

Tópicos em Busca e Aprendizado de Máquina com Textos

com exemplos em Python e KNIME

Geraldo Xexéo

5 de abril de 2022 06:49

Copyright © 2022 Geraldo Xexéo.

Todos os direitos reservados.

Esta é um pré-publicação com o texto parcial em trabalho.

Contato com o autor pelo e-mail xexeo@cos.ufrj.br.

DRAFT

Sumário

Lista de Figuras	vii
Lista de Tabelas	xi
Lista de Programas	xiii
I. Introdução	1
1. Introdução	3
1.1. Objetivo	4
2. Textos e Documentos	7
2.1. O que é um texto?	7
2.2. Conceito geral de documento	8
2.3. Conceito Específico de Documento	10
2.4. A Necessidade da Compreensão do Texto	10
2.5. Estrutura da Língua	11
2.6. A Escrita	14
2.7. Múltiplos Significados do Texto	15
Tanto Mar (1975)	17
2.8. O Descasamento entre a Expressão e a Interpretação	18
2.9. A Codificação da Escrita no Computador	20
2.10. Distribuição das Palavras	21
2.11. Modelos Conceituais para Documentos	22
2.11.1. FRBR	22
2.11.2. DoCO	24
2.11.3. Dublin Core	24
2.11.4. Descrições orientadas ao formato	25

2.12.	Coleções de Documentos	25
2.13.	Exercícios	26
3.	Codificação de Caracteres	29
3.1.	A representação de informação	29
3.2.	Da variedade para o ASCII	31
3.3.	<i>Code pages</i>	34
3.4.	Unicode	34
3.5.	UTF-8	35
3.6.	O fim de linha	35
3.7.	Desafios apresentados pela codificação	36
3.8.	Detectando uma <i>code page</i> automaticamente	38
3.8.1.	Lendo com codificações em Python	40
3.9.	Tratando a codificação no KNIME	41
3.10.	Codificando strings	45
3.11.	Codificações por referência	46
3.12.	Codificação de Binários em Texto	48
3.12.1.	Base64	48
3.13.	Exercícios	49
4.	Arquivos Textuais	51
4.1.	Os formatos mais comuns	51
4.2.	Formatos de Edição	52
4.3.	Linguagens de Marcação	52
4.3.1.	HTML	53
4.4.	HTML 5	55
4.4.1.	Limpando HTML de arquivos com Python	56
4.4.2.	Limpando HTML de arquivos com KNIME	57
4.5.	Outros Formatos Específicos	60
4.6.	Formatos de Impressão	60
4.6.1.	PostScript	60
4.6.2.	PDF	60
4.6.3.	Extraindo texto do PDF em Python	61
4.6.4.	Extraindo texto do PDF em KNIME	61
4.7.	XML	65
4.8.	Programas Exemplo	67
II.	Pré-Processamento E Algoritmos Básicos	69
5.	Pré-Processamento	71
5.1.	Noções de Morfologia da Palavra	72
5.1.1.	Lematização e Stemming	73
5.2.	Tokenização	73

5.3.	<i>Case Folding</i>	74
5.4.	Stemmers	75
5.4.1.	Stemmers por Busca em Tabela	76
5.4.2.	<i>Porter Stemmer</i>	77
5.4.3.	<i>Snowball Stemmer</i>	77
5.4.4.	<i>Lancaster Stemmer</i>	77
5.4.5.	<i>Stemmers</i> Criados para o Português	78
5.4.6.	Usando os <i>Stemmers</i> em Python	79
5.5.	Remoção de <i>Stopwords</i>	85
5.5.1.	Usando o KNIME	91
5.5.2.	Problemas com <i>stopwords</i>	91
5.6.	Normalização de texto	93
5.6.1.	Estratégias gerais de correção	94
5.7.	Uma Cadeia de Pré-Processamento em KNIME	95
5.8.	Exercício	96
6.	Estruturas de Dados Chave-Valor	99
6.1.	Lista Invertida	99
6.1.1.	Exemplo de Listas Invertidas	99
6.2.	Melhorias na lista invertida	101
6.2.1.	Listas Invertidas em Python	101
6.2.2.	Implementação trivial em Python	101
6.3.	Tabelas <i>Hash</i>	102
6.4.	Caches em Python	103
7.	Medidas de Avaliação	105
7.1.	Medidas Clássicas de Avaliação	105
7.1.1.	Problemas com a Acurácia	107
7.2.	F	107
7.3.	Outras Medidas	107
III.	Recuperação de Texto	109
8.	Busca e Recuperação	111
8.1.	Motivação	111
8.1.1.	Escalas do Problema de Busca	111
8.1.2.	Como Atender o Usuário da Busca	112
8.1.3.	Uso da Recuperação da Informação no Tempo	112
8.2.	Principais Problemas	113
8.3.	A Tarefa de Recuperação	115
8.4.	Aboutness	115
8.5.	Relevância	117
8.5.1.	Atributos da relevância	118

8.5.2.	Manifestações de Relevância	119
8.5.3.	O Espaço das Relevâncias	120
8.6.	Modelo Conceitual da Recuperação da Informação	122
8.7.	Palavras-Chave e <i>Full Text Indexing</i>	123
9.	Representação Booleana	125
9.1.	Modelo Booleano	126
9.2.	Usando a Lista Invertida	127
9.3.	Processo básico de construção das listas invertidas	127
9.4.	Consultas booleanas	128
9.5.	Simulando o Modelo Booleano	128
9.6.	Estratégias para busca em longas listas invertidas	129
9.7.	Exemplos com o Modelo Booleano	129
9.8.	Exercícios	136
10.	Representação Vetorial	137
10.1.	Definições do Modelo Vetorial	140
10.2.	Medindo o peso de um termo em um documento	141
10.3.	Medindo a Similaridade entre Documentos	143
10.3.1.	BM25	143
11.	Softwares para Full-Text Search	145
11.1.	Lucene	145
11.1.1.	PyLucene	145
11.2.	Solr	146
11.3.	Elastic Search	146
11.4.	Bancos de Dados com Capacidade de <i>Full-Text Search</i>	146
11.5.	Software Nativo em Python	146
12.	Modelo Probabilístico	147
13.	Modelos de Linguagem	149
14.	Indexação por Semântica Latente	151
14.1.	A Decomposição em Valores Singulares	152
14.2.	Exemplo de Uso da LSI em Python	153
IV.	Classificação	161
15.	Algoritmos Clássicos de Classificação	163
15.1.	Naïve-Bayes	163
15.2.	SVM	163
16.	Redução de Dimensionalidade	165

17.Embeddings	167
Apêndices	177
A. Tópicos em Python	179
A.1. Dicionários	179
A.2. Strings	181
A.2.1. Índices de Strings	182
A.2.2. Fatias de Strings	182
A.2.3. O operador in	183
A.2.4. Comparando Strings	184
A.2.5. Funções com String	184
A.2.6. Exercícios	184
B. Tópicos Selecionados de Matemática	187
B.1. Autovetores e Autovalores	187

DRAFT

Lista de Figuras

2.1.	Primeira forma de interpretar sintaticamente a sentença “O menino viu a mulher de binóculo”, onde o menino tem o binóculo, baseado em (Pagani, 2022)	13
2.2.	Segunda forma de interpretar sintaticamente a sentença “O menino viu a mulher de binóculo”, onde a mulher tem o binóculo, baseado em (Pagani, 2022)	14
2.3.	Letra de 1975 da música Tanto Mar, de Chico Buarque.	17
2.4.	Descasamento expressão interpretação	19
2.5.	Registro fonético da frase <i>I ate an apple in August</i> segundo o <i>International Phonetic Alphabet</i> , para o inglês americano	20
2.6.	Distribuição das 10000 primeiras palavras no corpus Gutenberg, fornecido com o NLTK (Bird, Loper e Klein, 2009).	22
2.7.	Modelo de Entidades e Relacionamentos para o FRBR Group 1, baseado em (IFLA, 2009)	23
2.8.	Arquitetura do DoCO, obtida sob licença Creative Commons Atribuição 3.0, (Shotton e Peroni, 2015)	24
2.9.	Canal de Ruído de Shannon	28
3.1.	Um <i>Prolog</i> XML indicando a codificação	38
3.2.	Configurando a codificação default do KNIME.	42
3.3.	Um workflow KNIME só com um módulo, FileReader.	42
3.4.	Lendo um arquivo texto como uma sequência de linhas no KNIME. Módulo File Reader.	43
3.5.	Configurando a codificação correta do arquivo.	43
3.6.	Configurando a codificação errada, a visualização apresenta caracteres estranhos.	44
4.1.	A visualização de um arquivo HTML muito simples.	54

4.2.	Imagem parcial do HTML usado em uma página pessoal	55
4.3.	Linha do tempo do HTML ao HTML 5.	56
4.4.	Visão geral da plataforma Web, segundo WHATWG (2021)	56
4.5.	Workflow para ler, via URL, e limpar um arquivo HTML	57
4.6.	Recuperando uma página via sua URL.	58
4.7.	Configurando a remoção de tags de markup	58
4.8.	Resultado recuperado pelo nó de Leitura.	59
4.9.	Resultado pós remoção do markup.	59
4.10.	Workflow para ler e transformar um arquivo PDF em um Document do KNIME	62
4.11.	Configuração do nós PDF Parser	62
4.12.	Configuração do nó Document Viewer	63
4.13.	Vendo a lista de documentos lidos	63
4.14.	Vendo o detalhe de um documento, principalmente seu conteúdo textual.	64
4.15.	Exemplo de fragmento de arquivo xml.	65
5.1.	Classificação algoritmos de fusão, a partir de Frakes e Baeza-Yates (1992), Galvez, MoyaAnegón e Solana (2005) e Amri e Zenkouar (2018). O caminho em amarelo indica os algoritmos de Porter e o RSLP.	76
5.2.	Tempo de execução do stemmer de Porter com diferentes tamanhos de cache, potências de 2 de 0 a 524288 entradas, mostrando um limite prático para essa coleção, onde há 66392 formas únicas. Um cache de 32768 foi suficiente para atingir um desempenho próximo do máximo.	85
5.3.	Contagem das 50 palavras mais frequentes do corpus Gutenberg, pelo programa Programa 5.6	86
5.4.	Aba Preprocessing do nó Stop Word Filter no KNIME	91
5.5.	Aba Filter Options do nó Stop Word Filter no KNIME	92
5.6.	Cadeia de préprocessamento em KNIME	96
8.1.	Modelo estratificado da relevância, baseado em Saracevic (2017b).	120
8.2.	Tipos de relevância, baseado em Saracevic (2017b).	121
8.3.	Modelo Conceitual da Recuperação da Informação, segundo (Fuhr, 1992).	123
10.1.	A ideia básica do Modelo Vetorial é que os termos do documento, e da consulta, indicam seu significado, e que a similaridade entre a consulta e os documentos da coleção pode ser calculada em um espaço vetorial.	138
10.2.	Exemplo de problema com o uso do tamanho dos vetores	139
10.3.	Usando o cosseno dos vetores.	140
14.1.	Visão abstrata do LSI.	152
14.2.	Esquema do SVD	152
14.3.	Esquema do SVD após a redução de <i>rank</i>	153
14.4.	Vetores reduzidos do exemplo.	157

- A.1. As posições de indexação das strings estão na verdade entre as letras 183

DRAFT

DRAFT

Lista de Tabelas

3.1. Tabela ASCII, ou Unicode com a representação UTF-8	33
3.2. Alguns caracteres Unicode e suas representações UTF-8	35
3.3. Exemplo de como é feita uma codificação Base64	49
5.1. Lista de <i>stopwords</i> do sistema <i>Smart</i> (Salton e McGill, 1983a).	88
5.2. Lista de <i>stopwords</i> em português brasileiro obtida em http://www.ranks.nl/stopwords/	89
6.1. Representação de um lista invertida.	100
6.2. Exemplo de inserções em uma tabela de endereçamento direto com endereçamento aberto. Palavras que colidem estão em negrito.	103
7.1. Matriz de Confusão	106
7.2. Matriz de Confusão Desbalanceada	107
9.1. Matriz Termo-Documento, ou Matriz de Incidência, simples.	128
10.1. Exemplo de criação passo a passo dos vetores de documentos de uma coleção.	141
A.1. Alguns formas de usar dicionários em Python	180

DRAFT

Lista de Programas

3.1.	Exemplo de leitura com codificação	36
3.2.	Exemplo simples de uso do chardet	38
3.3.	Exemplo de programa que usa o chardet	39
3.4.	Uso do charset-normalizer.	39
3.5.	Uso do charset-normalizer com normalização automática.	39
3.6.	Comandos para abrir um arquivo escolhendo a codificação	40
3.7.	Exemplo de programa que usa o chardet e lê o arquivo	40
3.8.	Obtendo a sequência correta da codificação de uma string	45
3.9.	Salvando um mesmo arquivo em várias codificações.	45
3.10.	Processando entidades HTML	47
4.1.	Programa simples para obter todo o texto dentro de um arquivo HTML .	56
4.2.	Usando o pdfminer.six para extrair texto de um arquivo PDF	61
4.3.	Usando uma forma mais complexa de extrair texto de um arquivo PDF .	61
4.4.	Exemplo de arquivo XML usando um DFD.	66
4.5.	Exemplo de arquivo DTD.	66
4.6.	Exemplo de arquivo XSD.	66
4.7.	Funções de apoio, arquivo birtutis	67
4.8.	Lendo arquivos do corpus Folha, resulta em um dicionário com id dos artigos e indicando o texto sem tratamento.	67
5.1.	Tokenização com o NLTK	74
5.2.	Tokenização com o Gensim	74
5.3.	Uso dos stemmers para inglês no NLTK	79
5.4.	Uso dos stemmers para português no NLTK	81
5.5.	Usando stemmer com cache em Python	83
5.6.	Contagem das 50 primeiras palavras do corpus Gutenberg no NLTK . . .	85
5.7.	Removendo as <i>stopwords</i> com NLTK	87

5.8. Programa que usa <i>go-words</i> compostas	92
5.9. Corretor simples segundo Norvig	95
6.1. Criação simples de uma lista invertida	101
9.1. Programa simples para fazer consultas do tipo E e OU a uma lista-invertida	128
9.2. Preparação do Exemplo.	130
9.3. Cálculo da uma matriz termo documento.	130
9.4. Leitura de um documento palavra a palavra e registro dos documentos que possuem os termos índices em uma “lista invertida” construída por meio de um dicionário.	131
9.5. Calcula operadores E, OU e NÃO para duas listas.	132
9.6. Transforma um expressão lógica infix a pós-fixa.	133
9.7. Processa uma busca booleana na coleção em memória usando o documento original.	134
9.8. Processa uma busca booleana na coleção em memória usando índice. . .	135
14.1. Declarações	153
A.1. Obtendo um caractere indexado em uma string	182
A.2. Obtendo um caractere indexado negativo em uma string	182
A.3. Fatiando uma string	182
A.4. Fatias que pulam letras.	183
A.5. Marcando só a posição final.	183
A.6. Cada letra equivale a um número.	184

Lista de Acrônimos

ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
CSS	Cascade Style Sheets
DCMI	Dublin Core Metadata Initiative
DoCO	Document Components Ontology
EBCDIC	Extended Binary Coded Interchange Code
FRBR	Functional Requirements for Bibliographic Records
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standard Organizations
MARC	Machine Readable Cataloging Record
RFC	Request For Comments
SGML	Standard Generalized Markup Language
UTF	U T F
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	eXtensible Markup Language

DRAFT

Parte I.

Introdução

DRAFT

Introdução

Este livro foi composto a partir das notas de aula da cadeira de **Busca e Mineração de Texto**, parte do curso de mestrado e doutorado do Programa de Engenharia de Sistemas e Computação da COPPE, o Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa em Engenharia, da Universidade Federal do Rio de Janeiro.

Sua origem é um curso de Busca e Recuperação da Informação, que foi dado por mais de 10 anos, que também olhava outras áreas, como busca de imagens. Porém, com o desenvolvimento crescente em todas áreas, passou a ter o foco em texto, mas não só em busca, já que o interesse da comunidade se abriu para várias outras questões. Entre elas, o autor, e seu orientados, se envolveram principalmente com análise de sentimentos e plágio.

Tendo em vista essa origem, o livro é escrito para quem tem uma boa formação em Computação, em especial em programação, sendo Python a linguagem de escolha, e com conhecimento básico de tópicos como HTML, XML, internet, etc.

O assunto desse livro é uma interseção entre várias áreas de pesquisa, incluindo, mas não limitada a:

- Processamento de Linguagem Natural;
- Linguística e Linguística Computacional;
- Recuperação da Informação;
- Ciência da Informação;
- Biblioteconomia;
- Aprendizado de Máquina;
- Inteligência Artificial e Computacional;
- Representação do Conhecimento;
- Estatística, e
- Processamento de Texto.

Uma maneira de entender o processamento de texto, ou o processamento de linguagem natural, é por meio das tarefas incluídas nessa área. Em especial, este livro trata das seguintes tarefas:

- Busca *ad-hoc*, isto é, buscar um conjunto de documentos que responde a uma consulta, a tarefa básica do Google; e
- Classificação de documentos, isto é, classificar um conjunto de documentos de acordo com um conjunto de rótulos pré-estabelecidos, por meio do aprendizado de máquina (em construção), e
- Detecção de Plágio, uma tarefa complexa que busca candidatos a plágio em uma primeira etapa e a detecção específica do plágio em uma segunda etapa (em construção).

Outras tarefas possíveis ainda não tratadas diretamente neste livro, mas que podem utilizar as mesmas representações e algoritmos aqui tratados:

- Filtragem, a escolha de documentos ao longo do tempo para uma certa consulta fixa;
- Roteamento, filtragem com ranking;
- Recomendação, seleção de documentos para um usuário a partir de informações sobre o uso do sistema por outros usuários;
- Agrupamento, a organização de um conjunto de documentos em grupos por similaridade;
- Extração da Informação, a extração de informações específicas ou genéricas de um documento ou de um conjunto de documentos, e
- o Reconhecimento de Entidades Nomeadas.

Todas essas tarefas dependem de outros assuntos, que são citados e tratados no livro em diferentes profundidades, como:

- Álgebra Linear e Álgebra Linear Computacional;
- Estruturas de Dados em Memória;
- Estruturas de Dados em Disco;
- Bancos de Dados Relacionais e Não Relacionais;
- Algoritmos de Aprendizado de Máquina;
- Redes Neurais e Redes Neurais Profundas;
- Programação (em Python), e
- Desenvolvimento No-Code e Low-Code.

1.1. Objetivo

O principal motivo deste livro é habilitar o aluno para iniciar uma dissertação de mestrado e doutorado que trate ou utilize de processamento de texto para algumas aplicações:

- Busca e Recuperação da Informação, em especial na tarefa de busca *ad-hoc*;

- Mineração de Texto, em especial nas tarefas de classificação, inclusive análise de sentimento, e
- Identificação de Plágio.

Os objetivos de ensino-aprendizado são capacitar os leitores a:

- conhecer o que é texto e o que é texto em sua forma digital e as formas como é encontrado;
- conhecer as tarefas de busca e mineração de texto;
- entender a dificuldade de cada tarefa;
- conhecer e aplicar os algoritmos básicas para as tarefas principais;
- conhecer e aplicar ferramentas de software típicas para as tarefas principais;
- realizar experimento na área, e
- conhecer e aplicar as medidas de avaliação para as tarefas.

Textos e Documentos

Este capítulo trata do que é um texto e um documento. As duas primeiras seções tratam de perguntas que parecem ter uma resposta simples, mas na verdade geraram discussões e redefinições ao longo do tempo. “O que é um texto?”, perguntam os linguistas, principalmente, preocupados com sua capacidade de analisá-los e estudá-los. “O que é um documento”, por sua vez, é uma pergunta clássica na Ciência da Informação, discutida desde o século XIX.

2.1. O que é um texto?

Um **texto** pode ser definido de forma muito objetiva, e com uma visão computacional, como uma sequência estruturada de símbolos convencionados que registra uma informação, de acordo com uma língua, por meio da escrita. Um texto seria, então, basicamente o que é conhecido, nas linguagens de programação, como uma *string*, mas não qualquer *string*: uma *string* que registra uma informação em uma língua, usando para isso uma forma escrita da mesma.

Para a Linguística, texto se refere a qualquer passagem, falada ou escrita, de qualquer tamanho, que forma um todo unificado (Haliday e Hasan, 1976, pg. 1). Espera-se que um texto tenha um significado. Este livro não trata de registros sonoros de enunciados falados em uma língua, tais registros precisam estar transcritos para serem tratadas pelas técnicas aqui descritas.

Essas duas definições bastam para o trabalho apresentado neste livro, porém é importante citar que, para outros trabalhos, como a análise crítica, definições mais complexas podem ser usadas. Haliday e Hasan (1976, pg. 23), ainda no mesmo capítulo da última definição, redefinem texto como “uma passagem de discurso que é coerente em duas visões: é coerente em respeito ao contexto da situação, e portanto consistente no registro, e é coerente em respeito a si mesma, e portanto coesiva.”. Como consistência no registro deve ser entendido uma continuidade de significado em relação a situação. Essa definição

exige bem mais do texto: que ele tenha um significado e que o significado faça sentido no contexto. Atkins, Clear e Ostler (1992) mostram que uma definição similar, “uma série de sentenças e parágrafos coerentes” falha com uma quantidade de “unidades de linguagem” sobre as quais é preciso trabalhar: pequenos anúncios, artigos de jornais, poemas. Nessa lista é possível adicionar mensagens em redes sociais, por exemplo.

É importante chamar atenção que há alguma expectativa, em todo o trabalho com um texto, que ele seja coerente de alguma forma, porém a importância da coerência, e consequentemente do significado, varia de acordo com a técnica usada e com a expectativa do usuário.

2.2. Conceito geral de documento

A questão de definir o que é um **documento** parece ser ainda mais complicada que a anterior. Principalmente após a invenção da impressão por tipos móveis por Gutemberg, depois com a sistematização da pesquisa científica, e mais atualmente após a internet, os documentos se tornaram cada vez mais importantes na sociedade, servindo de base de pesquisas a campanhas políticas, o que pode ser percebido no uso de documentos, e análises variadas de seu conteúdo, como suporte a opiniões abalizadas e *fake news*. Organizá-los se tornou uma profissão, que se transformou em uma área de pesquisa (Briet, 1951; Buckland, 1997; Document Academy, 2022).

O que era inicialmente chamado de “Documentação”, como prática ou Ciência, se transformou na Ciência da Informação em torno de 1950, e se preocupa, entre outras coisas, com permitir o acesso sistemático a documentos, seu registro e sua descrição. A questão seminal levantada por esses pesquisadores foi a mesma que se faz nessa seção: o que é um documento? Uma explicação frequente foi: “Qualquer expressão do pensamento humano”, já outra forma de pensar é “Qualquer coisa a qual podemos aplicar as técnicas de documentação” (Buckland, 1997).

Originalmente, documentos eram vistos apenas nos registros textuais, como cartas, contratos e livros, porém, com a evolução da Ciência da Informação, e de outras formas de mídia, como fotografias e filmes, houve uma ampliação do conceito. Já no século XIX, pioneiros como Paul Otlet, um dos precursores do hipertexto em rede, discutiam documentos de forma funcional, o que levaria a incluir esculturas, objetos de museu, e até mesmo animais, sendo que documento poderia ser visto como uma “evidência física organizada” (Buckland, 1997).

Em 1937, o Instituto Internacional para Cooperação Intelectual definiu tecnicamente um documento como “Qualquer fonte de informação, em forma material, capaz de ser usada para referência, ou estudo, ou como uma autoridade. Exemplos: manuscritos, material impresso, ilustrações, diagramas, espécimes de museu, etc.” (Buckland, 1997). Essa abordagem, mais generalista, dominou por algum tempo o pensamento dos “documentalistas”, que hoje são “cientistas da informação” (Buckland, 1997).

Briet (1951) propôs, por meio de exemplos, o que seria um documento. Segundo ela “Uma estrela é um documento? Um seixo rolado por um rio é um documento? Um animal vivo é um documento? Não. Mas são documentos as fotografias e os catálogos de estrela, as pedras de um museu de mineralogia, os animais catalogados e expostos em um zoológico” (Briet, 1951). Em seu texto, seminal na área, ela explica que a própria criação dos documentos forçou a tarefa de organizá-los, e com isso aparecem os “documentalistas”.

Meyriat (1981) fala de **documentos por intenção**, objetos como livros e jornais, que são criados como documentos, e **documentos por atribuição**, como o antílope exposto de Brie. Sua definição, “O documento pode ser definido como um objeto que suporta a informação, que serve para comunicar e que é durável (a comunicação pode, assim, ser repetida).” (Meyriat, 1981, 2016), parece bem adequada. Ainda, segundo Meyriat (2016), “Cada mensagem tem um significado e não se pode definir um documento independentemente do significado da mensagem que ele tem a função de transmitir”.

Como todo objeto pode ser encarregado de suportar informação, a noção de documento é muito mais ampla do que a de escrito (ou texto), e, do mesmo modo, se pode escrever em objetos diferentes, como estátuas (Meyriat, 1981). Isso dá um protagonismo ao receptor da mensagem: é ele que vê a informação no objeto, ao interpretá-lo. Ainda Meyriat (1981) dá o exemplo do jornal usado para enrolar um peixe, que perde, pelo menos temporariamente, sua função de documento, mas que pode recuperá-la mais tarde se alguém o usa para ler uma notícia. Mais ainda, o “documento sempre pode receber novas questões, com a esperança de se obter informações novas em resposta” (Meyriat, 2016). Por exemplo, um texto que indique o horário de saída de um trem antes da invenção dos automóveis teve uma utilidade fugaz, no dia em que o trem partiu, e depois não teve utilidade nenhuma por anos. Muito mais tarde, pôde se tornar útil, quando perguntas foram feitas sobre o impacto da introdução dos automóveis no uso de trens. Finalmente, Meyriat (2016) conclui que “O objeto próprio da abordagem documentária, ..., é, portanto, a informação”.

Isso leva a concluir que o importante no documento não é o registro físico, como um livro, mas sim a informação associada a eles, e que essa informação depende do desejo e da interpretação tanto do emissor quanto do receptor, sendo transformada pelo tempo, pelo espaço e pelos agentes que agem sobre ela (Meyriat, 1981).

Se tudo que contém informação, ou que pode ser visto como um suporte de informação pode ser um documento, o que diferencia os documentos? A Document Academy (2022) apresenta alguns aspectos que fazem algo ser um documento:

- **Indexalidade**, pois são representações que apontam para ou discutem outras coisas, conceito também chamado de *aboutness*;
- **Pluralidade**, possuindo vários aspectos, como mental, material e social;
- **Estabilidade** (*Fixity*), em sua forma física, sendo um *móvel imutável*, porém a Web trouxe também um grau de fluidez;

- **Documentalidade**, a capacidade de produzir, suportar, permitir, encorajar, permitir, influenciar, tornar possível, proibir... em seus arranjos com outras coisas, e
- **Produtividade**, que permite a construção de outros documentos a partir dele (Document Academy, 2022).

Documentos, porém, como chama a atenção Briet (1951), não são apenas textos, mas textos tratados de alguma forma, de maneira que possam ser organizados e catalogados. Isso é feito por meio de dados sobre o documento, ou dados sobre os dados, o que é chamado de metadado. Uma mensagem de micromídia social, um livro, notícias de jornal, são documentos não só porque contém texto, mas também porque é possível identificá-los de várias formas, como autor, fonte, data e hora de produção, etc. Textos, porém, ao contrário de outros objetos, já contém dados que permitem identificá-los: o próprio texto.

2.3. Conceito Específico de Documento

Especificamente neste livro, o documento é a unidade com que o texto é trabalhado de forma que seu significado é considerado coerente o suficiente para poder ser o resultado de uma busca ou um objeto a ser classificado. Então, dependendo do sistema, um documento pode ser um livro, um capítulo, um parágrafo, uma linha, um relatório, uma mandado judicial, um tweet, a transcrição de uma fala, etc.

Nas tarefas que esse livro trata especificamente, é normalmente importante usar um conjunto de documentos, que é chamado de *corpus*¹ ou *dataset*, o último sendo um termo típico do aprendizado de máquina. Esse conjunto de documentos pode ser usado como base de dados de um sistema de recuperação de informação, na prática servindo de ruído no processo, atrapalhando a busca², de informação usada para uma tarefa de aprendizado de máquina, etc.

2.4. A Necessidade da Compreensão do Texto

A partir do que foi visto anteriormente, é possível afirmar que todas as tarefas descritas neste livro processam texto de alguma forma, sendo esses textos normalmente organizados em um corpus, que não é nada mais que um conjunto de documentos que passou com uma curadoria que tinha um objetivo.

Para esse processamento, o que se busca é alguma representação, possivelmente para comparar representações ou extrair informações, do significado, ou dos múltiplos significados, de um ou mais textos, ou de um texto e uma consulta, ou requisito de informação.

¹Cujo plural é corpora

²A tarefa de busca é muito fácil em uma base com um só documento

Por exemplo: na busca o objetivo é encontrar um texto que seja relevante a uma necessidade de informação do usuário, na classificação os textos são rotulados como pertencentes a alguma classe, no agrupamento os textos são organizados em grupos por similaridade, na identificação de entidades nomeadas se busca extrair nomes de pessoas, lugares, ou datas que estão em um texto, etc.

Supostamente, todas essas operações devem obter resultados relativos aos vários significados dos documentos, provavelmente dentro de um contexto ou de uma tarefa. Na prática, são usados algoritmos, alguns simples, outros extremamente complexos, sobre representações e descrições dos textos.

Todas os problemas que serão tratados computacionalmente neste livro implicariam que, como seres humanos, tivéssemos a capacidade de processar imensas quantidades de texto com a mesma facilidade que podemos processar, enquanto lemos e pensamos sobre eles, uma sentença ou um livro, e na compreensão de texto em uma, ou mais, línguas. É importante chamar a atenção que mesmo essa solução não seria perfeita, porque há problemas de compreensão de texto até para humanos, ligados não só ao poder cognitivo, mas também ao contexto específico e geral em que o texto é processado. Com o computador, apesar de termos um grande aumento de velocidade, os problemas de buscar um significado se acumulam.

Essa dificuldade já se iniciam na própria língua nativa de uma pessoa, sem falar das outras línguas. Por exemplo, a sentença “A manga é amarela” não pode ser compreendida totalmente se não há uma contextualização: ela trata de uma manga como parte de peças de vestuário ou uma fruta?

2.5. Estrutura da Língua

A abordagem de “entender o texto” para algumas das tarefas típicas que fazemos com um computador é terrivelmente complicada. Hoje em dia, muito desse trabalho é feito sem que o texto seja realmente compreendido, no sentido em que não é criada uma explicação explícita do significado do texto de forma compreensível por um ser humano, mas por meio de representações que possuem mais ou menos significado, ou são apenas abstrações matemáticas.

Algumas dessas representações levam em conta que grande parte do significado do texto está nas palavras ou sequência de palavras, conhecidas como **n-gramas**, mas como veremos mais adiante neste capítulo, isto pode não ser verdade, por causa do contexto e das interpretações que vão além do que está escrito de forma literal e mesmo além da intenção do autor.

Mesmo traduções, cuja abordagem inicial dos pesquisadores era criar um modelo do significado da sentença em uma língua e reescrever esse significado em outra língua, hoje são feitas sem o conhecimento real das línguas, ou seja, sem conhecimento linguístico e do significado das sentenças, mas apenas por aprendizado de máquina e estatística.

Essa complexidade pode ser melhor analisada quando é conhecido a forma como o estudo de uma língua, ou das linguagens em geral, é feito em uma estrutura em camadas:

- o **fonético**, o estudo dos sons da língua;
- o **morfológico**, que estuda partes significativas das palavras;
- o **léxico**, que trata das palavras como um todo;
- o **gramatical** ou **sintático**, que trata das estruturas das sentenças;
- o **semântico**, que trata do significado dessas estruturas;
- o **pragmático**, como a língua é usada para alcançar objetivos, e
- e o de **discurso**, que estuda unidades de texto.

Todos esses níveis podem possuir ambiguidade, tanto na língua falada quanto na língua escrita. Por exemplo, em chinês é necessário descobrir as fronteiras entre as palavras, pois a escrita não possui espaço, já em alemão, mesmo possuindo espaços, algumas palavras precisam ser divididas em suas partes. Outro exemplo é a necessidade de corrigir palavras escritas de forma errada, por exemplo em uma mensagem de mídia social, usando para isso o conhecimento da similaridade dos sons representados pela escrita. Assim, podemos corrigir a palavra “minino” para “menino”, por exemplo.

O problema de compreender o texto, para um computador e para um humano, é resolver essas e outras ambiguidades.

A verdade é que isso não só é difícil, como pode ser impossível até para um humano, principalmente se não há um contexto ao redor. Além disso, as técnicas que usamos, por vezes, aumentam essa ambiguidade. Uma representação comum para indexação de texto, conhecida como Bag of Words (BoW) considera apenas a presença da palavra no documento, não sua ordem. Assim “Brasil vence Alemanha” tem a mesma representação que “Alemanha vence Brasil”, enquanto seu significado é totalmente diferente.

A questão do significado de “manga”, citada anteriormente, inclui um problema semântico sobre um objeto léxico, mas mesmo que saibamos todos os significados exatos das palavras, podemos ainda ter um problema de entender a sentença. Por exemplo, na sentença “O menino viu a mulher de binóculos”, quem estava com o binóculos? Um computador pode entender todos os léxicos, supondo que cada um tenha um significado único, ou escolhendo o significado mais comum, mas criaria, no mínimo, duas estruturas gramaticais distintas (sintaxe), cada uma com um significado (semântica) para o seu todo. Essas representações podem indicar que o menino usava o binóculo para ver a mulher, como na Figura 2.1, ou que o menino viu uma mulher que tinha um binóculo, como na Figura 2.2

Deve ficar claro que, fazendo a análise semântica, essas duas interpretações sintáticas poderiam gerar representações como:

```
----- Interpretação 1 -----  
existe(menino)  
existe(mulher)  
existe(vestido)
```

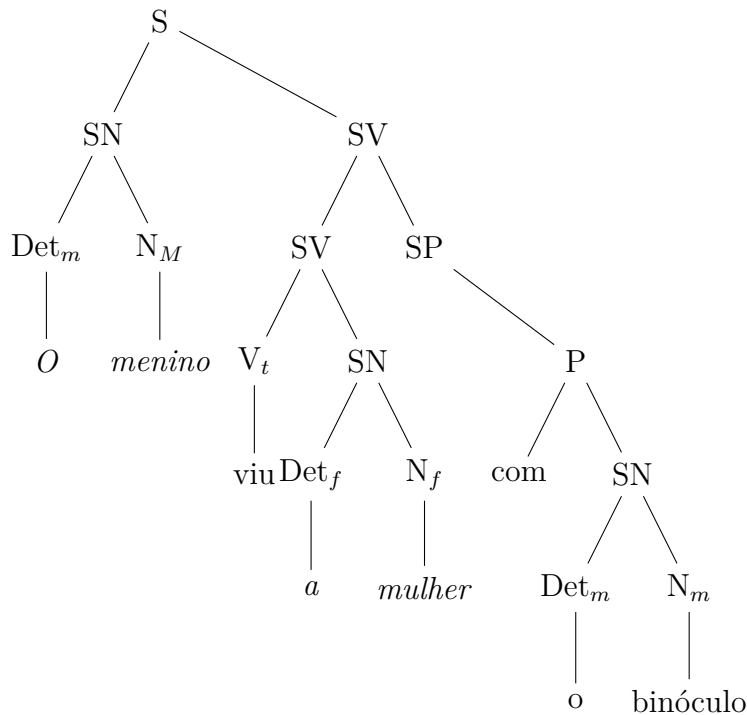


Figura 2.1.: Primeira forma de interpretar sintaticamente a sentença “O menino viu a mulher de binóculo”, onde o menino tem o binóculo, baseado em (Pagani, 2022)

```

existe(binóculo)
possui(menino,binóculo)
acao(menino,ver,mulher)
acao(menino,usa,binóculo)

```

----- Interpretação 2 -----

```

existe(menino)
existe(mulher)
existe(vestido)
existe(binóculo)
acao(menino,ver,mulher)
acao(mulher,carrega,binóculo)

```

Um exemplo de pragmática interessante é a prática de “pedir oferecendo”. Nessa caso, se um convidado fala para o dono de uma casa “Você não gostaria de um café?”, pode ser que seu desejo seja que o outro faça um café para ele. Toda a interpretação da frase leva a um leitor desavisado a interpretar que o emissor está oferecendo o café, porém no contexto específico, por exemplo, saber que a pessoa que oferece não sabe fazer café, ou não seria adequado fazê-lo já que é um visitante, ao nível da pragmática, levaria a deduzir que é uma frase que tem como objetivo que o receptor faça o café, possivelmente para ambos.

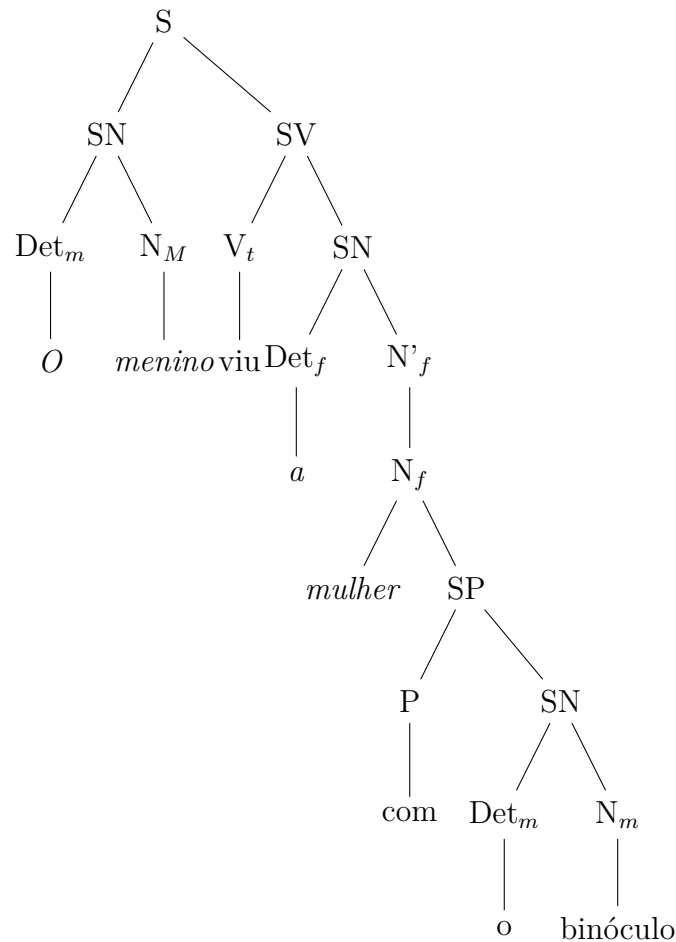


Figura 2.2.: Segunda forma de interpretar sintaticamente a sentença “O menino viu a mulher de binóculo”, onde a mulher tem o binóculo, baseado em (Pagani, 2022)

Assim, antes de ser complicada quando escrita, a língua é complicada *per si*. Neste livro, várias complicações do nível fonético, que exigem bom poder computacional, como a da identificação dos fonemas e separação das palavras faladas, são evitadas, já que tratamos apenas do texto escrito.

2.6. A Escrita

Segundo Rogers (2005), a escrita é “o uso de marcas gráficas para representar enunciados linguísticos específicos”. Ainda o mesmo autor diz que “A língua é um sistema complexo que reside em nosso cérebro e permite produzir e interpretar enunciados”(Rogers, 2005). Logo, o escrita, e por consequência o texto, não é a língua, mas uma forma de representar mensagens na língua.

Chomsky (2002) disse que uma língua natural, como o português ou o inglês³ é uma linguagem no sentido de ser “um conjunto (finito ou infinito) de sentenças, cada uma finita em comprimento e construída a partir de um conjunto finito de elementos”. Como disse Rogers (2005), uma pessoa, para se comunicar, possui, de forma intuitiva, as regras de formação dessa linguagem, ou seja, de sua gramática(Chomsky, 2002). A escrita torna esse enunciado, na forma de um texto, visível (ou invés de audível) e permite que seja registrado de certas maneiras(Rogers, 2005).

Assim, a escrita representa a língua, e apenas a língua, como forma de comunicação(Rogers, 2005). Na escrita ocidental, esses símbolos são letras que se compõe em palavras, que formam sentenças, que formam documentos. Estágios intermediários, como estruturas gramaticais da sentença e organização dos documento em parágrafos, seções e capítulos também são possíveis. Para isso são necessários sistemas de escrita.

Um **sistema de escrita** é uma forma de comunicação por símbolos, que registram uma língua falada por caracteres ou grafemas. Para entender melhor o que é um sistema de escrita, é possível adotar uma classificação proposta por Daniels (2003):

- **logográficos**, onde os grafemas são ideogramas ou pictogramas que denotam **morfemas**, as unidades indivisíveis da língua, como o chinês;
- **silábicos**, onde os grafemas representam sílabas, como o Katakana para japonês;
- **abugidas**, onde os grafemas representam consoantes com uma vogal inerente e variações são indicadas por diacríticos, como o Hebraico;
- **abjads**, onde os grafemas são consoantes e as vogais são representadas por diacríticos, como o Devanagari,;
- **alfabéticos**, onde os grafemas representam vogais, como os alfabetos Latino, Cirílico e Grego;
- **trácicos (*featural*)**, onde cada parte de um símbolo representa um traço fonético, como o Coreano.
- **sistemas mistos**.

2.7. Múltiplos Significados do Texto

Um texto pode ter muitos significados, pois sua compreensão depende do autor, do leitor, de seus contextos e de um contexto compartilhado por ambos. Já vimos que isso acontece por meio de interpretações diferenciadas de sentenças, devido a questões lexicais, gramaticais, etc. Porém podem existir mais significados, tanto para questões específicas, como o significado de uma palavra, como para o texto em geral.

Ditados populares, por exemplo, possuem um sentido denotativo, literal, e um conotativo, não literal. Quando falamos “Água mole em pedra dura, tanto bate até que fura”,

³Em inglês, uma *natural language*, mas acredito que a melhor tradução é língua natural, pois em português língua e linguagem são conceitos diferentes que são traduzidos para a mesma palavra em inglês: language.

estamos falando que a água desgasta as pedras, porém o que queremos dizer é que a persistência faz com que alcancemos o que parece impossível (Louro, 2007). Usado em um contexto específico, porém, o significado conotativo pode ser mais específico, como “pediu tanto que conseguiu”. Além disso, na geologia, a sentença pode ser tomada pelo que diz literalmente.

O autor e o leitor podem ser influenciados por várias estruturas psico-cognitivas, como sua intenção, sua capacidade de tradução dessa intenção em texto, sua capacidade de interpretação, seu conhecimento da língua, etc. O autor, por exemplo, pode querer passar sua mensagem de uma forma não literal, por meio de uma linguagem figurada. E o leitor pode, também, procurar interpretar outras coisas além do texto, como o estado mental do autor.

Ainda quanto ao leitor, sua interpretação vai levar em conta a tarefa que está realizando ao ler o texto. Por exemplo, um mesmo texto jornalístico sobre o governo pode ser entendido por diferentes leitores como uma descrição de fatos, um libelo de oposição, uma provocação aos governantes, ou, mais tarde na história, apenas um exemplo de palavras usadas em uma época específica.

Os significados se multiplicam entre o autor e o leitor porque pode existir uma linguagem figurada, porque a leitura pode ser feita com outra intenção, porque o leitor não entende a ironia ou outra figura de linguagem, porque referências temporais se perderam, etc.

A letra da música Tanto Mar de Chico Buarque de Holanda (Figura 2.3), em sua primeira versão, de 1975, oferece uma oportunidade de interpretação. Seu texto aparente fala sobre uma festa que ocorreu quando o Eu-lírico estava ausente e doente em outro lugar.

Um leitor atual sem o conhecimento histórico, ou ingênuo na época, pode imaginar que Chico Buarque realmente fala de uma festa de um amigo distante que foi decorada com cravos, a qual talvez não tenha ido por estar doente. Estudando o contexto da época, fica claro que é uma homenagem a “Revolução dos Cravos”, que derrubou, em Portugal, o ditador Marcelo Caetano, herdeiro político de Salazar, e que ao mesmo tempo declara que as coisas não vão bem no Brasil, em pleno governo Médici, da ditadura militar.

Há uma mensagem original do autor? Com o conhecimento que temos do contexto em que esse texto foi criado, ela pode ser descrita como o Eu-lírico afirmando que Fico feliz porque o povo derrubou o ditador em Portugal, mas aqui no Brasil ainda temos uma ditadura e precisamos de alguma inspiração ou apoio para derrubá-la. Com nosso conhecimento histórico, o contexto, podemos dizer que essa era a opinião do autor, o que pode nem sempre ser verdade em relação ao Eu lírico. Sabendo que a música foi censurada, também entendemos que ficou claro para os censores esse significado⁴.

Um outro leitor pode não estar interessado no significado da letra como uma declaração de apoio à revolução, mas sim na capacidade dos censores em perceber mensagens quase

⁴A música pode ser escutada, também na versão de 1978, em <https://www.youtube.com/watch?v=Pj5VuYSmd4k>

Tanto Mar (1975)

Chico Buarque de Holanda

Sei que estás em festa, pá
Fico contente
E enquanto estou ausente
Guarda um cravo para mim

Eu queria estar na festa, pá
Com a tua gente
E colher pessoalmente
Uma flor do teu jardim

Sei que há léguas a nos separar
Tanto mar, tanto mar
Sei também quanto é preciso, pá
Navegar, navegar

Lá faz primavera, pá
Cá estou doente
Manda urgentemente
Algum cheirinho de alecrim!

Figura 2.3.: Letra de 1975 da música Tanto Mar, de Chico Buarque.

que disfarçadas dentro das músicas. Um terceiro leitor pode querer entender quais são as referências à cultura portuguesa. Outro leitor pode estar buscando inspiração para rimas ou tentando entender a poética do autor.

Certos detalhes que dão mais significado dependem de conhecimentos prévios, como o uso de “pá”, típico de Portugal, o uso da segunda pessoa do singular, que não é comum no Rio de Janeiro onde vive Chico Buarque, e a citação à cultura portuguesa que acontece em “Sei também quanto é preciso, pá, Navegar, navegar”⁵. Todas esses significados estão além do texto em si, e ligam o verdadeiro significado a Portugal. Há um detalhe

⁵Referência a frase “navegar é preciso, viver não é preciso”, famosa em língua portuguesa por estar no quase-fado “Os Argonautas” (Velloso, 1969), do artista brasileiro Caetano Velloso, como referência a texto do “O Livro do Desassossego”, de Bernardo Soares, heterônimo de Fernando Pessoa, mas que na verdade é original do general romano Pompeu, conforme Plutarco. No caso, a frase foi dita por Pompeu para convencer os tripulantes a levar suprimento a Roma, que vivia forte crise (navegar é preciso), em face aos riscos da navegação da época (viver não é preciso). Essa frase, então, pode ser associada à necessidade de mudar o Brasil da época, a custo da própria vida. Isso é um exemplo da enorme quantidade de conhecimento que pode estar associada a 8 palavras em uma música (Neto, 2017).

a mais na música que se refere a Portugal, e que não aparece no texto: ela é um fado, mesmo que feito por um brasileiro ⁶.

Vamos supor agora que um pesquisador queira saber se houve influência da Revolução dos Cravos na política ou na cultura brasileira. Certamente essa música é uma prova que alguma coisa aconteceu. Porém, como a mensagem não é passada de forma direta, uma busca pelos termos Salazarismo e Revolução dos Cravos não chegará a ela. Uma busca feita no Google, em 30 de junho 2020, por cravos encontrou 95 artigos seguidos sobre o problema dermatológico até a primeira entrada sobre outro assunto, que é sobre um livro infantil. Apenas a entrada 101 apontou um livro sobre a Revolução dos Cravos, que começa citando a música de Chico Buarque. Se tivesse a paciência para chegar nessa entrada, o pesquisador poderia então buscar a música completa usando palavras chave retiradas desse texto.

Um exemplo clássico de dificuldade de interpretação é o livro *Flatland: A Romance of Many Dimensions*, cuja primeira edição é de 1884, que é reconhecido pela maioria dos seus leitores como uma sátira social à posição da mulher na sociedade, mas por vezes acusado de ser um livro misógino, que defende uma posição inferior, ao ponto do autor ter sentido necessidade de se explicar em sua segunda edição. Seu texto, porém, fala sobre um mundo habitado por quadrados, círculos e linhas. Sua classificação, direta do texto (*full-text*), dificilmente apresentará relação com o tema “relações sociais no século XIX”, já que é, na superfície do texto, uma fantasia.

É claro que textos técnicos de Ciências Exatas tem menos interpretações possíveis que textos narrativos ou líricos, porém ainda assim há complicações. Um mesmo problema, ou técnica de solução, por exemplo, pode ser conhecido por vários nomes⁷.

2.8. O Descasamento entre a Expressão e a Interpretação

A figura 2.4 busca mostrar que existe um descasamento entre a ideia original de um autor e a compreensão, ou interpretação, que o leitor tem sobre o documento produzido pelo primeiro, principalmente quando esse caminho é mediado por um sistema de busca.

Esse descasamento é aumentado pelo ruído inserido pelo processo de registro, indexação e busca, apesar de já existir pelo próprio processamento cognitivo do emissor e do receptor da mensagem.

Na figura há um autor ou emissor, que possui uma ideia original. Ele precisa contar essa ideia, e o faz por meio de uma língua. Nesse ponto já existe o que podemos chamar de ruídos no processo, pois nem toda língua possui palavras para todas as ideias. Por

⁶O leitor pode procurar a versão final da música, lançada apenas em 1978, onde a revolução dos cravos já tinha “murchado”

⁷Já passei bom tempo tentando achar um nome para um tipo de problema que atingi, até que descobri que o problema já tinha sido publicado com seis nomes diferentes.

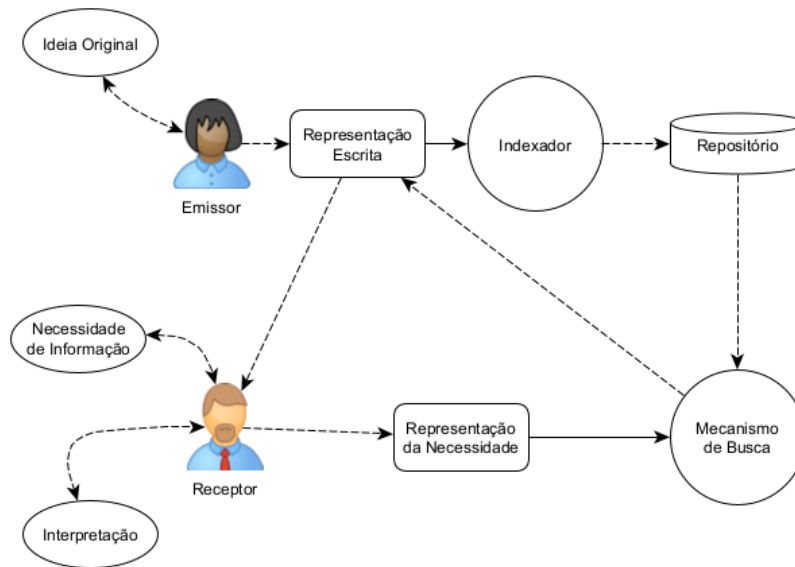


Figura 2.4.: O descasamento entre a expressão e a interpretação de uma ideia. As linhas tracejadas indicam traduções onde há ruído.

exemplo, um escritor brasileiro terá dificuldade de escrever em inglês exatamente o que entendemos por saudade, ou teria dificuldade de descrever em português o conceito alemão de *schadenfreude*, que significa a experiência de prazer causada pela má fortuna de outra pessoa, como rir de alguém que escorrega na casca de banana. Este texto, escrito, é representado de alguma forma que permite sua indexação ou classificação uma base de dados. É possível também imaginar que o próprio autor não sabe muito bem como expressar sua ideia, ou também não sabe ou não quer escrever na norma culta, o que é comum nas redes sociais.

Na parte inferior da figura há outra pessoa, que tem um desejo por alguma informação. Esse desejo pode ser vago, ou mesmo um total desconhecimento do que precisa realmente saber. Ela tem então que expressar esse desconhecimento ou desejo na forma de uma consulta a um sistema de busca e recuperação, que é uma interface com a base de dados. Nessa expressão ela faz suposições sobre o comportamento do sistema e tenta representar, da melhor forma possível, seu desejo de informação. Hoje em dia, porém, na maioria das vezes esse desejo é representado como uma ou poucas palavras.

O sistema então retorna (outras) representações, como o título ou um pequeno resumo, das representações dos documentos que encontrou na base. O usuário então deve selecionar entre as possivelmente várias sugestões do sistema, para poder então ler o texto original do autor, e dar sua interpretação. Tanto na determinação da consulta, na escolha entre as representações apresentadas do documento, quanto na interpretação do documento propriamente dito, existe a interferência de vários fatores, contextuais, cognitivos ou outros, que podem de uma forma geral serem considerados ruídos na comunicação entre o autor (emissor) e o leitor (receptor).

2.9. A Codificação da Escrita no Computador

A escrita é um processo de codificação. Em relação a língua falada, há uma correspondência não muito precisa entre letras, modificadas ou não, e sequência de letras, aos sons que proferimos. Na prática isso hoje obriga os linguistas a ter um conjunto próprio de símbolos para representar os fonemas⁸, sendo o mais usado o Alfabeto Fonético Internacional (International Phonetic Association, 1999). Um exemplo disso pode ser visto na frase em inglês *I ate an apple in August*, onde as vogais representam fonemas diferentes, sendo que às vezes uma vogal representa dois fonemas e outras vezes duas vogais representam um fonema. A figura 2.5 exemplifica a variação de sons de vogais na língua inglesa falada nos EUA.

I ate an apple in August.
aɪ eɪt ən 'æpəl ɪn 'agəst .

Figura 2.5.: Registro fonético da frase *I ate an apple in August* segundo o *International Phonetic Alphabet*, para o inglês americano

As letras podem ser escritas de várias formas no papel. Em geral diferenciamos amplamente em letra cursiva e letra de forma, mas existe uma variedade de formas pessoais de escrever as letras cursivas e de forma, além de fontes criadas por designers que seguem essas características, antes mesmo da invenção do computador.

Quando um texto é colocado no computador ele deve ser codificado de alguma forma. Uma forma é tirar um retrato do texto escrito (*scan*) e guardar o texto na forma de uma imagem. O problema é que processar essa imagem para que o computador possa tratar o texto propriamente dito é um processo muito difícil. Além disso, imagens gastam muita memória e não tornariam a comunicação entre computador e pessoas fácil.

Assim, desde de o começo da Computação, letras, e espaços, foram codificadas em bits, que são guardados na memória do computador. Essa codificação é inspirada em antigas codificações para comunicação, como o código Morse, que usa traços, pontos e espaços, o Baudot, que usa 5 bits, ou mesmo pelo código Braille.

No momento, nos computadores e na maior parte das aplicações, vivemos uma fase final de transição, entre um padrão fortíssimo no passado e ainda existente em documentos antigos, o ASCII, e um novo padrão, o Unicode, que engloba o ASCII, sendo seu formato UTF-8 o mais comum.

Além disso, os documentos mais sofisticados, como os visualizados em processadores de texto como o Word, ou em arquivos PDF, são codificados de forma adicional para indicar sua representação. Por exemplo, um documento com os caracteres “ABCD” pode

⁸Menores unidades sonoras que formam as palavras de uma língua (Araújo, 2022)

ter um código dizendo que eles devem ser apresentados ao usuário, ou impressos, com a fonte Arial.

O importante é, então, notar que existe um texto escrito, que é **representado** no computador na forma de uma sequência de bits por um processo de codificação. De acordo com o tipo de arquivo, esse texto pode ser entremeado com informações para a sua visualização. Isso é um fator complicador para as máquinas, que para tratar o texto precisam limpar tudo que não é o texto. Ou pior, para poder gerar algum contexto, como o uso de negrito e itálico, precisam entender o que não é texto.

A representação de letras e o formato de documentos serão tratados de forma mais detalhada em outros capítulos.

2.10. Distribuição das Palavras

Como qualquer um pode reconhecer, as palavras são usadas com frequência diferente na língua e nos documentos de uma língua. Algumas palavras são muito comuns, como “não” e outras raras, como “filauçioso”, um sinônimo de presunçoso. Em inglês, no Brown Corpus, a palavra “the” sozinha representa 7% de todas as palavras.

George Kingsley Zipf, um professor de linguística de Harvard, propôs o que é hoje conhecido como a Lei de Zipf: a frequência de ocorrência de algum evento P em função do seu rank n , quando o rank é determinado pela frequência de ocorrência, segue uma lei de potência $P_n \approx \frac{1}{n^a}$ com a perto da unidade.

Essa propriedade pode ser verificada observando a distribuição das 10.000 palavras mais frequentes presentes no corpus Gutenberg, fornecido com o NLTK (Bird, Loper e Klein, 2009), apresentada na Figura 2.6.

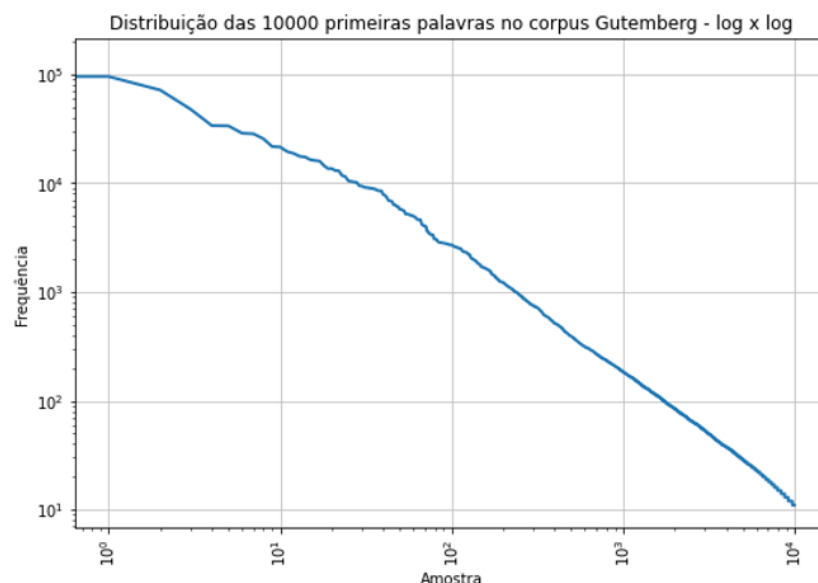


Figura 2.6.: Distribuição das 10000 primeiras palavras no corpus Gutenberg, fornecido com o NLTK (Bird, Loper e Klein, 2009).

2.11. Modelos Conceituais para Documentos

Nesta seção são apresentados modelos conceituais, isto é, formas de entender um documento. Esses modelos são descritos de várias maneiras, modelos de entidades e relacionamentos, ontologias, coleções de metadados, sendo que algumas vezes o foco é o modelo em si, em uma descrição ontológica do que é o documento, outras vezes é uma forma de trocar informação, onde a definição do formato é mais importante.

2.11.1. FRBR

Os Requisitos Funcionais para Registros Bibliográficos, **FRBR**, é um modelo de entidades e relacionamentos que “fornece um arcabouço estruturado, claramente definido, para relacionar dados que são registrados em registros bibliográficos e para as necessidades dos usuários desses registros” (IFLA, 2009). É, na prática, uma definição formal de termos usados de forma arbitrária anteriormente (Tillett, 2004). O FRBR é parte do *IFLA Library Reference Model*, que inclui também o FRAD, *Functional Requirements for Authority Data* e o FRSAD, o *Functional Requirements for Subject Authority Data* (Riva, Buf e umer, 2017).

As entidades do FRBR estão divididas em grupos. O Grupo 1, na Figura 2.7 esclarece os termos Obra, Expressão, Manifestão e Item. A ideia é que Obras são realizadas por Expressões que estão incorporadas em Manifestações, que estão exemplificadas em Itens. Um conto, por exemplo, é uma obra, que pode ser expressa na forma de um texto, ou um áudio, que podem ser publicados sozinhos, ou incorporados em outra manifestação,

como uma coletânea, e que são manipulados com unidades específicas, como exemplares de livros ou CDs.

Deve ficar claro que sempre trabalhamos com itens, porém o que miramos é normalmente o significado da Obra.

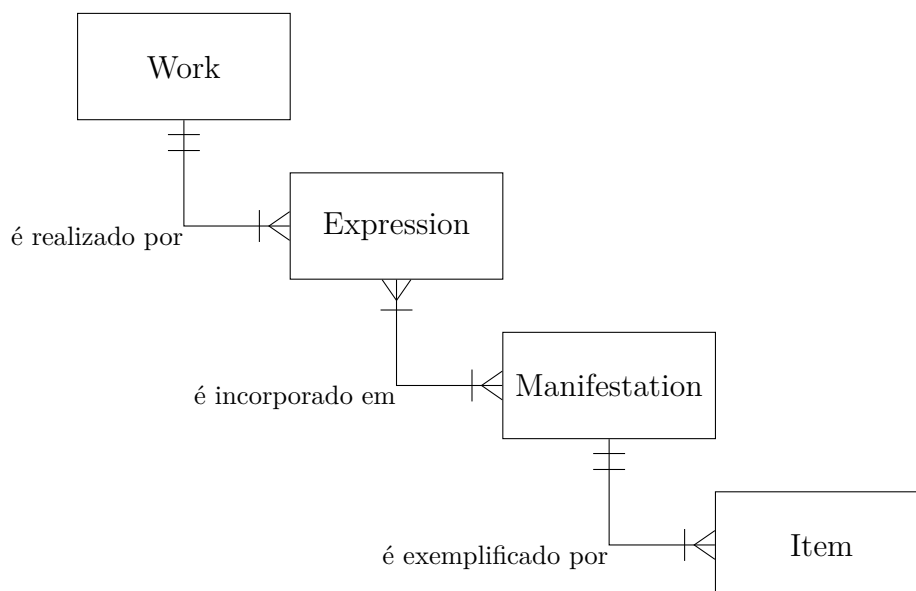


Figura 2.7.: Modelo de Entidades e Relacionamentos para o FRBR Group 1, baseado em (IFLA, 2009)

O Grupo 2 discute responsabilidades, e apresenta duas entidades: Pessoa e Corpo Corporativo. A ideia é que uma Obra é criada por uma ou mais entidades do Grupo 2, que por sua vez pode realizar expressões, produzir manifestações e possuir itens (IFLA, 2009).

O Grupo 3 também apresenta novas entidades: Conceito, Objeto, Evento ou Lugar. Ele discute relacionamentos da Obra. Uma Obra, então, tem como assunto entidades dos três grupos (IFLA, 2009).

O modelo também propõe atributos. Uma obra, por exemplo, possui os atributos: título, forma, data, outras características de distinção, contexto, ... (IFLA, 2009). Não cabe a este texto destrinchar o FRBR, mas algumas observações são importantes.

A primeira é que todo o trabalho de processamento é feito sobre itens, na verdade, representações de itens na forma digital. Porém, toda o resultado visa tratar das obras e de seus múltiplos significados.

A segunda, de ordem mais prática, é que o FRBR vê o documento, ou obra, como um todo.

2.11.2. DoCO

DoCO é a **Document Components Ontology**, uma ontologia de que “fornece um vocabulário escrito estruturado sobre components de documentos, sendo eles estruturais (como blocos, parágrafos, seções) ou retóricos (introdução, discussão, figura)” (Constantin et al., 2016). É composto de três partes: *Document Components*, *Discourse Elements Ontology* e *Pattern Ontology* (Constantin et al., 2016). Sua arquitetura é descrita na Figura 2.8.

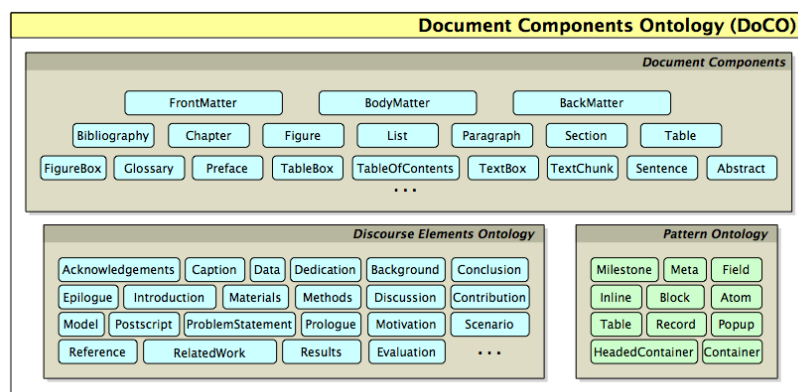


Figura 2.8.: Arquitetura do DoCO, obtida sob licença Creative Commons Atribuição 3.0, (Shotton e Peroni, 2015)

Fica claro que DoCO, ao contrário do FRBR, permite discutir a estrutura do documento de pelo menos duas formas.

2.11.3. Dublin Core

DCMI é o acrônimo do **Dublin Core Metadata Initiative**, uma “organização que suporta a inovação em *design* de metadados e melhores práticas na ecologia de metadados” (DCMI, 2022a). Sua origem, e missão atual, é manter o Dublin Core, um padrão de termos de meta-dados criado para a Web em 1995 (DCMI, 2022c), e que hoje é parte de diferentes padrões, como o ANSI/NISO Z39.85-2021 (NISO, 2013). Essa organização mantém diferentes formas de metadados, isto é, dados que descrevem documentos, entre elas o *Dublin Core* (DCMI, 2022c), que hoje é composto, entre outros documentos, do *DCMI Metadata Terms* (DCMI, 2022b).

Os principais termos definidos no Dublin Core (NISO, 2013) são: *title*, *creator*, *subject*, *description*, *publisher*, *contributor*, *date*, *type*, *format*, *identifier*, *source*, *language*, *relation*, *coverage* e *rights*.

2.11.4. Descrições orientadas ao formato

Um conjunto de metadados define um modelo conceitual do que é o dado, porém, nos formatos descritos nessa seção, o foco principal é o formato.

MARC, ou *Machine Readable Cataloging Record* é um padrão, hoje na versão MARC 21, mantido pela Biblioteca do Congresso Amricano e utilizado mundialmente, para registros bibliográficos. Ele é composto de vários formatos de registros: bibliográfico, autoridades, posses, classificação e informação sobre comunidades (Library of Congress, 2020). Os formatos são expressos em sequências de bytes, porém existe uma versão MARC XML, porém ela não foi definida sobre a semântica, mas sobre os campos do MARC. BIBFRAME é um padrão que se propõe a substituir o MARC (Library of Congress, 2022).

Outros formatos conhecidos para descrição de metadados de documentos são os usados por sistemas gerenciadores de referências bibliográficas, entre eles: .bib, .RIS, refer, e o formato de softwares muito usados como EndNote, Papyrus, Zotero, Mendeley, etc.

Atualmente, muitos metadados o descritos em XML ou JSON. Por exemplo, o Twitter descreve seus tweets em JSON quando eles são recuperados.

2.12. Coleções de Documentos

Atkins, Clear e Ostler (1992) definem quatro tipos de coleções de textos:

- **Arquivo:** um repositório de textos eletrônicos legíveis que não são ligados de uma forma coordenada;
- **Biblioteca Eletrônica de Textos:** uma coleção de textos eletrônicos em um formato padronizado dentro de certas convenções relacionadas a conteúdo, etc,mas sem restrições rigorosas de seleção;
- **Corpus:** uma parte da biblioteca eletrônica, construído a partir de critérios explícito, para um objetivo específico, e
- **Subcorpus:** uma parte de um corpus, podendo ser um componente estático de um corpus complexo ou uma seleção dinâmica de um corpus feita durante uma análise *on-line* (Atkins, Clear e Ostler, 1992).

Várias áreas de pesquisa têm interesse em corpus, sendo seu uso muito comum para validação de algoritmos e aprendizado de máquina, na Computação, mas também para estudos de linguística. Alguns corpus são muito usados, como:

- Em inglês
 - *Brown University Standard Corpus of Present-Day American English*, conhecido como **Brown Corpus**, criado em 1961, com mais de 1 milhão de palavras e 500 documentos, disponível no NLTK.
 - Vários corpus fornecidos pela agência de notícias *Reuters*: Reuters, Reuters-21578, RCV1, RCV2, TRC2.
 - Corpus com valor histórico, considerados pequenos atualmente, mas que podem ser utilizados para testes, como o Cystic Fibrosis e a Cranfield Collection.

- Movie Reviews, usado para análise de sentimentos.
- GOV2 Test Collection.
- 20 Newsgroup dataset.
- TIPSTER.
- Em português, disponibilizados pela Linguateca⁹
 - CETEM Público, um corpus de aproximadamente 180 milhões de palavras em português de Portugal formado por notícias do jornal Público.
 - CETEM Folha, um corpus de cerca de 24 milhões de palavras em português do Brasil formado por notícias da Folha de São Paulo.
 - CHAVE, contendo textos do Público e da Folha de São Paulo, em 1994 e 1995, com um total de 210.734 notícias.
 - Floresta Treebank, de português brasileiro, disponível no NLTK.
 - MacMorpho Corpus, de português brasileiro, disponível no NLTK.
 - ReLi, para análise de sentimentos.

Alguns corpora podem ser livremente acessados, outros são oferecidos gratuitamente sob licença para pesquisadores, e ainda existem os pagos. Por exemplo, o corpus English Gigaword Fifth Edition¹⁰ custa US\$6,000, para não sócios do LDC¹¹. Já os corpus Reuters são gratuitos, mas entregues sob a assinatura de uma licença, e todos os corpus do NLTK são gratuitos e usados praticamente de forma livre.

2.13. Exercícios

Exercício 2.1:

Considere o artigo “Fellipe Duarte, Danielle Caled, Geraldo Xexéo: Minmax Circular Sector Arc for External Plagiarism’s Heuristic Retrieval stage. Knowl. Based Syst. 137:1-18 (2017)”¹². Faça as seguintes descrições do artigo:

- Com o Dublin Core, usando XML.
- Com DoCO, usando a notação oferecida em <http://www.sparontologies.net/examples>.
- Com o formato do bibtex, ou biblatex.

Exercício 2.2:

Considere os formatos indicados no exercício anterior, discuta suas similaridade e diferenças, e adequabilidade ao uso.

Exercício 2.3:

⁹<https://www.linguateca.pt/>

¹⁰<https://catalog.ldc.upenn.edu/LDC2011T07>

¹¹A subscrição padrão do LDC custa US\$2,400

¹²<https://doi.org/10.1016/j.knosys.2017.08.013>

Que formato parece mais adequado, e por que, para descrever cada documento em uma coleção destinada a guardar:

- livros,
- arquivos científicos,
- páginas web recuperadas sobre um assunto,
- mensagens de tweeter, e
- processos legais,
- leis e decretos,
- patentes,
- mensagens de correio eletrônico,
- notícias de jornal.

Exercício 2.4:

Investigue na internet, e relate, se existem formatos, internacionais, de outras nações, e especialmente brasileiros, específicos de metadados para:

- leis e decretos,
- processos burocráticos do governo,
- processos jurídicos em um tribunal,
- patentes, e
- notícias de jornal.

Exercício 2.5:

Usando o site <https://visl.sdu.dk/visl/pt/parsing/automatic/complex.php>, faça a análise sintática da frase “O menino envergonhado viu ontem o homem sozinho com o binóculo novo”. Comente a solução, única, apresentada pelo analisador e qual o seu significado.

Exercício 2.6:

Explique por que a sentença “A mãe vestiu a menina com o vestido vermelho”, que aparentemente tem a mesma estrutura de “o menino viu o homem com o binóculo”, não oferece ambiguidade para um leitor, mas pode ser complexa para um programa de computador.

Exercício 2.7:

Análise o formato BIBFRAME, principalmente em relação ao formato MARC 21 e ao padrão FRBR. Discuta como os tags do formato BIBFRAME foram criados em relação a expectativa que temos de uma arquivo XML.

Exercício 2.8:

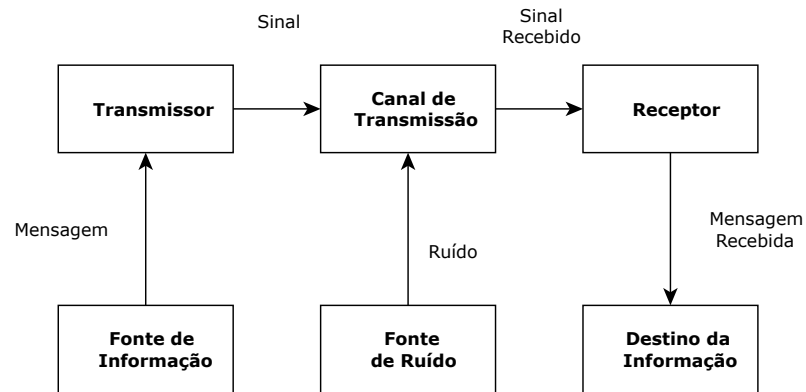


Figura 2.9.: Canal de Ruído de Shannon

A Figura 2.9 mostra o modelo clássico de canal de ruído de Shannon. Discuta como podemos modelar o processo de comunicação entre duas pessoas por meio de um texto usando esse conceito.

Exercício 2.9:

Escolha um modelo conceitual para documentos e o descreva usando entre 1.200 e 1.500 palavras.

Exercício 2.10:

Escolha um corpus e o descreva usando entre 500 e 750 palavras.

Exercício 2.11:

Faça um programa que execute os quatro algoritmos de *stemming* em inglês do NLTK que foram expostos aqui com caches e teste tamanhos de cache que sejam potência de 2, de 2 a 2048. Compare velocidades e o tamanho ideal do cache.

Codificação de Caracteres

Um texto, no computador, é uma sequência de caracteres disponibilizado de diversas formas: como arquivos, mensagens em redes sociais, guardados em bancos de dados, gerados por meio de programas, etc.

Para processar um texto, o primeiro passo é entender que caracteres foram usados para construí-lo. Este capítulo trata da representação de caracteres em computadores.

3.1. A representação de informação

Alguns autores dizem que os computadores processam números, e que é preciso representar caracteres como números para que os computadores os entendam, mas isso é uma simplificação falsa.

Na verdade, computadores processam sinais binários, os bits, e esses sinais são agrupados, de forma arbitrária e criada pelos fabricantes, em palavras, ou bytes¹

A escolha de 8 bits, e seus múltiplos, como tamanho da CPU, foi tomada ao longo do tempo, tanto por necessidades ou facilidades técnicas, como por tendências de mercado e necessidades de retrocompatibilidade. Deve ficar claro que, apesar dos computadores terem instruções especializadas para operações matemáticas, essas instruções supõem também uma representação, seja ela binário com complemento-de-2, Binary Coded Decimal (BCD)², IEEE 470 para números de ponto flutuante ou outra. O conjunto de instruções do Intel i7 possui pelo menos 14 tipos diferentes de adição, de acordo com a representação de um número.

¹A palavra *byte*, hoje associada a exatamente 8-bits, já significou apenas uma unidade completa que podia ser processada pelo computador.

²Na verdade existem vários formatos de BCD, que podem ser vistos na página da Wikipedia https://en.wikipedia.org/wiki/Binary-coded_decimal

O uso de 4 bits em um dos primeiros microprocessadores produzidos, o Intel 4004, garantia que era possível guardar pelo menos 1 algarismo decimal descrito no formato BCD, que seria usado para um cliente fazer calculadores (Leibson, 2021).

Os 8-bits são usados atualmente, entre outros motivos, pelo fato de terem sido lançados micro-processadores de 8-bits que causaram a revolução da microcomputação, como o Intel 8080, o Zilog Z-80 e o MOS Technology 6502, sendo que a partir daí foram usados múltiplos de 8, como 16, 32, 64 e 128, para manter uma compatibilidade. Os 8-bits, inicialmente, era apenas um dos tamanhos possíveis da “palavra” do computador. Por exemplo, os CDC Cyber 170, no final dos anos 1970, usavam caracteres de 6-bits, endereços de 18 bits, instruções de 15 ou 30 bits e palavras de 60 bits, e para os quais um “byte”, tinha 12 bits (Control Data Corporation, 1975).

Assim, fica estabelecido que tanto a representação típica em 8-bits, quanto o uso de números para indicar como um caracter é representado, são construções culturais e tecnológicas que não significam que caracteres são números, ou que 8 bits sejam realmente necessários.

Em todo caso, para evitar escrever caracteres como sequência de bits, normalmente é usada uma a representação como inteiro decimal (ou hexadecimal) da sequência de bits usadas no computador para descrever seu código.

Assim, um caracterer 'A', em código ASCII e em UTF-8, é representado como a sequência de 8 bits '01000001', ou com o valor inteiro decimal 65, ou ainda pelo valor inteiro hexadecimal 0x41³. Já em código EBCDIC, hoje de uso muito restrito, é representado o valor 193 ou 0xC1.

Computadores têm que ser binários?

Não há nenhuma obrigação real ou lei da física que obrigue os computadores a serem baseados na lógica binária. Porém é mais fácil trabalhar com a unidade mínima de informação, o bit, e juntá-los em grupos para otimizar o processamento.

Tanto Knuth (1997, p. 208) quanto Claude Shannon expressaram opinião que um computador baseado em lógica ternária seria mais interessante que um binário.

Para exemplificar é possível citar o Setun um computador que usava lógica ternária, criado sob liderança de Sergei Sobolev e Nikolay Brusentsov, na Moscow State University, e fabricado pela empresa Kazan (Brousentsov et al., 2021; Wikipedia, 2020).

³A partir de agora, quando necessário, principalmente para diferenciar de números em decimal, hexadecimais serão representados por números começando com os caracteres “0x”. Em outros momentos, quando estiver claro o contexto, serão escritos em grupos de dois caracteres.

3.2. Da variedade para o ASCII

No início da computação a preocupação com padrões de representações de textos era pequena, e cada fabricante foi criando uma regra própria. Linguagens de programação, como APL, podiam exigir caracteres específicos.

Nas comunicações, porém, a necessidade de padrões para troca de comunicação já era estabelecida. O código Morse, baseado em “pontos e traços”, possibilitou a telegrafia. Mackenzie (1980) descreve várias codificações para transmissão de informações, como o código CCITT⁴ # 2, para telégrafos, com 58 caracteres e 6 bits, estabelecido em 1931 e ainda códigos específicos para cartões perfurados.

Já no final do anos 1950, foi iniciada a discussão de um novo código, que se tornaria o *American Standard Code for Information Interchange* (ASCII)⁵, apresentado na Tabela 3.1. Durante a criação apareceram como requisitos fornecer: letras, numerais, pontuações, símbolos comerciais e matemáticos e ainda caracteres de controle. Isso levou à necessidade de representar mais de 64 caracteres, ultrapassando os 6-bits usados até então, logo pelo menos 7 bits deveriam ser usados. Existiam fortes argumentos para o uso de 8-bits, mas eles foram vencidos pelos defensores dos 7-bits, que economiza bits, e decorrentemente, o tamanho de uma mensagem enviada de forma serial. Entre os argumentos dos defensores dos 7 bits estava: “As 128 combinações disponíveis em um conjunto de caracteres de 7-bits satisfaz os requisitos de troca de informação e controle da grande maioria dos usuários” (Mackenzie, 1980). As discussões sobre o tema são bastante técnicas, e seus efeitos perduram até hoje, porém mostram a ingenuidade do início da Computação, além de uma preocupação imediata com o alfabeto da língua inglesa e a ausência de letras acentuadas. Além disso, existiram vários padrões ASCII.

Mas como o ASCII assumiu um papel preponderante e venceu todas as outras codificações? Isso pode ser parcialmente atribuído ao “memorandum 127”, assinado em 11 de março de 1968, pelo presidente americano Lyndon B. Johnson, que não só adotava o ASCII com padrão federal, mas também obrigava que qualquer computador comprado tivesse a capacidade de usar não só o ASCII, mas também os formatos prescritos para fita magnética e de papel (Johnson, 1968)⁶. Outro fator importante foi sua adoção pelos microcomputadores. Mesmo assim, a IBM, usava e ainda usa, em alguns de seus sistemas, como o z/OS, um formato criado por ela como padrão em seus sistemas, conhecido como EBCDIC, *Extended Binary Coded Interchange Code* (IBM, 2010).

A maioria dos programadores, mesmo hoje em dia, têm a necessidade, em algum momento, de se referir a Tabela ASCII, isto é, aos números usados para representar os caracteres de acordo com o “Código Padrão Americano para Intercâmbio de Informações”, que pode ser visto na Tabela 3.1. Não só ela é importante historicamente, aparece em livros de programação, como é mantida pelo código mais novo, o Unicode UTF-8.

⁴ *Comité Consultatif International Telegraphic et Telephonique*

⁵ Muitos pensam ser o número 2 em algarismos romano, mas a pronúncia correta é *ass-kii*

⁶ Uma forma de entrada e saída comum na época e hoje abandonada, como os cartões perfurados.

Por que esses códigos

Os códigos ASCII não foram escolhidos de forma totalmente arbitrária.

Os dígitos, por exemplo, vão do número 0x30 até 0x39. Isso significa que se seus bits iniciais forem jogado fora, os últimos representam o dígito em código decimal binário.

Já as letras foram escolhidas de forma que somando ou subtraindo 32 a elas, ou seja, ligando apenas um bit, fosse possível trocá-las de letras maiúsculas para minúsculas e vice-versa.

Tabela 3.1.: Tabela ASCII, ou Unicode com a representação UTF-8

Dec	Controle	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	‘
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(72	H	104	h
9	TAB (horizontal tab)	41)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

3.3. Code pages

Como ficou demonstrado mais tarde, 7 bits foram insuficientes para as reais necessidades dos usuários. Eles não cobriam, por exemplo, caracteres acentuados, ou outros alfabetos. Com isso foram criadas centenas de variações que usavam os 8 bits disponíveis em um byte para codificar caracteres, o que foi chamado de *code pages*. São mais de 200 code pages no Microsoft Windows, outras para sistemas operacionais diferentes.

Depois dos fabricantes lançarem seus próprios formatos, a ISO acabou criando várias extensões de 8 bits para o ASCII, nos padrões ISO-8859. Entre eles, os mais conhecidos no Ocidente são o ISO 8859-1, ou *ISO Latin 1*, que atendia a maioria das línguas da Europa Ocidental, o ISO 8859-2, para as línguas do Leste Europeu, o ISO 8859-5 para o alfabeto Cirílico.

Os padrões ISO são levemente diferentes das *code pages* da Microsoft. Devido ao poder da Microsoft, a codepage 1252, um super-conjunto da ISO 8859-1, acabou sendo o padrão de fato para as linguagens ocidentais, incluindo o próprio inglês.

A expansão de padrões e *code pages* levou a uma nova tentativa de padronização. Essa tentativa se consolidou na padronização dos caracteres no Unicode, que fornece um código para cada caracter, chamado de *code point*, e uma forma de representação dos *code points* em uma sequência de bits.

3.4. Unicode

Unicode é um padrão internacional que “fornece um número único para cada caracter, não importa que plataforma, dispositivo, aplicação ou linguagem”(Unicode, Inc., 2017). Unicode apresenta um conjunto de restrições adicionais ao padrão ISO/IEC 10646, porém possui os mesmos códigos de caracteres e formas de codificação que o padrão, sendo ambos sincronizados.

Os códigos são representados por números inteiros decimais, sendo facilmente mapeáveis nos bytes do computador moderno. Os primeiros 127 caracteres do Unicode são compatíveis com os 127 códigos do ASCII. Em sua versão 14.0 o padrão Unicode continha 144.697 caracteres de várias linguagens ou, mais corretamente, *scripts*, incluindo gráficos (Unicode, Inc., 2021).

O Unicode não representa linguagens, mas sim coleções de letras e símbolos usadas para representar informação em um sistema de escrita, conceito conhecido como *scripts*. Um *script* pode representar um sistema de escrita atual, antigo ou histórico. *Scripts* não são iguais a sistemas de escrita. Por exemplo, japonês pode ser escrito nos *scripts* Han, Hiragana ou Katakana, seguindo o mesmo sistema de escrita.

Unicode também não indica como as letras, ou outros símbolos, são escritas. Assim, todas as linguagens que usam um “a” minúsculo, usam o mesmo “a” minúsculo, e a

forma de representá-lo é escolhida por meio de uma fonte, como Arial, que indica como o “a” minúsculo é desenhado.

3.5. UTF-8

Os caracteres Unicode, para ser usados, precisam ser representados de alguma forma. A forma mais direta é o **UTF-32** (Unicode Transformation Format), que é simplesmente a representação do número do caracter Unicode em 32 bits, o que permite 4.294.967.295 caracteres. Isso significa, porém, que cada caracter usa 4 bytes, o que é um desperdício. UTF-8 pode codificar 1.112.064 *code points* do Unicode (Unicode, Inc., 2019).

A forma mais usada para representar os caracteres é o **UTF-8**, um código de tamanho variado, que representa os caracteres Unicode com 8, 16, 24 ou 32 bits (Unicode, Inc., 2019). No Windows, UTF-8 corresponde a *code page* 65001. A Tabela 3.2 exemplifica caracteres unicodes com sua representação em UTF-8 com vários tamanhos.

Tabela 3.2.: Alguns caracteres Unicode e suas representações UTF-8

Unicode		UTF-8		
Nome	Código	Exemplo	Decimal	Hexadecimal
LATIN CAPITAL LETTER A	U+0041	A	65	41
GREEK SMALL LETTER ALPHA	U+03B1	α	206 177	CE B1
EURO SIGN	U+20AC	€	226 130 172	E2 82 AC
GRINNING FACE EMOJI	U+1F600	☺	240 159 152 128	F0 9F 90 80

3.6. O fim de linha

Arquivos ou mensagens em texto normalmente são divididas em linhas. Lamentavelmente, motivos diferentes levaram os criadores dos sistemas operacionais a usar formas diferentes de indicar um fim de linha.

Seguindo a tradição dos teletipos, o MSDOS e MS Windows adotam dois caracteres para indicar o fim de linha, o CR (Carriage Return) e o LF (Line Feed), nessa ordem. Como teletipos usavam folhas de papel, para mudar de linha era necessário voltar o “carro”, isto é, a cabeça de impressão, e pular uma linha na folha.

Os sistemas baseados em Unix usam um só caracter, o LF. Isso inclui o Mac OSX em diante. Curiosamente, o sistema do Mac pré-OSX usava apenas o CR.

Em EBCDIC existe o caracter EOL (*end of line*).

3.7. Desafios apresentados pela codificação

O principal desafio apresentado pela codificação é o fato de ser impossível conhecer qual é a codificação antes de processar um texto, seja na forma de arquivo ou mensagem, e, ao mesmo tempo, é essencial conhecê-la para poder fazer o processamento correto.

Por exemplo, seja um arquivo que aberto com um editor de texto simples que apresenta os seguintes 12 caracteres:

aáãäåëèêëçÇ

Se salvo com a codificação UTF-8, ele apresentará os seguintes bytes (em hexadecimal):

61 C3 A1 C3 A0 C3 A2 C3 A4 65 C3 A9 C3 A8 C3 AA C3 AB C3 A6 C3 87

Já se codificado com a code page 1252 (Western European) do MS Windows, apresentará os seguintes bytes:

61 E1 E0 E2 E4 65 E9 E8 EA EB E7 C7

Ao programar, isso exige algum cuidado. Por exemplo, em Python (Van Rossum e Drake, 2009) existem várias variáveis do sistema que indicam a codificação. A leitura de arquivos é feita, por default, usando a que está em `locale.getpreferredencoding()`.

O programa Programa 3.1 lê dois arquivos apresentados anteriormente, um em UTF-8 e outro na *code page* 1252. Ele mostra vários valores de codificação do sistemas indicando UTF-8, mas o único importante indica CP1252. Se o arquivo em CP1252 for lido em UTF-8 há um erro, pois alguns códigos não são permitidos em UTF-8.

Programa 3.1: Exemplo de leitura com codificação

```

1 import sys
2 import locale
3
4 print("Default encoding: {}".format(sys.getdefaultencoding()))
5 print("File system encoding: {}".format(sys.getfilesystemencoding()))
6 print("Local preferred encoding: {}".format(locale.getpreferredencoding()
    ↪ ))
7
8 f = open("Exemploutf8.txt", "r")
9 print("Default open encoding: {}".format(f.encoding))
10 for i in f:
11     print(i)
12 f.close()
13
14 f = open("Exemploutf8.txt", "r", encoding="utf-8")

```

```
15 print("Forcing utf-8, we get: {0}".format(f.encoding))
16 for i in f:
17     print(i)
18 f.close()
19
20 # File in cp1252
21 f = open("ExemploWindowsWestern1251.txt", "r")
22 print("Default open encoding: {0}".format(f.encoding))
23 for i in f:
24     print(i)
25 f.close()
26
27
28 f = open("ExemploWindowsWestern1251.txt", "r", encoding='cp1252')
29 print("Forcing cp-1252, we get: {0}".format(f.encoding))
30 for i in f:
31     print(i)
32 f.close()
```

O resultado de executar esse programa é⁷:

Saída do Programa 3.1

```

1 Default encoding: utf-8
2 File system encoding: utf-8
3 Local preferred encoding: cp1252
4 Defaut open encoding: cp1252
5 aÃäÅ ÃçÄðÃŕÃĺÃŕÃñÃğÃ
6 Forcing utf-8, we get: utf-8
7 aääääëèèèëçÇ
8 Defaut open encoding: cp1252
9 aääääëèèèëçÇ
10 Forcing cp-1252, we get: cp1252
11 aääääëèèèëçÇ

```

Recomenda-se fortemente que toda abertura de arquivo especifique a codificação, principalmente porque não há garantia que o a codificação preferida seja UTF-8 em todas as plataformas que o Python executa.

Atualmente, a grande maioria dos textos usados na internet está sendo produzida em UTF-8, porém isso não é uma verdade absoluta. Além disso, muito texto hoje contido em banco de dados está em um formato diferente, baseado no ASCII, mas usando uma *code page* que precisa ser identificada.

⁷Esperando que todos os caracteres apareçam aqui como aparecem no console IPython.

Alguns formatos, como o XML, podem indicar a codificação. Por exemplo, um *Prolog* XML, como o da Figura 3.1, é um elemento opcional no arquivo XML, mas se existe, pode indicar a codificação. Além disso, o padrão XML define UTF-8 como a codificação padrão.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Figura 3.1.: Um *Prolog* XML indicando a codificação

Já uma página obtida pelo protocolo HTTP possui um cabeçalho “Content-type”, que supostamente forneceria a codificação correta. Isso também deveria ser padrão nos arquivos HTML por meio da marcação `<meta_http-equiv="content-type">`.

UTF-8 é a codificação default para os formatos: HTML5, CSS, JavaScript, PHP e SQL. Também Python 3 adota Unicode, isso significa que a codificação padrão de leitura de programas Python supõe arquivos em Unicode. Isso também significa que se uma string Python é convertida em uma lista de bytes, a string e a lista podem ter tamanhos diferentes.

Uma *stream* qualquer UTF (como um arquivo) pode começar indicando sua codificação com uma sequência específica conhecida como *Byte Order Mark* (BOM), que é um caracter U+FEFF. Em UTF-8 isso é a sequência 0xEF,0xBB,0xBF. Esse uso não é uma recomendação e é, na prática, incomum.

3.8. Detectando uma *code page* automaticamente

Não existe uma forma padrão de se detectar a codificação de um texto que é desconhecido. Existem porém heurísticas que podem ser usadas.

O pacote `chardet` 4.0.0] é um “Universal encoding detector for Python 2 and 3”, i.e., se propõe a resolver esse problema em Python, por meio de modelos treinados. Ele pode ser encontrado em <https://pypi.org/project/chardet/>.

Em sua documentação (Pilgrim, Blanchard e Cordasco, 2015) o pacote apresenta um exemplo simples de uso, via o interpretador:

Programa 3.2: Exemplo simples de uso do `chardet`

```
1 >>> import urllib.request
2 >>> rawdata = urllib.request.urlopen('http://yahoo.co.jp/').read()
3 >>> import chardet
4 >>> chardet.detect(rawdata)
5 {'encoding': 'EUC-JP', 'confidence': 0.99}
```

No mesmo documento, também apresentam uma solução para descoberta da codificação em um programa:

Programa 3.3: Exemplo de programa que usa o chardet

```

1 import urllib.request
2 from chardet.universaldetector import UniversalDetector
3
4 usock = urllib.request.urlopen('http://yahoo.co.jp/')
5 detector = UniversalDetector()
6 for line in usock.readlines():
7     detector.feed(line)
8     if detector.done:
9         break
10 detector.close()
11 usock.close()
12
13 print(detector.result)

```

Também é possível usar o pacote `Charset_Normalizer`⁸. Esse pacote é capaz de trabalhar de forma semelhante ao `chardet`, como mostra o Programa 3.4.

Programa 3.4: Uso do charset-normalizer.

```

1 import os
2 from charset_normalizer import detect
3
4 path = 'F:\Github\Dados-Exercicios-Texto\Capitulo Codificacao de
      ↳ Caracteres'
5
6 os.chdir(path)
7 files = os.listdir(os.getcwd())
8
9
10 for fname in files:
11     f = open(fname, "rb")
12     result = detect(f.read())
13     print(fname+": "+result["encoding"])

```

O pacote `charset-normalizer` pode já fornecer o texto normalizado, usando o `best()`, como no Programa 3.5.

Programa 3.5: Uso do charset-normalizer com normalização automática.

```

1 import os
2 from charset_normalizer import from_path
3
4 LINES = 5

```

⁸<https://pypi.org/project/charset-normalizer/>

```
5 COLUMNS = 72
6
7 path = 'F:\Github\Dados-Exercicios-Texto\Capitulo Codificacao de
    ↪ Caracteres'
8
9 os.chdir(path)
10 files = os.listdir(os.getcwd())
11
12 text = str(from_path("base.txt").best())
13
14 for i in range(LINES):
15     print(text[i*COLUMNS:i*COLUMNS+COLUMNS])
```

Saída do Programa 3.5

```
1 \chapter{Arquivos Textuais}
2
3 Nesse capítulo é feita uma introdução aos
4 formatos de arquivos que encontramos e que devem ser processados para q
5 ue possam ser realizadas tarefas de indexação, busca e aprendizado de má
6 quina com textos.
7
8 Muitos desses formatos não podem ser lidos diretame
9 nte por seres humanos, ou podem ser lidos apenas em certas condições, en
```

Esses programas, porém, podem apresentar erros mesmo quando atingem graus de certeza muito altos, assim, o programador deve ficar atento.

3.8.1. Lendo com codificações em Python

É simples abrir um arquivo para leitura com uma codificação específica, bastando para isso declarar o argumento `encoding` no comando `open`, como no Programa 3.6.

Programa 3.6: Comandos para abrir um arquivo escolhendo a codificação

```
1 f = open(fname, encoding="utf-8")
2 g = open(gname, encoding="iso-8859-1")
```

Assim para ler um arquivo tentando prever sua codificação, com o pacote `chardet` são necessárias duas leituras, sendo que a primeira deve fazer a leitura de forma binária (em Python), como é feito no Programa 3.7. Já para o pacote `charset-normalizer`, o Programa 3.5 mostra como isso pode ser feito em uma só leitura.

Programa 3.7: Exemplo de programa que usa o `chardet` e lê o arquivo

```
1 from chardet.universaldetector import UniversalDetector
```

```

2
3 # lê para prever
4 f = open("Exemploutf8.txt", "rb")
5 detector = UniversalDetector()
6 for line in f:
7     detector.feed(line)
8     if detector.done: break
9 detector.close()
10 f.close()
11
12 print(detector.result)
13
14 # lê para obter as linhas de forma correta
15
16 f = open("Exemploutf8.txt", "r", encoding=detector.result["encoding"])
17 for line in f:
18     print(line)
19 f.close()

```

3.9. Tratando a codificação no KNIME

Como o KNIME possui nós com comportamentos distintos, duas estratégias são possíveis para tratar a codificação de textos (strings).

A primeira é usar a codificação default do KNIME. Ela pode ser alterada no arquivo `knime.ini` ou nas preferências em General:Workspace:Text File Encoding. O valor default para o KNIME é CP1252. Isto é exemplificado na Figura 3.2.

A outra opção é usar um módulo “File Reader”, exemplificado na Figura 3.3. Ele é feito para ler arquivos de dados, mas pode ser usado também para ler um texto linha a linha. Para isso é possível marcar o delimitador de coluna como “\r”, isto é, o caracter LF, como na Figura 3.4, e usar o tab Encoding para definir a forma de codificação, como na Figura 3.5. Caso a codificação esteja errada, é normalmente possível identificar no *preview*, como na Figura 3.6.

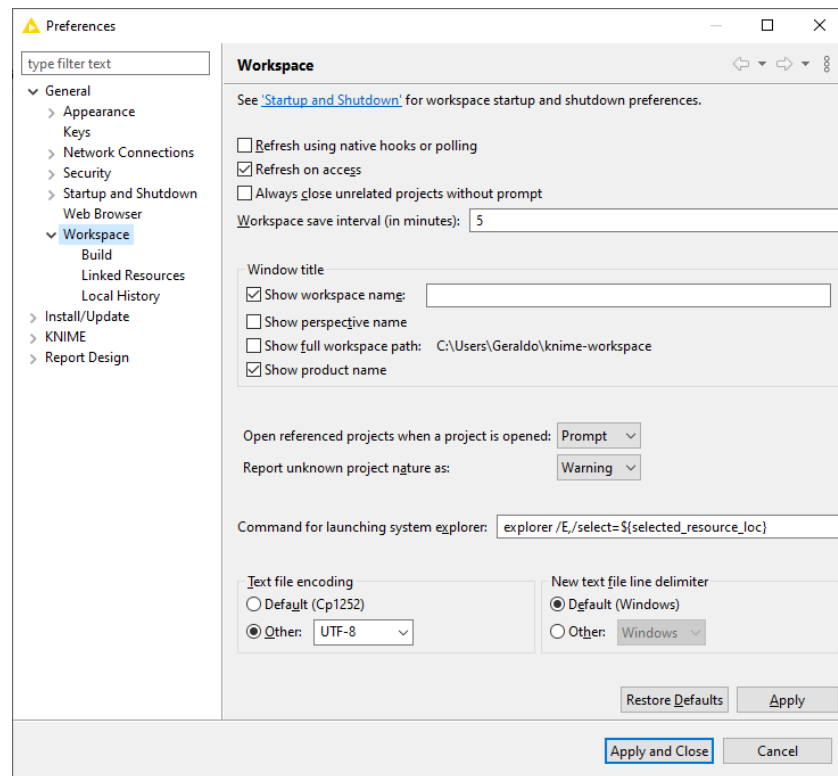


Figura 3.2.: Configurando a codificação default do KNIME.



Figura 3.3.: Um workflow KNIME só com um módulo, FileReader.

3.9. Tratando a codificação no KNIME

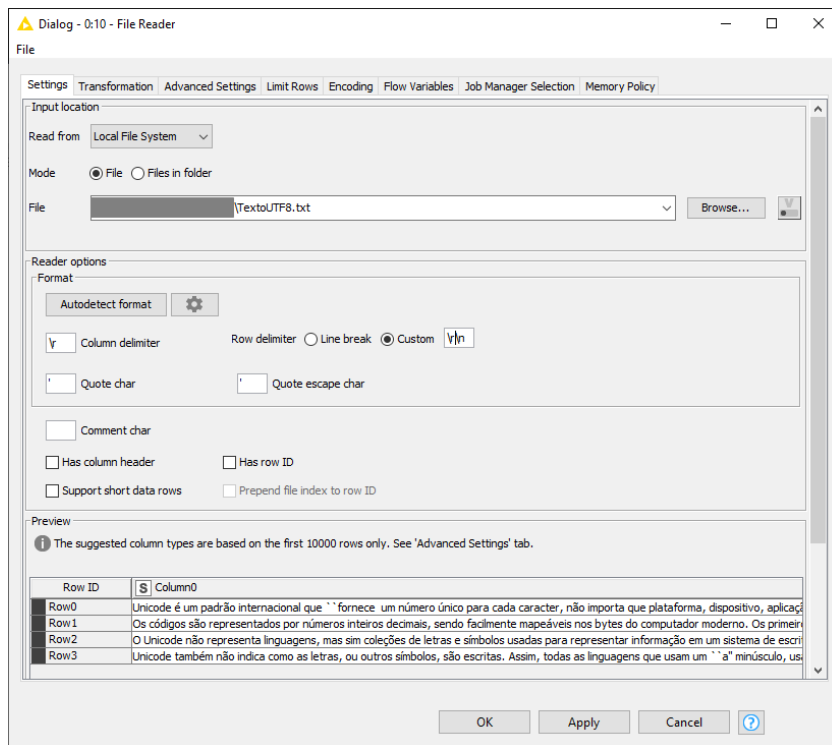


Figura 3.4.: Lendo um arquivo texto como uma sequência de linhas no KNIME.
Módulo File Reader.

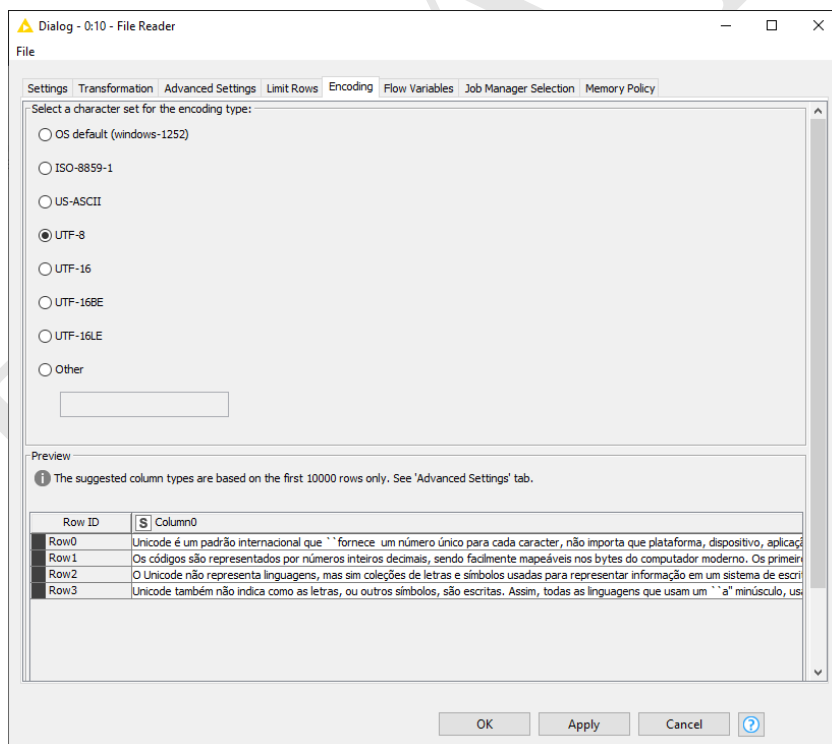


Figura 3.5.: Configurando a codificação correta do arquivo.

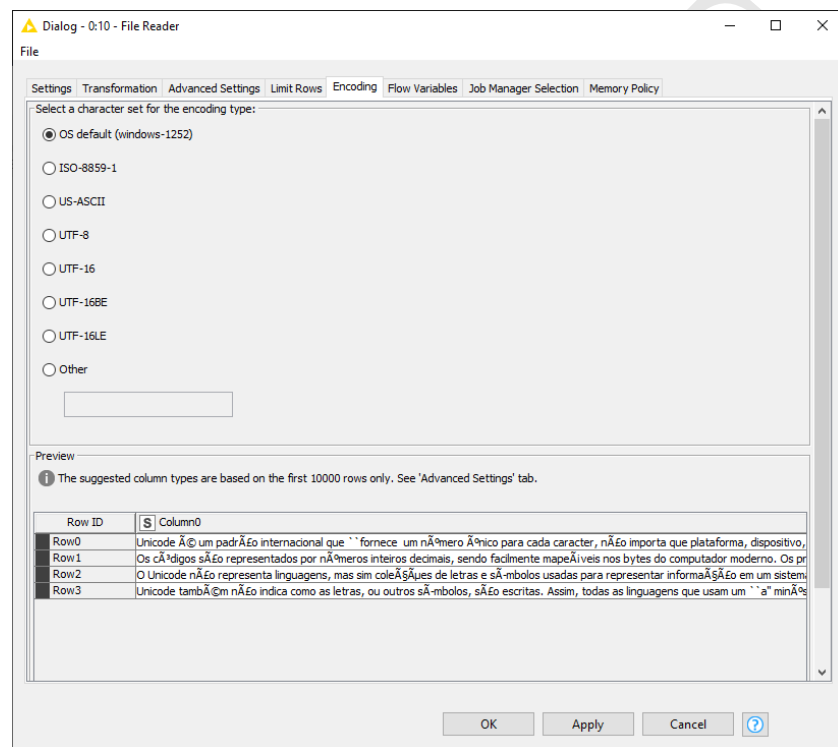


Figura 3.6.: Configurando a codificação errada, a visualização apresenta caracteres estranhos.

3.10. Codificando strings

Em Python, pode ser necessário em algum momento gerar a codificação de uma string, em algum código específico. Isso é feito com a função `byte`, como no Programa 3.8:

Programa 3.8: Obtendo a sequência correta da codificação de uma string

```

1 import locale
2
3 print("Locale é: {}".format(locale.getpreferredencoding()))
4
5 string = "coração brasileiro"
6 print("A string '{}' com tamanho {}".format(string, len(string)))
7
8 encodings = ['utf-8', 'ascii', 'cp1252']
9
10 for enc in encodings:
11     try:
12         sb = bytes(string, enc)
13         print("Encoding: {2} - '{}' - len {}".format(sb, len(sb), enc))
14     except Exception as e:
15         print(str(e)[0:70])

```

A saída desse programa é:

Saída do Programa 3.8

```

1 Locale é: cp1252
2 A string 'coração brasileiro' com tamanho 18
3 Encoding: utf-8 - 'b'cora\xc3\xa7\xc3o brasileiro'' - len 20
4 'ascii' codec can't encode characters in position 4-5: ordinal not in
5 Encoding: cp1252 - 'b'cora\xe7\xe3o brasileiro'' - len 18

```

Também é possível simplesmente escolher a codificação quando se escreve o arquivo, como mostra o Programa 3.9.

Programa 3.9: Salvando um mesmo arquivo em várias codificações.

```

1 import os
2
3 path = 'F:\Github\Dados-Exercicios-Texto\Capitulo Codificacao de
    ↳ Caracteres'
4
5 os.chdir(path)
6
7 f = open("base.txt", "r", encoding="utf-8")
8 text = f.read()
9

```

```
10 encoders = ['utf-32', 'cp1252']
11
12 for i in range(len(encoders)):
13     g = open("texto "+str(i)+".txt", "w", encoding=encoders[i])
14     g.write(text)
15     g.close()
```

3.11. Codificações por referência

Alguns padrões definem como representar em um arquivo símbolos desejados sem que eles estejam na codificação utilizada. Nesse caso, os símbolos que não podem ser representados são representados de acordo com alguma regra, usando apenas símbolos existentes na codificação.

Um exemplo disso são as entidades HTML (*HTML entities*). Elas são definidas de duas formas:

`&entity_name;`

ou

`&entity_decimal_number;`

ou ainda

`ntity_hexadecimal_number;`

As referências numéricas são equivalentes aos *code points* do Unicode.

Por exemplo, em HTML, ` ` identifica um *non-breaking space*, já `´` identifica um “a” com acento agudo, “á”. O símbolo de copyright, “©”, pode ser gerado a partir de `©`, ou, usando a forma numérica, `©`.

O W3C definiu alguns conjuntos de entidades, para HTML, MathML, XML e HTML 5.

Outras linguagens, como \LaTeX apresentam também códigos similares para alguns ou todos os caracteres, como `\copyright` para o “©”.

A vantagem de usar essa forma de representação é que qualquer caractere pode ser representado apenas com os caracteres ASCII ou com um conjunto menor ainda de números.

Ao processar, porém, é necessário tratar esses códigos adicionais de alguma forma. Em Python isso pode ser feito com o pacote `html` e a função `unescape`, como no Programa 3.10.

Programa 3.10: Processando entidades HTML

```

1 import html
2
3 import html
4
5
6 T = ["Teste&copy;\u0141\tá", u"Teste&copy;\u0141\tá",
7      b"Teste&copy;\u0141\t", r"Teste&copy;\u0141\tá"]
8
9 for s in T:
10     try:
11         print(s)
12         print(html.unescape(s))
13     except Exception as e:
14         print("Erro:"+str(e))

```

Cujo resultado é:

	Resultado do Programa 3.10
1 Teste©	á
2 Testeℓ	á
3 Teste©	á
4 Testeℓ	á
5 b'Teste©\\u0141\t'	
6 Erro:a bytes-like object is required, not 'str'	
7 Teste©\u0141\tá	
8 Testeℓ\u0141\tá	

Interpretando o resultado é possível entender o que acontece, sabendo o significado das letras “u”, unicode, “r” raw, e “b”, bytes.

A primeira string, "Teste©\u0141\tá" é comum. Nesse caso ela é processada normalmente como string, o que faz com que tanto o tab quanto o código unicode inserido. Ao ser processada pelo `unescape` do `html`, o faz corretamente. A segunda string é uma string unicode, e funciona exatamente como a normal, pois em Python 3 as string normais são unicode⁹.

A terceira string é uma sequência de bytes na verdade. Ela não aceita, já na compilação, o “á”, que foi retirado. Além disso ela dá erro no processamento `html.unescape`.

A quarta string é do tipo raw, e não aceita “\” como *escape*, porém aceita também o processamento `html.unescape`.

⁹Em Python 2 elas são `bytestring`.

Representação interna de strings em Python 3

Em Python 3 as strings tem uma representação interna múltipla, que tem como objetivo otimizar o funcionamento. Assim, a partir da versão 3.3, a representação interna de uma string depende da necessidade criada pelos caracteres da própria string e pode ser Latin-1 (utf-8), UCS-2 (predecessora do UTF-16, de 2 bytes e de tamanho fixo), ou UCS-4 (predecessora do UTF-32).

A PEP 393 descreve a *Flexible String Representation* para Python (Löwis, 2010).

3.12. Codificação de Binários em Texto

Uma estratégia usada em alguns protocolos de transmissão é a transformação de binários em texto, normalmente ASCII. Isso evita que algumas configurações de 8-bits sejam interpretadas de alguma forma indesejável pelos protocolos, como os caracteres de comando. Isso é muito usado, por exemplo, para a transmissão de imagens, ou qualquer outro tipo de arquivo binário, por meio de correio eletrônico, mesmo havendo extensões 8-bits do protocolo original (Postel, 1982).

Existem vários formatos que seguem esse princípio, e a Wikipedia fornece uma boa lista¹⁰: Ascii85, Base32, Base36, Base45, Base58, Base62, Base64, Base85, BinHex, uuencode, ... Nesse texto vamos exemplificar todos usando o Base64, um dos formatos usado no MIME (Freed e Borenstein, 1996).

3.12.1. Base64

Base64 é um conjunto de esquemas de codificação que permite transmitir dados binários na forma de texto. É especialmente usado no protocolo SMTP (Klensin, 2008; Postel, 1982) de transferência de e-mail, para transportar imagens ou documentos, por meio do MIME (Freed e Borenstein, 1996), o que causa um aumento tamanho nos arquivos. Claramente, não é comum representar texto como Base64, porém documentos completos, por exemplo em PDF, podem estar “escondidos” dentro desse formato.

Em Base64 são usados 64 caracteres ASCII, para indicar 64 grupos de 6 bits. Na implementação MIME de Base64 são usados, nessa ordem e começando por zero, os caracteres A-Z, a-z, 0-9 e os caracteres “+” e “/”. Qualquer necessidade de completar os bits, *padding*, é feita com o caractere “=”. Assim, por exemplo, o ‘A’ é o número 0, o ‘b’ é o número 27, o ‘+’ é o número 62. A Tabela 3.3 exemplifica uma codificação da palavra “Man” em ASCII para Base64, cujo resultado seria “TWFu”. Como Base64 codifica 8 bits em 6 bits, cada 3 bytes codificados gera exatamente 4 caracteres na string.

¹⁰https://en.wikipedia.org/wiki/Binary-to-text_encoding

Tabela 3.3.: Exemplo de como é feita uma codificação Base64

Entrada	M								a								n							
ASCII	0x4d								0x61								0x6e							
bits	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Sexteto	19								22								5							
Saída	T								W								F							

3.13. Exercícios

Atenção: os arquivos a serem usados para os exercícios desse capítulo podem ser encontrados em: <https://github.com/LINE-PESC/Dados-Exercicios-Texto>

Exercício 3.1:

No diretório “Capitulo Codificacao de Caracteres” estão disponíveis alguns arquivos “.txt”, criados a partir do arquivo “base.txt”, porém cada um com uma codificação. Analise esses arquivos visualmente e tente adivinhar que formato usam. É possível usar editores que mostram os bytes, como o disponível em <https://hexed.it/>. Depois use o Python e tente detectar automaticamente os formatos com o módulo `chardet`.

Exercício 3.2:

Analise os arquivos “*.txt” do diretório “Capitulo Codificacao de Caracteres” e obtenha quantos bytes possuem e quantos caracteres mostram, se visualizados com um editor de texto. Leia os arquivos no formato correto.

Exercício 3.3:

Use o formato MARC 21 para registrar o livro “Abbott, Edwin (1991). Flatland: A Romance of Many Dimensions. 2ª ed. Princeton University Press”. Crie o record MARC 21 duas vezes, uma considerando o formato MARC-8 e outra o formato UCS/Unicode. Salve ambos em arquivos e leia os arquivos com um editor de bytes, como o disponível em <https://hexed.it/>. Qual diferença entre eles? Imprima também, com e sem codificação para utf-8. O que significam os caracteres especiais que aparecem no registro?

DRAFT

Arquivos Textuais

Nesse capítulo é feita uma introdução aos formatos de arquivos que encontramos e que devem ser processados para que possam ser realizadas tarefas de indexação, busca e aprendizado de máquina com textos.

Muitos desses formatos não podem ser lidos diretamente por seres humanos, ou podem ser lidos apenas em certas condições, enquanto outros são de fáceis compreensão.

O objetivo de conhecer os formatos é compreender os métodos de transformar o conteúdo desses arquivos em arquivos simples de texto, que serão usados nas tarefas de processamento de texto.

4.1. Os formatos mais comuns

Os formatos de arquivo são criados com diferentes motivações, o que leva a uma série de requisitos, e restrições, e também por escolhas pessoais ou institucionais. Assim, enquanto o formato em que são guardados os arquivos fonte de L^AT_EX deve poder ser lido por seres humanos, o formato básico resultante do seu processamento se destina a ser processado por programas de computador e é de difícil leitura para seres humanos.

A maioria do processamento que é feito hoje em textos usa poucos tipos de arquivos: arquivos de texto simples, arquivos no formato do Word, e arquivos do tipo PDF, HTML, XML ou JSON.

Esses formatos podem ser divididos, de forma geral, em três tipos: formatos de edição, formatos de marcação ou formatos de impressão.

Os formatos de edição podem ser editados diretamente e não possuem nenhuma marcação. Normalmente programas de computador são guardados nesse formato. Eles estão sempre codificados de alguma forma, como visto no Capítulo 3.

Os formatos de marcação ainda podem ser editados diretamente, apresentando um arquivo legível para o ser humano. Editores de texto podem esconder a marcação, usando uma apresentação WYSIWYG, mas sempre vão guardá-los de forma que possam ser abertos por um editor comum.

Formatos de edição e marcação são criadas para atender simultaneamente o homem e a máquina, podendo variar entre formatos facilmente interpretáveis por humanos, até alguns que são de difícil interpretação.

Finalmente, formatos de impressão são feitos para indicar como um texto deve ser impresso ou visualizado. Um dos formatos de impressão mais conhecidos, PostScript, é na verdade uma linguagem de programação que pode ser entendida por um especialista.

Se um arquivo em um formato, ou codificação, for lido com a suposição que foi escrito em outro formato, erros aparecerão.

Além disso, vamos tratar de alguns formatos que podem ser encontrados não só em arquivos, mas também em *streams* de dados. O processamento, porém, é semelhante.

4.2. Formatos de Edição

Pouco é preciso falar de formatos de edição que não tenha sido tratado no Capítulo 3, pois normalmente a única maneira deles causarem algum problema é serem lidos de acordo com uma codificação diferente da planejada.

Arquivos desse tipo normalmente são chamados de arquivos texto e têm como característica usarem a extensão “.txt”, mas isso não é uma regra determinante. Outros formatos, com mais estrutura, como um *Comma Separated File* ou “.cvs” podem ser tratados, algumas vezes, como um arquivo texto simples.

Muitos sistemas e experimentos têm como primeiro passo do processamento de um arquivo textual a sua transformação em um formato simples de texto.

4.3. Linguagens de Marcação

Muitos arquivos textuais são construídos por meio de linguagens de marcação. Em uma linguagem de marcação, caracteres são usados tanto para o conteúdo como para definir, de forma imperativa ou declarativa, o que deve ser feito por um processador específico, como um editor de texto ou uma impressora.

A marcação não é parte do texto, mas está inserida nele. Assim, em L^AT_EX, por exemplo, escrevemos

```
\textbf{Negrito}
```

quando queremos que o texto “Negrito” esteja em negrito, como em “**negrito**”.

Entre as linguagens de marcação mais utilizadas estão: HTML, XML, SGML, \LaTeX , RTF, Open XML Document (formato usado no Microsoft Word), OpenDocument (.odt).

Stand-off markup

Uma alternativa ao uso de linguagens de marcação seria usar o que é conhecido como “*stand-off markup*”, ou “*stand-off annotation*”: anotações separadas que fazem referência ao texto original. Por exemplo, um arquivo adicional dizendo onde começa o negrito, por exemplo no caracter 12302 do arquivo, e onde acaba, no caracter 12308, no caso.

A ideia foi defendida por Nelson (1997), e tem uma versão descrita por Thompson e McKelvie (1997), porém não é difundida atualmente.

As linguagens de marcação também podem ser divididas em (Coombs, Renear e DeRose, 1987)¹:

- Descritivas², como HTML, SGML, XML;
- Procedurais, como \TeX , PostScript ou Lout, e
- De Apresentação, como Wikis.

4.3.1. HTML

HyperText Markup Language foi um formato originalmente criado por Tim Berners-Lee, baseado em SGML, para servir de linguagem de marcação para o World Wide Web (WWW). Junto com o protocolo HTTP ele formou uma das tecnologias de maior sucesso na história.

Apesar de hoje HTML possuir vários tipos de marcação, inicialmente possuía poucos rótulos, que permitiam descrever efeitos desejados na apresentação do texto e a criação de ligações de hipertexto via rede. Ao longo do tempo os rótulos foram se avolumando e novas características apareceram, como a possibilidade de executar programas dentro de uma página HTML, em JavaScript (Mozilla, 2021), ou de criar estilos variados e possivelmente dinâmicos, com CSS (W3C, 2021b).

Não é fácil processar corretamente HTML, porque os arquivos desse formato são criados com muita liberdade e é comum que possuam erros. Na prática, os principais processadores de HTML, os navegadores WWW, perdoam muitos erros.

Porém, normalmente a única necessidade, ou a necessidade principal quando realizando atividades de indexação ou aprendizado de máquina é limpar o HTML, ficando apenas com o texto.

¹Os autores também citam um tipo *Puntuational*, referente a marcações sintáticas que dão informações sobre enunciados escritos, como espaços, e que não é preciso considerar no momento.

²Provavelmente o nome mais adequado seria declarativas

Os padrões da Internet/WWW

Os padrões da Internet são estabelecidos atualmente pela Internet Engineering Task Force, sendo denominados RFC, de *Request for Comments*, e numerados sequencialmente (IETF, 2021).

A maioria dos padrões relacionados a Web, ou ao WWW, é estabelecida pelo *World Wide Web Consortium* (W3C) W3C, “an international community where member organizations, a full-time staff, and the public work together to develop Web standards. Led by Web inventor and Director Tim Berners-Lee and CEO Jeffrey Jaffe, W3C’s mission is to lead the Web to its full potential.” (W3C, 2021a).

Um arquivo HTML moderno, e muito simples, pode ter a seguinte aparência:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Título da Página</title>
5 </head>
6 <body>
7
8 <h1>Cabeçalho</h1>
9 <p>Um parágrafo simples</p>
10
11 Usando entidades HTML &copy;
12
13 </body>
14 </html>
```

Sua visualização seria a da Figura 4.1

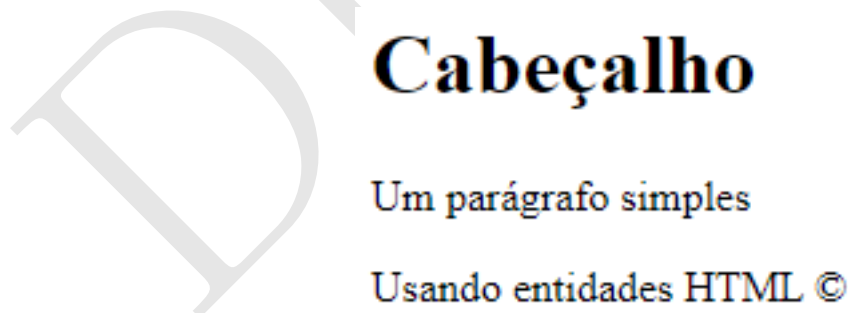


Figura 4.1.: A visualização de um arquivo HTML muito simples.

A questão em HTML é que palavras escolher, isto é, como retirar o texto desejado do arquivo. Em primeiro lugar existem as *tags*, cujo seu texto pode desprezado, isto é,

não queremos processar algo como TITLE. Além disso, existem as entidades HTML, que podem ser interessantes, ou não, e normalmente têm que ser traduzidas, nesse caso um `´`, por exemplo, deve virar um `'á'` ou um `'a'`, de acordo com o que é necessário para o sistema.

Por exemplo, no arquivo HTML acima, as partes que devem ser trabalhadas incluem a palavra “Cabeçalho” e a frase “Um parágrafo simples”, mas o que fazer com “Título da Página”? Devemos considerá-lo ou não como parte do documento para indexação, aprendizado de máquina, ou ainda outra aplicação?

Uma página web geralmente possui um HTML complexo, gerado por um conjunto de programas e manipulação humana. A Figura 4.2 mostra um um fragmento do HTML da página pessoal do autor, gerado por meio do WordPress.

```
.amaz_td {background:#ddddd;}
</style>
<meta name="description" content="This is the personal site of Geraldo Xexéo."><meta
name="Keywords" content="Computing, Games, Software Engineering, Information Retrieval"><!--
personal --><meta name="Generator" content=""><!-- BOF: ./personal-templates/simple/themes.show --
><!-- EOF: ./personal-templates/simple/themes.show --><!--<BASE HREF="http://162.215.248.21/
index.html">-->
<title></title>
</head>
<body onload="" data-gr-c-s-loaded="true"><!-- BOF: layouts/personal/l15/new/splash.html -->
<table border="0" cellpadding="0" cellspacing="0" class="t_height" dir="LTR" style="background:
url() width="100%">
  <tbody>
    <tr>
      <td height="101" valign="top">
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
          <tbody>
            <tr>
              <td align="left" width="492"></td>
              <td align="right" height="79" style="background-image:url(widgets/gen_90.1.gif)"
                valign="middle" width="100%"></td>
            </tr>
          </tbody>
        </table>
      </td>
    </tr>
  </tbody>
</table>
```

Figura 4.2.: Imagem parcial do HTML usado em uma página pessoal

4.4. HTML 5

HTML 5 complica as coisas um pouco, porque é uma junção de HTML, Javascript, CSS e XML. Isso faz com que não só que seja mais difícil fazer o *parsing* do arquivo, como também implica em decidir o que deve e o que não deve ser aceito como texto, de acordo com necessidades específicas.

Parte da mudança da mentalidade da Web ao longo do tempo foi que antes o navegador só era responsável por tarefas básicas de visualização. Hoje é possível colocar partes móveis de texto e mexer dinamicamente na estrutura da página, sendo na prática um ambiente de execução. A Figura 4.3 mostra a evolução de HTML ao longo do tempo até a versão inicial do HTML 5 (WHATWG, 2021), com as principais influências para o desenvolvedor, e que continua recebendo adições, principalmente no CSS (W3C, 2021b).

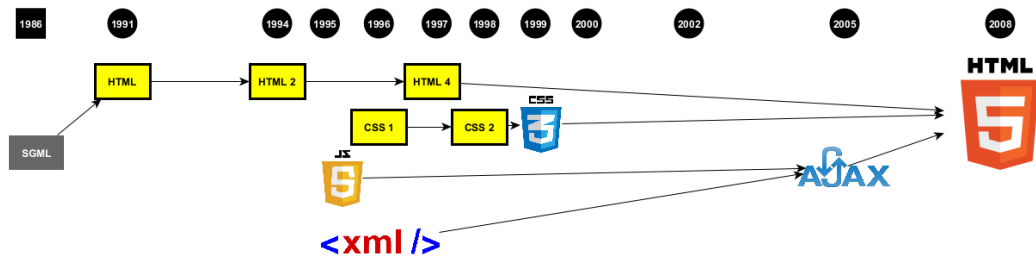


Figura 4.3.: Linha do tempo do HTML ao HTML 5.

Já a Figura 4.4 dá uma visão geral da plataforma Web (WHATWG, 2021).

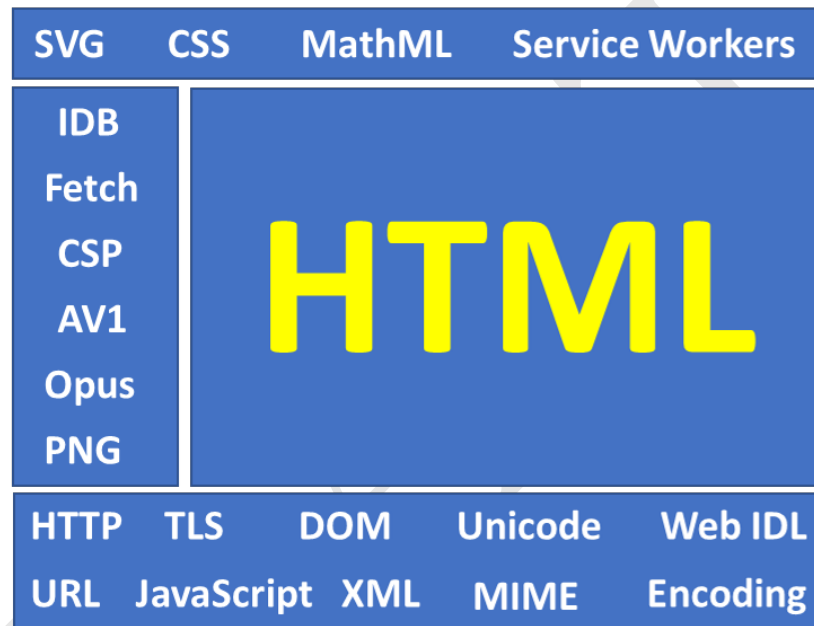


Figura 4.4.: Visão geral da plataforma Web, segundo WHATWG (2021)

4.4.1. Limpando HTML de arquivos com Python

Para se obter o texto que está contido em um arquivo HTML, qualquer que seja sua versão, é necessário então limpar, ou filtrar, a linguagem de marcação do arquivo.

Existem várias opções para fazer essa limpeza. `lxml` é um pacote Python para tratar XML e HTML. Com ele é possível eliminar toda a marcação com facilidade, como no Programa 4.1, e obter como resultado uma string contendo o texto.

Programa 4.1: Programa simples para obter todo o texto dentro de um arquivo HTML

```
1 import lxml
```

```

2
3 f = open('processahtmlsimplesexemplo.html', 'r', encoding='utf-8')
4 origin = f.read()
5
6 html = lxml.html.fromstring(origin)
7 text = str(html.text_content())
8
9 print(text.strip())

```

A resposta, apresentada a seguir, remove os rótulos HTML e trata as entidades:

Saída do Programa 4.1

```

1 Título da Página
2
3
4
5 Cabeçalho
6 Um parágrafo simples
7
8 Usando entidades HTML 1

```

4.4.2. Limpando HTML de arquivos com KNIME

Usando as extensões é possível desenhar um workflow KNIME que lê um arquivo indicado por uma URL e depois remove todo o markup dele, como é demonstrado na sequência de sequência de Figuras 4.5, 4.6, 4.7, 4.8, 4.9.

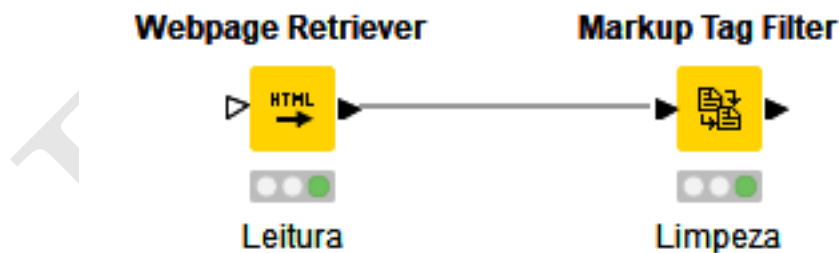


Figura 4.5.: Workflow para ler, via URL, e limpar um arquivo HTML

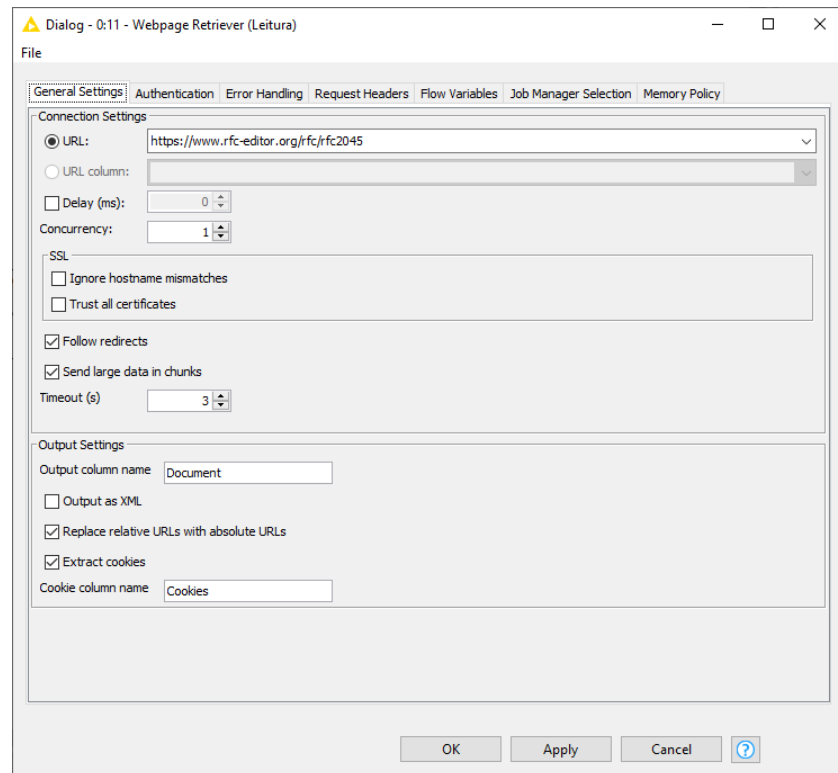


Figura 4.6.: Recuperando uma página via sua URL.

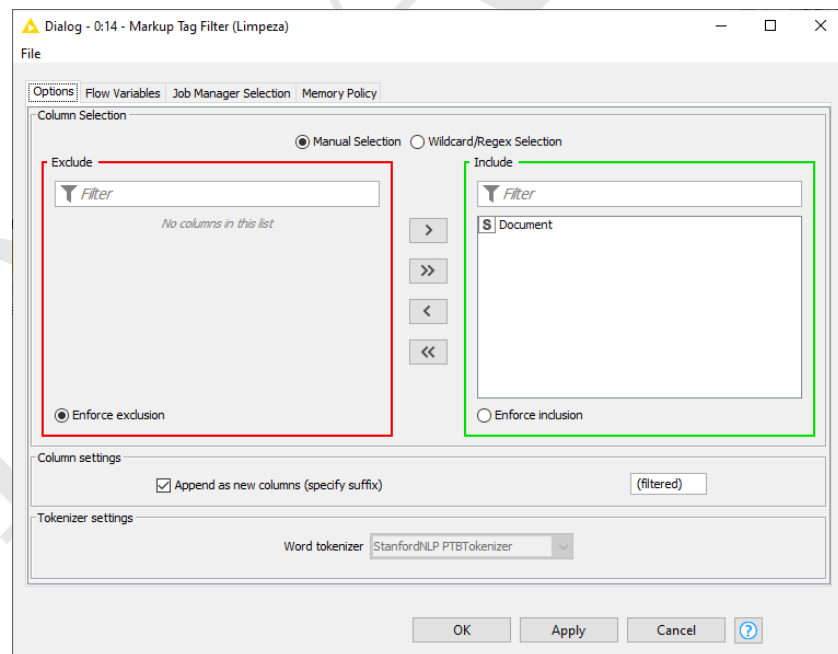
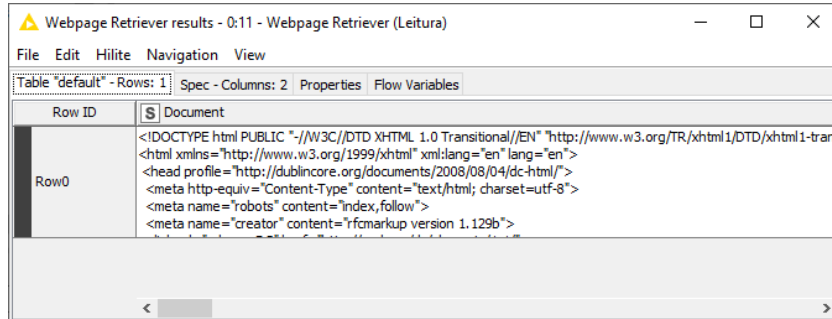


Figura 4.7.: Configurando a remoção de tags de markup



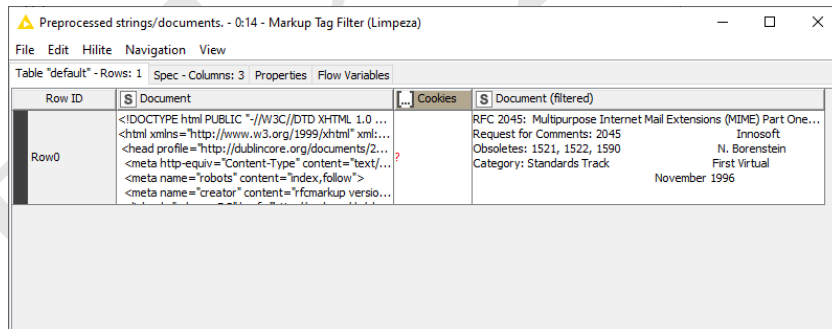
Webpage Retriever results - 0:11 - Webpage Retriever (Leitura)

File Edit Hilite Navigation View

Table "default" - Rows: 1 Spec - Columns: 2 Properties Flow Variables

Row ID	Document
Row0	<pre><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-trans <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"> <head profile="http://dublincore.org/documents/2008/08/04/dc-html/"> <meta http-equiv="Content-Type" content="text/html; charset=utf-8"> <meta name="robots" content="index,follow"> <meta name="creator" content="rfcmarkup version 1.129b"></pre>

Figura 4.8.: Resultado recuperado pelo nó de Leitura.



Preprocessed strings/documents. - 0:14 - Markup Tag Filter (Limpeza)

File Edit Hilite Navigation View

Table "default" - Rows: 1 Spec - Columns: 3 Properties Flow Variables

Row ID	Document	Cookies	Document (filtered)
Row0	<pre><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 ... <html xmlns="http://www.w3.org/1999/xhtml" xml:... <head profile="http://dublincore.org/documents/2... <meta http-equiv="Content-Type" content="text/... <meta name="robots" content="index,follow"> <meta name="creator" content="rfcmarkup versio...</pre>	<pre>RF 2045: Multipurpose Internet Mail Extensions (MIME) Part One... Request for Comments: 2045 Obsoletes: 1521, 1522, 1590 Category: Standards Track</pre>	<pre>Innosoft N. Borenstein First Virtual November 1996</pre>

Figura 4.9.: Resultado pós remoção do markup.

4.5. Outros Formatos Específicos

Ao longo do tempo, provavelmente milhares de formatos foram desenvolvidos contendo texto, alguns deles certamente perdidos. Por exemplo, durante os anos 1980 se utilizava no Brasil o processador de texto Carta Certa, que saiu de linha em meados da década de 90. Já em torno de 2010, em um projeto que participei, foi impossível ler um arquivo em seu formato.

Mais informação sobre vários formatos de arquivo pode ser obtida em <https://docs.fileformat.com/word-processing/> e outros sites da internet.

4.6. Formatos de Impressão

Formatos de impressão são arquivos utilizados primariamente para impressão, mesmo que possam ser compreendidos por seres humanos. A Adobe chama essas linguagens de Linguagem de Descrição de Página (Adobe Systems Inc., 1999), e as divide entre linguagens de formato estático e de formato dinâmico.

A principal linguagem de impressão é PostScript, porém, hoje em dia, o principal formato usado não é o PostScript (PS, arquivos .ps), mas sim o Page Description Format (PDF, arquivos .pdf), que utiliza um subset de PostScript para descrever páginas, dentro de um esquema de arquivo específico, composto de objetos.

4.6.1. PostScript

PostScript é uma linguagem de programação interpretada baseada em pilha com capacidades gráficas, criada com a finalidade de “descrever a aparência de textos, formas gráficas e imagens em páginas impressas ou mostradas em um monitor” (Adobe Systems Inc., 1999) em dispositivos de saída do tipo *raster*. Ele é uma linguagem de formato dinâmica, Turing completa (Adobe Systems Inc., 1999).

Um exemplo que imprime “Hello World!” em PostScript seria.

```
/Helvetica-Bold findfont 26 scalefont setfont
20 40 moveto
.5 setgray
(Hello World!) show
```

4.6.2. PDF

Portable Document Format (ou **PDF**) é um formato de arquivo, desenvolvido pela Adobe Systems em 1993, para representar documentos de maneira independente do

aplicativo, hardware, e sistema operacional usados para criá-los. Um arquivo PDF pode descrever documentos que contenham texto, gráficos e imagens num formato independente de dispositivo e resolução.

4.6.3. Extraindo texto do PDF em Python

Vários pacotes ajudam a extrair o texto de um arquivo PDF, como o `pdfminer.six`. O Programa 4.2 mostra uma forma simples de obter todo o texto de um arquivo pdf. Já o Programa 4.3 mostra o uso de funções específicas, dando maior controle ao programador.

Programa 4.2: Usando o `pdfminer.six` para extrair texto de um arquivo PDF

```

1 import pdfminer.high_level as pdfh
2
3 text = pdfh.extract_text("arquivo.pdf")
4
5 print(len(text))

```

Programa 4.3: Usando uma forma mais complexa de extrair texto de um arquivo PDF

```

1 from io import StringIO
2
3 from pdfminer.converter import TextConverter
4 from pdfminer.layout import LAParams
5 from pdfminer.pdfdocument import PDFDocument
6 from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
7 from pdfminer.pdfpage import PDFPage
8 from pdfminer.pdfparser import PDFParser
9
10 output_string = StringIO()
11 with open("arquivo.pdf", 'rb') as in_file:
12     parser = PDFParser(in_file)
13     doc = PDFDocument(parser)
14     rsrcmgr = PDFResourceManager()
15     device = TextConverter(rsrcmgr, output_string, laparams=LAParams())
16     interpreter = PDFPageInterpreter(rsrcmgr, device)
17     for page in PDFPage.create_pages(doc):
18         interpreter.process_page(page)
19
20 print(output_string.getvalue())

```

4.6.4. Extraindo texto do PDF em KNIME

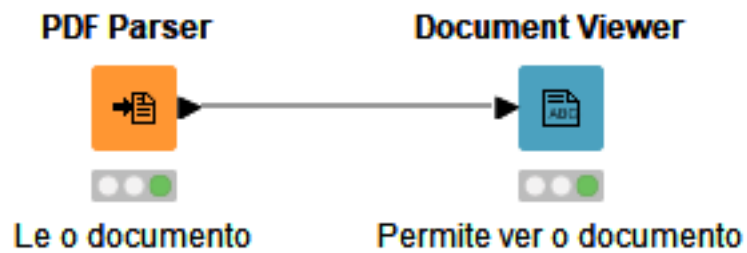


Figura 4.10.: Workflow para ler e transformar um arquivo PDF em um Document do KNIME

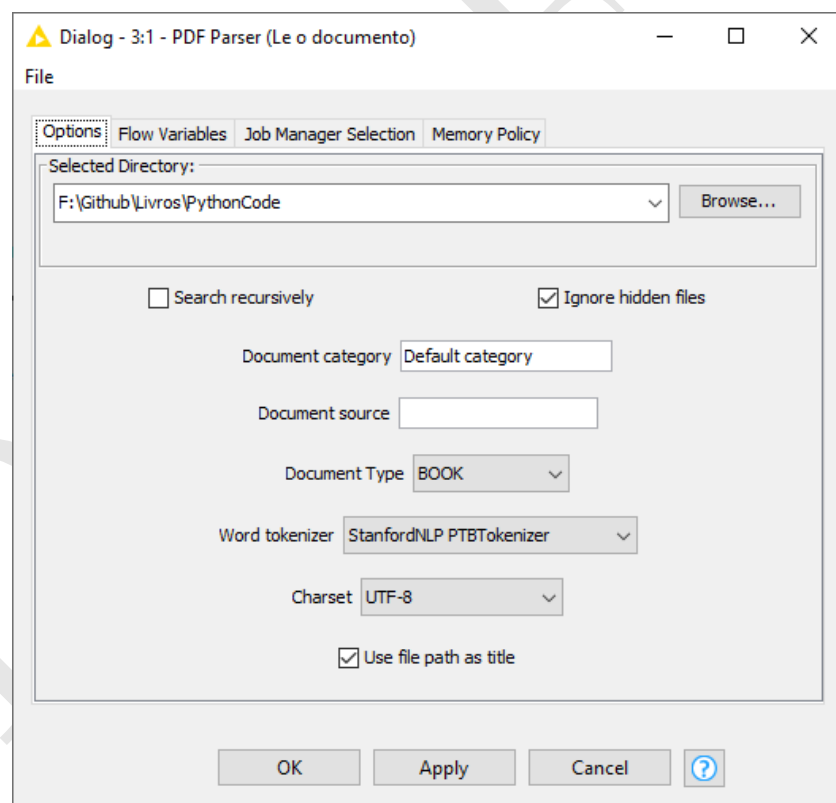


Figura 4.11.: Configuração do nós PDF Parser

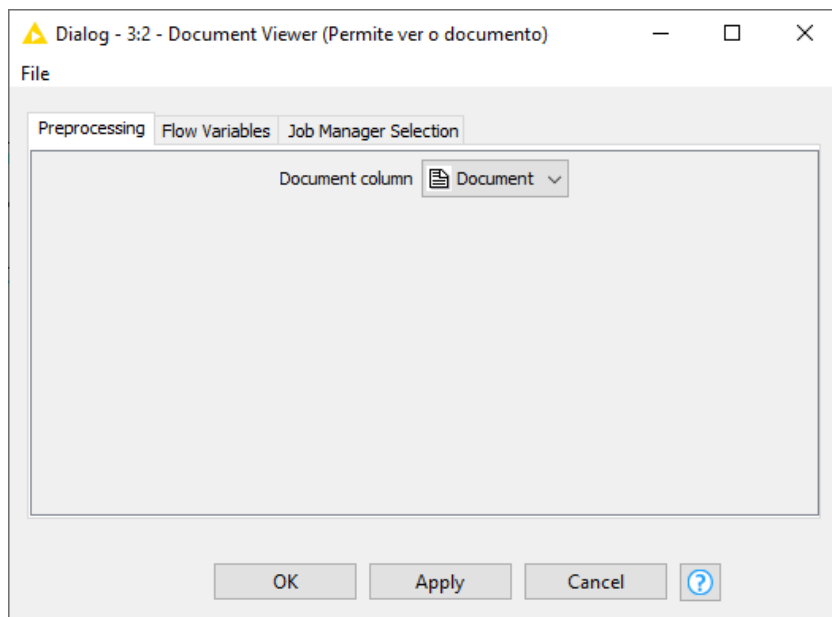


Figura 4.12.: Configuração do nó Document Viewer

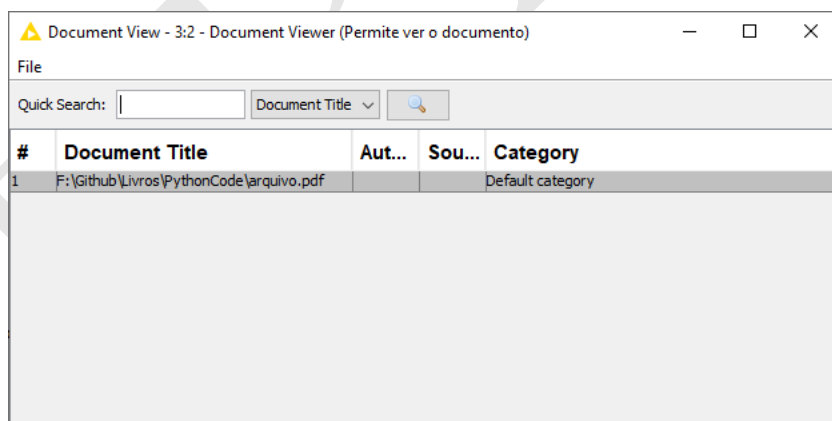


Figura 4.13.: Vendo a lista de documentos lidos

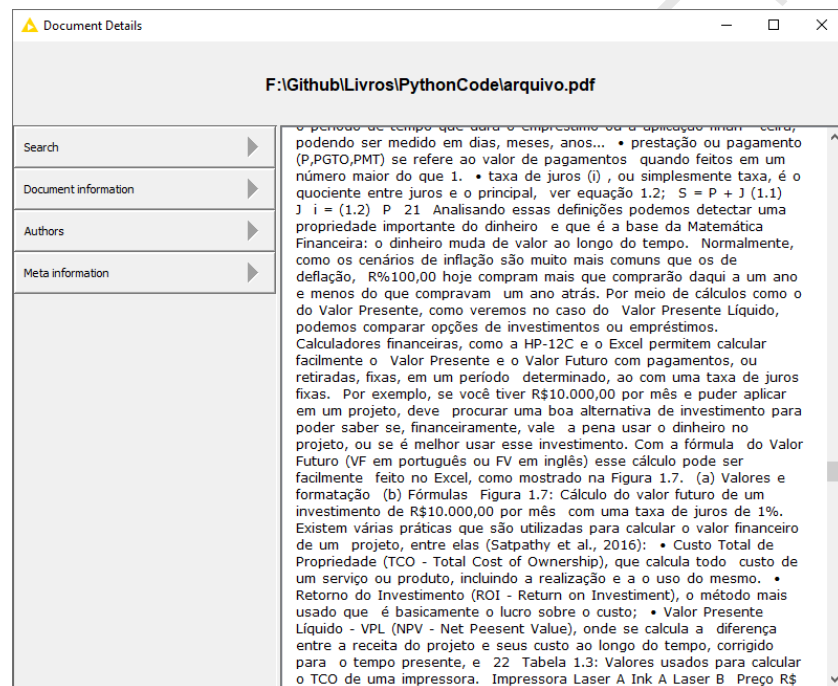


Figura 4.14.: Vendo o detalhe de um documento, principalmente seu conteúdo textual.

4.7. XML

XML, de *eXtensible Markup Language*³ (W3C, 2008), é um formato projetado para armazenar e transferir conteúdo, de forma legível para humanos, mas não necessariamente fácil de entender. Muitas coleções de documentos usadas em experimentos, como vários Corpora, são fornecidos em XML.

É interessante notar que o XML usado atualmente é o XML 1.0 (W3C, 2008), cuja versão mais nova é mais recente que a versão mais nova do XML 1.1 (W3C, 2006).

A linguagem possui um conjunto de *tags* fixas, mas em cada arquivo a grande parte delas é definida arbitrariamente pelo criador, sendo algumas vezes descritas em esquemas. A Figura 4.15 mostra um arquivo desse tipo.

```
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>
      Two of our famous Belgian Waffles with plenty of real maple syrup
    </description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>
      Light Belgian waffles covered with strawberries and whipped cream
    </description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>
      Belgian waffles covered with assorted fresh berries and whipped cream
    </description>
    <calories>900</calories>
  </food>
```

Figura 4.15.: Exemplo de fragmento de arquivo xml.

Um dos problemas de tratar XML é que os dados podem aparecer tanto dentro das *tags* como na forma de texto livre entre o início e o fim de uma marcação.

Um arquivo XML é considerado bem-formatado quando segue as regras de sintaxe da linguagem, e válido quando segue as regras estabelecidas em um esquema, isto é, é validado frente a um esquema. Esse esquema pode ser definido em *XML Schema Definition*, um arquivo XSD (W3C, 2006, 2012), ou em um formato mais antigo, o

³A pronúncia em inglês é ex-em-el.

Document Type Definition, o DTD. XSD usa a própria XML para definir o esquema de outro arquivo XML, enquanto DTD usa um formato diferente.

Por exemplo, um arquivo XML, como o apresentado no Programa 4.4, pode mostrar uma mensagem. Ele segue as regras sintáticas de XML:

- tem um elemento raiz (mensagem);
- todos os elementos tem um *tag* que os fecha (com a barra no início).
- os *tags* são diferenciadas maiúsculas de minúsculas (apenas subtendido no exemplo).
- os elementos estão hierarquizados em árvore corretamente, e
- os valores de atributo estão entre aspas.

Programa 4.4: Exemplo de arquivo XML usando um DFD.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mensagem SYSTEM "Mensagem.dtd">
3 <mensagem>
4 <para>Alice</para>
5 <de>Bob</de>
6 <titulo>Lembrete</titulo>
7 <corpo>Não esqueça a sua promessa!</corpo>
8 </mensagem>
```

Como segue as regras da sintaxe, o Programa 4.4 é um arquivo bem formado. Mas será ele válido? Para ser válido, deve usar, da forma correta, apenas os elementos descritos no DTD a que ele se refere, como o da Programa 4.5.

Programa 4.5: Exemplo de arquivo DTD.

```
1 <!DOCTYPE mensagem
2 [
3 <!ELEMENT mensagem (para,de,titulo,corpo)>
4 <!ELEMENT para (#PCDATA)>
5 <!ELEMENT de (#PCDATA)>
6 <!ELEMENT titulo (#PCDATA)>
7 <!ELEMENT corpo (#PCDATA)>
8 ]
```

É possível notar, no Programa 4.5, que não é permitida a repetição de qualquer elemento. Outra maneira, mais moderna, de descrever as mesmas regras que estão nesse arquivo é o Programa 4.6.

Programa 4.6: Exemplo de arquivo XSD.

```
1 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
  ↳ qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
2 <xs:element name="mensagem">
3 <xs:complexType>
```

```

4 <xs:sequence>
5 <xs:element type="xs:string" name="para"/>
6 <xs:element type="xs:string" name="de"/>
7 <xs:element type="xs:string" name="titulo"/>
8 <xs:element type="xs:string" name="corpo"/>
9 </xs:sequence>
10 </xs:complexType>
11 </xs:element>
12 </xs:schema>

```

4.8. Programas Exemplo

Programa 4.7: Funções de apoio, arquivo birtutils

```

1 import glob
2
3 def fix_path(path):
4     if path[-1] != "/":
5         return path + "/"
6     else:
7         return path
8
9 def get_all_files(path, extension, recursive=False):
10     if recursive:
11         return glob.glob(fix_path(path)+"**/*."+extension, recursive=True)
12     else:
13         return glob.glob(fix_path(path)+"*."+extension)

```

Programa 4.8: Lendo arquivos do corpus Folha, resulta em um dicionário com id dos artigos e indicando o texto sem tratamento.

```

1 from bs4 import BeautifulSoup
2 import birtutils as bu
3
4
5 def grab_folha_docs(file):
6     all_data = {}
7     with open(file, "r", encoding="utf_8") as f:
8         text = f.read()
9         data_as_xml = BeautifulSoup(text, "xml")
10        the_docs = data_as_xml.find_all('DOC')
11        for d in the_docs:
12            all_data[d.find('DOCNO').text] = d.find("TEXT").text

```

```
13     return all_data
14
15
16 if __name__ == "__main__":
17
18     path = '<path to folder>'
19
20     print(path)
21     path = bu.fix_path(path)
22
23     print(path)
24
25     all_data = {}
26     for fn in bu.get_all_files(path,"sgml",recursive=True):
27         all_data |= grab_folha_docs(fn)
```

Parte II.

**Pré-Processamento E Algoritmos
Básicos**


DRAFT

Pré-Processamento

Após o processamento do arquivo original, um texto está representado como um arquivo simples, que pode ser lido por um humano, contendo caracteres codificados em UTF-8, possivelmente contendo marcações ou outros artefatos característicos de um ou outro formato. Nesse caso duas atividades básicas devem ser feitas: a limpeza do texto para obter apenas uma sequência válida de palavras que queremos processar, ou seja, a conversão em texto puro, e a alteração desse texto puro para uma coleção de termos, de forma a melhorar o funcionamento do sistema sendo construído.

Tarefas adicionais podem incluir entender a organização das palavras ou re-organizá-las de alguma forma, por exemplo, quando o arquivo original as posiciona fora da sequência em que aparece no arquivo (como é possível fazer, por exemplo, com arquivos PostScript).

A lista de tarefas que compõe o pré-processamento de texto que podem ser necessárias é grande e inclui:

- eliminação de símbolos indesejados no processamento, como TABs, CRs e LF's;
- eliminação de comandos ou marcações que não pertencem ao texto, como marcações HTML
- eliminação ou tradução de códigos que representam imagens, como , que não pertencem a língua sendo processada, ou que, em sequência, possuem um significado específico, como “:-)” para um rosto sorrido.
- correção de erros de digitação ou de ortografia
- eliminação de acentos e variações aplicadas às letras, como diacríticos, ou criação de representações alternativas, de modo a atender erros ;
- transformação para minúsculas ou maiúsculas, como desejado;
- tokenização, i.e., separação do documento em tokens, ou palavras;
- eliminação das conjunções verbais;
- eliminação das variações de gênero, número, etc;
- lematização ou radicalização (*stemming*).

5.1. Noções de Morfologia da Palavra

Palavras recebem, a partir de um elemento central, flexões que as caracterizam em categorias gramaticais. Nas línguas em geral existem muitas formas de flexões, nem todas existentes em português:

- Modo indicativo, subjuntivo, imperativo, interrogativo, injuntivo, optativo, potencial, jussivo, cohortativo, deôntico (comissivo, diretivo, volitivo), etc.;
- Tempo presente, pretérito, futuro, não pretérito, não-futuro;
- Voz ativa, passiva, reflexiva, média;
- Aspecto perfeito, imperfeito, mais que perfeito, contínuo, progressivo, habitual;
- Caso dativo, locativo, subjetivo, nominativo, acusativo, genitivo, objetivo, possessivo;
- Pessoa primeira, segunda, terceira;
- Número singular, plural, dual, trial, quadral, paucal menor, paucal maior;
- Gênero - masculino, feminino, neutro;
- Grau (adjetivo) - comparativo, superlativo, e
- Grau (substantivo) aumentativo, diminutivo.

Essas flexões existem como conjugações, característica dos verbos, e as declinações, características dos substantivos, adjetivos e pronomes. Essas partes são elementos mórficos ou morfemas.

Assim, um **morfema** é um elemento constituinte da palavra, que podem ser (Lucca e Graças Volpe Nunes, 2002):

- **raiz**, o elemento mais simples a que a palavra pode ser reduzida, obtido quando todos os outros elementos são eliminados, por exemplo “bann” para “abandonar”;
- **tema**, “contituído pelo radical mais uma vogal temática à qual são acrescentadas as desinências casual para substantivos e adjetivos, e verbal para os verbos” (Lucca e Graças Volpe Nunes, 2002);
 - **radical**, o elemento que dá o significado básico da palavra, como “pedr” para “pedr”, “pedrinha”, “pedregulho” e “apedrejar”;
 - **vogal temática**, um elemento somado ao tema para que ele se junto com outros morfemas, podendo ser nominais ou verbais;
- **afixos**, emementos que são agregados a uma raiz ou radiale que alteram o sentido da palavra;
 - **prefixos**, os afixos agregados no início da palavra;
 - **sufixos**, os afixos agregados no fim das palavras;
- **desinência**, elementos que indicam as flexões da palavra;
 - **desinência nominais**, indicam gênero e número (em português);
 - **desinência verbal**, indicam tempo, modo, número, pessoa e formal nominal do verbo.

5.1.1. Lematização e Stemming

Um **lema**, ou **forma canônica**, é a forma dicionarizada de uma palavra. No português o lema é o singular masculino, para substantivos e adjetivos, e o infinitivo para os verbos.

Já o *stem* é o lema ou o radical de uma palavra, assim, um *stemmer* deveria ser um algoritmo que, dada uma palavra, fornece o lema ou o radical da mesma (Lucca e Graças Volpe Nunes, 2002). O *stemmer* mais conhecido, porém, é o **Porter Stemmer** (Porter, 1980), que se autodenomina um “removedor de sufixos”. *Stemmers* normalmente fazem simplificações ou premissas práticas sobre o processo, e, por causa disso, podem agregar palavras que no dicionário não estariam agregadas, como “casa”, “casar” e “casamento”. Supõe-se que um lematizador siga estritamente as regras da gramática.

Lematizar, então, é representar uma palavra pelo seu lema. Essa atividade existe no contexto lexicográfico (Lucca e Graças Volpe Nunes, 2002), como na criação de dicionários, e pode ser feita por um algoritmo, enquanto um *stemmer* que não é usado no contexto lexicográfico, é normalmente um algoritmo aplicado a palavras que, em suas várias variações, resulta em um elemento comum que contém significado, que se segue a estratégia de Porter, remove apenas os sufixos.

5.2. Tokenização

A tarefa de tokenização¹ realiza a separação de textos contínuos em unidades separadas. Segundo Manning, Raghavan e Schütze (2009) “Dada um sequência de caracteres e uma unidade definida de documentos, tokenização é a tarefa de cortar ele em pedaços, chamados *tokens*, talvez jogando fora alguns caracteres como os de pontuação” (Manning, Raghavan e Schütze, 2009).

Em Python, é normalmente uma lista de palavras, ou um vetor quando são usados pacotes adicionais. Assim, uma string como ‘Essa é uma frase’, tokenizada resulta em uma lista ['Essa', 'é', 'uma', 'frase'].

Essa descrição faz que a tarefa pareça fácil, e realmente é uma das mais fáceis a princípio, porém, como tudo, aparecem dificuldades no mundo real. Por exemplo, como tokenizar uma palavra como “guarda-chuva”? E palavras compostas que não têm hífen, como “Rio de Janeiro”, conhecidas como expressões multipalavra? Em duas palavras ou em uma só? E como tratar um hífen que separa a palavra entre duas linhas, para juntar suas partes? Que símbolos devem ser considerados como separadores de palavras? Como tratar todos os símbolos possíveis dentro de um conjunto de caracteres? E se forem usados comandos HTML para representar letras, como em ‘a¸ão’? Em que momento do tratamento das frases devemos tokenizar e ficar com as palavras? A pontuação será tokenizada ou desprezada?

¹Essa tarefa já foi traduzida como atomização, porém esse nome é muito menos usado.

Na prática, devido a tantos problemas, temos que ou nos dedicar a tokenização com muito detalhe, ou usar uma ferramenta que já faça isso. Por sorte, existem muitas. Andrade (2021) sugere cinco formas de fazê-lo em Python:

1. usar a função `split`, o que só funciona em casos simples;
2. usar o NLTK;
3. usar o sklearn (Pedregosa et al., 2011);
4. usar o spaCy (Honnibal e Montani, 2017), ou
5. usar o Gensim (ehem e Sojka, 2010);

Não vamos tratar aqui da função `split` do Python porque ela já foi usada no exemplo do Programa 6.1 e só é capaz de usar um único caractere como separador. Deixaremos o uso do sklearn e do spaCy como exercícios.

Usando o NLTK (Bird, Loper e Klein, 2009), o processo é fácil, como apresentado no Programa 5.1, que produz o resultado que o segue.

Programa 5.1: Tokenização com o NLTK

```
1 text = "Um exemplo de texto, com vírgulas e palavras como guarda-chuva."  
2 from nltk.tokenize import word_tokenize  
3 tokens = word_tokenize(text)  
4 print(tokens)
```

Saída do Programa 5.1

```
1 ['Um', 'exemplo', 'de', 'texto', ',', 'com', 'vírgulas',  
2  'e', 'palavras', 'como', 'guarda-chuva', '.']
```

Tokenizadores diferentes podem ter abordagens diferentes de tokenização. O NLTK padrão nos dá palavras e pontuações, já usando o Gensim, por exemplo, ficamos só com as palavras.

Programa 5.2: Tokenização com o Gensim

```
1 text = "Um exemplo de texto, com vírgulas e palavras como guarda-chuva."  
2 from gensim.utils import tokenize  
3 tokens = list(tokenize(text))  
4 print(tokens)
```

Saída do Programa 5.2

```
1 ['Um', 'exemplo', 'de', 'texto', 'com', 'vírgulas',  
2  'e', 'palavras', 'como', 'guarda', 'chuva']
```

5.3. Case Folding

Case folding é o processo de colocar todas as letras na forma minúscula ou maiúscula. Em Python, a função `str.lower()` faz isso rapidamente.

Esta ação pode não ser adequada no caso de extração de informações, principalmente na tarefa de reconhecimento de entidades nomeadas, já os substantivos próprios começam com letra maiúscula em inglês e português.

5.4. Stemmers

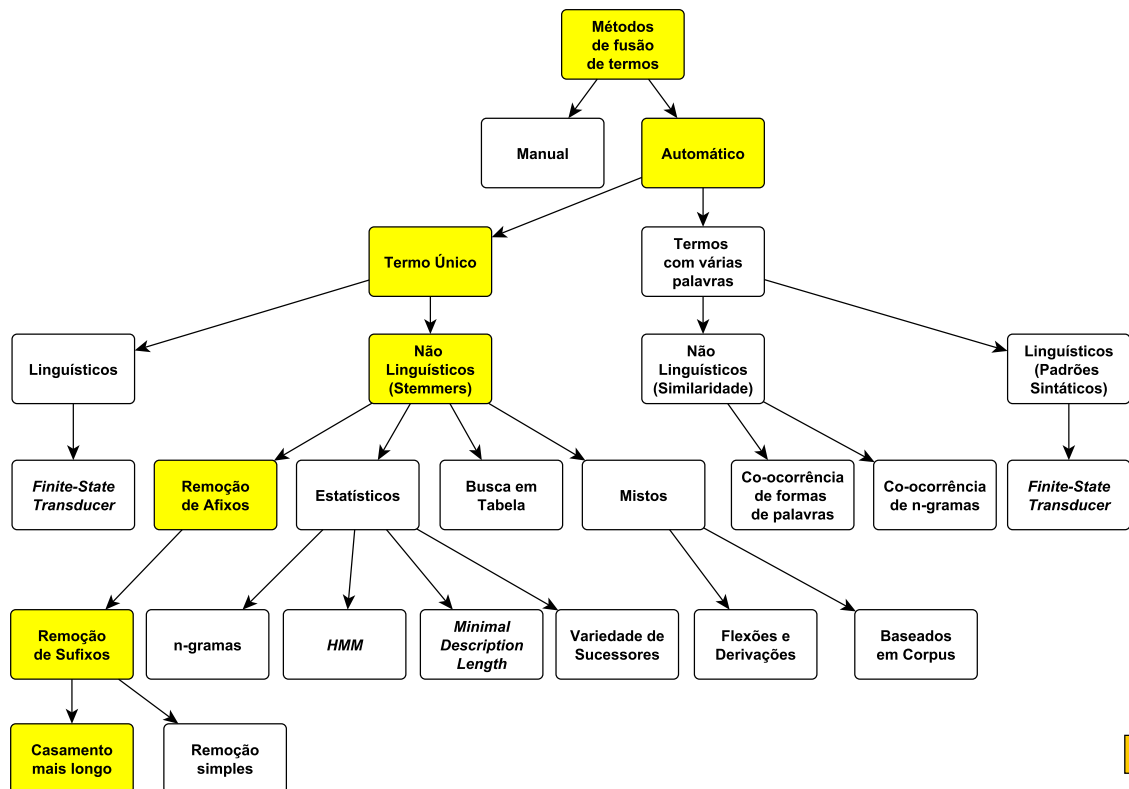
Stemmers são então algoritmos, normalmente heurísticas, que fornecem um possível raiz ou radical de uma palavra, evitando que sistemas de processamento de texto tenham que tratar de todas as suas variações e ajudando na interpretação de um possível significado do texto e, na prática, reduzindo o número de chaves de um sistema de busca ou o número de atributos de um sistema de classificação de texto.

Lovins (1968) publicou o primeiro artigo sobre *stemmers*. Ela define um algoritmo de stemming como “um procedimento computacional que reduz todas as palavras para uma mesma raiz (ou, se prefixos não são tratados, para um mesmo radical) para uma forma comum, usualmente por remover de cada palavras seus sufixos derivacionais e infleccionais” (Lovins, 1968). Seu artigo cita três abordagens anteriores: um por John W. Tukey, o segundo por Michael Lesk orientado por Gerard Salton, e o terceiro por James I. Dolby.

A definição de Lovins deixa claro que um algoritmo de stemming é um algoritmo de fusão (*conflation*). A Figura 5.1 representa uma unificação das classificações de Frakes e Baeza-Yates (1992), (Galvez, MoyaAnegón e Solana, 2005) e Amri e Zenkouar (2018) para esse tipo de algoritmo. Nesse diagrama, o *stemmer* mais conhecido hoje em dia, o de Porter (1980), segue o caminho amarelo. Muitas heurísticas de remoção de afixos, porém, removem apenas os sufixos. Neste texto tratamos apenas dos algoritmos mais tradicionais, e mais usados, por estarem amplamente disponíveis.

Lovins (1968) cita dois princípios que norteiam o desenvolvimento de **stemmers**. O primeiro é a iteração, que busca remover os sufixos por meio de iterações planejadas, já que eles normalmente são adicionados às palavras em uma ordem específica. O segundo é o do casamento mais longo, que escolhe o maior final possível quando é dado uma opção por mais de um final. Além disso, discute que os *stemmers* podem ser livres ou sensitivos ao contexto, onde a existência de um contexto ajuda decidir se uma regra pode ou não ser usada. Por exemplo, exceções podem ser criadas para evitar que um contexto comum, como “abilidade”, que pode ser aplicado a palavras como “sociabilidade” e “computabilidade”, não seja aplicado a palavra “habilidade” (Lovins, 1968). Sua proposta é um *stemmer* sensitivo ao contexto, não iterativo e do tipo casamento mais longo.

A seguir são descritos alguns algoritmos, os de truncagem implementados no NLTK, porém é necessário avisar que essas descrições são baseadas nos textos originais, já antigos, e pecam por falta de uma linguagem mais adequada.



27

Figura 5.1.: Classificação algoritmos de fusão, a partir de Frakes e Baeza-Yates (1992), Galvez, MoyaAnegón e Solana (2005) e Amri e Zenkour (2018). O caminho em amarelo indica os algoritmos de Porter e o RSLP.

5.4.1. Stemmers por Busca em Tabela

Os stemmers mais simples que podemos imaginar são os de busca em tabela. Nesse caso só é necessário ter uma lista de pares palavra-radical. Sua implementação por meio de uma tabela de espalhamento ou de uma estrutura de busca em árvore deixa esse algoritmo muito rápido, apesar de depender de memória.

Essa abordagem, se existisse uma tabela feita por linguistas, seria bastante eficiente e correta, porém não existe uma tabela padrão desse tipo. Além disso, podem existir palavras fora da lista, porque foram recém criadas, ou porque a lista é limitada em algum aspecto.

Abordagens semelhantes, porém, são feitas, por exemplo, em NLTK existe um lematizador feito a partir do WordNet², que retorna a mesma palavra se ela não é encontrada. Também é possível ser mais agressivo e substituir cada palavra por seu significado (*synset* no WordNet).

²https://www.nltk.org/_modules/nltk/stem/wordnet.html

5.4.2. Porter Stemmer

Porter (1980) propôs um dos stemmers que mais influenciou a área, criado de forma empírica. Ele possui uma heurística com 5 passos:

1. remoção de plurais;
2. unificação de padrões de sufixos;
3. manipulação de transformações necessárias para palavras especiais
4. remoção adicional de sufixo
5. remoção de vogal final

O algoritmo é fornecido no NLTK e uma implementação em Python, feita por Vivake Gupta é oferecida no site mantido pelo próprio Porter³, que documenta algumas melhorias no artigo original.

Esse *stemmer* é considerado congelado e melhorias são feitas no *snowball*

A segunda versão, não congelada, do algoritmo de Porter é conhecida como Porter 2. Ela apresenta, em geral, os mesmos passos, com detalhes adicionais em cada passo. Ambas estão implementadas no NLTK^{4,5}.

5.4.3. Snowball Stemmer

Snowball é uma “pequena linguagem de processamento de strings projetada para criar algoritmos de stemming para uso na Recuperação da Informação” (Betts e Bouton, 2022) criada por Porter (2001). A partir dessa linguagem é mantido um algoritmo baseado no algoritmo original de Porter, conhecido como *porter2*, que é traduzido para várias linguagens⁶, além de stemmers para várias outras linguagens, incluindo o português. O site também fornece programas em Snowball para vários outros stemmers⁷. O algoritmo está presente na NLTK (Bird, Loper e Klein, 2009). O nome é uma homenagem a linguagem de processamento de texto SNOBOL.

5.4.4. Lancaster Stemmer

O algoritmo de Lancaster (Paice, 1990), proposto por Paice é baseado em regras, compostas de 5 partes (Paice, 1990):

1. um final de palavra, com um ou mais caracteres;
2. um flag indicando que a forma deve estar intacta;

³<https://tartarus.org/martin/PorterStemmer/>

⁴<https://www.nltk.org/api/nltk.stem.porter.html>

⁵<https://www.nltk.org/api/nltk.stem.snowball.html>

⁶<https://github.com/snowballstem/snowball>

⁷<https://snowballstem.org/algorithms/>

3. um dígito especificando quantas letras devem ser removidas do final da palavra, sendo zero um valor válido;
4. uma string opcional a ser somada no final da palavra, e
5. um símbolo de continuação.

As regras então são organizadas em seções, de acordo com o final das palavras, e o seguinte algoritmo é usado para tratar cada palavra:

- Selecionar a seção relevante
 - Inspecionar a última letra da forma
 - Se não há seção correspondente a essa letra, terminar
 - Se há, considerar a primeira regra da seção relevante
- Verificar aplicabilidade da regra
 - Se as letras finais da forma não correspondem a regra, vá para 4
 - Se as letras casam, e o *flag* de intacto está ativo, e a forma não está intacta, vá para 4
 - Se as condições de aceitação não são válidas, vá para 4 (ver passo seguinte)
- Aplicar regra
 - Apagar no final da forma o número de caracteres especificado na regra
 - Se há uma string a adicionar, adicioná-la
 - Se o símbolo de continuação é “.”, terminar
 - Se o símbolo de continuação é “>”, vá para 1 (isto é, busque mais uma regra)
- Busque por outra regra
 - Vá para a próxima regra
 - Se a seção relevante terminou, terminar
 - Se não, vá para 2

A implementação desse algoritmo no NLTK⁸ por Steven Tomcavage é bastante clara,

5.4.5. Stemmers Criados para o Português

Vários *stemmers* foram criados especificamente para o português, entre eles o STEMBR (Alvares, Garcia e Ferraz, 2005), o *Snowball Stemmer* para português (Snowball Project, 2022) e o RSLP (Lopes, 2004), para os dois últimos há implementações no NLTK.

O algoritmo em português do Snowball define 3 regiões, R1, R2 e RV, relativas as posições de vogais e consoantes da palavra, porém não cabe defini-las aqui. Maiores detalhes podem ser encontrados na página oficial do algoritmo (Snowball Project, 2022).

A partir daí, ele segue 5 passos (Snowball Project, 2022):

1. Remoção de sufixos padronizados;
2. Se não foi removido um final no passo 1, remover sufixos de verbos;
3. Remover sufixos em RV;
4. Remover sufixos residuais, e

⁸<https://www.nltk.org/api/nltk.stem.lancaster.html>

5. Tratar final restante da palavra (casos específicos).

O **RSLP** (Lopes, 2004) também é um algoritmo de truncagem e remoção de sufixos, muito interessante porque definido em regras simples de serem entendidas, semelhantes as de Paice (1990), e usa uma heurística de 8 passos:

1. Eliminar o “s” final das palavras
2. Reduzir para o masculino, para palavras que terminam em “a”
3. Reduzir para o singular
4. Reduzir o advérbio
5. Reduzir o substantivo
6. Reduzir a conjugação, se o sufixo ainda não foi removido
7. Remoção da vogal, se o sufixo ainda não foi removido
8. Remoção dos acentos

Cada passo é formado por um conjunto de regras examinadas em sequência, onde apenas a primeira regra válida é aplicada. As regras são caracterizadas por quatro componentes:

1. sufixo a ser removido
2. tamanho mínimo do radical restante
3. sufixo de substituição, se necessário
4. lista de exceções a regra

Esse stemmer possui 199 regras, sendo que uma regra típica é a tupla {ona, 3, ão, {carona}}.

O RSLP foi implementado no `org.apache.lucene.analysis.pt`⁹. Um outra implementação foi feita em Ruby por Andrew S. Aguiar¹⁰. Uma implementação em Java foi feita por Anael Carvalho para o Elastic Search¹¹. As regras foram reimplementadas em Python para o NLTK por Tiago Tresoldi para o NLTK^{12,13} e podem ser obtidas na Kaggle¹⁴.

5.4.6. Usando os *Stemmers* em Python

O Programa 5.3 apresenta uma implementação do uso dos 3 *stemmers* aqui descrito com o NLTK.

Programa 5.3: Uso dos stemmers para inglês no NLTK

```
1 from nltk.stem.porter import PorterStemmer
2 from nltk.stem.snowball import SnowballStemmer
```

⁹https://lucene.apache.org/core/7_6_0//analyzers-common/org/apache/lucene/analysis/pt/RSLPStemmerBase.html

¹⁰<https://gist.github.com/andrewaguiar/07e04b694d578d6c855dce02853c4410>

¹¹<https://github.com/anaelcarvalho/elasticsearch-analysis-rslp>

¹²<https://www.nltk.org/api/nltk.stem.rslp.html>

¹³https://www.nltk.org/_modules/nltk/stem/rslp.html

¹⁴<https://www.kaggle.com/datasets/nltkdata/rslp-stemmer>

```

3 from nltk.stem.lancaster import LancasterStemmer
4 from nltk.tokenize import word_tokenize
5 import string
6
7 porter_st = PorterStemmer()
8 porter_nltk_st = PorterStemmer(mode=PorterStemmer.NLTK_EXTENSIONS)
9 snow_st = SnowballStemmer("english")
10 lanc_st = LancasterStemmer()
11
12 # passage from Moby Dick from Herman Melville
13 text = """Shipmates, this book, containing only four chapters four yarns
    ↪ is
14 one of the smallest strands in the mighty cable of the Scriptures.
15 """
16 # remove pontuation
17 text = text.translate(str.maketrans('', '', string.punctuation))
18 tokens_en = word_tokenize(text)
19
20 p_tokens = list(map(porter_st.stem, tokens_en))
21 pn_tokens = list(map(porter_nltk_st.stem, tokens_en))
22 ln_tokens = list(map(lanc_st.stem, tokens_en))
23 sn_tokens = list(map(snow_st.stem, tokens_en))
24
25 print("-"*65)
26 print("{0: <10} | {1: <10} | {2: <11} | {3: <10} | {4: <10}".format(
27     "original",
28     "Porter",
29     "Porter NLTK",
30     "Lancaster",
31     "Snowball"))
32 print("-"*65)
33 for i in range(len(p_tokens)):
34     print("{0: <10} | {1: <10} | {2: <11} | {3: <10} | {4: <10}".format(
    ↪ tokens_en[i], p_tokens[i], pn_tokens[i], ln_tokens[i], sn_tokens[i])
    ↪ )
35 print("-"*65)

```

Saída para o Programa 5.3

```

1 -----
2 original   | Porter    | Porter NLTK | Lancaster | Snowball
3 -----
4 Shipmates  | shipmat   | shipmat     | shipm     | shipmat
5 this       | thi       | thi         | thi       | this
6 book       | book      | book        | book      | book

```

7	containing	contain	contain	contain	contain
8	only	onli	onli	on	onli
9	four	four	four	four	four
10	chapters	chapter	chapter	chapt	chapter
11	four	four	four	four	four
12	yarns	yarn	yarn	yarn	yarn
13	is	is	is	is	is
14	one	one	one	on	one
15	of	of	of	of	of
16	the	the	the	the	the
17	smallest	smallest	smallest	smallest	smallest
18	strands	strand	strand	strands	strand
19	in	in	in	in	in
20	the	the	the	the	the
21	mighty	mighti	mighti	mighty	mighti
22	cable	cabl	cabl	cabl	cabl
23	of	of	of	of	of
24	the	the	the	the	the
25	Scriptures	scriptur	scriptur	scriptures	scriptur
26	-----				

O Programa 5.4 apresenta o uso dos *stemmers* Snowball e RSLP para português com o mesmo software.

Programa 5.4: Uso dos stemmers para português no NLTK

```

1 from nltk.stem import RSLPStemmer
2 from nltk.stem.snowball import SnowballStemmer
3 #Trecho de Dom Casmurro por Machado de Assis
4 texto = """Uma noite destas, vindo da cidade para o Engenho Novo,
    ↪ encontrei
5 num trem da Central um rapaz
6 aqui do bairro, que eu conheço de vista e de chapéu. Cumprimentou-me,
7 sentou-se ao pé de mim, falou da lua e dos ministros,
8 e acabou recitando-me versos."""
9 texto = texto.translate(str.maketrans('—,.?!', ' '))
10 snow_p_st = SnowballStemmer("portuguese")
11 rslp_st = RSLPStemmer()
12 tokens_ma = word_tokenize(texto)
13 pt_tokens = list(map(snow_p_st.stem, tokens_ma))
14 pt_n_tokens = list(map(rslp_st.stem, tokens_ma))
15 print("—" * 35)
16 print("{0: <12} | {1: <10} | {2: <10}".format("Original", "Snowball", "RSLP"
    ↪ ))
17 print("—" * 35)

```

```

18 for i in range(len(pt_tokens)):
19     print("{0: <12} | {1: <10} | {2: <10}".format(tokens_ma[i],pt_tokens[i]
        ↪ ],pt_n_tokens[i]))
20 print("--"*35)

```

Saída para o Programa 5.4

```

1 -----
2 Original      | Snowball      | RSLP
3 -----
4 Uma           | uma           | uma
5 noite         | noit          | noit
6 destas        | dest          | dest
7 vindo         | vind          | vind
8 da            | da            | da
9 cidade        | cidad         | cidad
10 para          | par           | par
11 o             | o             | o
12 Engenho       | engenh        | engenh
13 Novo          | nov           | nov
14 encontrei     | encontr       | encontr
15 num           | num           | num
16 trem          | trem          | tr
17 da            | da            | da
18 Central       | central       | centr
19 um            | um            | um
20 rapaz         | rapaz         | rapaz
21 aqui          | aqu           | aqu
22 do            | do            | do
23 bairro        | bairr        | bairr
24 que           | que           | que
25 eu            | eu            | eu
26 conheço       | conheç        | conheç
27 de            | de            | de
28 vista         | vist         | vist
29 e             | e             | e
30 de            | de            | de
31 chapéu        | chapéu        | chapéu
32 Cumprimentou  | cumpriment    | cumpriment
33 me            | me            | me
34 sentou        | sent          | sent
35 se            | se            | se
36 ao            | ao            | ao
37 pé            | pé            | pé
38 de            | de            | de

```

```

39 mim      | mim      | mim
40 falou    | fal      | fal
41 da       | da       | da
42 lua      | lua      | lua
43 e        | e        | e
44 dos      | dos      | do
45 ministros | ministr  | ministr
46 e        | e        | e
47 acabou   | acab     | acab
48 recitando | recit    | recit
49 me       | me       | me
50 versos   | vers     | vers
51 -----

```

Um **stemmer** pode se favorecer de uma estratégia de cache se processando muitas palavras.

Programa 5.5: Usando stemmer com cache em Python

```

1 from nltk.corpus import gutenber
2 from nltk.tokenize import word_tokenize
3 from nltk.stem.porter import PorterStemmer
4 from functools import lru_cache
5 from time import perf_counter
6 import string
7
8 # Junta todos os textos do Gutenberg
9 text = ''
10 for f in gutenber.fileids():
11     print(f)
12     text += gutenber.raw(f)
13 text = text.translate(str.maketrans('','',string.punctuation))
14
15 # Stem todos sem cache
16 print("Iniciando stemming Porter sem cache...")
17 tic = perf_counter()
18 tokens_en = word_tokenize(text)
19 porter_st = PorterStemmer()
20 p_tokens = list(map(porter_st.stem, tokens_en))
21 toc = perf_counter()
22
23 print(f"Tempo sem cache {toc - tic:0.4f} seconds")
24
25 #Stem all with cache
26

```

```

27 @lru_cache(maxsize=1000)
28 def cached_stem(word):
29     return porter_st.stem(word)
30
31 print("Iniciando stemming Porter com cache...")
32 tic = perf_counter()
33 pc_tokens = list(map(cached_stem, tokens_en))
34 toc = perf_counter()
35 print(f"Tempo com cache {toc - tic:0.4f} seconds")
36
37 print(cached_stem.cache_info())

```

Saída para o Programa 5.5

```

1 austen-emma.txt
2 austen-persuasion.txt
3 austen-sense.txt
4 bible-kjv.txt
5 blake-poems.txt
6 bryant-stories.txt
7 burges-busterbrown.txt
8 carroll-alice.txt
9 chesterton-ball.txt
10 chesterton-brown.txt
11 chesterton-thursday.txt
12 edgeworth-parents.txt
13 melville-moby_dick.txt
14 milton-paradise.txt
15 shakespeare-caesar.txt
16 shakespeare-hamlet.txt
17 shakespeare-macbeth.txt
18 whitman-leaves.txt
19 Iniciando stemming Porter sem cache...
20 Tempo sem cache 59.8562 seconds
21 Iniciando stemming Porter com cache...
22 Tempo com cache 14.6462 seconds
23 CacheInfo(hits=1684211, misses=451115, maxsize=1000, currsize=1000)

```

Apesar do algoritmo de Porter ser rápido, processando mais de 2 milhões de palavras em aproximadamente 60s, com o uso de um cache LRU esse tempo pode ser levado a aproximadamente 2 ou 3 segundos, como mostra a Figura 5.2, o que pode ser bom no processamento de grandes coleções. A análise do número de *misses* indica o número de formas distintas, 66392 palavras. No computador onde foi testado, um cache de 32768 itens já apresentou um desempenho de 2,8s, enquanto um cache de 65536, onde cabem

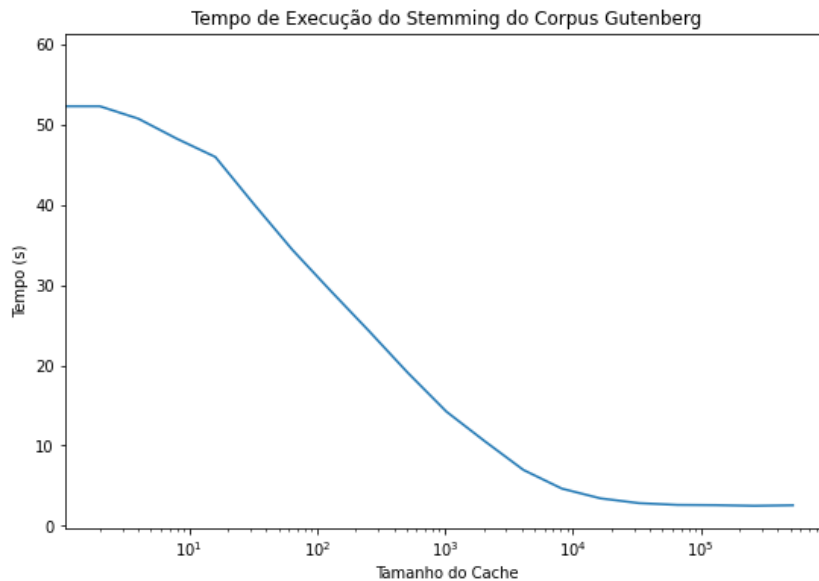


Figura 5.2.: Tempo de execução do stemmer de Porter com diferentes tamanhos de cache, potências de 2 de 0 a 524288 entradas, mostrando um limite prático para essa coleção, onde há 66392 formas únicas. Um cache de 32768 foi suficiente para atingir um desempenho próximo do máximo.

quase todas, apresenta um desempenho de 2,6s. Um cache onde cabiam todas as palavras apresentou um desempenho equivalente.

5.5. Remoção de *Stopwords*

Stopwords são palavras filtradas do texto antes de um processamento Manning, Raghavan e Schüze (2008). Outra definição é a de serem palavras funcionais que não trazem significado ao texto, como conjunções e preposições, logo não interessantes para um índice. Além disso, são palavras que não precisam ser indexadas porque sua presença é tão comum nos documentos que elas não fazem diferença nos algoritmos de busca e classificação, já que não diferenciam um do outro (Nothman, Qin e Yurchak, 2018).

O Programa 5.6, calcula e gera a Figura 5.3, com o gráfico da frequência das 50 palavras mais usadas no corpus Gutenberg do NLTK. É possível notar duas propriedades: a distribuição se parece com a distribuição de Zipf e as palavras mais frequentes são as mesmas que encontramos nas listas de *stopwords*.

Programa 5.6: Contagem das 50 primeiras palavras do corpus Gutenberg no NLTK

```
1 from nltk.probability import FreqDist
2 from nltk.corpus import gutenberg
3 import string
4 import matplotlib.pyplot as plt
```

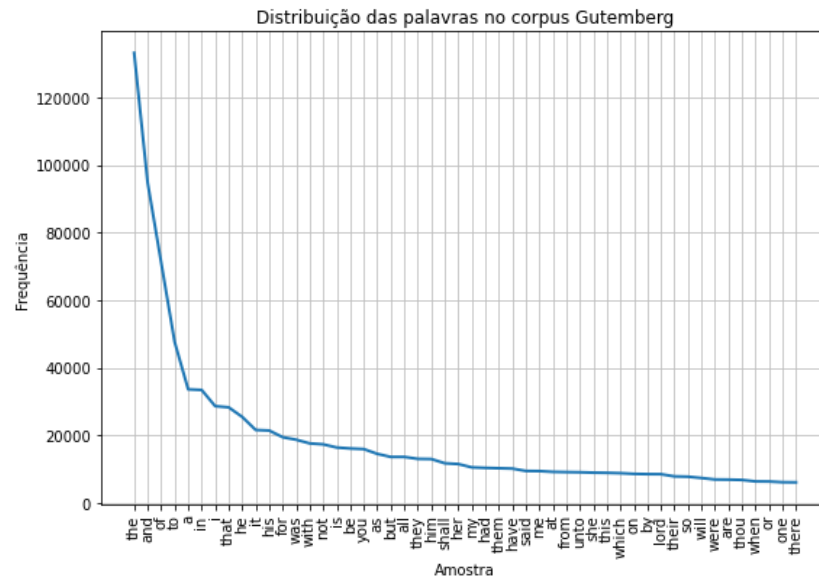


Figura 5.3.: Contagem das 50 palavras mais frequentes do corpus Gutenberg, pelo programa Programa 5.6

```

5
6 text = ''
7 for f in gutenber.fileids():
8     print(f)
9     text += gutenber.raw(f)
10 text = text.translate(str.maketrans('','',string.punctuation))
11
12 fd = FreqDist(word.lower() for word in text.split())
13 print("Frequencies done!")
14 p = fd.plot(50,show=False,title="Distribuição das palavras no corpus
    ↪ Gutenberg")
15 p.set_xlabel("Amostra")
16 p.set_ylabel("Frequência")
17 plt.show()

```

Normalmente as *stopwords* são retiradas do índice a partir de um filtro feito com uma lista predefinida por seres humanos, mas elas podem também ser criadas a partir da coleção. Não há um acordo específico sobre que palavras devem ou não ser removidas, ou mesmo se devem ser removidas.

No passado, devido ao tamanho dos índices, era muito importante remover essas palavras, porém atualmente muitos sistemas não o fazem, principalmente porque podem ser usadas em n-gramas. Muitos experimentos de classificação, hoje em dia, são feitos com e sem *stopwords* para analisar seu efeito no processo.

Vários pacotes de processamento de texto fornecem listas de *stopwords* em diferentes línguas. Além disso, elas podem ser encontradas em:

- <http://snowball.tartarus.org/>
- <https://countwordsfree.com/stopwords>
- <http://www.ranks.nl/stopwords/>
- <http://members.unine.ch/jacques.savoy/clef/>
- <https://cran.r-project.org/web/packages/stopwords/>
- <https://github.com/igorbrigadir/stopwords/tree/21fb2ef>
- http://ir.dcs.gla.ac.uk/resources/linguistic_utils/

O sistema *Smart* (Salton e McGill, 1983a) usava a lista de *stopwords* da Tabela 5.1, sendo ela considerada a primeira lista disponível (Nothman, Qin e Yurchak, 2018).

Programa 5.7: Removendo as *stopwords* com NLTK

```

1 import nltk
2 from nltk.corpus import stopwords
3 from nltk.tokenize import word_tokenize
4
5 stop_en = stopwords.words('english')
6 stop_pt = stopwords.words('portuguese')
7
8
9 print("Stopwords in English:\n")
10 print(stop_en)
11
12 test_frase = "There is no such thing as a free lunch"
13
14 print("\n\nFiltrando: ", test_frase)
15
16 word_tokens = word_tokenize(test_frase)
17 filtered_tokens = [w for w in word_tokens if not w.lower() in stop_en]
18
19 print(filtered_tokens)
20
21 print("\n\n\n\nStopwords em português:\n")
22 print(stop_pt)
23
24 frase_teste = "Não existe almoço grátis"
25
26 print("\n\nFiltrando: ", frase_teste)
27
28 tokens_palavra = word_tokenize(frase_teste)
29 tokens_filtrados = [p for p in tokens_palavra if not p.lower() in stop_pt]
30

```

Tabela 5.1.: Lista de *stopwords* do sistema *Smart* (Salton e McGill, 1983a).

a a's able about above according accordingly across actually after afterwards again
against ain't all allow allows almost alone along already also although always am among
amongst an and another any anybody anyhow anyone anything anyway anyways anywhere
apart appear appreciate appropriate are aren't around as aside ask asking associated
at available away awfully b be became because become becomes becoming been before
beforehand behind being believe below beside besides best better between beyond both
brief but by c c'mon c's came can can't cannot cant cause causes certain certainly
changes clearly co com come comes concerning consequently consider considering contain
containing contains corresponding could couldn't course currently d definitely described
despite did didn't different do does doesn't doing don't done down downwards during e
each edu eg eight either else elsewhere enough entirely especially et etc even ever every
everybody everyone everything everywhere ex exactly example except f far few fifth first
five followed following follows for former formerly forth four from further furthermore
g get gets getting given gives go goes going gone got gotten greetings h had hadn't
happens hardly has hasn't have haven't having he he's hello help hence her here here's
hereafter hereby herein hereupon hers herself hi him himself his hither hopefully how
howbeit however i i'd i'll i'm i've ie if ignored immediate in inasmuch inc indeed indicate
indicated indicates inner insofar instead into inward is isn't it it'd it'll it's its itself j just
k keep keeps kept know knows known l last lately later latter latterly least less lest let
let's like liked likely little look looking looks ltd m mainly many may maybe me mean
meanwhile merely might more moreover most mostly much must my myself n name
namely nd near nearly necessary need needs neither never nevertheless new next nine
no nobody non none noone nor normally not nothing novel now nowhere o obviously
of off often oh ok okay old on once one ones only onto or other others otherwise ought
our ours ourselves out outside over overall own p particular particularly per perhaps
placed please plus possible presumably probably provides q que quite qv r rather rd re
really reasonably regarding regardless regards relatively respectively right s said same
saw say saying says second secondly see seeing seem seemed seeming seems seen self
selves sensible sent serious seriously seven several shall she should shouldn't since six so
some somebody somehow someone something sometime sometimes somewhat somewhere
soon sorry specified specify specifying still sub such sup sure t t's take taken tell tends
th than thank thanks thanx that that's thats the their theirs them themselves then
thence there there's thereafter thereby therefore therein theres thereupon these they
they'd they'll they're they've think third this thorough thoroughly those though three
through throughout thru thus to together too took toward towards tried tries truly try
trying twice two u un under unfortunately unless unlikely until unto up upon us use used
useful uses using usually uucp v value various very via viz vs w want wants was wasn't
way we we'd we'll we're we've welcome well went were weren't what what's whatever
when whence whenever where where's whereafter whereas whereby wherein whereupon
wherever whether which while whither who who's whoever whole whom whose why will
willing wish with within without won't wonder would would wouldn't x y yes yet you
you'd you'll you're you've your yours yourself yourselves z zero

Tabela 5.2.: Lista de *stopwords* em português brasileiro obtida em <http://www.ranks.nl/stopwords/>.

a ainda alem ambas ambos antes ao aonde aos apos aquele aqueles
as assim com como contra contudo cuja cujas cujo cujos da das de
dela dele deles demais depois desde desta deste dispoe dispoem diversa
diversas diversos do dos durante e ela elas ele eles em entao entre essa
essas esse esses esta estas este estes ha isso isto logo mais mas mediante
menos mesma mesmas mesmo mesmos na nas nao nas nem nesse neste
nos o os ou outra outras outro outros pelas pelas pelo pelos perante
pois por porque portanto proprio propios quais qual qualquer quando
quanto que quem quer se seja sem sendo seu seus sob sobre sua suas tal
tambem teu teus toda todas todo todos tua tuas tudo um uma umas
uns

```
31 print(tokens_filtrados)
```

Saída do Programa 5.7

```
1 Stopwords in English:
2
3 ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves',
4 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours',
5 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she',
6 "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
7 'they', 'them', 'their', 'theirs', 'themselves', 'what',
8 'which', 'who', 'whom', 'this', 'that', "that'll", 'these',
9 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been',
10 'being', 'have', 'has', 'had', 'having', 'do', 'does',
11 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
12 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for',
13 'with', 'about', 'against', 'between', 'into', 'through',
14 'during', 'before', 'after', 'above', 'below', 'to', 'from',
15 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under',
16 'again', 'further', 'then', 'once', 'here', 'there', 'when',
17 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few',
18 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not',
19 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't',
20 'can', 'will', 'just', 'don', "don't", 'should', "should've",
21 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren',
22 "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn',
23 "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven',
24 "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't",
25 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't",
26 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't",
```

```
27 'won', 'won't', 'wouldn', 'wouldn't']
28
29
30 Filtrando: There is no such thing as a free lunch
31 ['thing', 'free', 'lunch']
32
33
34
35
36 Stopwords em português:
37
38 ['de', 'a', 'o', 'que', 'e', 'é', 'do', 'da', 'em',
39 'um', 'para', 'com', 'não', 'uma', 'os', 'no', 'se',
40 'na', 'por', 'mais', 'as',
41 'dos', 'como', 'mas', 'ao', 'ele', 'das', 'à', 'seu',
42 sua', 'ou', 'quando', 'muito', 'nos', 'já', 'eu', 'também',
43 'só', 'pelo', 'pela', 'até', 'isso', 'ela', 'entre',
44 'depois', 'sem', 'mesmo', 'aos', 'seus', 'quem', 'nas',
45 'me', 'esse', 'eles', 'você', 'essa', 'num', 'nem',
46 'suas', 'meu', 'às', 'minha', 'numa', 'pelos', 'elas',
47 'qual', 'nós', 'lhe', 'deles', 'essas', 'esses', 'pelas',
48 'este', 'dele', 'tu', 'te', 'você', 'vos', 'lhes', 'meus',
49 'minhas', 'teu', 'tua', 'teus', 'tuas', 'nosso', 'nossa',
50 'nossos', 'nossas', 'dela', 'delas', 'esta', 'estes',
51 'estas', 'aquele', 'aquela', 'aqueles', 'aquelas', 'isto',
52 'aquilo', 'estou', 'está', 'estamos', 'estão', 'estive',
53 'esteve', 'estivemos', 'estiveram', 'estava', 'estávamos',
54 'estavam', 'estivera', 'estivéramos', 'esteja', 'estejamos',
55 'estejam', 'estivesse', 'estivéssemos', 'estivessem', 'estiver',
56 'estivermos', 'estiverem', 'hei', 'há', 'havesmos', 'hão',
57 'houve', 'houvemos', 'houveram', 'houvera', 'houvéramos',
58 'haja', 'hajamos', 'hajam', 'houvesse', 'houvéssemos',
59 'houvessem', 'houver', 'houvermos', 'houverem', 'houverei',
60 'houverá', 'houveremos', 'houverão', 'houveria', 'houveríamos',
61 'houveriam', 'sou', 'somos', 'são', 'era', 'éramos', 'eram',
62 'fui', 'foi', 'fomos', 'foram', 'fora', 'fôramos', 'seja',
63 'sejamos', 'sejam', 'fosse', 'fôssemos', 'fossem', 'for',
64 'formos', 'forem', 'serei', 'será', 'seremos', 'serão',
65 'seria', 'seríamos', 'seriam', 'tenho', 'tem', 'temos',
66 'tém', 'tinha', 'tínhamos', 'tinham', 'tive', 'teve',
67 'tivemos', 'tiveram', 'tivera', 'tivéramos', 'tenha',
68 'tenhamos', 'tenham', 'tivesse', 'tivéssemos', 'tivessem',
69 'tiver', 'tivermos', 'tiverem', 'terei', 'terá', 'teremos',
70 'terão', 'teria', 'teríamos', 'teriam']
```

```

71
72
73 Filtrando: Não existe almoço grátis
74 ['existe', 'almoço', 'grátis']

```

No NLTK é possível estender a stopwords padrão usando o método `extend`, cujo parâmetro é uma lista de *strings*.

5.5.1. Usando o KNIME

O Knime possui um nó dedicado para stopwords, o “Stop Word Filter”. Ele recebe uma tabela de documentos e pode aplicar tanto uma lista padrão, usando uma entre 12 línguas, incluindo o português, ou uma lista customizada, proveniente de uma tabela.

Duas são as abas de configuração importantes no nó “Stop Word Filter”: a *Preprocessing* (Figura 5.4) e a *Filter Options* (Figura 5.5).

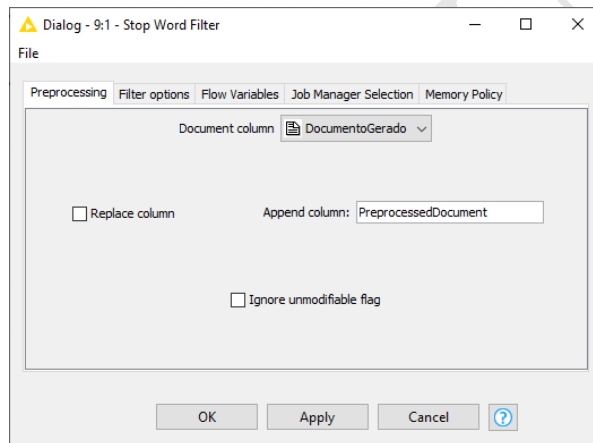


Figura 5.4.: Aba Preprocessing do nó Stop Word Filter no KNIME

5.5.2. Problemas com *stopwords*

A primeira pergunta que vem com o uso de *stopwords* é quando removê-las dentro do fluxo de pré-processamento. Essa pergunta pode ficar mais complicada caso seja desejada uma análise léxica ou sintática da frase.

Também é possível tratar *stopwords* automáticas, tanto as eliminando como as substituindo. Siglas, por exemplo, podem ser, ou não, desconsideradas. Siglas especiais, como “007” ou “3D” podem ser substituídas por palavras com significado semelhante, como “James Bond” ou “tridimensional”. Outras siglas, fora de um vocabulário de controle, podem ser filtradas.

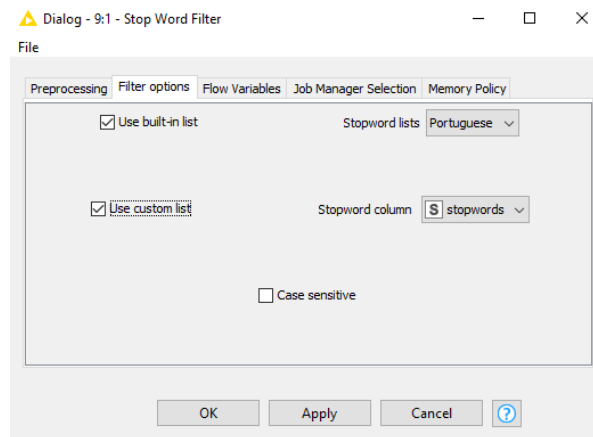


Figura 5.5.: Aba Filter Options do nó Stop Word Filter no KNIME

Outra questão interessante é como fazer com termos compostos que desejamos indexar. Por exemplo, pode ser interessante indexar o trigramma “banco de dados”. Esses termos podem estar em uma lista positiva, conhecida como *go-words*.

Programa 5.8: Programa que usa *go-words* compostas

```

1 import nltk
2 from nltk.corpus import stopwords
3 from nltk.tokenize import word_tokenize
4
5
6 def check_a_go(frase, indice, comb):
7     seq = [ w.lower() for w in frase[indice+1:indice+len(comb)+1] ]
8     if seq == comb:
9         return True
10    else:
11        return False
12
13 def check_go(frase, word, indice, go):
14     if word in go:
15         for comb in go[word]:
16             if check_a_go(frase, indice, comb):
17                 return [word]+comb
18     else:
19         return False
20
21
22
23
24 stop_pt = stopwords.words('portuguese')

```

```

25 frase_teste = "A criação dos bancos de dados foi muito influente no\
26     desenvolvimento de software, permitindo o use de diferentes \
27     linguagens de programação. São Paulo é uma cidade grande."
28
29 go_words = {"bancos" : [["de", "dados"], ["bancos","financeiros"]],
30             "linguagens" : [["de", "programação"]],
31             "são" : [["paulo"]]}
32
33
34 word_tokens = word_tokenize(frase_teste)
35
36 filtered = []
37
38 i = 0
39 while i < len(word_tokens):
40     cur_word = word_tokens[i].lower()
41     if comb := check_go(word_tokens, cur_word, i, go_words):
42         filtered.append(comb)
43         i += len(comb)
44     elif cur_word not in stop_pt:
45         filtered.append(cur_word)
46     i += 1
47 print(filtered)

```

5.6. Normalização de texto

Normalização é o ato de corrigir um texto com erros ortográficos, isto é, palavras escritas erradas. Os erros podem vir de:

- o autor não sabe escrever a palavra, ou tem uma dúvida e opta pelo modo errado, ou
- o autor sabe escrever a palavra, mas erra ao fazê-lo, e não percebe o erro, por exemplo, digitando uma tecla muito próxima da tecla correta, como trocar um “o” por um “p”.
- o autor escolher usar um vocabulário informal, uma gíria, uma abreviação, a oralidade.

Uma palavra pode estar errada absolutamente, sendo chamada de palavra não real, como a palavra “atrazo”, ou apenas em um contexto, como a palavra “contínua”, que é usada no lugar da palavra “continua”, na frase “João contínua a fazer o mesmo erro”.

Parte dos erros provindo do desconhecimento da forma correta de escrever a palavra vêm do fato de que a relação entre fonemas e grafemas não é biunívoca, isto é, não existe

exatamente um grafema para cada fonema e um fonema para cada grafema. Isso pode ser visto em francês, onde quase todas as conjugações de um verbo regular são faladas do mesmo jeito, mas são escritas diferentes, ou dos vários sons que uma mesma letra pode assumir, até mesmo em uma mesma palavra, em inglês. As relações entre grafemas e fonemas podem ser:

- Poligâmicas, quando letras representam diferentes sons, dependendo da posição em que estão na palavra. Em português, o “s” tem o som da palavra “sapo”, semelhante ao c-cedilha, quando aparece no início da palavra, mas tem som de “z” quando aparece entre duas vogais, como em “asa”, e
- Poliândrica, quando duas letras podem gerar o mesmo som, como o som de “k” gerados pelas letras “c”, quando sucedida das vogais “a”, “o” ou “u”, e pelo ditongo “qu”, quando sucedido pelas vogais “e” ou “i”. Assim escrevemos “copo” e “quinto”, e falamos “kopo” e “kinto”.

Os erros podem ser categorizados segundo a forma como são consertados, de acordo com a necessidade de inserção, deleção, substituição ou transposição de letras.

5.6.1. Estratégias gerais de correção

Existem duas estratégias gerais de correção:

1. com aprendizado de máquina, e
2. sem aprendizado de máquina.

Norvig propõe um algoritmo simples que explica, de forma geral, como funciona um corretor de textos. A ideia básica é:

1. Ter um corpus com muitos exemplos de palavras corretamente escritas e gerar um dicionário;
2. Pegar uma palavra para corrigir, sem ideia do contexto;
3. Se a palavra está no dicionário, ela é aceita;
4. Caso contrário, pesquisa no dicionário todas as palavras candidatas a substituí-la que possuam uma distância de edição igual a 1;
5. Se existirem essas palavras, aceita a com maior frequência;
6. Caso contrário, faz o mesmo com a distância 2;
7. Caso ainda não tenha encontrado, aceita a palavra.

Como gerar todas as palavras com distância 1:

1. Apaga todos os caracteres 1 a 1;
2. Transpõe todos os caracteres, posição por posição;
3. Troca todos os caracteres por todos os outros possíveis;
4. Insere um caractere extra em todos os intervalos.

Esse passo a passo geraria 234 palavras para tentar corrigir a palavra “word”, caso ela não estivesse no dicionário. Se forem geradas todas as palavras com distância 1 a partir

dessas palavras, temos todas as palavras com distância 2, que serão 60580 no caso da palavra “word”.

Programa 5.9: Corretor simples segundo Norvig

```

1 def P(word, N=sum(WORDS.values())):
2     "Calcula a probabilidade da palavra aparecer"
3     return WORDS[word] / N
4
5 def correction(word):
6     "Busca a correção mais provável da palavra"
7     return max(candidates(word), key=P)
8
9 def candidates(word):
10    "Gera todas as correções possíveis de distância de edição 1 e 2"
11    return (known([word]) or known(edits1(word)) or
12            known(edits2(word)) or [word])
13
14 def known(words):
15    "O conjunto de palavras existentes no texto"
16    return set(w for w in words if w in WORDS)
17
18 def edits1(word):
19    "Calcula todas as palavras a 1 edição de distância"
20    letters = 'abcdefghijklmnopqrstuvwxyz'
21    splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
22    deletes = [L + R[1:] for L, R in splits if R]
23    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
24    replaces = [L + c + R[1:] for L, R in splits if R for c in
25                ↪ letters]
26    inserts = [L + c + R for L, R in splits for c in
27               ↪ letters]
28    return set(deletes + transposes + replaces + inserts)
29
30 def edits2(word):
31    "Calcula todas as palavras a duas edições de distância"
32    return (e2 for e1 in edits1(word) for e2 in edits1(e1))

```

5.7. Uma Cadeia de Pré-Processamento em KNIME

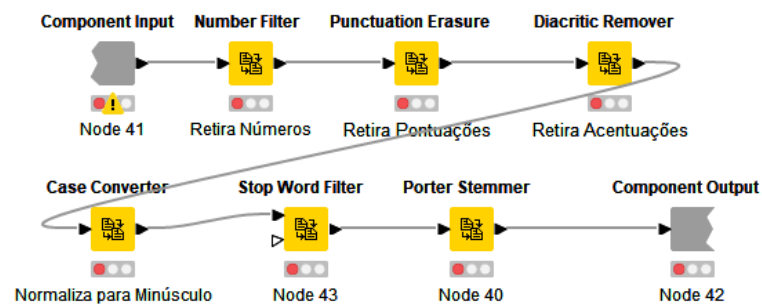


Figura 5.6.: Cadeia de pré-processamento em KNIME

5.8. Exercício

Exercício 5.1:

Faça um programa que remova todos os caracteres de um texto em português que não seja uma letra ou letra acentuada.

Exercício 5.2:

Faça um programa que corrija as palavras de um texto em português seguindo o algoritmo proposto por Jurafsky.

Exercício 5.3:

Faça um programa que coloque as palavras de um texto em português em minúsculas, funcionando para os acentos e o c-cedilha.

Exercício 5.4:

Faça um programa que coloque todos os verbos de um texto em português no infinitivo.

Exercício 5.5:

Faça um programa que coloque todas as palavras de um texto em português no singular.

Exercício 5.6:

Faça um programa que coloque todas as palavras de um texto em português no gênero masculino (ou feminino).

Exercício 5.7:

Faça um programa que remova um grupo de palavras definidas como *stop-words* de um texto em português.

DRAFT

DRAFT

Estruturas de Dados Chave-Valor

Este capítulo apresenta algumas estruturas de dados adequadas para o modelo de recuperação de informação, onde dada uma chave deve ser obtido um valor.

6.1. Lista Invertida

A lista invertida (Zobel e Moffat, 2006; Zobel, Moffat e Ramamohanarao, 1998), ou o índice invertido, ou *inverted files*, ou ainda *posting file*, é a estrutura básica usada na maioria dos sistemas de indexação de documentos. O nome vem do fato que normalmente a indexação permite que se recupere todas as partes de um objeto conhecendo uma identificação do objeto, porém na lista invertida o objeto é recuperado a partir de suas partes.

Em qualquer representação de um documento por meio de suas partes constituintes, os termos, é muito provável que, comparado com todos os termos encontrados em todos os documentos de um conjunto, cada documento tenha apenas uma pequena quantidade de termos. Isso faz com que as representações matriciais termo \times documento sejam esparsas, isto é, contenham muitos zeros. A lista invertida é uma forma de fazer essa representação de forma a economizar espaço.

6.1.1. Exemplo de Listas Invertidas

Sejam três documentos, que chamemos de **exemplo A**:

1. Maria comprou um quilo de banana prata no mercado.
2. Um quilo de prata vale muito dinheiro.
3. Maria tem dinheiro para comprar um quilo de banana, mas não de prata.

Um lista invertida desses documentos pode ser representada como na Tabela 6.1, após a substituição das maiúsculas por minúsculas.

Tabela 6.1.: Representação de um lista invertida.

Termo (chave)	Documentos
banana	1, 3
comprar	3
comprou	1
de	1, 2, 3
dinheiro	2, 3
maria	1, 3
mas	3
mercado	1
muito	2
no	1
não	3
para	3
prata	1, 2, 3
quilo	1, 2, 3
tem	3
um	1, 2, 3
vale	2

É importante notar que para processá-las é importante que duas propriedades sejam mantidas: para o acesso rápido as chaves elas devem estar em uma estrutura que favoreça a busca, sendo ordenadas, como em uma lista ordenada (complexidade para a busca $O(n)$), árvore binária (complexidade para a busca $O(\log n)$), ou em uma *BTree* ou *B+Tree* (complexidade para a busca $O(\log n)$), ou colocadas em uma estrutura de espalhamento (*hash*) (complexidade para a busca $O(1)$). Além disso, os documentos apontados também devem estar ordenados, de modo que seja possível usar algoritmos rápidos para testar se uma chave está em dois documentos (É lógico).

Na Tabela 6.1 podemos ver algumas características que estão ou não tratados. Foi ignorada a quantidade de vezes que uma palavra aparece em um texto, como a palavra “de” no item 3. Também podemos ver que a palavra “Um” foi colocada em minúsculas, e que a conjugação dos verbos separa “comprou” de “comprar”, o que não é necessariamente desejado. Podemos também ver que em uma coleção, é possível que algumas palavras apareçam em muitos documentos, e outras só em um. A própria escolha das palavras, ou tokens, é importante. Essas e outras questões são tratadas na fase de pré-processamento do texto.

Um uso da comum lista invertida, além de obter todos os documentos que possuem uma palavra, é obter todos os documentos que possuem um conjunto de palavras, usando o E ou ou OU lógico. O algoritmo da conjunção, conhecido como *merge*, implica em usar dois ponteiros que andam nas listas ordenadas de documentos e é descrito claramente em Manning, Raghavan e Schütze (2009).

6.2. Melhorias na lista invertida

Várias melhorias podem ser feitas na lista invertida.

- usar *skip-pointers* para acelerar o algoritmo de merge;
- guardar a documentos por palavra na lista, para poder escolher as menores listas em caso de uma conjunção de múltiplas palavras, acelerando o processo;
- guardar todas as posições da palavra em um documento;
- guardar quantas instâncias da palavra aparecem em um documento;
- processar bigramas (duas palavras em sequência) e outros n-gramas¹;
- usar estruturas de disco para manter listas maiores que a memória RAM, e
- usar cache com as estruturas de disco.

6.2.1. Listas Invertidas em Python

Em Python puro, a maneira mais eficiente de realizar uma lista invertida em memória é por meio de um dicionário, que podem ser salvas diretamente em um arquivo, se necessário, com pickle, ou sua versão mais rápida, cPickle, ou mesmo em JSON, para compatibilidade com outras linguagens. Se a lista não couber em memória, será necessário uma estrutura em disco mais complexa, o que não é uma alternativa comum em Python. Também pode se usar sqlite 3, que é leve e rápido, com uma tabela contendo a chave e a lista de documentos, evitando *joins*.

Existem alguns pacotes que oferecem estruturas em disco acessadas no paradigma chave-valor. BTrees² é um pacote criado para ZODB³, um banco de dados de objeto nativo em Python, que também é uma opção de uso.

Outras opções são possíveis para bases chave-valor: Berkeley DB, HDF5 ou Kyoto Cabinet.

6.2.2. Implementação trivial em Python

A implementação em Python do Programa 6.1 é bastante simples, e foi usada para gerar as linhas da Tabela 6.1.

Programa 6.1: Criação simples de uma lista invertida

```

1 # o método split só admite um separador, então removemos as vírgulas
2 base = [ "Maria comprou um quilo de banana prata no mercado" ,
3 "Um quilo de prata vale muito dinheiro" ,
4 "Maria tem dinheiro para comprar um quilo de banana mas não de prata"]

```

¹Na verdade, se a busca contém um trigramma ou uma sequência maior de palavras, o uso de bigramas que compõe o n-grama é capaz de atender a demanda rapidamente(Manning, Raghavan e Schütze, 2009)

²<https://pypi.org/project/BTrees/>

³<https://zodb.org/en/latest/> e tutorial em <https://zodb.org/en/latest/tutorial.html>

```

5
6 lista_invertida = {}
7
8 for i in range(len(base)):
9     fraselc = str.lower(base[i]).split()
10    for palavra in fraselc:
11        docs = lista_invertida.get(palavra, set())
12        docs.add(i+1)
13        lista_invertida[palavra]=docs
14
15 chaves = [i for i in lista_invertida.keys() ]
16 chaves.sort()
17
18 for i in chaves:
19     print(" ",i," & ",str(lista_invertida[i]), " \\\\")

```

6.3. Tabelas *Hash*

Tabelas *Hash* ou Tabelas de Espalhamento são uma estrutura de dado cuja complexidade de tempo esperado de inserção, busca e remoção é $O(1)$, porém ocupam um espaço que deve ser pré-allocado, logo maior do que o necessário a cada momento.

Uma tabela de espalhamento é um estrutura de dados baseada em um espaço reservado, um arranjo ou vetor, onde o índice é calculado a partir da chave, por meio de uma função específica que pode ser calculada rapidamente, a função de espalhamento (Cormen et al., 2009). O vetor tem um tamanho proporcional ao número esperado de chaves, que devem ser distribuídas de forma pseudo-aleatória no mesmo por meio da função de espalhamento. Se duas chaves tiverem o mesmo *hash*, uma estratégia de tratamento de colisões é usada.

Um caso especial, conhecido como *hash* perfeito, faz com que todas as chaves se distribuam no vetor completo sem colisão, sendo $O(1)$ o tempo de pior caso. Um caso degenerado, todas as chaves vão ser mapeadas na mesma célula do vetor, e tem o tempo $O(n)$, supondo que o tratamento de colisão usa uma lista encadeada ou outra estrutura de busca sequência.

A forma mais simples de função de *hash* é o endereçamento direto. Nesse caso a chave é usada diretamente como o índice do vetor. Se palavras forem mapeadas em um número inteiro representando os primeiros caracteres das mesmas, isso pode implicar em uma vetor que pode ser usado. Os sufixos, porém, podem causar um grande número de colisões.

Usando as duas primeiras chaves do exemplo A como referência, a Tabela 6.2 mostra um passo a passo das inserções das chaves para as duas primeiras frases. É usada uma tabela de endereçamento direto (Cormen et al., 2009) baseada na primeira letra

Tabela 6.2.: Exemplo de inserções em uma tabela de endereçamento direto com endereçamento aberto. Palavras que colidem estão em negrito.

Índice Direto	1 maria	2 comprou	3 um	4 quilo	5 de	Passo e Palavra		8 no	9 mercado	10 vale	11 muito	12 dinheiro
a						6 banana	7 prata					
b						banana	banana	banana	banana	banana	banana	banana
c		comprou	comprou	comprou	comprou	comprou	comprou	comprou	comprou	comprou	comprou	comprou
d					de	de	de	de	de	de	de	de
e												dinheiro
f												
g												
h												
i												
j												
k												
l												
m	maria	maria	maria	maria	maria	maria	maria	maria	maria	maria	maria	maria
n								no	no	no	no	no
o									mercado	mercado	mercado	mercado
p							prata	prata	prata	prata	prata	prata
q				quilo	quilo	quilo	quilo	quilo	quilo	quilo	quilo	quilo
r											muito	muito
s												
t												
u			um	um	um	um	um	um	um	um	um	um
v										vale	vale	vale
x												
z												

da palavra. Como estratégia de tratamento de colisão vamos usar a de pegar a célula vazia seguinte, o que é o tipo mais simples de endereçamento aberto (Cormen et al., 2009). No exemplo da Tabela 6.2 as palavras “dinheiro”, “mercado” e “muito” colidem, respectivamente com “de” e “maria”, e acabam algumas células a frente.

Tabelas de Espalhamento são tratadas detalhadamente no capítulo 11 de Cormen et al. (2009).

Em Python, os dicionários, `dict`, são implementados como tabelas de espalhamento com endereçamento aberto. Ao serem criados, os dicionários possuem 8 slots, e muda de tamanho quanto está com dois terços de sua ocupação⁴. A implementação em CPython tem mais de 5500 linhas⁵.

6.4. Caches em Python

Veja <https://realpython.com/lru-cache-python/>

⁴<https://stackoverflow.com/questions/327311/how-are-pythons-built-in-dictionaries-implemented>

⁵<https://github.com/python/cpython/blob/main/Objects/dictobject.c>

DRAFT

Medidas de Avaliação

Mecanismos de busca ou de aprendizado de máquina podem ser avaliados de várias formas. Para muitos, há um interesse na complexidade do algoritmo, ou na velocidade de execução em certa configuração de computadores. Os algoritmos de processamento de texto, porém, tem como característica principal não serem precisos, nem mesmo quando avaliados por seres humanos diferentes, logo as medidas mais interessantes devem determinar a satisfação do usuário com a resposta.

Mas não é fácil medir, principalmente em experimentos controlados com grande número de dados, a satisfação do usuário. Para isso, então, foi estabelecido que o conceito de relevância é um proxy adequado. O precursor desses estudos foi Cyril Cleverdon, nos Experimentos Cranfield.

Dessa forma foram definidos 3 elementos de um *benchmark*, uma coleção de documentos, uma coleção de consultas e uma avaliação de cada documento da coleção como relevante ou não-relevante para cada consulta. É comum que essa avaliação seja feita por vários especialistas, e que haja alguma discordância entre eles.

Para atender os métodos de avaliação, cada julgamento pode ser binário, isto é, apenas dizer se o documento é relevante ou não, ou usar alguma escala, como dar um valor no conjunto 0, 1, 2. Esse trabalho, porém, não é fácil, porque é necessário julgar toda a coleção, mas é interessante que a coleção seja muito grande.

Existem várias coleções usadas como padrão para experimentos de busca e recuperação. Duas coleções muito usadas atualmente são a NIST-GOV2, com 25 milhões de documentos, e a Reuters-RCV1, com mais de 800 mil documentos.

7.1. Medidas Clássicas de Avaliação

Seja T uma tarefa realizada sobre um conjunto C para a qual se espera como resposta um conjunto A , onde $A \in C$, e seja H_T uma heurística que busca implementar a tarefa T

e responde com um conjunto B , onde $B \in C$, então é possível definir, para P as seguintes medidas de avaliação:

- a **precisão** P , representando a proporção de acertos da heurística (Equação 7.1) em relação ao que foi obtido por ela;
- a **revocação**, do inglês *recall*, R , representando a proporção do que devia ter sido obtido pela heurística (Equação 7.2) pelo que foi obtido por ela;
- a **acurácia** A_c , representando a quantidade de acertos entre o que foi encontrado e também entre o que não devia ser encontrado e não foi (Equação 7.3).

$$P(H_T) = \frac{|A \cap B|}{|B|} \quad (7.1)$$

$$R(H_T) = \frac{|A \cap B|}{|A|} \quad (7.2)$$

$$A_c(H_T) = \frac{|A \cap B| + |(C - A) \cap (C - B)|}{|C|} \quad (7.3)$$

Esses valores também podem ser entendidos por meio da **matriz de confusão** (Tabela 7.1).

Tabela 7.1.: Matriz de Confusão

	Recuperados (B)	Não-Recuperados ($C - B$)
Deviam ser recuperados (A)	Verdadeiros Positivos (VP) $A \cap B$	Falso Negativo (FN) $A \cap (C - B)$
Não Deviam ser recuperados ($C - A$)	Falsos Positivos (FP) $B - A$	Verdadeiros Negativos (VN) $(C - A) \cap (C - B)$

Levando em conta a notação da Tabela 7.1, as fórmulas podem ser reescritas como:

$$P(H_T) = \frac{VP}{VP + FP} \quad (7.4)$$

$$R(H_T) = \frac{VP}{VP + FN} \quad (7.5)$$

$$A_c(H_T) = \frac{VP + VN}{VP + VN + FP + FN} \quad (7.6)$$

Se $B = A$ então $P = R = A_c = 1$. Porém, se $B = C \implies R = 1$, porém $A_c = \frac{A}{C}$ e $P = \frac{A}{C}$, mas esse resultado não tem interesse nenhum. Já $P = 1$ pode acontecer quando B é pequeno, e esse resultado pode ser de algum interesse.

7.1.1. Problemas com a Acurácia

Se o conjunto B é de tamanho muito diferente do conjunto $C - B$, o que é o caso mais comum na tarefa de recuperação da informação, então a acurácia não é uma boa medida, o que exemplificado na matriz de confusão.

Tabela 7.2.: Matriz de Confusão Desbalanceada

	Recuperados	Não-Recuperados
Deviam ser recuperados	VP=1	FN=9
Não Deviam ser recuperados	FP=0	VN=990

$$A_c(H_T) = \frac{1 + 990}{1000} = 0.991 \quad (7.7)$$

Como a revocação é baixa, 10%, mesmo com a precisão alta, esse resultado é insuficiente na maioria dos casos. Mesmo assim a acurácia é alta. Por isso a acurácia não é usada na recuperação da informação, e só tem valor nas tarefas de classificação e agrupamento quando as classes são bem balanceadas.

7.2. F

7.3. Outras Medidas

DRAFT

Parte III.

Recuperação de Texto

DRAFT

Busca e Recuperação

8.1. Motivação

Pessoas precisam de informações, seja por questões ligadas ao trabalho, ao estudo, a diversão ou algum interesse pessoal. As duas fontes de informação existentes são pessoas ou registros criados por essas pessoas, em textos, vídeos, sons, etc.

Este capítulo trata da busca de documentos. Esse sempre foi um problema razoavelmente importante, porém com a explosão da quantidade de informações disponíveis na forma de documentos digitais, sejam eles banco de dados, documentos, páginas web ou outros, a busca passou a ser uma tarefa muito mais importante na sociedade. Onde antes o interesse maior era achar algo relevante sobre uma necessidade da pessoa dentro de uma coleção fechada em um ambiente como uma biblioteca, hoje é um ato diário, por meio do uso do WWW ou de aplicativos.

8.1.1. Escalas do Problema de Busca

Imagine uma criança procurando uma informação em sua biblioteca particular contendo 50 revistas em quadrinhos. Ela deseja saber a origem do Homem Aranha. Seria simples para ela folhear (*browse*) suas 50 revistas e verificar se há alguma informação relevante.

Um passo além, imagine um estudante de Computação chegando à biblioteca de sua universidade em busca de aprender a programar em Python, onde há no total 50 mil livros. Já não faz mais sentido o estudante olhar livro a livro, percorrendo os corredores. Nas bibliotecas, porém, existe várias tecnologias que permitem ao estudante saber melhor qual livro: uma classificação usando um método padronizado (o UDC) e catálogos por assunto, por título, ou por autor. Algumas bibliotecas podem até ter todos seus livros classificados de formas mais sofisticadas, como sistemas computacionais que registram inclusive resumos. Por um catálogo desse tipo, o estudante vai saber que livros de Python estão disponíveis e até sua posição na biblioteca, mas qual o livro que será útil para

ele? O estudante provavelmente ainda vai ter que selecionar alguns e folheá-los de forma a entender sua complexidade, pois pode ser que nada saiba sobre programação, ou ao contrário, que já saiba programar até mesmo em Python e só queira tirar uma dúvida.

Finalmente, a situação mais comum hoje em dia. Nosso usuário da internet busca, na Web, um material sobre empreendedorismo, para descobrir como pode pensar um negócio que deseja abrir. Quantos milhares ou milhões de documentos ou páginas web tratam sobre o assunto, com que profundidade, em que língua¹? Qual desses documentos atende a necessidade do usuário?

8.1.2. Como Atender o Usuário da Busca

É difícil atender o usuário por vários motivos. Alguns deles estão relacionados ao discutido no Capítulo ??, ou seja, basicamente a interpretação dos textos, ou da própria comunicação com o sistema de busca, pelas pessoas.

Outras estão ligadas a tecnologia. Primeiro, é muito difícil compreender automaticamente o que está no documento. Depois a conversa do sistema com o usuário tem que ser simples, a não ser em casos muito especializados.

Outros problemas estão ligados a quantidade de documentos disponíveis. São muito documentos, alguns cópias de outros.

Um outro problema é a qualidade da informação. Mentiras óbvias e sutis.

8.1.3. Uso da Recuperação da Informação no Tempo

Antes dos computadores já existiam livros e catálogos. Bibliotecas existem desde aproximadamente 2750 A.C.(W. G. Stock e M. Stock, 2013). Logo, o problema de achar algo desejado dentro de uma coleção, existe há mais de 4000 anos. Ao longo desse tempo houve alguns marcos:

- **2000 a.c.**, mais antiga lista de documentos, relacionava 62 títulos em tabletes de argila;
- 1300 AC, lista com títulos e informações descritivas;
- 650 AC, lista com 20.000 tabletes de argila, com título, possuidor, numeração e outros detalhes, provavelmente o primeiro catálogo;
- Séc III e II AC, Biblioteca de Alexandria e os Pinakes. Introdução dos conceitos de autor e ordem alfabética, entre outros. 700.000 rolos de papiro. Até hoje a destruição da Biblioteca de Alexandria, quando estava para ser transportada para Roma, é lastimada pelos historiadores;
- Séc VI, São Bento ensina os monges a copiar documentos, tornando-os os primeiros copistas e catalogadores de livros;

¹Uma busca feita por Google com a palavra empreendedorismo em 2 de janeiro de 2021 encontrou 132 milhões de hits.

- Séc VII até o IX, surgem os inventários de livros;
- Entre 822 e 842 surgem catálogos mais organizados, entre eles, em 831, o do mosteiro beneditino de Sait-Requier, na França. Bibliotecas crescem de tamanho, mas ainda na ordem de 600 ou 700 livros.
- Final do Séc. XIII, as bibliotecas monásticas inglesas criam um catálogo coletivo;
- 1389, é criado o primeiro catálogo organizado como tal, no convento de St. Martin, em Dover;
- 1560, Trefler, um monge beneditino, desenvolve um sistema de classificação e localização, e um catálogo de 5 partes.
- Século XVIII, se desenvolve a pesquisa científica e aparece o primeiro código de catalogação nacional, na França, em 1791. Catalogação passa a ser feita em fichas.
- 1839, regras de Pannizzi
- 1876 Cutter publica um código de catalogação com normas;
- 1876 Melvil Dewey inventa o sistema Dewey de classificação, usado até hoje;
- 1895 Paul Otlet e La Fontaine inventam o CDU (Classificação Decimal Universal) com uma notação que permite falar da interseção de assuntos;
- 1830 Classificação facetada
- 1945-55 Infância da Recuperação da Informação
- 1945 Vannecar Bush escreve “As We May Think” e praticamente inventa o hipertexto e a referência (link);
- WRU Searching Selector, primeiro catálogo computadorizado;
- 1960s, estabelecimento da área de Recuperação da Informação, conceituação de Coleção de Texto, Revocação, Precisão, etc...
- 1963, 1990, Ted Nelson inventa o termo Hipertexto e propõe um sistema similar ao WWW;
- 1972, início da DARPA NET;
- 1991, criação do WWW;
- 1994, primeiro *crawler*, WebCrawler, lançado para indexar o WWW, por Brian Pinkerton;
- 1995 Google!

8.2. Principais Problemas

Esta seção apresenta uma pequena revisão dos principais problemas da Recuperação da Informação. Alguns desses problemas são tratados detalhadamente ao longo do livro.

- Tamanho da “Coleção”
 - O tamanho da coleção sempre foi um problema na área de Recuperação de Informações, pois cada evolução dos computadores, ao mesmo tempo que trouxe um aumento na capacidade de processar os documentos, também trouxe um aumento da capacidade dos computadores de guardar documentos. Assim, se os dados iniciais era apenas títulos e resumos de pequenas coleções

de documentos, eles foram evoluindo para grandes coleções, até chegar ao tamanho incontrolável da internet atual, que tudo abarca.

- Simplicidade da consulta
 - Apesar dos seres humanos precisarem de informação sofisticada, dificilmente possuem o conhecimento para produzir consultas sofisticadas, que expressem bem essa necessidade de informação. A maioria dos sistemas de uso genérico, como o Google, tem como desafio buscar a partir de palavras chaves ou de perguntas em linguagens naturais. Sistemas especializados, porém, como sistemas dedicados a busca de precedentes legais, costumam possuir linguagens mais sofisticada que permitem declarar conjunções, negativas, etc.
- Quantidade de documentos que atendem a consulta
 - Sempre um desafio, a quantidade de documentos que atende uma consulta chegou ao número de milhões com os mecanismos de busca na internet. Escolher qual o documento mais adequado, ou colocar os documentos em uma ordem de prioridade, é um problema ainda em aberto.
- Existência de informação de baixa qualidade, desatualizada, incompleta ou intencionalmente falsa
 - Os mecanismos de busca, basicamente, tentam achar documentos que versem sobre o assunto determinado pela consulta. Um questão importante é se os documentos retornados apresentam informações corretas e atualizadas.
- Multiplicidade de línguas
 - Leitores podem ser capazes, ou não, de ler em diversas línguas, o que aumenta a utilidade do mecanismo de buscas. As línguas, porém, apresentam particularidades e dificuldades próprias.
- Múltiplas interpretações do texto, incluindo as literais e as não literais
 - Cada texto pode ser interpretado de várias formas e com várias intenções, porém as palavras do texto normalmente só são reconhecidas por seu significado quando isoladas, não por seus significados não literais
- Usos do texto além da leitura do seu conteúdo, como análise de estilo
 - Alguns usuários dos sistemas de busca não estão interessados na mensagem propriamente dita, mas na forma como ela é expressada.
- Meta-dados
 - Documentos possuem meta dados, como autor, data da publicação, que podem estar ou não explícitos no documento
- Entender o usuário
 - Sendo a finalidade do mecanismo de busca resolver uma necessidade real de informação do usuário, entender o usuário, tanto cognitivamente como psicologicamente, e tanto de forma global quanto no momento exato da pesquisa, é uma tarefa que pode levar a melhorias no desempenho.
- Entender as alterações do estado do usuário ao longo de um processo de buscas de muitos passos
 - Ao fazer a busca o usuário passa por vários estados mentais, e conhecer esses estados permite também melhorar o desempenho do buscador
- Relacionamentos entre os documentos

- Que permite entender mais sobre o significado de cada documento e quando um pode ser útil para entender o outro, ou substituí-lo.

8.3. A Tarefa de Recuperação

Dado um conjunto de documentos, que é chamado de *corpus*, e uma necessidade de informação de um usuário, a tarefa de recuperação consiste em encontrar todos os documentos relevantes a necessidade de informação, possivelmente de forma ordenada, do mais relevante para o menos relevante.

Esta tarefa apresenta um nível de dificuldade muito alto, já que, para o problema geral de recuperar um documento dada uma necessidade de informação, tratado anteriormente, as seguintes condições se aplicam:

- O usuário pode não saber qual a sua verdadeira necessidade de informação;
- Os documentos são representados de uma forma que não indica toda a sua condição de preencher a necessidade de informação;
- O sistema não é capaz de fazer todas as operações necessárias para descobrir o que é realmente relevante para o usuário, e
- A necessidade de informação, e com isso a noção de relevância de um documento, muda ao longo do uso do sistema.

O Paradoxo da Busca

Se o usuário soubesse o que deseja, não precisaria de um mecanismo de busca. Necessariamente, a um estado cognitivo confuso ou incompleto para que uma busca seja necessária.

Formar uma consulta de qualidade é uma das partes mais difíceis para responder a uma **necessidade de informação**.

8.4. Aboutness

Aboutness é um conceito sem uma tradução específica para o português tá tendo sido traduzido para “atinência”, “tematicidade”, “sobrecidade”, “concernência” ou como sinônimo de “assunto”. É usada, ao mesmo tempo, para indicar a determinação pelo assunto, tema ou tópico de um documento, que não seja necessariamente mencionado no título ou no texto, e para indicar sobre o que é o documento, sendo uma questão antiga da Ciência da Informação, e possivelmente a principal pergunta da Ciência da Informação (Guedes, 2009). O tópico também é usado em outras áreas, como a Filosofia.

Não há, por um lado, entre os principais autores da área em Ciência da Informação, uma concordância absoluta com uma definição específica da palavra, porém, por outro lado,

algumas ideias prevalecem, o que permite um entendimento aceitável do significado do termo. Uma delas é que “documentos possuem um assunto intrínseco, inalterável, e outro, ou outros, que podem ser atribuídos (de acordo com a conveniência do sistemas em que esteja inserido ou das necessidades e expectativas de quem o venha consultar)” (Guedes, 2009).

Essa ideia está desenvolvida nos conceitos de *aboutness* extensional e intensional de Fairthorne (1969), ou, respectivamente, de *aboutness* e significado (*meaning*) (Beghtol, 1986). É semelhante a questão tratada na Seção 2.2 sobre as várias interpretações que podem ser dadas ao documento, como discutido por (Meyriat, 1981).

A determinação do assunto então lado identifica o motivo do autor ao escrever o documento, mas estará sempre subordinada aos diferentes elementos em consideração pelo leitor ao fazer a leitura. Cabe então, ao responsável por indexar um documento, criar um bom índice, ou uma boa entrada em um índice para um documento, que forneça uma boa formulação do “aboutness” de um documento (Haliday e Hasan, 1976; Hutchins, 1978), em seus vários momentos de uso. Isso cabe tanto a um humano escolhendo palavras-chave em um vocabulário controlado, quanto a uma máquina criando índices automaticamente.

P. Bruza, Song e Wong (1999) apresenta algumas propriedades básica da *Aboutness*, partindo do conceito de portador da informação (*Information Carrier*) (P. Bruza e T. Huibers, 1996). Por definição, um portador da informação A é dito *sobre* (*about*) um outro portador da informação B se a informação contida em B é válida em A , com a notação $A \models_a B$.

Dado um *framework* $\{IC, \rightarrow, \oplus, \perp\}$, onde IC é um conjunto de portadores de informação, $CI = \{A, B, C, \dots\}$, $A \rightarrow B$ indica que A contém a informação que B carrega, \oplus indica a composição e \perp representa a preclusão (incompatibilidade de assuntos) (P. D. Bruza e T. W. C. Huibers, 1994; P. Bruza e T. Huibers, 1996; P. Bruza, Song e Wong, 1999):

1. Reflexividade: $A \rightarrow A$;
2. Transitividade: $A \rightarrow B, B \rightarrow C \implies A \rightarrow C$
3. Assimetria: $A \rightarrow B \not\Rightarrow B \rightarrow A$;
4. Contenção e Composição: $A \oplus B \rightarrow A; A \oplus B \rightarrow B$
5. Absorção: $A \rightarrow B \implies A \oplus B = A$
6. Não contenção conflituosa: $A \rightarrow B \implies A \not\perp B$
7. Contenção-Preclusão: $A \rightarrow B, B \perp C \implies A \perp B$

Essas sentenças podem ser lidas como: um documento é sobre si mesmo. Se um documento é sobre um segundo documento e este é sobre um terceiro, então o primeiro também é sobre o terceiro. Se um documento é sobre outro não é necessário que o segundo seja sobre o primeiro. Uma composição de documentos é sobre cada documento da composição. Se um documento é sobre outro, então não é incompatível com o outro, e finalmente se um documento é sobre um segundo e esse segundo é incompatível com um terceiro, então o primeiro documento também é incompatível com o terceiro.

Esse framework permite o desenvolvimento de um sistema axiomático e teoremas. Porém, um dos problemas atingidos foi que, dentro de certos parâmetros, *Aboutness* não é monotônica em relação a termos específicos, pois podem ser dados exemplos como “Surfe é sobre ondas” e “Surfe e internet não é sobre ondas”. Isto pode acontecer, por exemplo, porque termos tem acepções diferentes tanto isoladamente quanto em conjunto.

Os trabalhos sobre *Aboutness* são comuns em outras áreas que não a Recuperação de Informação, como a Filosofia, porém tentativas de encontrar uma Teoria da *Aboutness* (P. D. Bruza e T. W. C. Huibers, 1994; P. Bruza, Song e Wong, 1999) que pudesse ser usada de maneira eficaz, ou seu uso em sistemas de recuperação, não parecem ter evoluído na literatura.

Apesar de a *Aboutness* de um documento, em sua concepção mais ampla, parecer ser a norma da indexação, a verdadeira questão não é se o documento é sobre algo, mas sim se interessa ao futuro leitor. Esse outro conceito é conhecido como relevância.

Em todo caso, é importante notar que toda representação de um documento busca representar sobre o que ele é, isto é, sua *aboutness*.

8.5. Relevância

A palavra relevância pode ser usada como um termo não definido, pois todos possuem uma ideia do que significa (Saracevic, 2017b). Porém, para ser possível discuti-la, é melhor desenvolver um entendimento do que significa, e mesmo uma definição. van Rijsbergen (1979, pg. 4) deixa claro que a relevância é estabelecida por humanos, e sistemas computacionais precisam ter um modelo do que é relevância de forma a quantificá-la.

Algo é relevante caso se enquadre no que o buscador está querendo, mas não necessariamente a pergunta feita. Para uma mesma pergunta, um documento pode ser importante para uma pessoa e não ser para outra. Imagine um biólogo e um fã de automobilismo pesquisando por “Jaguar”: os documentos que consideram relevantes serão provavelmente diferentes. Além disso, um documento pode ou não ser relevante por ser conhecido ou não do usuário.

Borlund (2000) afirma que a relevância, ou não, de um documento, entre vários retornados em uma consulta a um sistema de busca de informações, pode mudar após o usuário ler um outro documento. Por exemplo, buscando conhecer um evento histórico, digamos que a Revolução dos Cravos em Portugal, após ler um texto, um segundo texto pode não trazer mais nenhuma informação adicional, se tornando não-relevante.

Em slides de uma aula proferida na ESSIR 2003, (van Rijsbergen, 2003) cita alguns autores, sem dar a referência completa, que trazem as seguintes afirmações:

- Goffman (1969) diz que “a relevância da informação em um documento depende do que já se sabe sobre o assunto, e afeta, em seu turno, a relevância dos documentos examinados a seguir”;

- Maron² diz que “apenas por um documento ser sobre o assunto buscado pelo cliente, não quer dizer que ele o julgará relevante”, e
- Borlund (2000)³ “a relevância, ou irrelevância, de um documento recuperado pode afetar o estado atual de conhecimento do usuário, resultando na mudança de sua necessidade de informação, o que pode levar à mudança das percepções e interpretações, pelos usuário, dos documentos recuperados a seguir”.

Já Raber (2003) cita uma definição de (van Rijsbergen, 1990): “A medida ou grau de correspondência ou utilidade existente entre um texto ou documento e uma consulta ou requisito de informação determinada por uma pessoa”, que é bastante objetiva, e chama atenção para o fato que a relevância é sempre em função de uma pessoa, que possui um estado cognitivo específico em um certo momento, e pode se referir tanto ao texto desejado como a um requisito de informação.

Um definição razoavelmente formal, mas genérica, de relevância foi dada por (Saracevic, 2017b), em um artigo seminal na área: “A relevância é o **A** de um **B** existindo entre um **C** e um **D** como determinado por um **E**”, sendo que:

- **A** pode ser “medida, grau, estimativa,...”;
- **B** pode ser “correspondência, utilidade, encaixe, ...”;
- **C** pode ser “documento, informação fornecida, fato, ...”;
- **D** pode ser “consulta, pedido, requisito de informação, ...”, e
- **E** pode ser “usuário, juiz, especialista de informação, ...”(Saracevic, 1975).

É possível perceber que a definição de Saracevic aceita, em seu formato, a definição de van Rijsbergen, o que é, de fato, intencional.

8.5.1. Atributos da relevância

Mais modernamente, (Saracevic, 2017b) faz um sumário dos atributos da relevância:

- **Relação**, a relevância aparece expressando uma relação, frequentemente em comunicações que envolvem pessoas, informações ou objetos de informação;
- **Intenção**, a relação de relevância envolve motivação e intenções, além de objetivos, papéis e expectativas;
- **Contexto**, a intenção sempre vem em um contexto, sendo direcionada para ele, havendo um contexto interno, ligado a estados cognitivos e afetivos, e um contexto externo, direcionado para situações, tarefas ou problemas, podendo ainda envolver componentes sociais e culturais;
- **Inferência**, essa relação precisa ser avaliada, sendo criada nessa base;
- **Seleção**, a inferência pode envolver uma seleção entre fontes que competem em direção a maximização dos resultados, e a minimização do esforço de trabalhar com eles;
- **Interação**, inferência é obtida em um processo dinâmico, interativo onde a interpretação dos outros atributos pode mudar, quando o contexto muda, e

²Não foi possível identificar esta citação

³Esta publicação é provavelmente sua tese de doutorado.

- **Medição (Mensuração)**, a relevância envolve uma avaliação (graduada) da efetividade ou grau de maximização de um relação dada para uma intenção direcionada a um contexto(Saracevic, 2017b).

Isto é, a relevância é uma relação, entre pessoas e objetos de informação, que ocorre dentro de um contexto, interno e externo, onde são estabelecidas intenções que permitem inferências, onde são atribuídas medidas, que levam a seleção entre fontes de informação por meio de interações.

Além disso, Saracevic (2017b) ainda afirma que a relevância pode ou não estar conectada com a verdade⁴.

8.5.2. Manifestações de Relevância

Saracevic (2017a) faz um sumário das manifestações da relevância, de acordo com vários autores:

- **Sistêmica ou Algorítmica**
 - É a relação entre a consulta e a informação (ou objetos de informação) no sistema, como recuperados ou não recuperados, por um algoritmo ou procedimento dado.
 - O critério é a efetividade comparada ao inferir a relevância.
 - Cada sistema tem seus meios de representar, organizar e comparar os objetos com a consulta.
 - A intenção é recuperar um conjunto de objetos que o sistema inferiu (construiu) como sendo relevante para uma consulta.
- **Tópica**
 - É a relação entre tópico (assunto) expresso em uma consulta e o tópico coberto pela informação (objeto de ifnormação).
 - *Aboutness* é o critério sobre o qual a topicalidade é inferida, porém a teoria do *aboutness* não continuou a progredir.
 - Assume-se que consulta e objeto de informação podem ser associados a um tópico.
- **Cognitiva**
 - É a relação entre o estado cognitivo do usuário e a informação (objeto de informação)
 - Os critérios de inferência são: correspondência cognitiva, quantidade de informação, (informativeness) novidade, qualidade da informação, etc.
- **Situacional**
 - É a relação entre a situação/tarefa ou problema e os objetos de informação (recuperados, arquivados ou existentes).
 - Os critérios de inferência são: utilidade na tomada de decisão, propriedade da informação, redução da incerteza (ser apropriada, não ownership).
 - Pode envolver fatores sociais e culturais.

⁴O que parece relevante em tempos de *fake-news*

- **Afetiva**

- É a relação entre as intenções, objetivos, emoções e motivações do usuário e os objetos de informação (recuperados, arquivados ou existentes).
- Os critérios de inferência são: satisfação, sucesso, realização.
- É razoável defender que todas as outras manifestações de relevância estão sujeitas a afetiva, principalmente a situacional.

No mesmo livro, Saracevic ainda reapresenta um modelo estratificado da relevância (Saracevic, 2007), adaptado na Figura 8.1, inserido no contexto da Recuperação da Informação, que considera camadas diferentes interagindo, integrando diversos pontos de vista. O modelo é dividido em duas grandes partes, o usuário e o computador, que se comunicam interativamente por meio de interface (Saracevic, 2017b).

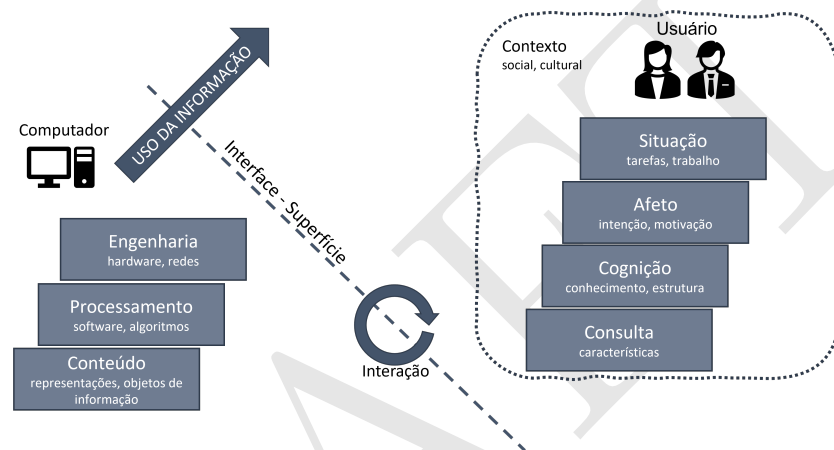


Figura 8.1.: Modelo estratificado da relevância, baseado em Saracevic (2017b).

Considerando os conceitos de Necessidade Real de Informação (RIN) e de Necessidade Percebida de Informação (PIN), propostos por (Mizzaro, 1998), a Figura 8.2 descreve o seguinte quadro sobre a relevância: ao realizar uma tarefa o usuário tem uma Necessidade Real de Informação, para a qual ele não é capaz, dado que falta informação, de perceber precisamente, havendo então uma Necessidade Percebida de Informação. Para poder fazer a consulta, então, ele imagina uma forma de expressar essa PIN, na forma de tópicos em que está interessado. Essa expressão é então traduzida por ele para a linguagem de consulta. É essa consulta que vai permitir ao sistema, por meio de algoritmos, determinar uma resposta, que deve atender os tópicos, tendo uma relevância cognitiva e situacional. O usuário, então, analisa a resposta, o que mudará principalmente o seu PIN, e possivelmente mesmo a sua RIN, iniciando um novo ciclo da interação.

8.5.3. O Espaço das Relevâncias

Mizzaro (1998) apresenta a ideia de que existem várias relevâncias e que podem ser definidos um conjunto de espaços de relevâncias, em quatro dimensões, que descreve como a relevância deve ser entendida ao longo de um processo de busca de informações:

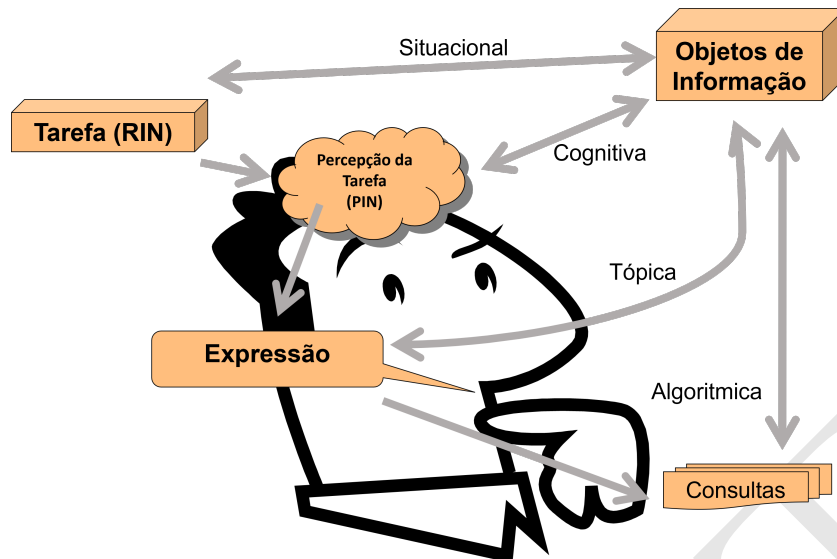


Figura 8.2.: Tipos de relevância, baseado em Saracevic (2017b).

1. Recursos de Informação, que são encontrados e avaliados, como documentos, substitutos de documentos (*surrogates*), e a informação;
2. Representação do Problema do Usuário, que são usados para fazer a busca, que possui uma necessidade real (RIN), uma necessidade percebida (PIN), a representação mental da RIN, um pedido ou solicitação (*request*), em linguagem natural, e uma consulta, em uma linguagem específica;
3. o tempo, e
4. Componentes, que são o tópico, a tarefa e o contexto.

A RIN se transforma em PIN por um processo de percepção. Já a PIN se transforma em um pedido por um processo de expressão. Finalmente, o pedido se transforma em consulta por um processo de formalização (Mizzaro, 1998). Todos esses processos estão sujeitos a erros, ruídos e incertezas.

Há uma grande dificuldade em perceber a real necessidade de informação, já que se busca algo que não se sabe, ou seja, enquanto há um vazio cognitivo, a pessoa está com uma imagem incompleta do mundo, o que é conhecido como *Anomalous State of Knowledge* (ASK) (Belkin, N. e Brooks, 1982), ou *Incomplete State of Knowledge* (ISK), *Uncertain State of Knowledge* (USK), ou seja, um Estado de Conhecimento Anômalo, Incompleto ou Incerto (Mizzaro, 1998).

Já a expressão sofre do problema do descasamento entre o vocabulário usado no pedido e os termos do documento. Pode haver ambiguidade, sinonímia, homonímia, e outras relações que interferem na facilidade de relacionar informação e expressão. Além disso, o uso de rótulos, e não expressões completas, também dificulta esse processo (Mizzaro, 1998).

Os recursos de informação e a representação do problema podem ser vistos como se associando em pares. Essas associações vão evoluindo pelo tempo, em função do usuário estar realizando a busca dentro de um, ou mais, dos 3 componentes.

Assim a relevância pode ser vista como acontecendo, e evoluindo no tempo dentro de 3 tipos de recursos, 4 tipos de representação e 7 combinações dos Componentes (Mizzaro, 1998). Essa evolução sempre começa em um tempo inicial, com a RIN, que vai evoluindo até a construção da consulta, que pode se repetir várias vezes. A análise das repostas a consulta pode alterar tanto a RIN, quanto a PIN, quanto a expressão, quanto a consulta, causando uma evolução do espaços.

1. RecInfo = { *Surrogate*, Documento, Informação }
2. Repr = { Consulta, Expressão, PIN, RIN }
3. Comp = { {Tópico} {Tarefa}, {Contexto}, {Tópico, Tarefa}, {Tópico, Contexto}, {Tarefa, Contexto}, {Tópico, Tarefa, Contexto} }
4. Tempo = { $t(rin_0), t(pin_0), t(r_0), t(q_0), t(q_1)$ }

Mizzaro (1998) tenta representar dimensões em diagramas que indicam como o processo de busca de uma informação pode ser visto nessas várias dimensões.

Mizzaro (1998) ainda chama a atenção que o julgamento de relevância é feito por um juiz, dentro de um tipo de relevância, mas também por um tipo de juiz, que pode ou não ser usuário do sistema de busca, como pode ter disponível um ou mais tipos de recursos e também várias formas de representar seu julgamento, tudo isso caracterizado em um momento do tempo.

8.6. Modelo Conceitual da Recuperação da Informação

Fuhr (1992) propõe um modelo conceitual, ilustrado na Figura 8.3, para a Recuperação de Informação: dado um conjunto finito de documentos \underline{D} , e um conjunto finito de consultas \underline{Q} , em um certo momento, para certa pessoa, existe um conjunto finito R de julgamentos de relevância, $R : \underline{D} \times \underline{Q} \rightarrow \mathbb{R}$.

Um sistema de recuperação da informação faz esse mapeamento segundo um certo algoritmo. Porém, o sistema trabalha sobre representações D e Q dos documentos e consultas, o que exige dois mapeamentos α_D e α_Q , de documentos em representações de documentos, e de consultas em representações de consultas. Por exemplo, no modelo vetorial, documentos são representados como vetores de termos.

Genericamente falando, não são as representações que são processadas pelo algoritmo, mas sim suas descrições, implementações, por exemplo, um vetor pode ser implementado diretamente ou por uma lista encadeada, se for esparso. Para chegar a essas descrições, D' e Q' , são necessários outros mapeamentos, β_D e β_Q , e delas é obtida um **valor de status de recuperação**, \mathfrak{R} , que é resultado da aplicação de um algoritmo nas descrições das representações dos documentos, e consultas, e logo uma aproximação, ou estimativa,

do julgamento de relevância. Mais tarde, Baeza-Yates e Ribeiro-Neto (2011a) removem um nível do modelo, mostrando apenas documento e representação.

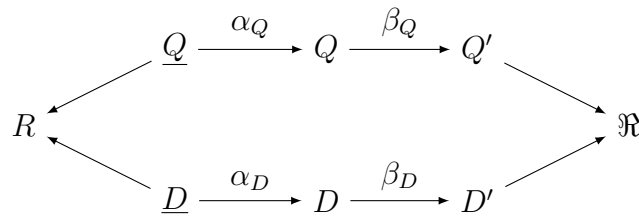


Figura 8.3.: Modelo Conceitual da Recuperação da Informação, segundo (Fuhr, 1992).

8.7. Palavras-Chave e *Full Text Indexing*

A maneira mais simples de representar sobre o que trata um texto é por um conjunto de palavras que, de acordo com uma ou mais perspectivas, expliquem sobre o que o texto fala.

A primeira abordagem que vem a tona é baseada na prática dos bibliotecários: o uso de palavras-chave, *keywords* ou *index terms*, cuidadosamente escolhidas e provavelmente controladas por um glossário, taxonomia ou ontologia. Essas palavras não precisam pertencer ao documento. Por exemplo, um texto jornalístico sobre o Fluminense⁵ poderia receber as palavras-chave “tricolor” e “pó-de-arroz”, mesmo que elas não fossem usadas no artigo.

O uso de palavras-chave tem como foco sobre o que é o documento, sua *aboutness*.

A segunda abordagem, que exige um poder computacional maior, porém exequível hoje em dia, é indexar todas as palavras do documento, com a premissa que elas tragam o significado do documento junto com elas, no que é chamado *full-text indexing*.

⁵O Fluminense é um time de futebol do Rio de Janeiro cuja bandeira tem três cores e a torcida usa tradicionalmente uma nuvem de pó-de-arroz.

DRAFT

Representação Booleana

A maneira mais simples de representar sobre o que trata um texto é por um conjunto de palavras que, de acordo com uma ou mais perspectivas, expliquem sobre o que o texto fala.

A primeira abordagem que vem a tona é baseada na prática dos bibliotecários: o uso de palavras chave cuidadosamente escolhidas e provavelmente controladas por um glossário.

A segunda abordagem, que exige um poder computacional maior, porém em geral exequível hoje em dia, é indexar todas as palavras do documento, com a premissa que elas tragam o significado do documento junto com elas, no que é chamado *full-text indexing*.

Essa premissa não é óbvia. Por exemplo, um pesquisador em busca de livros escritos com um estilo específico, como o literatura barroca, provavelmente pouco se aproveitará das palavras que realmente representam seu desejo, como “barroco”, e terá que buscar palavras esperadas na literatura desse tipo. Já um pesquisador usando um mecanismo de palavras chaves escolhidas certamente encontrará, entre elas, a descrição do estilo do documento que procura.

Levando isso em consideração, esse capítulo trata de como representar documentos por meio de palavras da forma mais simples, isto é, considerando apenas se a palavra está ou não presente na descrição do documento. Essa descrição, que não precisa ser única, pode ser na forma de um conjunto de palavras-chave escolhidas intencionalmente, ou pela coleção de palavras do documento, o que é conhecido como *full-text* na literatura, provavelmente pré-processadas de alguma forma, como explicado no Capítulo 5.

Essa representação, quando relativa ao texto completo do documento, é conhecida também como *bag of words*.

9.1. Modelo Booleano

Esse modelo foi definido *a posteriori* do seu uso. A utores importantes da área, que explicam o modelo, não registram sua invenção, como Baeza-Yates e Ribeiro-Neto (2011b), ou mesmo o Salton e McGill (1983a), que fala do modelo como “Sistemas Baseados em Arquivos Invertidos”, hoje conhecidos como listas invertidas ou índices invertidos. Cooper (1997) afirma que a proposta de usar consultas baseadas em expressões booleanas de descritores de documentos é da década de 50.

O Modelo Booleano (Baeza-Yates e Ribeiro-Neto, 2011a) é baseado na Teoria dos Conjuntos e na Lógica Booleana. Ele parte da ideia que as consultas q são expressões booleanas, onde cada termo representa se uma palavra-chave está ou não presente no documento, ou em uma consulta. Como em $q = k_a \wedge (k_b \vee \neg k_c)$, onde $k_i \in \{0,1\}$. Uma vantagem do Modelo Booleano é sua semântica precisa, o que permite também a definição formal.

Formalmente, seja um conjunto de termos $K = k_1, k_2, \dots, k_T$ e um conjunto de representações de documentos $D = d_1, d_2, \dots, d_N, d_i \in \wp(K)$, uma consulta Q tem a forma de uma expressão booleana formada com os termos k_j e os operadores E , OU e NO , onde k_j indica que o termo k_j deve pertencer a representação do documento desejado, e $NO(k_j)$ indica que o termo não deve pertencer ao documento (Baeza-Yates e Ribeiro-Neto, 2011a).

Consultas típicas então seriam “FLUMINENSE”, “FLUMINENSE E FLAMENGO” ou “FLUMINENSE OU TRICOLOR”. Uma consulta mais complexa poderia ser “(FLUMINENSE OU TRICOLOR) E (FLAMENGO E RUBRO-NEGRO) E NÃO(TRIBUNAL)”.

A resposta a uma consulta $R(Q)$ é a lista de documentos que atende aos requisitos da consulta. Ou seja, a resposta é $\{x|x \in D \wedge R(x) = 1\}$. Dessa forma os documentos são relevantes ou não, não havendo um grau de relevância.

Um maneira de entender um sistema baseado no Modelo Booleano é que ele faz um mapeamento de um conjunto de documento $D = d_1, d_2, \dots, d_N$ em um conjunto de vetores em $\{0,1\}^T$, onde T é a quantidade de termos. Dessa maneira, cada documento pode ser visto como descrito por um vetor de 0s e 1s de tamanho fixo. Isso é equivalente a representar uma consulta como a Forma Normal Disjuntiva (Baeza-Yates e Ribeiro-Neto, 2011a), da seguinte maneira, dada uma consulta q :

$$\begin{aligned} q &= k_a \wedge (k_b \vee \neg k_c) \\ q_{dnf} &= (k_a \wedge k_b \wedge k_c) \vee (k_a \wedge k_b \wedge \neg k_c) \vee (k_a \wedge \neg k_b \wedge \neg k_c) \end{aligned}$$

E cada termo k_i pode ser representado como um vetor, no caso com 3 dimensões apenas:

$$q_{dbf} = (1,1,1) \vee (1,1,0) \vee (1,0,0)$$

onde cada q_{cc} , como $(1,0,0)$ é um componente conjuntivo.

Além disso, considera-se que existe uma função $g_i(x)$, que retorna o peso associado ao termo k_i em x , que pode ser um documento ou uma consulta. No modelo booleano, $g_i(x) \in 0,1$ (Baeza-Yates e Ribeiro-Neto, 2011b).

A similaridade entre a consulta q e um documento d_j é:

$$\text{sim}(d_j, q) = \begin{cases} 1, & \text{if } \exists \vec{q}_{cc} | (\vec{q}_{cc} \in \vec{q}_{fnd}) \wedge (\forall k_i, g_i(\vec{d}_j) = g_i(\vec{q}_{cc})) \\ 0, & \text{otherwise} \end{cases} \quad (9.1)$$

O que podemos compreender dessa formalização é que o sistema vai buscar documentos que possuam uma representação exatamente igual a de um dos componentes conjuntivos.

9.2. Usando a Lista Invertida

Como visto na Seção 6.1, uma lista invertida é, de uma forma bem geral, uma estrutura de dados onde as partes permitem recuperar o todo. É chamada invertida porque normalmente do todo se recuperam as partes.

Então se um documento d_i é descrito pelas palavras-chave $k_i^1, k_i^2, \dots, k_i^n$, a lista invertida permitirá que, conhecido um k_i^j , o documento d_i seja encontrado.

Uma lista-invertida pode ser implementada como uma lista encadeada, porém em Python e em memória, a maneira mais simples é implementá-la como um *dictionary*.

Assim, supondo que temos um conjunto de 3 documentos d_1, d_2, d_3 , onde $d_1 = k_1, k_2$, $d_2 = k_1, k_3$ e $d_3 = k_2, k_3$, a lista-invertida poderia ser representada como: `li = { 'k1': set('d1', 'd2'), 'k2': set('d1', 'd3'), 'k3': set('d2', 'd3') }`.

Essa lista invertida é construída com conjuntos (*sets*) para que seja possível aproveitar as funções de união e interseção.

Outras representações são possíveis e possivelmente mais rápidas, e são tratadas ao longo deste capítulo.

9.3. Processo básico de construção das listas invertidas

O processo básico de construção das listas invertidas parte da Matriz de Incidência ou Matriz Termo-Documento, que indica, para cada termo, qual o seu peso em cada documento, sendo que no caso do Modelo Booleano, esse peso é apenas 0 ou 1, caso não apareça ou apareça no documento, respectivamente.

Seja um conjunto de documentos com a Matriz de Incidência apresentada na Tabela 9.1. A matriz apresenta 4 termos e 5 documentos, sendo que o documento d_1 e d_5 possuem a mesma representação.

Tabela 9.1.: Matriz Termo-Documento, ou Matriz de Incidência, simples.

	Documentos				
Termos	d_1	d_2	d_3	d_4	d_5
k_1	1	1	0	0	1
k_2	0	1	1	0	0
k_3	1	0	1	0	1
k_4	0	0	0	1	0

9.4. Consultas booleanas

Uma consulta típica a um sistema de palavras-chaves é a consulta booleana. Esse tipo de consulta foi, e ainda é, muito usado em sistemas de biblioteca e permite operações rápidas.

Usando o exemplo da Tabela 9.1, suponha a consulta $(k_1 \wedge k_2) \vee k_4$. Esse modelo leva ao dbf $(1,1,0,0) \vee (1,1,0,1) \vee (1,1,1,1) \vee (1,0,0,1) \vee (1,0,1,1) \vee (0,1,0,1) \vee (0,1,1,1) \vee (0,0,0,1) \vee (0,0,1,1)$. Esses vetores devem ser comparados com as colunas da matriz. A resposta é d_2, d_4 .

9.5. Simulando o Modelo Booleano

O Programa 9.1 simula um corpus indexado por uma lista invertida, com capacidade de fazer operações de interseção e união, dadas as palavras chave.

Programa 9.1: Programa simples para fazer consultas do tipo E e OU a uma lista-invertida

```

1 class Corpus:
2
3     def __init__(self, dict):
4         self.dict = dict.copy()
5
6     def processa_e(self, chaves, current = None):
7         if not current:
8             current = set()
9
10        for k in chaves:
11            if current:
12                current &= self.dict[k]
13            else:
14                current = self.dict[k].copy()
15        return current
16

```

```

17 def processa_ou(self ,chaves, current = None):
18     if not current:
19         current = set()
20     for k in chaves:
21         current |= self.dict[k]
22     return(current)
23
24
25 if __name__ == '__main__':
26
27     corpus = Corpus({ 'k1' : {'d1', 'd2', 'd5'}, 'k2' : {'d2', 'd3'},
28                       'k3' : {'d1', 'd3', 'd5'}, 'k4' : {'d4'}
29     })
30
31     # (k1 e k2) ou k4
32     exp1 = corpus.processa_e(['k1','k2'])
33     corpus.processa_ou(['k4'],exp1)
34     print(exp1)

```

Saída do Programa 9.1

1 {'d2', 'd4'}

9.6. Estratégias para busca em longas listas invertidas

- Iniciar pelos “E”, que diminuem o tamanho dos resultados intermediários;
- Iniciar pelas chaves com menos documentos;
- Usar skip pointers;

9.7. Exemplos com o Modelo Booleano

Para exemplificar o funcionamento do Modelo Booleano, é apresentada uma implementação simples em memória construída em Python, com apoio de textos selecionados do corpus Guthemberg do NLTK. Os textos estão em inglês, sendo: Sense and Sensibility de Jane Austen; Julius Caesar, Hamlet e Macbeth, de Shakespeare; Moby Dick, de Herman Melville e Alice in Wonderland, de Lewis Carrol. Nos exemplos usaremos termos da língua inglesa, mas preferencialmente nomes de personagens que sabemos estar, ou não, nas obras: Hamlet, Macbeth, Caesar, Antony, Cleopatra, John, Ishamel, Elinor, Marianne e Alice.

Deve ficar claro que todos esses programas se aproveitam de estruturas e funções de Python que funcionam bem em memória, porém não escalam para o mundo real, onde a quantidade de documentos varia de milhares em um biblioteca para milhões na Internet.

Um sistema real, em vez de um dicionário, teria que utilizar a memória em disco e uma estrutura de dados de arquivos.

O programa da Listagem 9.2 que prepara na memória os seis documentos e dez termos do nosso exemplo.

Programa 9.2: Preparação do Exemplo.

```
1 import nltk
2 nltk.download('gutenberg')
3 docs=[0]*6 # Vetor de Documentos
4
5 # Escolhe 6 documentos entre os fornecidos
6 docs[0]=nltk.corpus.gutenberg.words('austen-sense.txt')
7 docs[1]=nltk.corpus.gutenberg.words('shakespeare-caesar.txt')
8 docs[2]=nltk.corpus.gutenberg.words('shakespeare-hamlet.txt')
9 docs[3]=nltk.corpus.gutenberg.words('shakespeare-macbeth.txt')
10 docs[4]=nltk.corpus.gutenberg.words('melville-moby_dick.txt')
11 docs[5]=nltk.corpus.gutenberg.words('carroll-alice.txt')
12
13 # Define os termos
14 words = ["Hamlet" ,
15         "Macbeth",
16         "Caesar",
17         "Antony",
18         "Cleopatra",
19         "John",
20         "Ishmael",
21         "Elinor",
22         "Marianne",
23         "Alice"
24         ]
25 words.sort()
26
27
28 mtd = [0]*len(words) # Matriz Termo Documentos
29 for i in range(len(words)):
30     mtd[i] = [0]*len(docs)
```

O programa da Listagem 9.3 é uma abordagem simples feita em Python para gerar um matriz termo documento, com a especificidade que a palavra sendo indexada foi escolhida antes e está listada na variável `words`.

Programa 9.3: Cálculo da uma matriz termo documento.

```
1 # Verifica se os termos estão presentes nos documentos
2 # e preenche a matriz termo documentos
```

```

3
4 for docn in range(len(docs)):
5     for wordn in range(len(words)):
6         if words[wordn] in docs[docn]:
7             mtd[wordn][docn]=1
8
9 # imprime para verificação
10
11 print("    DOCS-> ", "0 1 2 3 4 5")
12 for i in range(len(words)):
13     print("%10s  "%words[i],end='')
14     for j in range(len(docs)):
15         print(mtd[i][j],end=' ')
16     print()

```

O resultado principal da execução desse programa é a matriz termo documento:

	Resultado da execução do Programa 9.3						
	DOCS->	0	1	2	3	4	5
1	Alice	0	0	0	0	0	1
2	Antony	0	1	0	0	1	0
3	Caesar	0	1	1	1	1	0
4	Cleopatra	0	0	0	0	1	0
5	Elinor	1	0	0	0	0	0
6	Hamlet	1	0	1	0	0	0
7	Ishmael	0	0	0	0	1	0
8	John	1	0	0	0	1	0
9	Macbeth	0	0	0	1	0	0
10	Marianne	1	0	0	0	0	0

Nas coleções no mundo real, a matriz termo documento é muito grande e esparsa. Isso implica que em vez de representá-la como matriz ela é normalmente representada como uma lista invertida.

O programa de Listagem 9.3 utiliza de algumas facilidades permitidas por tudo caber em memória e algumas funções já estarem implementadas em Python. Um programa mais parecido com um programa de indexação, ainda todo em memória, usaria o laço da Listagem 9.4, onde é necessário passar pelo documento palavra por palavra para criar a lista invertida.

Programa 9.4: Leitura de um documento palavra a palavra e registro dos documentos que possuem os termos índices em uma “lista invertida” construída por meio de um dicionário.

```

1 lis_inver = {}
2
3 for docn in range(len(docs)):

```

```

4  for term in docs[docn]:
5      if term in words:
6          if docn not in lis_inver.get(term, []):
7              lis_inver[term] = lis_inver.get(term, [])+[docn]
8
9  for key in lis_inver:
10     print("%10s ->"%key, end=" ")
11     print(lis_inver[key])

```

O resultado do programa da Listagem 9.4 é:

Resultado da execução do Programa 9.4

```

1      John -> [0, 4]
2      Elinor -> [0]
3      Marianne -> [0]
4      Hamlet -> [0, 2]
5      Caesar -> [1, 2, 3, 4]
6      Antony -> [1, 4]
7      Macbeth -> [3]
8      Ishmael -> [4]
9      Cleopatra -> [4]
10     Alice -> [5]

```

Em nossos exemplos, como em muitos sistemas reais, não é possível aplicar o **NÃO** em uma só palavra, sendo ele um operador binário que equivale a diferença de conjuntos.

Para realizar as funções lógicas E, OU e **NÃO**, são necessárias funções específicas se queremos fazer o processo nós mesmos. Considerando que as listas estão ordenadas pelo número do arquivo na Listagem 9.5 são apresentadas: a função **pandq**, que calcula a conjunção de duas listas; a função **pnotb**, que calcula a lista de elementos de a que não estão em b, e a função **porb**, que calcula a lista da união das listas a e b.

Programa 9.5: Calcula operadores E, OU e **NÃO** para duas listas.

```

1  def pandq(p, q):
2      ans = []
3      while p and q:
4          if p[0] == q[0]:
5              ans.append(p[0])
6              p = p[1:]
7          elif p[0] < q[0]:
8              p = p[1:]
9          else:
10             q = q[1:]
11     return ans
12
13  def pnotq(p, q):

```

```

14  ans = []
15  while p and q:
16      if p[0] == q[0]:
17          p = p[1:]
18          q = q[1:]
19      elif p[0]<q[0]:
20          ans.append(p[0])
21          p = p[1:]
22      else:
23          q = q[1:]
24  if p:
25      ans.extend(p)
26  return ans
27
28  def porq(p,q):
29      ans = []
30      while p and q:
31          if p[0] == q[0]:
32              ans.append(p[0])
33              p = p[1:]
34              q = q[1:]
35          elif p[0]<q[0]:
36              ans.append(p[0])
37              p = p[1:]
38          else:
39              ans.append(q[0])
40              q = q[1:]
41  if p:
42      ans.extend(p)
43  if q:
44      ans.extend(q)
45  return ans

```

Python fornece funções mais eficiente para todas essas opções, mas o objetivo do código apresentado neste livro é demonstrar como o algoritmo utilizado, mesmo que algumas necessidades de sistemas reais sejam deixadas de fora.

Um maneira fácil de interpretar uma expressão booleana tradicional, na notação infixa é transforma-la para uma notação pós-fixa que pode ser imediatamente executada. Nesse caso uma expressão lógica (A NOT B) E C NOT (D OU NOT(F)) é traduzida para a lista ['A', 'B', 'NOT', 'C', 'D', 'F', 'NOT', 'OU', 'NOT', 'E']. O Programa da Listagem 9.6 faz isso.

Programa 9.6: Transforma um expressão lógica infixa em pós-fixa.

```

1 prioridade = {"(" : 10 , ")" : 10 , "NÃO" : 9 ,
2               "NOT" : 9 , "E" : 8 , "AND" : 8,
3               "OU" : 7 , "OR" : 7 , "+" : 7 ,
4               "." : 8 , "~" : 9,
5               "*" : 8 , "-" : 9}
6
7
8 def i2p(expr):
9     pilha = []
10    saida = []
11    # conserta espaços faltando
12    entrada = expr.replace("(", " ( ").replace(")", " ) ").split()
13    for token in entrada:
14        pt = prioridade.get(token, 0)
15        if pt == 0: # if operando, output
16            saida.append(token)
17        elif token == "(":
18            pilha.append(token)
19        elif token == ")":
20            while pilha and pilha[-1] != "(":
21                saida.append(pilha.pop())
22            pilha.pop()
23        elif not pilha: # if stack empty
24            pilha.append(token)
25        elif pt > prioridade.get(pilha[-1], 0):
26            pilha.append(token)
27        elif "(" in pilha[-1]: # if top stack is (
28            pilha.append(token)
29        else:
30            while pilha and prioridade[pilha[-1]] >= pt and pilha[-1] != "(":
31                saida.append(pilha.pop())
32            pilha.append(token)
33    while pilha :
34        saida.append(pilha.pop())
35    return(saida)

```

O programa da Listagem 9.7 processa uma expressão de consulta lógica de memória sem usar estruturas de índice, já que Python possui o operador `in`.

Programa 9.7: Processa uma busca booleana na coleção em memória usando o documento original.

```

1 def processa_booleano(expr, docs):
2     consulta = i2p(expr)
3     pilha = []

```



```

4 ops = prioridade.keys()
5 for tk in consulta:
6     if tk not in ops:
7         found = set()
8         for docn in range(len(docs)):
9             if tk in docs[docn]:
10                 found.add(docn)
11                 pilha.append(found)
12     if tk in ["E", "*", ".", "AND"]:
13         pilha.append(pilha.pop().intersection(pilha.pop()))
14     if tk in ["OU", "+", "OR"]:
15         pilha.append(pilha.pop().union(pilha.pop()))
16     if tk in ["NOT", "~", "-", "NÃO"]:
17         buf = pilha.pop()
18         pilha.append(pilha.pop().difference(buf))
19 return(pilha.pop())

```

Já o programa da Listagem 9.8 processa uma expressão de consulta lógica de memória usando as estruturas de índice e comparando-as por meio das funções mostradas na Listagem 9.5.

Programa 9.8: Processa uma busca booleana na coleção em memória usando índice.

```

1 def processa_booleano(expr, docs):
2     consulta = i2p(expr)
3     pilha = []
4     ops = prioridade.keys()
5     for tk in consulta:
6         if tk not in ops:
7             found = set()
8             for docn in range(len(docs)):
9                 if tk in docs[docn]:
10                     found.add(docn)
11                     pilha.append(found)
12         if tk in ["E", "*", ".", "AND"]:
13             pilha.append(pilha.pop().intersection(pilha.pop()))
14         if tk in ["OU", "+", "OR"]:
15             pilha.append(pilha.pop().union(pilha.pop()))
16         if tk in ["NOT", "~", "-", "NÃO"]:
17             buf = pilha.pop()
18             pilha.append(pilha.pop().difference(buf))
19     return(pilha.pop())

```

9.8. Exercícios

Exercício 9.1:

Implemente um programa similar ao implementado nesse capítulo, porém usando estruturas persistentes, como as citadas na Seção 6.1.

DRAFT

Representação Vetorial

No Capítulo 9, foi mostrada uma alternativa de representação para o Modelo Booleano onde os documentos são mapeados em vetores cujos pesos são sempre 0 ou 1. Como o Modelo Booleano não permitia o cálculo de um ranking, isto é, todas as respostas aparecem com igual importância, vários autores buscaram alternativas, como o Modelo Probabilístico (Robertson, 1977; van Rijsbergen, 1979), Modelos Booleanos Estendidos, e outros (Baeza-Yates e Ribeiro-Neto, 2011a). Um dos modelos de maior sucesso foi o Modelo Vetorial (Salton e McGill, 1983b), que até hoje influencia toda a área de Recuperação da Informação e de Aprendizado de Máquina.

No Modelo Vetorial, cada documento é representado por um vetor em que cada elemento representa o peso de um termo específico no documento, sendo que a noção de peso não é definida *a priori*, sendo apenas um número real, e existindo várias formas de calculá-lo. Também não é feita nenhuma exigência na ordem dos termos, porém é necessário que todos os elementos de cada posição sejam associados aos mesmos termos em todos os vetores. Nesse caso, são calculadas similaridades entre documentos ou entre documentos e consultas por meio de funções entre dois vetores, que são propostas para atender certas propriedades.

O peso é uma medida da importância do termo no documento. A ideia básica é que o vetor de termos de um documento indique sobre o que ele fala (*aboutness*).

Medidas simples são a própria frequência ou a frequência relativa do termo, porém as medidas de maior sucesso partiram do conceito conhecido como *tf-idf*, *term frequency-inverse document frequency*, que considera a importância do termo dentro do documento e a importância do termo como discriminador de documentos na coleção. Assim, um termo que aparece em todos os documentos não tem importância, sendo seu peso próximo de zero.

A Figura 10.1 mostra a ideia básica do Modelo Vetorial: os termos presentes no corpus definem um espaço vetorial, e indicam o significado dos documentos, e da consulta. Além disso, o modelo considera que a similaridade entre a consulta e os documentos da coleção

pode ser calculada dentro desse espaço, por exemplo como o inverso da distância, e mais frequentemente como uma distância angular entre os vetores, possuindo gradações.

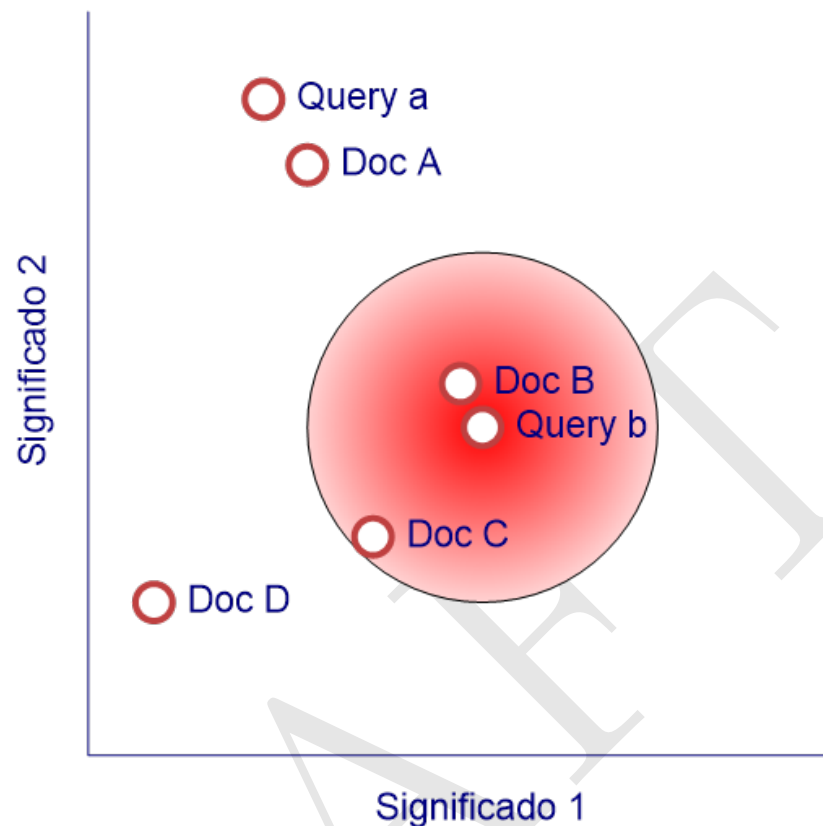


Figura 10.1.: A ideia básica do Modelo Vetorial é que os termos do documento, e da consulta, indicam seu significado, e que a similaridade entre a consulta e os documentos da coleção pode ser calculada em um espaço vetorial.

Algumas premissas do Modelo Vetorial original não são verdade. A principal é que as dimensões dos vetores são ortonormais. Como as dimensões são geradas pelos termos, isso significa que os termos seriam independentes entre si, ou seja, em última instância, que o fato de um termo aparecer em um documento é independente do fato de qualquer outro termo aparecer no documento. Isso, obviamente, não é verdade. Alguns termos costumam aparecer mais frequentemente juntos, o que é chamado de co-ocorrência, do que outros. Isso foi tratado em modelos subsequentes.

A primeira vista, a similaridade no Modelo Vetorial poderia ser vista como o inverso da distância entre os vetores de frequência. Isso, porém, acaba por não funcionar bem, porque o que importa no significado de um documento não deve ser o peso de um outro termo, mas sim a relação entre os pesos do termo. Ao usar o inverso da distância, poderiam ocorrer situações como a demonstrada na Figura 10.2, onde uma consulta q seria mais próxima dos documentos d_2 e d_3 , porém a proporção entre os pesos dos termos mais semelhante a q é a do documento d_1 .

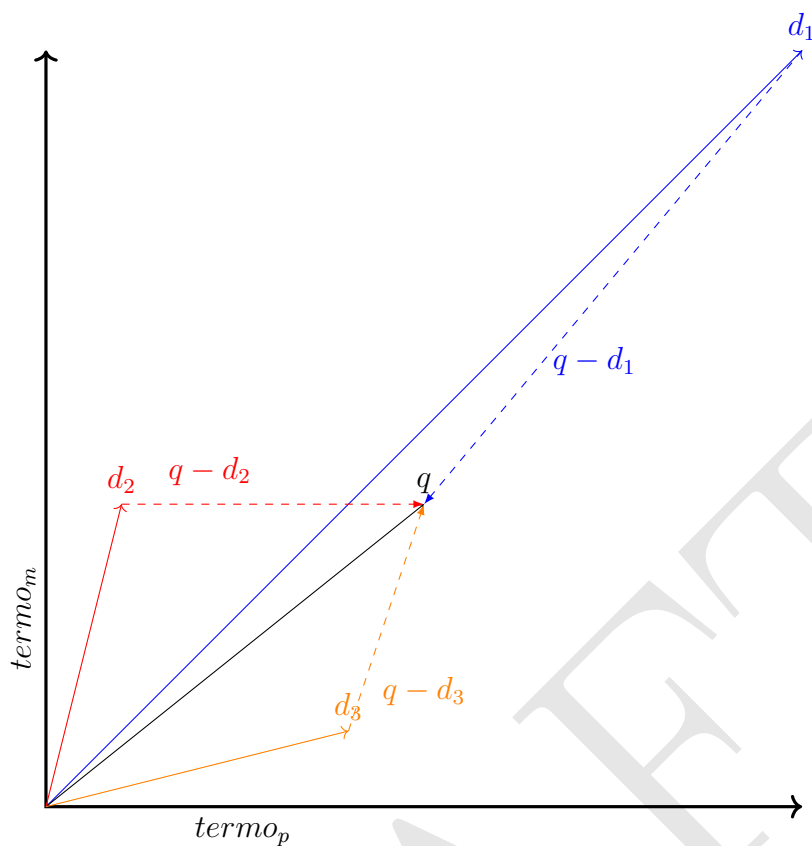


Figura 10.2.: Exemplo de problema com o uso do tamanho dos vetores

Por isso, a medida mais simples e razoável para o Modelo Vetorial é relacionada ao ângulo entre dois vetores, e na prática se usa o cosseno do ângulo, que é máximo quando o ângulo é zero. Na Figura 10.3, é possível ver que $\beta < \theta < \alpha$.

Atualmente fórmulas mais complexas são utilizadas, de forma a melhorar o desempenho do sistema, por exemplo corrigindo os efeitos do tamanho do documento, ou usando conceitos dos modelos probabilísticos.

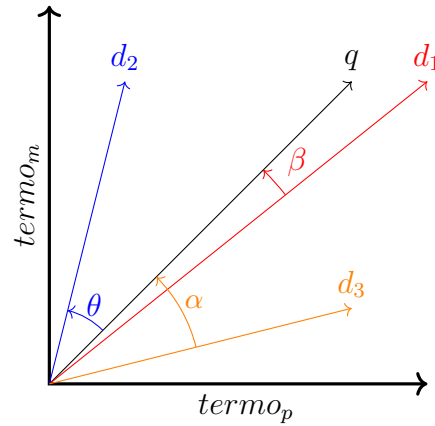


Figura 10.3.: Usando o coseno dos vetores.

10.1. Definições do Modelo Vetorial

Seja um conjunto D de N documentos d_i , ou corpus, $D = \{d_1, d_2, \dots, d_N\}$, compostos de T termos k_j onde cada documento é representado na forma $d_i = (w_{i1}, w_{i2}, \dots, w_{iT})$, onde w_{ij} representa o peso do j -ésimo termo no i -ésimo documento i . No Modelo Vetorial, $w_{ij} > 0$ sempre que $k_i \in d_j$ (Baeza-Yates e Ribeiro-Neto, 2011b).

Para cada termo k_j é associado um vetor unitário $\vec{k}_j = (0, 0, \dots, 1, \dots, 0)$ onde todos os valores são 0, menos o da j -ésima posição, que é 1. Por definição, esses vetores são considerados ortonormais, o que supõe que os termos ocorrem de forma independente nos documentos. Esta é uma decisão importante, que levará a tratamentos mais sofisticados que corrigem essa suposição.

Também é possível representar um documento como um dicionário em Python: $d = \{t_0 : w_{d0}, t_1 : w_{d1}, \dots, t_T : w_{dT}\}$.

As seguintes afirmações são válidas:

- Ambos documento $\vec{d}_i = (w_{i0}, w_{i1}, \dots, w_{iT})$ e consulta $\vec{q} = (w_{q0}, w_{q1}, \dots, w_{qT})$ são elementos do espaço euclidiano $E^n, n = T + 1$.
- Cada termo t_i corresponde a um vetor base \vec{e} do espaço E^n .
- O grau de relevância r de um documento D representado por um vetor \vec{w} relativo a consulta Q representada pelo vetor \vec{q} é baseado no produto escalar $\langle \vec{w}, \vec{q} \rangle$, e em geral pode ser considerado uma medida de similaridade $\text{sim}(\vec{d}, \vec{q})$.
- Se $\langle \vec{w}, \vec{q} \rangle = 0$, então o documento não é relevante e não é recuperado.
- Se $\langle \vec{w}, \vec{q} \rangle \neq 0$, então o documento é considerado relevante e é recuperado.
- Seja $\langle \vec{w}_i, \vec{q} \rangle \neq 0, i = 1, \dots, m$ correspondente aos documentos d_i . Então os documentos d_1, \dots, d_m são usados para construir a lista de acertos: d_1, \dots, d_m são ordenados decrescentemente em ordem do grau de relevância e são apresentados ao usuário nessa ordem (Baeza-Yates e Ribeiro-Neto, 2011b).

id	Documento	Ordem dos Termos	Vetores de Frequência
1	O Pedro viu o homem.	o, pedro, viu, ho- mem	1: (2,1,1,1)
2	O homem era velho.	o, pedro, viu, homem, era, velho	1: (2,1,1,1,0,0) 2: (1,0,0,1,1,1)
3	Pedro tinha um binóculo.	o, pedro, viu, homem, era, velho, tinha, um, binóculo	1: (2,1,1,1,0,0,0,0,0) 2: (1,0,0,1,1,1,0,0,0) 3: (0,1,0,0,0,0,1,1,1)

Tabela 10.1.: Exemplo de criação passo a passo dos vetores de documentos de uma coleção.

É importante notar que o Modelo Vetorial não supõe ordem nas palavras, e a ordem escolhida no vetor pode ser arbitrária (Baeza-Yates e Ribeiro-Neto, 2011b). O importante é que todos os vetores tenham a mesma ordem. Normalmente, em *full-text retrieval*, os termos são adicionados a estrutura de dados usada para a criação dos vetores na primeira vez em que são encontrados. A Tabela 10.1 mostra um exemplo onde 3 documentos, frases no caso, são inseridos em ordem e o vetor vai mudando de tamanho para considerar as novas palavras. Cada palavra é inserida na ordem em que é encontrada. Na prática são criadas estruturas de dados temporárias, como uma árvore binária ou uma tabela de dispersão, para acelerar a busca pelos termos na contagem, e depois processá-las para gerar os vetores.

10.2. Medindo o peso de um termo em um documento

O peso de um termo em um documento pode ser medido de várias formas, que foram evoluindo ao longo do tempo.

A primeira medida que pode ser pensada é a frequência do termo. Essa medida é conhecida com $TF(i,j)$ e possui várias versões. Sendo D uma coleção de documento d_j e K uma coleção de termos k_i , $f_{i,j}$ é a frequência do termo k_i no documento d_j , várias definições são encontradas na literatura, como (Baeza-Yates e Ribeiro-Neto, 2011a;

Roelleke, 2013):

$$TF_{total}(i,j) = f_{i,j} \quad (10.1)$$

$$TF_{soma}(i,j) = \frac{f_{i,j}}{\sum_k f_{k,j}} \quad (10.2)$$

$$TF_{max}(i,j) = \frac{f_{i,j}}{\max_k f_{k,j}} \quad (10.3)$$

$$TF_{log}(i,j) = \begin{cases} 1 + \log_2(f_{i,j}) & , \text{ if } f_{i,j} > 0 \\ 0 & , \text{ caso contrário} \end{cases} \quad (10.4)$$

$$TF_K(i,j) = \frac{f_{i,j}}{f_{i,j} + K} \quad (10.5)$$

$$TF_{dupla}(i,j) = K + K \frac{f_{i,j}}{\max_i f_{i,j}} \quad (10.6)$$

Como foi observado antes neste capítulo, o fato de um termo ser importante em um documento não ajuda a definir a relevância se ele aparecer em todos os documentos. Para medir a importância de um termo como indicador de relevância dentro de uma coleção, foi definido o conceito de Frequência Inversa de Documento, conhecida pela sigla *IDF*. Seja N o número de documentos e n_i o número de documento com o termo i , várias fórmulas também foram propostas na literatura, como (Baeza-Yates e Ribeiro-Neto, 2011a; Roelleke, 2013):

$$IDF_{unrio} = 1 \quad (10.7)$$

$$IDF_{total}(i) = \frac{1}{n_i} \quad (10.8)$$

$$IDF_{soma}(i) = \log \frac{N}{n_i} \quad (10.9)$$

$$IDF_{suave}(i) = \log 1 + \frac{N}{n_i} \quad (10.10)$$

$$IDF_{max}(i) = \log 1 + \frac{max_i n_i}{n_i} \quad (10.11)$$

$$IDF_{BIR}(i) = \log \frac{N - n_i}{n_i} \quad (10.12)$$

$$IDF_{BIRsuave}(i) = \log \frac{N - n_i + K}{n_i + K} \quad (10.13)$$

Combinando TF e IDF vários esquemas podem ser obtidos, como por exemplo, juntado a Equação 10.4 com a Equação 10.9 (Baeza-Yates e Ribeiro-Neto, 2011a):

$$TF-IDF(i,j) = (1 + \log f_{i,j} \times \log \frac{N}{n_i}) \quad (10.14)$$

10.3. Medindo a Similaridade entre Documentos

O coseno do ângulo é uma medida de similaridade normalizada. Dados dois vetores d_1 e d_2 , o coseno é medido por:

$$\cos(\angle(d_1, d_2)) = \frac{d_1 \cdot d_2}{\|w_1\|_2 \|w_2\|_2} = \frac{\sum_{i=0}^T w_{1i} \times w_{2i}}{\sum_{i=0}^T w_{1i}^2 \sum_{i=0}^T w_{2i}^2} \quad (10.15)$$

10.3.1. BM25

$$\mathcal{B}_{i,j} = \frac{(K_1 + 1)f_{i,j}}{K_1 \left[(1 - b) + b \frac{\text{len}(d_j)}{\text{avg_doclen}} \right] + f_{i,j}} \quad (10.16)$$

$$\text{sim}_{BM25}(d_j, q) \sim \sum_{k_i[q, d_j]} \mathcal{B}_{i,j} \times \log \left(\frac{N - n_i + 0.5}{n_i + 0.5} \right) \quad (10.17)$$

DRAFT

Softwares para Full-Text Search

11.1. Lucene

Lucene, uma “biblioteca Java que fornece mecanismos poderosos de indexação e busca” (Apache Project, 2011a) é o padrão *de-facto* para mecanismos de busca *full-text*. É o *engine* que faz funcionar dois outros produtos muito utilizados na indústria, o Solr e o Elasticsearch.

11.1.1. PyLucene

PyLucene (Apache Project, 2011b) é um conjunto de *bindings* para Lucene em Python, feito em C.

O principal defeito de PyLucene é a enorme dificuldade de instalação. Consegui instalar apenas no Ubuntu, depois de horas de tentativas, para ele passar nos testes, mas não consegui usá-lo!

11.2. Solr

11.3. Elastic Search

11.4. Bancos de Dados com Capacidade de *Full-Text Search*

11.5. Software Nativo em Python

Modelo Probabilístico

Ao contrário dos outros modelos clássicos, o Modelo Probabilístico não é focado na representação, mas sim na forma como é calculada a relevância de um documento.

A abordagem principal, que inspira também o *feedback de relevância*, é que a probabilidade de um documento ser relevante pode ser calculada iterativa e interativamente mente, a partir da avaliação do usuário sobre os documentos apresentados em uma primeira consulta, ou sobre o próprio resultado inicial de uma comparação entre a consulta e todos os documentos da base, buscando aumentar a probabilidade de um documento ser relevante na resposta.

DRAFT

Modelos de Linguagem

DRAFT

DRAFT

Indexação por Semântica Latente

A Indexação por **Semântica Latente** (**LSI**¹), e a Análise por Semântica Latente, (**LSA**²) são baseadas em um método vetorial de representação de texto que visa, por meio de manipulações algébricas na Matriz Termo Documento, encontrar tópicos que representem o significado dos documentos a partir de combinações dos termos.

LSI pode ser visto apenas como uma manipulação algébrica que busca encontrar uma aproximação da matriz de termos e documento de forma que:

- sejam filtrados os ruídos causado por termos,
- seja economizado espaço na busca,
- seja economizado tempo na busca.

Para que haja economia de tempo e espaço é necessário que a dimensão da matriz aproximada seja muito menor que a da matriz original.

De forma abstrata, o objeto é mudar de um mapeamento termo-documento, como na Figura 14.1a, para uma mapeamento dos termos para conceitos e dos conceitos para documentos, como na Figura 14.1b.

¹Do inglês *Latent Semantic Indexing*

²Do inglês *Latent Semantic Analysis*

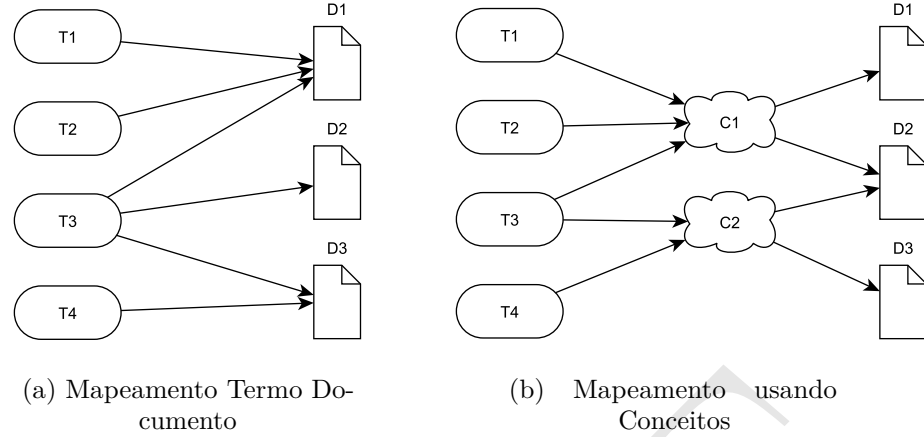


Figura 14.1.: Visão abstrata do LSI.

14.1. A Decomposição em Valores Singulares

Uma **decomposição em valores singulares** de uma matriz $\mathbf{A}_{t \times d}$, conhecida pela sigla **SVD**, é definida pela equação (Axler, 2015):

$$\mathbf{A}_{t \times d} = \mathbf{U}_{t \times m} \mathbf{\Sigma}_{m \times m} \mathbf{V}_{m \times d}^T = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (14.1)$$

Onde $m = \min(t, d)$, \mathbf{U} e \mathbf{V} possuem colunas ortonormais e $\mathbf{\Sigma}$ é diagonal. A matriz $\mathbf{\Sigma}$ é chamada de matriz dos valores singulares. Essa decomposição está esquematizada na Figura 14.2.

Uma aproximação de $\mathbf{A}_{t \times d}$ pode ser obtida escolhendo os r maiores valores singulares de **Sigma** e cortando as colunas e linhas equivalentes em \mathbf{U} e \mathbf{V}^T , como esquematizado na Figura 14.3.

Figura 14.2.: Esquema do SVD

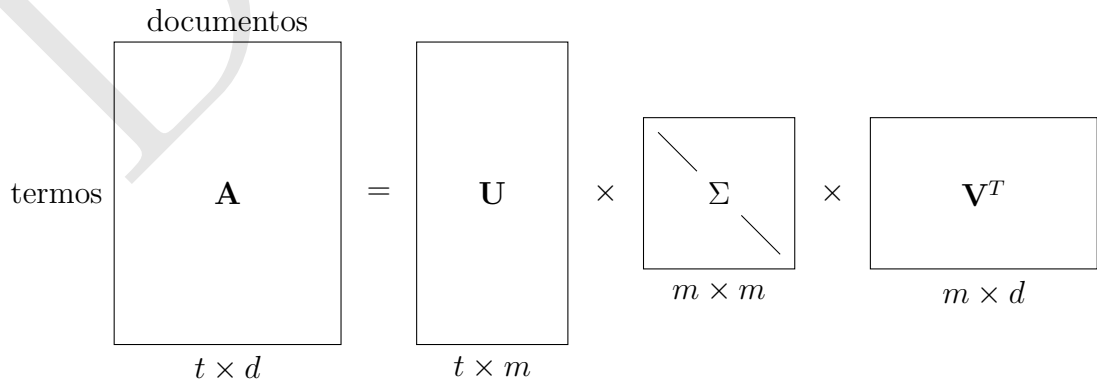
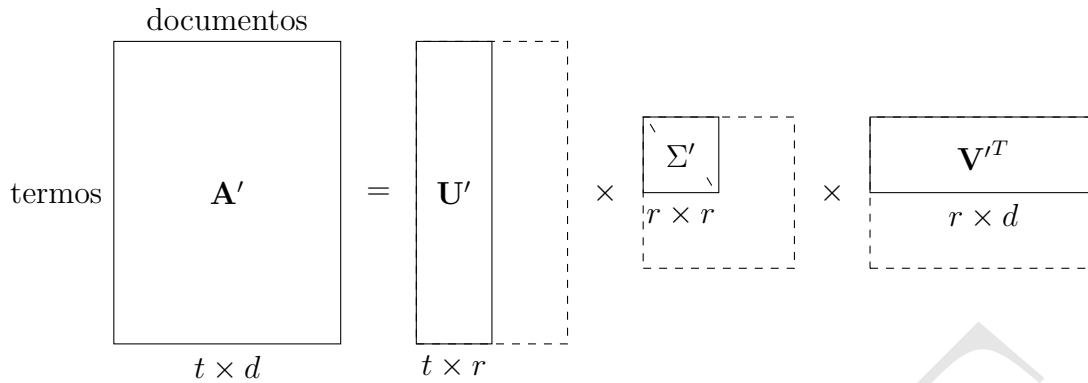


Figura 14.3.: Esquema do SVD após a redução de *rank*

14.2. Exemplo de Uso da LSI em Python

O exemplo que se segue é adaptado de um exemplo teórico de Grossman e Frieder (2004). Aqui é mostrada uma implementação em Python usando apenas o `numpy`, mas outros pacotes podem ser usados. Os Programas de 14.1 a 14.6 compõem na verdade um só programa.

Seja uma base composta de três documentos:

- d_1 Carregamento de outro destruído em um fogo.
- d_2 Entrega de prata chegou em um caminhão prata.
- d_3 Carregamento de outro chegou em um caminhão.

Nessa base será feita uma consulta q : ouro prata caminhão.

O Programa 14.1 realiza a preparação dos dados para fazer uma Indexação por Semântica Latente. Apesar de existirem outras bibliotecas que possam ajudar, vamos iniciar apenas com a `numpy` (linha 1). A `base` é uma lista que contém três sentenças na forma de *strings*.

A LSI ainda é método de similaridade de vetores que representam os documentos e a consulta, onde os vetores são uma modificação dos vetores originais do Modelo Vetorial. Assim, nas linhas 11 e 12, é definida uma função de similaridade do cosseno.

Além disso, na linha 14, é feita uma tokenização de forma muito simples. Como os exemplos (artificiais) não possuem sinais de pontuação, foi usada apenas a função `split`. Deve-se notar que como `base` é uma lista de *strings*, então aplicamos a função `split` a todos os elementos da lista, por meio da função `map`.

Programa 14.1: Declarações

```
1 import numpy as np #
2 np.set_printoptions(precision=2)
3
4
```

```

5 base = ["Carregamento de ouro destruído em um fogo",
6         "Entrega de prata chegou em um caminhão prata",
7         "Carregamento de ouro chegou em um caminhão"]
8
9 q = ["ouro", "prata", "caminhão"]
10
11 def cos_sim(A,B): #
12     return A.dot(B)[0]/(np.linalg.norm(A)*np.linalg.norm(B)) #
13
14 base_tokens = map(str.split,base) #

```

Continuando, agora com o Programa 14.2, é necessário construir a matriz termo documento. Como estamos usando um biblioteca numérica, e para facilitar o mapeamento entre as palavras e os números de linha, criamos dois dicionários, um onde dado uma palavra se obtém um número, e outro inverso. Como é um problema simples, já sabemos o tamanho exato da matriz, que chamaremos de **A**, na linha 19.

Entre a linha 21 e a linha 28 é feito o preenchimento correto da matriz termo documento. Já nas linhas 30 a 32 é criado o vetor de consulta **Q**. Ambos são impressos a seguir.

Programa 14.2: Construindo Termo Documento

```

15 d = 0
16 k = 0
17 numero = {}
18 palavras = {}
19 A = np.zeros((11,3),int)
20
21 for frase in base_tokens:
22     for token in frase:
23         if token not in numero:
24             numero[token] = k
25             palavras[k] = token
26             k += 1
27         A[numero[token],d] += 1
28     d += 1
29
30 Q = np.zeros((11,1),int)
31 for palavra in q:
32     Q[numero[palavra],0] = 1
33
34 print(A)
35 print(Q)

```

O SVD é feito no Programa 14.3, entre algumas impressões que mostram o que está sendo realizado. É importante chamar atenção para o fato que esse operação sofisticada

é feita em apenas uma linha (44). Tendo em vista que a fórmula do SVD é $A = U\Sigma V^T$, a função `svd` do subpacote de álgebra linear responde com o V transposto. Para obter o V original da forma, é feita a operação de transposição na linha 45.

Programa 14.3: Fazendo o SVD

```

36 for i in range(len(A)):
37     print("{0:13}  {1}".format(palavras[i],A[i]))
38
39 print("---*30)
40
41 for i in range(len(Q)):
42     print("{0:13}  {1}".format(palavras[i],Q[i]))
43
44 U, S, VT = np.linalg.svd(A,full_matrices=False) # svd!
45 V = VT.transpose() # transpõe VT para achar V
46
47 print("---*15+\"U\"+\"---*15)
48 print(U)
49 print("---*15+\"S\"+\"---*15)
50 print(S)
51 print("---*15+\"V\"+\"---*15)
52 print(VT)
53 print("---*15+\"VT\"+\"---*15)
54 print(V)

```

Feito o SVD, no Programa 14.4 é feita a diminuição dos ranks das matrizes, de forma a atender a aglutinação de palavras em conceitos proposta pelo SVD. É importante notar aqui que as operações podem ser realizadas apenas com operadores muito comuns em Python, sobre as matrizes retornadas pelo `svd`.

Na chamada a função `svd` foi usada a forma reduzida de U . Além disso, a resposta S não é uma matriz diagonal, mas um vetor de valores singulares. Para poder reconstruir A é necessário usar a função `diag`.

Programa 14.4: Diminuindo o rank

```

55 A_reconstructed = U @ np.diag(S) @ VT
56 print("---*15+\"Arec\"+\"---*15)
57 print(A_reconstructed)
58
59 m, n = A.shape
60 r = 2
61 U_r = U[:, :r]
62 VT_r = VT[:r, :]
63 V_r = VT_r.transpose()
64 S_r = S[:r]

```

```

65 A_r = U_r @ np.diag(S_r) @ VT_r
66 print("—" * 15 + "Ar" + "—" * 15)
67 print(A_r)
68
69 print("—" * 15 + "U_r" + "—" * 15)
70 print(U_r)
71 print("—" * 15 + "S_r" + "—" * 15)
72 print(S_r)
73 print("—" * 15 + "V_r" + "—" * 15)
74 print(V_r)
75 print("—" * 15 + "VT_r" + "—" * 15)
76 print(VT_r)

```

Para fazer a consulta é necessário fazer uma modificação no vetor de consulta para atender as novas dimensões das matrizes reduzidas, isso é feito com a fórmula $q_r = q^T U_r \Sigma_r^{-1}$, como pode ser visto no Programa 14.5.

No mesmo programa são então calculadas as similaridades. Segundo o exemplo, as similaridades estão ordenadas como $d_2 > d_3 > d_1$.

Programa 14.5: Fazendo a consulta

```

77 QLSI = Q.transpose() @ U_r @ np.linalg.inv(np.diag(S_r))
78 print("—" * 15 + "QLSI" + "—" * 15)
79 print(QLSI)
80
81 sim = [0,0,0]
82 for d in range(3):
83     sim[d] = cos_sim(QLSI, V_r[d])
84     print(f"Query {q} Document — {d} Similarity={sim[d]}")

```

A Programa 14.6 mostra todos os vetores reduzidos, em duas dimensões, e é possível verificar a ordenação das similaridades de forma visual. Esse gráfico foi gerado com o Figura 14.4

Programa 14.6: Plot dos vetores

```

85 import matplotlib.pyplot as plt
86
87 def plotvecs(M, labels=[], colors=[], do_axis=True, **kwargs):
88     rows, cols = M.shape
89     if not colors:
90         colors = ["black"] * rows
91
92     for i in range(rows):
93         xs = M[i, 0]
94         ys = M[i, 1]

```

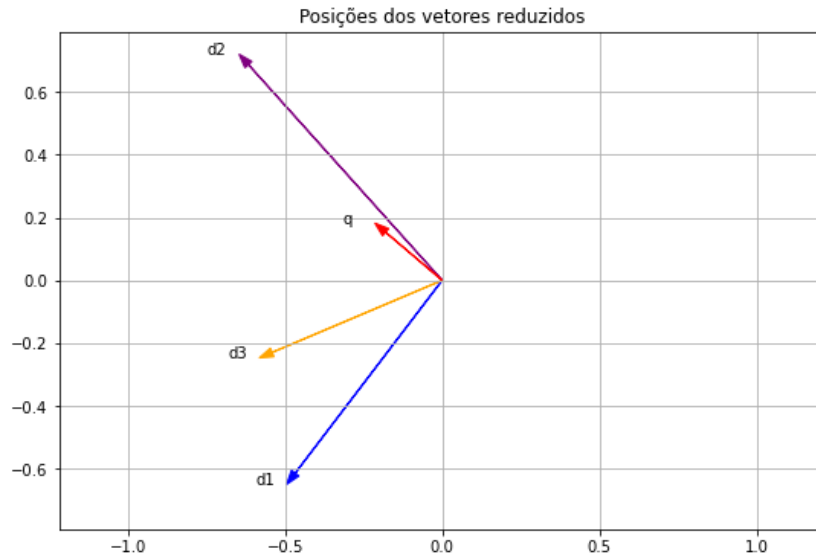


Figura 14.4.: Vetores reduzidos do exemplo.

```

95     plt.arrow(0,0,xs,ys,
96               length_includes_head=True,
97               head_width=.03,
98               color=colors[i],**kwargs)
99     if labels:
100         plt.text(xs-.1,ys,labels[i])
101
102     if do_axis:
103         maxesx = 1.1*abs(M[:,0]).max()
104         maxesy = 1.1*abs(M[:,1]).max()
105         plt.axis('equal')
106         plt.xlim([-maxesx,maxesx])
107         plt.ylim([-maxesy,maxesy])
108     plt.grid(visible=True, which='major')
109     return plt
110
111
112 plotvecs(V_r,["d"+str(i+1) for i in range(len(V_r)) ],
113          ["blue","purple","orange"])
114 plotvecs(QLSI,["q"],["red"],do_axis=False)
115 plt.title("Posições dos vetores reduzidos")
116 plt.show()

```

A seguir, a saída da execução do programa completo.

Saída do programa completo de LSI

```

1 [[1 0 1]
2  [1 1 1]
3  [1 0 1]
4  [1 0 0]
5  [1 1 1]
6  [1 1 1]
7  [1 0 0]
8  [0 1 0]
9  [0 2 0]
10 [0 1 1]
11 [0 1 1]]
12 [[0]
13  [0]
14  [1]
15  [0]
16  [0]
17  [0]
18  [0]
19  [0]
20  [1]
21  [0]
22  [1]]
23 Carregamento [1 0 1]
24 de             [1 1 1]
25 ouro          [1 0 1]
26 destruído     [1 0 0]
27 em            [1 1 1]
28 um            [1 1 1]
29 fogo          [1 0 0]
30 Entrega       [0 1 0]
31 prata         [0 2 0]
32 chegou       [0 1 1]
33 caminhão     [0 1 1]
34 -----
35 Carregamento [0]
36 de            [0]
37 ouro          [1]
38 destruído     [0]
39 em            [0]
40 um            [0]
41 fogo          [0]
42 Entrega       [0]
43 prata         [1]
44 chegou       [0]
```



```

45 caminhão      [1]
46 -----U-----
47 [[-0.26 -0.38 -0.15]
48 [-0.42 -0.07  0.05]
49 [-0.26 -0.38 -0.15]
50 [-0.12 -0.27  0.45]
51 [-0.42 -0.07  0.05]
52 [-0.42 -0.07  0.05]
53 [-0.12 -0.27  0.45]
54 [-0.16  0.3   0.2 ]
55 [-0.32  0.61  0.4 ]
56 [-0.3   0.2  -0.41]
57 [-0.3   0.2  -0.41]]
58 -----S-----
59 [4.1  2.36 1.27]
60 -----V-----
61 [[-0.49 -0.65 -0.58]
62 [-0.65  0.72 -0.25]
63 [ 0.58  0.26 -0.77]]
64 -----VT-----
65 [[-0.49 -0.65  0.58]
66 [-0.65  0.72  0.26]
67 [-0.58 -0.25 -0.77]]
68 -----Arec-----
69 [[ 1.00e+00 -7.49e-16  1.00e+00]
70 [ 1.00e+00  1.00e+00  1.00e+00]
71 [ 1.00e+00 -7.15e-16  1.00e+00]
72 [ 1.00e+00 -5.55e-16  1.44e-15]
73 [ 1.00e+00  1.00e+00  1.00e+00]
74 [ 1.00e+00  1.00e+00  1.00e+00]
75 [ 1.00e+00 -5.55e-16  1.44e-15]
76 [ 7.22e-16  1.00e+00 -1.28e-15]
77 [ 1.44e-15  2.00e+00 -2.55e-15]
78 [ 7.77e-16  1.00e+00  1.00e+00]
79 [ 7.77e-16  1.00e+00  1.00e+00]]
80 -----Ar-----
81 [[ 1.11  0.05  0.85]
82 [ 0.97  0.99  1.05]
83 [ 1.11  0.05  0.85]
84 [ 0.67 -0.15  0.45]
85 [ 0.97  0.99  1.05]
86 [ 0.97  0.99  1.05]
87 [ 0.67 -0.15  0.45]
88 [-0.15  0.93  0.2 ]

```

```

89  [-0.3   1.87  0.4 ]
90  [ 0.3   1.13  0.6 ]
91  [ 0.3   1.13  0.6 ]]
92  -----U_r-----
93  [[-0.26 -0.38]
94  [-0.42 -0.07]
95  [-0.26 -0.38]
96  [-0.12 -0.27]
97  [-0.42 -0.07]
98  [-0.42 -0.07]
99  [-0.12 -0.27]
100 [-0.16  0.3 ]
101 [-0.32  0.61]
102 [-0.3   0.2 ]
103 [-0.3   0.2 ]]
104 -----S_r-----
105 [4.1  2.36]
106 -----V_r-----
107 [[-0.49 -0.65]
108 [-0.65  0.72]
109 [-0.58 -0.25]]
110 -----VT_r-----
111 [[-0.49 -0.65 -0.58]
112 [-0.65  0.72 -0.25]]
113 -----QLSI-----
114 [[-0.21  0.18]]
115 Query ['ouro', 'prata', 'caminhão'] Document - 0 Similarity=-0.05395084366642496
116 Query ['ouro', 'prata', 'caminhão'] Document - 1 Similarity=0.9909874267484721
117 Query ['ouro', 'prata', 'caminhão'] Document - 2 Similarity=0.4479594658282973

```

Parte IV.

Classificação

DRAFT

Algoritmos Clássicos de Classificação

15.1. Naïve-Bayes

15.2. SVM

DRAFT

Redução de Dimensionalidade

Um dos problemas típicos com algoritmos clássicos de classificação é a enorme quantidade de atributos a serem usados, na prática um por palavra chave. Por isso, antes de usar esses algoritmos, é razoável usar algum método de redução de dimensionalidade.

A ideia principal da maioria dos métodos de redução de dimensionalidade é remover de antemão os termos que não ajudam, ou mesmo atrapalham, a classificação. Isso pode ser visto como uma remoção de ruído, considerando o texto um sinal onde os termos de “frequência mais alta” (aqui a frequência mais alta indica na verdade a raridade da palavra, não a contagem) são removidos para evitar que esse ruído de alta frequência atrapalhe a classificação.

A primeira abordagem, usando o modelo vetorial, é usar apenas os termos mais frequentes, possivelmente depois de retirar as stopwords. Na prática, essa técnica funciona razoavelmente bem a partir de 1024 termos.

Uma possibilidade é fazer uma codificação dos documentos usando o LSI, onde também é importante escolher o tamanho do vetor resultante para os documentos (r).

DRAFT

DRAFT

DRAFT

Bibliografia

- Abbott, Edwin (1991). *Flatland: A Romance of Many Dimensions*. 2^a ed. Princeton University Press.
- Adobe Systems Inc. (1999). *PostScript Language Reference (3rd Ed.)* USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201379228.
- Alvares, Reinaldo Viana, Ana Cristina Bicharra Garcia e Inhaúma Ferraz (2005). “STEMBR: A Stemming Algorithm for the BrazilianPortuguese Language”. Em: *EPIA 2005 - 12thPortuguese Conference on Artificial Intelligence*. Vol. Lecture Notes in Artificial Intelligence 3808. Covilhã, Portugal: Springer, pp. 693–701.
- Amri, Samir e Lahbib Zenkour (ago. de 2018). “Stemming and Lemmatization for Information Retrieval Systems in Amazigh Language”. Em: *Third International Conference, BDCA 2018*. Vol. 872. Third International Conference, BDCA 2018, Kenitra, Morocco, April 45, 2018, Revised Selected Papers, pp. 222–233. ISBN: 978-3-319-96291-7. DOI: 10.1007/978-3-319-96292-4_18.
- Andrade, Frank (13 de mar. de 2021). *5 Simple Ways to Tokenize Text in Python*. *Tokenizing text, a large corpus and sentences of different language*. Towards Data Science. URL: <https://towardsdatascience.com/5-simple-ways-to-tokenize-text-in-python-92c6804edfc4> (acesso em 14/03/2022).
- Apache Project (2011a). *Welcome to Apache Lucene*. Versão 9.0.0. Apache Software Foundation. URL: <https://lucene.apache.org/> (acesso em 10/03/2022).
- (2011b). *Welcome to PyLucene*. Apache Software Foundation. URL: <https://lucene.apache.org/pylucene/> (acesso em 10/03/2022).
- Araújo, Luciana Kuchenbecker (2022). *O que é fonema?* Brasil Escola. URL: <https://brasilescola.uol.com.br/o-que-e/portugues/o-que-e-fonema.htm> (acesso em 19/03/2022).
- Atkins, S., J. Clear e N. Ostler (1 de jan. de 1992). “Corpus Design Criteria”. *Literary and Linguistic Computing* 7.1, pp. 1–16. DOI: 10.1093/llc/7.1.1.

- Axler, Sheldon (2015). *Linear Algebra Done Right*. Ed. por Sheldon Axler e Kenneth Ribet. 3^a ed. Undergraduate Texts in Mathematics. Springer.
- Baeza-Yates, Ricardo e Berthier Ribeiro-Neto (2011a). *Modern Information Retrieval: The Concepts and Technology behind Search*. USA: Addison-Wesley Publishing Company.
- (2011b). *Modern Information Retrieval: The Concepts and Technology behind Search*. 2^a ed. USA: Addison-Wesley Publishing Company. ISBN: 9780321416919.
- Beghtol, C (1986). “Bibliographic Classification Theory And Text Linguistics: Aboutness Analysis, Intertextuality And The Cognitive Act Of Classifying Documents”. *Journal of Documentation* 42.2, pp. 84–113. DOI: <https://doi.org/10.1108/eb026788>.
- Belkin, N. J., Oddy R. N. e H. M. Brooks (jun. de 1982). “ASK for Information Retrieval Part I. Background and Theory”. *Journal of Documentation* 38.2.
- Betts, O. e R. Bouton (24 de mar. de 2022). *Snowball Stemming language and algorithms*. URL: <https://github.com/snowballstem/snowball> (acesso em 24/03/2022).
- Bird, Steven, Edward Loper e Ewan Klein (2009). *Natural Language Processing with Python*. OReilly Media Inc.
- Borlund, Pia (jan. de 2000). “Experimental components for the evaluation of interactive information retrieval systems”. *Journal of Documentation* 56.1, pp. 71–90. ISSN: 0022-0418. DOI: 10.1108/EUM0000000007110. URL: <https://doi.org/10.1108/EUM0000000007110>.
- Briet, S (1951). *Quest-ce que la documentation*. Paris.
- Brousentsov, N. P. et al. (2021). *Development of ternary computers at Moscow State University*. URL: <https://www.computer-museum.ru/english/setun.htm> (acesso em 25/12/2021).
- Bruza, P. D. e T. W. C. Huibers (1994). “Investigating Aboutness Axioms Using Information Fields”. Em: *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '94. Dublin, Ireland: Springer-Verlag, pp. 112–121. ISBN: 038719889X.
- Bruza, Peter e Theo Huibers (out. de 1996). “A study of aboutness in information retrieval”. *Artificial Intelligence Review* 10, pp. 381–407. DOI: 10.1007/BF00130692.
- Bruza, Peter, Dawei Song e Kam-Fai Wong (1999). “Fundamental Properties of Aboutness (Poster Abstract)”. Em: *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '99. Berkeley, California, USA: Association for Computing Machinery, pp. 277–278. ISBN: 1581130961. DOI: 10.1145/312624.312696. URL: <https://doi.org/10.1145/312624.312696>.
- Buckland, Michael K. (1997). “What Is a Document?” *Journal Of The American Society For Information Science* 48.9. John Wiley & Sons, Inc., pp. 804–809.
- Chomsky, Noam (2002). *Syntactic Structures*. 2^a ed. de Gruyter Mouton.
- Constantin, A. et al. (2016). “The Document Components Ontology (DoCO)”. *Semantic Web* 7.2, pp. 167–181. DOI: <http://dx.doi.org/10.3233/SW-150177>.
- Control Data Corporation (1975). *Control Data Cyber 170 Computer Systems: Hardware Reference Manual*. St. Paul, Minnesota.
- Coombs, James H., Allen H. Renear e Steven J. DeRose (nov. de 1987). “Markup Systems and the Future of Scholarly Text Processing”. *Commun. ACM* 30.11, pp. 933–947. ISSN:

- 0001-0782. DOI: 10.1145/32206.32209. URL: <https://doi.org/10.1145/32206.32209>.
- Cooper, William S. (1997). “Getting Beyond Boole”. Em: *Readings in Information Retrieval*. Ed. por Karen Sparck Jones e Peter Willet. Ed. por Edward Fox. The Morgan Kaufman Series in Multimedia Information and Systems. San Francisco: Morgan Fauffman Publishers.
- Cormen, Thomas H. et al. (2009). *Introdução aos Algoritmos*. 3ª ed. Elsevier.
- Daniels, Peter T. (2003). “Writing Systems”. Em: *The Handbook of Linguistics*. Ed. por Mark Aronoff e Janie Rees-Miller. Oxford: Blackwell. Cap. 3.
- DCMI (2022a). *About DCMI*. URL: <https://www.dublincore.org/about/> (acesso em 04/01/2022).
- (2022b). *DCMI Metadata Terms*. URL: <https://www.dublincore.org/about/> (acesso em 04/01/2022).
- (2022c). *Dublin Core*. URL: <https://www.dublincore.org/about/> (acesso em 04/01/2022).
- Document Academy, The (2022). *What is a document?* URL: <http://documentacademy.org/?what-is-a-document> (acesso em 04/01/2022).
- Fairthorne, R. A (1969). “Content analysis, specification and control”. *Annual Review of Information Science and Technology* 4, pp. 73–109.
- Frakes, William B. e Ricardo Baeza-Yates (30 de jun. de 1992). *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall. 512 pp. ISBN: 0134638379.
- Freed, N. e N. Borenstein (nov. de 1996). *RFC-2045 - Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. URL: <https://www.rfc-editor.org/rfc/rfc2045> (acesso em 30/12/2021).
- Fuhr, N. (1992). “Probabilistic Models in Information Retrieval”. *The Computer Journal* 35.3, pp. 243–255.
- Galvez, Carmen, Félix de MoyaAnegón e Víctor H. Solana (1 de jan. de 2005). “Term conflation methods in information retrieval”. *Journal of Documentation* 61.4, pp. 520–547. DOI: 10.1108/00220410510607507.
- Goffman, William (1969). “An Indirect Method of Information Retrieval”. *Information Storage and Retrieval* 4, pp. 361–373.
- Grossman, David A. e Ophir Frieder (2004). *Information Retrieval: Algorithms and Heuristics*. Netherlands: Springer.
- Guedes, Emanuel Guedson Ferreira (2009). “O Conceito Aboutness na Organização e Representação do Conhecimento”. Diss. de mest. Programa de Pós-Graduação em Ciência da Informaçãoda Faculdade de Filosofia e Ciências daUniversidade Estadual Paulista UNESP. URL: https://www.marilia.unesp.br/Home/Pos-Graduacao/CienciadaInformacao/Dissertacoes/guedes_egf_me_mar.pdf (acesso em 21/03/2022).
- Haliday, M. A. K. e Ruqaiya Hasan (1976). *Cohesion in English*. London: Longman.
- Honnibal, Matthew e Ines Montani (2017). “spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing”. To appear.

- Hutchins, W. J. (mai. de 1978). “The concept of ‘aboutness’ in subjectindexing”. *Aslib Proceedings* 30.5, pp. 172–181.
- IBM (2010). *z/OS Basic Skills: The EBCDIC character set - Application programming on z/OS*. URL: <https://www.ibm.com/docs/en/zos-basic-skills?topic=mainframe-ebcdic-character-set> (acesso em 27/12/2021).
- IETF (30 de dez. de 2021). *Internet Standards: RFCs*. Internet Engineering Task Force. (Acesso em 30/12/2021).
- IFLA (fev. de 2009). *Functional Requirements For Bibliographic Records. Final Report*. Rel. técn. International Federation of Library Associations e Institutions. URL: <https://repository.ifla.org/bitstream/123456789/811/2/ifla-functional-requirements-for-bibliographic-records-frbr.pdf> (acesso em 04/01/2022).
- International Phonetic Association (1999). *Handbook of the International Phonetic Association : A Guide to the Use of the International Phonetic Alphabet*. Cambridge University Press.
- Johnson, Lyndon B. (11 de mar. de 1968). *Memorandum Approving the Adoption by the Federal Government of a Standard Code for Information Interchange*. Ed. por Gerhard Peters e John T. Woolley. The American Presidency Project. URL: <https://www.presidency.ucsb.edu/node/237376> (acesso em 26/12/2021).
- Klensin, J. (out. de 2008). *RFC-5321 - Simple Mail Transfer Protocol*. URL: <https://www.rfc-editor.org/rfc/rfc5321.html>.
- Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201896842.
- Leibson, Steven (1 de dez. de 2021). “Which Was The First Microprocessor? Intel 4004, AiResearch MP944, or Four-Phase AL1?” *Electronic Engineering Journal*. URL: <https://www.eejournal.com/article/which-was-the-first-microprocessor/> (acesso em 09/03/2022).
- Library of Congress, The (13 de mar. de 2020). *MARC Standards*. URL: <https://www.loc.gov/marc/> (acesso em 04/01/2022).
- (2022). *BIBFRAME Ontology*. Versão 2.1.0. URL: <https://id.loc.gov/ontologies/bibframe.html> (acesso em 04/01/2022).
- Lopes, M. C. S. (2004). “Mineração de Dados Textuais Utilizando Técnicas de Clustering Para o Idioma Português”. Tese de dout. UFRJ.
- Louro, A. Tavares (2007). *O sentido denotativo e conotativo de provérbios’in Ciberdúvidas da Língua Portuguesa*. Ciberdúvidas da Língua Portuguesa. URL: <https://ciberduvidas.iscte-iul.pt/consultorio/perguntas/o-sentido-denotativo-e-conotativo-de-proverbios/20662> (acesso em 01/07/2020).
- Lovins, Julie Beth (mar.–jun. de 1968). “Development of a Stemming Algorithm”. *Mechanical Translation and Computational Linguistics* 11.1-2. URL: <https://aclanthology.org/www.mt-archive.info/MT-1968-Lovins.pdf> (acesso em 23/03/2022).
- Löwis, Martin v. (2010). *Flexible String Representation*. PEP 393. URL: <https://www.python.org/dev/peps/pep-0393/>.

- Lucca, J. L. De e Maria das Graças Volpe Nunes (nov. de 2002). *Lematização versus Stemming*. Rel. técn. NILC-TR-02-22. o Núcleo Interinstitucional de Lingüística Computacional.
- Mackenzie, Charles E. (1980). *Coded Character Sets, History, and Development*. Addison-Wesley.
- Manning, Christopher D., Prabhakar Raghavan e Hinrich Schütze (1 de abr. de 2009). *An Introduction to Information Retrieval*. Cambridge UP: Cambridge University Press. URL: <https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf>.
- Manning, Christopher D., Prabhakar Raghavan e Hinrich Schütze (2008). *Introduction to Information Retrieval*. USA: Cambridge University Press. ISBN: 0521865719.
- Meyriat, Jean (1981). “Document, documentation, documentologie”. *Schéma et Schématisation* 14, pp. 51–63.
- (2016). “Documento, documentação, documentologia”. Trad. por Marcílio de Brito, Cristina Ortega e Camila Mariana A. da Silva. *Revista Perspectivas em Ciência da Informação* 21.3, pp. 240–253. ISSN: 19815344. URL: <http://portaldeperiodicos.eci.ufmg.br/index.php/pci/article/view/2891> (acesso em 04/01/2022).
- Mizzaro, Stefano (jun. de 1998). “How many relevances in information retrieval?” *Interacting with Computers* 10.3, pp. 303–320. DOI: 10.1016/s0953-5438(98)00012-5.
- Mozilla (2021). *JavaScript*. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (acesso em 30/12/2021).
- Nelson, Theodor Holm (2 de out. de 1997). *Embedded Markup Considered Harmful*. URL: <https://www.xml.com/pub/a/w3j/s3.nelson.html> (acesso em 24/12/2021).
- Neto, Nelson Fordelone (11 de jul. de 2017). *Uma Homenagem Aos 73 Anos De Chico Buarque. Chico Buarque E A Revolução Dos Cravos*. Esquerda Diário. URL: <https://www.esquerdadiario.com.br/Chico-Buarque-e-a-Revolucao-dos-Cravos> (acesso em 29/03/2022).
- NISO (20 de fev. de 2013). *The Dublin Core Metadata Element Set*. NISO Standards ANSI/NISO Z39.85-2021. NISO - National Information Standards Organization. ISBN: 978-1-937522-14-8. URL: https://groups.niso.org/apps/group_public/download.php/10258/Z39-85-2012_dublin_core.pdf (acesso em 04/01/2022).
- Nothman, Joel, Hanmin Qin e Roman Yurchak (20 de jul. de 2018). “Stop Word Lists in Free Open-source Software Packages”. Em: *Proceedings of Workshop for NLP Open Source Software*. Association for Computational Linguistics. Melbourne, Australia, pp. 7–12.
- Pagani, Luiz Arthur (2022). *Diagramas em árvore como representação da estrutura sintática (Slides do curso UFPR - HL396 Língua Portuguesa IV)*. URL: https://docs.ufpr.br/~arthur/textos/apr/sintaxe/arv_apr.pdf (acesso em 06/01/2022).
- Paice, Chris D. (nov. de 1990). “Another Stemmer”. *SIGIR Forum* 24.3, pp. 56–61. ISSN: 0163-5840. DOI: 10.1145/101306.101310. URL: <https://doi.org/10.1145/101306.101310>.
- Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12, pp. 2825–2830.
- Pilgrim, Mark, Dan Blanchard e Ian Cordasco (2015). *chardet Docs Usage*. URL: <https://chardet.readthedocs.io/en/latest/usage.html> (acesso em 27/12/2021).

- Porter, M. F. (jul. de 1980). “An Algorithm for Suffix Stripping”. *Program* 14.3, pp. 130–137. URL: <https://tartarus.org/martin/PorterStemmer/def.txt> (acesso em 22/03/2022).
- (out. de 2001). *Snowball: A language for stemming algorithms*. URL: <https://snowballstem.org/texts/introduction.html> (acesso em 24/03/2022).
- Postel, Jonathan B. (ago. de 1982). *RFC-821 - Simple Mail Transfer Protocol*. URL: <https://www.rfc-editor.org/rfc/rfc821> (acesso em 30/12/2021).
- Raber, Douglas (2003). *The Problem Of Information: An Introduction to Information Science*. Lanham, MD: Scarecrow Press.
- ehkek, Radim e Petr Sojka (mai. de 2010). “Software Framework for Topic Modelling with Large Corpora”. English. Em: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. <http://is.muni.cz/publication/884893/en>. Valletta, Malta: ELRA, pp. 45–50.
- Riva, Pat, Patrick Le Buf e Maja umer (ago. de 2017). *IFLA Library Reference Model: A Conceptual Model for Bibliographic Information*. URL: <https://www.ifla.org/wp-content/uploads/2019/05/assets/cataloguing/frbr-lrm/ifla-lrm-august-2017.pdf> (acesso em 04/01/2022).
- Robertson, S. (1977). “The Probability Ranking Principle”. *Journal of Documentation* 33, pp. 294–304.
- Roelleke, Thomas (26 de jul. de 2013). *Information Retrieval Models: Foundations and Relationships*. Vol. 5. Synthesis Lectures on Information Concepts, Retrieval, and Services 3. Morgan & Claypool Publishers LLC, pp. 1–163. DOI: 10.2200/s00494ed1v01y201304icr027.
- Rogers, H. (2005). *Writing Systems: A Linguistic Approach*. Blackwell Textbooks in Linguistics. Wiley.
- Salton, Gerard e Michael J. McGill (1983a). *Introduction to Modern Information Retrieval*. New York: McGraw-Hill.
- (1983b). *Introduction to Modern Information Retrieval*. New York: McGraw-Hill.
- Saracevic, Tefko (1975). “Relevance: A Review of the Literature and a Framework for Thinking on the Notion in Information Science.” *Journal of the American Society for Information Science*, pp. 3–71.
- (2007). “Relevance: A Review of the Literature and a Framework for Thinking on the Notion in Information Science. Part II: Nature and Manifestations of Relevance”. *Journal of The American Society ofr Information Science and Technology*, 58 (13), pp. 1915–1933. DOI: 10.1002/asi.20682.
- (2017a). *The Notion of Relevance in Information Science Everybody knows what relevance is. But, what is it really?* Vol. 58. Synthesis Lectures on Information Concepts, Retrieval, and Services 50. Morgan & Claypool.
- (2017b). *The Notion of Relevance in Information Science Everybody knows what relevance is. But what is it really?* Synthesis Lectures On Information Concepts, Retrieval, And Services 50. Morgan & Claypool. DOI: 10.2200/S00723ED1V01Y201607ICR050.
- Shotton, David e Silvio Peroni (3 de jul. de 2015). *DoCO, the Document Components Ontology*. URL: <https://sparontologies.github.io/doco/current/doco.html> (acesso em 04/01/2022).

- Snowball Project (2022). *Portuguese stemming algorithm*. URL: <http://snowball.tartarus.org/algorithms/portuguese/stemmer.html> (acesso em 24/03/2022).
- Stock, W. G. e M. Stock (2013). *Handbook of Information Science*. Knowledge and Information. De Gruyter.
- Thompson, H. S. e D. McKelvie (1997). “Hyperlink semantics for standoff markup of read-only documents”. Em: *Proceedings of SGML Europe*. URL: <http://www.ltg.ed.ac.uk/~ht/sgmleu97.html>. (acesso em 24/12/2021).
- Tillett, Barbara (fev. de 2004). *What is FRBR? A Conceptual Model for the Bibliographic Universe*. Functional Requirements for Bibliographic Records. Library of Congress Cataloging Distribution Service. URL: <http://www.loc.gov/cds/FRBR.html> (acesso em 04/01/2022).
- Unicode, Inc. (24 de jul. de 2017). *What is Unicode?* URL: <https://unicode.org/standard/WhatIsUnicode.html> (acesso em 24/12/2021).
- (22 de ago. de 2019). *The Unicode Standard: A Technical Introduction*. URL: <http://www.unicode.org/standard/principles.html> (acesso em 25/12/2021).
 - (2021). *Unicode Version 14.0 Character Counts*. URL: https://www.unicode.org/versions/stats/charcountv14_0.html (acesso em 24/12/2021).
- van Rijsbergen, C. J. Keith (1979). *Information Retrieval*.
- (17 de mai. de 1990). “The Science of Information Retrieval: Its Methodology and Logic”. Em: *Conference Informatienvetenschap in Nederland (Conference Informatienvetenschap in. Nederland)*, RABIN, The Hague, p. 24.
 - (2003). *Introduction to Information Retrieval (ESSIR 2003) - Slides*. URL: <http://mrim.imag.fr/essir03/PDF/4.Rijsbergen.pdf> (acesso em 01/01/2022).
- Van Rossum, Guido e Fred L. Drake (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace. ISBN: 1441412697.
- Veloso, Caetano (1969). *Os Argonautas*. URL: <https://www.lettras.mus.br/caetano-veloso/44761/> (acesso em 29/03/2022).
- W3C (29 de set. de 2006). *Extensible Markup Language (XML) 1.1 (Second Edition)*. *W3C Recommendation 16 August 2006, edited in place 29 September 2006*. URL: <https://www.w3.org/TR/xml11/> (acesso em 29/12/2021).
- (26 de nov. de 2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. *W3C Recommendation 26 November 2008*. World Wide Web Consortium. URL: <https://www.w3.org/TR/xml/> (acesso em 29/12/2021).
 - ed. (5 de jan. de 2012). *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. *W3C Recommendation 5 April 2012*. W3C. URL: <https://www.w3.org/TR/xmlschema11-1/> (acesso em 29/12/2021).
 - (30 de dez. de 2021a). *About W3C*. W3C. URL: <https://www.w3.org/Consortium/> (acesso em 30/12/2021).
 - (30 de dez. de 2021b). *CSS Snapshot 2021*. W3C Group Note, 30 December 2021. Ed. por Tab Atkins Jr, Erika J. Etemad e Florian Rivoal. W3C.
- WHATWG (21 de dez. de 2021). *HTML Living Standard Last Updated 21 December 2021*. WHATWG (Apple, Google, Mozilla, Microsoft). URL: <https://html.spec.whatwg.org/> (acesso em 30/12/2021).

- Wikipedia (26 de dez. de 2020). *Setun*. URL: <https://en.wikipedia.org/wiki/Setun> (acesso em 25/12/2021).
- Zobel, Justin e Alistair Moffat (2006). “Inverted Files for Text Search Engines”. *ACM Computing Surveys* 38.2, 6–es. ISSN: 0360-0300. DOI: 10.1145/1132956.1132959. URL: <https://doi.org/10.1145/1132956.1132959>.
- Zobel, Justin, Alistair Moffat e Kotagiri Ramamohanarao (dez. de 1998). “Inverted Files versus Signature Files for Text Indexing”. *ACM Trans. Database Syst.* 23.4, pp. 453–490. ISSN: 0362-5915. DOI: 10.1145/296854.277632. URL: <https://doi.org/10.1145/296854.277632>.

Apêndices

DRAFT



Tópicos em Python

A.1. Dicionários

Dicionários em Python são implementados com uma hash-table com endereçamento aberto que cresce dinamicamente quando está 2/3 ocupada, iniciando com 8 *slots*.

`d =` create an empty dictionary and assign it to `d` `d[key] = value` assign a value to a given dictionary key `d.keys()` the list of keys of the dictionary `list(d)` the list of keys of the dictionary `sorted(d)` the keys of the dictionary, sorted `key in d` test whether a particular key is in the dictionary `for key in d` iterate over the keys of the dictionary `d.values()` the list of values in the dictionary `dict([(k1,v1), (k2,v2), ...])` create a dictionary from a list of key-value pairs `d1.update(d2)` add all items from `d2` to `d1` `defaultdict(int)` a dictionary whose default value is zero

Tabela A.1.: Alguns formas de usar dicionários em Python

Código	Resultado
<code>d = {}</code>	cria um dicionário vazio
<code>d = {1 : 10 , 2 : 20 , ...}</code>	cria um dicionário a partir de duplas chave:valor
<code>d=defaultdict(f)</code>	cria um dicionário cujo valor default é fornecido por uma função f
<code>dict([(k1,v1), (k2,v2), ...])</code>	cria um dicionário a partir de uma lista de tuplas chave-valor
<code>d[key] = value</code> <code>d1.update(d2)</code>	insere um par chave valor no dicionário adiciona os itens do d2 no d1
<code>key in d</code> <code>list(d)</code> <code>sorted(d)</code> <code>d.keys()</code> <code>d.values()</code>	testa se uma chave está em um dicionário retorna uma lista de chaves do dicionário retorna as chaves do dicionário ordenadas retorna uma <i>iterable view</i> das chaves do dicionário retorna uma <i>iterable view</i> dos valores do dicionário
<code>for key in d</code>	faz uma iteração nas chaves do dicionário, e não nos pares

A.2. Strings

A maneira de representar texto em Python é por meio do Tipo `str`, conhecido como `string`¹.

Exemplos de string são:

```
1 "A"
2 'A'
3 "Uma frase"
4 'Uma frase'
5 """ Uma frase longa
6 muito longa
7 com várias linhas"""
```

Como visto no exemplo acima, strings devem ser delimitadas por aspas (`"`), apóstrofes (`'`), ou três aspas quando se deseja uma string que seja muito longa e use muitas linhas².

Existem muitas funções que trabalham com strings no pacote `string`, que serão vistas mais tarde.

As strings podem ser concatenadas para criar outras strings. Para isso também se usa o operador `+`. Uma string vazia, sem caracteres, é indicada abrindo e fechando a string sem nada no meio, como em `" "`. Uma string pode conter espaços, como em `" "`, uma string com um só espaço.

O tamanho de uma string é dado pela função `len(s)`³. Essa é uma função muito usada, inclusive com outros tipos de dados.

Seguem alguns exemplos:

```
1 s1 = "alfa" + "beto"
2 s2 = "alfa" + " " + "beto"
3 t1 = len("alfabeto")
4 t2 = len("a l f a b e t o")
```

Não existe um operador para subtrair strings, mas existem funções que permitem modificá-las de várias formas, ou buscar strings dentro de strings.

Na seção A.2.5 veremos mais funções sobre strings.

¹Em português muitas vezes usamos o termo sequência de caracteres, mas não há uma tradução direta e a palavra “string” será usada no texto

²Muitas linguagens de programação possuem dois tipos, um de caracteres, ou seja, uma letra apenas, e um de strings, possivelmente sendo o segundo um vetor do primeiro. Em Python, não existe um tipo caractere. Uma string de uma letra é também uma string

³Nesse livro usamos uma marca especial para permitir que o leitor veja o espaço, ele não existe e não deve ser digitado no programa.

A.2.1. Índices de Strings

Um dos operadores mais poderosos sobre strings é o `[]`. Ele permite indexar um caractere específico da String ou pegar uma fatia da String. Ele também permite contar do início para o fim, ou do fim para o início (nesse caso usando números negativos).

A primeira letra de uma string está na posição zero.

Programa A.1: Obtendo um caractere indexado em uma string

```
1 print("beto"[0])
2 print("beto"[3])
```

———— Saída para o Programa A.1 ————

```
1 b
2 o
```

Na contagem ao contrário, a última letra está na posição `-1`.

Programa A.2: Obtendo um caractere indexado negativo em uma string

```
1 print("beto"[-1])
2 print("beto"[-3])
```

———— Saída para o Programa A.2 ————

```
1 o
2 e
```

Como vemos, há uma pequena diferença na forma de contar. Ela existe por alguns motivos. Primeiro, não existe o número `-0`, segundo, ela permite que o trabalho com fatias funcione melhor.

A.2.2. Fatias de Strings

Fatias são pedaços de strings.

Mais interessante ainda que poder pegar caracteres um a um, em Python podem pegar pedaços ou fatias de strings usando também o operador `[]`.

Primeiro vamos ver alguns exemplos mais simples, usando a notação início e fim da fatia, para frente e para trás. Para isso, usamos o operador fatia no formato `[<início>:<fim>]`.

Programa A.3: Fatiando uma string

```
1 print("beto"[0:1])
2 print("beto"[0:2])
3 print("beto"[-3:-1])
```

Saída para o programa Programa A.3

```
1 b
2 be
3 et
```

Para entender como funciona devemos entender que as posições nas fatias, positivas ou negativas, não indicam verdadeiramente as posições, mas sim os intervalos entre as letras. A Figura A.1 demonstra isso. Na figura devemos notar que a posição positiva final não pode ser usada em índices, apenas em fatias.

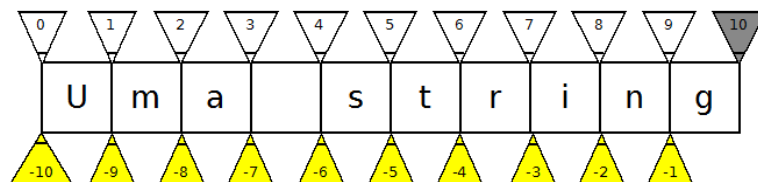


Figura A.1.: As posições de indexação das strings estão na verdade entre as letras

A notação de fatias também pode ser usada com 3 argumentos, como em [*<início>*:*<fim>*:*<passo>*].

Dessa maneira, podemos criar fatias especiais que pulam letras, como em:

Programa A.4: Fatias que pulam letras.

```
1 print("alfabeto brasileiro"[1:12:2])
```

Saída para o programa Programa A.4

```
1 laeoba
```

Outra característica: se a posição inicial, ou final, é deixada em branco, então significa que queremos a fatia a partir do início, ou fim, da string.

Programa A.5: Marcando só a posição final.

```
1 print("alfabeto"[:5])
2 print("alfabeto"[5:])
```

Saída para o programa Programa A.5

```
1 alfab
2 eto
```

E se deixamos início e fim vazios uma cópia da string é criada.

A.2.3. O operador in

Strings, como todas as sequências, possui um operador `in` que diz se um elemento está dentro dela. Também é possível verificar se um elemento não pertence a uma string usando o operador `not in`.

```
>>> "1" in "1234"
True
>>> "5" in "1234"
False
>>> "1" not in "1234"
False
>>> "5" not in "1234"
True
```

A.2.4. Comparando Strings

A princípio, strings são comparadas em sua ordem lexicográfica, isto é, primeiro na "ordem alfabética", depois pelo tamanho da string.

Na verdade, cada caractere tem um valor inteiro, ou seu *Unicode code point*. Esse valor pode ser calculado com a função `ord(c)`, onde (c) é uma string de 1 caractere apenas. Também podemos calcular o caractere dado um número, com a função `chr(i)`, onde i é um número inteiro até 1.114.111, ou 0x10FFFF, o maior caractere Unicode.

Por necessidades de representação, "A" e "a" são caracteres diferentes para o computador.

Programa A.6: Cada letra equivale a um número.

```
1 print(ord("a"))
2 print(ord("A"))
3 print("a"<"A")
```

Saída para o programa Programa A.6

```
1 97
2 65
3 False
```

A.2.5. Funções com String

- `lower()`, coloca todas as letras em minúscula;
- `split()`, devolve uma lista de tokens, considerando apenas uma caractere para separação, e
- `upper()`, coloca todas as letras em maiúscula.

A.2.6. Exercícios

Exercício A.1:

Ana recebeu de Beto uma mensagem em código, onde cada terceira letra apenas deve ser lida. Se a mensagem recebida foi "asefduds grtrwewe jyacgmsdo", qual a mensagem original?

```
print("asefduds grtrwewe jyacgmsdo"[2::3])
```

Veja que como as letras de uma string são contadas a partir do número zero, a primeira terceira letra é a de índice 2.

Exercício A.2:

Dê o resultado para as seguintes expressões:

```
print("alfabeto brasileiro"[:2])
print("alfabeto brasileiro"[11:5:-1])
print("alfabeto brasileiro"[:-1])
```

Exercício A.3:

Ana resolveu responder a Beto, usando novamente um código, porém agora eles combinaram inverter a string e depois escolher apenas as letras pares. Lembre, porém, que eles sempre começam contando a primeira letra como 1. A mensagem de Ana para Beto foi 'fohduaeryoemwatny sokhrnjertm wudEn'

```
1 'fohduaeryoemwatny sokhrnjertm wudEn'[::-1][1::2]
```

Nessa solução usamos dois passos. Como o operador [] acontece primeiro a esquerda, então primeiro invertemos a string para depois pegar as letras pares.

DRAFT

Tópicos Seleccionados de Matemática

B.1. Autovetores e Autovalores

Seja uma matriz quadrada \mathbf{A} , que pode ser vista como indicando uma transformação linear $\mathbb{R}^N \rightarrow \mathbb{R}^N$ em um mesmo espaço vetorial V . Como \mathbf{A} opera sobre o mesmo espaço vetorial, é também chamada de um operador (Axler, 2015).

Um subespaço U de V é invariante sob \mathbf{A} se $u \in U \implies \mathbf{A}u \in U$ (Axler, 2015).

Seja $v \in V$ e $v \neq 0$, e U definido como o conjunto de todos os múltiplos escalares de v (Axler, 2015):

$$U = \{\lambda v : \lambda \in \mathbb{R}\} \quad (\text{B.1})$$

Se U é invariante sob um operador \mathbf{A} , então existe um λ tal que (Axler, 2015):

$$\mathbf{A}v = \lambda v \quad (\text{B.2})$$

Seja \mathbf{A} um operador em V , um número $\lambda \in \mathbb{R}$ é um **autovalor** de \mathbf{A} se existe um $v \in V$ tal que $v \neq 0$ e $\mathbf{A}v = \lambda v$ (Axler, 2015).

Supondo que o operador \mathbf{A} tem o autovalor λ , então o vetor $v \in V$ é tal que $v \neq 0$ e $\mathbf{A}v = \lambda v$ é chamado de **autovetor** de \mathbf{A} (Axler, 2015).

DRAFT

Índice Remissivo

- abjads, **15**
- Aboutness, **115**
- abugidas, **15**
- acurácia, **106**
- afixos, **72**
- agrupamento, 11
- alfabéticos, **15**
- Arquivo, **25**
- ASCII, 31, 48
- autovalor, **187**
- autovetor, **187**

- Bag of Words, 12
- base64, 48
- BCD, 29
- Biblioteca Eletrônica de Textos, **25**
- BOM, 38
- Brown Corpus, **25**
- busca, 11

- caracteres, 15
- Carriage Return, 35
- Case folding, **74**
- CCITT, 31
- CDC Cyber 170, 30
- chardet, 38, 40
- charset-normalizer, 40
- classificação, 11

- Code Pages, 34
- Codificação de Binários em Texto, 48
- concatenando strings, 181
- conflation, 75
- Corpus, **25**
- CP1252, 36, 41
- CR, 35
- CSS, 38
- Código Morse, 31

- DCMI, **24**
- decomposição em valores singulares, **152**
- desinência, **72**
- desinência nominais, **72**
- desinência verbal, **72**
- discurso, **12**
- DoCO, **24**
- Document Components Ontology, **24**
- Documentalidade, **10**
- documento, **8**
- documentos por atribuição, **9**
- documentos por intenção, **9**
- DTD, 66
- Dublin Core Metadata Initiative, **24**

- EBCDIC, 31
- ELEMENT, 66
- end of line, 35

- EOL, 35
- Estabilidade, **9**
- fatias de strings, 182
- fonético, **12**
- forma canônica, **73**
- FRBR, **22**
- fusão, 75
- grafemas, 15
- gramatical, **12**
- HTML, 38, 53, 56
- HTML5, 38
- identificação de entidades nomeadas, 11
- Indexalidade, **9**
- indexando strings, 182
- JavaScript, 38
- KNIME, 41
- lema, **73**
- len
 - strings, 181
- LF, 35, 41
- Line Feed, 35
- logográficos, **15**
- LSA, **151**
- LSI, **151**
- Lucene, **145**
- léxico, **12**
- Mac OSX, 35
- MARC, **25**
- matriz de confusão, **106**
- Matriz Termo Documento, 151
- metadado, 10
- MIME, 48
- morfema, **72**
- morfemas, **15**
- morfologia, 72
- morfológico, **12**
- n-gramas, **11**
- necessidade de informação, 11
- NLTK, 74
- Page Description Format, 60
- PCDATA, 66
- PDF, **60**, 60
- pdfminer, 61
- PHP, 38
- Pluralidade, **9**
- Porter Stemmer, **73**
- PostScript, **60**, 60
- pragmático, **12**
- precisão, **106**
- prefixos, **72**
- Produtividade, **10**
- PS, 60
- PyLucene, **145**
- Python, 56
- radical, **72**
- raiz, **72**
- revocação, **106**
- RFC, 54
- RSLP, **79**
- script, 34
- Semântica Latente, **151**
- semântico, **12**
- Setun, 30
- SGML, 53
- silábicos, **15**
- sintático, **12**
- sistema de escrita, **15**
- sistemas mistos, **15**
- Snowball, **77**
- SQL, 38
- sqlite 3, 101
- Stemmers, **75**
- Stopwords, **85**
- str, 181
- string, 181
- strings
 - [], 182
 - concatenação, 181
 - fatias, 182
 - indexando, 182

len, 181
 tamanho de uma..., 181
 Subcorpus, **25**
 sufixos, **72**
 SVD, **152**

 tamanho de uma string, 181
 tema, **72**
 texto, **7**
 trácicos (*featural*), **15**
 Unicode, 34

UTF-32, **35**
 UTF-8, **35**, 38
 uuencode, 48

 vogal temática, **72**

 W3C, 54

 XML, 38, 56, 65
 XML 1.0, 65
 XML 1.1, 65

DRAFT