UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
BACHARELADO CIÊNCIA DA COMPUTAÇÃO

Leonardo de Abreu Schmidt

ALGORITMOS DE SIMILARIDADE E INDEXAÇÃO PARA TEXTOS DIFUSOS

Santa Maria, RS
2017

Leonardo de Abreu Schmidt

ALGORITMOS DE SIMILARIDADE E INDEXAÇÃO PARA TEXTOS DIFUSOS

Trabalho de Conclusão de Curso apresentado ao Bacharelado Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Ciência da Computação**

Orientador: Prof. Dr. Sergio Luis Sardi Mergen (UFSM)

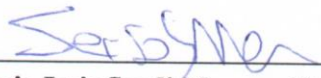
435
Santa Maria, RS
2017

Leonardo de Abreu Schmidt

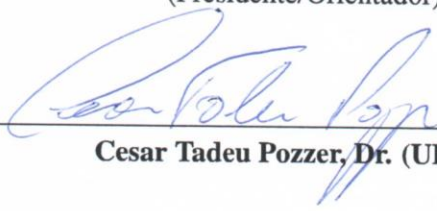
ALGORITMOS DE SIMILARIDADE E INDEXAÇÃO PARA TEXTOS DIFUSOS

Trabalho de Conclusão de Curso apresentado ao Bacharelado Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Bacharel em Ciência da Computação**

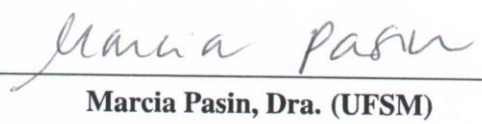
Aprovado em 11 de dezembro de 2017:



Sergio Luis Sardi Mergen, Dr. (UFSM)
(Presidente/Orientador)



Cesar Tadeu Pozzer, Dr. (UFSM)



Marcia Pasin, Dra. (UFSM)

Santa Maria, RS

2017

AGRADECIMENTOS

Agradeço primeiramente a Deus. Agradeço aos meus pais, Wolnei e Silaine, que sempre me apoiaram e me incentivaram, e que sempre me deram todas as possibilidades de estudo. Muito obrigado pai e mãe pela confiança que depositaram em mim, vocês são o motivo maior de todo meu esforço. Obrigado ao meu irmão, do qual estive longe na maior parte desse tempo de graduação, e nem sempre consegui estar presente. Obrigado aos meus avós por sempre me ajudarem e me apoiarem, vocês são pessoas incríveis.

Agradeço a minha namorada Carin, principalmente nesse último semestre, pela confiança, paciência, e pela ajuda. Agradeço também a todos meus amigos pelo companheirismo de todos esses anos. Obrigado também aos meus professores, em especial aos meus orientadores de projetos durante a graduação, professora Ana Wink, professora Marcia Pasin e meu orientador do trabalho de graduação, professor Sergio Mergen, com os quais aprendi muito em todos esses anos. Obrigado a todos que de alguma forma me ajudaram nessa conquista.

"Motivação é a arte de fazer as pessoas fazerem o que você quer que elas façam porque elas o querem fazer."

(DWIGHT EISENHOWER)

RESUMO

ALGORITMOS DE SIMILARIDADE E INDEXAÇÃO PARA TEXTOS DIFUSOS

AUTOR:
LEONARDO DE ABREU SCHMIDT
ORIENTADOR: SERGIO LUIS SARDI MERGEN (UFSM)

O presente trabalho estuda o reconhecimento de caracteres manuscritos, utilizando a técnica de redes neurais artificiais como classificador e um valor de *threshold* para geração de saídas difusas. Todas as classificações que ficam com *score* maior que o valor de *threshold* são consideradas como candidatos, ou seja, pode haver mais de uma letra por posição na palavra reconhecida. Essas saídas, em forma de palavras difusas, são pesquisadas em uma estrutura indexada difusa, que é o foco de desenvolvimento do trabalho. Essa estrutura desenvolvida é baseada em uma árvore BK-Tree, que utiliza o algoritmo de cálculo de distância de Levenshtein modificado, para indexar e pesquisar palavras. O algoritmo proposto melhora o tempo de busca devido a capacidade de pesquisar simultaneamente mais de uma palavra por vez além de melhorar os resultados de saída da rede neural, pois considera mais possibilidades. Os resultados foram melhores em relação aos algoritmos de indexação e busca originais da BK-Tree e Levenshtein.

Palavras-chave: Redes Neurais Artificiais. Distância de Levenshtein. BK-Tree. *Threshold*.

ABSTRACT

SIMILARITY AND INDEXING ALGORITHMS FOR FUZZY TEXTS

AUTHOR:
LEONARDO DE ABREU SCHMIDT
ADVISOR: SERGIO LUIS SARDI MERGEN (UFSM)

The present work studies the recognition of handwritten characters, using the technique of artificial neural networks as a classifier and a threshold value for the generation of diffuse outputs. All classifications with a score greater than the threshold value are considered as candidates, that is, there may be more than one letter per position in the recognized word. These outputs, in the form of diffuse words, are searched in a diffused indexed structure, which is the focus of this work. The developed structure is based on a BK-Tree, which uses a modified Levenshtein's distance algorithm to index and search words. The algorithm proposed improves the search time due to the ability to simultaneously search more than one word at a time and improves the output results of the neural network, since it considers more possibilities.

Keywords: Artificial Neural Network. Levenshtein Distance. BK-Tree. Threshold.

LISTA DE FIGURAS

LISTA DE TABELAS

LIST OF ALGORITHMS

LISTA DE ABREVIATURAS E SIGLAS

BK-TREE	*Buckhard Keller Tree*
LCS	*Longest Commom Subsequence*
MDB	*Multiple Difuse BK-Tree*
RNA	Rede Neural Artificial
UFSM	Universidade Federal de Santa Maria
VPT	*Vantage Point Tree*

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Dentro da área de reconhecimento de padrões, um dos problemas existentes envolve o reconhecimento de palavras a partir de imagens. Diferentes técnicas de classificação podem ser usadas para esse fim, como as redes neurais artificiais (RNAs) e os algoritmos K-Means e DB-Scan. Essas técnicas podem descobrir o caractere representado em uma imagem de entrada. Uma sequência de caracteres reconhecidos a partir de imagens é unida para compor a palavra correspondente.

Nos últimos anos, a computação neural emergiu como uma tecnologia prática, com aplicações bem-sucedidas em muitos campos. A maioria dessas aplicações está preocupada com os problemas de reconhecimento de padrões, e faz uso de arquiteturas de rede feed-forward, como a perceptron multi-camada e a rede de função de base radial (BISHOP, 1995).

O processamento e aprendizado simbólico, devido as suas características básicas, possuem limitações no que diz respeito a manipulação: de incertezas, de valores aproximados, de informações contraditórias, e de uma maneira geral, de informações quantitativas (MINSKY, 1991).

De modo geral, a partir de uma imagem de entrada, os classificadores retornam a classe de saída (o caractere) que tenha obtido o maior escore, desde que esse escore esteja acima de um ponto de corte (*threshold*). No entanto, existe a possibilidade de que a saída gerada pelo classificador não seja correta.

Dois cenários são possíveis. O primeiro é o de que nenhuma classe de saída esteja acima do *threshold*. Por exemplo, a palavra 'CARMA' é formada após o reconhecimento de cinco imagens de entrada ('C','A','R','M','A'). Caso o classificador tenha falhado na classificação do terceiro caractere, a palavra gerada estaria equivocada ('CAMA').

O outro cenário é o de que várias classes estejam acima do *threshold*. Como o classificador escolhe o caractere com o escore mais alto, é possível que ele faça a escolha errada. No exemplo analisado, os caracteres 'R' e 'P' poderiam ter ficado acima do *threshold* no reconhecimento do terceiro caractere de 'CARMA'. Caso o escore de 'P' seja mais alto, a palavra gerada também estaria equivacada ('CAPMA').

As falhas dos classificadores podem ser reduzidas a partir do uso de técnicas de simila-

ridade e indexação de texto. Por exemplo, a distância de edição de Levensthein pode informar o quão similares duas palavras são. A partir de duas sequências de caracteres, o algoritmo retorna o número de operações de edição necessárias para transformar uma sequência na outra.

Já as árvores BK-trees são mecanismos de indexação que usam essa distância de edição para recuperar, a partir de um dicionário, a palavra indexada que mais se assemelhe a alguma palavra usada na busca. Assim, mesmo que o classificador tenha gerado a palavra errada ('CAMA' ou 'CAPMA'), a busca no índice pode corrigir o problema, retornando a palavra indexada mais semelhante.

1.2 PROBLEMA

As técnicas de indexação de texto e de distância de edição são apropriadas quando o objeto de busca é uma palavra cujos caracteres são exatos. No caso em questão, os caracteres da palavra são difusos (incertos), uma vez que eles foram gerados a partir de um classificador. Isso pode levar a busca no índice a retornar a palavra errada.

Por exemplo, caso o classificador tenha reconhecido a palavra 'CAMA', a busca no índice recupera essa mesma palavra, caso ela esteja indexada. Isso ocorre porque o índice não explora a informação de que a palavra correta possui cinco caracteres (mesmo que um deles não tenha sido reconhecido).

Caso o classificador tenha reconhecido a palavra 'CAPMA', a busca do índice recuperará várias possibilidades de resposta, como 'CAMA', 'CALMA' e 'CARMA', já que todas são igualmente similares à 'CAPMA'. Isso ocorre porque o método de busca não explora o fato de que é mais provável que o terceiro caractere da palavra seja o 'R', de acordo com o escore gerado pelo classificador.

1.3 OBJETIVOS

O objetivo principal desse trabalho é criar extensões do algoritmo de distância de edição de Levenshtein e do algoritmo de busca baseada em BK-trees.

O algoritmo modificado para o cálculo de distância de edição recebe duas sequências, assim como no algoritmo original. No entanto, uma das sequências contem caracteres difusos. Ou seja, o caractere correto para cada uma das posições da sequência é incerto. Além disso, o algoritmo retorna duas informações: a distância mínima de edição e a distância máxima. A

distância mínima considera o cenário otimista, em que nas posições onde existe incerteza, o caractere escolhido é o caractere correto. A distância máxima considera o cenário pessimista, em que nas posições onde existe incerteza, o caractere escolhido é o caractere errado.

O algoritmo de busca baseado em bk-tree também deve ser modificado. A mudança refere-se ao uso do algoritmo de distância de edição estendido. Como esse algoritmo retorna as distâncias mínima e máxima, o método de busca deve ser revisto para que ambas informações sejam utilizadas.

O método de busca proposto foi comparado com outros dois métodos, em um ambiente de teste cujo objetivo é processar as saídas geradas por uma rede neural. A comparação foi feita tanto em função da taxa de acerto da busca e do número de cálculos de distância de edição necessários.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve técnicas de reconhecimento de padrões relevantes ao escopo deste trabalho que podem gerar saídas incorretas para algum objeto de entrada. Além disso, o capítulo também descreve formas de contornar esse problema, a partir do uso de funções para o cálculo de similaridade, cálculo da distância entre objetos e uso de mecanismos de indexação.

2.1 RECONHECIMENTO DE PADRÕES

A área de Reconhecimento de Padrões é uma das principais áreas de estudo em Inteligência Artificial. Ela se preocupa com o estudo do aprendizado de máquina no que diz respeito à generalização de características em dados, e à classificação de objetos de acordo com sua classe.

O reconhecimento de padrões conta com uma gama de técnicas e algoritmos que podem ser aplicados aos mais diversos tipos de dados. O princípio dessas técnicas está na capacidade de generalização de informações de entradas a partir de um processo de *"treinamento"* onde é buscada a convergência de cada entrada mapeada para uma *saída desejada*. Normalmente o processo é repetido diversas vezes até que exista uma convergência aceitável, ou um nível de taxa de erro abaixo do permitido.

Em essência, o reconhecimento abrange o seguinte problema: "Dado exemplos de sinais complexos e a decisão correta para eles, construir uma tomada de decisões automáticas para um fluxo de exemplos futuros" (TOU; GONZÁLEZ, 1977).

O reconhecimento de padrões pode muitas vezes ser modelado como um problema de classificação, onde um objeto é classificado em uma classe dentre um número finito de classes possíveis. A Figura 2.1 mostra a arquitetura da solução desenvolvida. O classificador faz uso de um modelo que define como a classificação é realizada, sendo alguns desses modelos técnicas que envolvem aprendizagem da máquina a partir de um processo de treinamento.

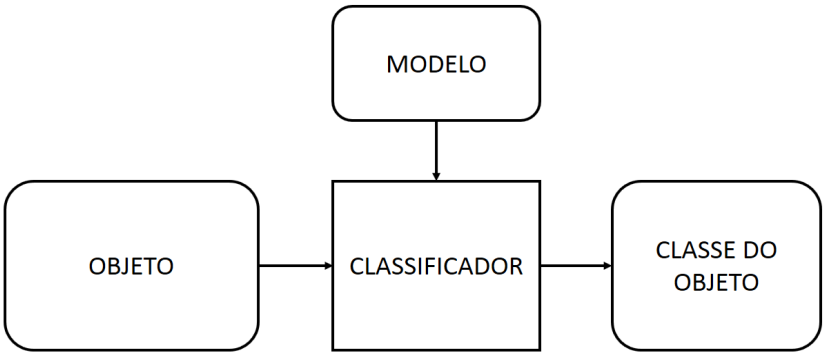


Figura 2.1: Modulo reconhecedor. Fonte: próprio autor.

Para o presente trabalho são tratados apenas técnicas de reconhecimento de padrões que consigam gerar saídas difusas. As próximas seções descrevem duas dessas técnicas.

2.1.1 Redes Neurais Artificiais

A técnica de Redes Neurais Artificiais (*Artificial Neural Networks*) não é nova. Os primeiros passos foram ainda na década de 40, com analogias entre células vivas e processos eletrônicos, e a formalização do processo de um neurônio. As redes neurais surgiram a partir de comparações com modelos da forma como os seres humanos podem abordar tarefas de reconhecimento de padrões (TOU; GONZÁLEZ, 1977).

As Redes Neurais Artificiais são dispositivos não-lineares, inspirados na funcionalidade dos neurônios biológicos, aplicados no reconhecimento de padrões, na otimização e na previsão de sistemas complexos (TRELEAVEN; PACHECO; VELLASCO, 1989).

Problemas de reconhecimento de padrões, como classificação de caracteres impressas ou manuscritos, são atualmente resolvidos com precisão aceitável usando classificadores tradicionais ou redes neurais de diferentes arquiteturas baseadas em diferentes conjuntos de características (ROGOVA, 1994).

Essas redes são arquiteturas compostas de modelos matemáticos de neurônios reais, juntamente com sua estrutura coletiva em uma rede, com diversos neurônios. O objetivo é de que essa estrutura artificial processe sinais de entrada de um determinado domínio, e tenha uma saída. Esse processo é repetido várias vezes para diversos conjuntos de dados do mesmo domínio, agregando assim à rede a capacidade de generalizar sobre dados semelhantes àqueles apresentados a rede.

Uma rede neural possui os seguintes componentes fundamentais em sua organização.

- Entradas: entradas constituem os exemplos que uma rede possui para aprender padrões. Pode-se fazer uma analogia com o aprendizado infantil. Quando algo é mostrado a uma criança diversas vezes e lhe é dito o nome do objeto, a criança começará a associar aquele objeto ao nome.

  Em uma rede neural artificial existem diversas entradas, cada uma com o dado a ser usado como entrada e sua classe correta já classificada. Se a rede errar a classificação de um objeto, ela irá realizar ajustes no seu modelo para que consiga acertar a classificação da próxima vez.

- Pesos: pesos são, de forma direta, "arestas"que ligam os neurônios de uma camada com a camada seguinte conforme Figura 2.2. É neles que se armazenam o conhecimento real da rede e é onde todo o aprendizado está armazenado, e portanto é a única unidade que tem seu valor alterado durante o treinamento da rede. Quando uma determinada entrada é classificada erroneamente, são os pesos que devem ser modificados para que a rede minimize o erro na predição.

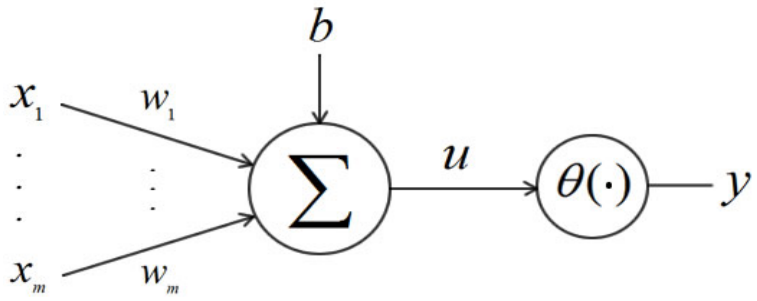


Figura 2.2: Neurônio artificial. Fonte: próprio autor.

- Neurônios: constituem a parte essencial da rede. Basicamente funcionam como uma unidade de processamento de sinais. Duas funções constituem o neurônio artificial.

  – Função somatório: realiza o somatório dos valores das multiplicações das entradas (X_i) pelos seus respectivos pesos (W_i) mais um valor chamado de Bias (b) que aumenta o grau de liberdade da rede. A ativação do neurônio não depende mais de sua entrada mas também desse novo parâmetro de configuração livre. Isso faz com que a rede convirja corretamente mesmo sem ajuda dos valores da entrada conforme equação (2.1), onde M é o numero de amostras (entradas) da rede.

$$(\sum_{i=1}^{M} X_i * W_i) + b \tag{2.1}$$

– Função ativação: tem como entrada o somatório (equação (2.1)) e aplica sobre esse valor uma função de ativação (θ), passando a saída para a próxima etapa da rede. Existem algumas funções de ativação amplamente utilizadas em redes neurais, tais como a função tangente hiperbólica, escada e sigmoide (equação (2.2)) onde o parâmetro λ representa a saída da função somatório ((2.1)). Cada função trabalha em uma faixa de valores e são utilizadas para fins específicos.

$$\frac{1}{1+e^{-\lambda}} \tag{2.2}$$

- Camadas: uma rede neural é configurada de acordo com um número de camadas, e um número de neurônios por camada. Em uma rede pode haver N camadas, e em cada camada um número diferente de neurônios. A primeira camada é sempre chamada de camada de entrada (*input layer*), e é responsável pela passagem dos valores de sinais para a primeira camada de neurônios. É possível haver camadas intermediárias, também chamadas camadas ocultas, e por fim uma camada de saída, que terá tantos neurônios quantos forem os sinais de saída (classes do problema).

  É importante ressaltar que a rede neural é *full-connected*, ou seja, cada neurônio de uma camada é conectado com todos os neurônios da camada anterior, e assim sucessivamente, conforme a Figura 2.3

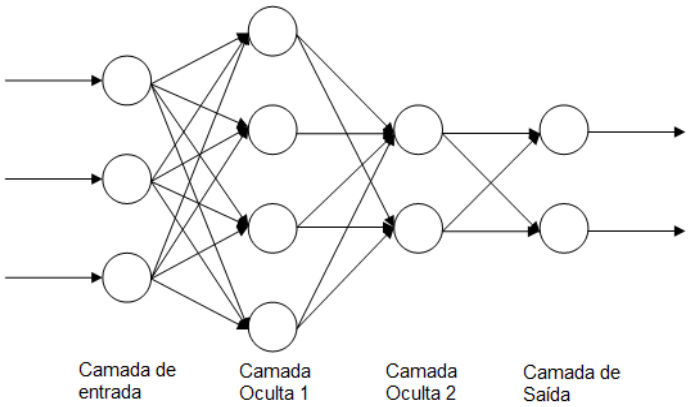


Figura 2.3: Rede neural. Fonte: próprio autor.

- Saídas desejadas: são as classes de saída. É possível ter para o mesmo problema um

número de saídas variáveis, dependendo de como é o projeto da rede e quantas classes de objetos existem. Aqui o problema é transcrito para uma configuração onde determinados neurônios da última camada são ativados e outros não, dependendo da entrada fornecida.

Algo importante que deve-se notar aqui é que uma rede neural é supervisionada, ou seja, ao treinar sobre alguns dados, sempre tem-se de antemão para cada exemplo de entrada do conjunto de treinamento sua saída desejada correspondente.

Redes Neurais Artificiais não são totalmente determinísticas, ou seja, toda saída da rede é sempre associada à uma classe com um nível de certeza associado. Esse nível de certeza é o que garante ou não um convergência esperada para associação do objeto com sua classe, e pode ser associada com um ponto de corte de segurança.

Esse limiar de corte (*threshold*) aplicado a saída de uma rede pode retornar saídas difusas, ou seja, retornar mais de uma classificação, caso mais de uma classe atinja a porcentagem de probabilidade.

2.1.2 K-Means

K-Means é um algoritmo de agrupamento de dados, que embora não tenha em si capacidade de aprender, consegue associar características entre informações N-dimensionais (múltiplas características) e agrupá-las em K-Clusters, onde cada *cluster* possui elementos semelhantes entre si.

O objetivo da clusterização de dados, também conhecido como análise de *cluster*, é descobrir os agrupamentos naturais de um conjunto de padrões, pontos ou objetos (JAIN, 2010). O objetivo é dividir M pontos de N dimensões em K *clusters* de modo que a soma de quadrados dentro do *cluster* minimize (HARTIGAN; WONG, 1979).

Esse algoritmo é utilizado para agrupamento de objetos dispostos em um espaço N-dimensional. O algoritmo requer como entrada uma matriz de M pontos em N dimensões e uma matriz de K centros de *cluster* iniciais também de N dimensão (HARTIGAN; WONG, 1979).

Como exemplo, pode-se citar um ponto em um espaço Euclidiano bidimensional. Esse objeto possui características associada que são suas posições nas 2 dimensões (X e Y), conforme Figura 2.4. Outro exemplo poderia ser o agrupamento de pessoas de acordo com suas idades, sexo e altura (dimensões do problema). Portanto o algoritmo tem o propósito de agrupar um grupo de objetos de acordo com determinadas características comuns a todos os objetos do

grupo.

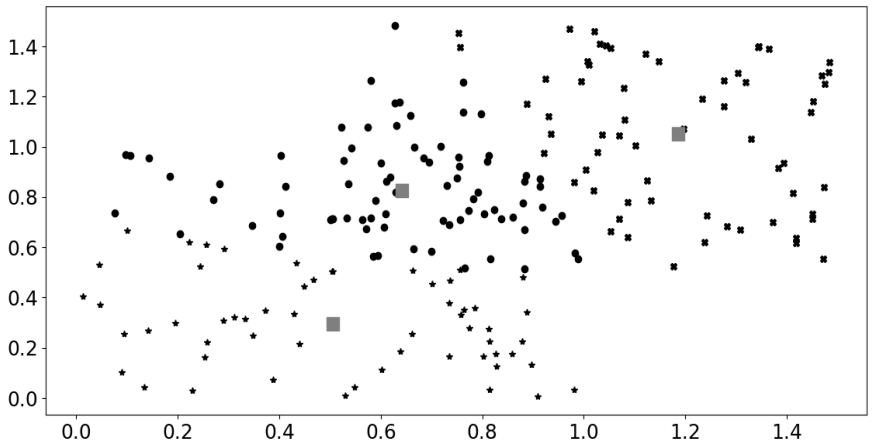


Figura 2.4: Exemplo de clusters resultantes da execução do algoritmo K-means. Fonte: próprio autor.

O algoritmo 1 demonstra um pseudo-código do algoritmo de K-Means que segue os passos:

1. Parametrização: número C de *clusters* (classes), conjunto de D objetos N-dimensionais.

2. Inicialização: inicializa a posição dos C *clusters* aleatoriamente no espaço N-dimensional

3. Cálculo de distância: calcula a distância de cada objeto em relação aos C *clusters*. Para o algoritmo K-means é utilizada a distância euclidiana (equação (2.3)), onde p representa um objeto e q o *cluster* em relação ao qual se está calculando distância, sendo que N é o número de dimensões dos dados.

$$\sqrt{\sum_{i=1}^{N}(p_i - q_i)^2} \tag{2.3}$$

4. Cálculo do agrupamento: agrupa cada objeto ao *cluster* de menor distância calculado no passo 3.

5. Recálculo do centroide: recalcula a posição do centroide do *cluster j* como sendo a média das posições de todos os objetos *d* daquele *cluster*, conforme a equação (2.4).

$$\frac{1}{D}\sum_{i=1}^{D} d(j) \tag{2.4}$$

6. Repete os passos 3 a 5 até que nenhum objeto mude mais de cluster.

Ao final o resultado são N *clusters* com os objetos mais semelhantes entre si, conforme Figura 2.4. É interessante ressaltar que assim como a Rede Neural, o algoritmo K-means também possui um número pré-definido de classes (nesse caso *clusters*), e a confiabilidade de sua classificação de determinado objeto pode ser associado a quão próximo esse objeto está de seu *cluster*.

Porém se existir um limiar de corte da distância calculada, possivelmente para alguns objetos, haveria mais de um cluster como possibilidade de classificação, o que retornaria uma classificação difusa das entradas. Por exemplo, a Figura 2.5 mostra o raio de alcance definido para um ponto que se quer classificar. Nesse caso, tanto o centróide C1 quanto C2 são possibilidades de classificação.

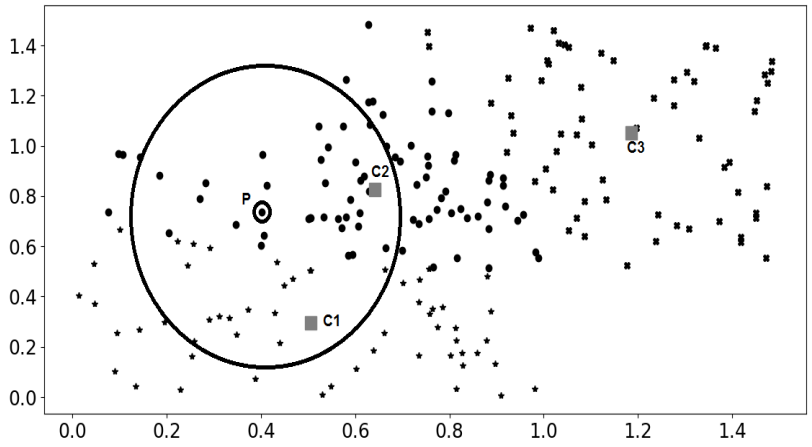


Figura 2.5: Exemplo de clusters resultantes da execução do algoritmo K-Means com duas possibilidades de classificação. Fonte: próprio autor.

Algorithm 1 Algoritmo K-Means

```
1:  procedure KMEANS(numero_grupos, pontos)
2:      grupos ← criar_grupos(numero_grupos)
3:      mudanca ← Verdadeiro
4:      for k ∈ grupos do
5:          k.posicao ← gerar_posicao_aleatoria()
6:      while mudanca == Verdadeiro do
7:          mudanca ← Falso
8:          for p ∈ pontos do
9:              for k ∈ grupos do
10:                 funcao_distancia(p.position, k.position)
11:             if p.grupo ≠ grupo_mais_proximo then
12:                 p.grupo = grupo_mais_proximo
13:                 mudanca ← Verdadeiro
14:         for k ∈ grupos do
15:             for p ∈ k.pontos do
16:                 media ← media + p.posicao
17:             media ← media/k.numero_de_pontos
18:             k.centroide ← media
        return grupos
```

2.2 ALGORITMOS PARA CÁLCULO DE DISTÂNCIA

Distância é uma medida de dissimilaridade entre objetos, o resultado do cálculo de distância entre dois objetos nos retorna o quão distante eles estão. Quanto maior for o valor da distância mais diferentes são esses objetos. A distância entre dois objetos é medida a partir das suas propriedades. Por exemplo, as propriedades poderiam ser idade e altura de pessoas, como mostra a Figura 2.6. Nesse caso, as pessoas são representadas em um espaço bidimensional, e a distância euclidiana informa o quão dissimilares duas pessoas são. Como mostra o exemplo, P2 é mais similar a P1 do que a P3.

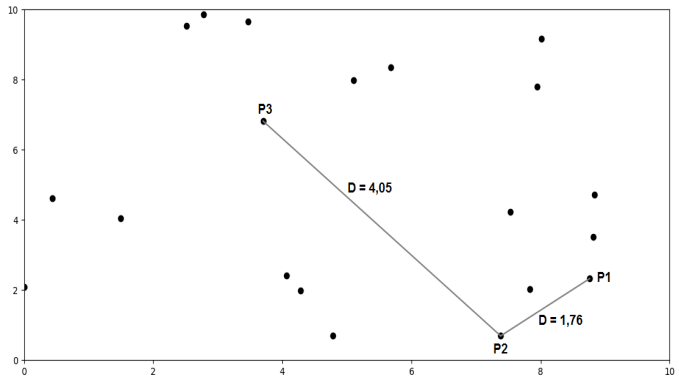


Figura 2.6: Distância entre dois objetos. Fonte: próprio autor.

No exemplo descrito, apenas duas propriedades são usadas, o que permite a representação dos objetos em um espaço bi-dimensional. No entanto, a distância pode ser calculada sobre um número variável de propriedades. Levando a uma representação dos objetos em um espaço n-dimensional. Segundo (SCHWEIZER; SKLAR, 1960) uma distância d deve obedecer as seguintes regras:

1. Simetria: $d(x,y) = d(y,x)$

   A distância medida de um objeto x para um objeto y com o mesmo cálculo de distância deve resultar no mesmo valor.

2. Não negatividade: $0 < d(x,y) < \infty$, $x \neq y$

   Também é definido que não existe distâncias negativas.

3. Igualdade: $d(x,y) = 0$, $x = y$

   O limite é alcançado quando os objetos são iguais, portando sua posição no espaço métrico N-dimensional é a mesma e a distância será 0.

4. Inegualdade Triangular: $d(x,y) \leq d(x,z) + d(z,y)$

   A propriedade da inegualdade triangular define que a distância calculada entre dois objetos x e y é sempre menor ou igual que a soma das distâncias de x e y para um 3º objeto z no mesmo espaço métrico.

2.2.1 Algoritmo de Distância de Levenshtein

Também conhecido como distância de edição, o algoritmo de Levenshtein determina a distância entre textos.

A correspondência aproximada de cadeias de caracteres é um problema recorrente em muitos ramos da ciência da computação, com aplicações para busca de textos, biologia computacional, reconhecimento de padrões, processamento de sinal, etc (BAEZA-YATES; NAVARRO, 1998).

Basicamente o algoritmo recebe como entrada duas cadeias de caracteres e retorna a distância entre elas. Essa distância representa a contagem de operações de adição, remoção ou substituição de caracteres que deve ser realizada para que uma palavra se torne outra.

O algoritmo trabalha com a criação de uma tabela onde cada célula é preenchida com a ajuda de duas funções, uma função de custo e uma função de mínimo. Um exemplo é ilustrado na Tabela 2.1. Dois textos são comparados: a palavra 'facil' e a palavra 'dificil'. As células da matriz são preenchidas com base na comparação entre as letras dessas palavras. O valor da célula referente à última linha e coluna mostra o número de operações de edição necessário para transformar uma palavra em outra. Ou seja, três operações são necessárias para transformar 'facil' em 'dificil'.

		F	A	C	I	L
	0	1	2	3	4	5
D	1	1	2	3	4	5
I	2	2	2	3	3	4
F	3	2	3	3	4	4
I	4	3	3	4	3	4
C	5	4	4	3	4	4
I	6	5	5	4	3	4
L	7	6	6	5	4	3

Tabela 2.1: Tabela do algoritmo de distância de Levenshtein

A função de custo compara os caracteres C1 e C2, de acordo com a Equação 2.5. Caso sejam iguais, o custo é 0. Se forem diferentes, o custo é 1.

$$\begin{cases} C1 \neq C2 \rightarrow 1 \\ C1 \equiv C2 \rightarrow 0 \end{cases} \tag{2.5}$$

As células da matriz são preenchidas de acordo com a Equação 2.6. Usa-se o valor mínimo entre três valores: o valor da célula imediatamente acima da célula atual somada de um, o valor da célula imediatamente a esquerda somada de um, e o valor da da célula na diagonal superior esquerda somada do resultado da função de custo.

$$\text{Mínimo de } \begin{cases} matrix[i, j-1] + 1 \\ matrix[i-1, j] + 1 \\ matrix[i-1, j-1] + custo \end{cases} \tag{2.6}$$

Algorithm 2 Distância de Levenshtein

```
1:  procedure LEVENSHTEIN(i, j)
2:      i ← 0
3:      while i < string2.tamanho do
4:          j ← 0
5:          while j < string1.tamanho do
6:              custo ← CUSTO(i − 1, j − 1)
7:              minimo ← MINIMO(i, j)
8:              matriz[i, j] ← minimo
9:              j++
10:         i++
11:     return matriz[string2.tamanho][string1.tamanho]

1:  procedure CUSTO(i, j)
2:      if string1.posicao(i) == string2.posicao(j) then
3:          return 0
        return 1

1:  procedure MINIMO(i, j, custo)
2:      v1 ← matriz[i − 1][j] + 1
3:      v2 ← matriz[i][j − 1] + 1
4:      v3 ← matriz[i − 1][j − 1] + custo
5:      return valor_minimo(v1, v2, v3)
```

2.2.2 Distância de Hamming

A distância de Hamming também retorna a quantidade de operações a serem feitas sobre uma cadeia de símbolos (caracteres) para converte-la em uma segunda cadeia de símbolos. Porém, a distância de Hamming aceita apenas a substituição de caracteres como operação. Dessa forma, o algoritmo se aplica apenas a cadeias de caracteres de mesmo tamanho.

A distância de Hamming é o somatório de bits que diferem comparando-se cada posição de duas cadeias de bits de mesmo tamanho (HAMMING, 1950). Para exemplificar, tomam-se duas palavras de mesmo tamanho, "elabore"e "melhore", e verifica-se posição a posição essas palavras conforme a tabela 2.2.

Posição							Distância
0	1	2	3	4	5	6	
M	**E**	**L**	**H**	O	R	E	4
E	**L**	**A**	**B**	O	R	E	

Tabela 2.2: Distância de Hamming entre duas palavras

Salienta-se que não é necessário que os caracteres a serem substituídos sejam consecutivos, permitindo a existência de caracteres ou sequencias de caracteres a serem substituídos em separado ao longo da palavra.

Algorithm 3 Distância de Hamming

```
1: procedure HAMMING(string1, string2, tamanho)
2:     if string1.tamanho ≠ string2.tamanho then
3:         return null
4:     distancia ← 0
5:     for i ∈ intervalo(0, tamanho) do
6:         if string1.posicao(i) ≠ string2.posicao(i) then
7:             distancia ← distancia + 1
8:     return distance
```

A distância entre objetos mede o quão dissimilares dois objetos são. Porém, em muitos casos é mais adequado usar algum valor que reflita a similaridade entre objetos.

2.3 FUNÇÕES DE SIMILARIDADE

As funções de distância retornam quantidades de operações a serem realizadas para transformar uma palavra em outra, a quantidade de operações se torna a distância entre duas palavras, porem não há um limite de operações a serem realizadas, ou seja, não pode-se dizer que uma palavra está 100% de distância de outra palavra, não há uma escala. Para isso existem as funções de similaridade. Medidas de similaridade têm sido amplamente utilizadas na classificação de textos e algoritmos de agrupamento (LIN; JIANG; LEE, 2014).

A similaridade pode ser medida entre diversos tipos de objetos. Como o foco desse trabalho é o reconhecimento e transcrição de palavras, apenas funções que tratam de similaridade em cadeias de caracteres são abordadas. Quantificar a semelhança entre cadeias de caracteres é um importante problema científico que atraiu muito interesse porque a informação chave pode ser expressa por sequências simbólicas em muitos aplicativos como recuperação de texto, processamento de sinal e computação biológica (YUJIAN; BO, 2007).

Existem maneiras de medir similaridade entre duas palavras. Dentre elas estão os coeficientes de Dice e Jaccard além da SMC (*Sequence Most Common*).

- **Coeficiente de Dice**:

  Na fórmula da equação (2.7) *String1* e *String2* são cadeias de caracteres entre as quais se quer calcular similaridade. A função *tok* representa a quebra da cadeia em uma lista de caracteres. Portanto o resultado da função de similaridade é 2 vezes o número de caracteres iguais nas duas palavras dividido pelo tamanho das duas palavras juntas. É possível perceber que a semelhança será de 100% nesse caso quando as duas strings forem iguais.

$$Dice = \frac{2*|tok(String1) \bigcap tok(String2)|}{|tok(String1)|+|tok(String2)|} \tag{2.7}$$

- **Coeficiente de Jaccard**:

  O coeficiente de Jaccard é uma medida usada para comparar a semelhança e a diferença de conjuntos de objetos. O coeficiente Jaccard mede a similaridade entre conjuntos de amostras finitas e é definido como o tamanho da interseção dividido pelo tamanho da união dos conjuntos de objetos (retirando-se os objetos pertencentes a interseção) conforme equação (2.8).

Na equação 2.8 a função tok representa a quebra da string em um conjunto de caracteres, e as strings 1 e 2 representam as duas cadeias de caracteres das quais se está medindo similaridade:

$$Jaccard = \frac{|tok(String1) \bigcap tok(String2)|}{|tok(String1)|+|tok(String2)|-|tok(String1) \bigcap tok(String2)|} \tag{2.8}$$

- **LCS:** Longest Commom Subsequence (subsequencia comum mais longa) é uma medida de similaridade entre duas sequências de símbolos. De forma simples ela é uma sequência de símbolos comuns entre duas cadeias de caracteres que aparecem da esquerda pra direita, e não precisam ser consecutivos.

  No tabela 2.3 há duas sequências de caracteres e uma subsequencia comum mais longa entre elas denotada pelas letras marcadas em cada palavra. Nota-se que as sequências não necessitam serem consecutivas na palavras, apenas precisam ser comuns entre ambas.

S1:	AA**ACC**G**T**
S2:	**CA**CC**CCT**
LCS:	**ACCT**

Tabela 2.3: Tabela de LCS

Basicamente o valor de similaridade do LCS é dado pelo tamanho da sequência comum, dividido pelo tamanho da menor palavra da comparação conforme a Equação (2.9). Isso porque a subsequência comum é no máximo do mesmo tamanho da menor palavra e, caso seja igual, retorna uma similaridade de 100%.

$$LCS = \frac{|subsequancia\_comum|}{|menor\_palavra|} \tag{2.9}$$

- **Normalização de cálculos de distância**: O valor resultado de uma função de cálculo de distância pode ser normalizado realizando-se uma relação entre o resultado do cálculo e o tamanho da maior palavra comparada. Realizando uma análise com o pior caso em distância, onde as palavras comparadas são totalmente diferentes, ou seja, todos seus caracteres são diferentes, pode-se perceber que a quantidade de operações será no máximo

a quantidade de caracteres da maior palavra, seja adicionando, alterando ou removendo caracteres.

Uma possível solução para normalizar a distância por exemplo, de Levenshtein, é a divisão do valor resultante pelo tamanho da maior palavra conforme a equação (2.10).

$$Norm = \frac{|distancia\_levenshtein|}{|maior\_palavra|} \tag{2.10}$$

2.4 ÍNDICES DE BUSCA EM ESPAÇO MÉTRICO

Um espaço métrico é definido por uma função de distância D, e essa função é definida pelas propriedades de identidade, não negatividade, simetria e inegualdade triangular (BOZKAYA; OZSOYOGLU, 1997).

Pode-se ter em um espaço objetos distribuídos em N dimensões, e elas é que definem as características de seus objetos, e é em relação a essas dimensões que são calculadas as distâncias entre eles.

Por exemplo, caso os objetos sejam representador por coordenadas espaciais, pode-se medir a distâncias entre eles a partir da distância euclidiana. Caso os objetos sejam palavras, é possível usar as medidas de distância apresentadas na seção 2.2.

A partir do momento que os objetos estão distribuídos em um espaço N dimensional é possível realizar buscas por determinados objetos de duas principais formas. A primeira está em iniciar a busca por um determinado ponto e encontrar os M objetos mais próximos, ou como segunda opção capturar todos os objetos dentro de um limite de alcance a partir do ponto onde está a busca, sempre levando em conta as N dimensões do espaço.

Essa busca é realizada através de estruturas de índice que usam o cálculo de distância para diminuir o número de entradas a serem comparadas. Nas seções seguintes são apresentadas algumas das estruturas que utilizam cálculos de distância como meio de indexação e busca. A Figura 2.7 mostra de forma geral como funciona a busca de um objeto através de uma estrutura indexada.

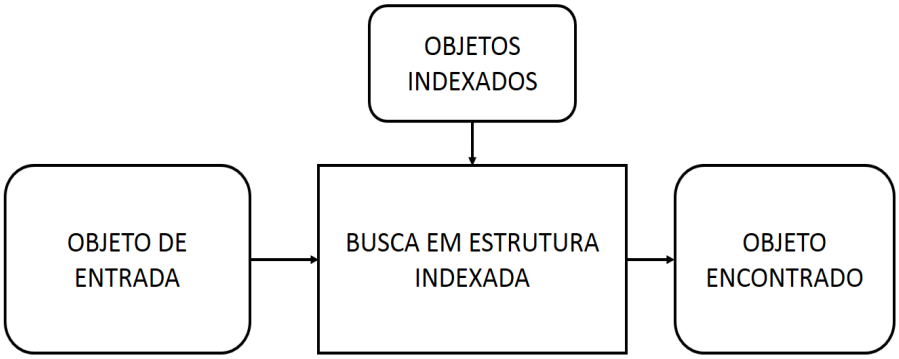


Figura 2.7: Módulo Busca Indexada. Fonte: próprio autor.

2.4.1 Burkhard Keller Tree

É uma estrutura de árvore que pode armazenar palavras e a adição e busca de seus elementos é baseada no cálculo de um algoritmo de distância, como por exemplo a distância de Levenshtein.

A Figura 2.8 mostra o formato de uma árvore BK-Tree construída com 5 palavras (some, same, soft, salmon, soda, mole). Nota-se que as as palavras são dispostas nos respectivos nodos conforme suas distâncias para com as palavras nos nodos mais acima e semelhantes a ela. E nas arestas se localizam as distâncias calculadas entre o nodo filho e o nodo pai.

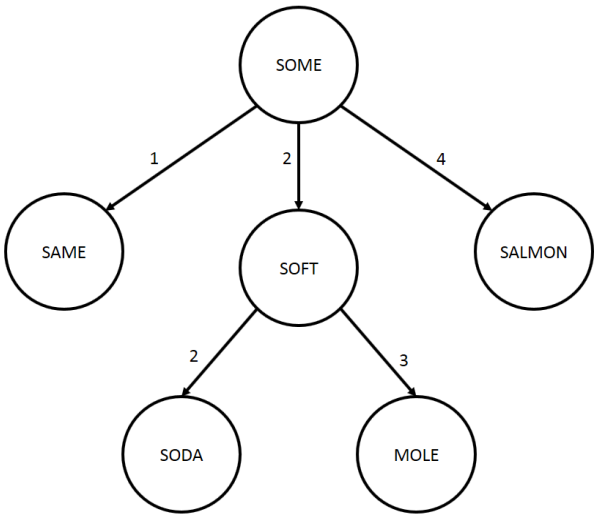


Figura 2.8: BK-Tree construída. Fonte: próprio autor.

O Algoritmo 4 descreve como ocorre a adição de um elemento na árvore. Inicialmente a palavra a ser indexada é comparada com a palavra na raiz da árvore pelo cálculo da distância

de Levenshtein. O valor resultante é procurado nas arestas de distância entre a raiz e seus nodos filhos. Caso a distância já exista, o algoritmo recursivamente chama novamente a função passando a palavra a ser inserida e o nodo filho que possui a mesma distância para o nodo pai. Caso não exista nenhum nodo filho que tenha a distância calculada, então um novo nodo é criado para a palavra, tendo como pai o nodo atual.

Algorithm 4 Função de adição de novo objeto na árvore BK-Tree

```
1: procedure ADD(nodo, objeto)
2:     l ← Distancia(nodo.objeto, objeto)
3:     distancia ← l.distancia()
4:     for nodofilho ∈ nodo.filhos do
5:         if nodofilho.distancia == distancia then
6:             ADD(nodofilho, objeto)
7:             return null
8:     nodo.add(objeto, distancia)
```

O Algoritmo 5 descreve como ocorre a busca. Inicialmente a palavra buscada é comparada com a armazenada na raiz. Essa distância deve ser menor que o valor de vizinhança N. Se for menor, o resultado é adicionado à lista de resultados como uma possível solução (linha 5). Então a busca é passada aos nodos filhos. Os nós filhos são visitados apenas se for possível encontrar a resposta seguindo por esse ramo. Para isso, a distância do filho para o pai deve estar dentro de um intervalo que satisfaça a inegualdade triangular.

Algorithm 5 Função de busca em BK-Tree

```
1: procedure BUSCA(nodo, objeto)
2:     l ← Distancia(nodo.objeto, objeto)
3:     distancia = l.distancia()
4:     if distance < N then
5:         resultados.add(distancia, nodo.objeto)
6:     for nodofilho ∈ nodo.filhos do
7:         if distancia − N <= nodo.distancia & nodo.distancia <= distancia + N
   then
8:             Busca(nodofilho, objeto)
```

2.4.2 *Vantage Point Tree*

Assim como a BK-Tree, a VP-Tree também é um tipo de estrutura em árvore, que segrega os dados em um espaço métrico pela escolha de um ponto de vantagem (*Vantage Point* ou pivô). A VP-Tree é do tipo binária.

Cada nodo na árvore possui as seguintes informações:

- Points: lista de pontos contidos
- VP: *Vantage Point* escolhido da lista de pontos
- Radius: um raio dentro do qual está contido os pontos da lista
- Inside: pontos dentro do raio
- Outside: pontos fora do raio

O algoritmo 6 constrói a *Vantage Point Tree* seguindo os passo abaixo:

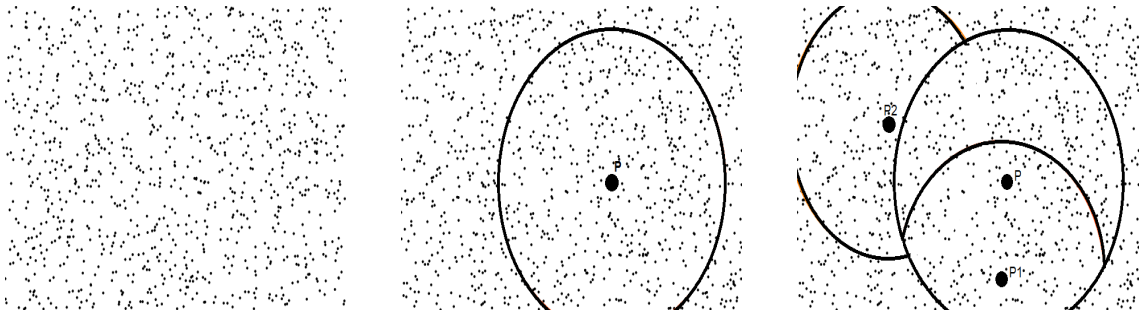
1. Primeiramente um ponto de vantagem é escolhido entre a listagem de pontos disponíveis.
2. Em seguida é definido um raio de alcance partindo do ponto de vantagem. Esse raio de alcance é calculado como sendo a média da distância do ponto de vantagem, para os demais pontos.
3. Os pontos dentro e fora do raio de alcance são adicionados em listas distintas.
4. Recursivamente os passos 1, 2 e 3 são executados para a lista de pontos dentro do raio, e para a lista fora do raio.

Algorithm 6 Algoritmo Vantage Point Tree

```
1:  procedure VPTREE(pontos)
2:      ponto_de_vantagem ← selecionar_ponto_aleatorio(pontos)
3:      distancia_media ← 0
4:      for p ∈ pontos do
5:          distancia ← funcao_distancia(p.posicao, ponto_de_vantagem.posicao)
6:          distancia_media ← distancia_media + distancia
7:      distancia_media ← distancia_media/pontos.tamanho
8:      pontos_dentro ← Lista()
9:      pontos_fora ← Lista()
10:     for p ∈ pontos do
11:         distancia ← funcao_distancia(p.posicao, ponto_de_vantagem.posicao)
12:         if distancia < raio then
13:             points_dentro.add(p)
14:         else
15:             points_fora.add(p)
16:     VPTREE(points_dentro)
17:     VPTREE(points_fora)
```

A Figura 2.9 mostra o processo de criação da *Vantage Point Tree* com pontos dispostos em um espaço bidimensional (Figura 2.9(a)). Na Figura 2.9(b) selecionam-se aleatoriamente um ponto de vantagem (ponto P na figura) é calculado um raio de alcance.

Após esse processo existem os pontos dentro do raio e fora dele (Figura 2.9(c)). O processo é recursivamente chamado para o grupo de pontos dentro e fora do círculo até que não sejam geradas novos nodos de árvore que estejam vazios.



(a) Pontos dispostos em um espaço 2D

(b) Ponto de vantagem P e o raio calculado

(c) construindo recursivamente a árvore

Figura 2.9: Exemplo de saída do algoritmo de indexação *Vantage Point*. Fonte: próprio autor.

3 ALGORITMOS DESENVOLVIDOS

A solução desenvolvida está dividida basicamente em dois módulos separados, um módulo de reconhecimento e um módulo de busca, que realizam respectivamente o reconhecimento de padrões em caracteres, e a busca da palavra mais semelhante a partir de um dicionário.

O fluxo geral da execução da implementação atua de acordo com a Figura 3.1.

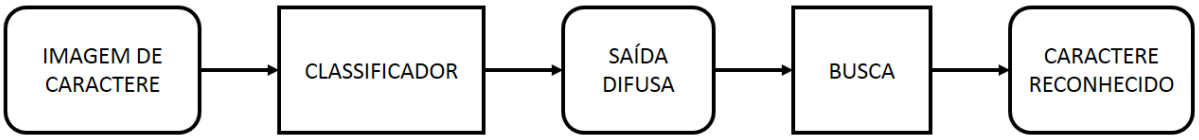


Figura 3.1: Fluxo de execução geral. Fonte: próprio autor.

3.1 MÓDULO CLASSIFICADOR

O módulo classificador (descrito na Figura 3.2) realiza a classificação da imagem do caractere do alfabeto em 26 classes (26 letras). Juntamente com as entradas de imagens dos caracteres de cada palavra será inserido um parâmetro de *threshold* que terá a função de, uma vez treinada a rede, eliminar todas as letras cujo escore tenha ficado abaixo do ponto de corte. Dessa maneira a imagem de uma letra "A"por exemplo pode ser classificada pela rede como um "A"ou um "P"caso essas duas letras resultem em convergência acima do *threshold* passado para a rede.

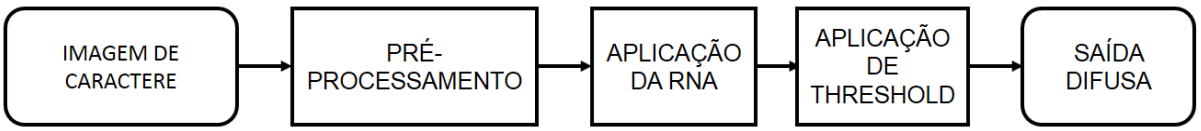


Figura 3.2: Fluxo de funcionamento do módulo de reconhecimento de padrões. Fonte: próprio autor.

3.1.1 Pré-processamento dos dados

As imagens que foram utilizadas do *dataset* foram as imagens de caracteres manuscritos, pois o software desenvolvido não leva em consideração imagens coloridas, ou em ângulos distorcidos. Porém alguns pré-processamentos ainda se fazem necessários para padronizar a entrada do módulo reconhecedor.

3.1.1.1 Redução de dimensionalidade

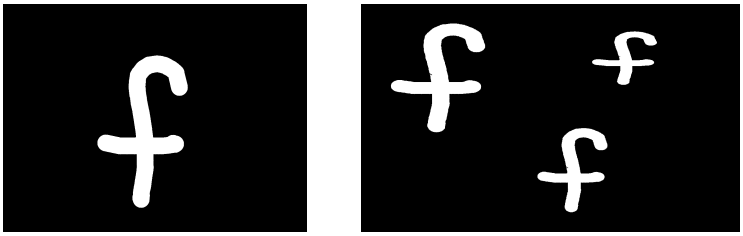
Toda imagem é composta basicamente de 3 canais de cores que unidas formam as demais cores visíveis ao olho humano. São elas vermelho, verde e azul. Com essas três cores é possível formar as demais cores do espectro visível.

Imagens digitais também são compostas composta por esses 3 canais de cores em cada pixel, e cada canal possui intensidade de 0 a 255. Porém como o trabalho realizado é com imagens de caracteres manuscritos com intensidades em preto e branco, não é necessário que se use a intensidade de todos os três canais. Para isso as imagens serão processadas com o que é chamado de *escala em cinza* (*greyscale*). Isso fará com que o pixel seja formado por apenas um canal e a imagem passa a ser uma matriz bidimensional. A transformação segue a equação (3.1).

$$greyscale(pixeis\_img) = \sum_{x=0}^{W} \sum_{y=0}^{H} \frac{\left(\sum_{canal=0}^{3} canal\_rgb\right)}{3} \tag{3.1}$$

3.1.1.2 Corte de bordas

Um dos problema com reconhecimento de imagens diz respeito à escala de um objeto presente na imagem. É consenso que as imagens mostradas nas Figuras 3.3(a) e 3.3(b) representam a mesma imagem. A única diferença é sua escala na imagem.

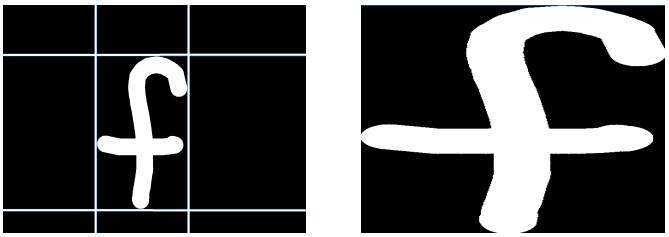


(a) Imagem em escala grande da letra F

(b) Diferente escala do mesmo objeto

Figura 3.3: Imagens em diferentes escalas. Fonte: próprio autor.

A primeira operação de pré-processamento aplicado nas imagens é o corte das bordas, deixando apenas uma região com a parte "útil"da imagem, ou seja, onde realmente se localiza o objeto. Esse corte é feito partindo uma linha de cada um dos lados da imagem em direção ao lado oposto. Quando há uma troca brusca na coloração de um pixel, a linha de corte é terminada.



(a) Linhas de corte (b) Letra F com as bordas cortadas

Figura 3.4: Processo de corte de bordas. Fonte: próprio autor.

3.1.1.3 Redimensionamento

Quando um corte é realizado em uma imagem, a área de pixeis desta imagem é que está diminuindo, ou seja, ocorre o redimensionando dela. Portanto para que todas as imagens estejam no mesmo padrão de tamanho, é necessário redimensionar essas imagens. O redimensionamento realiza um processo igual ao ilustrado na Figura 3.5, onde a imagem é redimensionada para o tamanho padrão mantendo o formato da figura.

(a) Letra com bordas cortadas (14x14) (b) Figura com bordas cortadas redimensionada (28x28)

Figura 3.5: Processo de redimensionamento após corte das bordas. Fonte: próprio autor.

3.1.2 Rede Neural Artificial

O classificador usado é uma Rede Neural Artificial. Seu papel é reconhecer a imagem do caractere manuscrito e classifica-lo conforme as 26 classes (letras do alfabeto). A rede neural foi construída com o auxilio da biblioteca PyBrain (SCHAUL et al., 2010) .

Para treinamento da rede foram utilizados 104 exemplos (imagens), sendo 4 exemplos de treinamento por classe (letra), e para teste o mesmo modelo, com diferentes exemplos.

Para cada palavra existe um conjunto de imagens com o desenho manuscrito dos caracteres dessa palavra. Depois de treinada, a rede passa a receber imagens e devolver o caractere

correspondente. Unindo-se a classificação de cada imagem de caractere da palavra é formada então a palavra reconhecida pela rede.

3.1.3 Aplicação do Threshold

O *threshold* é um ponto de corte que atua na saída da rede, excluindo resultados que não atinjam um determinado valor definido. Ele é responsável por gerar a difusão na saída da rede, ou seja, todas as classificações que obtiverem um valor maior ou igual ao valor de *threshold* são consideradas, gerando mais de uma possibilidade, uma difusão.

Um exemplo de como ocorre a difusão em uma letra de uma palavra qualquer é mostrado na Figura 3.6. No exemplo, a imagem da letra minúscula manuscrita L é passada pela rede, sendo classificada entre R, P e L, sendo que ao final um *threshold* de 90% é passado pelos valores de convergência de saída, e as classes que atingirem esse valor de *threshold* são tomadas como possibilidades de serem a letra correta. No caso as letras R e L são as possibilidades de serem as letras corretas para a imagem passada.

É importante lembrar que a classificação da rede poderá ser difusa nos caracteres de uma palavra classificada pela rede. A saída difusa será escrita em um arquivo texto seguindo a regra definida pela expressão regular (c|c+)*, onde c representa um caractere. Essa saída será posteriormente lida e decodificada pelo módulo de busca. O módulo irá retornar então, a palavra mais próxima à palavra classificada pela rede.

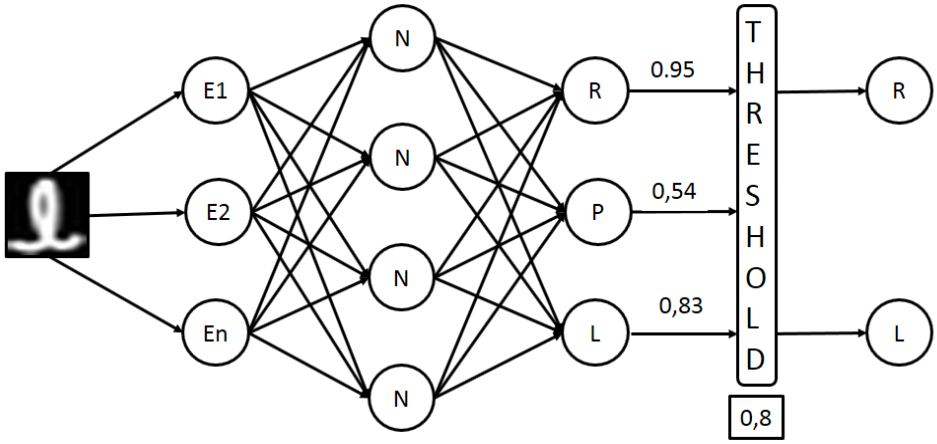


Figura 3.6: Difusão em uma rede neural. Fonte: próprio autor.

Na tabela 3.1 são mostradas diferentes formas de palavras difusas aceitas pela nova versão do algoritmo. Na primeira palavra difusa existem duas possibilidades de caracteres na

3º posição, dessa forma pode-se formar tanto a palavra CAMA quanto a palavra CASA. Na segunda possibilidade existem novamente duas possibilidades de caracteres para a 1º posição, porém pode-se notar que a segunda possibilidade com a letra I, a palavra formada não existe, o que valida a ideia de busca da palavra difusa em um dicionário. Como último exemplo há uma dupla difusão ao longo da palavra o que possibilita a criação de quatro combinações (anagramas) na palavra.

Palavra Difusa
CA{SM}A
{CI}ADEIRA
{PR}AI{NM}EL

Tabela 3.1: Exemplo de palavras difusas

3.2 MÓDULO DE BUSCA

A entrada do módulo de busca será uma cadeia de caracteres com possibilidades de difusão. Como exemplo de entrada do módulo pode haver a palavra *{mn}adeira* onde "m"e "n"foram possibilidades de letras para a rede que ultrapassaram o limiar de corte. No módulo de busca apenas a palavra 'madeira' seria encontrada, uma vez que a outra opção ('nadeira') não deve existir no dicionário.

A diferença no módulo de busca é de que tanto os algoritmos de distância de Levenshtein e a estrutura da árvore BK-Tree estão modificadas para aceitar tais entradas difusas (Figura 3.7), e realizar apenas uma busca por entrada, e não várias buscas por entrada (cada uma representando uma possibilidade).

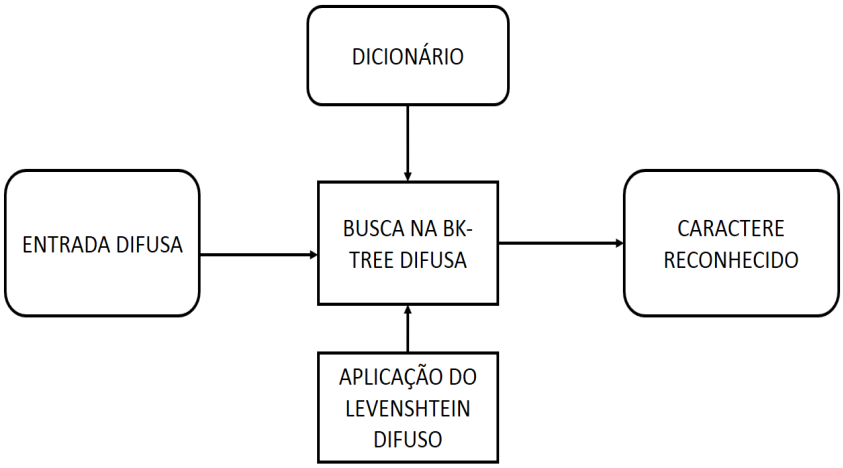


Figura 3.7: Fluxo de funcionamento do módulo de busca indexada. Fonte: próprio autor.

3.2.1 Levenshtein Difuso

Conforme explicado na subseção 2.2.1, o algoritmo de Levenshtein serve como uma ferramenta para o cálculo de distância entre duas palavras, retornando o número de operações necessárias para transformar uma palavra em outra. Esta subseção mostra a modificação do algoritmo de Levenshtein permitindo a aceitação de entrada de palavras difusas.

A diferença na nova versão é de que para cada célula será dado um valor de **mínimo** e **máximo** em relação a diferença de duas letras. Para tanto, as funções de custo e mínimo apresentadas na subseção 2.2.1, foram modificadas para aceitar o novo modelo.

A função de custo retorna na versão normal do algoritmo o valor 0 caso as letras comparadas sejam iguais e o valor 1 caso sejam diferentes. Na versão modificado, o mínimo recebe zero se alguma das difusões corresponde ao caractere comparado. Já o máximo recebe um caso alguma das difusões não corresponda ao caractere comparado. A tabela 3.2 demonstra cada caso.

Casos	Letra 1	Letra 2
Caso 1: retorno [0,0]	C	C
Caso 2: retorno [1,1]	C	L
Caso 3: retorno [0,1]	C	{C,L}

Tabela 3.2: Casos de retorno da função de Custo

A Tabela 3.3 exemplifica cada um dos casos listados com sua ocorrência através do

cálculo de distância de Levenshtein Difuso entre as palavras 'PAZ' e 'P{AC}{CZ}'.

		P	**A**	**Z**
	0,0	1,1	2,2	3,3
P	1,1	0,0	1,1	1,1
{AC}	2,2	1,1	0,1	1,1
{CZ}	3,3	1,1	1,1	0,1

Tabela 3.3: Exemplo da função de custo para difusão em palavras

A função de mínimo também é modificada para aceitar os valores de mínimo e máximo criados pela função de custo em cada célula. Ela será chamada uma vez para calcular o valor de mínimo usando os valores de mínimo de cada célula, e outra vez utilizando os valores de máximo.

A Tabela 3.4 mostra como o custo calculado (à esquerda) é usado par gerar os valores de mínimo (à direita). A última célula da tabela indica que a palavra difusa está no mínimo a 0 operações de distância da outra palavra e no máximo a 2 operações de distância.

		P	**A**	**Z**
	0,0	1,1	2,2	3,3
P	1,1	0,0	1,1	1,1
{AC}	2,2	1,1	0,1	1,1
{CZ}	3,3	1,1	1,1	0,1

		P	**A**	**Z**
	0,0	1,1	2,2	3,3
P	1,1	0,0	1,1	1,1
{AS}	2,2	1,1	0,1	1,2
{CZ}	3,3	2,2	1,2	0,2

Tabela 3.4: Exemplo da função de mínimo para difusão em palavras

3.2.2 Busca em estrutura indexada

O cálculo da distância do algoritmo de Levenshtein Difuso retorna dois valores de distância, um valor de mínimo e um valor de máximo. Esses dois valores auxiliam na busca da palavra que mais se parece com essa palavra difusa. E para realizar a busca é preciso comparar essa palavra difusa com uma lista de outras palavras conhecidas.

Pode-se comparar a palavra difusa com um dicionário inteiro, palavra a palavra, e retornar as palavras mais próximas. O problema de fazer isso está no custo. O número de cálculos de distância seria tão grande quanto a quantidade de palavras no dicionário.

Uma possível solução é a indexação das palavras desse dicionário em uma estrutura de dados e a busca pela palavra mais próxima através desse índice. A subseção 3.2.2.1 demonstra a modificação da BK-Tree para aceitar entradas difusas, como parâmetro de busca.

3.2.2.1 Multiple Diffuse BK-Tree

A BK-Tree recebe uma palavra de entrada e retorna a palavra mais similar dentro de um intervalo de busca. Nessa versão modificada, chamada de *Multiple Diffuse BK-Tree* (MDB), a árvore é construída da mesma forma, porém sua busca será em um intervalo que cobre as distâncias máximas e mínimas do algoritmo de Levenshtein Difuso. A vantagem aqui está em que o algoritmo realiza uma única busca por palavra difusa, e não inúmeras buscas para todas as combinações da difusão da palavra.

O algoritmo 7 demonstra a execução da BK-Tree Difusa. Ela utiliza o algoritmo de Levenshtein Difuso para cálculo da distância entre as palavras indexadas na estrutura e a palavra de entrada (que pode ser difusa).

Algorithm 7 Função de busca em BK-Tree difusa

```
1: procedure BUSCA(nodo, objeto)
2:     l ← LevenshteinDifuso(nodo.objeto, objeto)
3:     dist = l.dist()
4:     if dist.min < N then
5:         resultados.add(dist, nodo.objeto)
6:     for nodofilho ∈ nodo.filhos do
7:         if dist.min − N <= nodo.dist & nodo.dist <= dist.max + N then
8:             Busca(nodofilho, objeto)
```

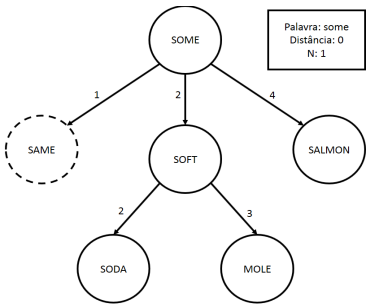
Pode-se notar que ainda é utilizado um valor N de vizinhança para a busca, porém, o intervalo mínimo e máximo não está mais entre [distância-N, distância+N] e sim entre [distância.mínimo-N, distância.máximo+N], conforme a linha sete do algoritmo.

Na linha quatro também existe uma mudança, pois tendo a distância como dois valores, o nodo para ser adicionado na lista de resultados deve ter seu valor de mínimo menor que o valor N de vizinhança.

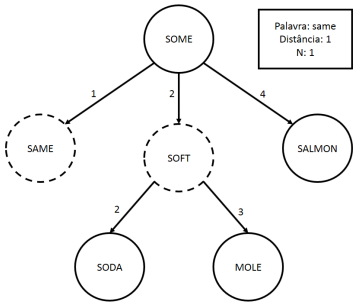
As Figuras 3.8(a) e 3.8(b) representam a busca de duas palavras, "same"e "some", na árvore BK-Tree original. A Figura 3.8(c) mostra a busca na BK-Tree difusa da palavra s{oa}me.

As distâncias indicadas em cada Figura são calculadas em relação a raiz SOME, e a

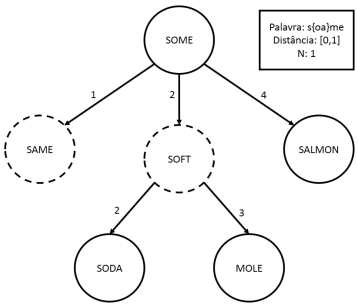
partir daí os filhos são avaliados para verificar se a pesquisa continua no nodo filho. Os nodos visitados nas duas primeiras árvores são ambos incluídos na MDB, apenas utilizando uma busca.



(a) Busca da palavra some na BK-Tree original



(b) Busca da palavra same na BK-Tree original



(c) Busca da palavra difusa soame na BK-Tree difusa

Figura 3.8: Funcionamento da busca em BK-Tree original. Fonte: próprio autor.

4 RESULTADOS

Nessa seção são apresentados os resultados da solução do problema proposto nesse trabalho. Os testes realizados utilizam como entrada frases aleatórias do livro "ADM: Princípios de administração"(WILLIAMS, 2009), contendo entre 60 e 120 palavras, que são classificadas e buscadas pelos respectivos módulos propostos e então retornar uma frase como saída.

4.1 PROCESSAMENTO DA FRASE

A Figura 4.1 demonstra como a frase de entrada é quebrada em uma lista de palavras que por sua vez é dividida em letras. Cada letra de cada palavra torna-se uma imagem com a ilustração de um caractere manuscrito que será passado para o módulo classificador iniciar todo o processo.

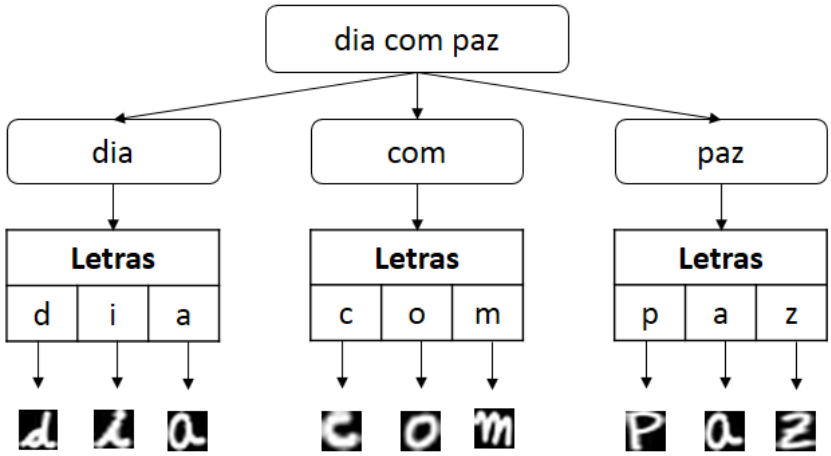


Figura 4.1: Fluxo de transformação da frase em entrada para a rede neural artificial. Fonte: próprio autor

4.2 DICIONÁRIO DE PALAVRAS

Para realizar buscas por palavras semelhantes, a estrutura BK-Tree (original e difusa) necessita indexar palavras para que a busca seja realmente efetiva. Portanto, um dicionário é indexado pela estrutura, sendo possível então encontrar palavras iguais ou semelhantes as pesquisadas.

Esse modelo de indexação facilita a troca de contexto para qualquer língua, pois o módulo classificador reconhece letra por letra (unidade textual), e a árvore BK-Tree utiliza os

objeto indexados para realizar a busca. Assim, pode-se trocar o conjunto de palavras a ser indexado e todo o programa passa a reconhecer palavras de uma outra língua. Para o presente trabalho foi feito uso de um conjunto de palavras em português, retiradas de um dicionário [1], contendo 32103 palavras.

4.3 BASE DE IMAGENS

A base de dados para caracteres manuscritos utilizada foi desenvolvida no MSR - Microsoft Research Lab na Índia. O dataset, chamado Chars74K, contém um conjunto de arquivos de imagens de caracteres fotografadas em ângulos, rotações, escalas e cores diferenciados, e um conjunto com imagens de caracteres manuscritos. O conjunto de dados impressos à mão foi capturado usando o tablet PC com a espessura da caneta ajustada para corresponder à espessura média encontrada nos quadros de informação escritos à mão (CAMPOS; BABU; VARMA, 2009).

A Figura 4.2 mostra um exemplo de imagem do caractere F minúsculo contido nessa base de dados. Abaixo algumas informações sobre a base de imagens:

- 74 mil imagens de caracteres
- 64 classes (0-9, A-Z, a-z)
- 55 caracteres por classe
- 3410 caracteres manuscritos usando um Tablet
- 55 voluntários
- Resolução: 1200x900

[1] Link: https://github.com/titoBouzout/Dictionaries/blob/master/Portuguese%20(Brazilian).dic

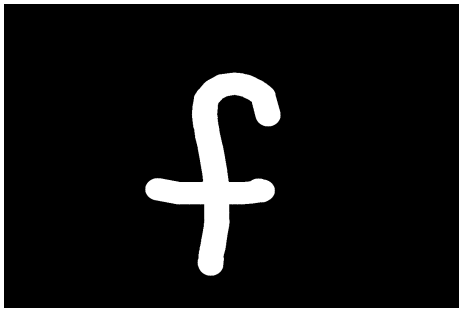


Figura 4.2: Exemplo de imagem de letra manuscrita contida no dataset Chars74K. Fonte: próprio autor

4.4 CONFIGURAÇÃO DA REDE NEURAL ARTIFICIAL

As imagens treinadas tem tamanho de 28x28 pixeis, sendo um pixel para cada entrada da rede. A rede está configurada conforme a Tabela 4.1.

Configuração	
Camadas	3
Entradas	784
Classes	26
Neurônios por Camada	784,150,26
Treinamento	
Epocas	300
Erro mínimo	0.1

Tabela 4.1: Configuração da Rede Neural

O treinamento foi configurado com 300 épocas de treinamento. Uma época descreve uma rodada de execução onde a rede treina uma vez cada exemplo de treinamento, e a cada execução de todos os exemplos de treinamento uma época a mais é contabilizada.

Como a quantidade de épocas não garante sozinha a convergência, também é fornecido um número de erro mínimo. Ou seja, a rede neural treina por no máximo 300 épocas ou até atingir um erro inferior ou igual a 0.1 (10%).

4.5 ALGORITMOS TESTADOS

A solução proposta no trabalho é chamada de **MDB (*Multiple Difuse BK-Tree)***, onde a saída difusa da rede será buscada utilizando a BK-Tree e Levenshtein difusos. Como forma de comparar a eficiência dessa nova solução foram implementados 2 algoritmos com formas simples de solução do problema.

- ***Baseline 1***: a primeira delas é a transformação da palavra difusa retornada pela rede em um conjunto de palavras que representa todos os anagramas (combinações) de possibilidades daquela palavra difusa. Por exemplo, a palavra difusa CA{SM}A, pode ser transformada na palavra CASA e na palavra CAMA. A ideia é de que o módulo de busca utilize o a BK-Tree original aliada ao cálculo de distância de Levenshtein original para buscar ambas as palavras. Essa solução gera provavelmente mais operações de cálculo de distância em relação ao MDB.

- ***Baseline 2***: retira o *threshold* da saída da rede, e retorna a classificação com a maior convergência (mesmo uma convergência baixa). Isso significa que a saída da rede será uma palavra completa sem difusões. O módulo de busca novamente é composto da BK-Tree e Levenshtein originais.

4.6 MEDIDAS DE AVALIAÇÃO

Os algoritmos são avaliados de acordo com os seguintes critérios de avaliação:

- **Taxa de acerto**: comparação entre a frase original e a frase de saída formada pelas palavras pesquisadas no módulo de busca. É realizada uma comparação palavra a palavra verificando a existência da palavra da frase de saída na frase original.

- **Número de cálculo de distância realizados**: o número de chamadas de cálculo de distância representa a quantidade de vezes que o algoritmo de Distância de Levenshtein original ou Levenshtein Difuso foi chamado pela implementação.

- **Tempo de execução**: o tempo de execução medido se refere ao tempo de busca de todas as palavras da frase na BK-Tree original ou difusa.

4.7 RESULTADOS

Nessa seção são apresentados os resultados dos testes aplicados aos algoritmos mencionados na Seção 4.5, em relação as medidas de avaliação descritas na seção 4.6.

A Figura 4.3 mostra a quantidade de palavras em cada frase de teste e a quantidade de difusões geradas em cada frase. Na média geral é possível perceber que em todas as frases ocorreram mais difusões do que palavras, o que significa que existem palavras com mais de uma difusão presente.

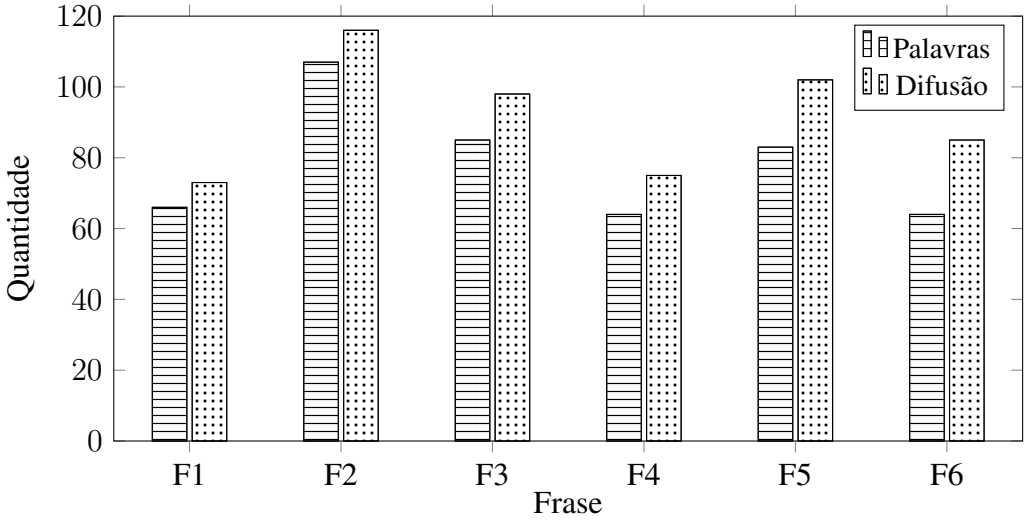


Figura 4.3: Número de palavras por frase e número de difusões por frase

O gráfico da Figura 4.4 mostra na primeira barra de cada frase o número médio de difusões por palavra, ou seja, quantas letras por palavra sofreram difusões. A segunda barra mostra a quantidade média de letras por difusão, ou seja, quantas possibilidades de letras foram geradas em média para cada difusão.

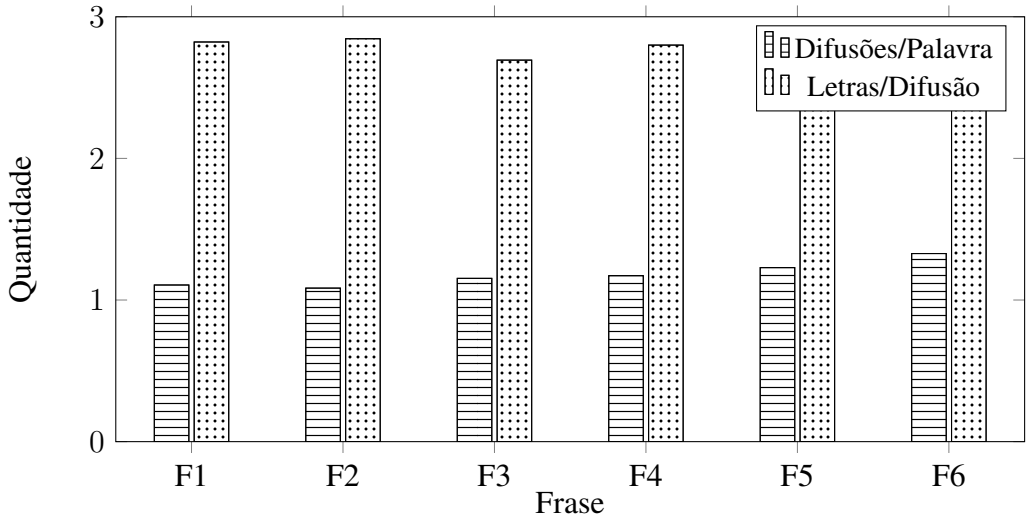


Figura 4.4: Número de difusões por palavra e letras por difusão

O gráfico da Figura 4.5 mostra a taxa de acerto dos algoritmos para as frases testadas. Os resultados foram melhores para o MDB em relação aos demais, ficando com uma taxa de acerto de 90 à 95%. O resultado alcançado pelo baseline 2 mostra a fragilidade da rede em classificar sozinha um objeto para um número relativamente grande de classes (26 letras).

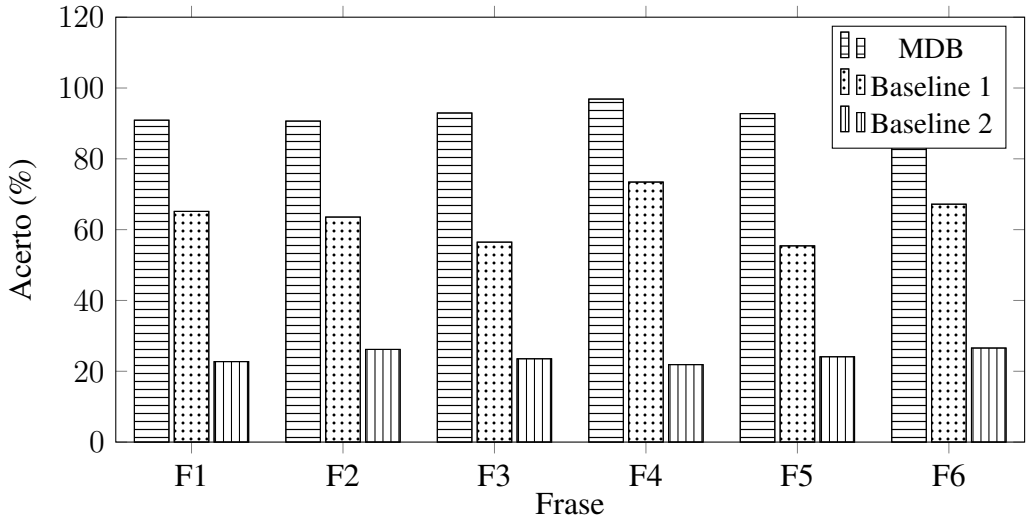


Figura 4.5: Taxa de acertos de cada algoritmo em cada frase.

A gráfico da Figura 4.6 apresenta a quantidade de chamadas do programa ao algoritmo de distância de Levenshtein. As operações estão dispostas em uma escala de milhões, e a diferença entre o MDB e o *baseline 1* é em média de 3 a 4 milhões de cálculos de distância. O *baseline 2* trouxe os melhores resultados. Porém, o ganho observado não compensa a taxa ínfima de acertos, como é visto no gráfico da Figura 4.5.

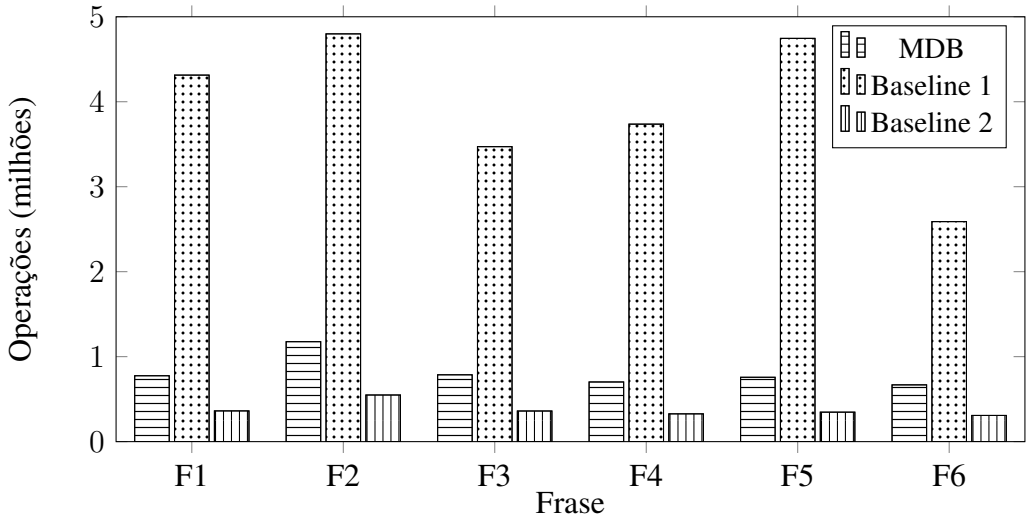


Figura 4.6: Quantidade de operações realizadas por cada algoritmo em cada frase de teste.

O último gráfico (Figura 4.7) representa o tempo de execução. Basicamente esse gráfico segue aproximadamente o mesmo padrão da Figura 4.6, ou seja, um número maior de computações implica em um maior de tempo de execução.

Porém é interessante observar a escala de tempo associada ao gráfico. Enquanto a busca das palavras da frase na MDB chegou a no máximo 5 segundos, a implementação do *baseline 1* (onde todos os anagramas são buscados) chegou a quase 30 segundos. Convém destacar que, mesmo tendo o *baseline 2* se sobressaído no tempo de execução, a sua taxa de acerto é bastante inferior à taxa obtida pelo MDB.

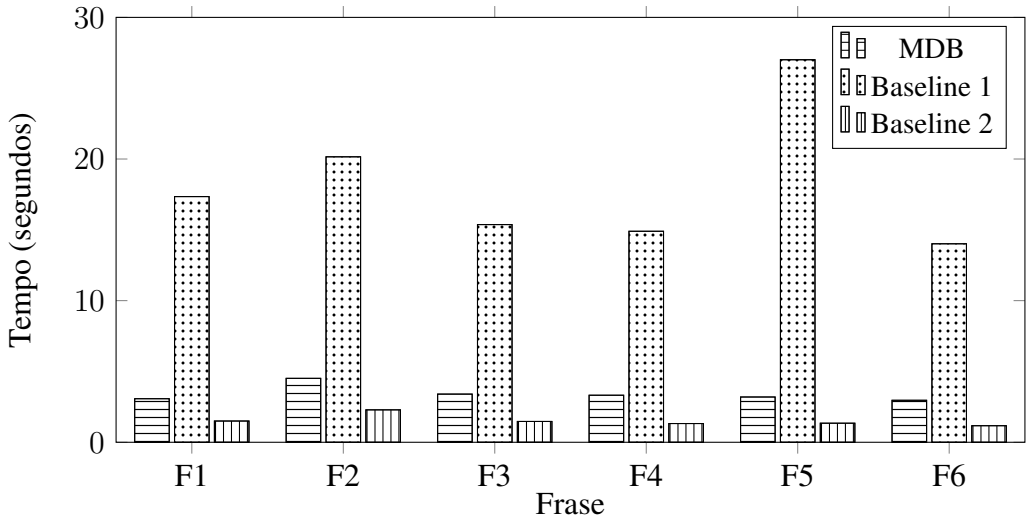


Figura 4.7: Tempo de busca de todas as palavras de cada frase por cada algoritmo testado.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O presente trabalho teve por objetivo a construção de uma extensão de algoritmos de distância e de indexação de objetos textuais. A extensão permitiu a busca por objetos similares a partir de uma entrada textual difusa, através da sua habilidade de resolver erros em palavras. Em relação à abordagem original, os experimentos mostraram uma relação promissora de custo-benefício, obtendo-se taxas consideravelmente maiores de acerto com uma queda sutil de desempenho.

O fato de o algoritmo realizar uma única busca por palavra independente do número de difusões facilita a busca por inúmeras possibilidades simultâneas. Esse é um diferencial da proposta, pois ela diminui o tempo de busca para o mesmo número de possíveis palavras.

No presente trabalho a busca indexada de palavras difusas é aplicada à saída de um módulo classificador, construído sobre uma rede neural artificial. Porém, existem outras aplicações sobre as quais pode-se aplicar busca por palavras difusas. Um exemplo é a listagem de palavras existentes em um anagrama, onde uma palavra possui suas letras dispostas em desordem. Uma busca pela versão difusa dessa palavra poderia indicar qual o texto real contido nela. Outro exemplo é a sugestão de correções ortográficas baseadas em difusões ocorridas por erros de digitação. Ou seja, quando uma tecla é pressionada (principalmente em dispositivos móveis com telas *touchscreen*) existe a possibilidade de erro, pois uma tecla vizinha pode sem intenção ser pressionada no lugar da tecla correta. Essa ocorrência pode ser vista como um problema de busca de palavras indexadas por uma estrutura difusa, onde cada erro pode gerar uma difusão contendo a letra pressionada e letras adjacentes a essa.

Uma das ideias em relação a futuras implementações desse módulo é a inserção de um caractere "curinga"para casos omissos da rede, onde não tenha sido possível convergir para nenhuma classe. O curinga serviria para permitir que um caractere desconhecido possa corresponder a qualquer caractere do alfabeto.

Também existe a possibilidade de teste com implementação de diferentes tipos de estruturas de indexação, como a *vantage point tree* (VPT). Além disso, pode-se implementar outros algoritmos de busca, como o KNN (*k nearest neighbors*), que busca os vizinhos mais próximos do objeto de busca.

REFERÊNCIAS

BAEZA-YATES, R.; NAVARRO, G. Fast approximate string matching in a dictionary. In: STRING PROCESSING AND INFORMATION RETRIEVAL: A SOUTH AMERICAN SYMPOSIUM, 1998. PROCEEDINGS. **Anais...** [S.l.: s.n.], 1998. p.14–22.

BISHOP, C. M. **Neural networks for pattern recognition**. [S.l.]: Oxford university press, 1995.

BOZKAYA, T.; OZSOYOGLU, M. Distance-based Indexing for High-dimensional Metric Spaces. **SIGMOD Rec.**, New York, NY, USA, v.26, n.2, p.357–368, June 1997.

CAMPOS, T. E. de; BABU, B. R.; VARMA, M. Character recognition in natural images. In: INTERNATIONAL CONFERENCE ON COMPUTER VISION THEORY AND APPLICATIONS, LISBON, PORTUGAL. **Proceedings...** [S.l.: s.n.], 2009.

HAMMING, R. W. Error Detecting and Error Correcting Codes. **Bell System Technical Journal**, [S.l.], v.29, n.2, p.147–160, 1950.

HARTIGAN, J. A.; WONG, M. A. Algorithm AS 136: a k-means clustering algorithm. **Journal of the Royal Statistical Society. Series C (Applied Statistics)**, [S.l.], v.28, n.1, p.100–108, 1979.

JAIN, A. K. Data clustering: 50 years beyond k-means. **Pattern Recognition Letters**, [S.l.], v.31, n.8, p.651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

LIN, Y.-S.; JIANG, J.-Y.; LEE, S.-J. A similarity measure for text classification and clustering. **IEEE transactions on knowledge and data engineering**, [S.l.], v.26, n.7, p.1575–1590, 2014.

MINSKY, M. L. Logical versus analogical or symbolic versus connectionist or neat versus scruffy. **AI magazine**, [S.l.], v.12, n.2, p.34, 1991.

ROGOVA, G. Combining the results of several neural network classifiers. **Neural Networks**, [S.l.], v.7, n.5, p.777 – 781, 1994.

SCHAUL, T. et al. PyBrain. **Journal of Machine Learning Research**, [S.l.], 2010.

SCHWEIZER, B.; SKLAR, A. Statistical metric spaces. **Pacific journal of mathematics**, [S.l.], v.10, n.1, p.313–334, 1960.

TOU, J.; GONZÁLEZ, R. **Pattern recognition principles**. [S.l.]: Addison-Wesley, 1977. (Applied mathematics and computation).

TRELEAVEN, P.; PACHECO, M.; VELLASCO, M. VLSI architectures for neural networks. **IEEE micro**, [S.l.], v.9, n.6, p.8–27, 1989.

WILLIAMS, C. **Principles of Management**. [S.l.]: South-Western/Cengage Learning, 2009.

YUJIAN, L.; BO, L. A Normalized Levenshtein Distance Metric. **IEEE Trans. Pattern Anal. Mach. Intell.**, Washington, DC, USA, v.29, n.6, p.1091–1095, June 2007.