Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Métodos de compressão de dados com pré-processamento via clusterização aplicados a contextos linguísticos

Lucas Silva Amorim

Santo André - SP, Agosto de 2022

Lucas Silva Amorim

Métodos de compressão de dados com pré-processamento via clusterização aplicados a contextos linguísticos

Projeto de Graduação apresentado ao Centro de Matemática, Computação e Cognição, como parte dos requisitos necessários para a obtenção do Título de Bacharel em Ciência da Computação.

Universidade Federal do ABC – UFABC

Centro de Matemática, Computação e Cognição

Bacharelado em Ciência da Computação.

Orientador: Prof.ª Dr.ª Cristiane M. Sato

Santo André - SP

Agosto de 2022

Resumo

Em um cenário onde a informação é um item cada muito valioso, a quantidade de bytes de texto consumidos cresce diariamente. Portanto, é muito importante diminuir o tamanho dos dados armazenados, tarefa que é feita pelos algoritmos de compressão. Neste trabalho, exploramos e testamos algumas técnicas para potencializar a compressão de dados textuais. A clusterização aliada à compressão baseada em *palavras* se mostrou eficiente na melhoria da taxa de compressão dos algoritmos, conforme mostram os experimentos realizados.

Palavras-chaves: compressão. texto. clusterização.

Abstract

In a scenario where information is a very valuable item, the amount of text bytes consumed grows daily. Therefore, it is very important to reduce the size of the stored data, a task that is done by the compression algorithms. In this work, we explore and test some techniques to enhance the compression of textual data. The clustering combined with the compression based on *words* proved to be efficient in improving the compression rate of the algorithms, as shown by the experiments carried out.

Keywords: compression. text. clustering.

Sumário

Introdução

Em um mundo cada vez mais informatizado, a demanda por armazenamento e transmissão de grandes quantidades de dados cresce "exponencialmente". Algoritmos de compressão de dados são amplamente utilizados por protocolos de transmissão de informações [5] e em sistemas de banco de dados [7], isso porque o processo de compressão permite a redução total de dados para representar uma certa informação.

Um algoritmo de compressão pode ser classificado como sendo **com perda**, quando reconstrói apenas uma aproximação do conteúdo original. Esses algoritmos são comumente utilizados na compressão de mídias como vídeos, imagens e áudio, onde a perda de alguns bits não tem uma interferência relevante na recuperação do conteúdo original. Em contrapartida, os algoritmos do tipo **sem perda**, são aqueles em que o conteúdo original que foi comprimido é totalmente recuperado após a descompressão, sendo utilizados principalmente em textos, onde uma simples troca de caracteres pode comprometer o significado do conteúdo original.

Existem diversos tipos de algoritmos de compressão sem perda, neste trabalho utilizaremos os **baseados em probabilidades** e **baseados em dicionários**. Os algoritmos **baseados em probabilidade** constroem o código a partir das probabilidades de ocorrência de cada símbolo dentro do texto. Já os **baseados em dicionários** tem como ideia central encontrar padrões nos dados substituindo esses padrões por tokens que ocupam "menos memória" [6].

Classificamos a compressão de dados como uma sub-área da Teoria da Informação, já que todos os resultados e limites teóricos para os algoritmos são fundamentados pela mesma. De fato, todo algoritmo de compressão supõe que a mensagem possui um certo nível de redundância, e toma vantagem desse fato para representar a informação de maneira mais eficiente. Portanto, podemos supor que a **taxa de compressão** de um algoritmo está intimamente relacionada com a sua entropia (esse assunto será melhor detalhado durante em capítulos posteriores), o que nos dá um indício de que **minimizar a entropia**, pode **maximizar** a eficiência na compressão.

O foco deste trabalho, estará justamente em explorar de maneira empírica essa relação entre a entropia associada a um texto e a taxa de compressão, utilizando técnicas de pré-processamento para minimização da entropia. Vamos utilizar técnicas de clusterização de dados [8] para minimizar a entropia da base que será experimentada a fim de otimizar o processo de compressão. Através deste trabalho espera-se validar a hipótese de que a clusterização pode ser utilizada como um método de melhoria em algoritmos de compressão de textos, bem como apresentar os fundamentos teóricos que sustentam esta hipótese.

O texto está dividido em três partes. A primeira parte apresenta a definição de código, entropia e a relação entre esses conceitos, através de lemas e teoremas que fundamentam estes resultados. A segunda parte apresenta os algoritmos de compressão utilizados no trabalho (Huffman, Huffword, LZ77, WLZ 77), bem como alguns detalhes de implementação e otimizações que serão utilizadas posteriormente. Ainda nesta mesma parte, são apresentados alguns conceitos básicos e definições fundamentais para o entendimento do método de clusterização (*k-means*) que será aplicado como pré-processamento. A parte final do trabalho apresenta o experimento que testa a clusterização como um método de melhoria na taxa de compressão dos algoritmos apresentados, detalhando as tecnologias e métodos utilizados, bem como os resultados obtidos.

Observa-se que, algumas das demonstrações omitidas nos textos utilizados como base, foram incluídas por elaboração do autor deste texto (e devidamente indicadas como autorais), como parte do esforço intelectual para a construção deste trabalho. Por fim, a ideia geral do trabalho foi concebida para explorar os aspectos diversos do conhecimento adquirido durante o curso de Bacharelado em Ciência da Computação.

Parte I

Fundamentação Teórica

1 Conceitos, definições e resultados fundamentais em compressão de dados

Este capítulo apresenta algumas definições, conceitos e resultados fundamentais para o entendimento das técnicas de compressão que serão discutidas em capítulos posteriores.

1.1 Código

Dado um conjunto A, usaremos a notação A^+ para definir o conjunto que contém todas as cadeias não-vazias formadas pelas possíveis combinações de A. Ou seja,

$$A^+ = \left\{(a_1, a_2, \ldots, a_n) \colon n \in \mathbb{N},\ n > 0,\ a_i \in A\ \forall i \in \{1, \ldots, n\}\right\}$$

Por simplicidade de notação, utilizaremos $a_1 a_2 \cdots a_n$ para denotar a sequência $(a_1, a_2, \ldots, a_n)$ quando não houver ambiguidade.

Um **alfabeto** A é um conjunto finito. Chamaremos os elementos de A de **símbolos** ou **letras** e os elementos de A^+ de **cadeias** ou **palavras**.

Um **código** C mapeia cada símbolo $m \in M$ para uma cadeia em W^+. Mais precisamente, um código é uma função injetora de um conjunto M para W^+. O conjunto M é chamado de **alfabeto de origem** e W é chamado de **alfabeto código**. Chamaremos cada cadeia na imagem de C de **palavra-código**.

Dentro deste contexto, definimos o **comprimento** de uma palavra $w \in W^+$, denotado por $\ell(w)$, como o inteiro positivo que representa o tamanho da sequência w.

Tome como exemplo o alfabeto de origem $M = \{a, b, c\}$ e o alfabeto código $W = \{0, 1\}$ composto somente por valores binários, poderíamos definir um código C da seguinte forma:

Tabela 1 – Tabela do código C

alfabeto de origem	palavra-código
a	1
b	01
c	00

Neste exemplo, o **comprimento** da palavra-código associado à letra "b" é dado por $\ell(C(b)) = \ell(01) = 2$.

As palavras-código associadas a cada símbolo podem ter um tamanho *fixo* ou *variável* [1]. Códigos nos quais palavras-código possuem um comprimento fixo são chamados

de **códigos de comprimento fixo**, enquanto os que possuem alfabetos de comprimento variáveis são chamados **códigos de comprimento variável**. Note que o exemplo anterior é um código de comprimento variável. Provavelmente o exemplo mais conhecido de código de **comprimento fixo** seja código ASCII, que mapeia 64 símbolos alfa-numéricos (ou 256 em sua versão estendida) para palavras-código de 8 bits. Todavia, no contexto de compressão de dados procuramos construir códigos que podem variar em seu comprimento baseados na sua probabilidade associada, a fim de reduzir o tamanho médio do conteúdo original ao codificá-lo [1].

1.1.1 Códigos unicamente decodificáveis e livres de prefixo

Dado um alfabeto de origem M, chamamos as palavras em M^+ de **mensagens**. De fato, queremos codificar e decodificar mensagens utilizando códigos e não somente símbolos isolados. Podemos estender a noção de código para mensagens naturalmente da forma a seguir. Dado um código $C\colon M \to W^+$, defina **código estendido** $C^+ : M^+ \to W^+$ por

$$C^+(m_1 m_2 \dots m_k) = \text{concat}(C(m1), C(m_2), \dots, C(m_k)), \text{ para todo } m_1 m_2 \dots m_k \in M^+,$$

onde concat$(\cdot)$ é a função de concatenação de sequências.

Um código $C\colon M \to W^+$ é dito **unicamente decodificável** se C^+ é uma função injetora. Ou seja, toda mensagem é mapeada para uma palavra única [2].

Um **código livre de prefixo** é um código em que nenhuma palavra-código é prefixo de outra. Uma palavra $w_1 \cdots w_k$ é **prefixo** de uma palavra $v_1 \cdots v_\ell$ se $k \leq \ell$ e $w_1 \cdots w_k = v_1 \cdots v_k$. Ou seja, um código $C\colon M \to W^+$ é livre de prefixo se vale que, para quaisquer $m, m' \in M$ distintos, $C(m)$ não é prefixo de $C(m')$.

Por exemplo, o código que possui sua imagem no conjunto de palavras-código $W^+ := \{1, 01, 000, 001\}$ não possui nenhuma cadeia que é prefixo de outra, portanto é considerado um **código livre de prefixo**. Códigos livres de prefixo não apenas são unicamente decodificáveis como podem ser *decodificados instantaneamente*, pois, ao processar uma cadeia de sequência de palavras-código podemos decodificar cada uma delas sem precisar verificar o início da seguinte.

Todo código $C\colon M \to W^+$ pode ser representado por uma árvore $T(C)$ onde [2]

- as arestas são rotuladas por símbolos de W e,
- cada símbolo $m \in M$ é associado a um nó $v(m)$ da árvore,
- para cada símbolo $m \in M$, a palavra-código $C(m)$ é obtida pelo caminho da raiz até $v(m)$ através da concatenação dos rótulos das arestas do caminho

- cada folha da árvore é $v(m)$ para algum $m \in M$.

Note que, se C é um código livre de prefixo, todos os símbolos em M são mapeados para folhas, pois se $v(m)$ está no caminho da folha até $v(m')$ ($v(m)$ é ancestral de m'), então $C(m)$ é prefixo de $C(m')$. A árvore a seguir representa o código da Tabela 3 (claramente um código livre de prefixo).

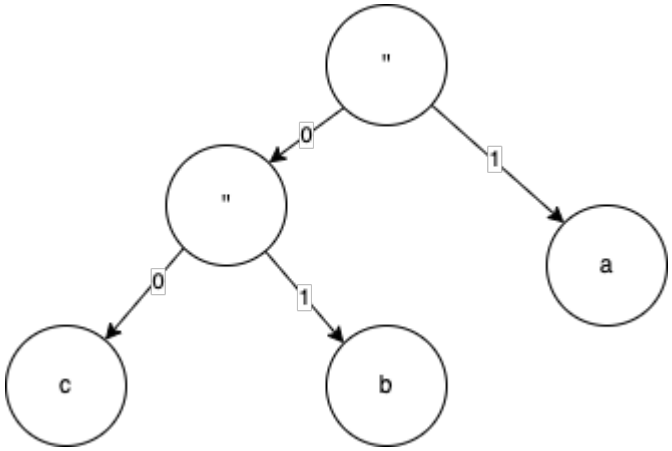

Figura 1 – Árvore de um código livre de prefixo

1.2 Relações fundamentais com a Teoria da Informação

A codificação é comumente dividida em duas componentes diferentes: *modelo* e *codificador*. O *modelo* identifica a distribuição de probabilidade das mensagens baseado em sua semântica e estrutura. O *codificador* toma vantagem de um possível *enviesamento (bias)* apontado pela modelagem, e utiliza as probabilidades associadas para reduzir a quantidade de dados necessária para representar a mesma informação (substituindo os símbolos que ocorrem com maior frequência por palavras-código menores) [1]. Isso significa que a codificação está diretamente ligada às probabilidades associadas a cada símbolo do alfabeto de origem.

Nesta seção, vamos construir o embasamento teórico necessário para entender a relação entre as probabilidades associadas e o comprimento das mensagens, e consequentemente criar uma noção dos parâmetros que devem ser maximizados ou minimizados para alcançar uma codificação eficiente.

1.2.1 Conceitos básicos em Probabilidade

Neste texto, iremos trabalhar com espaços de probabilidade finitos, pois lidamos com alfabetos de origem finitos. Um **espaço de probabilidade finito** é um par (Ω, P), onde Ω é um conjunto finito e $P\colon \Omega \to \mathbb{R}_{\geq 0}$ tal que $\sum_{\omega \in \Omega} P(\omega) = 1$.

O conjunto Ω é chamado de **espaço amostral** e P é chamada de **função de probabilidade**. Um **evento** E é um sunconjunto de Ω e sua probabilidade é definida como $P(E) := \sum_{\omega \in E} P(\omega)$ [4].

Uma **variável aleatória** X associa um número real a cada um dos elementos de Ω. Em outras palavras, X é uma função que mapeia os elementos do espaço amostral para números reais. Uma variável aleatória é chamada **discreta** quando os valores dos experimentos associados a ela são números finitos ou ao menos infinitos que podem ser contados [4].

Podemos descrever melhor uma variável aleatória X, atribuindo probabilidades sobre os valores que esta pode assumir. Esses valores são atribuídos pela **distribuição de probabilidade**, denotada por p_X, definida para cada $x \in \mathbb{R}$ como

$$p_X(x) = P(\{\omega \in \Omega : X(\omega) = x\}). \tag{1.1}$$

Por simplicidade de notação utilizamos $P(X = x)$ para denotar $P(\{\omega \in \Omega : X(\omega) = x\})$. Note que a pré-imagem de X é o espaço amostral e, portanto,

$$\sum_{x \in im_X} p_X(x) = 1, \tag{1.2}$$

onde im_X denota a imagem de X.

O **valor esperado** (ou **esperança**) da variável aleatória X é definido como [4]:

$$\mathbf{E}[X] = \sum_{x \in im_X} x p_X(x). \tag{1.3}$$

1.2.2 Comprimento médio do código

Seja p a distribuição de probabilidade associada ao alfabeto de origem M. Seja $C : M \to W^+$ um código. Definimos o **comprimento/tamanho médio** de C como:

$$\ell_a(C) = \sum_{m \in M} p(m)\ell(C(m)). \tag{1.4}$$

Estamos interessados em códigos unicamente decodificáveis com o menor comprimento médio possível. Dada um espaço de probabilidade (M, P) e um alfabeto código W, Um código $C : M \to W^+$ unicamente decodificável é **ótimo** se $\ell_a(C)$ é mínimo, isto é, para qualquer código unicamente decodificável C' temos que

$$\ell_a(C) \leq \ell_a(C'). \tag{1.5}$$

Assim, temos o seguinte problema de otimização

$$\begin{aligned} \min \quad & \ell_a(C) \\ \text{sujeito a} \quad & C : M \to W^+ \text{ é um código unicamente decodificável} \end{aligned} \tag{1.6}$$

1.2.3 Entropia

A **entropia de Shannon** aplica as noções de entropia física (que representa a aleatoriedade de um sistema) à Teoria da Informação [2]. Dada uma variável aleatória X com distribuição de probabilidade p, definimos **entropia** de X como

$$H(X, p, b) = \sum_{x \in im_X} p(x) \log_b \frac{1}{p(x)}, \tag{1.7}$$

onde b é comumente tomado como 2, mas pode assumir valores inteiros maiores do que 2.

Por esta definição temos que quanto menor o enviesamento da função de distribuição de probabilidade relacionada ao sistema, maior a sua entropia. Em outras palavras, a entropia de um sistema está intimamente ligada a sua "desordem".

Shannon (1948) [3] aplica o mesmo conceito de entropia no contexto da Teoria da Informação, "substituindo" o conjunto de estados pelo conjunto de símbolos M, isto é, M é interpretado como um conjunto de possíveis símbolos, tendo como $p(m)$ a probabilidade de cada $m \in M$. Baseado na mesma premissa, Shannon mede a informação de cada símbolo da seguinte forma

$$i(m, b) = \log_b \frac{1}{p(m)}. \tag{1.8}$$

1.2.4 Comprimento de Código e Entropia

Nas seções anteriores, o comprimento médio de um código foi definido em função da distribuição de probabilidade associada ao seu alfabeto de origem. Da mesma forma, as noções de **entropia** relacionada a um conjunto de mensagens têm ligação direta com as probabilidades associadas a estas. A seguir, será mostrado como podemos relacionar o comprimento médio de um código a sua entropia através da **Desigualdade de Kraft-McMillan** , e por consequência estabelecer uma relação direta entre **a entropia de um conjunto de mensagens e a otimalidade do código associada a estas mensagens**.

Teorema 1 (Desigualdade de Kraft-McMillan)**.** *Sejam M e W conjuntos finitos tais que $|W| \geq 2$. Seja $b := |W|$. Para todo código unicamente decodificável $C : M \to W^+$, vale que*

$$\sum_{m \in M} b^{-\ell(C(m))} \leq 1. \tag{1.9}$$

Além disso, para toda função $f : M \to \mathbb{N}$ tal que que satisfaça

$$\sum_{m \in M} b^{-f(m)} \leq 1, \tag{1.10}$$

existe um código livre de prefixo (e, portanto, unicamente decodificável) tal que $\ell(C(m)) = f(m)$ para todo $m \in M$.

Antes de apresentar a demonstração do Teorema 1, vamos explorar as suas consequências para comprimentos de códigos ótimos. Uma de suas consequências principais é que o valor da solução ótima para o problema de otimização 1.6

$$\begin{aligned} \min \quad & \ell_a(C) = \textstyle\sum_{m\in M} p(m)\ell(C(m)) \\ \text{sujeito a} \quad & C : M \to W^+ \text{ é um código unicamente decodificável} \end{aligned}$$

é igual ao valor ótimo do seguinte problema de otimização

$$\begin{aligned} \min \quad & \textstyle\sum_{m\in M} p(m) f(m) \\ \text{sujeito a} \quad & \textstyle\sum_{m\in M} b^{-f(m)} \leq 1 \\ & f : M \to \mathbb{N} \end{aligned}$$

Além disso, se f^* é uma solução ótima para o problema acima, então existe um código $C^* : M \to W^+$ livre de prefixo para o qual f^* determina os comprimentos de suas palavras-código. Portanto, podemos concluir que dentro os códigos unicamente decodificáveis de comprimento médio ótimo sempre existe um que é livre de prefixo.

Vamos ver, em capítulos posteriores, que o algoritmo de Huffman produz um código livre de prefixo que é ótimo dentre os códigos livres de prefixo. A discussão acima mostra que tal código também é ótimo dentre todos os códigos unicamente decodificáveis.

A seguir apresentamos a demonstração da Desigualdade de Kraft-McMillan.

Prova do Teorema 1 (Autoral). Primeiramente, mostraremos a equação (1.9).

Seja $C : M \to W^+$ um código unicamente decodificável. Seja $b := |W|$. Seja $\ell_{\max} = \max_{m\in M} \ell(C(m))$. Seja k um inteiro positivo. Seja

$$W_k := \left\{ C^+(m_1 \cdots m_k) : m_1, ..., m_k \in M \right\}$$

e seja

$$W_{k,j} := \{ w \in W_k : \ell(w) = j \},$$

para $j \in \mathbb{N}$. Temos que

$$\begin{aligned} \left(\sum_{m\in M} b^{-\ell(C(m))} \right)^k &= \sum_{m_1,...,m_k \in M} b^{-\sum_{i=1}^k \ell(C(m_i))} \\ &= \sum_{w\in W_k} b^{-\ell(w)} \\ &= \sum_{j=1}^{k\cdot\ell_{\max}} |W_{k,j}| \cdot b^{-j} \\ &\leq \sum_{j=1}^{k\cdot\ell_{\max}} b^j \cdot b^{-j}, \qquad \text{pois } |W_{k,j}| \leq b^j \\ &= k \cdot \ell_{\max} \end{aligned}$$

Logo,

$$\sum_{m\in M} b^{-\ell(C(m))} \leq (k\cdot \ell_{max})^{\frac{1}{k}},$$

para todo k inteiro positivo. Como $\lim_{k\to\infty}(k\cdot \ell_{max})^{\frac{1}{k}} = 1$, temos que

$$\sum_{m\in M} b^{-\ell(C(m))} \leq 1.$$

Agora vamos provar que a condição (1.10) é suficiente para a existência de um código livre de prefixo com comprimentos dados por f. Seja $f : M \to \mathbb{N}$ satisfazendo (1.10). Seja $(m_1, \ldots, m_n)$ tal que $M = \{m_1, \ldots, m_n\}$ e $f(m_1) \leq \cdots \leq f(m_n)$. Seja $\ell_i := f(m_i)$ para todo $i \in \{1, \ldots, n\}$.

Agora vamos construir um código livre de prefixo C em uma ordem crescente de tamanho, de maneira em que $\ell(m_i) = \ell_i$. Seja $\ell_{\max}$ o maior valor de ℓ_i.

- Para j de 1 a $\ell_{\max}$, defina W^j como todas as palavras em W^+ de comprimento j e defina $P_j := \varnothing$.
- Para i de 1 a n, faça
  - seja $C(m_i)$ qualquer palavra em $W_{\ell_i} \setminus P_{\ell_i}$,
  - adicione $C(m_i)$ a P_{ℓ_i} e adicione todas as palavras que têm $C(m_i)$ como prefixo a cada P_{ℓ_j} com $j > i$.

O procedimento acima produz um código livre de prefixo C desde que P_{ℓ_i} não seja o conjunto W_{ℓ_i} inteiro para cada $i \in \{1, \ldots, n\}$. Ou seja, podemos pensar em P_{ℓ_i} como um conjunto de palavras proibidas para m_i. Seja $b = |W|$. Note que inicialmente W_{ℓ_i} tem tamanho b^{ℓ_i} e P_{ℓ_i} é vazio, pois nenhuma palavra é proibida. Para cada $k < i$, ao atribuir um código de W_{ℓ_k} para m_k, adicionamos $b^{\ell_i-\ell_k}$ palavras a P_{ℓ_i}.

Suponha para uma contradição que o procedimento falha em atribuir uma palavra-código para m_i. Então

$$b^{\ell_i} = |P_{\ell_i}| = \sum_{k=1}^{i-1} b^{\ell_i-\ell_k}.$$

Portanto, dividindo os dois lados por b^{l_i}, obtemos

$$1 = \sum_{k=1}^{i-1} b^{-\ell_k} < \sum_{k=1}^{n} b^{-\ell_k} = \sum_{m\in M} b^{-f(m)}$$

o que contradiz o fato que a condição (1.10) é satisfeita. □

A seguir vamos ver a relação de entropia com comprimento médio ótimo. Primeiramente, veremos que a entropia é um limitante inferior para o comprimento médio de códigos unicamente decodificáveis.

Lema 2 (Entropia como limite inferior)**.** *Dados um conjunto finito M e uma função de probabilidade p para M, vale que*

$$H(M, p, b) \leq \ell_a(C),$$

para todo código unicamente decodificável $C : M \to W^+$ tal que $|W| = b$.

Por outro lado, veremos também que um código de comprimento médio ótimo tem valor no máximo a entropia somada com 1.

Lema 3 (Entropia como limite superior)**.** *Dados um conjunto finito M e uma função de probabilidade p para M, existe um código $C : M \to W^+$ livre de prefixo tal que*

$$\ell_a(C) \leq H(M, p, b) + 1$$

onde $b = |W|$.

A seguir apresentamos as demonstrações dos lemas acima.

Prova do Lema 2 (Blelloch, 2013 [2]). Seja $C : M \to W^+$ um código unicamente decodificável e seja $b := |W|$. Seja p uma função de probabilidade para M. Queremos provar que $H(M, p, b) - \ell_a(C) \leq 0$.

Substituindo $H(M, p, b)$ e $\ell_a(C)$ por suas definições (equação (1.7) e equação (1.4), resp), temos

$$\begin{aligned} H(M, p, b) - \ell_a(C) &= \sum_{m \in M} p(m) \log_b \frac{1}{p(m)} - \sum_{m \in M} p(m)\ell(C(m)) \\ &= \sum_{m \in M} p(m) \left(\log_b \frac{1}{p(m)} - \ell(C(m)) \right) \\ &= \sum_{m \in M} p(m) \left(\log_b \frac{1}{p(m)} - \log_b b^{\ell(C(m))} \right) \\ &= \sum_{m \in M} p(m) \log_b \left(\frac{b^{-\ell(C(m))}}{p(m)} \right) \end{aligned}$$

Agora, tome X como uma variável aleatória e p_X como a distribuição de probabilidade associada a ela. Tome $f : \mathbb{R} \to \mathbb{R}$ como uma função côncava tal que $f(X)$ existe e está bem definida. Pela **Desigualdade de Jensen** podemos afirmar que $f(E[X]) \leq E[f(X)]$. Ou ainda, utilizando a definição 1.3 para o valor esperado:

$$f(\sum_{x \in X} x p_X(x)) \leq \sum_{x \in X} p_X(x) f(x)$$

Sabe-se que a função $\log_b$ é côncava e que o p é a distribuição de probabilidade sobre M. Aplicando a Desigualdade de Jensen ao resultado obtido anteriormente deduzimos que:

$$\sum_{m \in M} p(m) \log_b \left(\frac{b^{-\ell(C(m))}}{p(m)} \right) \leq \log_b \left(\sum_{m \in M} b^{-\ell(C(m))} \right)$$

Agora, aplicamos a desigualdade de Kraft-McMillan e concluímos

$$H(M,p,b) - \ell_a(C) \leq \log_b 1 = 0,$$

como queríamos demonstrar. □

Prova do Lema 3 (Blelloch, 2013 [2]). Seja (M,p) um espaço de probabilidade finita. Podemos supor que $p(m) > 0$ para todo M, pois elementos de M com probabilidade zero podem ser ignorados.

Seja $f : M \to \mathbb{N}$ definida como $f(m) = \left\lceil \log_b \frac{1}{p(m)} \right\rceil$ para todo $m \in M$. Temos que

$$\begin{aligned}
\sum_{m\in M} b^{-f(m)} &= \sum_{m\in M} b^{-\left\lceil \log_b \frac{1}{p(m)} \right\rceil} \\
&\leq \sum_{m\in M} b^{-\log_b \frac{1}{p(m)}} \\
&= \sum_{m\in M} p(m) \\
&= 1
\end{aligned}$$

Ou seja, f satisfaz a condição (1.10). De acordo com a desigualdade de Kraft-McMillan existe um código livre de prefixo C com palavras-código de tamanho dado por f, portanto

$$\begin{aligned}
\ell_a(C) &= \sum_{m\in M} p(m) f(m) \\
&= \sum_{m\in M} p(m) \left\lceil \log_b \frac{1}{p(m)} \right\rceil \\
&\leq \sum_{m\in M} p(m) \left(1 + \log_b \frac{1}{p(m)} \right) \\
&= 1 + \sum_{m\in M} p(m) \log_b \frac{1}{p(m)} \\
&= 1 + H(M,p,b).
\end{aligned}$$

□

Parte II

Algoritmos de compressão e pré-processamento

2 Algoritmos de compressão sem perda

Os algoritmos de compressão podem ser categorizados em duas diferentes classes: os de compressão **com perda** e **sem perda**. Os **algoritmos de compressão com perda** admitem uma baixa porcentagem de perda de informações durante a codificação para obter maior performance, muito úteis na transmissão de dados em streaming por exemplo. Nos **algoritmos de compressão sem perda** o processo de codificação deve ser capaz de recuperar os dados em sua totalidade, geralmente utilizados em casos onde não pode haver perda de informações (como por exemplo, compressão de arquivos de texto).

Neste capítulo serão apresentados dois dos principais algoritmos de compressão sem perda (**Código de Huffman** e **Lempel-Ziv 77**), bem como algumas variações úteis para o propósito do presente trabalho.

2.1 Código de Huffman

O **algoritmo de Huffman** (desenvolvido por David Huffman em 1952 [9]) é um dos componentes mais utilizados em algoritmos de compressão sem perda, servindo como base para algoritmos como o Deflate (utilizado amplamente na web). Os códigos gerados a partir do algoritmos de Huffman são chamados **Códigos de Huffman**.

O código de Huffman é descrito em termos de como ele gera uma árvore de código livre de prefixo com alfabeto código $W = \{0, 1\}$. A árvore é binária e assumimos rótulo 0 nas arestas de um nó para um filho da esquerda e rótulo 1 nas arestas de um nó para um filho da direita. Considere o conjunto de símbolos M e, para cada símbolo $m_i \in M$, seja p_i a probabilidade associada a m_i.

Algorithm 1 Algoritmo de Huffman

$Forest \leftarrow []$

for all $m_i \in M$ **do** ▷ Inicializando floresta
  $T \leftarrow newTree()$
  $node \leftarrow newNode()$
  $node.weight \leftarrow p_i$ ▷ $w_i = p_i$
  $T.root \leftarrow node$
  $Forest.append(T)$ ▷ Adiciona um nova árvore na floresta
end for

while $Forest.size > 1$ **do**
  $T1 \leftarrow ExtractMin(Forest)$ ▷ Devolve a árvore cuja raiz é mínima, e retira da floresta
  $T2 \leftarrow ExtractMin(Forest)$
  $HTree \leftarrow newTree()$
  $HTree.root \leftarrow newNode()$

  $HTree.root.left \leftarrow T1.root$
  $HTree.root.right \leftarrow T2.root$
  $HTree.root.weight \leftarrow T1.root.weight + T2.root.weight$
  Forest.append(HTree)
end while

2.1.1 Análise assintótica

Seja n o tamanho do conjunto de símbolos M. Para que o algoritmo percorra toda a floresta, formada por uma árvore para cada $m \in M$, serão necessárias n iterações. Considerando que as funções *ExtractMin()* e *.append()* foram construídas a partir de uma fila de prioridades de **heap** e essas operações podem ser realizadas em tempo $O(\log_2 n)$, o algoritmo será executado em tempo $O(n \log_2 n)$ [2].

2.1.2 Corretude

O teorema a seguir (provado por Huffman, 1952 [9]) afirma que os códigos de Huffman minimizam o comprimento médio, ou seja, são ótimos dentre os códigos livres de prefixo.

Nesta seção, considere um alfabeto de origem $M = \{m_1, \ldots, m_n\}$ com probabilidade p_i para cada símbolo m_i e alfabeto código $W = \{0, 1\}$. Dizemos que um código $C : M \to W^+$ livre de prefixo é **ótimo** se $\ell_a(C) \leq \ell_a(C')$ para todo código $C' : M \to W^+$ livre de prefixo.

Teorema 4. *Seja C_H um código gerado pelo algoritmo de Huffman. Temos que C_H é ótimo.*

Antes de provar o teorema, começamos com uma simples observação.

Lema 5. *Seja $C : M \to W^+$ um código livre de prefixo ótimo. Para todo $i, j \in \{1, \ldots, n\}$, se $p_i > p_j$, então $\ell(C(m_i)) \leq \ell(C(m_j))$*

Demonstração. Sejam $i, j \in \{1, \ldots, n\}$ e suponha que $p_i \geq p_j$. Seja $w_i = C(m_i)$ e seja $w_j = C(m_j)$. Para efeito de contradição, suponha que $\ell(w_i) > \ell(w_j)$. Agora construa um novo código C', trocando w_i por w_j. Ou seja, $C'(m_k) = C(m_k)$ para todo $k \in \{1, \ldots, n\} \setminus \{i, j\}$, $C'(m_i) = w_j$ e $C'(m_j) = w_i$. Dado o comprimento médio $\ell_a(C')$ é

$$\begin{aligned}\ell_a(C') &= \ell_a(C) + p_j \cdot (\ell(w_i) - \ell(w_j)) + p_i \cdot (\ell(w_j) - \ell(w_i)) \\ &= \ell_a(C) + (p_j - p_i)(\ell(w_i) - \ell(w_j)) \\ &< \ell_a(C),\end{aligned}$$

pois $p_j - p_i < 0$ e $\ell(w_i) - \ell(w_j) > 0$. Isso contradiz o fato do código C ser ótimo. □

Nota: Perceba que em uma árvore de Huffman, o comprimento de uma palavra-código também representa seu nível na árvore.

Prova do Teorema 4 (Blelloch, 2013 [2]). A prova se dará por indução sobre o tamanho n de M. Para a base, tome $n = 2$. Nesse caso, o resultado é trivialmente verdadeiro considerando que o algoritmo de Hufman gera um código que atribui um bit pra cada símbolo de M. Suponha então que $n > 2$ para o passo indutivo. Sejam x e y nós associados a símbolos m_a e m_b respectivamente tais que $p_a \leq p_b \leq p_i$ para todo $i \in \{1, \ldots, n\} \setminus \{a, b\}$, onde a e b são distintos. Ou seja, x e y poderiam ser os nós escolhidos como o primeiro par pelo algoritmo de Huffman.

Seja m' um novo símbolo com probabilidade $p_a + p_b$ e seja $M' = (M \cup \{m'\}) \setminus \{m_a, m_b\}$. Note que $|M'| < n$. Seja T' uma árvore obtida pelo algoritmo de Huffman para

M'. Por hipótese de indução o código C' da árvore T' é ótimo para M'. A árvore T obtida pelo algoritmo de Huffman é a árvore T' adicionando os nós x e y como filhos do nó associado a m'. Seja C o código da árvore T.

Agora iremos comparar com um código ótimo. Afirmamos que

Existe um código C^* livre de prefixo ótimo com árvore T^* tal que x e y são **irmãos**. (2.1)

Seja z o pai de x e de y em T^*. Note que a árvore obtida a partir T^* pela remoção de x e y e associando z ao símbolo m' (de probabilidade $p_a + p_b$) gera um código C^{**} para M'. Pela otimalidade de C', temos que

$$\ell_a(C^*) = \ell_a(C^{**}) + p_a + p_b \geq \ell_a(C') + p_a + p_b = \ell_a(C),$$

o que mostra a otimalidade de C.

Finalmente provamos a afirmação em (2.1). Dado um código C'' livre de prefixo ótimo com árvore T'', pelo lema 5, sabemos que os símbolos com as menores probabilidades estão nos menores níveis de T'' (ou seja, associados a palavras-código de maior comprimento). Portanto, x e y são folhas em T''. Seja $w \in \{x, y\}$ tal que w é x se o nível de x é menor ou igual ao nível de y e seja v o nó em $\{x, y\} \setminus \{w\}$. (Note que w só pode ser y no caso em que $p_a = p_b$, pelo lema 5). Seja z o pai de w. Se w é o único filho de z, então poderíamos mudar w para z (pois z não é associado a nenhum símbolo dado que C'' é livre de prefixo). Portanto, w tem um nó **irmão** w'. Se w' não é v, trocar w' com v gera um código de comprimento menor ou igual ao de C'' pelo lema 5. □

2.2 Lempel-Ziv 77 (LZ77)

2.2.1 Algoritmos de Lempel-Ziv

Nos anos de 1977 e 1978, Jacob Ziv e Abraham Lempel publicaram dois artigos apresentando os algoritmos **LZ77** e **LZ78** [10], que serviriam como base para uma família de algoritmos de compressão (conforme mostrado na Figura 2), chamados de algoritmos de **Lempel-Ziv**. Os algoritmos de Lempel-Ziv realizam o processo de compressão baseado em um **dicionário** de mensagens vistas anteriormente (diferente do Algoritmo de Huffman, que utiliza a probabilidade associada a cada símbolo). Tanto o LZ77 quanto o LZ78 têm um funcionamento parecido, que se resume em substituir partes da entrada por referências a partes iguais anteriormente processadas, e diferem na maneira em que procuram por repetições a serem substituídas.

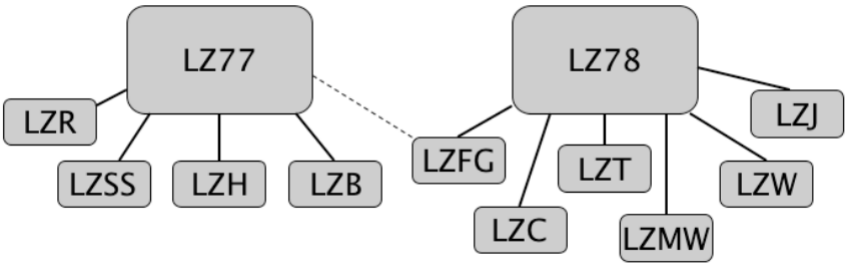

Figura 2 – Familia de algoritmos Lempel Ziv

2.2.2 Descrição do LZ77

O LZ77 e suas variações utilizam a técnica de *janela deslizante* para encontrar mensagens correspondentes. A janela é dividida em duas partes separadas por um *cursor* (que se move conforme novas mensagens são codificadas).

A mensagem à esquerda do cursor é chamada de **dicionário**, e contém todos os símbolos já codificados. Já a mensagem à direita do *cursor* é chamada de **lookahead buffer**. A ideia geral do algoritmo é substituir as mensagens do *lookahead buffer* por **tokens**, onde cada *token* é constituído por uma *tupla* que "aponta" para a maior mensagem correspondente a cadeia no início do *lookahead buffer*.

A *tupla* que constitui os *tokens* é formada por três valores: um valor inteiro que indica quantas posições para trás o *cursor* deve retornar até encontrar o início da cadeia, um segundo valor inteiro indicando o tamanho da cadeia, e um caracter (preenchido com m_{cursor} ou *null* para **tokens não vazios**).

Os *tokens* podem ser vazios, quando não encontramos nenhuma mensagem correspondente, neste caso, os dois primeiros valores da *tupla* são preenchidos com zeros. Por exemplo, se durante a compressão de um texto encontrássemos o símbolo "a" pela primeira

vez (isso é, não existe correspondência para ele no *dicionário*), a saída seria o *token vazio* $(0, 0,$"a"$)$.

Algorithm 2 Algoritmo Lempel-Ziv 77

```
cursor ← 0
tokens ← []
while cursor < M.size() do
    position, len ← findLongestMatch(M, cursor) ▷ Devolve a posição e tamanho da maior cadeia correspondente
    if len > 0 then ▷ Cria novo token, que aponta para a cadeia encontrada
        tokens.append([position, len, null])
        cursor ← cursor + len
    else
        tokens.append((0, 0, M[cursor])) ▷ Token vazio
        cursor ← cursor + 1
    end if
end while
```

2.2.3 Função "*findLongestMatch*"

O *LZ77* possui um processo de decodificação muito eficiente, porém consome muitos recursos computacionais na codificação. Isto é, ele é mais eficiente para casos onde pretende-se decodificar o arquivo múltiplas vezes, ou ainda decodificá-lo numa máquina com menor poder computacional. O "gargalo" da compressão está na técnica de busca pelo *dicionário* para encontrar a mensagem correspondentes mais longa.

O método convencional utilizado é a **busca linear**, que consiste em comparar cada posição no dicionário com o *lookahead buffer* e selecionar a maior correspondência encontrada. Tome N como o tamanho do dicionário e M como o tamanho do *lookahead buffer*, no pior caso, a busca linear é realizada em tempo $O(NM)$. A busca será executada $M + N$ vezes, isto é, a complexidade geral do algoritmo é $O(N^2M + M^2N)$.

2.2.4 Melhorias na performance da função "*findLongestMatch*"

Uma das possíveis abordagens para melhorar a busca linear é utilizar estruturas de dados auxiliares, que indexam os símbolos do dicionário para tornar a busca mais rápida. Bell (1993) [11] testa diferentes estruturas de dados (árvores trie, hash tables, árvores binárias de busca entre outros). A performance das estruturas diverge entre diferentes tipos de dados, e de que métrica está sendo priorizada (taxa de compressão ou tempo de execução). Para este trabalho utilizaremos a **lista ligada** como estrutura de dados auxiliar, pois ela oferece um bom aumento de performance sem um grande impacto no espaço de armazenamento necessário.

Para construir uma estrutura de dados auxiliar com uma lista ligada, podemos criar uma lista para cada símbolo no dicionário, que contenha os índices onde o símbolo pode ser encontrado. Para limitar a janela de busca, podemos construir a lista como uma *fila*, onde ao atingirmos a capacidade máxima da janela de busca, removemos o menor índice (que estaria no início da fila). Note que, o tamanho máximo da lista depende da quantidade de bits que queremos utilizar para construir o *token*.

Encontramos a maior mensagem correspondente, comparando o *lookahead buffer* com as cadeias de símbolos iniciadas a partir de índices contidos na lista que corresponde ao primeiro símbolo do *lookahead buffer*. Neste contexto, o pior caso ocorrerá quando todos os símbolos no dicionário forem iguais, pois todas as posições no dicionário seriam checadas (levando a uma performance equivalente a da busca linear). Entretanto, para uma base de dados de texto real com grande quantidade de dados isso raramente acontece, e apenas uma quantidade $K \ll N$ de possíveis mensagens são realmente checadas.

2.3 Compressão baseada em palavras

A implementação padrão para a maioria dos algoritmos de compressão utiliza como *alfabeto de origem* elementos de 8 bits (também conhecidos como ***caracteres***). Tal método limita a correlação de cadeias mais longas, e tem sua eficiência limitada quando aplicado a grande quantidade de dados. Em contrapartida, se definirmos cada símbolo do *alfabeto de origem* como uma sequência caracteres, poderíamos tirar vantagem de longas correspondências e talvez obter melhores resultados na compressão.

Quando a natureza dos dados é previamente conhecida, podemos tomar vantagem da sua estrutura para definir os símbolos de uma maneira mais eficiente. Em especial, linguagens "faladas" possuem uma estrutura hierárquica bem definida, que vai desde o agrupamento de letras em sílabas, silabas em palavras, palavras em frases e assim por diante.

Neste contexto, definiremos uma ***palavra*** como uma sequência maximal de *caracteres* alfanuméricos, divididos por sequências de caracteres não alfanuméricos chamados *separadores* (baseado na definição feita por Bentley (1986) [12]). A compressão baseada em palavras é uma modificação dos algoritmos de compressão clássicos, que considera cada palavra como um símbolo (i.e, o *alfabeto de origem* passa a ser composto por *palavras*, não por *caracteres*).

2.3.1 Huffword

O método de compressão *Huffword* é uma modificação da implementação canônica do *código de Huffman*. Desenvolvido por Moffat e Zobel (1994) [13], ele utiliza *palavras* em seu alfabeto de origem.

O algoritmo funciona de maneira bem similar à implementação canônica, o arquivo é pré-processado separando as *palavras* e os *separadores* como duas entradas distintas. Depois o algoritmo de Huffman canônico é aplicado a cada uma das partes, sendo que as probabilidades agora estão associadas às palavras e não mais aos caracteres. A codificação associada a entrada também é armazenada em duas *strings* distintas, assim como as tabelas de códigos também são distintas.

2.3.2 WLZ77

Platos (2008) [14] implementa a versão do LZ77 baseado em palavras (também chamado WLZ77) aplicando uma ideia semelhante à apresentada anteriormente . O arquivo é pré-processado para obter uma lista de *palavras* e *separadores*. Ao contrário do Huffword as palavras e separadores são processados simultaneamente pela implementação canônica do LZ77, gerando uma sequência única de *tokens*.

Vale notar que, neste caso, o último elemento da *tupla* no token não é mais um caractere e sim uma palavra. Isso significa que para o caso dos *tokens vazios*, o tamanho do *token* irá variar de acordo com o tamanho da palavra.

3 Clusterização de dados aplicada a textos

O processo de clusterização de dados consiste em agrupar os objetos (que compõe o conjunto de dados) em n diferentes **agrupamentos** (clusters), de maneira em que os objetos de um mesmo grupo sejam similares e os de grupos diferentes dissimilares. Para isto, faz-se essencial uma definição clara da similaridade entre os objetos, que pode variar dependendo da natureza do dado em análise.

Podemos construir o processo de clusterização de diferentes maneiras [15]: Na **clusterização sem sobreposição** cada objeto deve pertencer a exatamente um cluster. Em contrapartida podemos agrupar os objetos permitindo algumas sobreposições. Podemos ainda clusterizar dados por **hierarquia**, onde os dados são organizados em níveis (quase como uma árvore).

3.1 Similaridade em dados textuais

Conforme introduzido anteriormente, definir uma métrica de similaridade entre os objetos é essencial para o funcionamento dos algoritmos de clusterização. Para o propósito deste trabalho, iremos explorar técnicas que extraem essa similaridade em dados textuais.

Para uma base de dados textual, utilizaremos o termo **"texto bruto"** para se referir ao conteúdo de cada registro na base de dados. Também adotaremos o termo **documento** para partições dos dados (por exemplo, uma linha da tabela). Estenderemos a definição de *palavras* e *separadores* utilizadas no capítulo anterior, definidas sob um ponto de vista linguístico (não relacionados diretamente a notação de conjuntos definida no capítulo 1).

3.1.1 Vetorização TF-IDF

Análise de texto é umas das aplicações mais exploradas no campo dos sistemas inteligentes. Porém, pode ser uma tarefa difícil utilizar texto bruto como entrada para os algoritmos de *machine learning*. Geralmente, tais algoritmos "lidam melhor" com características numéricas.

O processo de transformar documentos em vetores de características numéricas é conhecido como **vetorização** [16]. Em geral, procuramos uma medida que atribua pesos a cada palavra, de acordo com a sua "importância" no documento.

Poderíamos atribuir pesos às palavras de maneira direta e intuitiva, atrelando o peso diretamente a frequência em que aquela palavra ocorre em um documento. Chamamos esta medida de **TF** (*term frequency*).

$$tf(t,d) = \frac{count(t \in d)}{count(d)} \tag{3.1}$$

Onde t é a palavra (ou termo) alvo da medição, e d é o documento que contém a palavra t e pertence a coleção de documentos D.

Entretanto, algumas palavras (como "and" ou "him" no inglês) adicionam pouca ou nenhuma informação relevante para o texto em si, apesar de sua alta frequência, essas palavras são conhecidas como ***stop words***. Remover as *stop words* pode diminuir ruídos nos dados utilizados para o modelo de classificação, sendo um recurso muito importante para a construção de clusters com maior significado semântico.

Para compensar este fator, o **IDF** (*inverse document frequency*) pondera o quão "incomum" é um determinado termo entre diferentes documentos, atribuindo valores baixos a termos que se repetem muito em diferentes documentos [16].

$$idf(t, D) = \log \frac{len(D)}{count(d \in D : t \in d)} \tag{3.2}$$

A métrica **TF-IDF** multiplica as equações 3.2 e 3.1, de maneira que a informação da frequência de um termo é ponderada pela sua "exclusividade" entre os documentos. Obtemos assim uma métrica que aproxima a zero os símbolos menos relevantes, e atribuí valores maiores a símbolos "mais relevantes".

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D) \tag{3.3}$$

3.2 Clusterização K-means

O método de clusterização *k-means* (ou algoritmo de Lloyd's [18]) é um algoritmo iterativo que particiona os dados em k clusters, definidos por **pontos centrais** (onde k é uma das entradas para o algoritmo). A ideia em geral é encontrar k pontos centrais posicionados de maneira a minimizar a distância dos dados aos pontos centrais mais próximos. Podemos definir o k-means da seguinte forma:

1. Escolha os k primeiros pontos centrais. A escolha pode ser **aleatória**, ou utilizando o *k-means++* (algoritmo para inicializar os pontos centrais).

2. Calcule a distância de cada dado em relação a cada ponto central.

3. Atribua cada dado ao ponto central mais próximo.

4. Calcule a distância média entre os pontos e seus respectivos centros para cada cluster, e obtenha novas localizações para os pontos centrais (diminuindo a distância média).

5. Repita os passos 2, 3, e 4 até que os clusters não mudem, ou atingir o número máximo de iterações.

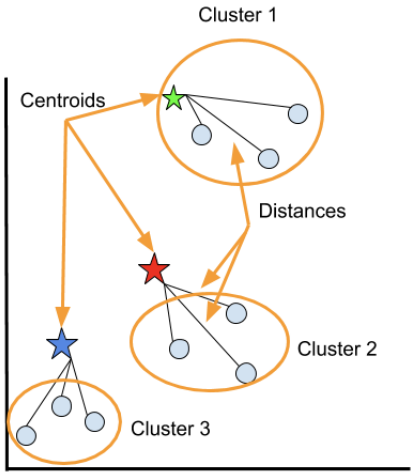

Figura 3 – Exemplo de clusterzacao k-means

3.2.1 Método de elbow

O resultado do *k-means* é diretamente afetado pelo número de clusters definidos na inicialização do algoritmo. Existem diversas formas para determinar o número "ideal" de clusters, uma delas é o **método de elbow** [19].

Neste método, executa-se o *k-means* para um range pré-definido de valores em k. Depois disso, calcula-se a soma das distâncias quadráticas (também chamada de **inércia**) de cada ponto para o seu respectivo ponto central. 0 resultado para cada k é plotado em um gráfico de linha (conforme mostra a figura 4). Seleciona-se o número de clusters que corresponde ao "cotovelo" (elbow) do gráfico. Por intuição, sabemos que o resultado com maior "agrupamento"estará no menor k, entretando, selecionar o menor valor possível para k, certamente levaria a um *overfitting* no modelo.

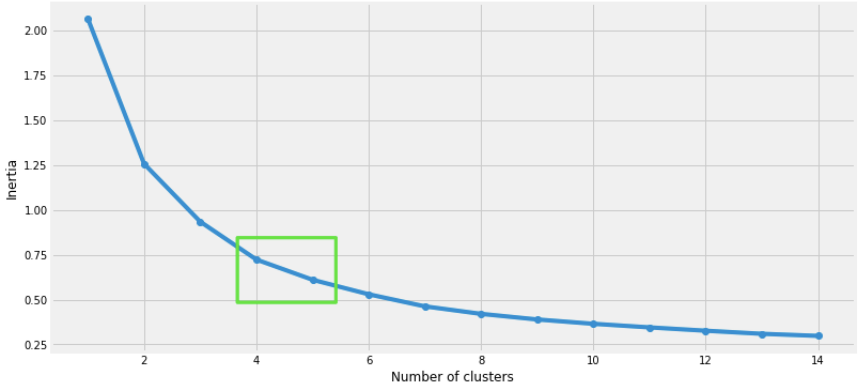

Figura 4 – Método de elbow

Parte III

Aplicação: Melhoria na compressão de dados textuais através da Clusterização

4 Clusterização como pré-processamento na compressão de textos

Nos capítulos anteriores, construímos a estrutura teórica necessária para compreender a *compressão de texto* e sua relação com a *teoria da informação*. Fica claro que os algoritmos apresentados, tomam vantagem de alguma redundância presente na mensagem para representar a informação de maneira mais eficiente. Isto é, a **eficiência** destes algoritmos está intimamente ligada com a **entropia** do texto a ser comprimido, nos dando um indício de que **minimizar** a entropia, pode ser uma forma de **maximizar** a **taxa de compressão**.

O capítulo 3 apresenta uma técnica para organizar o texto em *grupos de mensagens similares* (**clusterização**), o que significa diminuir a variação de termos dentro de cada cluster (ou reduzir a sua **entropia**). Neste capítulo, exploraremos de maneira empírica a relação entre a entropia associada a um texto e a taxa de compressão dos algoritmos apresentados no capítulo 2.

A seguir, será descrito um experimento que utiliza a **clusterização** como pré-processamento de uma base de dados textual, visando uma melhora na performance dos algoritmos de compressão *sem perda*. Foram realizados experimentos utilizando ou não o pré-processamento, a fim de medir o impacto da clusterização para os algoritmos testados (que serão listados em seções posteriores).

Todo o experimento foi desenvolvido em *Python* (devido ao seu vasto ferramental para manipulação de dados) e o código está disponível no GitHub [1]. O código fonte está organizado nas seguintes camadas:

- **dataset**: Contém os arquivos *.csv* que serão carregados como base de dados.
- **compressors**: Classes e scripts com a implementação dos codificadores utilizados.
- **stackexchange_compression_experiments.ipynb**: *Python notebook* com o código fonte de todo o experimento (pré-processamento dos dados, particionamento das bases, aplicação dos algoritmos de compressão, plot de gráficos, entre outros).

[1] Repositório disponível em <https://github.com/lucas170198/pgc-comp-repo/tree/master/implementacao>

4.1 Escolha e tratamento de dados

Para a realização do experimento, a escolha da base de dados é uma etapa primordial. É necessária uma base de dados robusta, com uma grande quantidade de **texto** e que permita a criação de *clusters* contendo assuntos diversos.

Para o experimento, foi selecionada a base de dados "*Transfer Learning o Stack Exchange Tags*", que está disponível no *kaggle*. Trata-se de um conjunto de dados extraídos do website *Stack Exchange* (um grande fórum online, com conteúdo de diversos assuntos e uma grande variedade de dados textuais).

As principais informações contidas na base de dados são o título das questões, o conteúdo da questão (em formato HTML) e as *tags* que classificam o conteúdo em diferentes tópicos (biologia, culinária, criptografia, robótica e viagem). O tamanho total da base de dados original é de aproximadamente **50MB**.

4.1.1 Criação do dataframe principal

Os dados extraídos do *kaggle* estavam originalmente separados em arquivos *csv*, um para cada tópico. Os arquivos foram concatenados e somados em um único *dataframe*, utilizando a biblioteca *pandas*. Para cada arquivo, foram selecionadas **1500** linhas (devido às limitações do hardware utilizado para o teste), o *dataframe* final gerado possui aproximadamente **7MB** de informação.

```
1 # Loading dataframe
2 def load_dataset(dataset_name):
3     f_path = 'dataset/stack-exchange-tags/{dataset}.csv'.format(dataset =
    dataset_name)
4     full_path = os.path.join(os.getcwd(), f_path)
5     return p.read_csv(full_path, index_col=0, nrows=1500)
6
7 ds_names = ['biology', 'cooking', 'crypto', 'diy', 'robotics', 'travel']
8 frames = [load_dataset(name) for name in ds_names]
9 df = p.concat(frames)
```

Listing 4.1 – Carregando base de dados

4.1.2 Sanitização das colunas

Como parte da preparação dos dados para o experimento, as colunas do *dataframe* foram **sanitizadas**. O processo de sanitização consiste em remover as *stopwords*, pontuações, tags *html* e outras informações que podem ser problemáticas para o experimento (principalmente na etapa de clusterização). A partir da sanitização, novas colunas foram geradas no *dataframe* (contendo o prefixo "sanitized').

```
from wordcloud import STOPWORDS
import re\
stop_words = set(STOPWORDS)

## Sanitizing columns
def sanitize_column(data):
    # convert to lower
    data = data.lower()
    # strip html
    data = re.sub(r'\<[^<>]*\>','',data.lower())
    #removing pontuation
    data = re.sub(r'[^a-zA-Z0-9]',' ',data)
    # Remove new lines and tabs
    data = re.sub(r'\s',' ',data)

    #strip data
    data = data.strip()

    #Spling words
    data = data.split()

    #Remove extra blank space
    data = list(filter(lambda s: s != ' ', data))

    # Remove stop words
    data =  list(filter(lambda s: s not in stop_words,data))

    #remove single chars words
    data = list(filter(lambda s: len(s) > 1, data))

    return data
def array_column_to_text(column):
    text = ''
    for w in column:
        for s in w:
            text += ' ' + s
    return text
df['sanitezed_content'] = df['content'].apply(sanitize_column)
```

Listing 4.2 – Sanitização do texto

A biblioteca **worldcloud** foi utilizada para visualizar a distribuição dos termos de maneira gráfica. A Imagem 5 demonstra a representação gráfica das palavras de acordo com a sua frequência.

Figura 5 – Cluster de palavras

4.2 Compressão de textos

Os testes foram realizados com dois algoritmos clássicos de compressão *sem perda* e suas versões baseadas em palavras (**Huffman**, **Huffword**, **LZ77**, **WLZ77**). Neste conjunto de algoritmos estão presente algoritmos baseados em *probabilidades associadas* e *dicionários* Os testes foram realizados sobre quatro algoritmos de compressão **sem perda** (apresentados no capítulo 2), sendo dois baseados em *caracteres* e os outros dois em *palavras* Uma hipótese razoável a ser validada é que com a versão baseada em palavras, o impacto da clusterização na taxa de compressão seja mais significativo. A seguir, serão descritos alguns detalhes específicos da implementação dos compressores para o experimento (dando ênfase nos detalhes não contidos nos algoritmos em pseudocódigo apresentados no capítulo 2).

4.2.1 Classes auxiliares

Para facilitar a reutilização de código na aplicação dos diferentes compressores, foi criada a classe auxiliar (*TextCompressor*). Nesta classe, são expostos os método *encode(text)* e *decode()* que servem como um "contrato" para a implementação dos algoritmos de compressão. Cada classe que modela um compressor herda da classe base *TextCompressor*, criando sua própria implementação dos métodos *encode* e *decode*.

```
1 class TextCompressor:
2     def __init__(self):
3         self.originaltext = None
4         self.stats = None
5     def encode(self, text):
6         self.originaltext = text
7         pass
8     def decode(self):
9         pass
```

Listing 4.3 – Classe TextCompressor

Outra classe auxiliar implementada foi a *CompressionStats*, uma interface para padronizar a leitura das métricas.

```
class CompressionStats:
    def _avg_code(self):
        return (self.compressedtextsize / self.originaltextsize) * 8
    def _compression_rate(self):
        return 100 - (self.compressedtextsize / self.originaltextsize) *
    100
    def __init__(self, originaltext, compressedtext):
        self.originaltextsize = len(originaltext)
        self.compressedtextsize = len(compressedtext)
    def __str__(self) -> str:
        return stats_text.format(csize=self._avg_code(), osize=self.
    originaltextsize, nsize=self.compressedtextsize, crate=self.
    _compression_rate())
```

Listing 4.4 – Implementaçào da classe base CompressionStats

4.2.2 Compressão baseada em palavras

Para a implementação dos algoritmos baseados em **palavras**, boa parte do código fonte utilizado nos algoritmos "canônicos" foram reutilizados. Como o *Python* é uma linguagem de **tipagem dinâmica**, a implementação "canônica" interpreta a entrada como um vetor de *símbolos* (sem necessariamente distinguir o tipo de símbolo). Com isso, podemos reaproveitar grande parte das funções escritas independente do alfabeto de origem utilizado.

O *Huffword* computa as *palavras* e *separadores* como duas entradas distintas. Para dividir a entrada entre *palavras* e *separadores*, foi utilizada a biblioteca *re*, que permite executar buscas em *strings* utilizando *regex*. A mesma lógica é aplicada ao *WLZ77*, diferindo do fato que desta vez, as *palavras* e *separadores* são processadas em um único vetor.

```
def _build_huffword_code(seq):
    freqs = frequency_dictionary(seq)
    huff_tree = canonical.build_huff_tree(freqs)
    code_table= canonical.build_code_table(huff_tree)
    return code_table

def huffword_encode(text):
    # Words huff tree
    words = re.findall(r'\w+', text)
    words_code = _build_huffword_code(words)

    # Non words huff tree
    nonwords = re.findall(r'\W+', text)
    nonwords_code = _build_huffword_code(nonwords)
```

```
15
16      encoded_string = ""
17
18      # When text start with non-word append 0, otherwise append 1
19      starts_with = 0
20      if text.startswith(words[0]):
21          starts_with += 1
22
23      # Append starts with as the first char
24      encoded_string += str(starts_with)
25
26      # Encode intercalating words and nonwords
27      w_index = 0
28      nw_index = 0
29
30      #When starts with non words
31      if not starts_with:
32          encoded_string += nonwords_code[nonwords[0]]
33
34      while w_index < len(words) or nw_index < len(nonwords):
35          if w_index < len(words):
36              word = words[w_index]
37              encoded_string += words_code[word]
38              w_index += 1
39
40          if nw_index < len(nonwords):
41              nonword = nonwords[nw_index]
42              encoded_string += nonwords_code[nonword]
43              nw_index += 1
44
45      return {'encoded' : encoded_string,
46              'words_meta' : (words, words_code),
47              'non_words_meta': (nonwords, nonwords_code)}
```

Listing 4.5 – Função *encode* para o *Huffword*

4.3 Partição de dados

Os testes com os algoritmos de compressão foram realizados com duas estratégias diferentes de partição. Na primeira, os dados são particionados **aleatoriamente** em n partes. Já para o teste com a clusterização, os mesmos dados são particionados em n *clusters* criados pelo algoritmo *k-means*. Em ambos os casos, cada algoritmo de compressão foi executado sobre as n partições (grupos) e as métricas foram computadas por média média aritmética simples.

4.3.1 Partição aleatória

Para o teste com partição aleatória, o *dataframe* original foi dividido em n partes iguais. É importante que tais divisões sejam aleatórias, a fim de evitar possíveis ruídos no teste.

Para garantir a aleatoriedade, foi utilizado o método *.sample()* da biblioteca *pandas*. Este método cria uma amostra aleatória de itens para um determinado eixo do objeto. O parâmetro $frac = 1$ faz com que o método retorne uma amostra com todos os dados originais (porém de maneira randômica). Em seguida, o vetor foi fracionado pela função *array_split()* da biblioteca *numpy*.

```
# Data to compress: Compress the original data raw
raw_text = df['content']

#Shuffle raw text
shuffle = raw_text.sample(frac=1)
partitions = np.array_split(shuffle, n_partitions)
```

Listing 4.6 – Partição aleatória de dados

4.3.2 Clusterização

Para o teste com partição por clusterização, o algoritmo *k-means* foi utilizado. A clusterização se inicia transformando os dados textuais em **vetores de características**, utilizando a técnica de vetorização *TF-IDF*. O vetor de características passa por uma redução de dimensionalidade via *PCA* (reduzindo os dados em **duas** componentes principais, para melhor visualização).

```
def identity_tokenizer(text):
  return text

vect = TfidfVectorizer(tokenizer=identity_tokenizer,lowercase=False)
tf_idf = vect.fit_transform(df['sanitezed_content'].values)
tf_idf_norm = normalize(tf_idf) # To PCA running
tf_idf_arr = tf_idf_norm.toarray()
p.DataFrame(tf_idf_arr, columns=vect.get_feature_names()).head(10)

# PCA component reduction
pca = PCA(n_components=2)
Y = pca.fit_transform(tf_idf_arr)
```

Listing 4.7 – Vetorização dos dados

Com os dados vetorizados em duas dimensões, o *método de elbow* é utilizado para encontrar o número “ideal” de clusters. O gráfico de elbow foi construído com os número

de clusters variando entre 1 e 7. De acordo com os resultados obtidos, foi utilizado $n = 3$ como parâmetro para o *k-means*.

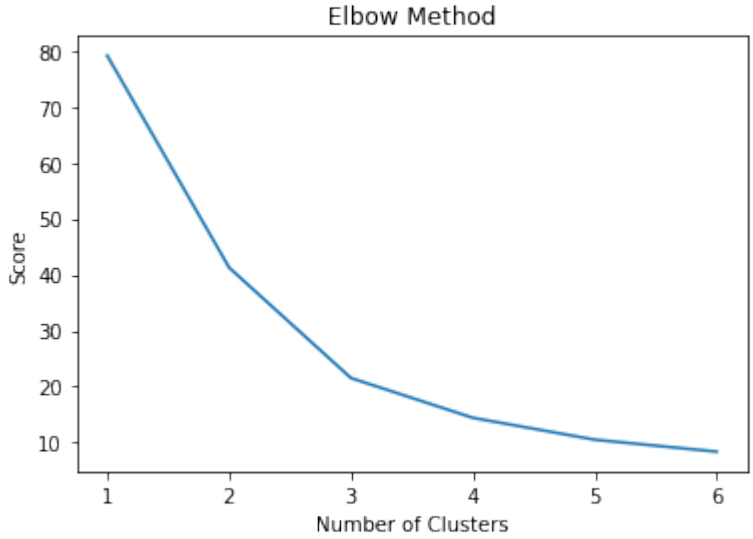

Figura 6 – Método de Elbow

Por fim, o *k-means* é aplicado a base de dados através do objeto $sklearn.cluster.KMeans$. O parâmetro *n_clusters* recebe o número de clusters encontrado no passo anterior. O parâmetro *init* indica a utilização do *k-means++* na inicialização dos pontos centrais. Já o parâmetro *max_inter*, define como 50 o número máximo de iterações do algoritmo *k-means*.

```
1 kmeans = KMeans(n_clusters=n_clusters, init='k-means++', max_iter=50)
2 fit_data = kmeans.fit(Y)
3 pred_class = kmeans.predict(Y)
4
5 display(fit_data.labels_)
6 df['Cluster'] = fit_data.labels_
```

Listing 4.8 – Vetorização dos dados

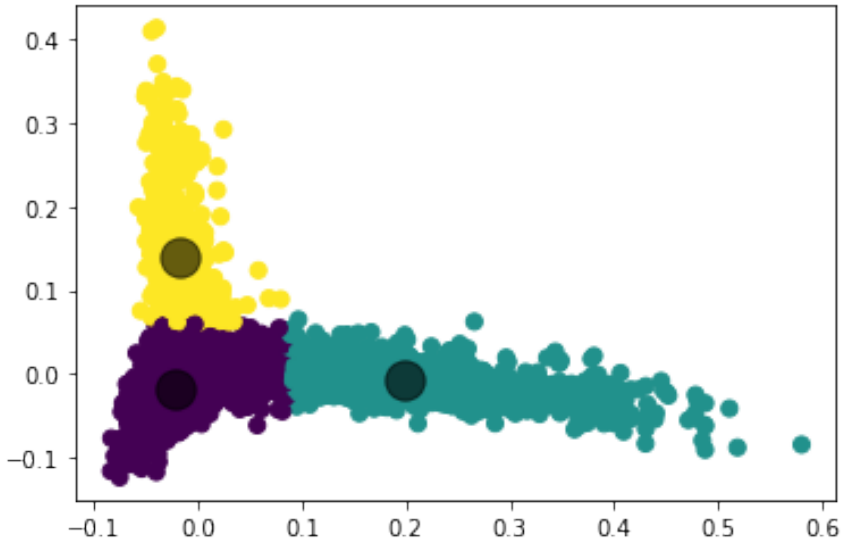

Figura 7 – Distribuição dos clusters (dados vetorizados)

4.4 Resultados

4.4.1 Taxa de compressão

Para validar a hipótese inicial, os dados particionados (aleatoriamente ou via compressão) foram comprimidos utilizando os algoritmos citados anteriormente. Cada partição foi processada de maneira **individual**, portanto, a performance geral de cada algoritmo foi obtida a partir da média aritmética entre os resultados das n partições.

Para compressão de dados sem perda o objetivo é reduzir o tamanho da mensagem, sem haver perda de informações. Assim, uma das principais métricas de sucesso nestes algoritmos é a **taxa de compressão**. Definimos a **taxa de compressão** como a razão entre o tamanho do documento codificado e o original.

$$taxa\ de\ compressão = \frac{tamanho\ do\ documento\ codificado}{tamanho\ do\ documento\ original} \tag{4.1}$$

Definimos Δ como o impacto da clusterização na taxa de compressão, que é obtido pela diferença entre os resultados com e sem a clusterização.

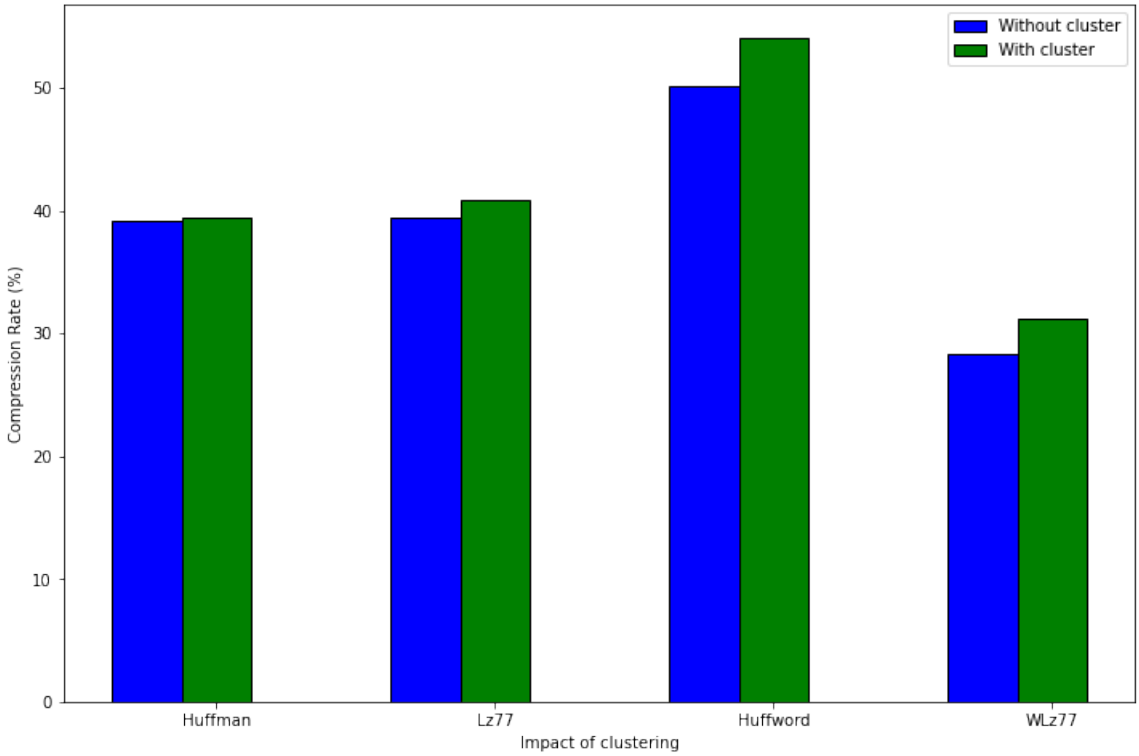

Figura 8 – Impacto da clusterização na taxa de compressão

Tabela 2 – Impacto da clusterização na taxa de compressão

Algoritmo	% compressão	% compressão (clusters)	Δ
Huffman	39.214536	39.395034	0.180498
Huffword	50.133603	54.096327	3.962724
LZ77	39.470517	40.795727	1.325210
WLZ77	28.369986	31.173907	2.803921

Conforme previsto no capítulo 2, a tabela 2 mostra que a clusterização teve um maior impacto nos algoritmos baseados em **palavras**. Em especial, o maior Δ foi alcançado para o *Huffword*,com uma melhoria de **quase 4%**. Curiosamente, a versão baseada em palavras do LZ77 se mostrou menos efetiva no geral do que sua versão canônica.

4.4.2 Tempo de execução

Outra métrica que auxilia na comparação entre os algoritmos é o **tempo de execução**. Para isso, foi utilizado o comando *%time%* do *jupyter notebook* (que mede o tempo de execução da célula em que o comando foi inserido).

Tabela 3 – Tempo de execução dos algoritmos de compressão

Algoritmo	Tempo de execução	Partição
Huffman	2.18s	aleatória
Huffword	2.8s	aleatória
LZ77	8min 24s	aleatória
WLZ77	1h 37min 41s	aleatória
Huffman	3.08s	k-means
Huffword	3.66s	k-means
LZ77	12min 24s	k-means
WLZ77	2h 30min 9s	k-means

Fica claro que as versões baseadas em palavras e clusterizadas via *k-means* consomem mais tempo de execução. Este dado já era esperado, já que o aumento de símbolos semelhantes leva a mais *tokenizações* e por consequência maior tempo de processamento.

4.5 Conclusão

Neste trabalho, combinamos o uso da compressão **baseada** em palavras com a clusterização, em busca de uma melhora na taxa de compressão para base textuais. Os resultados experimentais confirmam as hipóteses levantadas previamente, e chega-se à conclusão de que a clusterização impacta **positivamente** a **taxa de compressão** de textos. Entretanto, os experimentos também mostraram um aumento nos recursos consumidos (tanto pelo pré-processamento, quanto pela compressão em si). Espera-se que os resultados apresentados neste trabalho sejam úteis na implementação de ferramentas de compressão otimizadas, principalmente para grandes bases de dados textuais.

4.6 Trabalhos futuros

O intuito principal deste trabalho era apresentar a clusterização como potencializador para alguns algoritmos clássicos de compressão sem perda. Algoritmos mais modernos (como BZip e WBW), otimizados especificamente para grandes textos, podem trazer resultados ainda melhores com a compressão. Utilizar outros métodos de clusterização mais complexos (como a clusterização hierárquica), também podem trazer resultados diferentes para os testes.

Referências

[1] Hirschberg, D.S; Lelewer D.A; *Data compression,* Computing Surveys 19.3, 1987. Citado 3 vezes nas páginas 5, 6 e 7.

[2] Blelloch G.E; *Introduction to Data Compression,* Carnegie Mellon, (2013). Citado 6 vezes nas páginas 6, 9, 12, 13, 18 e 19.

[3] Shannon C.E; *A Mathematical Theory of Communication* The Bell System Technical Journal, (1948). Citado na página 9.

[4] Bertsekas D.P; Tsitsiklis J.N; *Introduction to Probability* M.I.T, Lecture Notes Course 6.041-6.431, (2000). Citado na página 8.

[5] COMPRESSÃO em HTTP; MDN Web Docs, 2021; Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Compression>; Acesso em: 16 de dez de 2021. Citado na página 1.

[6] COMPRESSION Techniques; Cambridge University, Raspberry Pi Foundationand National Centre for Computing education; Disponível em : <https://isaaccomputerscience.org/concepts/data_compr_loss?examBoard=all&stage=all.> Acesso em: 30 de maio de 2022. Citado na página 1.

[7] COMPACTAÇÃO de dados; Microsoft Docs. 2021; Disponível em: <https://docs.microsoft.com/pt-br/sql/relational-databases/data-compression/data-compression?view=sql-server-ver15>; Acesso em: 16 de dez de 2021. Citado na página 1.

[8] CLUSTERING in Machine Learning; Google Developers; Disponível em: <https://developers.google.com/machine-learning/clustering/overview>; Acesso em: 30 de maio de 2022. Citado na página 1.

[9] Huffman, D; *A Method for the Construction of Minimum-Redundancy Codes,* Proceedings of the IRE, (1952). Citado 2 vezes nas páginas 18 e 19.

[10] Ziv J; Lempel A; *A Universal Algorithm for Sequential Data Compression,* IEEE Transactions on Information Theory, (1977). Citado na página 21.

[11] Bell, T; Kulp, D; *Longest-match String Searching for Ziv-Lempel Compression,* Department of Computer Science, University of Canterbury, (1993). Citado na página 22.

[12] Bentley J.L; Sleator D.D; Tarjan R.E; Wei V.K; *A Locally Adaptive Data Compression Scheme,* CACM 29, (1986). Citado na página 23.

[13] Moffat A; Matthias P; *Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems,* ACM Transactions on Information Systems, (2020). Citado na página 23.

[14] Platos J; Dvorský, J; *Word-Based Text Compression,* (2008). Disponível em: <https://arxiv.org/abs/0804.3680>; Acesso em: 30 de maio de 2022. Citado na página 24.

[15] CLUSTERING Algorithms. Google Developers; Disponível em: <https://developers.google.com/machine-learning/clustering/clustering-algorithms>; Acesso em: 30 de maio de 2022. Citado na página 25.

[16] Qaiser S ; Ali R.; *Text Mining: Use of TF-IDF to Examine the Relevance of Words to Documents,* International Journal of Computer Applications. 181 (2018) Citado na página 26.

[17] Jolliffe I.T; Cadima J; *Principal component analysis: a review and recent developments,* The Royal Society, (2018). http://doi.org/10.1098/rsta.2015.0202 Nenhuma citação no texto.

[18] Lloyd; Stuart P; *Least Squares Quantization in PCM,* IEEE Transactions on Information Theory. Vol. 28, (1982). Citado na página 27.

[19] Humaira H; Rasyidah R; *Determining The Appropiate Cluster Number Using Elbow Method for K-Means Algorithm* (2020). Citado na página 28.