamente. Contudo, como a saída do etiquetador é passível de falhas devido a indeterminações lexicais ou a outros acidentes de processamento, o conjunto das regras genéricas induz frequentemente a geração de erros.

Por fim, a terceira tarefa do módulo gramatical corresponde ao pós processamento sintático, com a aplicação de procedimentos de aconselhamento gramatical para readequação da cadeia de entrada aos padrões perseguidos pela ferramenta.

3.3 Dificuldades presentes em outros idiomas

A correção de textos em inglês já conta com uma vasta pesquisa e com muitas ferramentas prontas que estão à disposição para serem utilizadas, todavia para as demais línguas ainda existe uma carência tanto em pesquisa como em ferramentas. Enquanto que as bases de dados utilizadas para treinamento (em soluções com aprendizado de máquina) devem corresponder ao idioma que se deseja trabalhar, as ferramentas desenvolvidas em geral acabam por se especializar no contexto abordado, e portanto tratam com maior qualidade os textos escritos em uma determinada língua, deixando de lado os problemas específicos das demais.

Saloot *et al.* (2014) desenvolvem um trabalho visando a normalização de textos em malaio, quarto idioma mais encontrado no Twitter (HONG *et al.*, 2011), o qual conta com uma arquitetura de sete módulos em que muitos se assemelham aos já citados anteriormente, porém com um módulo específico para o idioma Malaio. Esse módulo diferenciado consiste na normalização do estilo de escrita em blogs de abreviações comuns do dialeto, que palavras compostas por dois tokens iguais seguidos tem seu segundo token substituído pelo numeral "2" (por exemplo "word-word" seria substituído por "word2"); e a abreviação da negação, a qual no dialeto é feita precedendo a palavra pelo token "*tidak*" o qual na forma coloquial é substituído por "x".

Diversas complicações são enfrentadas por idiomas que não utilizam o alfabeto latino e apresentam características diferentes. O hebraico por exemplo apresenta diversos complicadores (WINTNER, 2004), como o problema morfológico da letra י (*iud*) aparecendo como sufixo das palavras, na qual gera dois tipos de significados: (1) pronome possessivo e (2) adjetivo. No caso da palavra ביתי (*baiti*) por exemplo, esta pode significar tanto "minha casa" assim como "caseiro", tendo o contexto como única ferramenta desambiguadora. Outro ponto que traz dificuldades ao hebraico diz respeito a aparição dos morfemas, em que na maioria das línguas as palavras aparecem sozinhas (separadas por um divisor), entretanto no hebraico as preposições, conjunções, coordenações e artigos aparecem como prefixo de outra palavra, por exemplo ואתה (*veatah*) tem significado "e você", sendo composto pelas partes ו (*ve* - e) e אתה (*ata* - você). Wintner (2004) lista uma série de trabalhos realizados em cima do idioma para superar as dificuldades da língua, que ainda apresenta uma solução limitada de normalização.

Assim como o hebraico, no árabe também é possível adicionar um número alto de afixos em cada vocábulo gerando uma quantidade elevada de possibilidades de palavras. No que tange a morfologia da palavra, é possível a adição de infixos, podendo em alguns casos manter a sua classe gramatical ou não, dependendo da palavra. Outra complicação está relacionada ao fato de existirem caracteres inexistentes em muitos teclados (SHAALAN *et al.*, 2003), impossibilitando a grafia correta.

Para o chinês e outras línguas não alfabéticas há um desafio diferente das demais: não existem delimitadores entre tokens, nem mesmo o espaço em branco. Cada caractere do chinês deve corresponder a um ideograma válido, entretanto a sequência de ideogramas pode corresponder a uma sequência inválida. Como exemplo 時間 significa "tempo", todavia se o segundo ideograma for erradamente escrito como 閒, o qual consiste de uma figura muito parecida, tem-se que a nova sequência de ideogramas é formada por dois ideogramas existentes, contudo geram uma combinação inválida. Desse modo os algoritmos de detecção e correção de textos em chinês têm a tarefa de identificar sequências de caracteres inválidos e não realizar a análise individual de cada caractere. Em geral os corretores são divididos em duas partes: (1) segmentador, o qual é responsável por realizar a divisão entre os ideogramas com intuito de dividir em palavras; e o (2) detector de erro, o qual avalia se a palavra é uma sequência válida de ideogramas (LEE *et al.*, 1999).

Assim como no chinês, o japonês e tailandês também não apresentam delimitadores e precisam de tratamento diferenciado. Na busca pelo candidato a normalização do token, os algoritmos de distância de edição não conseguem ser utilizados como são feitos para o português e inglês, isto porque o tamanho médio dos tokens é menor que 2 ideogramas, e a quantidade de caracteres existentes ultrapassa a barreira dos 3000 (NAGATA, 1996).

3.4 Normalização de textos em outros domínios

Alguns outros domínios específicos também se utilizam da aplicação de correção de textos, e por características específicas podem usar técnicas diferentes das anteriormente explicadas. Cucerzan e Brill (2004) exploram a correção de escrita em consultas realizadas por usuários em máquinas de buscas. Para esse caso existe um complicador extra: o número restrito de palavras, dificultando assim a utilização do contexto para ajudar na correção. Em geral as consultas tendem a ter no máximo três palavras, e assim precisam de um tratamento diferenciado. Os autores questionam que

para esse problema é preciso escolher um *threshold* no qual define um limiar entre corrigir com maior precisão e portanto menor recuperação, deixando de fora a correção de "*donadl duck*" para "*donald duck*", ou aumentar a recuperação e diminuir a precisão trocando "*log wood*" para "*dog food*".

Conhecendo das complicações, a técnica construída por Cucerzan e Brill (2004) consiste na utilização do *log* de consultas já realizadas por outros usuários para ajudar na correção. Assim, o texto errado se transforma no formato correto de escrita atingindo pontos intermediários que são estabelecidos por consultas presentes no *log*.

A identificação e resolução de acrônimos também aparece como um tipo de correção de textos, em que tem aplicações práticas como por exemplo para auxiliar programas TTS (*Text-to-Speech*), os quais dão melhor resultado lendo o verdadeiro significado da sigla ao invés de ler o som que as letras da sigla geram. Assim, o trabalho de Pennell e Liu (2010) e sua continuação de Pennell e Liu (2011) focam na resolução de siglas geradas apenas pela deleção de caracteres da real palavra, criando um algoritmo que automatize a criação de regras com intuito de gerar uma base com acrônimos da língua inglesa para que, quando aparecerem em textos, possam ser substituídas pela real palavra.

3.5 Medidas de similaridade

Muitas das técnicas de normalização de texto apresentadas anteriormente necessitam utilizar medidas de similaridade para criar suas soluções e assim realizar a correção apropriada. Estas medidas conseguem correlacionar duas palavras através de um número, o qual pode ser comparado e eleito o melhor candidato. Dentre os algoritmos existentes têm-se (GONDIM, 2006): distância de edição de Levenshtein (LEVENSTEIN, 1965), Smith Waterman (SMITH & WATERMAN, 1981), modelo estocástico (RISTAD & YIANILOS, 1998), métrico de Jaro (WINKLER, 1999), distância de Hamming (SANKOFF & KRUSKAL, 1983), distância de edição Soundex (HALL & DOWLING, 1980), função de distância Covington (COVINGTON, 1996) e outros.

Além das medidas de similaridade, existem também as funções codificadoras, que transformam a palavra em uma outra representação que apresenta outro significado. Dentre as funções existentes destacam-se as codificações fonéticas, as quais têm como objetivo representar o som da leitura da palavra. Sendo assim, cada idioma pode conter uma série de funções que realizam a codificação em formatos diferentes, entre as quais

se destacam os algoritmos Soundex e Metaphone (PHILIPS, 2000). Uma vez codificadas, as palavras podem ter suas distâncias de edição calculadas criando assim novas medidas.

3.6 Formatos de validação

O problema de normalização de textos apresenta variados formatos de validação, os quais têm determinadas características que fazem cada solução optar por um formato. A escolha correta possibilita a análise adequada da qualidade do algoritmo, auxiliando por exemplo em técnicas que necessitem da fase de treinamento. Por este motivo é importante explorar os diversos formatos de validação e entendê-los quando cada um deve ser aplicado.

O conjunto mais simples de métricas corresponde a precisão, recuperação e fator F, as quais são encontradas nos mais variados tipos de problemas e que também podem servir para avaliar a normalização de textos (COETSEE, 2014). Sua aplicação direta, em geral, é utilizada quando não há a possibilidade de alteração no número de tokens, uma vez que não possuem fatores de normalização. A precisão é definida como a fração da quantidade de alterações corretamente realizadas sobre o número total de alterações:

$$Precisão = \frac{número\ de\ alterações\ corretas}{número\ de\ alterações}$$

Já a recuperação corresponde a fração da quantidade de alterações corretamente realizadas sobre o total de correções que deveriam ser realizadas:

$$Recuperação = \frac{número\ de\ alterações\ corretas}{número\ de\ normalizações\ que\ deveriam\ ocorrer}$$

O fator F por fim, corresponde à média harmônica normalizada da precisão e recuperação:

$$Fator\ F = \frac{2 \times precisão \times recuperação}{precisão + recuperação}$$

Para os casos em que a quantidade de tokens se modifica durante a correção é possível se utilizar da métrica *Word Error Rate* (WER) (COETSEE, 2014), expressada pela fórmula matemática:

$$WER = \frac{S + D + I}{N}$$

Onde:

- S representa a quantidade de substituições que ocorrem da frase original para a frase candidata;

- D a quantidade de deleções que ocorrem na frase candidata em relação a frase original;
- I a quantidade de inserções que ocorrem na frase candidata em relação a frase original;
- N o número total de tokens presentes na frase original.

No caso da frase “A qui está chuvendomtmsm”, por exemplo, que se tem como candidata a frase “Aqui está chovendo muito mesmo”, os valores da fórmula correspondem a respectivamente I=2 (inserção das palavras ”muito” e “mesmo”), D=1 (deleção da palavra “A”), S=2 (substituição das palavras “qui” por “aqui” e “chuvendomtmsm” por “chovendo”) e N=4 (quantidade de tokens presentes na frase original).

Combinando a ideia de precisão, recuperação e WER, tem-se o surgimento de outras métricas: as taxas de reconhecimento e corretude (COETSEE, 2014). Estas são expressadas pelas seguintes fórmulas:

$$Precisão\ de\ corretude = \frac{número\ de\ mudanças\ pelo\ token\ correto}{número\ de\ mudanças\ ocorridas}$$

$$Recuperação\ de\ corretude = \frac{número\ de\ mudanças\ pelo\ token\ correto}{número\ de\ mudanças\ que\ deveriam\ ocorrer}$$

$$Precisão\ de\ reconhecimento = \frac{número\ de\ mudanças\ no\ local\ correto}{número\ de\ mudanças\ ocorridas}$$

$$Recuperação\ de\ reconhecimento = \frac{número\ de\ mudanças\ no\ local\ correto}{número\ de\ mudanças\ de\ deveriam\ ocorrer}$$

A precisão e recuperação de corretude privilegiam soluções em que a correção proposta está correta. Já a precisão e recuperação de reconhecimento privilegiam soluções que a correção é feita onde deveria ser feita, mesmo que esta corresponda a uma forma errada de correção. Essas medidas são maneiras de medir separadamente a capacidade do sistema de reconhecer erros e de corrigi-los.

Outro método existente para a avaliação de textos que permite que sua correção tenha tamanho diferente do texto original é o BLEU (***Bil**ingual **E**valuation **U**nderstudy*) (PAPINENI *et al.*, 2002), também utilizado para avaliar o desempenho de máquinas de tradução. Este, ao contrário dos métodos anteriores, permite que a normalização atinja uma entre **n** possibilidades de correções corretas, o que é comum no problema, uma vez que determinadas correções podem levar a diferentes estados válidos. O BLEU é regido pela seguinte fórmula:

$$BLEU = BP \times e^{(\sum_{n=1}^{N} w_n \times \log p_n)}$$

Onde BP corresponde ao fator de ajuste que beneficia os candidatos a tradução que estejam no tamanho correto, p_n correspondem a precisão de normalização utilizando n-gramas e w_n os fatores que balanceiam os termos. Cada p_n é responsável por avaliar o texto candidato em relação as suas possíveis correções utilizando de uma abordagem de n-grama, sendo assim p_1 utiliza 1-grama, enquanto p_2 utiliza bigramas e assim por diante. As abordagens de p_n consistem em verificar a porcentagem de aparecimento de cada n-grama do candidato nas diversas possibilidades de correção.

Enquanto os casos anteriores medem a qualidade de correção de um texto, uma outra métrica conhecida como N-Best mede a qualidade da correção das palavras individualmente. Para isso verifica com qual porcentagem as correções corretas se apresentam entre as **n** primeiras sugestões de normalização do vocábulo. Assim sendo, se o valor de 1-Best corresponder a 80% quer dizer que em 80% dos casos a real correção é a primeira opção reconhecida pelo algoritmo, enquanto se o valor de 2-Best corresponder a 95% quer dizer que em 95% dos casos a real correção está entre as duas primeiras opções sugeridas pelo algoritmo.

3.7 Considerações finais

Este capítulo deu uma visão geral sobre as soluções existentes que são aplicadas no contexto de normalização de textos. As técnicas descritas utilizam soluções fixas, em que o desenvolvedor é responsável por inserir todo o conhecimento que o algoritmo usa, assim como soluções em que utiliza o aprendizado de máquina para aquisição do conhecimento. Também foi visto que para a correção em editores de texto é necessário aplicar uma técnica diferente que apresente grande quantidade de conhecimento já cadastrado em sua base e um conjunto de passos a seguir para realizar as correções.

No que tange aos idiomas, viu-se também que ainda existe uma grande quantidade de problemas em aberto que necessitam de novas soluções para aperfeiçoar seus algoritmos de normalização, resultado da baixa quantidade de pesquisas realizadas fora da literatura em inglês.

Por fim, fez-se a análise dos diversos formatos de métricas existentes para validação do problema de normalização, em que cada uma realiza um tipo de análise e deve ser corretamente escolhida para cada tipo de solução.

4 Programação genética

Em inúmeros problemas da computação, para se atingir a solução, é necessário o desenvolvimento de um programa o qual deve ser composto por uma sucessão de funções, chamadas em uma ordem pré-estabelecida, e que trocam informações. Contudo, em diversas ocasiões, a formulação deste não é trivial devido a sua complexidade, dentre as quais destacam-se (KOZA, 1990): problemas de aprendizado de máquina, reconhecimento de padrão de digitais, formulação de estratégias de jogos e determinação de soluções de problemas matemáticos.

Para ajudar na formulação de programas que resolvam problemas, como os supracitados, surgiu a programação genética (PG), advinda dos algoritmos genéticos (AG) formulados em 1960 pelo professor John Holland (POZO *et al.*, 2005). Os AGs e PGs simulam a evolução genética observada por Darwin, conhecida como seleção natural, na qual os indivíduos mais bem adaptados ao ambiente se reproduzem passando para as gerações seguintes os melhores genes.

Enquanto os AGs contam com uma estrutura simples, que manipulam uma sequência de caracteres de tamanho fixo e são utilizadas para encontrar a solução de problemas, as PGs criadas em 1992 por John Koza fazem uso de estruturas complexas, que realizam a criação automática de programas que encontram soluções (POZO *et al.*, 2005). Cada programa gerado pela PG deve consistir de: (1) uma solução não trivial, tal que não se crie um mapeamento direto de entradas e saídas através de uma tabela de conversão, mas que se produza as respostas corretas para cada uma das entradas, e (2) quando se dispõe de um conjunto de treinamento representativo, o programa gerado seja capaz de produzir o resultado correto para uma entrada diferente das presentes no conjunto de treinamento (POZO *et al.*, 2005).

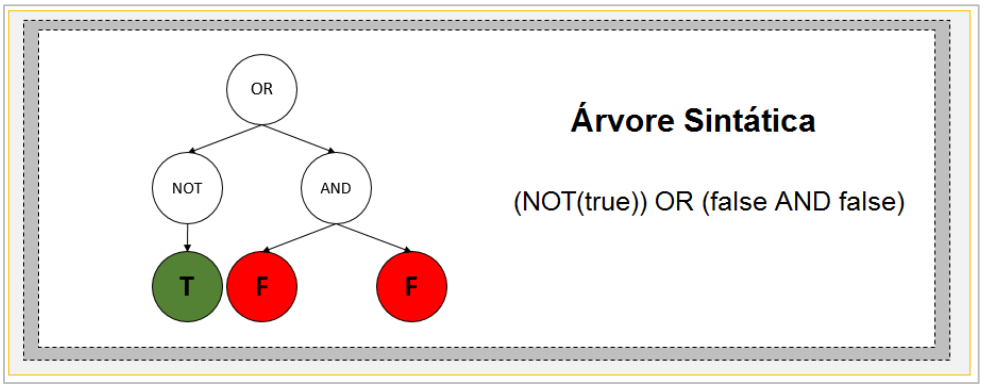

Figura 4 – Exemplo de árvore sintática de forma uma equação booleana

Os programas gerados pela PG, em geral, são representados através de uma árvore sintática, a qual consiste de uma estrutura de árvore que os nós são nomeados por símbolos de uma gramatica definida pelo problema, e que desta forma define o seu fluxo de execução (POLI *et al.*, 2008). A Figura 4 apresenta um exemplo de árvore sintática em que mostra a formação de uma equação booleana.

4.1 Funcionamento

O funcionamento da programação genética se assemelha a técnica dos algoritmos genéticos, divergindo apenas na complexidade dos nós e operações. A estrutura completa do funcionamento da PG se encontra na Figura 5, montada através da combinação do conhecimento gerado por EpochX (2012), Koza (1990), Poli *et al.* (2008) e Pozo *et al.* (2005) em um diagrama de atividades.

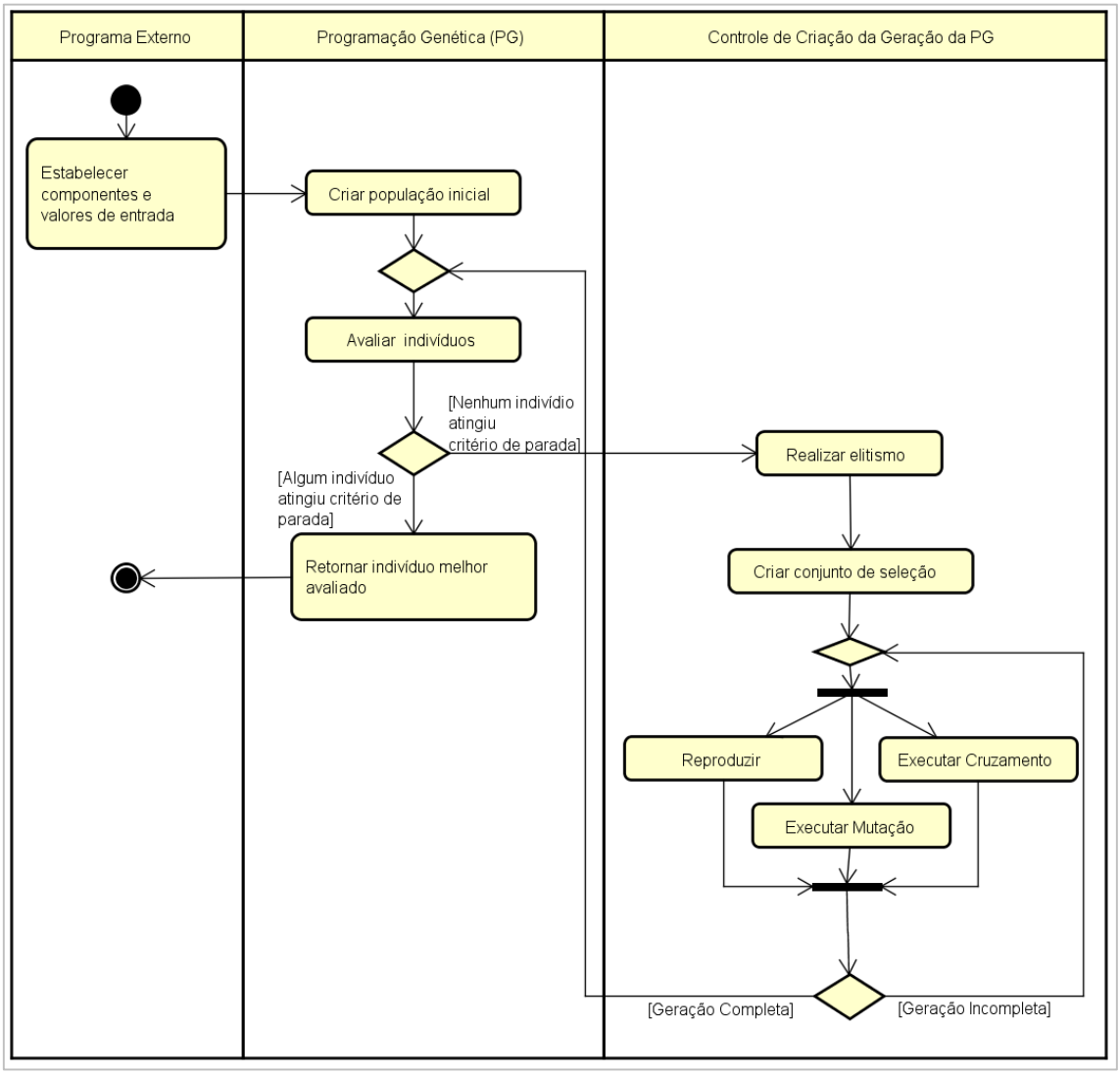

Figura 5 - Funcionamento geral da programação genética

Para iniciar o processamento da programação genética, deve-se definir todos os componentes (nós) disponíveis para a formação de uma árvore sintática. Estes são divididos em dois conjuntos onde o primeiro corresponde aos nós não terminais enquanto o segundo aos nós terminais.

Respeitando a profundidade máxima da árvore assim como o número máximo de nós, os quais são passados como parâmetro para a PG, é então inicializada a criação da população inicial dando origem aos indivíduos que farão parte da primeira geração. Dentre as técnicas disponíveis para geração dos indivíduos, quando utilizada a representação de árvores sintáticas, destacam-se os métodos (1) *full*, (2) *grow* e (3) *Ramped half-and-half* (POLI *et al.*, 2008). O método 1 gera indivíduos que são representados por árvores completas, ou seja, todos os seus galhos atingem a profundidade máxima de uma árvore. Já o método 2 permite a criação de árvores de qualquer tamanho, com a altura máxima respeitada, em que podem possuir qualquer tipo de configuração que seja possível de se gerar com os nós disponíveis. Por fim o método 3 realiza um aperfeiçoamento do *grow*, onde garante que dentro do conjunto dos indivíduos da população criada, os tamanhos de árvore variem em todo o intervalo permitido.

Ao se obter a primeira geração de indivíduos, estes passam por uma avaliação individual, onde cada um recebe uma nota de acordo com uma métrica pré-estabelecida. Analisando o indivíduo portador da melhor avaliação, confere-se o atingimento do critério de parada, onde em caso positivo este é entregue como saída da programação genética e caso contrário é dado início a criação da geração seguinte (POLI *et al.*, 2008).

A etapa de elitismo consiste do primeiro processamento que dá origem a indivíduos da nova geração, na qual transporta para a nova população os programas melhores avaliados na geração atual (EPOCHX, 2012). A quantidade de programas repassados para a nova população nesta etapa é definida a partir de um parâmetro passado como entrada na PG.

Uma vez que o tamanho da população deve ser mantido entre as gerações, é preciso adicionar os indivíduos faltantes. Estes são então gerados a partir de operações genéticas em cima de um subconjunto S de programas da geração atual. A formação de S é realizada através da definição da quantidade de elementos que este deve conter (o qual é passado como parâmetro para a PG) aliado a uma função de seleção que atua em cima da geração atual. É possível também definir que todos os indivíduos da geração

atual devem compor o subconjunto S, não realizando assim a função de seleção para a formação.

Dentre as funções de seleção existentes destacam-se o seletor randômico, que seleciona de forma aleatória um indivíduo do conjunto; o seletor de torneio (*Tournament Selector*), que pré seleciona **x** indivíduos, onde em uma espécie de torneio guarda apenas aquele que tem a melhor avaliação; e seletor probabilístico, que atribui uma probabilidade de seleção para cada indivíduo do grupo de acordo com a sua avaliação, que em seguida é realizada a busca levando-se em conta a probabilidade de cada um dos indivíduos ser escolhido.

Utilizando do conjunto de seleção criado e das operações genéticas, o restante da nova geração é criada, onde esta volta a passar pela função de avaliação individual de cada indivíduo dando início a um novo ciclo da PG.

4.2 Operações genéticas

O subconjunto de indivíduos da nova geração em que é criado a partir das operações genéticas tem sua formação concebida de maneira incremental, ou seja, indivíduo por indivíduo é adicionado até que o tamanho da população da geração seguinte seja atingido.

Para a adição de um novo indivíduo, primeiramente é escolhido qual das três operações genéticas será realizada através de um sorteio probabilístico, onde as chances de seleção são previamente configurados na PG. As operações correspondem a respectivamente o cruzamento (*crossover*), mutação (*mutation*) e reprodução (*reproduction*), onde cada uma tem sua importância. Embora não apresentado na arquitetura da Figura 5, é possível também que para a geração de um indivíduo seja aplicado uma combinação de operações genéticas, ao invés de apenas uma (POLI *et al.*, 2008).

Para a execução da operação genética, é necessário realizar a seleção do(s) candidato(s), feita com a utilização de uma função de seleção (já explicada na etapa de criação do conjunto de seleção S). No caso do cruzamento dois candidatos são selecionados, enquanto que para mutação e reprodução apenas é feita uma seleção.

4.2.1 Cruzamento

O principal operador genético é o cruzamento, onde dado dois indivíduos seus genes são combinados formando dois novos programas que herdam as características de seus antecessores. Dentre as técnicas de implementação destacam-se: cruzamento de um

ponto (um nó de cada um dos indivíduos é selecionado e então são trocados para a geração dos novos indivíduos), cruzamento de uma sub-árvore (um nó de cada um dos indivíduos é selecionado e então as sub-árvores com a raiz sendo os nós selecionados são trocados para a geração dos novos indivíduos).

A motivação por de trás dos cruzamentos consiste na chance da combinação dos genes gerar um novo indivíduo que partilhe dos melhores genes de seus antecessores, formando assim um novo ser melhor qualificado (POZO *et al.*, 2005).

4.2.2 Mutação

A mutação troca alguma característica presente no indivíduo, inserindo uma nova (ou pouco frequente) propriedade na população. Sua execução é vital para a preservação da diversidade genética da população (POZO *et al.*, 2005).

Em uma mutação, um indivíduo é selecionado e, através de alguma das funções de implementação, onde destacam-se a mutação de um ponto e mutação de uma sub-árvore que respectivamente alteram um nó ou uma sub-árvore por completo, cria-se um novo indivíduo herdando as características de seu antecessor com o acréscimo de alguma mudança genética.

4.2.3 Reprodução

A reprodução por sua vez se assemelha a fase de elitismo, onde os indivíduos não sofrem qualquer tipo de alteração e são passados para a próxima geração. Contudo, diferentemente do elitismo, na reprodução a escolha dos indivíduos que seguem para a próxima fase são decorrentes dos selecionados pela função de seleção, a qual pode tanto selecionar um indivíduo bem adaptado, como um indivíduo mal avaliado.

4.3 Função de avaliação

A função de avaliação (*fitness function*) serve para qualificar cada um dos indivíduos em cada uma das gerações. Uma vez que a PG tem as gerações criadas a partir de operações genéticas que selecionam indivíduos de acordo com a sua nota, uma boa função de avaliação é vital para o sucesso da aplicação da técnica.

Em inúmeras situações, um problema pode ter mais de um objetivo, e portanto, a função de avaliação deve ser capaz de atender a esse requisito, obtendo assim a resposta mais qualificada sob todos os aspectos. Assim, algumas possibilidades de solução são sugeridas (POLI *et al.*, 2008): combinar os objetivos em uma única função escalar,

utilizar a noção da dominância de Pareto e utilização de funções de avaliação dinâmica[11].

4.4 Critério de parada

O critério de parada é importante de ser definido uma vez que este é o responsável por finalizar a execução da PG. Em geral são utilizados dois critérios, onde o primeiro consiste do estabelecimento de um número máximo de gerações, enquanto o segundo no atingimento da avaliação referente a solução perfeita.

Algumas soluções realizam a análise da qualidade da melhor solução encontrada entre as gerações, utilizando como critério de parada a interrupção da melhoria de qualidade do melhor indivíduo entre duas ou mais gerações (POZO *et al.*, 2005).

4.5 Exemplo de funcionamento da programação genética

Um exemplo simples do uso da programação genética consiste em encontrar a melhor função matemática que represente um conjunto de respostas dado um valor de entrada **x**. Assim, pode-se considerar que os nós não terminais das árvores sintáticas correspondem as quatro operações matemáticas fundamentais: adição (+), subtração (-), multiplicação (*) e divisão (/). Enquanto que os nós terminais correspondem aos números e a variável de entrada **x**. Cada árvore montada então dá origem a uma fórmula matemática. Suponha que a população inicial de quatro elementos seja como mostrada na Figura 6, em que cada árvore representa uma fórmula (indicada no extremo inferior de cada quadro).

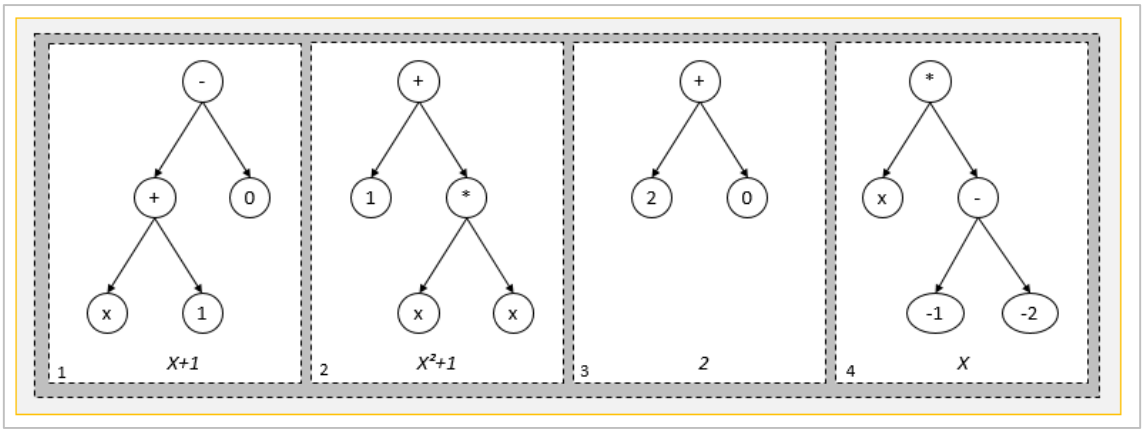

Figura 6 - População inicial do exemplo de programação genética (POLI et al.*, 2008)*

Se o objetivo for encontrar a função x^2+x+1, então cada um dos elementos da geração inicial (representado pela linha tracejada em vermelho) tem a sua avaliação gerada pela área formada entre as duas curvas, como mostrado na Figura 7.

[11] A função de avaliação dinâmica permite o uso de diferentes funções de avaliação ao longo das gerações.

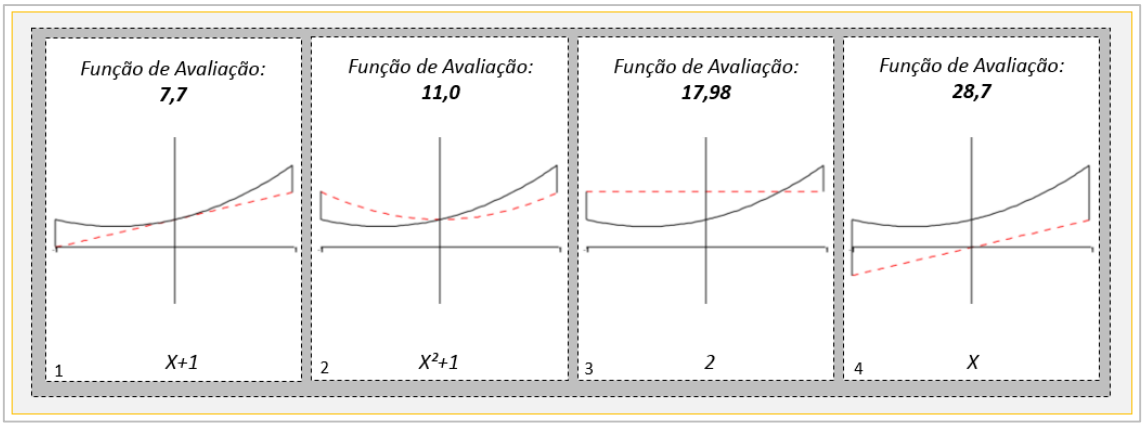

Figura 7 - Avaliação de cada programa da programação genética (POLI et al.*, 2008)*

Supondo então que a programação genética escolhe os dois primeiros indivíduos para realizar o cruzamento, o terceiro para a mutação e o último para reprodução, a geração seguinte é criada como mostrada na Figura 8. Dos quatro elementos da nova população, um representa exatamente a fórmula em que se deseja encontrar, o qual corresponde ao segundo elemento. Sendo assim, como a função de avaliação encontrará valor 0 para este indivíduo, este será entregue como solução da PG e esta é então encerrada.

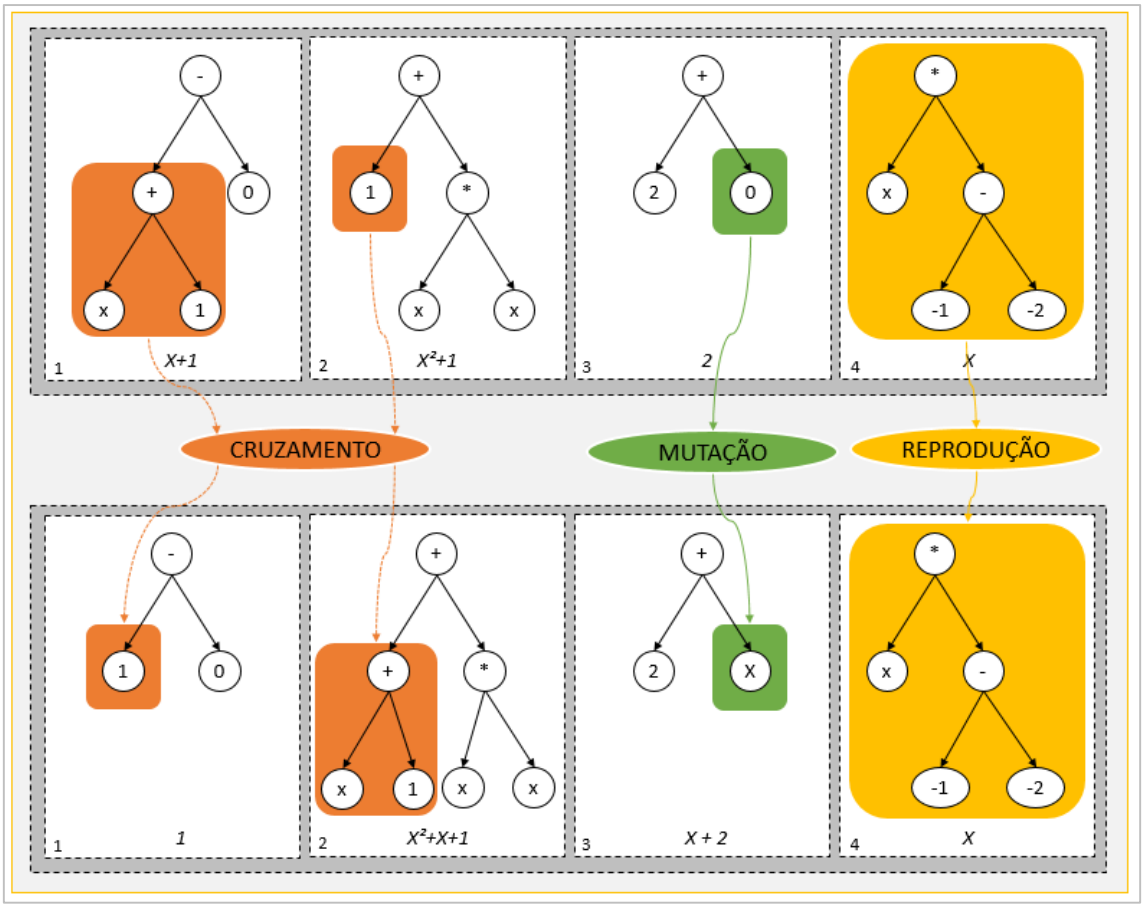

Figura 8 - Criação da geração seguinte

4.6 Considerações finais

Como visto neste capítulo, a programação genética surge como uma maneira eficiente de gerar programas que solucione problemas não triviais. Para isso, se inspira na teoria de evolução natural de Darwin, a qual inicialmente também motivou a criação dos algoritmos genéticos.

No que tange a normalização de textos, o uso da programação genética se mostra como uma alternativa de como encontrar um fluxo de execução, através de uma árvore sintática, que melhor adeque um conjunto de procedimentos a aplicar tal que possibilite assim realizar todas as devidas correções de um texto.

5 Proposta de solução

Neste capítulo é exposta a proposta do trabalho, a qual inicia pela apresentação das motivações envolvidas para em seguida realizar a formalização e explicação da arquitetura.

5.1 Motivação

A normalização de textos continua sendo um problema não solucionado na literatura, especialmente com o surgimento do internetês que permite total liberdade de escrita, agregando uma enorme quantidade de erros de grafia aos textos, como mostrado no capítulo 2. O capítulo 3 mostra diversas técnicas para o problema de normalização, as quais variam entre soluções de textos em linguagem culta ou coloquial, e com a presença ou ausência da aplicação do aprendizado de máquina. A proposta deste trabalho visa unir alguns dos benefícios apresentados nas técnicas anteriores, criando uma solução eficiente e diferente das demais.

A utilização de um conjunto de regras para a realização da tarefa de correção tem se mostrado eficiente nos editores de textos, que as têm em grande quantidade, e desempenham papel primordial para executar a correção de forma dinâmica, ou seja, corrigir à medida que o usuário realiza a criação do texto. Nos editores de textos, todos os preceitos necessários para colocar um texto na linguagem culta são pré-cadastrados em seu sistema, levando consigo todo o conhecimento a ser utilizado pelo algoritmo normalizador, podendo a medida do tempo conseguir conhecimentos adicionais tais que permitam melhorar a qualidade do corretor[12]. Contudo, as tentativas de correção de vocábulos realizados pelos editores se baseiam em erros tipográficos e cognitivos, não contendo regras para erros característicos do internetês.

Devido a vasta amplitude de tipos de erros, não é simples a criação de regras que deem suporte para corrigir os erros gerados pelo internetês. Com o objetivo de garantir o alcance necessário na correção, pode ser pertinente o uso do aprendizado de máquina, o qual consegue realizar uma análise em larga escala tal que a capacidade humana não é capaz, criando assim programas melhores adaptados ao problema.

A Figura 9 aponta uma possibilidade de como pode ser acoplada a aprendizagem de máquina em soluções fixas baseadas em regras. A área 1 (Figura 9) apresenta o cenário atual das soluções existentes no campo da normalização de texto, onde um

[12] Os corretores ortográficos dos celulares, por exemplo, conseguem aprender o mapeamento entre o que o usuário digita e o que ele queria escrever, corrigindo assim de forma mais eficiente os principais erros de grafia feitos pelo usuário.

corretor (representado pela caixa preta) pode ser fruto de duas diferentes abordagens de solução (representadas nas caixas cinzas): (1) corretor com estrutura e parâmetros fixos, o qual consiste de soluções prontas desde seu concebimento, e que não fazem uso do aprendizado de máquina e (2) corretor com estrutura e parâmetros variáveis, que consiste de soluções as quais necessitam da realização de uma fase de treinamento para definir os melhores valores de parâmetros e possíveis estruturas não estabelecidos inicialmente.

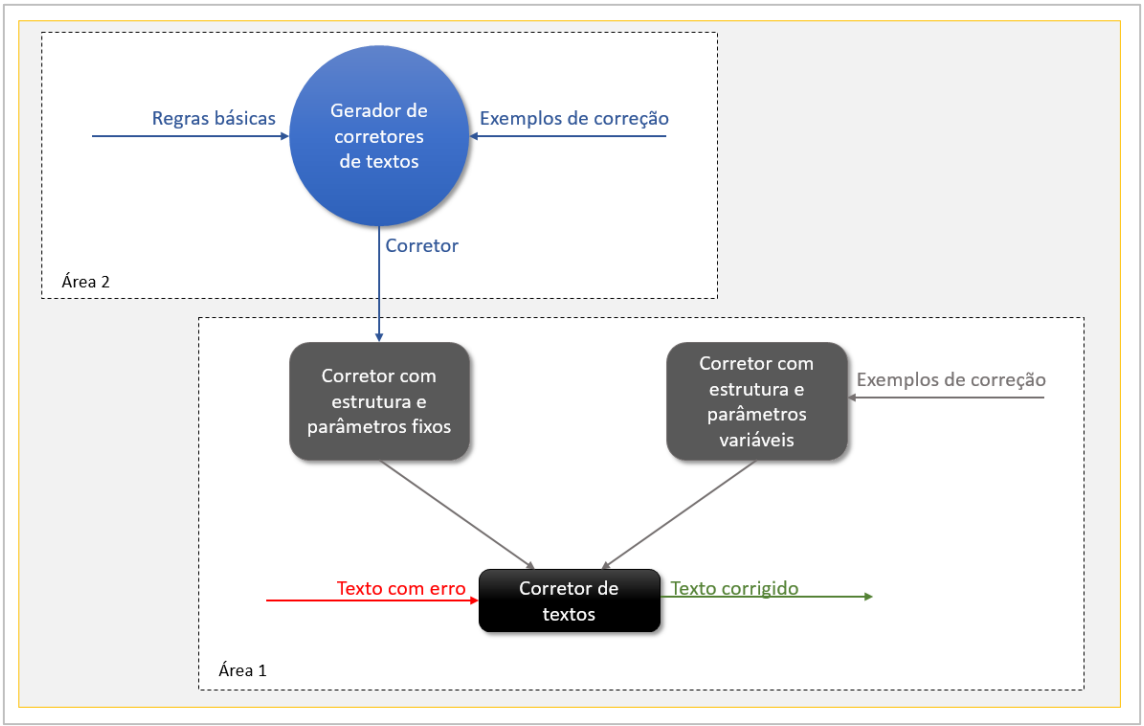

Figura 9 - Inclusão do aprendizado de máquina para programas baseado em regras

Para adicionar o aprendizado de máquina em soluções fixas baseadas em regras, é adicionado um módulo acima (área 2 da Figura 9) dos corretores de estrutura e parâmetros fixos. Este módulo adicional consiste de um gerador de corretores, que recebe como entrada um conjunto de regras básicas que são combinadas em uma determinada ordem, formando assim um programa de estrutura e parâmetros fixos. Cada programa gerado é qualificado de acordo com um conjunto de exemplos de correções, tal que a partir de suas combinações, consegue evoluir para programas melhores qualificados.

O procedimento a ser utilizado para realizar a evolução dos programas criados pelo gerador é a programação genética, na qual partindo de um conjunto inicial de programas, os evolui até atingir programas melhores adaptados, como já explicado no capítulo 4.

5.2 Representação e execução de um programa

Para a utilização da programação genética na construção de soluções baseadas em regras, é necessário criar uma forma de árvores sintáticas representarem um programa, que neste caso seja voltado para a normalização de textos.

A técnica elaborada neste trabalho determina que todos os nós da árvore representem uma função, tal que desempenhe alguma tarefa em cima dos tokens que por eles passarem. Uma frase errada é então considerada um token que, partindo da raiz da árvore, percorre todos os nós até atingir as suas folhas, retornando em seguida até o seu ponto de partida (nó raiz). No fim do processamento, são obtidas então todas as sugestões de normalização encontradas para os vocábulos presentes no texto.

A Figura 10 apresenta um exemplo simples de um programa rodando em uma árvore sintática. A árvore contém dois nós e a frase errada consiste de apenas uma palavra (“Legaaau”), que em seu formato correto de escrita corresponde a “Legal”. O quadro 1 mostra o instante anterior ao início do processamento, onde os nós na cor cinza ainda não foram processados, e o token com erro de escrita se encontra na raiz da árvore. Ao iniciar a execução, mostrada no quadro 2, o token recebe o processamento do primeiro nó, que elimina as suas letras repetidas, transformando-o em “Legau”. Em seguida (quadro 3) é executado o segundo nó, que através da distância de edição, identifica a palavra mais próxima como “Legal”. Os quadros 4 e 5 representam respectivamente o retorno do token pela árvore sintática, onde neste caso os nós não apresentam nenhuma execução na fase de subida, e o token que chega a raiz no fim do processamento corresponde ao formato correto de escrita que se queria atingir.

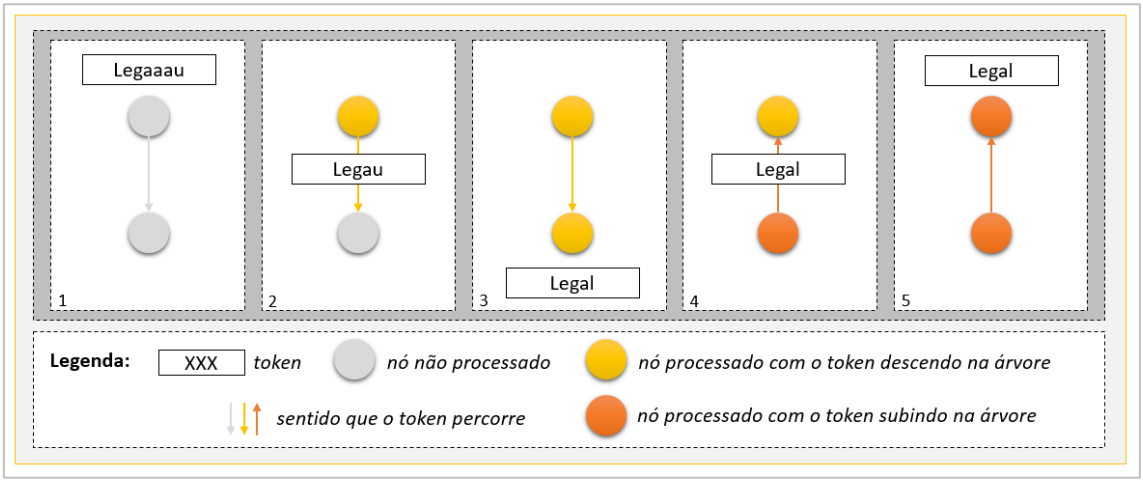

Figura 10 - Primeira simulação de um programa simples representado em uma árvore sintática

Um exemplo de um programa ligeiramente mais complexo rodado em uma árvore sintática é apresentado na Figura 11. Neste caso, a frase errada (“Vc ñ”) é

formada por duas palavras cujo formato correto de escrita corresponde a “Você não”. O instante anterior ao início da execução é mostrado na quadro 1, onde se nota que toda a frase (“Vc ñ”) é encarada pelo programa como apenas um token.

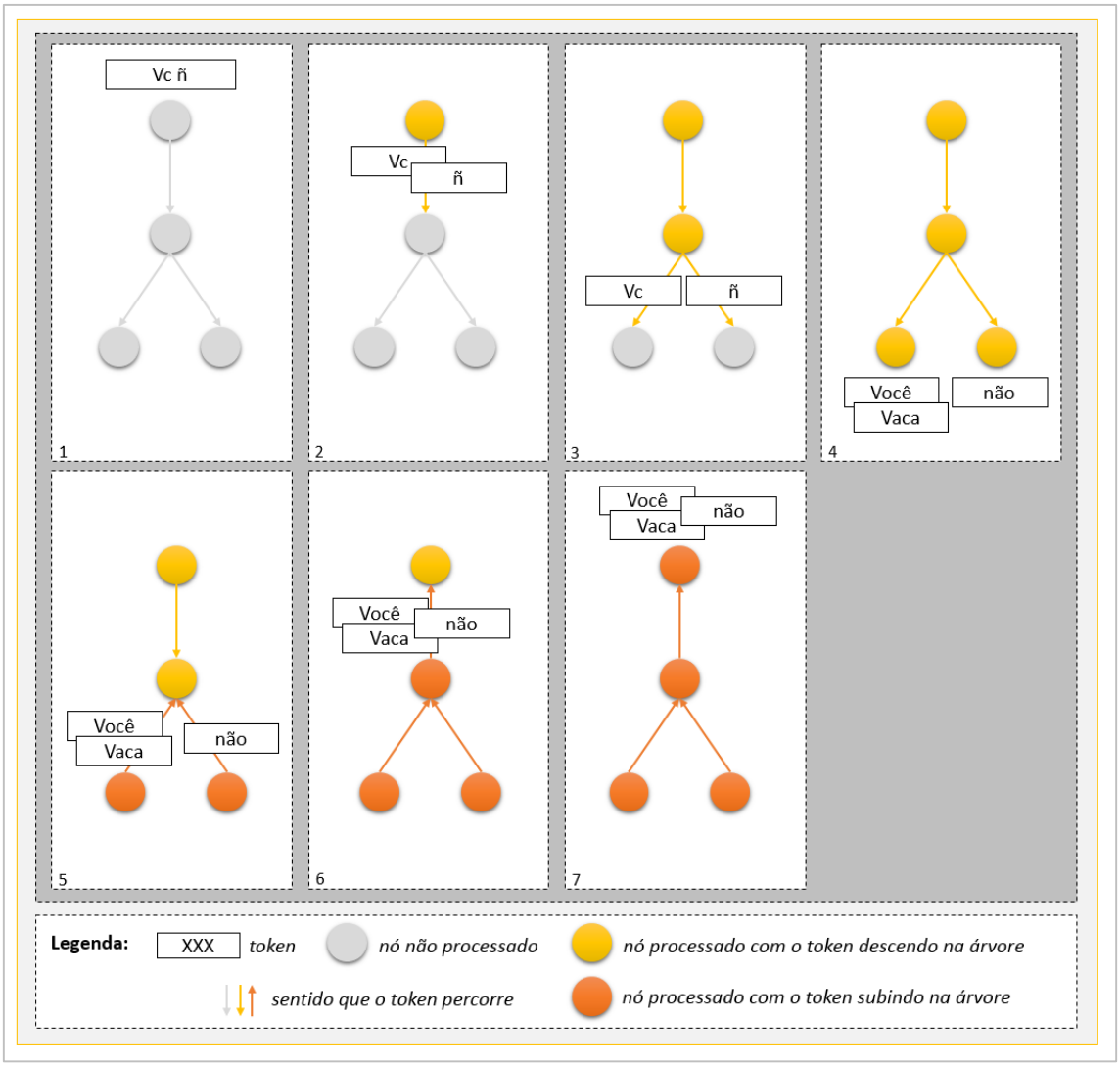

Figura 11 - Segunda simulação de um programa simples representado em uma árvore sintática

O quadro 2 mostra a execução do primeiro nó, que separa as palavras utilizando o espaço em branco para realizar as divisões, resultando em dois tokens (“Vc” e “ñ”). A execução do nó seguinte (quadro 3) bifurca o caminho que os tokens percorrem, colocando para o lado esquerdo aqueles com quantidade de caracteres maior que um e para a direita com quantidade igual a um.

O quadro 4 mostra o processamento de dois nós. O nó à esquerda busca em uma base de palavras aquelas em que, dentre as letras que a compõem, apresentem as mesmas consoantes (seguindo a mesma ordem de aparecimento) que o token que entrou no nó. Já o nó à direta busca por palavras na base que tenham em sua composição as mesmas consoantes e acentos do token adentrado. Assim, para “Vc” são identificadas as

possíveis normalizações “Você” e “Vaca”, enquanto para “ñ” é identificado a normalização “não”. Nos quadros seguintes (5, 6 e 7) os tokens sobem pela árvore sem sofrerem alterações até atingirem o nó raiz, obtendo como resposta a sugestão de normalização “não” para o token “ñ” e as sugestões “Você” e “Vaca” para “Vc”.

5.3 Nós necessários para normalização de textos

Utilizando da técnica explicada em 5.2 para execução de um programa através de uma árvore sintática, é necessária a realização de uma análise para identificação dos tipos de operações vitais que precisam ser modeladas pela arquitetura através de nós com intuito de conseguir realizar a normalização de textos.

Como já mostrado, a frase com erros de digitação ao chegar no programa corretor é identificada como apenas um token, não importando a quantidade de palavras nela presente. Entretanto, a identificação do vocábulo correto na maior parte dos casos deverá ser feita a partir da análise individual de uma palavra e por isso, como mostrado no exemplo da Figura 11, faz-se necessário a existência de um nó que realize a operação de separação em tokens menores, para que cada palavra individualmente possa ser tratada e corrigida caso necessário.

Os tratamentos que os tokens recebem visam ajudar a mapear as diversas variações dos tipos de erros de grafia característicos do internetês, e nas demais categorias de erros, ao seu formato culto da língua. Esse tratamento corresponde a execução de modificações e codificações dos vocábulos, indicando a necessidade de um nó que realize este tipo de operação, como o exemplo mostrado na Figura 10, onde o token tem removido de si todas as repetições de letras quando estas aparecem juntas.

Uma vez que as características de cada token pertencente a uma frase com erros podem ser distintas, é necessária a aplicação de tratamentos diferenciados para que as incorreções sejam devidamente reconhecidas e tratadas. Visando separar os tokens pelas suas características, um nó com operação de ramificação pode ser acoplado, como mostrado na Figura 11, tal que este determine caminhos distintos para tokens a partir da análise de alguma característica.

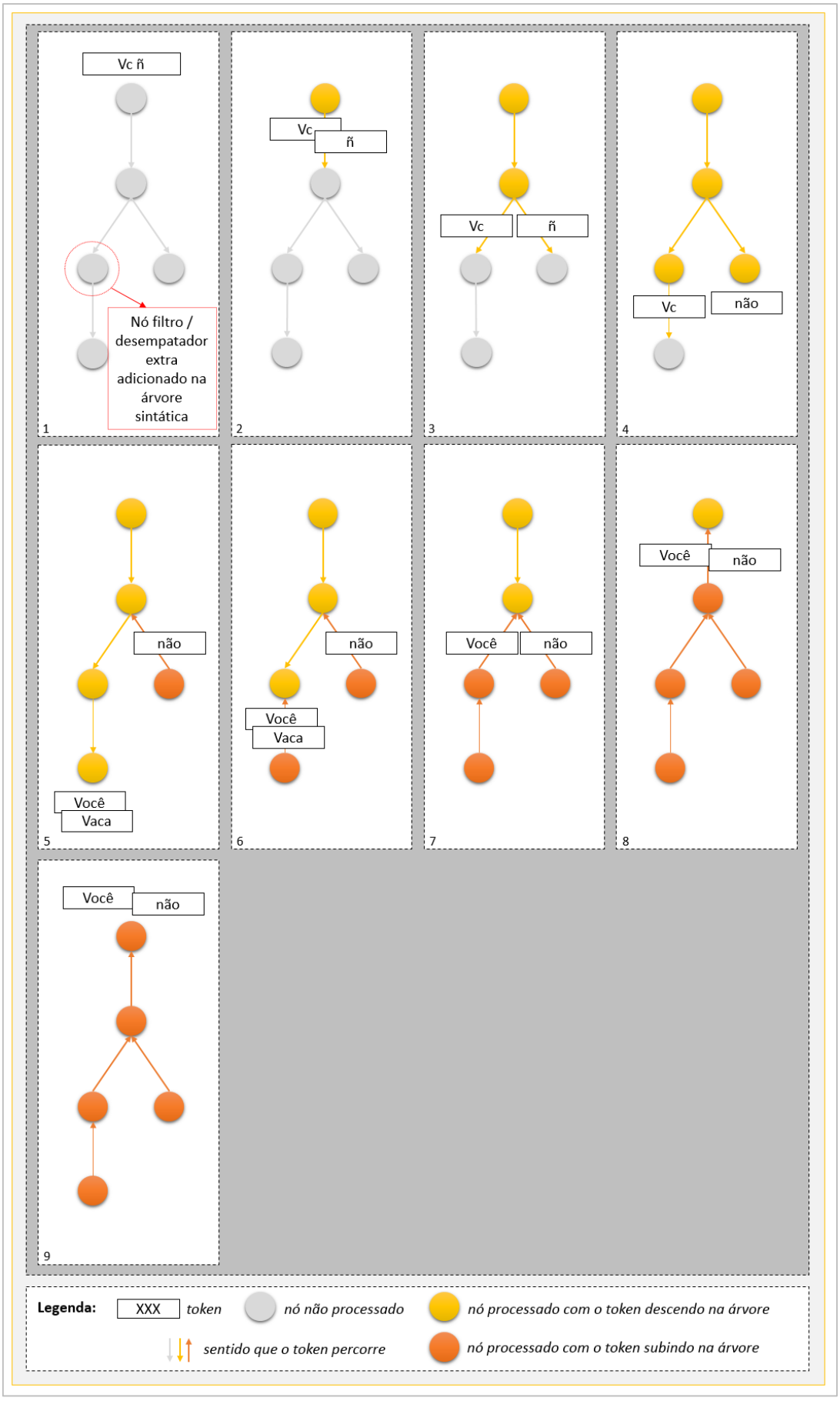

Figura 12 - Simulação de um programa em uma árvore sintática com nó filtro/desempatador

Quando os tokens percorrem a árvore sintática e atingem os nós folhas, estes são responsáveis por mapear o vocábulo do token (possivelmente errado) em zero ou mais possibilidades de correções. Os exemplos presentes na Figura 10 e Figura 11 mostram respectivamente a identificação do vocábulo “Legal” como possibilidade de correção de “Legau”, “não” para “ñ”, e “Vaca e “Você” para “vc”.

Sabendo do objetivo do trabalho que consiste em identificar um programa capaz de realizar a correção de forma automática dos textos, não é de interesse que a árvore sintática tenha em sua resposta final mais de uma sugestão de correção para um determinado vocábulo. Para que sejam eliminadas sugestões sobressalentes para um vocábulo é necessário que quando os tokens caminhem dos nós folhas em direção a raiz da árvore tenham-se nós que desempenhem papel de filtro ou desempatador. A Figura 12 apresenta uma árvore sintática derivada da Figura 11 onde há a presença de um nó adicional desempatador, no qual filtra os tokens que por ele passa, reencaminhando para o nó pai apenas aquele mais frequente na base. No caso de “Vc”, vence a sugestão de correção “Você”, a qual é mais frequente que a palavra “Vaca” no léxico. Desta maneira ao atingirem a raiz (Figura 12), existe apenas uma opção de normalização para cada vocábulo, corrigindo assim a frase originalmente errada.

5.4 Arquitetura

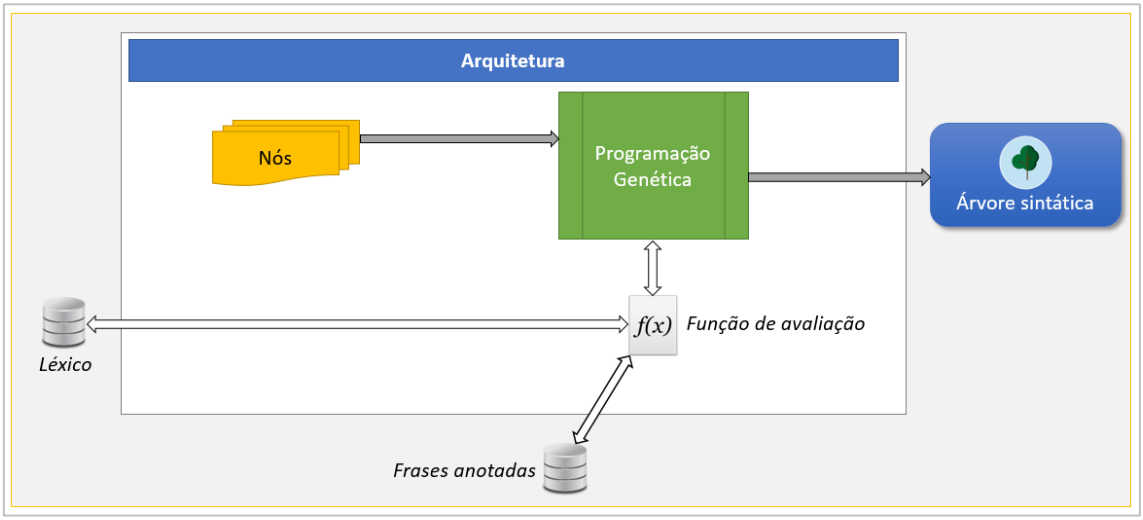

Figura 13 - Visão geral simplificada da arquitetura proposta

A arquitetura proposta visa permitir a elaboração de um programa (no formato de árvore sintática) qualificado a corrigir os mais variados tipos de erros de grafia que se tenha interesse, como por exemplo os erros tipográficos, cognitivos e propositais, no qual neste último está contido toda a linguagem do internetês. A Figura 13 mostra a

imagem da arquitetura proposta em seu formato simplificado, que para facilitar o entendimento do leitor é gradativamente complementada até atingir seu formato final.

O formato simplificado da arquitetura contém duas entradas externas, três componentes internos e uma saída. Das entradas externas têm-se:

1. Léxico: corresponde a um conjunto de palavras válidas no idioma em análise para serem utilizadas pelos programas gerados pela PG como referência da língua. Podem conter também informações adicionais como a frequência de aparecimento de cada vocábulo, assim como o aparecimento (e a respectiva frequência) de sequências de palavras (armazenadas no formato de n-gramas);
2. Frases anotadas: correspondem a um conjunto de frases, as quais podem conter vocábulos com erros de grafia, que carregam junto consigo todas as suas respectivas possibilidades corretas de escrita.

Os três componentes internos à arquitetura são:

1. Função de avaliação: corresponde a função utilizada para avaliar uma determinada árvore sintática gerada pela programação genética. Essa etapa é descrita com detalhes em 5.4.3;
2. Nós: correspondem aos elementos que são combinados e dão origem as árvores sintáticas, em outras palavras, correspondem aos vértices das árvores que representam um programa de normalização de textos. A formação e funcionamento desse conjunto de elementos é descrito em 5.4.1 e 5.4.2;
3. Programação genética: corresponde ao componente responsável por realizar todas as etapas da programação genética, como descrito no capítulo 4.

Por fim, a saída externa corresponde a:

1. Árvore sintática: corresponde ao indivíduo da programação genética que obteve melhor qualificação de acordo com a função de avaliação escolhida. Esta então condiz com o melhor programa encontrado para realizar a normalização de textos, quando aplicada neste contexto a arquitetura.

Tendo em vista a descrição de cada elemento da arquitetura, o funcionamento desta consiste nos seguintes passos: (1) os nós são entregues ao componente da programação genética, que realiza a criação da população inicial; (2) fazendo uso da

base de dados do léxico para obter os vocábulos válidos do idioma, cada indivíduo da PG é qualificado segundo uma função de avaliação para as frases anotadas; (3) a geração seguinte de indivíduos é criada e o ciclo da PG é repetido até que se atinja o critério de parada; (4) quando atingido o critério é retornada a árvore sintática melhor qualificada segundo a função de avaliação da PG.

5.4.1 Nós

Como mostrado em 5.3, propõe-se a existência de cinco operações vitais que precisam ser modeladas pela arquitetura através de nós (sumarizadas na Tabela 9), para que estes deem origem a árvore sintática de um programa normalizador de textos, no qual é executado com a técnica descrita em 5.2.

Tabela 9 – Operações vitais que necessitam ser modeladas pela arquitetura

Nº	Operação	Exemplo
1	Tokenização	Dado um token (por exemplo “Você não”), realizar a operação de tokenização para dividir este em palavras separadas (“Você” e “não”)
2	Modificação e codificação	Dado um token (por exemplo “Legaaau”), realizar uma operação que o modifique ou codifique (por exemplo, remover letras repetidas, tendo como resultado “Legau”)
3	Ramificação por característica	Dado um token, escolher por qual caminho este deve seguir (por exemplo em uma ramificação que envia para um lado tokens com tamanho igual a 1 – exemplo “ñ” – e para o outro os demais – exemplo “Vc”)
4	Mapeamento para as (zero ou mais) possibilidades de normalização.	Dado um token (por exemplo “Vc”), identificar as suas possibilidades de normalização (por exemplo “Você” e “Vaca”)
5	Filtragem ou desempate.	Dado um conjunto de possibilidades de normalização para um token (por exemplo as possibilidades “Você” e “Vaca” para correção de “Vc”), utilizar uma função que eleja apenas um candidato (por exemplo escolhendo o vocábulo “Você” como o postulante vencedor para a correção)

Com objetivo de modelar as operações vitais e permitir a inserção de regras pelos usuários, são criados os módulos e funções, os quais quando combinados dão origem a um nó. Os módulos consistem de estruturas fixas que são pré-estabelecidas pela arquitetura do trabalho, enquanto as funções são métodos desenvolvidos pelos

usuários que precisam unicamente seguir uma assinatura de função definida pela arquitetura, para que assim codifique a regra que se deseja inserir.

A Figura 14 apresenta a simulação de um programa que exibe um esboço da inserção dos módulos e funções na árvore sintática. No exemplo, estão disponíveis o módulo com operação de modificação e codificação e o módulo com operação de mapeamento para as possibilidades de correção. Deste modo, a árvore estruturada faz uso do primeiro módulo em duas ocasiões (primeiro e segundo nó, em que cada um utiliza de uma função diferente), e do segundo módulo para o nó folha (que faz uso de outra função).

Figura 14 - Simulação de um programa simples com módulos e funções

A execução do programa representado pela árvore sintática na Figura 14 para o token “9dades” em que seu formato correto corresponde a “novidades” inicia no quadro

2 com a execução do primeiro nó, no qual substitui os numerais pelo seu respectivo nome. O passo seguinte (execução do segundo nó – quadro 3) retira todo caractere que não corresponder a uma consoante, resultando no token “nvdds”. O último nó executado durante a descida do token corresponde ao nó folha (quadro 4), em que identifica as palavras formadas pelas mesmas consoantes presentes no token (com a mesma ordem de aparecimento). Com a palavra “novidades” identificada como única sugestão de correção, esta sobe até atingir o nó raiz (como mostrado nos quadros 5, 6 e 7).

Uma vez que os nós são criados a partir da junção de módulos e funções, a visão da arquitetura é então evoluída e mostrada na Figura 15. Como visto, surge uma nova entrada externa correspondente às funções (que codificam regras) criadas pelos usuário; e um componente interno onde estão definidos os módulos da arquitetura.

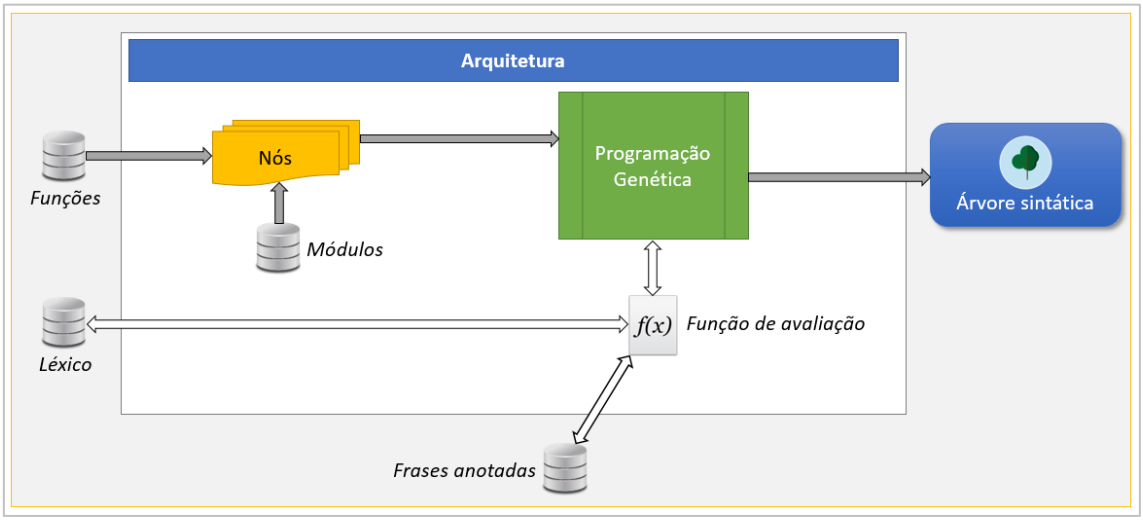

Figura 15 - Evolução da arquitetura com adição de componentes para geração do componente "nós"

5.4.1.1 Componentes de um módulo

Um nó consiste de um módulo com uma função específica acoplada em si. Esta função representa uma regra criada pelo usuário de forma codificada, enquanto que o módulo consiste na estrutura pré-definida pela arquitetura onde demarca a disposição por onde os tokens se movimentam e recebem os devidos processamentos.

A Figura 16 apresenta todos os componentes que podem estar presentes para a formação de um módulo, a começar pelo quadro 1 onde este exibe os dois tipos de módulos possíveis: não terminal (identificado pela cor laranja) e terminal (identificado pela cor verde). Deste modo, nós formados por módulos terminais correspondem as folhas das árvores sintáticas, enquanto os demais consistem de nós não folhas.

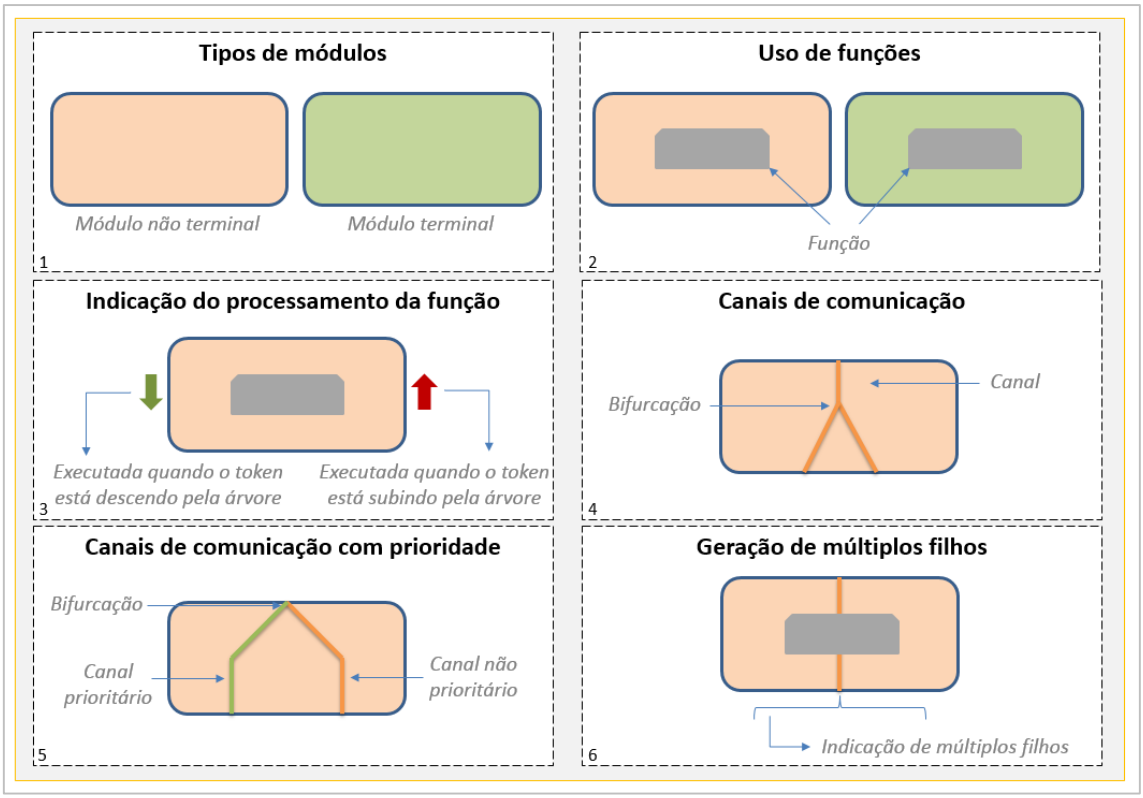

Figura 16 - Representação gráfica dos componentes presentes em um módulo

Para representar uma função que deve ser processada pelo módulo, é utilizado um componente cinza (mostrado no quadro 2 da Figura 16) para representar a indicação do local onde a função é chamada. Para os módulos não terminais em que transitam tokens em ambos os sentidos (da raiz em direção as folhas e vice-e-versa) é necessário definir em que instante a função deve ser chamada e assim, como mostrado no quadro 3, a presença de setas realiza a indicação: (1) a seta verde define que a função é chamada quando o token que passa pelo módulo está caminhando em direção as folhas e (2) a seta vermelha indica a chamada da função quando o token está seguindo em direção a raiz. Para os módulos terminais não há a indicação das setas pois como estes não tem nós filhos, não existe diferenciação em entre subida e descida de token.

Com a finalidade de guiar os tokens dentro dos módulos, são definidos os canais de comunicação (representados pela linha laranja no quadro 4). No caso de todos os módulos que serão definidos pela arquitetura, estes devem apresentar a conexão de um canal em sua margem superior (por onde recebem o token advindo de seu nó pai), e para o caso dos módulos não terminais apresentar também uma ou mais conexões de canais em sua margem inferior (por onde repassa os tokens para os nós filhos). Em situações de bifurcação (exemplo mostrado no quadro 4), o token original (quando caminha em direção as folhas) segue por um dos lados e envia uma cópia para o outro. Entretanto,

quando um token está caminhando em direção a raiz da árvore e atinge um ponto bifurcador, este deve esperar até que todos os demais tokens transitando na sub-árvore abaixo da bifurcação atinjam o mesmo ponto, para que estes sejam agrupados em uma lista única e assim deem prosseguimento a sua caminhada para o topo da árvore.

O quadro 5 apresenta o canal prioritário (representado pela cor verde) no qual sempre deve aparecer junto com o canal de comunicação comum (laranja) em uma bifurcação. Na situação em que os tokens estão transitando a caminho das folhas, a prioridade entre canais não apresenta diferenciação de comportamento, seguindo o original por um lado da bifurcação e uma cópia pelo outro. Contudo, quando um token caminha em direção a raiz e atinge um ponto bifurcador, a lista única gerada para dar prosseguimento a caminhada para o topo da árvore não conta com todos os elementos advindo de ambos os canais. Os tokens que chegam pelo lado prioritário são todos mantidos, enquanto aqueles oriundos do canal menos prioritário devem perpetuar apenas os que indicarem uma sugestão de correção para um token que não recebeu sugestões advindas no canal prioritário.

Com a estrutura criada pelos componentes mostrados nos quadros 1, 2, 3, 4 e 5 da Figura 16, os tokens transitam em direção às folhas e em seguida de volta para a raiz. Quando estão caminhando para baixo, os canais de comunicação transportam componentes unitários de tokens; em contrapartida ao se movimentarem para cima, estes transportam listas de tokens (as quais podem conter zero, um ou mais elementos). O último tipo de notação que pode estar presente na estrutura de um nó é mostrado no quadro 6, em que consiste de uma chave aberta abaixo do módulo. Esta representa uma notação especial para os módulos que gerarem como saída para seu nó filho mais de um token, necessitando executar toda a sub-árvore abaixo tantas vezes quanto a quantidade de tokens que este produzir. Em cada execução, um token é enviado para o nó filho, respeitando assim a regra de que os canais carregam apenas um token quando estes estão transitando em direção as folhas. Após a execução da sub-árvore para cada um dos tokens, as listas que cada processamento retornou são combinadas, gerando assim uma listagem única para dar prosseguimento na subida dos tokens em direção a raiz. Um exemplo completo do funcionamento descrito acima é mostrado na Figura 25 após a definição dos módulos em 5.4.1.2.

5.4.1.2 Os módulos e suas funções

Utilizando dos componentes definidos em 5.4.1.1, são criados os módulos da arquitetura tais que estes implementem as operações vitais explicitadas na Tabela 9. Para cada módulo é especificado um tipo de função, a qual se aloca no lugar da caixa cinza apresentada no quadro 2 da Figura 16, e é representada por uma cor diferente para especificar visualmente seu tipo.

A cada módulo definido é apresentado a sua representação visual com os componentes previamente explicados, a assinatura da função a qual o módulo permite acoplamento, e a representação em LISP do módulo, a qual é útil para poder representar uma árvore sintática como um encadeamento de funções matemáticas. A assinatura de função especificada apresenta-se na linguagem JAVA, e para representação de um token utiliza de uma classe genérica com nome Token.

5.4.1.2.1 Módulo gerador

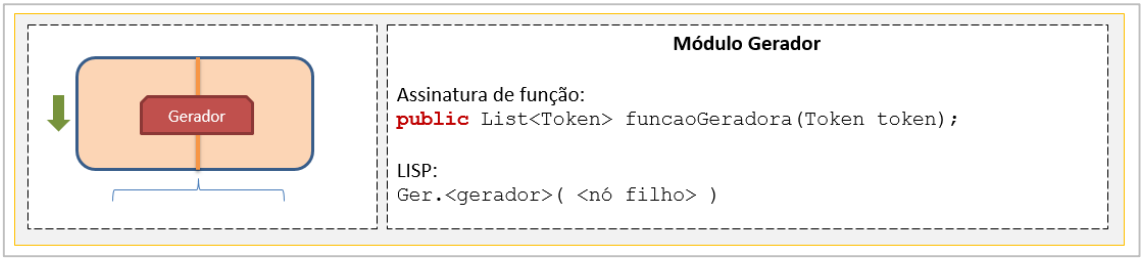

Figura 17 - Definição do módulo gerador

O módulo gerador (Figura 17) visa possibilitar a criação de nós tais que estes sejam capazes de realizar a operação de geração de tokens (operação vital 1 da Tabela 9). Quando os tokens estão transitando na árvore em direção às folhas e passam por um nó gerador, este é então encaminhado como parâmetro para a função geradora a qual retorna uma lista com zero ou mais tokens. Para cada um dos tokens gerados, um de cada vez, o módulo repassa-o para o seu nó filho. A medida que cada chamada retorna uma lista de sugestões de normalização, estas são agrupadas em uma listagem única a qual é repassada para o pai do nó gerador.

A função geradora além de permitir implementar a geração de tokens separando pelo espaço em branco, permite diversas outras operações de divisão, por exemplo:

- Geração de tokens a partir da quebra do token original no aparecimento de qualquer tipo de pontuação (ponto final, vírgula, exclamação, etc.);
- Geração de tokens consistidos de bigramas de palavras. Por exemplo, para um token que é composto por $<p_1\ p_2\ p_3\ p_4>$ onde os p_i's representam

as palavras do token, geram-se os seguintes bigramas de tokens <p_1 p_2>, <p_2 p_3> e <p_3 p_4>;

- Geração de tokens consistidos de n-gramas de letras. Por exemplo, para um token que é composto por uma única palavra <$l_1 l_2 l_3$> onde os l_i's representam as letras da palavra, geram-se os seguintes tokens <l_1>, <$l_1 l_2$>, <$l_1 l_2 l_3$>, <l_2>, <$l_2 l_3$>, <l_3>.

O código LISP que representa o módulo consiste do prefixo "Ger." que identifica seu tipo, seguido pelo nome da função geradora acoplada (colocada no lugar de "<gerador>"). O parâmetro passado ("<nó filho>") corresponde a codificação LISP da sub-árvore abaixo do nó.

5.4.1.2.2 Módulo codificador

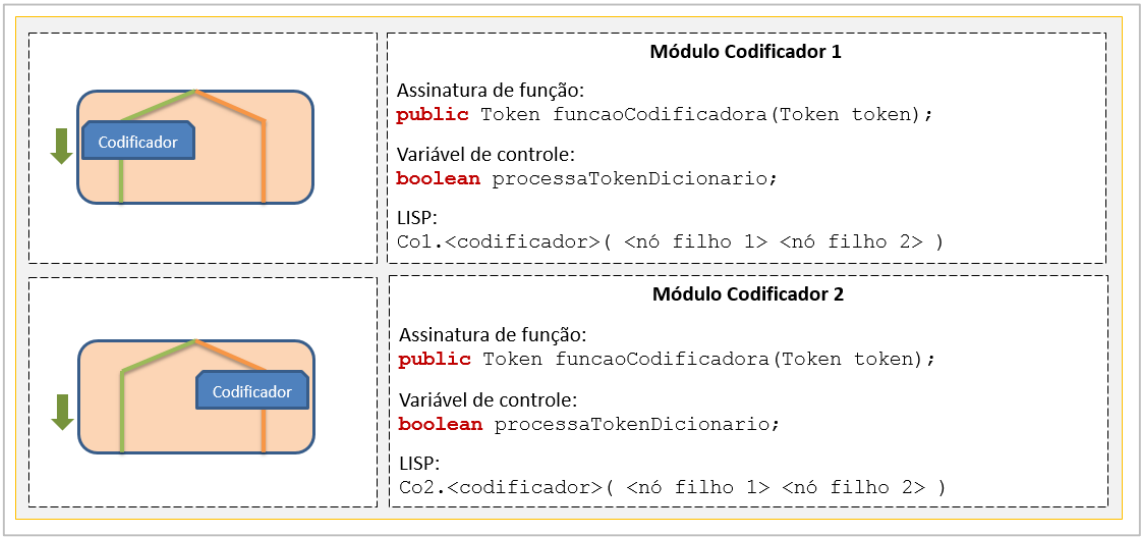

Figura 18 - Definição dos módulos codificadores

O módulo codificador (Figura 18) conta com duas variações de sua estrutura, as quais permitem a execução de modificações e codificações dos tokens (operação vital 2 da Tabela 9). Ambas as variantes consistem de um módulo o qual quando recebe um token percorrendo a árvore em direção aos nós folhas, este é imediatamente clonado devido a bifurcação do canal (com a divisão gerando respectivamente um lado prioritário e outro não). A diferenciação das duas estruturas está justamente relacionada a localização da função codificadora, podendo esta localizar-se no lado prioritário ou não.

A presença de dois canais, onde um preserva o token enquanto o outro realiza uma alteração, garante a integridade do elemento em uma de suas ramificações, permitindo assim que outros tipos de tratamentos sejam realizados em pedaços distintos

da árvore sintática. A diferenciação entre os canais propicia a priorização entre as sugestões advindas de tratamentos distintos.

Os módulos codificadores, diferentemente dos demais, além de apresentarem uma função codificadora, também contêm uma variável booleana de controle na qual deve ser preenchida para cada função criada. Esta variável indica se o processamento de codificação deve também afetar o léxico ou não (a qual é explicado mais adiante nas funções identificadoras em 5.4.1.2.5).

Dentre as funções codificadoras criadas, têm-se como exemplo:

- Colocação de todas as letras do token em caixa baixa;
- Remoção das vogais do token;
- Reordenação das letras do token, colocando-as em ordem alfabética;
- Remoção de espaços em branco.

O código LISP do módulo codificador apresenta inicialmente seu prefixo identificando o tipo de sua estrutura. Em seguida tem-se o nome da função codificadora, e por fim o recebimento de dois parâmetros (diferentemente do ocorrido no módulo gerador que apresenta apenas um). Os parâmetros da função indicam respectivamente as sub-árvores que têm suas raízes consistidas respectivamente pelos dois nós filhos do módulo codificador.

5.4.1.2.3 Módulo separador

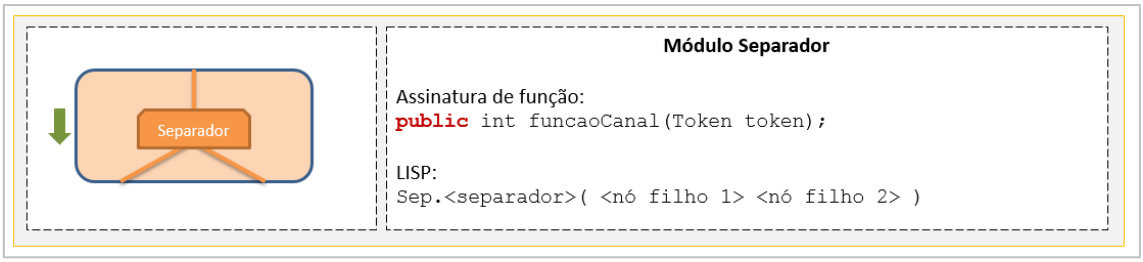

Figura 19 - Definição do módulo separador

O módulo separador (Figura 19) tem como propósito permitir a operação de ramificação, ou seja, determinar o caminho por onde o token que entrou no módulo deve prosseguir (operação vital 3 da Tabela 9). Deste modo, quando um token que está descendo pela árvore entra no nó separador, este é enviado para a função separadora na qual retorna a identificação de qual dos dois possíveis caminhos o token deve continuar a prosseguir, dando sua sequência em apenas um dos canais, e por isso recebe o nome de funcaoCanal. Quando os tokens retornam ao nó separador (no formato de lista),

caminhando em direção a raiz da árvore, estes passam direto pelo módulo sem qualquer tipo de processamento.

As funções separadoras determinam o caminho que o token deve prosseguir baseando-se unicamente nas características presentes nele, sem utilizar de qualquer outro tipo de informação. Assim, dentre as funções separadoras que podem ser criadas, têm-se como exemplo:

- Separação dos tokens pelo número de caracteres presentes;
- Separação dos tokens que apresentam apenas caracteres alfanuméricos;
- Separação dos tokens que apresentam repetição de caracteres;
- Separação dos tokens que apresentam numerais.

5.4.1.2.4 Módulo seletor

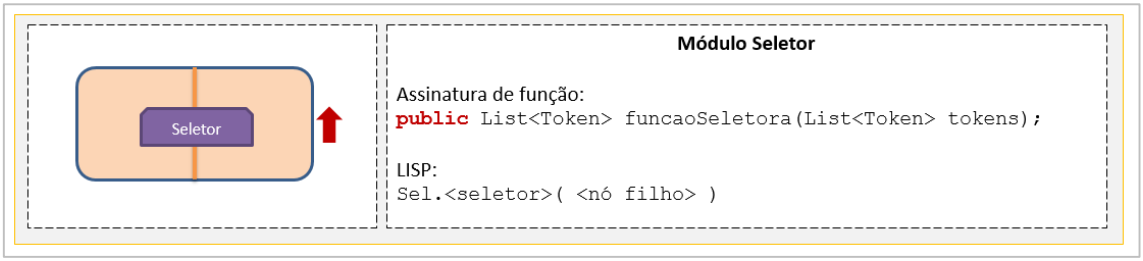

Figura 20 - Definição do módulo seletor

O módulo seletor (Figura 20), ao contrário dos demais, apresenta uma função (seletora) na qual é executada na fase de subida dos tokens, que recebe como parâmetro uma lista de tokens e tem como propósito realizar operações de filtragem e/ou desempate (operação vital 5 da Tabela 9). Quando os tokens unitários passam pelo nó seletor caminhando em direção as folhas, estes não sofrem qualquer tipo de alteração pois a função seletora não é disparada.

A função implementada pode realizar qualquer tipo de alteração nos tokens, assim como suas respectivas deleções. Por se tratar de um nó criado com objetivo de realizar a redução de possibilidades de sugestões, não é indicado a inserção de novos tokens em seu processamento, embora esta não seja uma ação impedida pela arquitetura e caso seja encontrado um propósito para sua utilização pode ser utilizado.

Dentre as funções seletoras que podem ser criadas, têm-se como exemplo:

- Descarte de sugestões de correção em que o token originalmente escrito contém letras distintas das presentes na sugestão;
- Escolha, dentre as opções de sugestão de correção para um mesmo token, aquela que corresponder ao vocábulo mais frequente na base;

- Escolha, dentre as opções de sugestão de correção para um mesmo token, aquela que obtiver menor distância de edição para o texto original;
- Alteração do gênero (masculino ou feminino) ou do grau (singular ou plural) da palavra em virtude da análise do contexto.

5.4.1.2.5 Módulo identificador

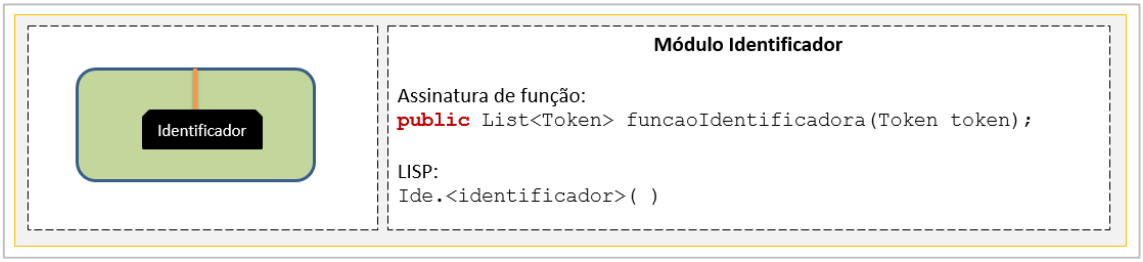

Figura 21 - Definição do módulo identificador

O módulo identificador (Figura 21) é o único da arquitetura do tipo terminal. Seu objetivo é realizar o mapeamento do token que atingiu o módulo com suas possibilidades de normalização (operação vital 4 da Tabela 9). Assim, cada token que entra no módulo é enviado para uma função identificadora, a qual, podendo fazer uso do léxico, realiza a identificação das sugestões de correção.

Para cada mapeamento criado entre um token e uma possibilidade de correção, uma instância de Token é inserida na lista que compõe o retorno de uma função identificadora. Sendo assim, caso para um mesmo token sejam mapeadas três sugestões através da função identificadora, três instâncias de Token são criadas e inseridas na lista de retorno, cada uma correspondendo a uma sugestão de correção.

Como explicado em 5.4.1.2.2, cada função codificadora apresenta uma variável booleana na qual indica se a função de codificação deve ser aplicada também no léxico no instante da identificação. Por este motivo, cada função identificadora que optar por fazer a busca no léxico das possibilidades de correção, pode executar (caso deseje) as funções codificadoras aplicadas ao token que entrou no módulo identificador (que tem sua variável booleana *processaTokenDicionario* acionada) nos vocábulos do léxico, respeitando a ordem de execução. Com o léxico em seu estado original ou a partir da cópia alterada pela execução das funções codificadoras, a função identificadora faz seu processamento resgatando possibilidades de correção conforme sua regra codificada. A presença de funções identificadoras de qualidade é vital para o sucesso da normalização, uma vez que são as únicas funções responsáveis por detectar as possibilidades de correção para os vocábulos.

Por se tratar de um módulo terminal, este não apresenta filhos, e assim pode ser visto que na codificação LISP não há a presença de parâmetros. Dentre as funções identificadoras, têm-se como exemplo:

- Detecção da palavra na base tal que esta seja igual ao token recebido como parâmetro;
- Detecção na base de todas as palavras tal que a letra inicial de cada sílaba esteja representada no token recebido como parâmetro, garantindo a mesma ordem de aparecimento;
- Identificação que o token consiste de uma representação de risada (característico das redes sociais);
- Detecção das palavras na base que tem a mesma codificação fonética que a apresentada pelo token recebido como parâmetro.

5.4.1.2.6 Módulo duplicador

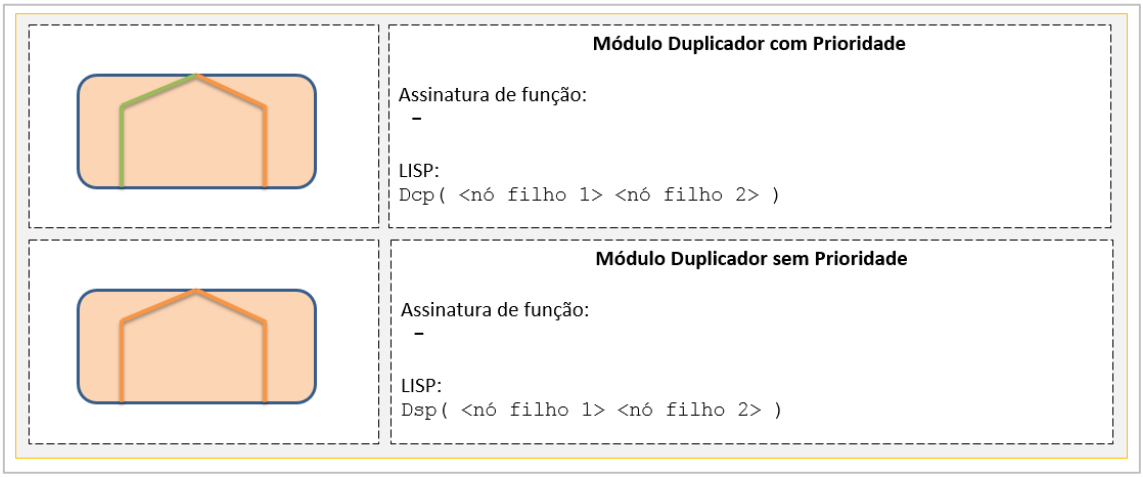

Figura 22 - Definição dos módulos duplicadores

Além dos cinco módulos anteriores criados a partir das motivações apresentadas pelas operações vitais (Tabela 9), é criado mais um tipo de módulo com duas variantes: os módulos duplicadores (Figura 22). Estes têm a estrutura muito parecida com os módulos codificadores, entretanto não apresentam funções. Deste modo, sua criação tem como motivação possibilitar a existência de uma variedade de tratamentos a partir de um certo ponto da árvore, criando para isso caminhos alternativos.

5.4.1.3 Criação dos nós

A partir dos módulos definidos pela arquitetura e dos diversos tipos de funções criadas pelos usuários tais que utilizem da assinatura pré-definida pela arquitetura, é

realizada então a criação dos nós os quais são usados pela programação genética para geração dos indivíduos.

A Figura 23 apresenta um exemplo de criação dos nós, em que no caso existem três funções diferentes para cada um dos tipos de assinatura (função geradora, codificadora, separadora, seletora e identificadora).

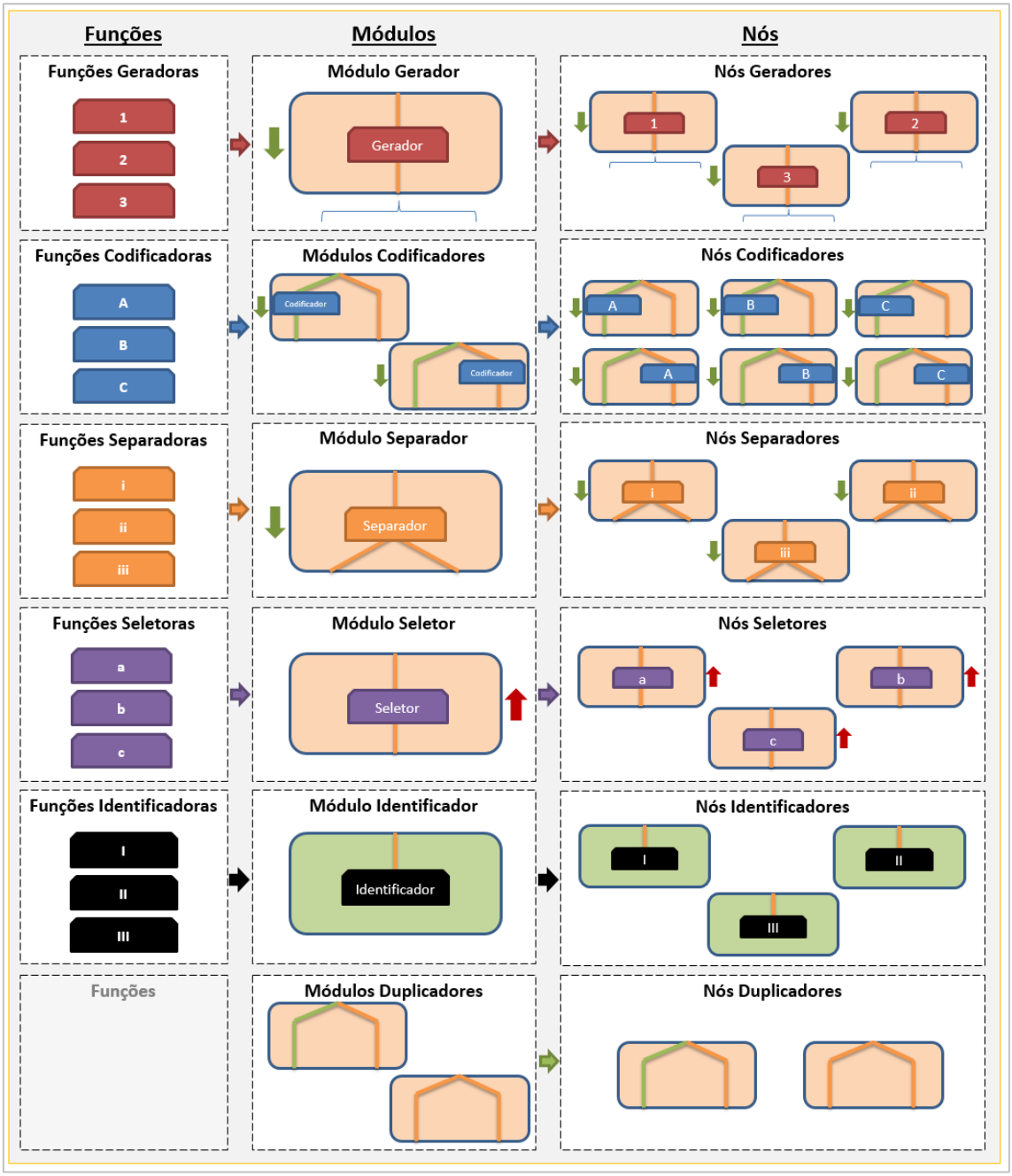

Figura 23 - Formação dos nós a partir dos módulos e suas funções

Nesta configuração exemplificada, a programação genética recebe vinte nós com os quais dará vida a uma população inicial de árvores sintáticas, evoluindo seus indivíduos para obter programas melhores adaptados ao problema em questão.

5.4.2 Execução de um programa normalizador

A partir das definições criadas para a arquitetura em 5.2 e 5.4.1 é possível determinar todo o funcionamento de um programa normalizador representado como uma árvore sintática.

5.4.2.1 Exemplo de funcionamento de um programa

Para exemplificar o funcionamento de um programa, criou-se uma árvore sintática com os módulos da arquitetura (Figura 24) tais que estes executassem a mesma correção realizada na Figura 10.

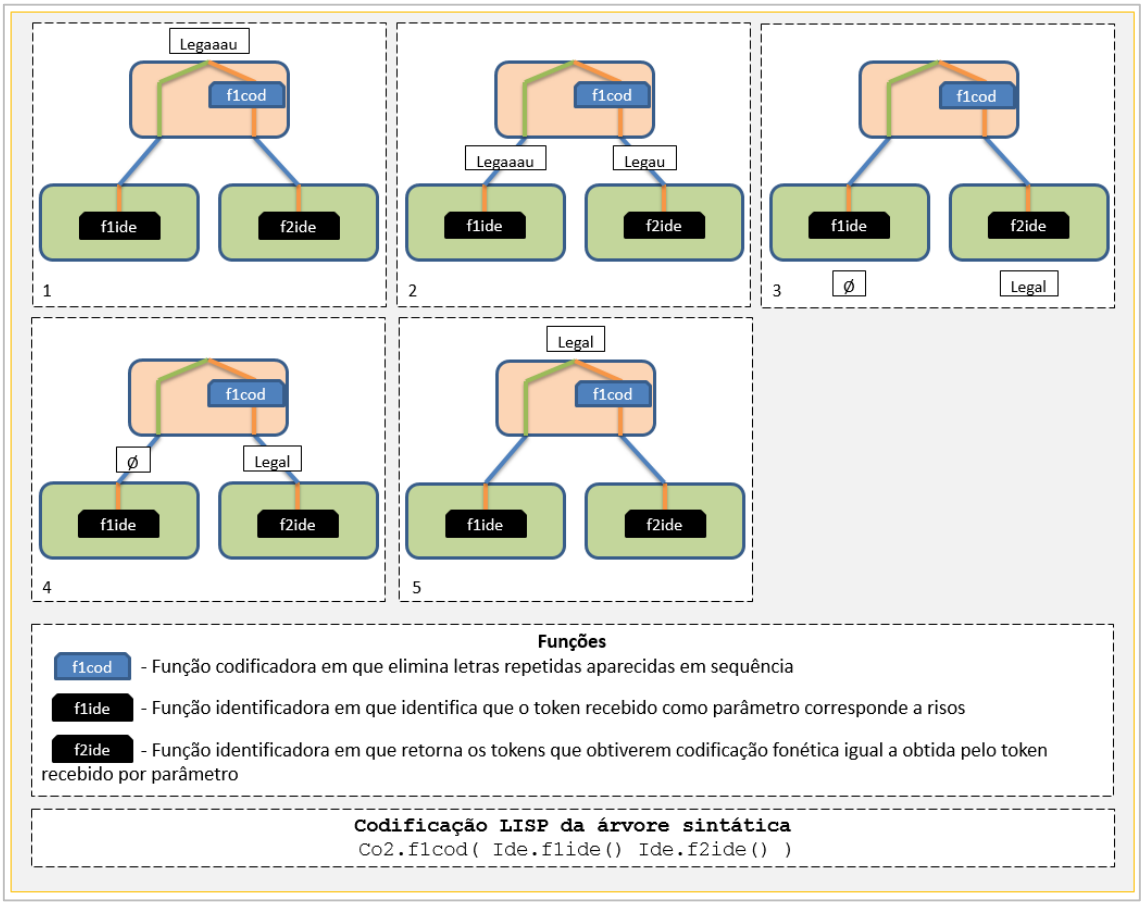

Figura 24 - Execução de programa simples com os módulos da arquitetura

O instante anterior ao início da execução é apresentado no quadro 1, o qual contém o token errado acima da raiz da árvore, e seus três nós com as respectivas funções acopladas. Ao iniciar a execução (quadro 2), o token é duplicado tendo uma cópia seguindo por cada um dos canais de comunicação dentro do módulo codificador. As saídas do nó correspondem respectivamente a uma instância não alterada do token original e uma codificada na qual é removida todas as letras repetidas aparecidas em sequência.

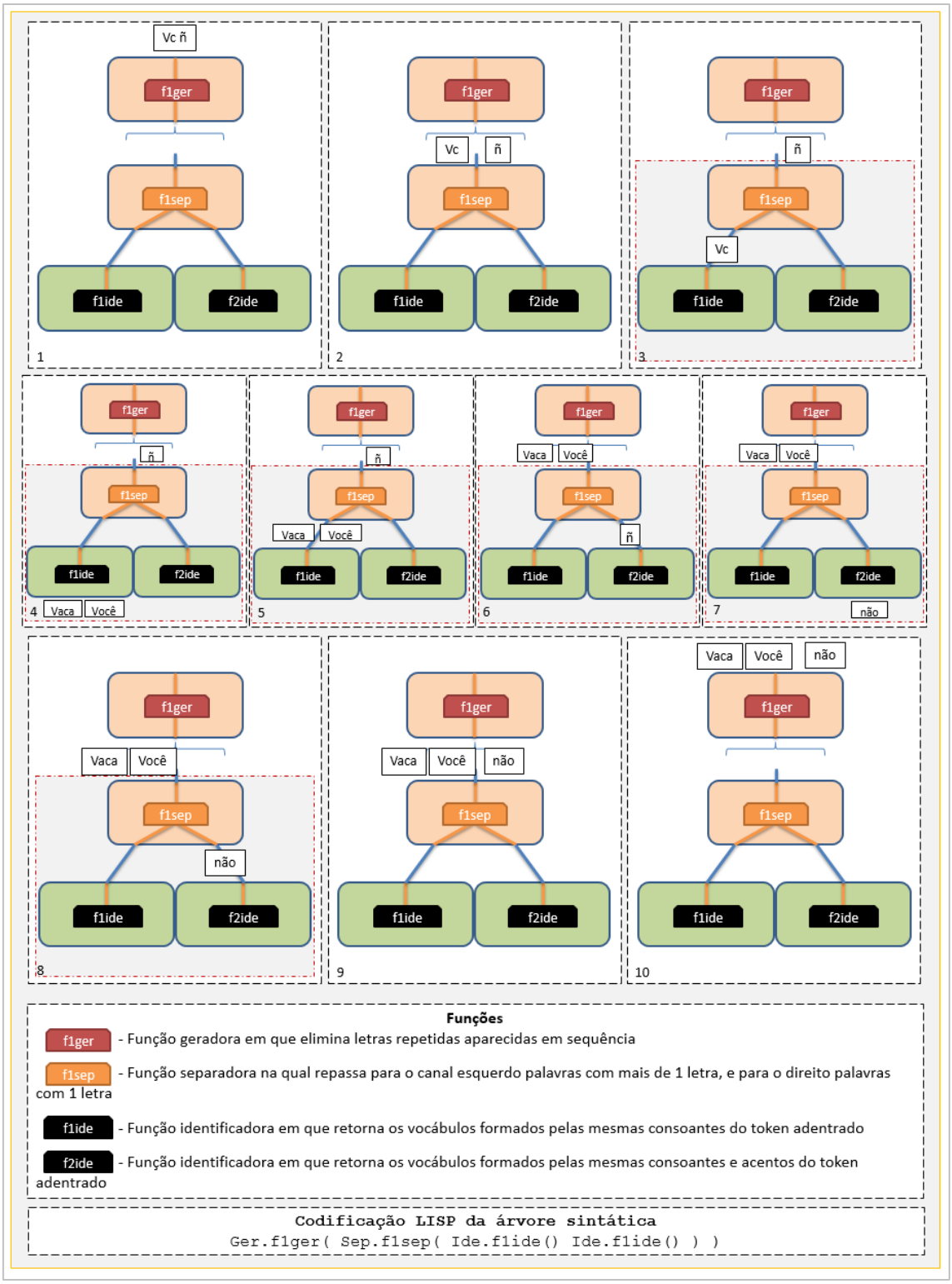

Figura 25 - Execução de programa simples (2) com os módulos da arquitetura

No quadro 3 (Figura 24) os tokens recebem os respectivos tratamentos dentro dos nós identificadores, em que para o token “Legaaau” não se encontra nenhuma possibilidade de normalização (identificado pelo caractere vazio “Ø”), e para “Legau” se identifica a instância “Legal”, na qual nos quadros seguintes (4 e 5) retorna até o nó raiz. É importante ressaltar o funcionamento do módulo codificador na subida dos

tokens, em que optou pela sugestão advinda do canal menos prioritário devido o retorno do canal prioritário não conter nenhum elemento.

A Figura 25 apresenta um exemplo de simulação mais complexo na qual faz o mesmo tratamento demonstrado na Figura 11. O quadro 1 apresenta a estrutura da árvore com a frase errada acima do nó raiz antes do início da execução da árvore sintática. Ao iniciar o processamento no quadro 2 com a execução do nó gerador, são gerados dois tokens a partir do original, no qual um de cada vez é repassado para a sub-árvore abaixo do nó, como mostrado nos quadros 3, 4 e 5 para o token "Vc" e nos quadros 6, 7 e 8 para o token "ñ". Ao encerrar os processamentos da sub-árvore, tem-se as listas de sugestões advindas dos dois tokens (mostradas no quadro 9), as quais são compiladas em uma listagem única e repassada para a raiz da árvore (quadro 10).

Como visto neste exemplo (Figura 25), quando há a presença de geradores na árvore sintática, existe uma diferença clara entre esta e a árvore de execução. A Figura 26 mostra um exemplo de árvore sintática mais complexa, contendo mais de um módulo gerador e sua respectiva árvore de execução (que por motivo de simplificação os módulos da arquitetura foram representado no formato de círculos).

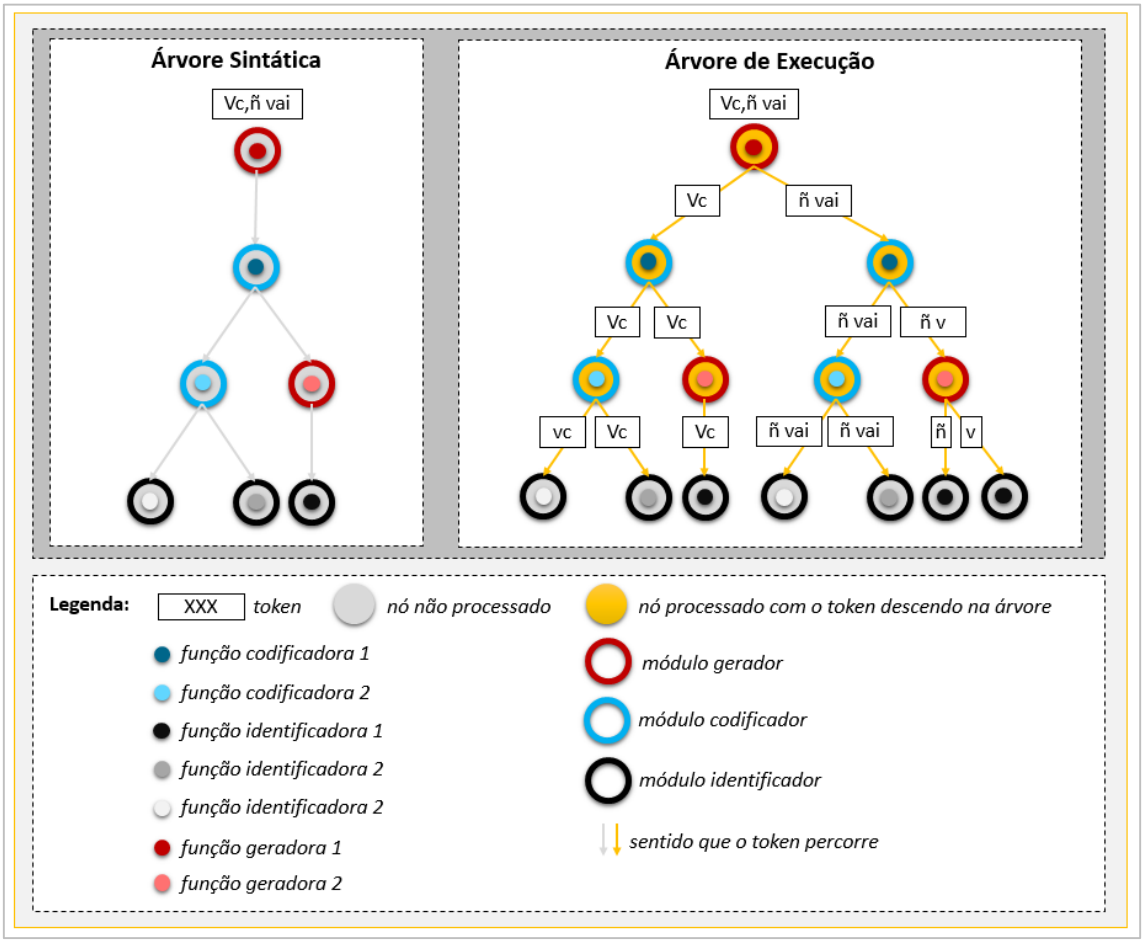

Figura 26 - Árvore sintática versus árvore de execução

A árvore de execução mostra todos os passos percorridos até a etapa anterior a execução dos módulos terminais, realizando assim apenas a execução da descida dos tokens. Nota-se que a presença de geradores faz o processamento aumentar tanto quanto seja a quantidade de filhos que estes criam a partir dos tokens, onde apenas um nó pode dar origem a um grande número de filhos, definidos pela quantidade de divisões realizadas pela função geradora. Deve-se então tomar cuidado ao realizar a implementação destas, certificando que de fato podem ser úteis.

5.4.2.2 Correção iterativa versus não iterativa

Como visto até o momento, as árvores sintáticas ao serem executadas retornam uma lista de sugestões para cada um dos vocábulos presentes na frase original. No caso de editores de textos, o estado atual atingido já é o necessário para prover o sistema de correção iterativa, na qual para cada token são oferecidos possibilidades de correções, em que o usuário opta por substituir ou não seu vocábulo digitado.

Entretanto, para o problema tratado neste trabalho, é necessário corrigir de forma automática todos os vocábulos que estão escritos de forma errada, sem a ajuda de um usuário. Deste modo, é preciso que a partir das sugestões disponibilizadas pela árvore sintática, a frase seja remontada em seu formato correto.

Para atingir esse objetivo são necessários a existência de dois componentes: (1) um processador de árvore sintática, o qual realiza a execução das árvores como já mostrado anteriormente, e (2) um analisador, responsável por remontar a frase. Com os novos componentes, a arquitetura proposta é evoluída como mostrado na Figura 27. A função de avaliação, a qual necessita qualificar cada um dos indivíduos gerados pela PG, envia a árvore sintática para um analisador junto com a frase em que quer realizar a normalização. O analisador por sua vez chama o processador da árvore sintática para a geração das sugestões para em seguida remontar a frase, retornando-a para a função de avaliação.

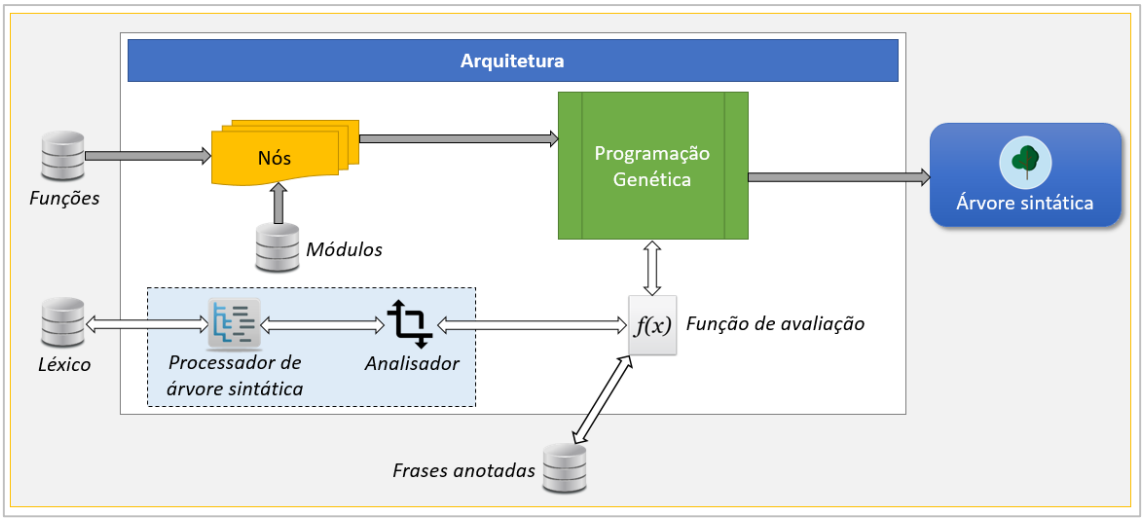

Figura 27 - Evolução da arquitetura com adição de componentes para geração da frase normalizada

Os dois novos componentes apresentados na Figura 27 aparecem dentro de uma caixa azul, a qual é adicionada pois representa o componente completo necessário para a execução de uma árvore sintática e portanto, é também utilizado fora da arquitetura para execução da melhor árvore sintática encontrada com intuito de realizar as correções, como mostrado na Figura 28. Para tanto, o componente recebe a árvore sintática em que define o programa corretor, tem acesso a um léxico com as palavras válidas do idioma, e assim dado uma frase com erros retorna a frase normalizada.

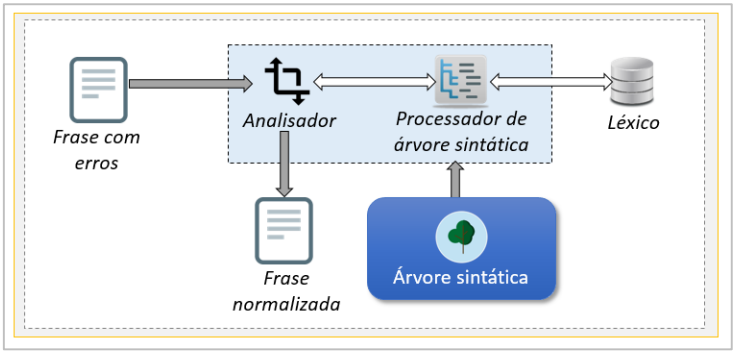

Figura 28 - Componente responsável por executar um programa definido por uma árvore sintática

Cada token sugerido como possibilidade de correção pela árvore sintática tem a identificação de sua posição inicial e final na frase original, para que se saiba a qual(is) vocábulo(s) esta correção pertence. O analisador utilizado na arquitetura funciona verificando as sugestões providas e a partir das que consistirem de uma sugestão única no intervalo de posições (compreendido entre a posição inicial e final), realiza a substituição do vocábulo sugerido na frase original. Sendo assim, todas as posições da frase original que obtiverem mais de uma possibilidade de sugestão, tem valor original mantido, ou seja, são descartadas as sugestões que apresentam conflitos.

5.4.3 Função de avaliação

Com intuito de avaliar a qualidade de cada indivíduo da população da PG e assim conseguir evoluí-los ao longo das gerações para obter programas melhor adaptados ao problema, faz-se necessário o uso de funções de avaliação (*fitness function*). Estas, têm papel fundamental na arquitetura, afinal são as principais avaliadoras de qualidade e determinante para a criação das gerações.

Uma função de avaliação é uma entrada externa que o usuário é capaz de definir. Esta pode ser única, ou pode variar ao longo das gerações através de algum critério. Para isso, a arquitetura é evoluída, atingindo seu estado final (Figura 29). O componente "função de avaliação" é substituído por um avaliador, o qual é chamado pela programação genética e, utilizando das frases anotadas e de uma função de avaliação fornecida pela entrada externa "Funções de avaliação", realiza a avaliação do indivíduo.

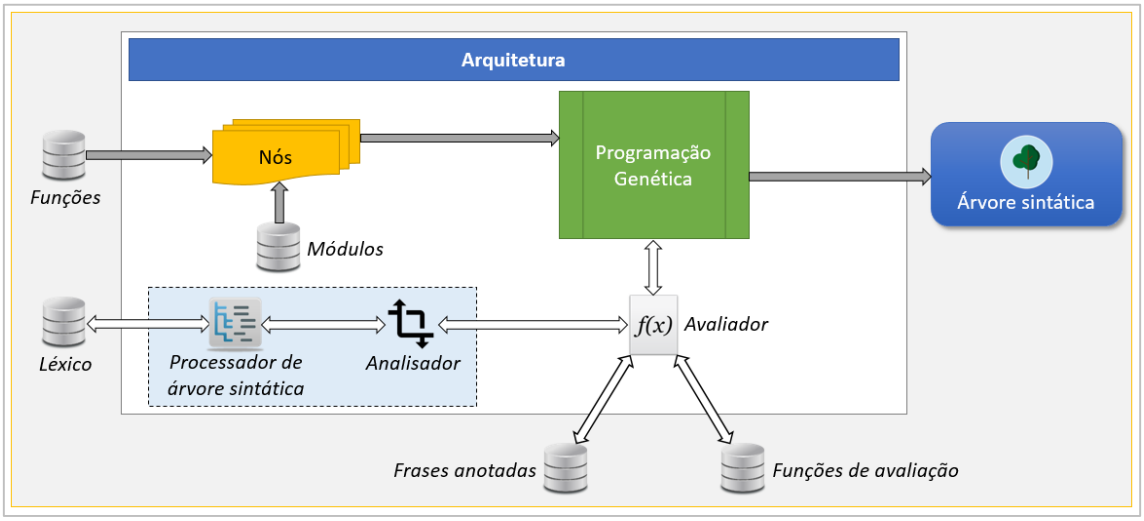

Figura 29 - Arquitetura completa

A cada geração da programação genética, cada indivíduo da população é enviado para o avaliador, o qual executa seu processamento para cada uma das frases anotadas passadas pela entrada externa e a avalia de acordo com a função de avaliação utilizada na geração.

Das métricas clássicas existentes explicadas em 3.6, a mais adequada para aplicação consiste no BLEU, que avalia a qualidade da normalização de um texto que pode atingir múltiplos estados corretos.

5.4.4 Execução de uma árvore sintática em LISP

No caso da implementação da arquitetura, as árvores sintáticas podem ser armazenadas de diversas maneiras, dentre elas através da codificação LISP apresentada anteriormente. De posse da codificação, para realizar a sua execução para uma

determinada entrada, deve-se percorrer a chamada LISP identificando cada uma das chamadas de módulo e função.

Cada módulo contém a sua implementação, que nos casos dos geradores, codificadores e separadores apresentam uma chamada de função para a respectiva função (geradora, codificadora e separadora) em seu processamento no momento de descida do token, enquanto que o módulo seletor apresenta a chamada de função (seletora) em seu processamento no momento de subida dos tokens. O módulo identificador apresenta apenas um tipo de processamento, o qual tem também a chamada para uma função identificadora. Os módulos duplicadores não apresentam em sua implementação chamadas para as funções.

Sendo assim, a codificação LISP é utilizada unicamente para definir os passos de execução de uma árvore sintática, na qual é processada através da implementação de seus módulos e funções.

5.5 Considerações finais

Este capítulo explicou toda a arquitetura proposta no trabalho, a qual tem como objetivo produzir um programa que melhor realize a correção de textos. A técnica escolhida para ser utilizada consiste da programação genética, que é uma solução de aprendizado de máquina, combinada com a técnica de aplicação de regras, criando deste modo uma solução híbrida no que tange a aquisição do conhecimento, advindos da: (1) base de treinamento em que o aprendizado de máquina extrai o máximo de informações possíveis e da (2) inserção manual do conhecimento do problema através das funções desenvolvidas pelos usuários.

Como os indivíduos na programação genética são representados no formato de árvore sintática, primeiramente criou-se uma técnica para sua representação e execução, a qual possibilitasse realizar todos os tipos de operações necessárias para a normalização. Em seguida levantou-se a lista com todos os tipos de operações que deveriam ser modeladas através de nós para depois realizar as suas formalizações (apresentadas em 5.4.1).

6 Aplicação da arquitetura e avaliação dos resultados

Este trabalho elaborou a proposta de uma arquitetura que possibilita a normalização de textos com o uso de uma abordagem híbrida de aprendizado de máquina e aplicação de regras. Contudo, a arquitetura consiste apenas na definição da estrutura e do fluxo de etapas que devem ser seguidas, utilizando de diversos componentes externos.

Este capítulo tem como objetivo mostrar o desenvolvimento dos componentes para um determinado cenário de aplicação, expor detalhes de implementação e analisar seus resultados.

6.1 Visão geral da implementação

Visando realizar a análise e validação da arquitetura proposta, desenvolveu-se a implementação de um programa que satisfizesse a todas etapas descritas no capítulo 5. Para isso, utilizou-se a linguagem de programação JAVA (versão 1.7) e diversas APIs disponíveis, dentre as quais destacam-se a *twitter-4j-stream-3.0.5*[13] para coletar tweets, *pdfbox-app-1.8.9* para leitura de arquivos PDFs (utilizado na etapa de criação do léxico) e do *framework* EpochX *Genetic Programming for Research*[14] na versão 1.4.1 para realização do módulo da programação genética. O código do programa que simula as etapas da arquitetura proposta pode ser encontrado em https://svn.riouxsvn.com/corrector.

A classe Token (mostrada na Figura 30) representa os tokens manipulados pela aplicação. Esta contém seis atributos: (1) tokenAtual, armazena o estado atual do token, o qual pode ser alterado em cada função codificadora e geradora que o token passar; (2) tokenOriginal, guarda o estado inicial do token e portanto quando este passa por uma função do tipo codificadora não tem seu valor alterado, entretanto quando é enviado a uma função geradora este também é fragmentado (por exemplo, o token “Vc ñ” ao ser divido pelo caractere de espaço gera uma instância com tokenOriginal tendo o valor “Vc” e outra com “ñ”); (3) tokenCorrecao, recebe valor apenas quando o token atinge as funções identificadoras e a respectiva sugestão de correção para o token é armazenada nesta variável (a qual só é alterada na situação em que as funções seletoras modificarem a sugestão); (4 e 5) posInicio e posFim, consistem nas variáveis de controle que indicam a posição na qual o token se localiza em relação a frase inicialmente inserida na árvore sintática, sendo assim estas só sofrem alteração quando passam pelas funções geradoras

[13] http://twitter4j.org/en/index.html
[14] http://www.epochx.org/

em que tem seus valores ajustados de acordo com a regra de geração (por exemplo, se o token inicial corresponder a “Vc ñ” e este sofrer a tokenização, o objeto referente a “Vc” terá posInicio e posFim valendo 0 e 2 respectivamente, enquanto que o objeto referente a “ñ” terá os valores 3 e 4); e (6) caminho, a qual armazena todas as funções (de todos os tipos) em que o token recebeu processamento.

```
public class Token implements Comparator<Token>{

    public String tokenAtual;
    public String tokenOriginal;
    public String tokenCorrecao;

    public int posInicio;
    public int posFim;

    private String caminho = "";

    // Construtores

    public Token(){ }

    public Token(int posInicio, int posFim, String tokenOriginal, String tokenAtual,
            String tokenCorrecao){
        this.posInicio = posInicio;
        this.posFim = posFim;
        this.tokenOriginal = tokenOriginal;
        this.tokenAtual = tokenAtual;
        this.tokenCorrecao = tokenCorrecao;
    }

    // Métodos

    public void addFuncao(String identificador){
        if (caminho.length()==0){ caminho = caminho+"|"+identificador+"|"; }
        else{ caminho = caminho+",|"+identificador+"|"; }
    }

    public String getCaminho(){ return this.caminho; }

    public Token clone(){
        Token clone = new Token();
        clone.posInicio = this.posInicio;
        clone.posFim = this.posFim;
        clone.tokenAtual = this.tokenAtual;
        clone.tokenOriginal = this.tokenOriginal;
        clone.tokenCorrecao = this.tokenCorrecao;
        clone.caminho = this.caminho;
        return clone;
    }

    @Override
    public int compare(Token t1, Token t2) {
        int sub = t1.posInicio-t2.posInicio;
        if (sub!=0){ return sub; }
        sub = t1.posFim-t2.posFim;
        return sub;
    }
}
```

Figura 30 - Implementação da classe Token

Para que os tokens se locomovam através das árvores sintáticas, é necessário o desenvolvimento da estrutura que corresponde aos nós, que são as instâncias entregues à programação genética para realizar a criação da população inicial e das gerações seguintes. Na arquitetura proposta existem cinco tipos de funções (geradoras, codificadoras, separadora, seletoras e identificadoras), as quais são codificadas em

classes “nós” que devem estender Node.java (classe presente no EpochX), tal que o *framework* compreenda estas como o conjunto de componentes a qual dará origem às árvores.

A Figura 31 apresenta a visão geral da disposição das classes, a qual se estende desde Node.java presente dentro do *framework* até as classes de função implementadas pelos usuários e que consistem de uma das entradas externas da arquitetura. As classes “Nós” (NoGerador, NoCodificador1, NoCodificador2, NoSeparador, NoSeletor e NoIdentificador) apresentam parte da codificação referente ao seu respectivo módulo além de uma instância da classe de função armazenada na variável da interface correspondente.

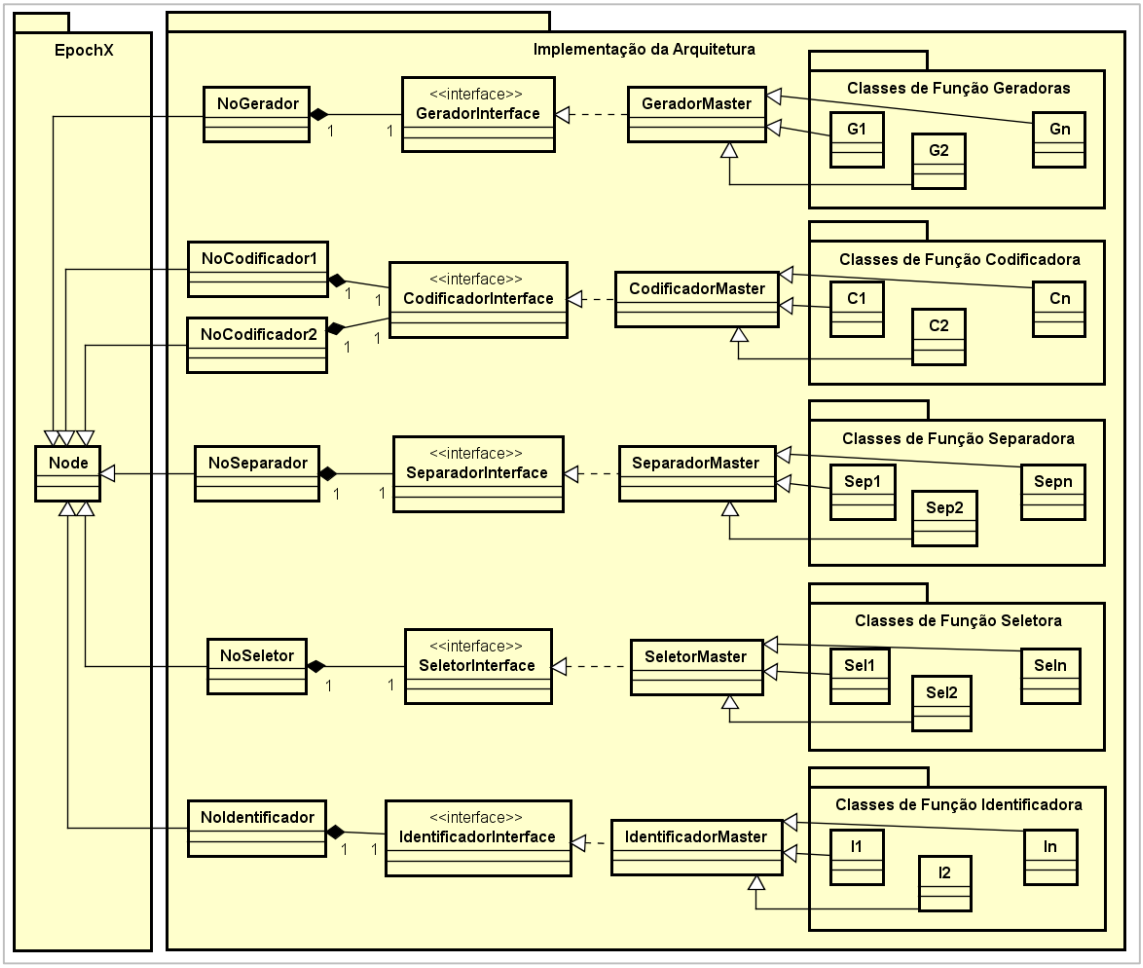

Figura 31 - Disposição das classes na implementação

Cada uma das interfaces presentes em seu respectivo nó descreve uma lista de métodos que devem ser implementados pelas classes (1) masters e (2) de função. A primeira contém a implementação dos métodos de controle enquanto a segunda apresenta a respectiva regra do usuário codificada (podendo esta ser uma função geradora, codificadora, separadora, seletora ou identificadora). Para ilustrar toda a

cadeia de classes e interfaces mostradas na Figura 31, são explicados com maiores detalhes (devido a sua simplicidade) as classes que definem um nó do tipo separador[15].

A Figura 32 apresenta a classe NoSeparador. Esta estende a classe Node (pertencente ao EpochX), apresenta três atributos, cinco métodos e um construtor. Entre os atributos encontram-se: (1) identificadorModulo, o qual identifica numericamente o módulo separador; (2) prefixo, que guarda o prefixo do módulo separador para poder realizar a criação da codificação LISP e identificar cada nó (composto pelo prefixo do módulo e identificação da classe de função); e (3) funcaoSeparadora, que consiste de uma interface que salva a instância da classe de função do tipo separadora.

```
public class NoSeparador extends Node {

    public static final int identificadorModulo = 7;
    protected String prefixo = "sep.";
    private SeparadorInterface funcaoSeparadora;

    // Construtor

    public NoSeparador(SeparadorInterface funcaoSeparadora) {
        super(null, null);
        this.setIdNodeModulo(identificadorModulo);
        this.funcaoSeparadora = funcaoSeparadora;
    }

    // Métodos do Nó

    public SeparadorInterface getSeparador(){ return this.funcaoSeparadora; }

    @Override
    public String getIdentifier(){ return prefixo+funcaoSeparadora.getIdentifier(); }

    public static int separa(Token tokenGP, SeparadorInterface separador){
        return separador.canal(tokenGP);
    }

    // Métodos do framework EpochX

    @Override
    public Boolean evaluate() { return null; }

    @Override
    public Class<?> getReturnType(final Class<?> ... inputTypes) { return Void.class; }
}
```

Figura 32 - Implementação da classe NoSeparador

Entre os métodos presentes em NoSeparador estão: (1) getSeparador, o qual retorna a função separadora acoplada ao nó; (2) getIdentifier, que devolve a string que identifica o nó (formada pelo prefixo do módulo mais a identificação da classe de função); (3) separa, que consiste no método referente ao módulo separador que recebe um token e uma função separadora, para então realizar a chamada do método "canal" (especificada na interface separadora explicada mais adiante) que retorna a indicação de por qual canal de comunicação o token deve prosseguir; (4 e 5) os dois últimos métodos

[15] As explicações dos demais nós podem ser encontrados no APÊNDICE B.

correspondem a implementações obrigatórias em classes que estendem Node.java, e servem unicamente para configuração do *framework*. O construtor por sua vez é responsável por informar para o *framework* quantos filhos o nó deve conter, especificado através da chamada do construtor de Node pela função “super”, passando o número de parâmetros nulos correspondente ao número de filhos.

A especificação da interface SeparadorInterface, que contém a instância da classe de função separadora é especificada na Figura 33, onde seus métodos correspondem a: (1) funcaoCanal, que é implementada dentro da classe de função separadora e tem como objetivo definir a indicação do canal de comunicação que o token deve prosseguir; (2) getIdentifier, responsável por capturar a identificação da classe de função; (3) canal, chamada na classe master antes do método funcaoCanal para que seja contabilizado as estatísticas de processamento; e (4, 5, 6 e 7) as demais correspondem aos métodos para controle de tempo de execução e quantidade de chamadas realizadas.

```
public interface SeparadorInterface {

    // método a ser implementado por cada classe de função separadora
    public int funcaoCanal(Token tokenGP);

    // métodos a serem implementados pela classe master
    public String getIdentifier();
    public int canal(Token tokenGP);

    // métodos utilizados na classe master para controle
    public long getTempo();
    public long getChamadas();
    public long getRelacao();
    public void zera();
}
```

Figura 33 - Implementação da interface SeparadorInterface

A implementação da classe SeparadorMaster pode ser vista na Figura 34, onde se pode destacar a implementação do método “canal”, que gera as estatísticas de tempo de execução e quantidade de chamadas realizadas, além de disparar a chamada para o método “funcaoCanal” presente na classe de função separadora.

```
public abstract class SeparadorMaster implements SeparadorInterface{

    public AtomicLong tempo = new AtomicLong(0);
    public AtomicLong chamadas = new AtomicLong(0);

    public String getIdentifier(){ return this.getClass().getSimpleName(); }

    @Override
    public int canal(Token tokenGP) {
        long t1 = System.nanoTime();
        int grupo = funcaoCanal(tokenGP);
        long t2 = System.nanoTime();
        tempo.addAndGet((t2-t1));
        chamadas.incrementAndGet();
        return grupo;
    }

    @Override
    public long getTempo() { return tempo.longValue(); }

    @Override
    public long getChamadas() { return chamadas.longValue(); }

    @Override
    public long getRelacao() {
        if (chamadas.longValue()>0){ return tempo.longValue()/chamadas.longValue(); }
        return 0;
    }

    @Override
    public void zera(){ tempo.set(0); chamadas.set(0); }
}
```

Figura 34 - Implementação da classe SeparadorMaster

Um exemplo de classe de função separadora é mostrado na Figura 35, onde esta retorna a indicação do primeiro canal de comunicação (valor 0) para um token formado por apenas um dígito, enquanto indica o segundo canal (valor 1) caso contrário.

```
public class G0tamanhoTokenIgual1_G1outros extends SeparadorMaster {

    @Override
    public int funcaoCanal(Token tokenGP) {
        int tamanho = tokenGP.tokenAtual.length();
        if (tamanho==1){
            return 0;
        }
        return 1;
    }
}
```

Figura 35 - Exemplo de classe de função separadora

6.2 Cenário de aplicação

O cenário para aplicação da arquitetura, escolhido para sua análise e validação, consiste da normalização de tweets em português. Estes, como já amplamente discutido no trabalho, contam com várias categorias de erros de escritas e estão limitados a 140 caracteres.

Visando atender a este cenário, foram desenvolvidas as classes de função adicionando o conhecimento necessário segundo estudos apresentados nos capítulos anteriores e adaptado para os tweets; além de se criar uma base anotada de tweets,

contendo seus respectivos textos originais associados a suas formas normalizadas. Por fim, ainda se fez a criação de um léxico para o português contendo a frequência de aparecimento de cada palavra em textos escritos em linguagem culta.

6.3 Classes de função

As classes de função consistem das codificações das regras inseridas pelo(s) usuário(s) na arquitetura. As subseções a seguir mostram as funções criadas que mais adiante são utilizadas nas execuções para validação.

6.3.1 Separadoras

Como já explicado em 6.1 na estrutura de classes de um nó separador, as classes de função separadora têm por objetivo indicar o canal de comunicação por onde o token deve prosseguir. Para ajudar no controle da criação das diversas funções é criado uma notação especial para nomeação das classes tal que esta ajude a identificar o seu respectivo funcionamento. Os nomes são formados por três partes (indicadas pela utilização de cores distintas):

G0*[canal0]*_G1*[canal1]*_GE*[retorno]*

A primeira parte deve conter a indicação do critério utilizado para guiar os tokens pelo primeiro canal, enquanto a segunda parte define o critério utilizado para o segundo canal. A terceira parte é opcional e se apresenta quando existe um critério adicional que define que o token não deve seguir por nenhum dos dois canais de comunicação em direção aos nós folhas, mas sim iniciar seu retorno ao nó raiz, sem conter portanto nenhuma sugestão de correção. Os trechos em itálico são respectivamente substituídos pela indicação do critério correspondente. A Tabela 10 apresenta a descrição das funções separadoras desenvolvidas.

Tabela 10 – Funções separadoras

[canal0]	***[canal1]***	***[retorno]***
Critério de seleção	Critério de seleção	Critério de seleção
apenasConsoantes Seleciona tokens compostos apenas por consoantes	**apenasConsoantesEvogais** Seleciona tokens compostos apenas por letras e que não foram selecionados pelo canal 0	**outros** Seleciona os demais tokens não selecionados pelos canais 0 e 1
apenasLetras Seleciona tokens compostos apenas por letras	**outros** Seleciona os demais tokens não selecionados pelo canal 0	

apenasLetrasComRepeticao Seleciona tokens compostos apenas por letras, tal que existam pelo menos dois caracteres iguais consecutivos em sua formação	**apenasLetrasSemRepeticao** Seleciona tokens compostos apenas por letras e não selecionados pelo canal 0	**outros** Seleciona os demais tokens não selecionados pelos canais 0 e 1
apenasLetrasComRepeticao TratamentoEspecialRS Seleciona tokens compostos apenas por letras, tal que existam pelo menos dois caracteres iguais consecutivos em sua formação (com tratamento especial para as letras R e S)	**apenasLetrasSemRepeticao** Seleciona tokens compostos apenas por letras e não selecionados no canal 0.	**outros** Seleciona os demais tokens não selecionados pelos canais 0 e 1
letrasEnumeros AmbosEapenas Seleciona tokens que tenham em sua composição tanto letras como números e sejam formados apenas por estes	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
letraTseguidoDeUmaVogal EmTokenDe2Caracteres Seleciona tokens de dois caracteres em que o primeiro consiste na letra **T** e o segundo em uma vogal	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
primeiraLetraVogal Seleciona tokens em que o primeiro caractere consiste de uma vogal	**primeiraLetraConsoante** Seleciona tokens em que o primeiro caractere consiste de uma consoante	**outros** Seleciona os demais tokens não selecionados pelos canais 0 e 1
tamanhoTokenIgual1 Seleciona tokens com 1 caractere	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
tamanhoTokenIgual2 Seleciona tokens com 2 caracteres	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
tamanhoTokenIgual3 Seleciona tokens com 3 caracteres	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
tamanhoTokenIgualMenor2 Seleciona tokens com 2 ou menos caracteres	**outros** Seleciona os demais tokens não selecionados pelo canal 0	

tamanhoTokenIgualMenor3 Seleciona tokens com 3 ou menos caracteres	**outros** Seleciona os demais tokens não selecionados pelo canal 0	
tokenCaracteristicoCerto Seleciona tokens que consistirem de uma representação válida para o português da disposição de vogais e consoantes	**outros** Seleciona os demais tokens não selecionados pelo canal 0	

6.3.2 Codificadoras

As classes de função codificadora podem ser acopladas em dois tipos de nós (como mostrado na Figura 18), para os quais a explicação da estrutura de classes encontra-se no APÊNDICE B.

Tendo em vista o tempo de processamento, três variáveis extras são definidas em cada uma das funções codificadoras, que ajudam as funções identificadoras a encontrar dentro do léxico seus respectivos mapeamentos:

1. obedeceLeiCanonica: indica se após a aplicação da codificação no token, este ainda pode obter seus possíveis mapeamentos no léxico através da combinação das suas respectivas representações canônicas;
2. obedeceLeiCanonicaOrdenada: indica se após a aplicação da codificação no token, este ainda pode obter seus possíveis mapeamentos no léxico através da combinação das suas respectivas representações canônicas com as letras ordenadas;
3. obedeceLeiCanonicaOrdenadaSemRepeticao: indica se após a aplicação da codificação no token, este ainda pode obter seus possíveis mapeamentos no léxico através da combinação das suas respectivas representações canônicas com as letras ordenadas e com posterior remoção de repetição de letras.

Visando separar os tipos de funções codificadoras, são criados três grupos distintos de funções, em que cada um recebe uma notação especial. O primeiro grupo corresponde aos codificadores (em que codificam o token) no qual sua notação consiste de três partes:

C[codificação]_N[descarte]_[variáveis]

A primeira parte contém a indicação da operação de codificação. A segunda consiste de um trecho opcional, sendo utilizado quando devido a alguma característica

(especificada no campo “descarte”) o token não é codificado e ao invés de seguir caminhando em direção aos nós folhas, inicia seu retorno para o nó raiz. A última parte referente às variáveis deve corresponder a quatro caracteres com cada um podendo valer T (*true*/verdadeiro) ou F (*false*/falso). Estes se referem ao valor das seguintes quatro variáveis (na respectiva ordem): (1) processaTokenDicionario (explicado em 5.4.1.2.2), (2) obedeceLeiCanonica, (3) obedeceLeiCanonicaOrdenada e (4) obedeceLeiCanonicaOrdenadaSemRepeticao. A Tabela 11 apresenta a funções codificadoras pertencentes a este primeiro grupo.

Tabela 11 – Funções codificadoras (grupo dos codificadores)

[codificação]	***[descarte]***	***[variáveis]***
Operação de codificação	Critério de descarte	
caixaBaixa Coloca todos os caracteres em caixa baixa		**TTTT**
esqueletoConsoante Coloca token no formato canônico eliminando todas as repetições de caracteres consecutivos	**tokensComAlgoDiferenteDeLetra** Descarta tokens que contiverem caracteres que não corresponderem a letras	**TFFT**
ordemAlfabetica Coloca as letras do token em ordem alfabética	**tokensComAlgoDiferenteDeLetra** Descarta tokens que contiverem caracteres que não corresponderem a letras	**TFTT**
primeiraLetraConsoantesVogais Mantém a primeira letra do token intocada, e com as demais coloca primeiramente todas as consoantes para em seguida colocar todas as vogais	**tokensComAlgoDiferenteDeLetra** Descarta token que contiverem caracteres que não corresponderem a letras	**TTTT**

O segundo grupo de classe de função codificadora corresponde aos removedores, que realizam uma operação de remoção de uma determinada característica presente no token. Sua notação consiste de três partes:

R*[remoção]*_N*[descarte]*_*[variáveis]*

A primeira parte consiste da indicação da operação de remoção a qual é realizada nos tokens, enquanto que as duas últimas partes têm o mesmo significado do grupo codificador. A Tabela 12 apresenta as funções codificadoras do grupo dos removedores.

Tabela 12 – Funções codificadoras (grupo dos removedores)

[remoção] Operação de remoção	***[descarte]*** Critério de descarte	***[variáveis]***
acentos Remove os acentos presentes no token		**TTTT**
espacos Remove os espaços presentes no token		**FTTT**
letraRepetidaComTratamentoRSnaoTrataCaixaEacento Remove caracteres consecutivos repetidos dando tratamento especial para as letras R e S, além de considerar letras com acentos diferentes e em caixas diferentes como caracteres distintos		**FFFT**
vogais Remove as vogais presentes no token		**TTTT**

O último grupo corresponde aos substituidores, que realizam a substituição de uma determinada característica no token por outra. Sua notação consiste também de três partes:

S*[substituição]*_N*[descarte]*_*[variáveis]*

As duas últimas partes contêm o mesmo significado que apresentaram nos dois primeiros grupos, enquanto que a primeira parte indica a operação de substituição que a função realiza. A Tabela 13 apresenta as funções codificadoras do grupo dos substituidores.

Tabela 13 – Funções codificadoras (grupo dos substituidores)

[substituição] Operação de substituição	***[descarte]*** Critério de descarte	***[variáveis]***
hPorAcentoAgudoNoFim Caso o token termine com a letra “h”, a remove e insere um acento agudo na letra anterior	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador[16]	**FTTT**
iPorEnoFim Caso o token termine com a letra “i”, a substitui por “e”	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**
kPorC Substitui todas as letras “k” por “c” no token	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**

[16] Um separador pode corresponder a um espaço em branco ou qualquer tipo de pontuação.

kPorCA Substitui todas as letras “k” pela sequência de letras “ca” no token	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**
kPorQu Substitui todas as letras “k” pela sequência de letras “qu” no token	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**
uPorOnoFim Caso o token termine com a letra “u”, a substitui por “o”	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**
wPorU Substitui todas as letras “w” por “u”	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**
xPorCH Substitui todas as letras “x” pela sequência de letras “ch”	**tokensComSeparador** Descarta tokens que apresentarem em sua composição um separador	**FTTT**

6.3.3 Geradoras

As classes de função geradoras têm a estrutura de classes parecida com as separadoras anteriormente explicadas, a qual é mostrada no APÊNDICE B. Sua notação consiste de apenas uma parte:

T*[operação]*

O campo “operação” deve conter assim a indicação do tipo de operação de geração de tokens a ser realizado na função. A Tabela 14 apresenta as operação de geração desenvolvidas.

Tabela 14 – Funções geradoras

[operação]
Operação de geração
espaco Gera tokens a partir da divisão do token original pelos espaços presentes
espacoCombinado Gera tokens a partir da combinação dos fragmentos gerados da divisão do token original pelos espaços presentes
letrasCombinadas Gera tokens a partir da combinação dos caracteres presentes no token original
pontuacao Gera tokens a partir da divisão do token original pelas pontuações presentes

6.3.4 Seletoras

As classes de função seletoras (onde sua estrutura é explicada no APÊNDICE B) são divididas em dois grupos, onde o primeiro consiste no tipo filtro e o segundo no tipo desempatador. O primeiro grupo tem como objetivo promover a eliminação de candidatos a correção que não satisfizerem a uma determinada característica. Sua notação de classe é dividida em três partes:

F*[força]*[filtro]_D*[desconsiderar]*

A primeira parte corresponde a força do filtro, a qual pode assumir apenas dois valores pré-definidos (FORTE ou FRACO). O filtro do tipo forte realiza a eliminação dos candidatos que não satisfizerem ao critério exigido independente de tudo, enquanto os filtros fracos abortam as eliminações caso tenham sido descartadas todas as sugestões de correção para determinado vocábulo. A segunda parte da notação corresponde a indicação do tipo de filtragem em que a classe realiza. A última parte por fim, consiste de um trecho optativo, em que indica uma ou mais características em que o filtro não leva em consideração para realizar sua avaliação. A Tabela 15 apresenta as funções criadas para este primeiro grupo.

Tabela 15 – Funções seletoras (grupo dos filtros)

[força]	***[filtro]*** Operação de filtragem	***[desconsiderar]*** O que desconsiderar
FORTE	**letrasOriginalPresentesEmCorrecaoNaMesmaOrdem** Todas as letras presentes no token original devem estar presentes (na mesma ordem de aparecimento) na sugestão de correção	**caixaEacentos** Desconsiderar a caixa das letras e suas acentuações
FRACO		
FORTE	**mesmasLetrasOriginalEcorrecaoSemConsiderarRepeticaoDeLetras** As letras presentes no token original e na sugestão devem ser as mesmas (desconsiderando as suas repetições)	**caixaEacentos** Desconsiderar a caixa das letras e suas acentuações
FRACO		
FORTE	**originalConsistidoPelaPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem** Cada letra do token original deve corresponder a primeira letra de cada sílaba da sugestão (respeitando a ordem de aparecimento)	**caixaEacentos** Desconsiderar a caixa das letras e suas acentuações
FRACO		
FORTE	**originalContendoPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem**	**caixaEacentos** Desconsiderar a caixa das

FRACO	Todas as primeiras letras das sílabas presentes na sugestão devem aparecer no token original (respeitando a ordem de aparecimento)	letras e suas acentuações
FORTE	**originalContidoEmCorrecao**	**caixaEacentos**
FRACO	Token original deve estar contido na sugestão	Desconsiderar a caixa das letras e suas acentuações
FORTE	**originalIgualCorrecao**	**caixa**
FRACO	Token original deve ser igual a sugestão	Desconsiderar a caixa das letras
FORTE	**originalIgualCorrecao**	**caixaEacentos**
FRACO	Token original deve ser igual a sugestão	Desconsiderar a caixa das letras e suas acentuações
FORTE	**originalIgualCorrecao**	
FRACO	Token original deve ser igual a sugestão	
FORTE	**originalPrefixoDeCorrecao**	**caixaEacentos**
FRACO	Token original deve ser prefixo da sugestão	Desconsiderar a caixa das letras e suas acentuações
FORTE	**originalSufixoDeCorrecao**	**caixaEacentos**
FRACO	Token original deve ser sufixo da sugestão	Desconsiderar a caixa das letras e suas acentuações
FORTE	**primeiraLetraOriginalIgualPrimeiraLetraCorrecao**	**caixaEacentos**
FRACO	Token original deve ter a mesma primeira letra da sugestão	Desconsiderar a caixa das letras e suas acentuações

O segundo grupo por sua vez corresponde aos desempatadores, que se diferem dos filtros por seu tratamento consistir da comparação entre as sugestões, enquanto que no grupo dos filtros as sugestões são analisadas individualmente. A notação de classe desse grupo é dividida em duas partes:

D*[desempatador]*_D*[desconsiderar]*

Enquanto a primeira parte consiste na indicação da operação de desempate, a segunda consiste de uma parte opcional com mesma função desempenhada no grupo dos filtros. A Tabela 16 apresenta as funções deste grupo.

Tabela 16 – Funções seletoras (grupo dos desempatadores)

[desempatador]	***[desconsiderar]***
Critério de desempate	O que desconsiderar
descarteSugestoesInternasAsOutras Descarta as sugestões de correção internas à outras	

descarteSugestoesSemelhantesDiferenciandoApenasEmCaixa Descarta as sugestões em que apresentem como diferença apenas a caixa da letra. Deixar apenas um representante	
descarteSugestoesSemelhantesDiferenciandoApenasEmCaixaEacentos Descarta as sugestões em que apresentem como diferença apenas a caixa da letra e a presença/ausência de acentuações. Deixar apenas um representante	
maiorFrequenciaDicionario Seleciona dentre as sugestões de correção para um vocábulo, aquela de maior frequência no léxico	
maiorFrequenciaDeBigramasEmComumComPrimeiraLetraIgual Seleciona dentre as sugestões de correção para um vocábulo, aquela que apresenta maior quantidade de bigramas (de letras) em comum com o token original	**caixa** Desconsiderar a caixa das letras
maiorFrequenciaDeTrigramasEmComumComPrimeiraLetraIgual Seleciona dentre as sugestões de correção para um vocábulo, aquela que apresenta maior quantidade de trigramas (de letras) em comum com o token original	**caixa** Desconsiderar a caixa das letras
menorDistanciaLevenstein Seleciona dentre as sugestões de correção para um vocábulo, aquela que apresentar menor distância de edição com o token original	**caixa** Desconsiderar a caixa das letras
menorDistanciaLevensteinDeCodificacoesMetaphonePro Seleciona dentre as sugestões de correção para um vocábulo, aquela que apresentar menor distância de edição de sua forma codificada pelo algoritmo Metaphone e o token original codificado pelo algoritmo MetaphonePro[17]	
preenchimentoMaisCompletoComMenorNumeroTokens Analisando as sugestões de correção entre todos os vocábulos, seleciona o conjunto que realizar o preenchimento mais completo em relação a normalização da frase com o menor número de tokens	

6.3.5 Identificadoras

As classes de função identificadora (com sua estrutura explicada no APÊNDICE B) são divididas em três grupos. O primeiro representa o grupo de funções que fazem acesso ao dicionário (léxico) inserido na arquitetura. Sua notação conta com apenas uma parte:

D[busca]

[17] MetaphonePro - Variação do algoritmo Metaphone desenvolvido por este trabalho, que leva em consideração a sonorização da vogal **e** por algumas letras quando estão em internetês

O campo “busca” tem a indicação do tipo de operação de busca realizada pela função. A Tabela 17 descreve os métodos desenvolvidos para esse grupo.

Tabela 17 – Funções identificadoras (grupo do acesso ao dicionário)

[busca]
Operação de busca realizada no dicionário
igual
Busca no léxico os tokens que, quando aplicados a mesma sequência de codificadores que influenciam o dicionário, forem idênticos ao token original recebido como entrada do método
igualDiminutivo
Busca no léxico os tokens que, quando aplicados a mesma sequência de codificadores que influenciam o dicionário, forem idênticos ao token original recebido como entrada no método com a substituição do diminutivo (terminado em “inho” ou ‘inha”) por sua forma normal

O segundo grupo corresponde aos métodos que fazem acesso ao léxico e que necessitam realizar uma operação de codificação sobre este. Os vocábulos do léxico quando codificados são utilizados unicamente para a mapear com o token recebido, o qual também pode sofrer uma operação de codificação. Uma vez que as instâncias do léxico são selecionadas, a lista criada para retorno conta com o estado original dos vocábulos. Para garantir a velocidade de execução do processamento, as codificações alternativas do léxico estabelecidas pelas funções são previamente realizadas, e apenas consultadas ao longo de sua execução. A notação para nomeação das classes deste grupo conta com apenas uma parte:

M*[busca]*

O campo “busca” representa a função de codificação a qual o léxico é submetido, além da regra de mapeamento entre o token recebido e os vocábulos do léxico codificados. A Tabela 18 apresenta as classes deste grupo.

Tabela 18 – Funções identificadoras (grupo do acesso ao léxico codificado)

[busca]
Operação de busca realizada no léxico mapeado
soundexEn
Busca todos os tokens do léxico tal que sua codificação Soundex para o inglês for igual a codificação (Soundex para inglês) do token recebido
soundexPt
Busca todos os tokens do léxico tal que sua codificação Soundex para o português for igual a codificação (Soundex para português) do token recebido

tokenComCadaLetraInicioDeUmaSilaba
Busca todos os tokens no léxico tal que estes em sua forma codificada contêm apenas a primeira letra de cada sílaba e sejam iguais ao token recebido
tokenComCadaLetraInicioENasalizacaoPorSilaba
Busca todos os tokens no léxico tal que estes em sua forma codificada contêm apenas a primeira letra de cada sílaba e mais suas respectivas letras de nasalização e sejam iguais ao token recebido

O último grupo corresponde as funções de identificação, que não fazem uso do léxico e são responsáveis por identificar um tipo de token. Sua notação consiste de apenas uma parte:

I*[identificação]*

O campo "identificação" deve conter o objeto a qual se deseja identificar. A Tabela 19 apresenta as classes desse grupo.

Tabela 19 – Funções identificadoras (grupo dos identificadores)

[identificação]
O que se deseja identificar
carinha
Identifica os *emoticons* em que representem rostos
hashtag
Identifica as *hashtags*
mention
Identifica as *mentions*
risosHuae
Identifica risos formados apenas pelo conjunto ou subconjunto de letras **h**, **u**, **a**, **e**
risosRs
Identifica risos formados apenas pelo conjunto ou subconjunto de letras **r**, **s**
url
Identifica URLs

Além dos três grupos de classes de função identificadora, ainda é adicionada uma classe extra com nome "*Nothing*", a qual tem como função apenas retornar o token recebido sem nenhuma sugestão. Essa função adicional é útil para que caminhos bifurcados possam ser interrompidos sem precisarem forçosamente implementar um tipo de sugestão (a qual pode prejudicar o resultado).

6.4 Base de tweets anotados

Uma vez que de acordo com as pesquisas realizadas neste trabalho não se encontrou uma base de tweets em português respectivamente anotada na Web, fez-se o seu desenvolvimento com intuito de utilizá-la para avaliar a arquitetura proposta e também para que demais trabalhos possam fazer uso da mesma para futuras comparações. A base desenvolvida está disponível para *download* em https://svn.riouxsvn.com/corrector/base_tweets_anotada.

Utilizando da API *twitter-4j-stream-3.0.5* para JAVA, recolheu-se um total de 6000 tweets, dentre os quais selecionou-se de forma aleatória 10% para anotação. Tem-se então para cada um dos 600 tweets selecionados além de seu texto original, também o texto no formato correto de escrita, mesmo que o original já se apresente corretamente grafado.

Para o processo de anotação da base, adotou-se as regras descritas na Tabela 20. Como pode ser percebido, são dados tratamentos especiais aos tokens específicos dos microblogs visando apenas identificá-los e mantê-los de forma intacta no tweet. A opção deste tipo de tratamento para esses tokens visa atender aos algoritmos de processamento de linguagem natural que necessitam entender quando um token é especifico do contexto (por exemplo as *hashtags* e *mentions*) mas que os utilizam para ajudar seu processamento (por exemplo na utilização dos *emoticons* para definir se o tweet trata-se de uma mensagem positiva ou negativa).

Com relação a linguagem utilizada nos tweets, por muitas vezes esta pode ser dúbia quanto ao seu significado, dificultando assim o entendimento quanto a correta pontuação a ser utilizada. Por este motivo, as pontuações presentes no texto original são mantidas, sem a realização de qualquer tipo de exclusão ou inserção.

Tabela 20 – Regras para realização da anotação da base de tweets

Nº	Regra
1	Todos os tokens que corresponderem a mais de uma palavra aglutinadas são corretamente separadas. Exemplo: Texto original: Vamos nósdois juntoshoje Aplicada a regra 1: Vamos nós dois juntos hoje
2	Todas as palavras que estão divididas em mais de um token são juntadas corretamente. Exemplo: Texto original: Estou A N I M A D A!!! Aplicada a regra 2: Estou ANIMADA!!!

3	Os tokens que corresponderem a uma *mention* são envoltas por @@ e a partir deste momento não sofrem mais qualquer tipo de alteração das regras seguintes. Exemplo: Texto original: @fferman vai hoje Aplicada a regra 3: @@@fferman@@ vai hoje
4	Os tokens que corresponderem a uma *hashtag* são envoltas por ## e a partir deste momento não sofrem mais qualquer tipo de alteração das regras seguintes. Exemplo: Texto original: Vai #mengaaao Aplicada a regra 4: Vai ###mengaaao##
5	Os tokens que corresponderem a uma URL são envoltas por && e a partir deste momento não sofrem mais qualquer tipo de alteração das regras seguintes. Exemplo: Texto original: Acessa ai www.cos.ufrj.br Aplicada a regra 5: Acessa ai &&www.cos.ufrj.br&&
6	Os tokens que corresponderem a uma representação de risada são envoltos por §§ e a partir deste momento não sofrem mais qualquer tipo de alteração das regras seguintes. Exemplo: Texto original: kkkkkk, você que fez besteira huahauhahua Aplicada a regra 6: §§kkkkkk§§, você que fez besteira §§huahauhahua§§
7	Os tokens que corresponderem a um *emoticon* são envoltos por ££ e a partir deste momento não sofrem mais qualquer tipo de alteração das regras seguintes. Por exemplo: Texto original: S2 adorei ontem =] Aplicada a regra 7: ££S2££ adorei ontem ££=]††
8	Como será mostrado nas regras seguintes, é permitido o uso de alguns caracteres de expressão regular para indicação de mais de uma possibilidade de correção para um ou mais determinados vocábulos. Portanto, são primeiramente escapados (utilizando colchete) todos os caracteres especiais utilizados (***? ( ) | ****). Exemplo: Texto original: Você vai amanhã? Aplicada a regra 8: Você vai amanhã[?]
9	Todo token que estiver grafado de forma incorreta é corrigido por sua escrita na forma culta. Exemplo: Texto original: Vc vai cmg tbm Aplicada a regra 9: Você vai comigo também No caso de se ter mais de uma possibilidade de correção, pode-se fazer uso dos caracteres de expressão regular (***? ( ) | ****). Exemplo: Texto original: Isso é p você Aplicada a regra 9 (com uso de expressão regular): Isso é pa?ra você
10	Dado a existência de palavras que não estão presentes na norma culta da língua mas que não consistem de palavras com erros de grafia, são envoltos por ¢¢ todas as palavras que não estiverem presentes no Dicionário Didático – Ensino Fundamental – Com Nova Ortografia (Edição 2008 / Editora SM) e não corresponderem a uma alteração de uma palavra presente. Exemplo: Texto original: Vou dar um rolê hoje Aplicada a regra 10: Vou dar um ¢¢rolê¢¢ hoje

6.5 Formação do léxico

Para que a tarefa de normalização se realize através da arquitetura proposta, é necessário o fornecimento de uma base léxica contendo a relação de palavras válidas do idioma para consulta. No cenário a ser aplicado, além das palavras também é útil para uso da frequência de aparecimento de cada vocábulo na base, para que assim se possa utilizar deste como um parâmetro adicional nas funções.

Dos léxicos existentes disponíveis identificados na pesquisa deste trabalho têm-se a base do ReGra (a qual conta com mais de 340 mil tokens junto com suas respectivas frequências de aparecimento) e a base do Unitex-PB chamada DELAS_PB (com aproximadamente 67 mil palavras porém sem suas frequências). Contudo, a utilização da base do ReGra necessitaria um tratamento inicial, pois além de apresentar as palavras válidas do português, apresenta diversas palavras pertencentes ao internetês, a qual se pretende corrigir neste trabalho e portanto um léxico com a sua presença comprometeria a qualidade do resultado.

Com as dificuldades apresentadas em cada uma das bases, fez-se então a criação de uma nova base a partir de textos de histórias (encontrados no formato PDF). Estes apresentam uma baixa probabilidade de conterem termos não pertencentes ao idioma, além de possibilitarem a contagem de frequência (mesmo que esta pertença a outro contexto).

Para seleção dos livros a serem utilizados para compor o léxico, escolheu-se de forma aleatória dez livros presentes em http://lelivros.website/, os quais são disponibilizados no formato PDF. Com a utilização da API PDFBox, fez-se a leitura dos livros e cada palavra encontrada foi inserida na base do léxico, atualizando sua respectiva frequência de aparecimento. A base criada conta com mais de 32 mil tokens e está disponível para *download* em https://svn.riouxsvn.com/corrector/palavras.backup.

6.6 Função de avaliação

Neste trabalho optou-se pela utilização de uma métrica tradicional para avaliação dos programas de normalização, o BLEU. A escolha desta métrica foi feita a partir das características do problema, onde para cada texto há mais de uma possibilidade de normalização. O BLEU é então utilizado em cada geração para avaliação de cada programa em cima dos elementos pertencentes a fase de treinamento.

Como o contexto em que quer-se aplicar a normalização neste trabalho apresenta uma grande quantidade de erros, é feita uma filtragem na base de treinamento realizando a validação em cada geração apenas para os tweets que inicialmente apresentam erros, deixando os que inicialmente já estão certos de lado, crendo que as palavras corretas presentes nos tweets errados se encarregarão de preservar os tokens certos intactos. Deste modo é possível ganhar tempo de processamento devido à não execução de programas a cada geração para os tweets que inicialmente já se encontram corretos.

Dentre os tipos de erros em um tweet a se normalizar, são desconsiderados os problemas de caixa alta e caixa baixa devido a não criação das regras que visem este tipo de correção, além da não alteração das pontuações inicialmente presentes nos tokens, uma vez que a base de tweets anotada preserva as pontuações do texto original.

6.7 Avaliação dos resultados

Com todos os componentes externos à arquitetura criados, como mostrado nas seções anteriores, é possível realizar a análise do resultado obtido com a aplicação da arquitetura.

6.7.1 Valores obtidos nos trabalhos da literatura

Dos trabalhos presentes na literatura, a métrica utilizada para comparação de resultados para o problema de normalização consiste no BLEU, tal que a Tabela 21 apresenta a descrição dos valores adquiridos por cada trabalho. Contudo, existem dois complicadores para se contrastar o estado da arte com a proposta de um trabalho: (1) dentre as técnicas que utilizam o BLEU para validação, nenhuma é voltada para o português e (2) os trabalhos não utilizam a mesma base e nem as mesmas especificações dos tipos de erros que estão tentando tratar. Deste modo torna-se difícil a comparação com os resultados da literatura.

Se analisados os resultados indicados pelos trabalhos (Tabela 21) nota-se uma discrepância grande entre a quantidade de aumento da métrica BLEU, a qual pode ser percebida até mesmo com a utilização de um mesmo algoritmo variando apenas a base, presente no trabalho de Gadde *et al.* (2011). Contudo, para se obter algum tipo de medida para comparação, calculou-se a média do aumento da métrica BLEU, obtendo um total de 44,41%. Se analisados apenas os cenários onde a base é composta de tweets, tem-se aumento médio de 41,94%.

Tabela 21 – Valores atingidos pelos trabalhos presentes na literatura

Trabalho	**[1]**	**[2]**	**[3]**			**[4]**	**[5]**
Idioma	Inglês	Inglês	Inglês			Alemão	Inglês*
Base	Tweets	SMSs	SMSs[1]	SMSs[2]	SMSs[3]	Tweets	Tweets
BLEU inicial	0,6799	0,4096	0,4480	0,3690	0,4100	0,7929	0,4201
BLEU final	0,7985	0,5390	0,7100	0,5990	0,5440	0,8766	0,8312
Aumento	17,44 %	31,59 %	58,48 %	62,33 %	32,68 %	10,55 %	97,82 %

* Os tweets em inglês são escritos por nativos de Singapura

Trabalhos:

[1] Kaufmann e Kalita (2010) / [2] Contractor *et al.* (2010) / [3] Gadde *et al.* (2011) / [4] Sidarenka *et al.* (2013) / [5] Saloot *et al.* (2015)

Dentre os trabalhos realizados para a língua portuguesa, destaca-se o de Avanço *et al.* (2014) em que sugere a correção dos vocábulos errados presentes nos comentários realizados por usuários em sites de compra e venda. A Tabela 22 apresenta o resultado obtido pela aplicação da técnica de Avanço *et al.* (2014) em comparação com a utilização da ferramenta Aspell. Embora o trabalho consiste da normalização de textos de microblogs em português, a técnica se restringe a corrigir tokens individuais já considerados errados, não precisando assim executar a tarefa de detecção de erro, mas sim apenas de correção. Os três valores disponibilizados correspondem a porcentagem de (1) normalizações corretas, (2) normalizações erradas e (3) vocábulos errados sem sugestão de normalização.

Tabela 22 – Validação do trabalho de Avanço et al. *(2014)*

Técnica utilizada	**Normalização correta**	**Normalização errada**	**Sem sugestão**
Avanço *et al.* (2014)	65,46 %	34,16 %	0,38 %
Aspell	46,94 %	49,43 %	3,63 %

6.7.2 Valores obtidos na validação da arquitetura proposta

Para a execução do programa que implementa a arquitetura proposta fez-se necessário definir alguns valores de entrada para a programação genética, os quais estão descritos na Tabela 23. É importante notar que a função de avaliação é definida como 1-BLEU uma vez que o *framework* utilizado considera como melhor avaliado o indivíduo que apresentar o menor valor numérico.

Tabela 23 – Entradas externas da programação genética

Entrada	**Valor**
Método de Inicialização	*Ramped half-and-half*
Profundidade Inicial Máxima	*6*
Profundidade Máxima	*20*
Operação de Cruzamento	Cruzamento de sub-árvore
Operação de Mutação	Mutação de sub-árvore
Conjunto de Seleção	Formado por todos os componentes da geração
Operação de Seleção	*Tournament Selector*
Operação de Validação a cada geração	*1 – BLEU*
Máximo de Gerações	*60*
Critério de Parada	*Atingir máximo de gerações ou algum indivíduo atingir a melhor avaliação possível (0)*
Valores de Probabilidade de Ocorrência das Operações Genéticas	*Reprodução = 10%* *Cruzamento [inicial = 0 %, final = 90 %]* *Mutação [inicial = 90 %, final = 0 %]* *Os valores do cruzamento e da mutação vão sendo alterados linearmente a medida que as gerações passam*
Tamanho da base de treinamento	*250*
Tamanho da base de teste	*350*
Elitismo	*10*

6.7.2.1 Análise do caso base

Utilizando dos valores definidos na Tabela 23 executou-se cinco vezes o programa gerado para implementar todas as etapas da arquitetura, atingindo os valores presentes na Tabela 24. Analisando os resultados obtidos nota-se que a média do aumento da métrica BLEU atinge 41,45%, a qual é muito próxima o valor de 41,94% encontrado na literatura e, portanto, pode-se dizer que o caso base conseguiu atingir a média do estado da arte[18].

Além do valor médio de aumento, é importante destacar a variância e desvio padrão encontrados, o qual demonstra que a oscilação de qualidade entre as execuções é baixa, mostrando estabilidade e convergência do método aplicado com as entradas e parâmetros estipulados.

[18] O APÊNDICE C apresenta um exemplo de programa encontrado em uma das 5 instâncias rodadas para o caso base descrito na Tabela 24.

Tabela 24 – Validação do caso base

Métrica	Inst. 1	Inst. 2	Inst. 3	Inst. 4	Inst. 5	Média
BLEU inicial	0,57766	0,56691	0,56234	0,56370	0,56316	0,56675
BLEU final	0,80531	0,80709	0,78689	0,79617	0,81298	0,80169
Aumento (%)	39,41 %	42,37 %	39,93 %	41,24 %	44,36 %	41,45 %

Em relação ao aumento (%) das instâncias obteve-se:	
Variância	3,16598
Desvio Padrão	1,77932

Do mesmo modo como é oportuno que o resultado obtido a cada instância de execução do programa não varie muito, a diferença de desempenho entre a base de treinamento e de teste também merece análise. A relação entre os resultados obtidos entre as bases demonstra se para o problema a base de treinamento é representativa o suficiente para corresponder a novos casos não treinados (presentes na base de teste). A Tabela 25 apresenta os valores de BLEU e seu respectivo aumento para ambas as bases.

Tabela 25 – Validação do caso base (base de treinamento versus teste)

Métrica	Inst. 1	Inst. 2	Inst. 3	Inst. 4	Inst. 5	Média
BLEU inicial treinamento	0,55944	0,57448	0,58088	0,57899	0,57974	0,57471
BLEU final treinamento	0,80017	0,82957	0,81337	0,83020	0,84870	0,82440
Aumento treinamento (%)	43,03 %	44,40 %	40,02 %	43,39 %	46,39 %	43,45 %
BLEU inicial teste	0,57766	0,56691	0,56234	0,56370	0,56316	0,56675
BLEU final teste	0,80531	0,80709	0,78689	0,79617	0,81298	0,80169
Aumento teste (%)	39,41 %	42,37 %	39,93 %	41,24 %	44,36 %	41,45 %
Diferença de aumento	-3,62	-2,03	-0,09	-2,15	-2,03	-2,00

Em relação ao aumento (%) na base de treinamento das instâncias obteve-se:	
Variância	4,29818
Desvio Padrão	2,07321

Em relação ao aumento (%) na base de teste das instâncias obteve-se:	
Variância	3,16598
Desvio Padrão	1,77932

Em relação à diferença de aumento do BLEU das instâncias obteve-se:	
Variância	1,25910
Desvio Padrão	1,12210

No que diz respeito a variância e desvio padrão de ambas as bases (treinamento e teste), estas apresentam comportamento parecido, onde a base de teste chega, neste caso, até a apresentar comportamento melhor que a base de treinamento. Em relação ao aumento do BLEU, a base de testes tem na média um decréscimo de apenas 2% em relação a base de treinamento, que é um valor baixo levando em consideração que o aumento médio do BLEU é de 43,45% e 41,45% para as bases de treinamento e teste, e desvio padrão baixo próximo a 1, o que indica que para essa configuração a base de treinamento é significativa o suficiente para representação dos tipos de erro.

Para permitir a comparação de outros trabalhos com o resultado obtido no caso base, são disponibilizadas na Tabela 26 outras métricas de avaliação do resultado. As métricas presentes correspondem a: (1) BLEU 1p, análise da precisão de reconhecimento da correção utilizando unigramas; (2) BLEU 2p, análise da precisão de reconhecimento da correção utilizando bigramas; (3) BLEU 3p, análise da precisão de reconhecimento da correção utilizando trigramas; (4) BLEU 4p, análise da precisão de reconhecimento da correção utilizando 4-gramas; (5) Precisão de frases, porcentagem de frases corretas; (6) Tokens corretos, quantidade de tokens corretos na base; (7) Tokens errados, quantidade de tokens errados na base; e (8) Tokens faltantes, quantidade de tokens faltantes na base.

Tabela 26 – Validação do caso base (outras métricas)

Métrica	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
BLEU 1p inicial	0,8781	0,8783	0,8764	0,8707	0,8787	0,8764
BLEU 1p final	0,9573	0,9583	0,9550	0,9555	0,9639	0,9580
BLEU 1p aumento (%)	9,02 %	9,11 %	8,96 %	9,73 %	9,70 %	9,30 %
BLEU 2p inicial	0,7608	0,7584	0,7546	0,7484	0,7571	0,7558
BLEU 2p final	0,9082	0,9099	0,9005	0,9043	0,9169	0,9080
BLEU 2p aumento (%)	19,37 %	19,97 %	19,34 %	20,83 %	21,11 %	20,12 %
BLEU 3p inicial	0,6578	0,6514	0,6477	0,6446	0,6496	0,6502
BLEU 3p final	0,8537	0,8558	0,8420	0,8480	0,8638	0,8527
BLEU 3p aumento	29,79 %	31,37 %	29,99 %	31,56 %	32,99 %	31,14 %
BLEU 4p inicial	0,5777	0,5669	0,5623	0,5637	0,5632	0,5668
BLEU 4p final	0,8053	0,8071	0,7869	0,7962	0,8130	0,8017
BLEU 4p aumento	39,41 %	42,37 %	39,93 %	41,24 %	44,36 %	41,45 %
Precisão de frases inicial	0,2257	0,2114	0,1971	0,2200	0,2057	0,2120
Precisão de frases final	0,5629	0,5657	0,4971	0,5343	0,5714	0,5463
Prec. de frases aumento (%)	149,37 %	167,57 %	152,17 %	142,86 %	177,78 %	157,68 %

Tokens corretos inicial	1793	1893	1891	1824	1903	1861
Tokens corretos final	2145	2241	2227	2177	2276	2213
Tokens corretos aumento (%)	19,63 %	18,38 %	17,77 %	19,35 %	19,60 %	18,94 %
Tokens errados inicial	529	527	554	536	557	541
Tokens errados final	236	232	264	241	237	242
Tokens errados redução (%)	55,39 %	55,98 %	52,35 %	55,04 %	57,45 %	55,23 %
Tokens faltantes inicial	729	772	851	784	819	791
Tokens faltantes final	237	232	270	249	231	244
Tokens faltantes redução (%)	67,49 %	69,95 %	68,27 %	68,24 %	71,79 %	69,18 %

A Figura 36 apresenta a visualização da melhora da qualidade do melhor programa encontrado em cada uma das cinco instâncias de execução através das sessenta gerações, onde os valores correspondem a fase de treinamento contendo exclusivamente os tweets que não estiverem inicialmente em seu estado correto de grafia.

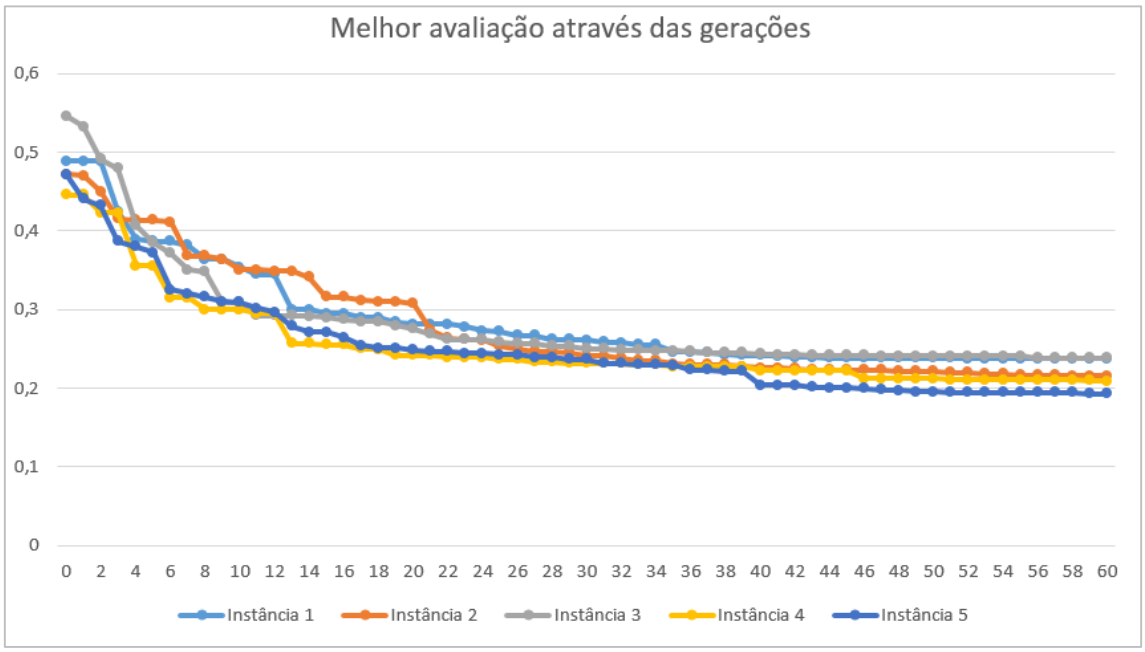

Figura 36 - Avaliação do melhor programa através das gerações

No intuito de realizar a comparação de aplicação do cenário desenvolvido utilizando a arquitetura proposta e outro sistema de correção, gerou-se a normalização da base de teste através da escolha da primeira opção de sugestão de correção para cada vocábulo indicada pelo Aspell, gerando assim frases normalizadas. Para que o Aspell não fique penalizado por não tratar tokens especiais dos tweets como as *mentions*, *hashtags* e URLs, estes três tipos de tokens são considerados automaticamente como

corretos (antes e depois da normalização), não precisando o Aspell reconhecê-los. O resultado obtido encontra-se na Tabela 27.

Tabela 27 – Validação do caso base utilizando o Aspell

Métrica	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
BLEU inicial	0,7409	0,7055	0,7508	0,7225	0,7115	0,7262
BLEU final	0,7246	0,6998	0,7308	0,7009	0,6998	0,7112
BLEU aumento (%)	-2,20 %	-0,81 %	-2,66 %	-2,99 %	-1,64 %	-2,07 %

Ao analisar os valores obtidos nota-se que em todos os casos a normalização do texto utilizando o Aspell piorou, mostrando que a arquitetura proposta por meio da utilização de regras especificas do contexto consegue resultados extremamente superiores a aplicação da correção com uma ferramenta que realiza a sugestão de correção para textos em linguagem culta.

Tabela 28 – Validação do caso base com substituição dos vocábulos do projeto Brazilis

Métrica	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
BLEU inicial	0,7431	0,7298	0,7395	0,7628	0,7379	0,7426
BLEU final	0,7761	0,7794	0,7773	0,8005	0,7744	0,7815
BLEU aumento (%)	4,44 %	6,80 %	5,11 %	4,94 %	4,94 %	5,25 %

Por fim, utilizando da mesma base de tweets anotadas para o português, realizou-se a substituição das palavras em internetês que estão respectivamente mapeadas com sua forma correta de escrita, disponibilizada pelo projeto Brazilis[19] do grupo de estudo Pros@[20]. Utilizando desta abordagem mais ingênua, tem-se na média um aumento de 5,25% da métrica BLEU, enquanto que o encontrado neste trabalho, com o uso da arquitetura proposta e as entradas externas desenvolvidas, totalizou na média 41,45% de aumento.

6.7.2.2 Variação de parâmetros

Com intuito de analisar a representatividade do tamanho da base de treinamento, é variada sua dimensão e analisados os resultados (presente na Tabela 29). Se analisada a média do aumento do BLEU pode-se perceber um crescimento (não linear) à medida

[19] http://143.107.183.175:12580/semanticnlp/index.php?id=index&id_sub=principal&dir_sub=includes/projects/brazilis&dir=includes/projects/brazilis&lang=pt-br

[20] http://nilc.icmc.usp.br/semanticnlp/index.php?id=principal&dir=includes&lang=pt-br

que a base de treinamento também aumenta, enquanto que o desvio padrão e a variância (em relação as instâncias de programas rodados para cada tipo de configuração) diminuem. Nota-se assim que além do tamanho da base de treinamento influenciar a sua representatividade em relação a base teste, este também aumenta a estabilidade de convergência para o próprio conjunto de treinamento.

Tabela 29 – Variação do caso base diversificando o tamanho da base de treinamento

Métrica	**T. base**	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
BLEU inicial	150	0,56508	0,55428	0,57511	0,56562	0,56166	0,56435
BLEU final	150	0,79993	0,77574	0,74526	0,77528	0,71948	0,76314
Aumento (%)	150	41,56 %	39,96 %	29,58 %	37,07 %	28,10 %	35,22 %
BLEU inicial	200	0,55686	0,57255	0,57710	0,55798	0,58754	0,57528
BLEU final	200	0,79290	0,77438	0,80332	0,79203	0,81809	0,78361
Aumento (%)	200	42,39 %	35,25 %	39,20 %	41,95 %	39,24 %	36,21 %
BLEU inicial	250	0,57766	0,56691	0,56234	0,56370	0,56316	0,56675
BLEU final	250	0,80531	0,80709	0,78689	0,79617	0,81298	0,80169
Aumento (%)	250	39,41 %	42,37 %	39,93 %	41,24 %	44,36 %	41,45 %

Em relação ao aumento (%) das instâncias obteve-se:

Métrica \ Tamanho da base de treinamento	150	200	250
Variância	29,71678	6,50370	3,16598
Desvio Padrão	5,45131	2,55024	1,77932

Para avaliar o uso das operações genéticas, variou-se a função onde define os valores da probabilidade de aplicação das operações em cada uma das gerações. Os resultados encontrados são mostrados na Tabela 30. Constatou-se assim que o melhor resultado obtido é adquirido quando utilizada a função linear de operação genética (com 41,45% de aumento do BLEU), enquanto o pior resultado quando é aplicada a função de cruzamento alto.

Tabela 30 – Variação do caso base diferenciando a função de aplicação das operações genéticas

Métrica	**Op. Genética**	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
BLEU inicial	Linear	0,57766	0,56691	0,56234	0,56370	0,56316	0,56675
BLEU final	Linear	0,80531	0,80709	0,78689	0,79617	0,81298	0,80169
Aumento (%)	Linear	39,41 %	42,37 %	39,93 %	41,24 %	44,36 %	41,45 %
BLEU inicial	Linear Invertida	0,57721	0,57823	0,57555	0,58886	0,55419	0,57481
BLEU final	Linear Invertida	0,75183	0,81466	0,79740	0,83075	0,79970	0,79887

Aumento (%)	Linear Invertida	30,25 %	40,89 %	38,55 %	41,08 %	44,30 %	38,98 %
BLEU inicial	Cruzamento Alto	0,57085	0,57622	0,55853	0,56618	0,56347	0,56705
BLEU final	Cruzamento Alto	0,81941	0,71500	0,73615	0,78913	0,78784	0,76951
Aumento (%)	Cruzamento Alto	43,54 %	24,08 %	31,80 %	39,38 %	39,82 %	35,70 %
BLEU inicial	Mutação Alta	0,55577	0,54650	0,55435	0,56826	0,57067	0,55911
BLEU final	Mutação Alta	0,80188	0,78995	0,76609	0,78254	0,78507	0,78511
Aumento (%)	Mutação Alta	44,28 %	44,55 %	38,20 %	37,71 %	37,57 %	40,42 %

Operações genéticas:

Linear – Reprodução: 10% / Cruzamento [inicial:0%, final: 90%] / Mutação [inicial: 90%, final: 0%] / *Os valores do cruzamento e da mutação vão sendo alterados linearmente a medida que as gerações passam*

Linear Invertido - Reprodução: 10% / Cruzamento [inicial:90%, final: 0%] / Mutação [inicial: 0%, final: 90%] / *Os valores do cruzamento e da mutação vão sendo alterados linearmente a medida que as gerações passam*

Cruzamento Alto – Reprodução: 10% / Cruzamento: 70% / Mutação: 20%

Mutação Alta – Reprodução: 10% / Cruzamento: 20% / Mutação: 70%

6.8 Velocidade de execução

A velocidade de execução consiste de um fator importante a ser analisado, uma vez que existem programas em que a qualidade do resultado é invalidada pelo seu tempo necessário de execução, por exemplo, um programa que prevê a variação da bolsa do dia seguinte baseado nos dados no dia atual não pode levar dois dias processando, já que quando obter o resultado este já não é mais útil.

Sendo assim foram medidas as velocidades de execução das duas etapas presentes: (1) tempo de execução do programa processado para simular as etapas da arquitetura e encontrar um programa adequado para normalização dos tweets e (2) velocidade média de normalização por tweet realizado pelo programa eleito como mais qualificado para a tarefa de normalização. A Tabela 31 apresenta os respectivos tempos de processamento para as cinco vezes em que o caso base foi rodado em um super computador com 40 gigas de memória alocada e 1 nó com 8 cores de processamento.

Tabela 31 – Tempo de processamento

Tempo	**Inst. 1**	**Inst. 2**	**Inst. 3**	**Inst. 4**	**Inst. 5**	**Média**
Execução da parte 1	3h e 42 min	3h e 23 min	3h e 4 min	3h e 31 min	3h e 55 min	3h e 31 min
Execução da parte 2	6,59 ms	6,13 ms	5,10 ms	5,98 ms	6,88 ms	6,14 ms

Analisando os resultados têm-se que os tempos de processamentos são factíveis, uma vez que a primeira parte como só deve ser rodada uma vez, para a geração do programa corretor, permite com que sua execução seja mais demorada (levando em média três horas e trinta e um minutos). Enquanto que o tempo médio gasto para se normalizar um tweet, que necessita ser baixo, é de apenas 6,14 milissegundos.

6.9 Considerações finais

Ao se aplicar um cenário de normalização de texto na arquitetura proposta, mostrou-se que esta é capaz de atingir os valores de correção presentes no estado da arte na literatura. Contudo, a qualidade da aplicação da arquitetura não é garantida uma vez que seu resultado depende das entradas externas fornecidas pelo usuário e a validação foi executada apenas na aplicação de um único cenário teste. Todavia, com os resultados apresentados, a proposta parece consistir de uma possibilidade real de alternativa de aplicação para problemas de normalização de texto.

7 Conclusão

7.1 Revisão do problema e da proposta

O trabalho consistiu no desenvolvimento de um novo formato de solução, em comparação com as técnicas existentes na literatura, para criação de programas normalizadores de texto. A correção dos textos em diversos casos se torna necessária devido à necessidade deste ser apresentado com clareza e coerência, tal que assim possam mostrar sua integridade e importância. A possibilidade de aplicação de um método de processamento de linguagem natural em cima de textos também exige que estes estejam bem redigidos, seguindo as regras estabelecidas pela norma culta da língua, tal que possam apresentar melhores resultados.

A solução alternativa à literatura proposta neste trabalho consiste na aplicação de uma arquitetura híbrida da utilização de uma técnica com aprendizado de máquina e do uso de uma solução baseada em regras. Estas, combinadas, tentam tirar proveito dos maiores benefícios presentes em cada lado, onde se destaca a origem da fonte do conhecimento a qual o programa utiliza durante a realização da normalização, advindos do aprendizado de máquina e das regras.

A arquitetura desenvolvida consiste de uma construção de um modelo genérico, o qual não faz suposições sobre as regras de escrita e nem restringe a aplicação a um conjunto de idiomas. Todo o conhecimento necessário que delimitam uma língua e um contexto de aplicação são passados através de entradas externas, mantendo assim a arquitetura apta a receber qualquer tipo de problema que se adeque às entradas exigidas.

Uma vez que os tratamentos que a arquitetura faz consistem na execução de módulos pré-definidos, caso sejam encontrados determinados procedimentos faltantes basta que sejam desenvolvidos novos módulos que os atendam, podendo estes apresentarem um ou mais novos tipos de função a serem acopladas ou utilizar das funções já existentes.

7.2 Contribuições

Este trabalho propiciou a elaboração de diversas contribuições através da análise da área de atuação, construção da arquitetura e dos componentes que esta necessita.

A primeira contribuição realizada consiste da criação de um agrupamento dos tipos de erros de escrita com alto nível de detalhes (apresentado em 2.3) em relação aos demais agrupamentos existentes, que foi criado a partir da análise das questões a respeito da grafia das palavras discutidos nos trabalhos presentes da literatura.

Uma vez escolhido que a arquitetura se basearia na utilização da programação genética para criação de programas corretores de textos, desenvolveu-se uma técnica de representação e execução de um programa normalizador de textos modelado em uma árvore sintática.

A criação de um novo método de solução do problema de normalização de textos consiste de outra contribuição, a qual pode ser utilizada assim como as técnicas de redes neurais, cadeia de Markov e outras presentes na literatura.

Uma vez que esta foi validada em cima do contexto de normalização de tweets em português, foi necessário o desenvolvimento de uma base de tweets anotadas com suas respectivas correções, a qual está disponibilizada para uso de demais trabalhos que a necessitem. Juntamente foi desenvolvido (e disponibilizado) um novo léxico para o português, o qual conta com exclusivamente palavras da norma culta e sua respectiva frequência de aparecimento, a qual está atrelada ao aparecimento das palavras no conjunto dos livros de histórias utilizados para montagem do léxico.

A partir da validação do cenário proposto, gerou-se os valores em diversas métricas do problema de normalização, a qual pode ser usada como base de comparação de novos trabalhos, que caso optem podem fazer uso da mesma base de tweets anotadas e léxico de consulta.

7.3 Trabalhos futuros

Dentre os próximos trabalhos a se realizar em cima do contexto explorado nesta pesquisa, têm-se a análise do desenvolvimento de novos módulos, os quais podem ser acoplados aos demais ou então substituir um existente por sua evolução. Os novos módulos podem tanto utilizar dos tipos de funções já elaborados, como também criar um novo grupo que tenha uma nova finalidade de existência.

Para reforçar a viabilidade de aplicação da proposta, novos casos de exemplo devem ser desenvolvidos tais que esses explorem a qualidade da aplicação da proposta e por ventura se reconheça melhorias a serem desenvolvidas.

Dentre os problemas já detectados na aplicação do caso de exemplo, destaca-se a necessidade da pesquisa de como realizar a otimização das árvores, ou seja, retirar de forma proativa nós que não estão interferindo no processamento mas estão presentes na árvore, muitas vezes representando a execução de regras que não atingem os tokens que por ali passam. A otimização das árvores leva a criação de programas mais eficientes,

além de intuitivamente parecer levar a evolução mais rápida e qualificada do programa melhor adaptado ao problema.

Na tentativa de se encontrar o melhor programa normalizador de textos para um determinado cenário, estudos devem ser realizados a respeito da variação da probabilidade de aplicação de cada uma das operações genéticas, em que o teste em cima de diferentes cenários sob mais conjunturas pode identificar melhores opções de aplicação da arquitetura. Além disso, deve-se analisar a existência de uma função de avaliação melhor a ser aplicada em cada geração do que o BLEU, tal que esta leve em consideração mais características da árvore além de seu resultado final, podendo assim beneficiar mais a um indivíduo da população do que outro, mesmo que ambos produzam o mesmo resultado, uma vez que um dos indivíduos pode estar muito mais próximo da evolução do resultado do que o outro.

No que diz respeito ao programa desenvolvido para simular a arquitetura proposta, deve-se realizar a sua otimização, fazendo este tirar maior proveito do processamento executado em cima das árvores através das gerações, o qual valores encontrados nas etapas intermediárias do processamento podem ser reutilizados, economizando assim tempo de processamento. Visando disponibilizar a implementação para uso geral, é necessário também aprimorar a modularização dos componentes, de forma a tornar seu uso mais simples e menos sujeito a erros de codificação dos usuários.

Na tentativa de aplicar a arquitetura proposta em outros domínios, pode-se primeiramente avaliar seu desempenho no problema de tradução automática, uma vez que os resultados obtidos no capítulo anterior foram calculados através da aplicação da métrica BLEU, a qual também é majoritariamente utilizada no cenário de tradução automática de textos e o problema se adequa as entradas externas da arquitetura.

Para conseguir melhores resultados no problema de normalização de tweets em português, realizar um estudo de novas funções a serem implementadas, tal que possam agregar conhecimento útil para a realização das correções, além de melhorar a qualidade do léxico desenvolvido, tal que este contenha um número maior de vocábulos e também conte com a presença de n-gramas para poderem auxiliar no processo de escolha do melhor token para correção.

Referências Bibliográficas

AGARWAL, S., GODBOLE, S., PUNJANI, D., et al., 2007. "How much noise is too much: A study in automatic text classification", *Seventh IEEE International Conference on Data Mining, 2007, ICDM 2007*, Omaha, NE: IEEE, 2007, pp. 3–12.

ALEGRIA, I., ARANBERRI, N., COMAS UMBERT, P.R., et al., 2014. "TweetNorm_es: an annotated corpus for Spanish microtext normalization", *Proceedings of the Ninth International Conference on Language Resources and Evaluation*, Reykjavik-Paris, European Language Resources Association-ELRA, 2014, pp. 2274–2278.

ANDRADE, G.N., TEIXEIRA, F., XAVIER, C., et al., 2012. "HASCH: Um Corretor Ortográfico Automático de Alto Desempenho para Textos Oriundos da Web", *Revista Eletrônica de Iniciação Científica*, v. 12, n. 3, 2012.

ASONOV, D., 2010. "Real-Word Typo Detection". In: HORACEK, Helmut, MÉTAIS, Elisabeth, MUÑOZ, Rafael & WOLSKA, Magdalena (eds.), *Natural Language Processing and Information Systems*. Springer Berlin Heidelberg. Lecture Notes in Computer Science, 5723, pp. 115–129.

ASPELL.NET. c2004. *Spell Checker Test Kernel Results*. Disponível em: <http://aspell.net/test/cur/>. Acesso em: 04 Dezembro 2015.

AVANÇO, L.V., DURAN, M.S., NUNES, M. DAS G.V., 2014. "Towards a Phonetic Brazilian Portuguese Spell Checker", *PROPOR 2014 - I Workshop on Tools and Resources for Automatically Processing Portuguese and Spanish*, Springer, 2014, pp. 24–31.

BAHRAINIAN, S.-A., DENGEL, A., 2013. "Sentiment Analysis and Summarization of Twitter Data", *2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)*, Dezembro 2013, pp. 227–234.

BANDYOPADHYAY, A., ROY, D., MITRA, M., et al., 2014. "Named Entity Recognition from Tweets⋆", *Proceedings of the 16th LWA Workshops: KDML, IR and FGWM*, 2014, pp. 218–225.

BISOGNIN, T.R., 2008. *Do internetês ao léxico da escrita dos jovens no Orkut*. Dissertação de Mestrado (Mestrado em Letras), Instituto de Letras, Programa de Pós-Graduação em Letras, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, Porto Alegre, Brasil.

BONTCHEVA, K., DERCZYNSKI, L., FUNK, A., et al., 2013. "TwitIE: An Open-Source Information Extraction Pipeline for Microblog Text.", *RANLP*, 2013, pp. 83–90.

BRILL, E., MOORE, R.C., 2000. "An improved error model for noisy channel spelling correction". *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, 2000, pp. 286–293.

BROWN, P.F., PIETRA, V.J.D., PIETRA, S.A.D., et al., 1993. "The mathematics of statistical machine translation: Parameter estimation", *Computational linguistics*, v. 19, 1993, MIT Press, Cambridge, MA, USA, pp. 263–311.

CHOUDHURY, M., SARAF, R., JAIN, V., et al., 2007. "Investigation and modeling of the structure of texting language", *International Journal of Document Analysis and Recognition (IJDAR)*, v. 10, 2007, Springer Verlag, pp. 157–174.

CHURCH, K.W., GALE, W.A., 1991. "Probability scoring for spelling correction", *Statistics and Computing*, v. 1, 1991, Kluwer Academic Publishers, pp. 93–103.

CLARK, A., 2003. "Pre-processing very noisy text", *Proc. of Workshop on Shallow Processing of Large Corpora*, Corpus Linguistic, Lancaster, 2003, pp. 12–22.

CLARK, E., ARAKI, K., 2011. "Text normalization in social media: progress, problems and applications for a pre-processing system of casual English", *Procedia-Social and Behavioral Sciences*, v. 27, 2011, Elsevier, pp. 2–11.

COETSEE, D., 2014. *Conditional random fields for noisy text normalisation*, Dissertação de Mestrado (Mestrado em Engenharia Elétrica e Eletrônica), Departamento de Engenharia Elétrica e Eletrônica, Faculdade de Engenharia em Stellenbosch, Universidade de Stellenbosch, Stellenbosch, África do Sul.

CONTRACTOR, D., FARUQUIE, T.A., SUBRAMANIAM, L.V., 2010. "Unsupervised cleansing of noisy text", *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, Association for Computational Linguistics, 2010, pp. 189–196.

CORNEY, D., MARTIN, C., GÖKER, A., 2014. "Spot the ball: Detecting sports events on Twitter", *Advances in Information Retrieval*, Springer, 2014, pp. 449–454.

COVINGTON, M.A., 1996. "An algorithm to align words for historical comparison", *Computational linguistics*, v. 22, 1996, MIT Press, Cambridge, MA, USA, pp. 481–496.

CUCERZAN, S., BRILL, E., 2004. "Spelling Correction as an Iterative Process that Exploits the Collective Knowledge of Web Users.", *Proceedings of EMNLP*, 2004, pp. 293–300.

CUI, A., ZHANG, M., LIU, Y., et al., 2011. "Emotion tokens: Bridging the gap among multilingual twitter sentiment analysis", *Information retrieval technology*, Springer, 2011, pp. 238–249.

DEEPAK, P., SUBRAMANIAM, V., 2012. "Correcting SMS Text Automatically", *Cover Story of the CSI Communications*, 2012, pp. 9–11.

DUARTE, E.S., 2013. *Sentiment analysis on twitter for the portuguese language*, Dissertação de Mestrado (Mestrado em Engenharia Informática), Faculdade de Ciências e Tecnologia, Universidade Nova Lisboa, Lisboa, Portugal.

DURAN, M.S., AVANÇO, L., NUNES, M.G.V., 2015. "A Normalizer for UGC in Brazilian Portuguese", *Proceedings of ACL-IJCNLP 2015*, pp. 38.

ELLEN, J., 2011. "Contrasting Machine Learning Approaches for Microtext Classification", *Proceedings of The 2011 World Congress in Computer Science, Computer Engineering and Applied Computing*, Las Vegas, Nevada, USA, Julho 18-21, 2011.

EPOCHX. c2012. Disponível em: <http://www.epochx.org/guide-algorithm.php>. Acesso em 05 Novembro 2015.

FERMAN, F., GARRIDO, L.B., DA SILVA, T.S., et al., 2015. "Método não supervisionado para monitoramento de assuntos de governo nos países de língua portuguesa", *Proceedings of the IV Brazilian Workshop on Social Network Analysis and Mining*, Recife, PE, Brasil, 2015.

G1-SP, 2015. G1. Disponível em: <http://g1.globo.com/tecnologia/noticia/2015/01/whatsapp-atinge-os-700-milhoes-de-usuarios-por-mes-em-todo-o-mundo.html>. Acesso em: 15 Outubro 2015.

GADDE, P., GOUTAM, R., SHAH, R., et al., 2011. "Experiments with artificially generated noise for cleansing noisy text", *Proceedings of the 2011 joint workshop on multilingual OCR and analytics for noisy unstructured text data*, ACM, 2011, pp. 4.

GARAAS, T., XIAO, M., POMPLUN, M., 2012. *Personalized Spell Checking using Neural Networks*. Disponível em: <http://www.cs.umb.edu/~marc/pubs/garaas_xiao_pomplun_HCII2007.pdf>. Acesso em: 5 Janeiro 2016.

GONDIM, F., 2006. *Algoritmo de Comparação de Strings para Integração de Esquemas de Dados*, Trabalho de Conclusão de Curso (Graduação), Graduação em Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, Brasil.

GONZALEZ, Z.M.G., 2007. *Linguística de corpus na análise do internetês*. Dissertação de Mestrado (Mestrado em Linguística Aplicada e Estudos da Linguagem), Pontifícia Universidade Católica de São Paulo, São Paulo, Brasil, 123 p..

HALL, P.A., DOWLING, G.R., 1980, "Approximate string matching", *ACM computing surveys (CSUR)*, v. 12, 1980, pp. 381–402.

HAN, B., BALDWIN, T., 2011. "Lexical Normalisation of Short Text Messages: Makn Sens a #Twitter", *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2011, pp. 368–378.

HARTMANN, N.S., AVANÇO, L.V., BALAGE, P.P., et al., 2014. "A Large Corpus of Product Reviews in Portuguese: Tackling Out-Of-Vocabulary Words", *International Conference on Language Resources and Evaluation, 9th.*, European Language Resources Association-ELRA, 2014.

HONG, L., CONVERTINO, G., CHI, E.H., 2011. "Language Matters In Twitter: A Large Scale Study", *Proceedings of 5th International AAAI Conference on Weblogs and Social Media ICWSM*, Barcelona, Espanha, Julho 17-21, 2011.

HUANG, Y., MURPHEY, Y.L., GE, Y., 2013. "Automotive diagnosis typo correction using domain knowledge and machine learning", *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, Abril 2013, pp. 267–274.

HUNSPELL. c2014. Disponível em: <http://hunspell.sourceforge.net/>. Acesso em: 05 Novembro 2015.

HU, R., 2013. *Lexical Normalisation of Tweeter Data*. Disponível em: <http://hurui.info/files/Lexical%20Normalisation%20of%20Tweeter%20Data.pdf>. Acesso em: 05 Janeiro 2015.

KAUFMANN, M., KALITA, J., 2010. "Syntactic normalization of twitter messages", *Proceedings of the 8th International conference on natural language processing ICON*, Kharagpur, India, Dezembro 8-11, 2010.

KINOSHITA, J., SALVADOR, L. DO N., MENEZES, C.E.D., 2005. "CoGrOO–Um Corretor Gramatical para a língua portuguesa, acoplável ao OpenOffice", *Anais do XXXI Conferência Latino Americana de Informática CLEI*, 2005.

KNIGHT, K., AL-ONAIZAN, Y., 1998. "Translation with finite-state devices", *Proceedings of the Third Conference of the Association for Machine Translation in the Americas on Machine Translation and the Information Soup*, Springer-Verlag, Londres, UK, 1998, pp. 421–437.

KOEHN, P., OCH, F.J., MARCU, D., 2003. "Statistical phrase-based translation", *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, Association for Computational Linguistics, 2003, pp. 48–54.

KOZA, J.R., 1990. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

KUKICH, K., 1992. "Techniques for automatically correcting words in text", *ACM Computing Surveys (CSUR)*, v. 24, 1992, pp. 377–439.

LEE, K.H., NG, M.K.M., LU, Q., 1999. "Text segmentation for Chinese spell checking", *Journal of the American Society for Information Science*, v. 50, 1999, pp. 751–759.

LEVENSTEIN, V., 1965. "Binary codes capable of correcting spurious insertions and deletions of ones", *Problems of Information Transmission*, v. 1, 1965, pp. 8–17.

LEWIS, PAUL, M., SIMONS, G.F., et al., 2015. Ethnologue: Languages of the World, Eighteenth edition. Dallas, Texas: SIL International. Online version: <http://www.ethnologue.com>. Acesso em: 17 Dezembro 2015.

LIU, X., ZHOU, M., WEI, F., et al., 2012. "Joint inference of named entity recognition and normalization for tweets", *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, Association for Computational Linguistics, 2012, pp. 526–535.

MACEDO, P.C., DE MORAES, M.R., 2010. "Uso de tabela de dispersão na correção de grafia incorreta: estudo com aplicação de algoritmos", *Universitas*, ano 3, n. 5, Agosto/Dezembro, 2010, pp. 29-46.

MAEDA, H., SHIMADA, K., ENDO, T., 2012. "Twitter Sentiment Analysis Based on Writing Style". In: ISAHARA, Hitoshi & KANZAKI, Kyoko (eds.), *Advances in Natural Language Processing*. Springer Berlin Heidelberg. Lecture Notes in Computer Science, 7614, pp. 278–288.

MARTINS, R.T., HASEGAWA, R., NUNES, M. DAS G.V., 2002. "Curupira: um parser funcional para a língua portuguesa", *Relatório Técnico*, 181, São Carlos: NILC, 2002.

MARTINS, T.B., GHIRALDELO, C.M., NUNES, M. DAS G.V., et al., 1996. *Readability formulas applied to textbooks in brazilian portuguese*, Notas do ICMSC-USP, Série Computação, n. 28, 1996, 11p..

MISHRA, R., KAUR, N., 2013. "A Survey of Spelling Error Detection and Correction Techniques", *International Journal of Computer Trends and Technology*, v. 4, 2013, pp. 372–374.

NAGATA, M., 1996. "Context-based spelling correction for Japanese OCR", *Proceedings of the 16th conference on Computational linguistics-Volume 2*, Association for Computational Linguistics, 1996, pp. 806–811.

NEBHI, K., BONTCHEVA, K., GORRELL, G., 2015. "ResToRinG CaPitaLiZaTion in# TweeTs", *Proceedings of the 24th International Conference on World Wide Web Companion*, International World Wide Web Conferences Steering Committee, 2015, pp. 1111–1115.

NUNES, M. DAS G.V., OLIVEIRA JR, O.N., 2000. "O processo de desenvolvimento do Revisor Gramatical ReGra", *Anais do XXVII SEMISH (XX Congresso Nacional da Sociedade Brasileira de Computação)*, 2000, pp. 6.

PAPINENI, K., ROUKOS, S., WARD, T., et al., 2002. "BLEU: a method for automatic evaluation of machine translation", *Proceedings of the 40th annual meeting on*

association for computational linguistics, Association for Computational Linguistics, 2002, pp. 311–318.

PENNELL, D., LIU, Y., 2011. "Toward text message normalization: Modeling abbreviation generation", *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Maio 2011, pp. 5364–5367.

PENNELL, D.L., LIU, Y., 2010. "Normalization of text messages for text-to-speech", *Proceedings of the 2010 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, IEEE, 2010, pp. 4842–4845.

PHILIPS, L., 2000. "The double metaphone search algorithm", *C/C++ users journal*, v. 18, 6, Junho 2000, pp. 38–43.

POLI, R., LANGDON, W.B., MCPHEE, N.F., et al., 2008. *A field guide to genetic programming*. Lulu.com.

POZO, A., CAVALHEIRO, A.F., ISHIDA, C., et al., 2005. *Computação evolutiva*, Universidade Federal do Paraná. Departamento de Informática. Apostila 61 p. Google Keyword: computação evolutiva. Disponível em: <http://www.inf.ufpr.br/~aurora/tutoriais/Ceapostila.pdf>. Acesso em: 05 Janeiro 2016.

REGRA. c1993. Disponível em: <http://www.nilc.icmc.usp.br/nilc/projects/regra.htm>. Acesso em: 14 Outubro 2015.

RINO, L.H.M., DI FELIPPO, A., PINHEIRO, G.M., et al., 2002. "Aspectos da Construção de um Revisor Gramatical Automático para o Português", *Estudos Linguísticos*, v. 31, maio 2002, pp. 1–6.

RISTAD, E.S., YIANILOS, P.N., 1998. "Learning string-edit distance", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 20, 1998, pp. 522–532.

SALOOT, M.A., IDRIS, N., MAHMUD, R., 2014. "An architecture for Malay Tweet normalization", *Information Processing & Management*, v. 50, 2014, pp. 621–633.

SALOOT, M.A., IDRIS, N., SHUIB, L., et al., 2015. "Toward Tweets Normalization Using Maximum Entropy", *Proceedings of the ACL-IJCNLP 2015*, Beijing, China, Julho 26-31, 2015, pp. 19-27.

SANKOFF, D., KRUSKAL, J.B., 1983. "Time warps, string edits, and macromolecules: the theory and practice of sequence comparison". In: *Reading: Addison-Wesley Publication*, 1983, edited by Sankoff, David; Kruskal, Joseph B., v. 1.

SEABRA, A.L., NUNES, G.M., CARVALHO, M.W.V. DE, 2011. "Análise de Erros Ortográficos em Redações", *E-Hum - Revista Científica das Áreas de Humanidades do Centro Universitário de Belo Horizonte*, v. 4, n. 2, 2011, pp. 87-101.

SHAALAN, K., ALLAM, A., GOMAH, A., 2003. "Towards automatic spell checking for Arabic", *Conference on Language Engineering, ELSE*, Cairo, Egito, 2003.

SHANNON, C.E., 2001. "A mathematical theory of communication", *ACM SIGMOBILE Mobile Computing and Communications Review*, v. 5, 2001, pp. 3–55.

SIDARENKA, U., SCHEFFLER, T., STEDE, M., 2013. "Rule-based normalization of German Twitter messages", *Proc. of the GSCL Workshop Verarbeitung und Annotation von Sprachdaten aus Genres internetbasierter Kommunikation*, 2013.

SILVA, F., SILVA, T., LOUREIRO, A., et al., 2010. "Internal Contexts Inference System for Ubiquitous Context-aware Applications", *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, Nova Iorque, NY, EUA: ACM, 2010, pp. 776–779.

SMITH, T.F., WATERMAN, M.S., 1981. "Identification of common molecular subsequences", *Journal of Molecular Biology*, v. 147, 1981, pp. 195–197.

SULTANIK, E.A., FINK, C., 2012. "Rapid geotagging and disambiguation of social media text via an indexed gazetteer", *Proceedings of ISCRAM*, v. 12, 2012, pp. 1–10.

TOUTANOVA, K., MOORE, R.C., 2002. "Pronunciation modeling for improved spelling correction", *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, 2002, pp. 144–151.

WANG, X., TOKARCHUK, L., POSLAD, S., 2014. "Identifying relevant event content for real-time event detection", *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, IEEE, 2014, pp. 395–398.

WINKLER, W.E., 1999. *The state of record linkage and current research problems*. Disponível em: <https://www.census.gov/srd/papers/pdf/rr99-04.pdf>. Acessado em: 05 Janeiro 2016.

WINTNER, S., 2004. "Hebrew computational linguistics: Past and future", *Artificial Intelligence Review*, v. 21, 2004, pp. 113–138.

ZORZI, J.L., CIASCA, S.M., 2009, "Análise de erros ortográficos em diferentes problemas de aprendizagem", *Revista CEFAC*, v. 11, 2009, pp. 406–416.

APÊNDICE A – Categorização com alto nível de detalhes conforme a literatura

Abaixo constam todas as categorias descritas na Tabela 7 seguidas pelos trabalhos que colocaram de forma explícita a categoria ou a indicaram descrevendo o erro:

- Necessitam de identificação:
  - *Mentions*: (BAHRAINIAN & DENGEL, 2013), (WANG *et al.*, 2014), (BONTCHEVA *et al.*, 2013) e (KAUFMANN & KALITA, 2010);
  - *Hashtags*: (BAHRAINIAN & DENGEL, 2013), (WANG *et al.*, 2014), (BONTCHEVA *et al.*, 2013) e (KAUFMANN & KALITA, 2010);
  - URLs: (BAHRAINIAN & DENGEL, 2013), (WANG *et al.*, 2014), (BONTCHEVA *et al.*, 2013) e (KAUFMANN & KALITA, 2010);
  - *Emoticons*: (GADDE *et al.*, 2011), (CUI *et al.*, 2011), (BONTCHEVA *et al.*, 2013), (KAUFMANN & KALITA, 2010), (CLARK & ARAKI, 2011) e (HAN & BALDWIN, 2011);
  - Onomatopeias: (ALEGRIA *et al.*, 2014);
- Capitalização: (NEBHI *et al.*, 2015), (SULTANIK & FINK, 2012), (GADDE *et al.*, 2011), (BONTCHEVA *et al.*, 2013), (KAUFMANN & KALITA, 2010), (HARTMANN *et al.*, 2014) e (CLARK, 2003);
- Repetição:
  - Caracteres: (WANG *et al.*, 2014), (CUI *et al.*, 2011), (KAUFMANN & KALITA, 2010) e (HAN & BALDWIN, 2011);
  - Pontuação: (CUI *et al.*, 2011) e (KAUFMANN & KALITA, 2010);
- Abreviação:
  - Acrônimo reconhecido: (BAHRAINIAN & DENGEL, 2013), (GADDE *et al.*, 2011), (CUI *et al.*, 2011), (BONTCHEVA *et al.*,

2013), (KAUFMANN & KALITA, 2010), (CLARK & ARAKI, 2011), (CONTRACTOR *et al.*, 2010) e (DEEPAK & SUBRAMANIAM, 2012);

    - Truncamento: (BONTCHEVA *et al.*, 2013), (HUANG *et al.*, 2013) e (HARTMANN *et al.*, 2014);
    - Acrônimo não reconhecido: (PENNELL & LIU, 2011), (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (HAN & BALDWIN, 2011), (CONTRACTOR *et al.*, 2010), (DEEPAK & SUBRAMANIAM, 2012), (HARTMANN *et al.*, 2014), (SALOOT *et al.*, 2014) e (CHOUDHURY *et al.*, 2007);
- Junção: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (HUANG *et al.*, 2013) e (CLARK, 2003);
- Separação: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (HUANG *et al.*, 2013) e (CLARK, 2003);
- Omissão de pontuação:
    - Separação de sentença: (KAUFMANN & KALITA, 2010) e (CLARK, 2003);
    - Composição de palavra: (CLARK & ARAKI, 2011) e (CLARK, 2003);
- Hífen:
    - Inserção: (CLARK, 2003);
    - Omissão: (CLARK, 2003);
- Acentuação:
    - Inserção: (ANDRADE *et al.*, 2012) e (HARTMANN *et al.*, 2014);
    - Omissão: (ANDRADE *et al.*, 2012) e (HARTMANN *et al.*, 2014);
    - Substituição: (ANDRADE *et al.*, 2012) e (HARTMANN *et al.*, 2014);
- Representação fonética:

    - Letra: (SULTANIK & FINK, 2012), (GADDE *et al.*, 2011), (KAUFMANN & KALITA, 2010), (HAN & BALDWIN, 2011), (CONTRACTOR *et al.*, 2010), (SALOOT *et al.*, 2014), (CHOUDHURY *et al.*, 2007) e (TOUTANOVA & MOORE, 2002);
    - Número: (PENNELL & LIU, 2011), (GADDE *et al.*, 2011), (HAN & BALDWIN, 2011), (CONTRACTOR *et al.*, 2010), (DEEPAK & SUBRAMANIAM, 2012) e (CHOUDHURY *et al.*, 2007);
    - Símbolo: -;
- Erro de contexto: (ASONOV, 2010);
- Erro de digitação:
    - Inserção: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (CLARK & ARAKI, 2011), (HAN & BALDWIN, 2011), (HARTMANN *et al.*, 2014), (CLARK, 2003) e (TOUTANOVA & MOORE, 2002);
    - Deleção: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (CLARK & ARAKI, 2011), (HAN & BALDWIN, 2011), (HARTMANN *et al.*, 2014), (CLARK, 2003) e (TOUTANOVA & MOORE, 2002);
    - Substituição: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (CLARK & ARAKI, 2011), (HAN & BALDWIN, 2011), (HARTMANN *et al.*, 2014), (CLARK, 2003) e (TOUTANOVA & MOORE, 2002);
    - Transposição: (GADDE *et al.*, 2011), (ZORZI & CIASCA, 2009), (CLARK & ARAKI, 2011), (HAN & BALDWIN, 2011), (HARTMANN *et al.*, 2014), (CLARK, 2003) e (TOUTANOVA & MOORE, 2002);
- Dialeto informal: (GADDE *et al.*, 2011), (CLARK & ARAKI, 2011), (CONTRACTOR *et al.*, 2010), (DEEPAK & SUBRAMANIAM, 2012) e (CHOUDHURY *et al.*, 2007);

- Erros específicos do português:
    - am versus ão: (ZORZI & CIASCA, 2009);
    - Porque / Porquê / Por que / Por quê: -.
- Censo: (CLARK & ARAKI, 2011);

APÊNDICE B – Estrutura das classes de função

Para a estrutura de classes das funções codificadoras, tem-se as classes NoCodificador1 e NoCodificador2, os quais estão no mesmo nível da classe NoSeparador, a qual já foi previamente explicada em 6.1, contudo existem duas classes para modelar os dois tipos de configuração do módulo codificador. A Figura 37 apresenta a implementação da classe NoCodificador1, onde difere de NoCodificador2 apenas no valor das variáveis *identificadorModulo* (que passa a valer 2) e *prefixo* (que passa a valer *cod.co.2*).

```
public class NoCodificador1 extends Node {

    public static final int identificadorModulo = 1;
    protected String prefixo = "cod.co.1.";
    private CodificadorInterface codificador;

    // Construtor

    public NoCodificador1(CodificadorInterface codificador) {
        super(null, null);
        this.setIdNodeModulo(identificadorModulo);
        this.codificador = codificador;
    }

    // Métodos do Nó

    public CodificadorInterface getCodificador(){ return this.codificador; }

    @Override
    public String getIdentifier(){ return prefixo+codificador.getIdentifier(); }

    public static Token codifica(Token tokenGP, CodificadorInterface codificador){
        return codificador.codifica(tokenGP);
    }

    // Métodos do framework EpochX

    @Override
    public Boolean evaluate() { return null; }

    @Override
    public Class<?> getReturnType(final Class<?> ... inputTypes) { return Void.class; }
}
```

Figura 37 - Implementação da classe NoCodificador1

A interface CodificadorInterface tem sua implementação mostrada na Figura 38, onde dentre as funções que deverão ser implementadas por cada classe de função codificadora, destaca-se a presença de quatro métodos extras: (1) convertePalavraDicionario, o qual é responsável por definir se este método deve também ser executado em cima das palavras do léxico caso a função identificadora opte pela sua aplicação e (2, 3 e 4) obedeceLeiCanonica, obedeceLeiCanonicaOrdenada e obedebeLeiCanonicaOrdenadaSemRepeticao as quais contêm o valor de cada variável já anteriormente explicada em 5.4.1.2.2.

```java
public interface CodificadorInterface {

    // métodos a serem implementados por cada classe de função codificadora
    public Token funcaoCodificadora(Token tokenGP);
    public boolean convertePalavraDicionario();
    public boolean obedeceLeiCanonica();
    public boolean obedeceLeiCanonicaOrdenada();
    public boolean obedeceLeiCanonicaOrdenadaSemRepeticao();

    // métodos a serem implementados pela classe master
    public String getIdentifier();
    public Token codifica(Token tokenGP);

    // métodos utilizados na classe master para controle
    public long getTempo();
    public long getChamadas();
    public long getRelacao();
    public void zera();
}
```

Figura 38 - Implementação da interface CodificadorInterface

A implementação da classe CodificadorMaster é mostrada na Figura 39.

```java
public abstract class CodificadorMaster implements CodificadorInterface{

    protected AtomicLong tempo = new AtomicLong(0);
    protected AtomicLong chamadas = new AtomicLong(0);

    public String getIdentifier(){ return this.getClass().getSimpleName(); }

    @Override
    public Token codifica(Token tokenGP) {
        long t1 = System.nanoTime();
        Token tokenGPCodificado = this.funcaoCodificadora(tokenGP);
        if (tokenGPCodificado!=null){
            if (tokenGPCodificado.tokenAtual.length()==0){ tokenGPCodificado = null; }
            else{ tokenGPCodificado.addFuncao(getIdentifier()); }
        }
        long t2 = System.nanoTime();
        tempo.addAndGet((t2-t1));
        chamadas.incrementAndGet();
        return tokenGPCodificado;
    }

    @Override
    public long getTempo() { return tempo.longValue(); }

    @Override
    public long getChamadas() { return chamadas.longValue(); }

    @Override
    public long getRelacao() {
        if (chamadas.longValue()>0){ return tempo.longValue()/chamadas.longValue(); }
        return 0;
    }
    @Override
    public void zera(){ tempo.set(0); chamadas.set(0); }
}
```

Figura 39 - Implementação da classe CodificadorMaster

Um exemplo de implementação de uma classe de função codificadora é mostrada na Figura 40, que no caso coloca todos os caracteres presentes no token em caixa baixa.

```java
public class CcaixaBaixa_TTTT extends CodificadorMaster {

    public boolean convertePalavraDicionario() { return true; }
    public boolean obedeceLeiCanonica() { return true; }
    public boolean obedeceLeiCanonicaOrdenada() { return true; }
    public boolean obedeceLeiCanonicaOrdenadaSemRepeticao() { return true; }

    @Override
    public Token funcaoCodificadora(Token tokenGP) {
        tokenGP.tokenAtual = tokenGP.tokenAtual.toLowerCase();
        return tokenGP;
    }
}
```

Figura 40 - Exemplo de classe de função codificadora

```java
public class NoGerador extends Node {

    public static final int identificadorModulo = 5;
    protected String prefixo = "ger.";
    private GeradorInterface gerador;

    // Construtor

    public NoGerador(GeradorInterface gerador) {
        super(nodeNull);
        this.setIdNodeModulo(identificadorModulo);
        this.gerador = gerador;
    }

    // Métodos do Nó

    public GeradorInterface getGerador(){ return this.gerador; }

    @Override
    public String getIdentifier(){ return prefixo+gerador.getIdentifier(); }

    public static List<Token> geraTokensFilhos(Token tokenGP, GeradorInterface gerador){
        List<Token> retorno = new Vector<Token>();
        if (tokenGP.getCaminho().contains("|"+gerador.getIdentifier()+"|")){
            retorno.add(tokenGP);
        }
        else{
            retorno = gerador.gera(tokenGP);
        }
        return retorno;
    }

    // Métodos do framework EpochX

    @Override
    public Boolean evaluate() { return null; }

    @Override
    public Class<?> getReturnType(final Class<?> ... inputTypes) { return Void.class; }
}
```

Figura 41 - Implementação da classe NoGerador

No caso da estrutura de classe das funções geradoras, tem-se inicialmente a classe NoGerador, a qual é mostrada na Figura 41. Por questões de velocidade de execução e característica das funções geradoras desenvolvidas, é colocada uma clausula antes da execução da função geradora, encontrada no método geraTokensFilhos da classe NoGerador, onde limita cada token passar no máximo uma vez por cada função geradora. No caso de uma nova tentativa de chamada de uma mesma função geradora

para um determinado token, este simplesmente passa pela clausula onde não é feita a sua aplicação e este segue normalmente em direção ao nó filho. A Figura 42 e Figura 43 mostram respectivamente as implementações da interface GeradorInterface e da classe GeradorMaster.

```
public interface GeradorInterface {

    // método a ser implementado por cada classe de função geradora
    public List<Token> funcaoGeradora(Token tokenGP);

    // métodos a serem implementados pela classe master
    public String getIdentifier();
    public List<Token> gera(Token tokenGP);

    // métodos utilizados na classe master para controle
    public long getTempo();
    public long getChamadas();
    public long getRelacao();
    public void zera();

}
```

Figura 42 - Implementação da interface GeradorInterface

```
public abstract class GeradorMaster implements GeradorInterface{

    public AtomicLong tempo = new AtomicLong(0);
    public AtomicLong chamadas = new AtomicLong(0);

    public String getIdentifier(){ return this.getClass().getSimpleName(); }

    @Override
    public List<Token> gera(Token tokenGP) {
        long t1 = System.nanoTime();
        List<Token> tokensRetorno = funcaoGeradora(tokenGP);
        if (tokensRetorno!=null){
            for (Token tokenGerado: tokensRetorno){ tokenGerado.addFuncao(getIdentifier()); }
        }
        long t2 = System.nanoTime();
        tempo.addAndGet((t2-t1));
        chamadas.incrementAndGet();
        return tokensRetorno;
    }

    @Override
    public long getTempo() { return tempo.longValue(); }

    @Override
    public long getChamadas() { return chamadas.longValue(); }

    @Override
    public long getRelacao() {
        if (chamadas.longValue()>0){ return tempo.longValue()/chamadas.longValue(); }
        return 0;
    }

    @Override
    public void zera(){ tempo.set(0); chamadas.set(0); }
}
```

Figura 43 - Implementação da classe GeradorMaster

A Figura 44 apresenta um exemplo de implementação de uma classe de função geradora, onde no caso esta separa o token nos aparecimentos do caractere espaço. É importante notar que a classe geradora é responsável por atualizar os valores das

variáveis *posInicio* e *posFim*, de modo que o novo token gerado tenha a sua posição em relação ao token original.

```
public class Tespaco extends GeradorMaster {

    @Override
    public List<Token> funcaoGeradora(Token tokenGP) {
        String[] tokensArray = tokenGP.tokenAtual.split(" ");
        String[] tokensOriginalArray = tokenGP.tokenOriginal.split(" ");
        List<Token> tokensRetorno = new Vector<Token>();
        if (tokensArray.length!=tokensOriginalArray.length){
            return tokensRetorno;
        }

        int inicio = tokenGP.posInicio;
        for (int i=0;i<tokensArray.length;i++){
            String s = tokensArray[i];
            String sOriginal = tokensOriginalArray[i];
            Token novoTokenGP = tokenGP.clone();
            novoTokenGP.tokenOriginal = sOriginal;
            novoTokenGP.tokenAtual = s;
            novoTokenGP.posInicio = i+inicio;
            novoTokenGP.posFim = i+inicio+sOriginal.length();
            inicio = inicio+sOriginal.length();
            tokensRetorno.add(novoTokenGP);
        }
        return tokensRetorno;
    }

}
```

Figura 44 - Exemplo de classe de função geradora

A estrutura de classes das funções seletoras tem inicialmente a classe NoSeletor, como mostrada a sua implementação na Figura 45. A Figura 46 apresenta a interface SeletorInterface e a Figura 47 a implementação da classe master SeletorMaster. A Figura 48 por fim, apresenta um exemplo de implementação de uma classe de função seletora, onde elimina todas as sugestões de correção onde o token original não consistir de um prefixo da sugestão.

```java
public class NoSeletor extends Node {

    public static final int identificadorModulo = 4;
    protected String prefixo = "sel.";
    private SeletorInterface seletor;

    // Construtor

    public NoSeletor(SeletorInterface seletor) {
        super(nodeNull);
        this.setIdNodeModulo(identificadorModulo);
        this.seletor = seletor;
    }

    // Métodos do Nó

    public SeletorInterface getSeletor(){ return this.seletor; }

    @Override
    public String getIdentifier(){ return prefixo+seletor.getIdentifier(); }

    public static List<Token> seleciona(List<Token> tokens, SeletorInterface seletor){
        List<Token> retorno = new Vector<Token>();
        if (tokens==null){
            tokens = new Vector<Token>();
        }
        List<Token> resposta = seletor.seleciona(tokens);
        if (resposta!=null){
            retorno.addAll(resposta);
        }
        return retorno;
    }

    // Métodos do framework EpochX

    @Override
    public Boolean evaluate() { return null; }

    @Override
    public Class<?> getReturnType(final Class<?> ... inputTypes) { return Void.class; }
}
```

Figura 45 - Implementação da classe NoSeletor

```java
public interface SeletorInterface {

    // método a ser implementado por cada classe de função seletora
    public List<Token> funcaoSeletora(List<Token> tokenGP);

    // métodos a serem implementados pela classe master
    public String getIdentifier();
    public List<Token> seleciona(List<Token> tokenGP);

    // métodos a serem implementados na classe master para controle
    public long getTempo();
    public long getChamadas();
    public long getRelacao();
    public void zera();
}
```

Figura 46 - Implementação da interface SeletorInterface

```java
public abstract class SeletorMaster implements SeletorInterface{

    public AtomicLong tempo = new AtomicLong(0);
    public AtomicLong chamadas = new AtomicLong(0);

    public String getIdentifier(){ return this.getClass().getSimpleName(); }

    @Override
    public List<Token> seleciona(List<Token> tokensGP) {
        long t1 = System.nanoTime();
        List<Token> retorno = funcaoSeletora(tokensGP);

        for (Token tokenGP: retorno){
            tokenGP.addFuncao(getIdentifier());
        }
        long t2 = System.nanoTime();
        tempo.addAndGet((t2-t1));
        chamadas.incrementAndGet();
        return retorno;
    }

    @Override
    public long getTempo() { return tempo.longValue(); }

    @Override
    public long getChamadas() { return chamadas.longValue(); }

    @Override
    public long getRelacao() {
        if (chamadas.longValue()>0){ return tempo.longValue()/chamadas.longValue(); }
        return 0;
    }

    @Override
    public void zera(){ tempo.set(0); chamadas.set(0); }
}
```

Figura 47 - Implementação da classe SeletorMaster

```java
public class FFORTEoriginalPrefixoDeCorrecao_DcaixaEacentos extends SeletorMaster {

    @Override
    public List<Token> funcaoSeletora(List<Token> tokensGP) {
        List<Token> retorno = new Vector<Token>();

        for (Token tokenGP: tokensGP){
            String original = StringUtils.retiraAcentos(tokenGP.tokenOriginal.toLowerCase());
            String correcao = StringUtils.retiraAcentos(tokenGP.tokenCorrecao.toLowerCase());

            if (correcao.startsWith(original)){
                retorno.add(tokenGP);
            }
        }

        return retorno;
    }
}
```

Figura 48 - Exemplo de classe de função seletora

Por fim, tem-se a estrutura de classe das funções identificadoras, onde a implementação da classe NoIdentificador é mostrada na Figura 49. Na Figura 50 é apresentada a interface com a definição dos métodos que devem ser implementados pela classe master, a qual esta última é mostrada na Figura 51, e pelas funções identificadoras, mostrada no exemplo da Figura 52, que no caso encontra os vocábulos

no léxico que tem a codificação pelo algoritmo Soundex para português igual codificação do token que chegou a função identificadora.

```
public class NoIdentificador extends Node {

    public static final int identificadorModulo = 8;
    protected String prefixo = "ide.";
    private IdentificadorInterface identificador;

    // Construtor

    public NoIdentificador(IdentificadorInterface identificador) {
        super();
        this.setIdNodeModulo(identificadorModulo);
        this.identificador = identificador;
    }

    // Métodos do Nó

    @Override
    public String getIdentifier() { return prefixo + identificador.getIdentifier(); }

    public static List<Token> identifica(Token tokenGP, IdentificadorInterface identificador,
            Map<String, Boolean> terminaisEncontrados) {
        List<Token> retorno = new Vector<Token>();

        List<Token> tokens = identificador.identifica(tokenGP);

        if (tokens != null) {
            for (Token token : tokens) {
                if (terminaisEncontrados.containsKey(token.tokenCorrecao)) {
                    terminaisEncontrados.put(token.tokenCorrecao, true);
                }
            }
            retorno.addAll(tokens);
        }

        return retorno;
    }

    public static List<Token> executa(Node node, Token tokenGP,
            IdentificadorInterface identificador) {
        List<Token> retorno = new Vector<Token>();

        List<Token> tokens = identificador.identifica(tokenGP);
        if (tokens != null) {
            retorno.addAll(tokens);
        }

        return retorno;
    }

    public IdentificadorInterface getIdentificador() { return this.identificador; }

    // Métodos do framework EpochX

    @Override
    public Boolean evaluate() { return null; }

    @Override
    public Class<?> getReturnType(final Class<?>... inputTypes) { return Void.class; }
}
```

Figura 49 - Implementação da classe NoIdentificador

```java
public interface IdentificadorInterface {

    // Método a ser implementado por cada classe de função identificadora
    public List<Token> funcaoIdentificadora(Token tokenGP);
    public boolean exigeRepresentacaoAdicionalDicionario();
    public String getRepresentacao(String s);

    // Métodos a serem implementados pela classe master
    public String getIdentifier();
    public List<Token> identifica(Token tokenGP);

    // Métodos utilizados na classe master para controle
    public long getTempo();
    public long getChamadas();
    public long getRelacao();
    public void zera();
}
```

Figura 50 - Implementação da interface IdentificadorInterface

```java
public abstract class IdentificadorMaster implements IdentificadorInterface{

    public AtomicLong tempo = new AtomicLong(0);
    public AtomicLong chamadas = new AtomicLong(0);

    public String getIdentifier(){ return this.getClass().getSimpleName(); }

    @Override
    public List<Token> identifica(Token tokenGP) {
        long t1 = System.nanoTime();

        List<Token> retorno = funcaoIdentificadora(tokenGP);
        if (retorno!=null){
            for (Token t: retorno){
                t.addFuncao(getIdentifier());
            }
        }

        long t2 = System.nanoTime();
        tempo.addAndGet((t2-t1));
        chamadas.incrementAndGet();
        return retorno;
    }

    @Override
    public long getTempo() { return tempo.longValue(); }

    @Override
    public long getChamadas() { return chamadas.longValue(); }

    @Override
    public long getRelacao() {
        if (chamadas.longValue()>0){ return tempo.longValue()/chamadas.longValue(); }
        return 0;
    }

    @Override
    public void zera(){ tempo.set(0); chamadas.set(0); }
}
```

Figura 51 - Implementação da classe IdentificadorMaster

```java
public class MsoundexPt extends IdentificadorMaster {

    public boolean exigeRepresentacaoAdicionalDicionario() { return true; }

    @Override
    public List<Token> funcaoIdentificadora(Token tokenGP) {
        List<Token> retorno = Dicionario.recuperaInstancias(getRepresentacao(tokenGP.tokenAtual),
                new MsoundexPt(), tokenGP);
        return retorno;
    }

    @Override
    public String getRepresentacao(String s) {
        if (s.length()==0){ return ""; }

        char[] x = s.toUpperCase().toCharArray();
        char firstLetter = x[0];

        for (int i = 0; i < x.length; i++) {
            switch (x[i]) {
                case 'P': case 'B': case 'M': { x[i] = '1'; break; }
                case 'F': case 'V': { x[i] = '2'; break; }
                case 'T': case 'D': case 'S': case 'Z': case 'Ç': { x[i] = '3'; break; }
                case 'L': case 'R': case 'N': { x[i] = '4'; break; }
                case 'X': case 'J': { x[i] = '5'; break; }
                case 'K': case 'Q': case 'G': case 'C': { x[i] = '6'; break; }
                default:  { x[i] = '0'; break; }
            }
        }

        String output = "" + firstLetter;
        for (int i = 1; i < x.length; i++){
            if (x[i] != x[i-1] && x[i] != '0'){ output += x[i]; }
        }

        output = output + "0000";
        String f = output.substring(0, 4);
        return f;
    }
}
```

Figura 52 - Exemplo de classe de função identificadora

APÊNDICE C – Exemplo de programa normalizador gerado pela arquitetura

Abaixo é apresentado a codificação LISP que representa um programa normalizador de texto, aplicado no cenário de tweets em português, encontrado em uma das instâncias rodadas para se determinar o valor base para validação da arquitetura, apresentada na Tabela 24.

Cada uma das funções identificadas na codificação LISP está descrita no capítulo 6. A codificação LISP do programa corresponde a:

*sep.G0tamanhoTokenIgualOuMenor2_G1outros(cod.cp.esq.SkPorCA_NtokensComSeparador_*
FTTT(del.FFORTEoriginalContendoPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem_DcaixaEac
entos(sep.G0tamanhoTokenIgual1_G1outros(sep.G0apenasConsoantes_G1apenasConsoantesEvogais_G
Eoutros(del.DmenorDistanciaLevenshtein_Dcaixa(ger.Tespaco(mat.Nothing()))
cod.cp.esq.Racentos_TTTT(del.DmenorDistanciaLevensteinDeCodificacoesMetaphonePro(mat.Iurl())
cod.cp.esq.Respacos_FTTT(mat.MtokenComCadaLetraInicioDeUmaSilaba() mat.Icarinha())))
ger.TletrasCombinadas(cod.cp.esq.SkPorC_NtokensComSeparador_FTTT(cod.cp.esq.SkPorC_NtokensC
omSeparador_FTTT(mat.Nothing() mat.IrisosRs()) mat.Digual())))
cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(sep.G0tamanhoTokenIgualOuM
enor2_G1outros(sep.G0apenasLetrasComRepeticaoTratamentoEspecialRS_G1apenasLetrasSemRepetic
ao_GEoutros(del.DmenorDistanciaLevenshtein_Dcaixa(cod.cp.dir.SuPorOnoFim_NtokensComSeparado
r_FTTT(mat.IrisosKkkk()
del.FFORTEoriginalIgualCorrecao(del.FFRACOoriginalIgualCorrecao_Dcaixa(mat.Imention()))))
cod.cp.dir.CesqueletoConsoante_NtokensComAlgoDiferenteDeLetra_TTTT(del.FFORTEoriginalContend
oPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem_DcaixaEacentos(del.FFORTEoriginalSufixoDe
Correcao_DcaixaEacentos(mcp(mat.IrisosKkkk() mat.Nothing())))
ger.Tespaco(msp(cod.cp.dir.SwPorU_NtokensComSeparador_FTTT(mat.DigualDiminutivoInhoa()
mat.MsoundexPt())
del.DmenorDistanciaLevenshtein_Dcaixa(cod.cp.dir.SuPorOnoFim_NtokensComSeparador_FTTT(mat.I
risosKkkk()
del.FFORTEoriginalIgualCorrecao(cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_T
FTT(del.FFORTEprimeiraLetraOriginalIgualPrimeiraLetraCorrecao_DcaixaEacentos(mat.Digual())
cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrat
aCaixaEacento_FFFT(mat.Digual() mat.Digual())
cod.cp.esq.Racentos_TTTT(cod.cp.dir.SiPorEnoFim_NtokensComSeparador_FTTT(mat.IrisosHuae()
mat.Icarinha()) cod.cp.dir.Rvogais_TTTT(mat.Ihashtag() mat.Icarinha()))))))))))
cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(cod.cp.esq.SkPorC_Ntokens
ComSeparador_FTTT(del.FFORTEoriginalContidoEmCorrecao_DcaixaEacentos(mat.Nothing())

cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.IrisosRs() mat.IrisosRs()))
sep.G0tamanhoTokenIgual3_G1outros(del.FFORTEoriginalSufixoDeCorrecao_DcaixaEacentos(mat.Imention()) del.FFRACOoriginalPrefixoDeCorrecao_DcaixaEacentos(mat.MsoundexEn()))))
sep.G0letrasEnumerosAmbosEapenas_G1outros(mat.Digual()
cod.cp.esq.SwPorU_NtokensComSeparador_FTTT(del.FFORTEoriginalSufixoDeCorrecao_DcaixaEacentos(mat.MsoundexEn())
cod.cp.esq.Rvogais_TTTT(del.DmenorDistanciaLevenshtein_Dcaixa(ger.TletrasCombinadas(mat.Icarinha()))
sep.G0tokenCaracteristicoCerto_G1outros(del.FFORTEmesmasLetrasOriginalEcorrecaoSemConsiderarRepeticaoDeLetras_DcaixaAcentos(mat.IrisosRs()) cod.cp.dir.Rvogais_TTTT(mat.Icarinha()
mat.Icarinha()))))))
ger.Tespaco(cod.cp.esq.SwPorU_NtokensComSeparador_FTTT(sep.G0tokenCaracteristicoCerto_G1outros(del.DmaiorFrequenciaDicionario(mcp(cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(del.FFORTEoriginalSufixoDeCorrecao_DcaixaEacentos(mat.MsoundexEn())
cod.cp.esq.SiPorEnoFim_NtokensComSeparador_FTTT(del.FFRACOoriginalIgualCorrecao_DcaixaEacentos(mat.Icarinha())
sep.G0tokenCaracteristicoCerto_G1outros(cod.cp.esq.Respacos_FTTT(mat.MtokenComCadaLetraInicioEletraNasalizacaoPorSilaba()
cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(mat.Digual() mat.Digual()))
del.FFORTEletrasOriginalPresentesEmCorrecaoNaMesmaOrdem_DcaixaEacentos(mat.MsoundexPt()))
))
cod.cp.dir.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(sep.G0tokenIniciadoPelaLetraTseguidaDeUmaVogal_G1outros(cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(sep.G0tamanhoTokenIgualOuMenor2_G1outros(sep.G0apenasLetrasComRepeticaoTratamentoEspecialRS_G1apenasLetrasSemRepeticao_GEoutros(del.DmenorDistanciaLevenshtein_Dcaixa(del.DmenorDistanciaLevenshtein_Dcaixa(cod.cp.dir.SuPorOnoFim_NtokensComSeparador_FTTT(mat.IrisosKkkk()
del.FFORTEoriginalIgualCorrecao(del.FFRACOoriginalIgualCorrecao_Dcaixa(mat.Imention()))))
sep.G0apenasLetras_G1outros(mat.IrisosKkkk() mat.DigualDiminutivoInhoa()))
cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(cod.cp.esq.SkPorC_NtokensComSeparador_FTTT(del.FFORTEoriginalContidoEmCorrecao_DcaixaEacentos(mat.Nothing())
del.FFORTEoriginalIgualCorrecao_DcaixaEacentos(del.FFORTEoriginalIgualCorrecao_DcaixaEacentos(cod.cp.dir.Rvogais_TTTT(del.DdescarteSugestoesSemelhantesDiferenciandoApenasEmCaixa(mat.MsoundexEn()) cod.cp.esq.SiPorEnoFim_NtokensComSeparador_FTTT(mat.MsoundexEn()
mat.IrisosRs())))))
sep.G0tamanhoTokenIgual3_G1outros(del.FFORTEoriginalSufixoDeCorrecao_DcaixaEacentos(mat.Imention()) del.FFRACOoriginalPrefixoDeCorrecao_DcaixaEacentos(mat.MsoundexEn()))))
mat.Ihashtag()) mat.MtokenComCadaLetraInicioEletraNasalizacaoPorSilaba()) mat.Ihashtag())))
cod.cp.dir.Rvogais_TTTT(mat.Icarinha() mat.Icarinha()))

cod.cp.esq.SkPorC_NtokensComSeparador_FTTT(del.DmenorDistanciaLevenshtein_Dcaixa(ger.TletrasCombinadas(mat.Icarinha())) cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Digual() cod.cp.dir.Racentos_TTTT(del.FFRACOoriginalConsistidoPelaPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem_DcaixaEacentos(sep.G0letraTseguidoDeUmaVogalEmTokenDe2Caracteres_G1outros(mat.Ihashtag() ger.Tespaco(cod.cp.esq.SwPorU_NtokensComSeparador_FTTT(del.DmenorDistanciaLevenshtein_Dcaixa(cod.cp.dir.SuPorOnoFim_NtokensComSeparador_FTTT(mat.IrisosKkkk() del.FFORTEoriginalIgualCorrecao(cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(del.FFORTEprimeiraLetraOriginalIgualPrimeiraLetraCorrecao_DcaixaEacentos(mat.Digual()) cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(mat.Digual() mat.Digual()) cod.cp.esq.Racentos_TTTT(cod.cp.dir.SiPorEnoFim_NtokensComSeparador_FTTT(mat.IrisosHuae() mat.Icarinha()) cod.cp.dir.Rvogais_TTTT(mat.Ihashtag() mat.Icarinha()))))))) cod.cp.esq.SkPorC_NtokensComSeparador_FTTT(cod.cp.esq.ShPorAcentoAgudoNoFim_NtokensComSeparador_FTTT(mat.Imention() cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Iurl() mat.Ihashtag())) cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Digual() cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Digual() mat.Iurl()))))))) ger.Tpontuacao(cod.cp.dir.SwPorU_NtokensComSeparador_FTTT(cod.cp.dir.Racentos_TTTT(del.FFRACOoriginalConsistidoPelaPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem_DcaixaEacentos(sep.G0letraTseguidoDeUmaVogalEmTokenDe2Caracteres_G1outros(mat.Ihashtag() ger.Tespaco(cod.cp.esq.SwPorU_NtokensComSeparador_FTTT(del.DmenorDistanciaLevenshtein_Dcaixa(cod.cp.dir.SuPorOnoFim_NtokensComSeparador_FTTT(mat.IrisosKkkk() del.FFORTEoriginalIgualCorrecao(cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(del.FFORTEprimeiraLetraOriginalIgualPrimeiraLetraCorrecao_DcaixaEacentos(mat.Digual()) cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(mat.Digual() mat.Digual()) cod.cp.esq.Racentos_TTTT(cod.cp.dir.SiPorEnoFim_NtokensComSeparador_FTTT(mat.IrisosHuae() mat.Icarinha()) cod.cp.dir.Rvogais_TTTT(mat.Ihashtag() mat.Icarinha()))))))) cod.cp.esq.SkPorC_NtokensComSeparador_FTTT(cod.cp.esq.ShPorAcentoAgudoNoFim_NtokensComSeparador_FTTT(mat.Imention() cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Iurl() mat.Ihashtag())) cod.cp.esq.ShPorAcentoAgudoNoFim_NtokensComSeparador_FTTT(mat.Imention() cod.cp.esq.SuPorOnoFim_NtokensComSeparador_FTTT(mat.Iurl() mat.Ihashtag()))))))) ger.Tpontuacao(cod.cp.dir.SwPorU_NtokensComSeparador_FTTT(del.DmaiorFrequenciaDicionario(mcp(cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(del.FFORTEoriginalSufixoDeCorrecao_DcaixaEacentos(mat.MsoundexEn()) cod.cp.esq.SiPorEnoFim_NtokensComSeparador_FTTT(del.FFRACOoriginalIgualCorrecao_DcaixaEacentos(mat.Icarinha()) sep.G0tokenCaracteristicoCerto_G1outros(cod.cp.esq.Respacos_FTTT(mat.MtokenComCadaLetraInicio

EletraNasalizacaoPorSilaba() del.FFRACOoriginalIgualCorrecao(mat.Icarinha()))
del.FFORTEletrasOriginalPresentesEmCorrecaoNaMesmaOrdem_DcaixaEacentos(mat.MsoundexPt()))
))
cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(del.DpreenchimentoMaisCompletoComMenorNumeroTokens(del.DdescarteSugestoesSemelhantesDiferenciandoApenasEmCaixa(del.FFORTEletrasOriginalPresentesEmCorrecaoNaMesmaOrdem_DcaixaEacentos(mat.MsoundexEn())))
cod.cp.dir.Racentos_TTTT(mat.IrisosHuae()
del.FFORTEoriginalConsistidoPelaPrimeiraLetraDeCadaSilabaCorrecaoNaMesmaOrdem_DcaixaEacentos(cod.cp.dir.CcaixaBaixa_TTTT(mat.Nothing() mat.MtokenComCadaLetraInicioDeUmaSilaba()))))
cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(del.FFORTEprimeiraLetraOriginalIgualPrimeiraLetraCorrecao_DcaixaEacentos(mat.Digual())
cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(mat.Digual() mat.Digual())
cod.cp.esq.Racentos_TTTT(cod.cp.dir.SiPorEnoFim_NtokensComSeparador_FTTT(mat.IrisosHuae()
mat.Icarinha()) cod.cp.dir.Rvogais_TTTT(mat.Ihashtag() mat.Iurl()))))))
cod.cp.esq.CordemAlfabetica_NtokensComAlgoDiferenteDeLetra_TFTT(del.FFORTEprimeiraLetraOriginalIgualPrimeiraLetraCorrecao_DcaixaEacentos(mat.Digual())
cod.cp.esq.SxPorCH_NtokensComSeparador_FTTT(cod.cp.dir.RletraRepetidaComTratamentoRSnaoTrataCaixaEacento_FFFT(mat.Digual() mat.Digual())
cod.cp.esq.Racentos_TTTT(cod.cp.dir.SiPorEnoFim_NtokensComSeparador_FTTT(mat.IrisosHuae()
mat.Icarinha()) cod.cp.dir.Rvogais_TTTT(mat.Ihashtag() mat.Iurl()))))))))))))