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

Segundo a ABNT, o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. Umas 10 linhas (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chaves: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Sumário

Introdução

Motivação

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Objetivos

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Parte I

Fundamentação Teórica

1 Conceitos e definições fundamentais em compressão de dados

Este capítulo apresenta algumas definições e conceitos fundametais 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 formadas pelas possíveis combinações de A. Um **código** C mapeia uma **mensagem** $m \in M$ para uma **cadeia de códigos** do alfabeto W (onde M é o **alfabeto de origem** e W **alfabeto código**). Chamaremos essa cadeia de códigos de **palavra-código**, denotada por w . Dentro deste contexto, definimos o **comprimento** da palavra-código w como $l(w)$, um **inteiro positivo** que representa o tamanho da cadeia que compõe a palavra-código 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 o código C da seguinte forma:

Tabela 1 – Tabela do código C

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

Isto é, o código C pode ser representado como uma função $\boldsymbol{C : M \rightarrow W^+}$. Neste exemplo, o **comprimento** da palavra-código associado à mensagem "a" é dado por $l(C(a)) = 1$.

As palavras-código associadas a cada mensagem podem ter um tamanho *fixo* ou *variável*. Códigos nos quais os alfabetos 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**. Provavelmente o exemplo mais conhecido de código de **comprimento fixo** seja código ASCII, que mapea 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, afim de reduzir o tamanho médio da *string* original ao codificá-la.

1.1.1 Códigos unicamente decodificáveis e livres de prefixo

Um código é **distinto** se pode ser representado como uma função **bijetora**, i.e, $\forall\ m_1, m_2 \in \mathrm{M}$, $C(m_1) \neq C(m_2)$. Um código distinto é dito **unicamente decodificável** se qualquer palavra-código pode ser identificada quando imersa em uma sequência de palavras-código.

Um **código livre de prefixo** é um código unicamente decodificável em que nenhuma palavra-código é prefixo de outra. Por exemplo, o código que possuí 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 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.

Um código livre de prefixo em que $W := \{0,\ 1\}$ pode ser modelado por uma **arvore binária**. Imagine que cada mensagem $m \in M$ é uma folha. A palavra-código $C(m)$ é o caminho p da raiz até a folha m, de maneira em que, para cada nó percorrido concatenamos um bit a p ("0"quando o nó está a esquerda e "1"quando está a direita), tal árvore é chamada **árvore do código livre de prefixo**. Tomando como exemplo o código representado pela Tabela 3 (claramente um código livre de prefixo), podemos representá-lo por uma árvore livre de prefixo da seguinte forma:

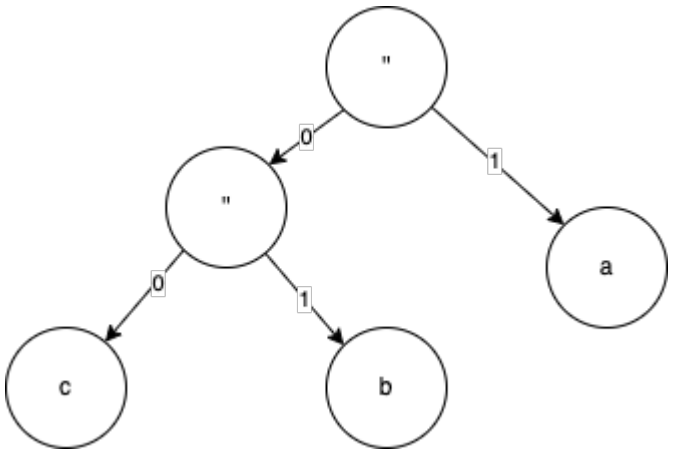

Figura 1 – Árvore livre de prefixo

1.2 Relações fundamentais com a Teoria da Informação

A codificação é comumente divida 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 *bias* apontado pela modelagem, e utiliza as probabilidades associadas para reduzir a quantidade de dados necessária para representar a mesma informação (substituindo as mensagens que

ocorrem com maior frequência por símbolos menores). Isso significa que, a codificação está diretamente ligada as probabilidades associadas a cada mensagem.

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 para alcançar uma codificação eficiente.

1.2.1 Distribuição de Probabilidade e Esperança

Dado um experimento e um espaço amostral Ω, uma **variável aleatória** X associa um número real a cada um dos possíveis resultados em 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. Podemos descrever melhor uma variável aleatória, atribuindo probabilidades sobre os valores que esta pode assumir. Esses valores são atribuídos pela **função de densidade de probabilidade**, denotada por p_X. Portanto, a probabilidade do evento $\{\ X = x\ \}$ é a função de distribuição de probabilidade aplicada a x, *i.e*, $p_X(x)$.

$$p_X(x) = P(\{X = x\}) \tag{1.1}$$

Note que, a variável aleatória pode assumir qualquer um dos valores no espaço amostral que possuem uma probabilidade $P > 0$, portanto

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

O **valor esperado** (ou **esperança**) da variável aleatória X é definido como

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

1.2.2 Comprimento médio do código

Seja p a distribuição de probabilidade associada ao alfabeto de origem M. Assuma que C é um código tal que $C(m) = w$, definimos o **tamanho médio** de C como:

$$l_a(C) = \sum_{m \in M} p(m) l(C(m)) \tag{1.4}$$

Um código C unicamente decodificável é **ótimo** se $l_a(C)$ é mínimo, isto é, para qualquer código unicamente decodificável C' temos que:

$$l_a(C) \leq l_a(C') \tag{1.5}$$

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. Dado um espaço de probabilidade X e a função p sendo a distribuição de probabilidade associada a X, definimos **entropia** como:

$$H(X,p) = \sum_{x \in X} p(x) \log_2 \frac{1}{p(x)} \tag{1.6}$$

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

Shannon (incluir referencia papper do Shannon) aplica o mesmo conceito de entropia no contexto da Teoria da Informação, "substituindo"o conjunto de estados S pelo conjunto de mensagem M, isto é, M é interpretado como um conjunto de possíveis mensagens, tendo como $p(m)$ a probabilidade de $m \in M$. Baseado na mesma premissa, Shannon mede a informação contida em uma mensagem da seguinte forma:

$$i(m) = \log_2 \frac{1}{p(m)}. \tag{1.7}$$

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**.

Lema 1 (Desigualdade de Kraft-McMillan)**.** ***Kraft.****. Para qualquer conjunto* L *de comprimento códigos que satisfaça:*

$$\sum_{l \in L} 2^{-l} \leq 1.$$

Existe ao menos um código livre de prefixo tal que, $|w_i| = l_i \ \ , \forall w \in W^+$.

Kraft-McMillan. *Para todo código binário unicamente decodificável* $\mathbf{C : M \rightarrow W^+}$.

$$\sum_{w \in W^+} 2^{-l(w)} \leq 1.$$

Demonstração.

Desigualdade de Kraft Sem perda de generalidade, suponha que os elementos de L estão ordenados de maneira em que:

$$l_1 \leq l_2 \leq ... \leq l_n$$

Agora vamos construir um código livre de prefixo em uma ordem crescente de tamanho, de maneira em que $l(w_i) = l_i$. Sabemos que um código é livre de prefixo se e somente se ,existe uma palavra-código w_j tal que nenhuma das palavras-código anteriores (w_1, w_2, w_{j-1}) são prefixo de w_j.

Sem as restrições de prefixo, uma palavra-código de tamanho l_j poderia ser construída de 2^{l_j} maneiras diferentes. Com a restrição apresentada anteriormente, considerando uma palavra w_k anterior a w_j (i.e, $k < j$), existem $2^{l_j - l_k}$ possíveis palavras-código em que w_k seria um prefixo, e que portanto não podem pertencer ao código. Chamaremos tal conjunto de "palavras-código proibidas". Vale notar que os elementos do conjunto de palavras-código proibidas são excludentes entre si, pois se duas palavras-código menores que j fossem prefixo da mesma palavra-código, elas seriam prefixos entre si.

Dito isto, podemos definir o tamanho do conjunto de palavras-código proibida para w_j.

$$\sum_{i=1}^{j-1} 2^{l_j - l_i}$$

A construção do código livre de prefixo é possível se e somente se, existir ao menos uma palavra-código de tamanho $j > 1$ que não está contida no conjunto das palavras-código proibidas.

$$2^{l_j} > \sum_{i=1}^{j-1} 2^{l_j - l_i}$$

Como o domínio do problema apresentado está restrito aos inteiros não negativos, podemos afirmar que:

$$\begin{aligned} 2^{l_j} > \sum_{i=1}^{j-1} 2^{l_j - l_i} &= 2^{l_j} \geq \sum_{i=1}^{j-1} 2^{l_j - l_i} + 1 \\ &= 2^{l_j} \geq \sum_{i=1}^{j} 2^{l_j - l_i} \\ &= 1 \geq \sum_{i=1}^{j} 2^{-l_i} \\ &= \sum_{i=1}^{j} 2^{-l_i} \leq 1 \end{aligned}$$

Substituindo n em j, chegamos a desigualdade de Kraft.

$$\sum_{l \in L} 2^{-l} \leq 1.$$

Note que os argumentos utilizados para a construção da prova possuem dupla-equivalência, portando concluem a prova nos dois sentidos.

Kraft-McMillan Suponha um código **unicamente decodificável** C qualquer, e faça $l_{max} = \max_w l(w)$.

Agora considere uma sequência de k palavras-código de $\boldsymbol{C : M \to W^+}$ (onde k é um inteiro positivo). Observe que:

$$\begin{aligned}
(\sum_{w\in W^+} 2^{-l(w)})^k &= (\sum_{w_1} 2^{-l(w_1)}) \cdot (\sum_{w_2} 2^{-l(w_2)}) \cdot ... \cdot (\sum_{w_k} 2^{-l(w_k)}) \\
&= \sum_{w_1}\sum_{w_2}...\sum_{w_k}\prod_{j=1}^{k} 2^{-l(w_j)} \\
&= \sum_{w_1,...,w_k} 2^{-\sum_{j=1}^{k} l(w_j)} \\
&= \sum_{w_k} 2^{-l(w^k)} \\
&= \sum_{j=1}^{k\cdot l_{max}} |\{w_k | l(w_k) = j\}| \cdot 2^{-j}.
\end{aligned}$$

Sabemos que existem 2^j palavras-código de tamanho j, isto é, $|\{w_k|l(w_k) = j\}| = 2^j$. Para que o código seja unicamente decodificável obtemos o seguinte limite superior:

$$(\sum_{w\in W^+} 2^{-l(w)})^k \leq \sum_{j=1}^{k\cdot l_{max}} 2^j \cdot 2^{-j} = k \cdot l_{max}$$

Logo,

$$\sum_{w\in W^+} 2^{-l(w)} \leq (k \cdot l_{max})^{\frac{1}{k}}$$

Note que a desigualdade é valida para qualquer $k > 0$ inteiro. Aproximando k ao infinito, obtemos a desigualdade de Kraft-McMillan.

$$\sum_{w\in W^+} 2^{-l(w)} \leq \lim_{k\to\infty} (k \cdot l_{max})^{\frac{1}{k}} = 1.$$

□

Lema 2 (Entropia como limite inferior para o comprimento médio)**.** *Dado um conjunto de mensagens* M *associado a uma distribuição de probabilidades* p *e um código unicamente decodificável* C*.*

$$H(M, p) \leq l_a(C)$$

Demonstração. Queremos provar que $H(M,p) - l_a(C) \leq 0$, dado que $H(M,p) \leq l_a(C) \Leftrightarrow H(M,p) - l_a(C) \leq 0$.

Substituindo a equação 1.6 em *H(M, p)* e 1.4 em $l_a(C)$, temos:

$$\begin{aligned} H(M,p) - l_a(C) &= \sum_{m \in M} p(s) \log_2 \frac{1}{p(m)} - \sum_{m \in M, w \in W^+} p(m) l(w) \\ &= \sum_{m \in M, w \in W^+} p(m) \left(\log_2 \frac{1}{p(m)} - l(w) \right) \\ &= \sum_{m \in M, w \in W^+} p(m) \left(\log_2 \frac{1}{p(m)} - \log_2 2^{l(w)} \right) \\ &= \sum_{m \in M, w \in W^+} p(m) \log_2 \frac{2^{-l(w)}}{p(m)} \end{aligned}$$

A **Desigualdade de Jansen** afirma que se uma função *f(x)* é côncava, então $\sum_i p_i \ f(x_i) \leq f(\sum_i p_i \ x_i)$. Como a função $\log_2$ é côncava, podemos aplicar a Desigualdade de Jansen ao resultado obtido anteriormente.

$$\sum_{m \in M, w \in W^+} p(m) \log_2 \frac{2^{-l(w)}}{p(m)} \leq \log_2 (\sum_{m \in M, w \in W^+} 2^{-l(w)})$$

Agora aplicamos a desigualdade de Kraft-McMillan, e concluímos que:

$$H(M,p) - l_a(C) \leq \log_2 (\sum_{m \in M, w \in W^+} 2^{-l(w)}) \Rightarrow H(M,p) - l_a(C) \leq 0.$$

□

Lema 3 (Entropia como limite superior para o comprimento médio de um código livre de prefixo ótimo)**.** *Dado um conjunto de mensagens* M *associado a uma distribuição de probabilidades* p *e um código livre de prefixo ótimo* C*.*

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

Demonstração. Sem perda de generalidade, para cada mensagem $m \in M$ faça *l(m)* $= \left\lceil \log_2 \frac{1}{p(m)} \right\rceil$. Temos que:

$$\begin{aligned} \sum_{m \in M} 2^{-l(m)} &= \sum_{m \in M} 2^{-\left\lceil \log_2 \frac{1}{p(m)} \right\rceil} \\ &\leq \sum_{m \in M} 2^{-\log_2 \frac{1}{p(m)}} \\ &= \sum_{m \in M} p(m) \\ &= 1 \end{aligned}$$

De acordo com a desigualdade de Kraft-McMillan existe um código livre de prefixo *C'* com palavras-código de tamanho *l(m)*, portanto:

$$\begin{aligned} l_a(C') &= \sum_{m\in M', w\in W'^+} p(m)l(w) \\ &= \sum_{m\in M', w\in W'^+} p(m)\left\lceil \log_2 \frac{1}{p(m)} \right\rceil \\ &\leq \sum_{m\in M', w\in W'^+} p(m)(1+\log_2 \frac{1}{p(m)}) \\ &= 1 + \sum_{m\in M', w\in W'^+} p(m)\log_2 \frac{1}{p(m)} \\ &= 1 + H(M) \end{aligned}$$

Pela definição de código livre de prefixo ótimo, $l_a(C) \leq l_a(C')$, isto é:

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

□

Parte II

Algoritmos de compressão e clusterização

2 Algoritmos de compressão sem perda

Os algoritmos de compressão podem ser categorizadas 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 uteis 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 uteis para o propósito do presente trabalho.

2.1 Código de Huffman

O **algoritmo de Huffman** (desenvolvido por David Huffman em 1952) 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. Considere o conjunto de símbolos M, com p_i sendo 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 arvore na floresta
end for

while $Forest.size > 1$ **do**
  $T1 \leftarrow ExtractMin(Forest)$ ▷ Retorna 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 interações (onde n é o tamanho do conjto M). Considerando que as funções *ExtractMin()* e *.append()* foram construídas a partir de uma fila de prioridades de **heap**, o algoritmo será executado em $O(n \log_2 n)$.

2.1.2 Corretude

O teorema a seguir (escrito por Huffman TODO REF ARTIGO) mostra que os códigos de Huffman são ótimos e livres de prefixo.

Lema 4. *Seja C um código ótimo livre de prefixo, com $\{p_1, p_2, ..., p_n\}$ sendo a distribuição de probabilidades associada ao código. Se $p_i > p_j$ então $l(w_i) \leq l(w_j)$*

Demonstração. Para efeito de contradição, assuma que $l(w_i) > l(w_j)$. Agora construa um novo código C', trocando w_i por w_j. Dado l_a como o comprimento médio do código C, o código C' terá o seguinte comprimento:

$$\begin{aligned} l'_a &= l_a + p_j(l(w_i) - l(w_j)) + p_i(l(w_j) - l(w_i)) \\ &= l_a + (p_j - p_i)(l(w_i) - l(w_j)) \end{aligned}$$

Pelas suposições feitas anteriormente o termo $(p_j - p_i)(l(w_i) - l(w_j))$ seria negativo, contradizendo o fato do código C ser um código ótimo e livre de prefixo (pois neste caso $l'_a > l_a$).

Nota* : Perceba que em uma árvore de Huffman, o tamanho da palavra código w_i também representa seu nível na árvore. □

Teorema 5. *O algoritmo de Huffman gera um código ótimo livre de prefixo.*

Demonstração. A prova se dará por indução sobre o número de mensagens pertencentes ao código. Vamos mostrar que se o Algoritmo de Huffman gera um código livre de prefixo ótimo para qualquer distribuição de probabilidades com n símbolos, então o mesmo ocorre para $n + 1$ símbolos.

Caso Base. Para n = 2 o teorema é trivialmente satisfeito considerando um código que atribui um bit pra cada símbolo do código.

Passo indutivo. Pelo lema 4 sabemos que as menores probabilidades estão nos menores níveis da árvore de Huffman (por ser uma árvore binária completa, o seu menor nível deve possuir ao menos dois nós). Já que esses nós possuiriam o mesmo tamanho, podemos muda-los de posição sem afetar o tamanho médio do código, concluindo assim que estes são nós **irmãos**.
Agora defina um conjunto de símbolos M de tamanho $n + 1$ onde T é a árvore de prefixo ótima construída a partir do Algoritmo de Huffman aplicado em M. Vamos chamar os dois nós de menor probabilidade na árvore de x e y (que pelo argumento anterior, são nós irmãos). Construiremos uma nova árvore T' a partir de T removendo os nós x e y, fazendo assim que o pai destes nós, que chamaremos de z, seja o de menor probabilidade (de acordo com a definição do Algoritmo de Huffman, $p_z = p_y + p_x$). Considere k como a profundidade de z, temos:

$$\begin{aligned} l_a(T) &= l_a(T') + p_x(k+1) + py(k+1) - p_z k \\ &= l_a(T') + p_x + p_y \end{aligned}$$

Sabemos pela hipótese de indução que $l_a(T')$ é mínimo, pois T' tem o tamanho n e foi gerada pelo algoritmo de Huffman. Note que independente da ordem que forem inseridos, os nós x e y irão adicionar a constante $p_z = p_x + p_y$ no peso médio do código. Como $l_a(T')$ é mínimo para um conjunto de símbolos de tamanho n e seu nó de menor peso tem distribuição de probabilidade p_z, $l_a(T)$ também é mínimo para o conjunto de símbolos M e logo T é ótimo e livre de prefixo. □

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 **LZ88**, 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 tem um funcionamento parecido, que se resume em substituir partes da entrada por referências à partes iguais anteriormente processadas, e diferem na maneira em que procuram por repetições a serem substituídas.

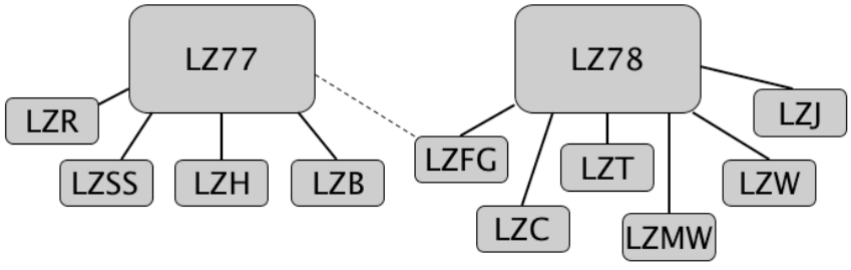

Figura 2 – Familia de algoritmos Lempel Ziv

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 codificadas. 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 *tolkens* 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 pra 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) ▷ Retorna 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 decodifica-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. (TODO: Citacao do artigo) testa diferentes estruturas de dados (arvores trie, hash tables, arvores binárias de busca entre outros). Para este projeto utilizaremos a **lista ligada** como estrutura de dados auxiliar (principalmente por possuir uma implementação mais simples).

Para construir uma estrutura de dados auxiliar com lista ligada, podemos criar uma lista para cada símbolo no dicionário, que contenha os indicies 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 indicie (que estaria no inicio 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 indicies 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 todos 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 aconteceria, e apenas uma quantidade $K \ll N$ de possíveis mensagens seriam 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 *carac-*

teres alfanuméricos, divididos por sequências de caracteres não alfanuméricos chamados *separadores* (baseado na definição feita por Bentley et al. TODO referencia). 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 (Incluir referencia) em 1994, 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 as 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 destintas.

2.3.2 WLZ77

(TODO REF ARTIGO) 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: 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).

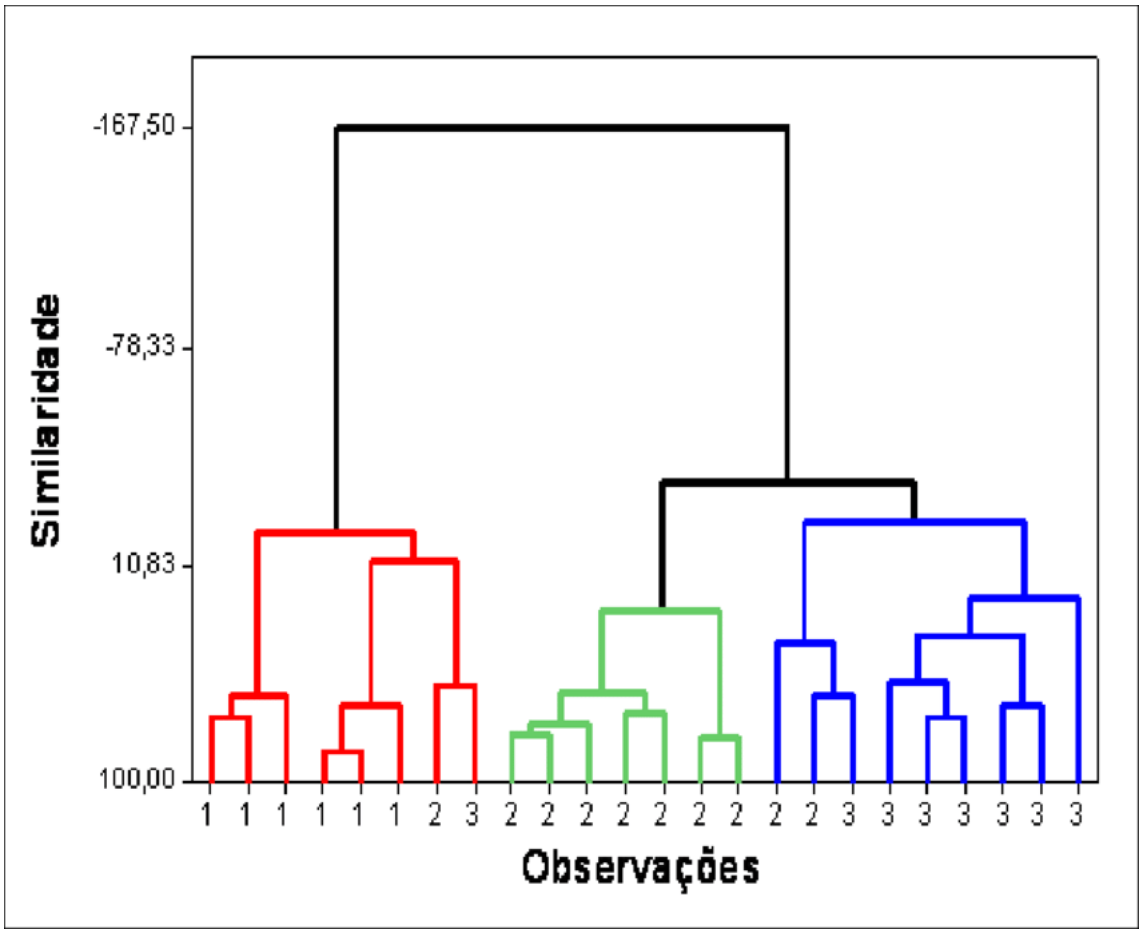

Figura 3 – Exemplo de agrupamento hierárquico

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 projeto, iremos explorar técnicas que extraem essa similaridades em dados textuais.

Para uma base 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 notações de conjuntos definida no capítulo 1).

3.1.1 Vetorização TF-IDF

Analise de texto é umas das aplicação 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**. Em geral, procuramos uma medida que atribua pesos a cada palavra, de acordo com a sua "importância" no documento.

Poderíamos atribuir pesos as 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.

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

A métrica **TF-IDF** multiplica as equações 3.2 e 4.4.1, 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 Redução de Dimensionalidade

Em casos em que a base de dados possuem uma quantidade elevadas de dimensões, reduzir a dimensionalidade significa procurar variáveis que possuem dependência entre si. No geral, busca-se manter apenas características relevantes aos dados, reduzindo sua complexidade e o poder computacional necessário para processar estes dados.

3.2.1 PCA

PCA (ou *principal component analysis*) é formada por um procedimento algébrico, que reduz as variáveis correlacionadas em um conjunto de variáveis não relacionadas (linearmente) chamadas *principal components* (PCs), iterativamente.

No contexto da clusterização, o PCA é utilizado principalmente para transformar dados com muitas dimensões em uma representação com menores dimensões (usualmente duas dimensões, facilitando a análise gráfica dos dados).

3.3 Clusterização K-means

O método de clusterização *k-means* (ou algoritmo de Lloyd's) 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 os melhores k pontos centrais, isto é, posicionar estes pontos 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 (de maneira a minimizar a distância média).

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

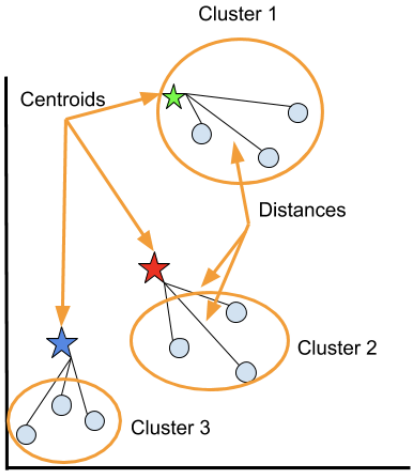

Figura 4 – Exemplo de clusterzacao k-means

3.3.1 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**.

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 5). Seleciona-se o número de clusters que corresponde ao "cotovelo" (elbow) do gráfico.

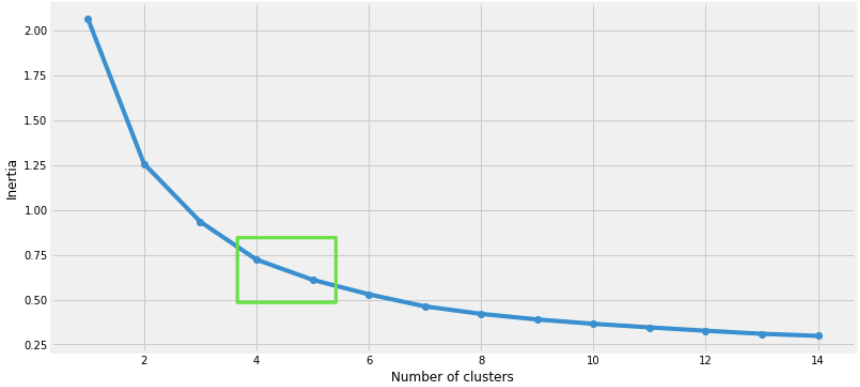

Figura 5 – Método de elbow

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, afim de medir o impacto da clusterização para os algoritmos testados (que serão listados em seções posteriores).

Todo o experimento foi desenvolvido em *Python* (principalmente ao seu vasto ferramental para manipulação de dados) e o código está disponível no GitHub [TODO REF GITHUB]. 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).

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 forum 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 questões (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 as limitações do hardware utilizado para o teste), o *dataframe* final gerado possuí 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 pré-fixo "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 6 demonstra a representação gráfica das palavras de acordo com a sua frequência.

Figura 6 – 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 baseado 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 pseudo-código 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, afim 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 items 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*.

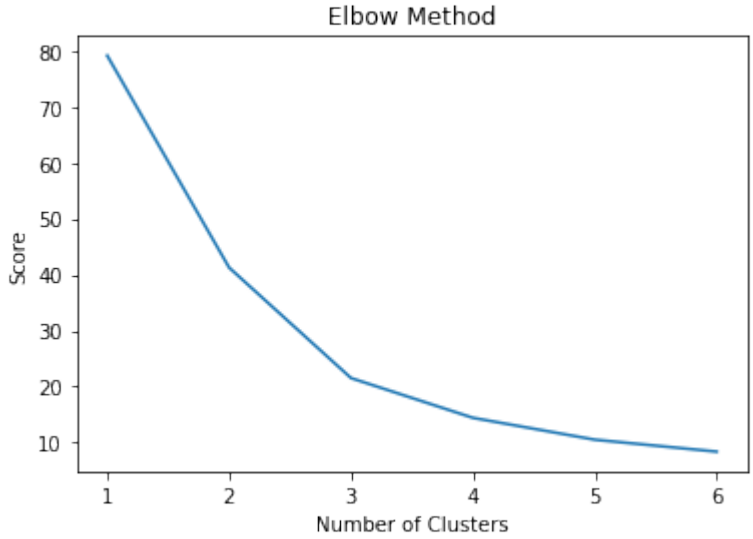

Figura 7 – Método de Elbow

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

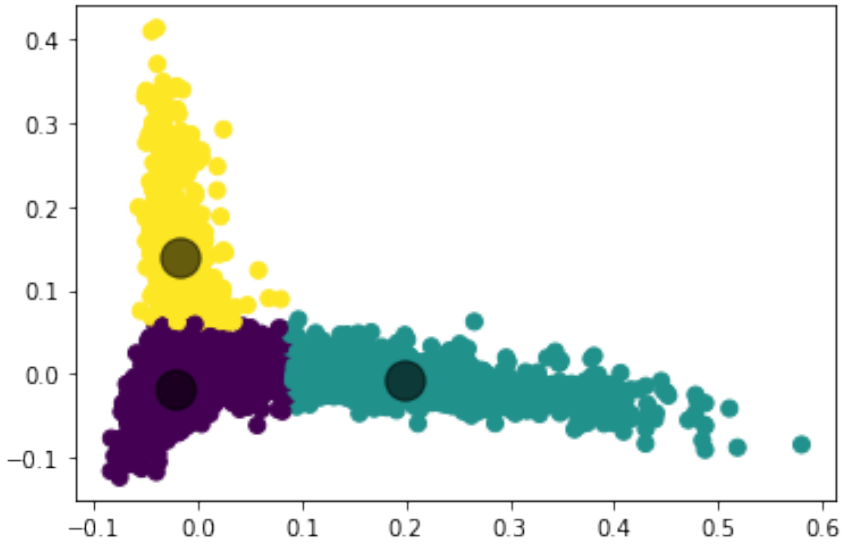

Figura 8 – Distribuição dos clusters (dados vetorizados)

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 uma comparação 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 a métrica impacto da clusterização na taxa de compressão, que é obtida pela diferença entre os resultados com e sem a clusterização.

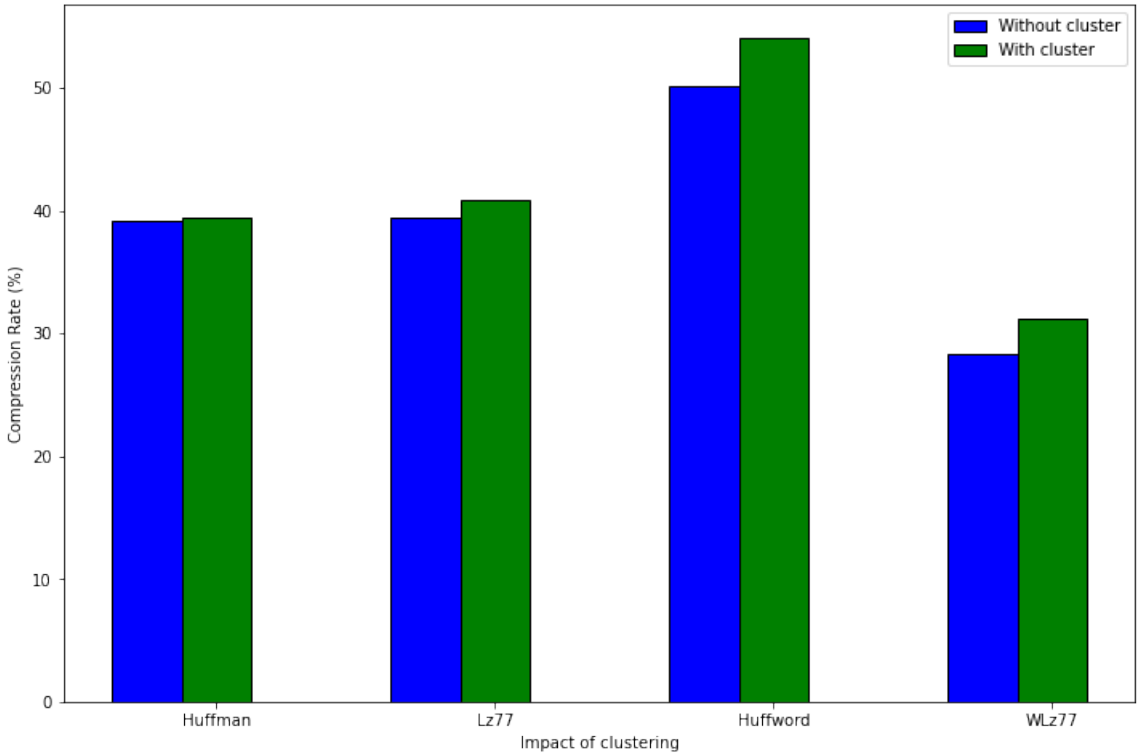

Figura 9 – Impacto da clusterização na taxa de compressão

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*, onde foi **alcançado 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 utilizada 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 palavra 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 a conclusão de que a clusterização impacta **positivamente** a **taxa de compressão** de textos. Entretanto, os experimentos também mostram um aumento nos recursos consumidos (tanto pelo pré-processamento, quanto pela compressão em si). Espera-se resultados apresentados neste trabalho sejam úteis na implementação de ferramentas de compressão otimizadas, principalmente para grande 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

HIRSCHBERG, D.S; LELEWER D.A; *Data compression,* Computing Surveys 19.3, 1987. Nenhuma citação no texto.

BLELLOCH G.E; *Introduction to Data Compression,* Carnegie Mellon, 2013 Nenhuma citação no texto.

BERTSEKAS D.P; TSITSIKLIS J.N; *Introduction to Probability* M.I.T, Lecture Notes Course 6.041-6.431, 2000 Nenhuma citação no texto.