Big Data Avançado e Mineração de Dados

Marcos Roberto Ribeiro

Formação Inicial e Continuada

INSTITUTO FEDERAL
Minas Gerais

+ IFMG
Campus Bambuí

Marcos Roberto Ribeiro

Big Data Avançado e Mineração de Dados

1ª Edição

Belo Horizonte

Instituto Federal de Minas Gerais

2022

Pró-reitor de Extensão	Carlos Bernardes Rosa Júnior
Diretor de Programas de Extensão	Niltom Vieira Junior
Coordenação do curso	Marcos Roberto Ribeiro
Arte gráfica	Ângela Bacon
Diagramação	Eduardo dos Santos Oliveira

FICHA CATALOGRÁFICA
Dados Internacionais de Catalogação na Publicação (CIP)

R484p Ribeiro, Marcos Roberto.

Big Data Avançado e Mineração de Dados [recurso eletrônico] / Marcos Roberto Ribeiro. – Belo Horizonte: Instituto Federal de Minas Gerais, 2022.

129p.: il. color.

E-book, no formato PDF.
Material didático para Formação Inicial e Continuada.
ISBN: 978-65-5876-025-2

1. Big data. 2. Preparação de dados. 3. Mineração de dados. 4. Classificação. 5. Agrupamento de dados. I. Título.

CDD 005.13

Catalogação: Douglas Bernardes de Castro - CRB-6/2802

Sobre o material

Este curso é autoexplicativo e não possui tutoria. O material didático, incluindo suas videoaulas, foi projetado para que você consiga evoluir de forma autônoma e suficiente.

Caso opte por imprimir este *e-book*, você não perderá a possibilidade de acessar os materiais multimídia e complementares. Os *links* podem ser acessados usando o seu celular, por meio do glossário de Códigos QR disponível no fim deste livro.

Embora o material passe por revisão, somos gratos em receber suas sugestões para possíveis correções (erros ortográficos, conceituais, *links* inativos etc.). A sua participação é muito importante para a nossa constante melhoria. Acesse, a qualquer momento, o Formulário “Sugestões para Correção do Material Didático” clicando nesse *link* ou acessando o QR *Code* a seguir:

Formulário de Sugestões

Para saber mais sobre a Plataforma +IFMG acesse

http://mais.ifmg.edu.br

Palavra do autor

Uma das tecnologias mais importantes da Computação são os bancos de dados presentes em sistemas computacionais utilizados em inúmeras atividades. Mesmo com os grandes avanços já alcançados, a Computação continua em franca evolução, principalmente na área de Banco de Dados. A partir da década de 2000, com a popularização da Internet, houve um crescente aumento na produção de dados pela humanidade. Essa quantidade massiva de dados é chamada de Big Data. Para analisar o Big Data, em diversas situações, é interessante utilizar técnicas de Mineração de Dados. Tais técnicas permitem a descoberta de informações úteis e não evidentes em bancos de dados. O presente curso, na forma de uma introdução a Big Data e Mineração de Dados, é muito interessante para despertar e iniciar a formação de novos profissionais nessa área.

Bons estudos!

Marcos Roberto Ribeiro

Apresentação do curso

Este curso está dividido em quatro semanas, cujos objetivos de cada uma são apresentados, sucintamente, a seguir.

SEMANA 1	- Conhecer os conceitos de Big Data e Mineração de Dados; - Conhecer os básico da linguagem de programação Python; - Desenvolver códigos básicos da linguagem Python.
SEMANA 2	- Conhecer os conceitos básicos de preparação de dados; - Desenvolver e entender códigos para preparação de dados.
SEMANA 3	- Conhecer os conceitos básicos de técnicas de classificação de dados; - Desenvolver e entender códigos para classificação de dados.
SEMANA 4	- Conhecer os conceitos básicos de agrupamento de dados; - Desenvolver e entender códigos para agrupamento de dados.

Carga horária: 40 horas.

Estudo proposto: 2h por dia em cinco dias por semana (10 horas semanais).

Apresentação dos Ícones

Os ícones são elementos gráficos para facilitar os estudos, fique atento quando eles aparecem no texto. Veja aqui o seu significado:

Atenção: indica pontos de maior importância no texto.

Dica do professor: novas informações ou curiosidades relacionadas ao tema em estudo.

Atividade: sugestão de tarefas e atividades para o desenvolvimento da aprendizagem.

Mídia digital: sugestão de recursos audiovisuais para enriquecer a aprendizagem.

Sumário

Objetivos
- Conhecer os conceitos de Big Data e Mineração de Dados;
- Conhecer os básico da linguagem de programação Python;
- Desenvolver códigos básicos da linguagem Python.

Mídia digital: Antes de iniciar os estudos, vá até a sala virtual e assista ao vídeo “Apresentação do curso”.

1.1 Introdução

A partir do início do século XXI, a Computação e suas tecnologias causaram uma verdadeira revolução em diversas áreas da sociedade (MARÇULA; FILHO, 2008). Isso proporcionou grandes melhorias e mudanças nas formas de comunicação, no ambiente de trabalho, na qualidade de vida e no funcionamento de empresas e órgãos governamentais (VELLOSO, 2014). Quando falamos de computação, basicamente, estamos nos referindo a processamento e dados. O processamento são todas as operações que o computador pode realizar e os dados são todas as informações que o computador pode armazenar.

A partir da década de 2000, com a popularização da Internet, houve um crescente aumento na produção de dados pela humanidade. Essa quantidade massiva de dados é chamada de Big Data (WIKIPÉDIA, 2022a). Alguns exemplos de sistemas que envolvem Big Data são aplicações de bolsas de valores, redes sociais e científicas. Muitas dessas aplicações podem gerar terabytes de dados em um dia ou mesmo em alguns minutos (CURRY, et al. , 2022). As principais características do Big Data são volume, variedade e velocidade. O volume diz respeito a grande quantidade de dados produzida, na ordem de terabytes diários ou maior (SOTO; LUNA; CANO, 2016).

Quanto à variedade, as fontes de dados são muito variadas como transações comerciais, redes sociais, sensores, celulares, redes sociais, etc. Além da variedade de fontes de dados, existem diferentes formatos e tipos de dados, desde dados estruturados, dados numéricos, até dados não estruturados, como textos, imagens, vídeos e áudios. Um dos maiores desafios do Big Data é gerenciar todos esses diferentes tipos de dados e fontes. A velocidade se refere ao tempo de processamento de dados de Big Data. Devido ao grande volume e variedade de dados, todo o processamento deve ser ágil para se obter as informações necessárias.

Muitas vezes, as ferramentas tradicionais de análise de dados não podem ser usadas no Big Data. Somado a isso, a natureza não trivial dos dados também impede o uso de ferramentas mais simples. A mineração de dados combina métodos tradicionais de análise de dados com algoritmos sofisticados para analisar bases de dados. Ademais, com

a mineração de dados, é possível analisar novos tipos de dados e avaliar de uma maneira diferente dados analisados por ferramentas tradicionais (ALKASEER, 2016).

A utilização de técnicas de mineração de dados envolve um processo chamado de descoberta de conhecimento em bancos de dados ou *knowledge discovery database* (KDD). Tal processo pode ser dividido nas etapas de preparação dos dados, mineração e análise dos resultados. A preparação dos dados pode incluir diversos processamentos como limpeza, seleção e transformação. A mineração propriamente dita é a execução para extração de um modelo ou informações úteis relacionadas à base de dados (IZBICKI; SANTOS, 2020).

Após a mineração, na etapa de análise de resultados, o modelo extraído pode ser utilizado por ferramentas ou sistemas para fazer previsão, classificação ou agrupamentos de novos dados. Quando a mineração extrai informações sobre uma base de dados, a etapa de análise de resultados é utilizada para avaliar tais informações. Dependendo do resultado obtido, o processo de KDD pode ser reiniciado na preparação de dados para ser feito de uma maneira diferente e chegar a novos resultados.

A Mineração de Dados é uma área de estudo muito ampla englobando diversos tipos de tarefas. Basicamente, podemos classificar as tarefas de mineração de dados nas seguintes categorias:

- Modelagem de previsão;
- Análise de agrupamentos;
- Detecção de anomalias;
- Análise de associação;

A modelagem de previsão visa construir um modelo para fazer previsões. A modelagem de previsão pode ainda ser subdividida em classificação e regressão. A classificação ocorre quando a previsão é para valores discretos e, na regressão, é feita a previsão de valores contínuos. A previsão é feita com base em um modelo treinado com dados existente e nos atributos do novo objeto a ser classificado. Um exemplo de classificação seria prever se um novo cliente será bom ou mal pagador com base nos seus demais dados como salário, idade, profissão etc (SOTO; LUNA; CANO, 2016).

A análise de agrupamento é usada para agrupar objetos semelhantes. Um exemplo interessante é o agrupamento de fotos de acordo com os rostos que aparecem nelas. As técnicas de detecção de anomalias servem para determinar se determinados objetos são muito diferentes de grupos de objetos. Uma aplicação interessante é a detecção de fraudes em cartões de crédito e declarações de impostos (KHANNA; AWAD, 2015).

A análise de associação é usada para descobrir padrões que representem relacionamentos significantes entre os dados. Geralmente, os padrões descobertos são representados por regras de implicação. Um exemplo interessante é descobrir itens que são comumente comprados juntos em uma loja ou supermercado.

1.2 A linguagem de programação Python

As linguagens de programação são ferramentas essenciais para o desenvolvimento de softwares, bem como a manipulação de dados de banco de dados. Uma das linguagens de programação mais usadas no mundo é a linguagem Python[1] (TIOBE, 2021; PYPL, 2021). Além disso, desde sua criação no final da década de 1980, a linguagem passou por uma constante evolução até chegar à versão 3 em 2008 (WIKIPÉDIA, 2022c). A linguagem Python é uma linguagem aberta, multiplataforma e utilizada em uma grande variedade de aplicações (MENEZES, 2019; RAMALHO, 2015).

Na maioria das distribuições do sistema operacional Linux, o ambiente Python já vem instalado por padrão. Para outros sistemas operacionais, como Windows e MacOS, é preciso baixar e instalar esse ambiente. Contudo, no presente curso, utilizaremos a ferramenta Spyder[2] que já conta com o ambiente Python. No Linux, a instalação da ferramenta está disponível nos sistemas de pacotes (TAGLIAFERRI, 2018). No caso do Windows e MacOS, é recomendável utilizar a instalação *Anaconda* (disponível em https://www.anaconda.com/products/distribution).

O Spyder é uma ferramenta livre e multiplataforma para desenvolvimento Python. A vantagem de utilizar essa ferramenta são os diversos recursos disponíveis na mesma como editor de código, console, explorador de variáveis, depurador e ajuda. O editor de código possui um explorador de código e autocomplemento. O console permite a execução de comandos Python. Com o explorador de variáveis é possível visualizar e modificar o conteúdo das variáveis. O depurador auxilia na execução passo a passo do código e o sistema de ajuda fornece documentação de comandos da linguagem.

Alguns recursos interessantes da ferramenta Spyder são os pontos de parada, a execução passo a passo e o explorador de variáveis. Tais recursos são muito úteis para fazer a depuração do código e encontrar possíveis erros de funcionamento.

Os pontos de paradas podem ser ativados com um clique duplo sobre o número de cada linha. Aparece um pequeno círculo vermelho antes do número da linha, se o ponto de parada foi ativado. Assim, quando o código é executado no modo depurar, o Spyder interrompe a execução no ponto de parada ativado.

A execução em modo depurar é feita pelo menu *Depurar / Depurar* ou CTRL-F5 (pelo teclado). O explorador de variáveis é o painel no lado esquerdo. Quando o código for executado, você consegue ver o valor de cada variável. A execução passo a passo, no menu *Depurar / Executar linha* ou CTRL+F10, permite executar uma linha do código de cada vez. A utilização dos recursos explicados de forma conjunta pode ser muito útil para encontrar possíveis erros no funcionamento dos códigos.

Os códigos em Python são usados para resolver problemas do mundo real e, ao serem executados, precisam trabalhar com diversos tipos de dados. A Figura 1 apresenta os tipos de dados básicos da linguagem Python (PSF, 2021). Os tipos de dados são usados principalmente para representar varáveis e literais. Os Literais são os dados informados literalmente no código. As variáveis são explicadas na próxima seção.

1 https://www.python.org/

2 http://www.spyder-ide.org

Tipo	Descrição
int	Números inteiros
float	Números fracionários
bool	Valores lógicos como **True** (verdadeiro) ou **False** (falso)
str	Valores textuais

Figura 1 – Tipos básicos de dados em Python
Fonte: Elaborado pelo Autor.

Uma variável representa uma posição de memória no computador cujo conteúdo pode ser lido e alterado durante a execução do código (BORGES, 2010). Em Python, deferente de outras linguagens de programação, não precisamos declarar as variáveis. As variáveis são criadas automaticamente quando ocorre uma atribuição de valor com o operador de igualdade (**=**). A atribuição de valores a variáveis pode ser realizada com literais, outras variáveis, resultados de expressões e valores retornados por funções.

A Figura 2 mostra um exemplo de código com os quatro tipos básicos de dados. No caso dos tipos textuais, é importante colocar o valor a ser atribuído entre aspas. Aqui cabe uma observação sobre a função **print()**. Podemos colocar quantos literais ou variáveis desejarmos separados por vírgula. A função irá escrever todos na tela. No caso das variáveis, a função escreve seu valor na tela (PILGRIM, 2009).

```
n1 = 10
n2 = 2.5
nome = 'José'
teste = True
print(n1, n2, nome, teste)
```

Figura 2 – Atribuição de valores a variáveis
Fonte: Elaborado pelo Autor.

Para que não ocorram erros no código, o nome de uma variável devem obedecer às seguintes regras:

- Deve obrigatoriamente começar com uma letra;
- Não deve possuir espaços em branco;
- Não pode conter símbolos especiais, exceto o sublinhado **(_)**;
- Não deve ser uma palavra reservada (uma palavra da já existente na linguagem, como **print**, por exemplo).

Dica do Professor: A linguagem Python, assim como diversas outras linguagens, é *case sensitive*. Nessas linguagens, a mudança de maiúscula para minúscula, ou vice-versa, modifica o nome da variável (ou outro elemento). Assim, os nomes **teste** e **Teste** são considerados distintos.

Um dos principais cuidados na escrita de códigos é a ter uma boa legibilidade. Um código escrito por você deve ser facilmente entendido por outra pessoa que vá estudá-lo ou usá-lo (ou por você mesmo no futuro). Alguns fatores que influenciam diretamente o entendimento de códigos são nomes sugestivos, endentação e comentários.

Os nomes sugestivos para variáveis e outras estruturas são importantes para que você identifique de forma rápida a referida variável ou estrutura. A endentação diz respeito aos espaços em branco a esquerda para alinhar e organizar os blocos de instruções. No caso do Python, a endentação é essencial porque é por meio dela que colocamos blocos de códigos dentro de outras instruções. Já os comentários textos não executáveis com a finalidade de explicar o código (SWEIGART, 2021). Os comentários começam com o caractere **#.**

```
# -*- coding: utf-8 -*-

n1 = 10         # Número inteiro
n2 = 2.5        # Número fracionário
nome = 'José' # Textual (sempre entre aspas)
teste = True  # Variável lógica
```

Figura 3 – Código com comentários
Fonte: Elaborado pelo Autor.

A Figura 3 mostra um exemplo de código com comentários em Python. Após cada atribuição, temos comentários comuns descrevendo o tipo de dado. Já na primeira linha, temos um comentário especial usado para especificar a codificação de caracteres usada no arquivo. Nesse exemplo e nos demais ao longo do livro usaremos sempre a codificação UTF8. A linguagem Python diversos outros comentários especiais usados para outras funções.

1.2.1 Entrada e saída de dados

Quando um algoritmo recebe dados do usuário, dizemos que ocorre uma entrada de dados. De forma análoga, quando o algoritmo exibe mensagens para o usuário, acontece uma saída de dados. Em Python, a entrada de dados é efetuada com a instrução **input()**. Dentro dos parênteses colocamos um texto para ser mostrado ao usuário antes da entrada dos dados. Normalmente, usamos uma variável para receber a resposta digitada. A função **input()** sempre retorna um valor textual digitado pelo usuário. Se precisarmos de outro tipo de dado, temos que realizar uma conversão[3].

No caso da saída de dados, temos a instrução **print()**. Essa instrução recebe as informações a serem mostradas para o usuário separadas por vírgula. Se usarmos uma variável, será mostrado o valor dessa variável. Em diversas situações é importante compor mensagens adequadas para o usuário combinando literais e variáveis. A Figura 4 exibe um código usando as funções **input()** e **print()**. A instrução **int()** é usada para converter texto retornado pelo **input()** para um número inteiro.

3 Podem ocorrer erros nessa conversão, mas trataremos desse detalhe.

```
print('Bem vindo!')
# Nome do usuário
nome = input('Informe seu nome: ')
# Idade convertida para inteiro
idade = int(input('Informe sua idade: '))
print() # Escreve uma linha em branco
# Escreve mensagem usando as variáveis
print('Olá,', nome, 'sua idade é', idade)
print('Até mais!')
```

Figura 4 – Código usando **input()** e **print()**
Fonte: Elaborado pelo Autor.

A instrução **print()**, por padrão, separa os elementos recebidos com um espaço em branco e, no final, escreve o caractere de nova linha (**'\n'**). Esse comportamento pode ser modificado através dos parâmetros **sep** (texto para separar os elementos) e **end** (texto a ser colocado no final da lina). Por exemplo, se quisermos que os elementos fiquem colados e a saída não passe para a próxima linha, podemos usar uma instrução no seguinte formato **print(..., sep='', end='')**. As reticências (...) devem ser substituídas pelos elementos a serem escritos na tela.

1.2.2 Operadores e expressões

Assim como a maioria das linguagens de programação, a linguagem Python possui operadores aritméticos, relacionais e lógicos (PSF, 2021). Com esses operadores é possível criar expressões que também podem ser combinadas, principalmente, para a realização de testes.

A realização de cálculos matemáticos está entre as principais tarefas feitas por algoritmos. Para executarmos tais cálculos, devemos utilizar os chamados operadores aritméticos. A Figura 5 mostra os operadores aritméticos disponíveis na linguagem Python. A ordem precedência dos operadores aritméticos é a mesma ordem precedência da matemática. Também podem ser usados parênteses para alterar a ordem de precedência. A Figura 6 mostra um exemplo simples com expressões matemáticas usando os operadores aritméticos.

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
//	Divisão inteira
%	Resto de divisão
**	Potenciação

Figura 5 – Operadores aritméticos do Python
Fonte: Elaborado pelo Autor.

```
ni = 10 // 3
print('10 // 3 =', ni)
ni = 10 % 3
print('10 % 3 =', ni)
nr = 2.34 * 3.58
print('2.34 * 3.58 =', nr)
nr = 2.99 / 4.1
print('2.99 / 4.1 =', nr)
nr = (10.5 - 7.8) * (3.2 + 200.43)
print('(10.5 - 7.8) * (3.2 + 200.43) =', nr)
```

Figura 6 – Código usando operadores aritméticos
Fonte: Elaborado pelo Autor.

Com o conteúdo visto até o momento, podemos fazer um código para simular uma calculadora simples. A ideia é obter dois números com o usuário e mostrar os resultados das possíveis operações aritméticas entre esses números. A Figura 7 mostra uma possível solução.

```
print('Informe dois números')
n1 = float(input('Primeiro número: '))
n2 = float(input('Segundo número: '))
r = n1 + n2
print(n1, '+', n2, '=', r)
r = n1 - n2
print(n1, '-', n2, '=', r)
r = n1 * n2
print(n1, '*', n2, '=', r)
r = n1 / n2
print(n1, '/', n2, '=', r)
```

Figura 7 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

O operador **+** pode ser usado também para concatenar variáveis e literais textuais. No entanto, não é possível concatenar diretamente um elemento textual e um elemento numérico. Nesse caso, é necessário converter o valor numérico para textual antes da concatenação. O código da Figura 8 demonstra como isso pode ser feito.

```
print('Informe os dados')
nome = input('Nome: ')
sobrenome = input('Sobrenome: ')
idade = int(input('Idade: '))
mensagem = nome + ' ' + sobrenome + ', ' + str(idade)
print(mensagem)
```

Figura 8 – Código para simular calculadora simples
Fonte: Elaborado pelo Autor.

Em Python, é comum utilizar os operadores aritméticos compostos. Eles são equivalentes à aplicação do operador seguido de uma atribuição. Por exemplo, se temos uma variável **x** valendo **10** e usamos a expressão **x += 2**, estamos somando **2** à variável **x**, ou seja, é o mesmo que usar a expressão **x = x + 2**.

Os operadores relacionais são usados para comparar dois valores. Os valores a serem comparados podem ser literais, variáveis ou expressões matemáticas. O resultado

dessa comparação é um valor lógico **True** (verdadeiro) ou **False** (falso). A Figura 9 apresenta os operadores relacionais disponíveis na linguagem Python.

Operador	Descrição
==	Igual
!=	Diferente
<	Menor
<=	Menor ou igual
>	Maior
>=	Maior ou igual

Figura 9 – Operadores relacionais
Fonte: Elaborado pelo Autor.

O código da Figura 10 implementa um comparador de valores que possibilita testar todos os operadores relacionais para dois números informados. Execute o referido código no Spyder e observe o resultado das comparações para diferentes valores.

```
n1 = int(input('Informe um número: '))
n2 = int(input('Informe outro número: '))
print(n1, '==', n2, '->', n1 == n2)
print(n1, '!=', n2, '->', n1 != n2)
print(n1, '<',  n2, '->', n1 < n2)
print(n1, '<=', n2, '->', n1 <= n2)
print(n1, '>',  n2, '->', n1 > n2)
print(n1, '>=', n2, '->', n1 >= n2)
```

Figura 10 – Código comparador de valores
Fonte: Elaborado pelo Autor.

Os operadores lógicos são equivalentes aos operadores da Lógica Proposicional. Em Python, temos o operador unário **not** e os operadores binários **and** e **or**. A Figura 11 mostra o funcionamento do operador **not**. Esse operador é usado para obter o oposto de um valor lógico.

Expressão	Resultado
`not True`	`False`
`not False`	`True`

Figura 11 – Operador lógico **not**
Fonte: Elaborado pelo Autor.

A Figura 12 descreve o funcionamento do operador lógico **and**. Podemos observar que esse operador retorna **True** apenas se os dois operandos forem **True**. Ou seja, se um dos operandos for **False**, o resultado do operador também será **False**.

Expressão	Resultado
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Figura 12 – Operador lógico **and**
Fonte: Elaborado pelo Autor.

A Figura 13 exibe o funcionamento do operador lógico **or**. Esse operador retorna **False** somente se os dois operandos forem **False**. Como o resultado de um operador relacionado é um valor lógico, em muitas situações, os operadores lógicos são usados para combinar as comparações relacionais.

Expressão	Resultado
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Figura 13 – Operador lógico **or**
Fonte: Elaborado pelo Autor.

O código da Figura 14 é um exemplo de utilização dos operadores lógicos. Utilize o Spyder para executar esse código algumas vezes informando valores diferentes para as variáveis **n1** e **n2**. Observe funcionamento do código e tente calcular sozinho os valores das variáveis **p**, **q** e **r**.

```
n1 = int(input('Informe um número: '))
n2 = int(input('Informe outro número: '))
p = (n1 > n2)
q = (n1 != n2)
r = not (p or q) and (not p)
print('p =', p)
print('q =', q)
print('r =', r)
```

Figura 14 – Código com expressões lógicas e relacionais
Fonte: Elaborado pelo Autor.

1.2.3 Utilizando funções

O Python possui uma biblioteca padrão com funções que podem ser usadas diretamente no código. Além disso, podemos importar diversas outras bibliotecas que podem ser utilizadas para as mais variadas tarefas (BORGES, 2010). A Figura 15 exibe algumas funções da biblioteca padrão. Além da biblioteca padrão, abordaremos funções relacionadas ao tipo textual e também a biblioteca de funções matemáticas.

Função	Funcionamento
abs(n)	Retorna o valor absoluto de **n**
chr(n)	Retorna o caractere representado pelo número **n**
ord(c)	Retorna o código correspondente ao caractere **c**
round(n, d)	Arredonda **n** considerando **d** casas decimais
type(o)	retorna o tipo de **o**

Figura 15 – Algumas funções da biblioteca padrão do Python
Fonte: Elaborado pelo Autor.

Na linguagem Python, o tipo de dado textual (**str**) possui funções relacionadas que auxiliam na manipulação de dados desse tipo (DOWNEY, 2015). Por serem funções associadas ao tipo **str**, é preciso usá-las com o dado desse tipo. Por exemplo, se tempos uma variável **x** do tipo **str**, podem chamar uma função **lower()** com a instrução **x.lower()**. A Figura 17 contém alguns exemplos de funções de manipulação de texto. O código da Figura 20 demostra a utilização de algumas dessas funções.

Função	Funcionamento
s.find(subtexto, ini, fim)	Procura **subtexto** em **s**, começando da posição **ini** até a posição **fim**. Retorna **−1**, se o **subtexto** não for encontrado.
s.format(x_1 , ..., x_n)	Retorna **s** com os parâmetros x_1, ..., x_n incorporados e formatados.
s.lower()	Retorna o **s** com todas as letras minúsculas.
s.replace(antigo, novo, n)	Retorna o **s** substituindo **antigo** por **novo**, nas **n** primeiras ocorrências.

Figura 16 – Algumas funções associadas ao tipo **str**
Fonte: Elaborado pelo Autor.

```
nome_completo = input('Informe seu nome completo: ')
sobrenome = input('Informe seu sobrenome: ')

pos = nome_completo.find(sobrenome)

if pos != -1:
    print('Seu sobrenome começa na posição ', pos)
else:
    print('Sobrenome não encontrado')

n = float(input('Informe um número: '))
print('{n:.8f}'.format(n=n))
```

Figura 17 – Código com manipulação de dados textuais
Fonte: Elaborado pelo Autor.

Além das funções da biblioteca padrão da linguagem, podemos importar bibliotecas adicionais. Uma dessas bibliotecas é a **math** contendo funções matemáticas (CORRÊA,

2020). A importação de bibliotecas adicionais é realizada com a instrução **import**, normalmente, incluída no início do código. A Figura 18 apresenta algumas das funções disponíveis na biblioteca **math**. Já o código da Figura 19 demostra como tais funções podem ser utilizadas.

Função	Funcionamento
ceil(x)	Retorna o teto de **x**
floor(x)	Retorna o piso de **x**
trunc(x)	Retorna a parte inteira de **x**
exp(x)	Retorna $\mathbf{e}^{\mathbf{x}}$
log(x, b)	Retorna o logaritmo de **x** em uma base **b**. Se a base não for especificada, retorna o logaritmo natural de **x**
sqrt(x)	Retorna a raiz quadrada de **x**
pi	Retorna o valor de π

Figura 18 – Algumas funções da biblioteca **math**
Fonte: Elaborado pelo Autor.

```
import math

n = float(input('Informe um número: '))
x = n * math.pi
print('x = n * pi = ', n)
print('Teto de x =', math.ceil(x))
print('Piso de x =', math.floor(x))
print('Log de x na base 10 =', math.log(x, 10))
print('Raiz de x =', math.sqrt(x))
```

Figura 19 – Código com funções matemáticas
Fonte: Elaborado pelo Autor.

1.2.4 Estruturas de decisão

Para resolver diversos tipos de problemas, precisamos usar as estruturas de decisão (também conhecidas como desvios condicionais ou estruturas condicionais). As estruturas de decisão são testes efetuados para decidir se uma determinada ação deve ser realizada (BORGES, 2010).

A estrutura de decisão mais simples utiliza uma única instrução **if**, seguida por uma expressão lógica e pelo bloco de instruções a ser executado se o valor da expressão lógica for verdadeiro (CEDER, 2018). Considere, por exemplo, o problema de verificar se um aluno foi aprovado. Isso é feito testando se a nota do aluno é maior ou igual a 60. A Figura 20 mostra o código para resolver esse problema.

```
nota = float(input('Informe a nota: '))
if nota >= 60:
    print('Aprovado')
print('Boas férias')
```

Figura 20 – Estrutura de decisão simples para testar aprovação de aluno
Fonte: Elaborado pelo Autor.

A endentação deve ser feita corretamente para especificar quais instruções estão dentro da estrutura condicional. Utilizamos quatro espaços para endentar um bloco de instruções. Observe que, depois da expressão lógica seguida por dois pontos (**:**), começa o bloco de instruções do **if**. Esse bloco deve ser obrigatoriamente endentado. No código, a mensagem “Aprovado” é escrita na tela somente quando a nota é maior ou igual a 60. Por outro lado, a mensagem “Boas férias” é mostrada incondicionalmente, pois não está dentro do desvio condicional.

Na estrutura de decisão simples, executamos alguma ação somente se a expressão lógica testada for verdadeira. Por outro lado, em certas situações, também precisamos realizar alguma medida se o teste for falso. Nesse caso, precisamos utilizar uma estrutura de decisão composta acrescentando a instrução **else** após o bloco de código da instrução **if**. A instrução **else** é seguida por outro bloco de código que será executado quando o teste for falso.

Como exemplo, vamos reconsiderar o problema de aprovação do aluno e escrever uma mensagem quando o mesmo for reprovado. A Figura 21 mostra o novo código contendo a mensagem “Reprovado” quando o aluno tem nota menor que 60.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 21 – Código com estrutura de decisão composta
Fonte: Elaborado pelo Autor.

Na estrutura de decisão simples, ocorre um único teste e o fluxo de execução do algoritmo pode seguir por um ou dois caminhos. Em determinadas situações, precisamos de algoritmos com vários testes e, por consequência, vários fluxos de execução. Para fazer isso, temos que inserir uma estrutura de decisão dentro de outra estrutura de decisão. Nesse caso, dizemos que as estruturas de decisão estão aninhadas.

Como exemplo, vamos considerar mais uma vez o problema de aprovação de aluno, mas, agora, vamos tratar as seguintes situações:

- Se o aluno tiver nota maior ou igual a 60, será aprovado;
- Se a nota for menor que 40, ao aluno é reprovado;
- Por fim, se a nota for maior ou igual a 40 e menor do que sessenta o aluno está de recuperação.

A Figura 22 apresenta o código para resolver o problema considerando as novas situações. Observe que, dentro do primeiro **else**, quando o aluno não é aprovado, ocorre um segundo teste para verificar se o aluno foi reprovado ou está de recuperação. O código pode ser escrito de outras formas, mudando a ordem dos testes, e obter o mesmo resultado.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else:
    if n >= 40:
        print('Reavaliação')
    else:
        print('Reprovado')
print('Boas férias')
```

Figura 22 – Código com estruturas de decisão aninhadas
Fonte: Elaborado pelo Autor.

Observe que, no código da Figura 22, tivemos que endentar mais o **if** mais interno. Caso tenhamos muitas estruturas de decisão aninhadas, a legibilidade pode ficar prejudicada. Assim, outra maneira de usar as estruturas de decisão aninhadas é de forma consecutiva, como mostrado na Figura 23. Nesse caso, incluímos um **if** imediatamente após o **else** para fazer o segundo teste. Também é possível juntar o **else** com o **if** formando a instrução **elif**, como é mostrado no código da Figura 24. Essa contração é especialmente útil quando temos muitos testes consecutivos a serem feitos.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
else if n >= 40:
    print('Reavaliação')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 23 – Código com estruturas de decisão consecutivas
Fonte: Elaborado pelo Autor.

```
n = float(input('Informe a nota: '))
if n >= 60:
    print('Aprovado')
elif n >= 40:
    print('Reavaliação')
else:
    print('Reprovado')
print('Boas férias')
```

Figura 24 – Código com estruturas de decisão consecutivas usando **elif**
Fonte: Elaborado pelo Autor.

Para exemplificar o uso de estruturas de decisão, consideraremos a solução de equações de segundo grau no formato $Ax^2 + Bx + C = 0$. O algoritmo para resolver esse problema deve fazer o seguinte:

1) Ler os termos A, B e C;

2) Garantir que temos uma equação de segundo grau testando se A é diferente de zero;

3) Se for uma equação de segundo grau, calculamos o delta ($\Delta = B^2 - 4 \times A \times C$);

4) Após o cálculo, temos três situações para o delta:

- Se o delta for menor que zero, a equação não possui raízes;
- Se o delta for igual a zero, então a equação possui uma única raiz;
- Por fim, se o delta é maior que zero temos duas raízes $X_1 = \frac{-B+\sqrt{\Delta}}{2 \times A}$ e $X_2 = \frac{-B-\sqrt{\Delta}}{2 \times A}$.

A Figura 25 apresenta o código para resolver equações de segundo grau.

```
import math
print('Informe os termos da equação Ax² + Bx +C')
a = float(input('A: '))
b = float(input('B: '))
c = float(input('C: '))
if a == 0:
    print('Não é uma equação de segundo grau')
else:
    delta = b**2 - 4 * a * c
    if delta < 0:
        print('A equação não tem raízes')
    elif (delta == 0):
        x1 = b * (-1) / 2 * a
        print('A equação possui a raiz:', x1, '')
    else:
        raiz_delta = math.sqrt(delta)
        x1 = (b * (-1) + raiz_delta)/ 2 * a
        x2 = (b * (-1) - raiz_delta)/ 2 * a
        print('A equação possui duas raízes:')
        print('x1 =', x1)
        print('x2 =', x2)
```

Figura 25 – Código para resolver equações de segundo grau
Fonte: Elaborado pelo Autor.

1.2.5 Estruturas de repetição

As estruturas de repetição possibilitam executar repetidamente blocos de instruções por um número definido de vezes ao até que uma dada condição seja atendida.

O laço de repetição **while** é indicado quando não sabe o número de repetições a serem executadas. Por exemplo, a listagem dos números pares menores do que um número informado pelo usuário. Nesse problema, não é possível saber o número de repetições, pois não tem como prever o número que o usuário informará.

No laço **while**, as repetições ocorrem enquanto uma determinada condição for verdadeira (CORRÊA, 2020). Assim, é importante garantir que a condição de parada realmente aconteça e o laço não fique repetindo indefinidamente. Vamos considerar

novamente o problema da listagem dos números pares. Nesse caso, a condição de parada pode ser quando chegarmos a um número maior do que o número informado pelo usuário.

A Figura 26 mostra o código para listagem de números pares. Usamos a variável **atual** para armazenar o número atual, começando pelo zero. O laço **while** usa a condição de parada **atual < n**, para verificar se o número atual ainda é menor que **n** informado pelo usuário.

```
atual = 0
n = int(input('Informe um número: '))
while atual < n:
    print(atual)
    atual += 2
```

Figura 26 – Listagem de números pares
Fonte: Elaborado pelo Autor.

A estrutura de repetição **while** possui uma condição que é testada antes mesmo de executar a primeira repetição. Enquanto essa condição for verdadeira, as repetições continuam acontecendo. Se o usuário informar zero, por exemplo, não acontece nenhuma repetição. Dentro do laço, somamos mais dois ao número atual para chegarmos ao próximo número par.

O laço de repetição **for** é indicado quando se sabe o número de repetições a serem feitas. Basicamente, o laço **for** percorre de forma automática os elementos de uma estrutura de lista. A maneira mais comum de utilizar o laço **for** é com a função **range()**. Essa função recebe um número inteiro **n** e gera um intervalo de números de 0 até **n – 1** (CORRÊA, 2020).

Para exemplificar o uso do laço **for**, vamos considerar o problema de somar 10 números informados pelo usuário. A Figura 27 mostra o código para resolver esse problema. Observe que, no laço, for usamos a função **range(10)** e a variável **cont** para percorrer os números do intervalo gerado pela função **range()**. Assim, na primeira repetição, a variável **cont** vale **0**, na segunda, vale **1**, e assim por diante até assumir o valor **9**.

```
soma = 0
for cont in range(10):
    n = float(input('Informe um número: '))
    soma += n
print(soma)
```

Figura 27 – Soma de 10 números usando o laço **for**
Fonte: Elaborado pelo Autor.

É possível notar, no código para soma dos 10 números, que a única função da variável **cont** é controlar as repetições do laço **for**. Quando temos uma variável que não é usada em nenhuma outra parte do código, podemos substituir essa variável pela variável anônima **_** (sublinhado).

Outro detalhe importante é a função **range()**. Além do fim do intervalo, podemos definir o início e a periodicidade dos números. Se usarmos, por exemplo, a instrução **range(2, 6)**, será retornado o intervalo de números "**2, 3, 4, 5**", ou seja, o primeiro número

é o início e o segundo número é o fim do intervalo. Lembrando que o número do fim não é incluído no intervalo. Quando incluímos um terceiro número, definimos a periodicidade dos números. Como exemplo, se usarmos a instrução **range(3, 19, 4)** teremos a sequência "**3, 7, 11, 15**". A sequência começa em **3**, e os demais números são obtidos somando **4** ao número atual, até atingir o fim do intervalo. Além disso, no lugar dos números, podemos usar qualquer variável ou expressão que retorne um número inteiro. Também podemos obter intervalos em ordem decrescente, por exemplo, a instrução **range(5,0,-1)** retorna a sequência "**5, 4, 3, 2, 1**".

O laço de repetição **while** precisa testar a condição de parada antes mesmo da primeira repetição. Entretanto, em diversos momentos, precisamos executar a primeira repetição antes de testar a condição de parada. Nesse caso, podemos criar um laço com a instrução **while True** e utilizar a instrução **break** para finalizar o laço de repetição.

Como exemplo, vamos considerar a soma de uma quantidade indeterminada de números informados pelo usuário. Devemos parar de somar apenas quando o usuário informar o número **0** (zero). A Figura 28 apresenta o código para resolver esse problema. É importante tomar um certo cuidado com laços **while True**, temos que garantir que a instrução **break** será executada em algum momento e o laço não fique repetido indefinidamente.

```
soma = 0
while True:
    n = float(input('Informe um número: '))
    if n == 0:
        break
    soma = soma + n
print('Soma dos números:', soma)
```

Figura 28 – Soma indefinida de números
Fonte: Elaborado pelo Autor.

Vamos considerar agora uma modificação no problema de somar números. Além do que já foi mencionado, suponha que os números negativos não devam ser somados. Diante disso, podemos usar a instrução **continue** para ignorar os números negativos e *pular* para a próxima repetição do laço (CEDER, 2018). O código da Figura 29 mostra a solução para a modificação do problema. As instruções **break** e **continue** podem ser usadas tanto no laço **while** quanto no laço **for**.

```
soma = 0
while True:
    n = float(input('Informe um número: '))
    if n < 0:
        continue
    if n == 0:
        break
    soma = soma + n
print('Soma dos números:', soma)
```

Figura 29 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

Assim como as estruturas de decisão, as estruturas de repetição também são extensamente usadas para resolver diversas categorias de problemas. Um problema

interessante para ser resolvido com laço de repetição é o cálculo de máximo divisor comum (MDC) com a técnica de Euclides (WIKIPÉDIA, 2021). Lembrando que o MDC de números é o maior número que os divide sem deixar resto. Basicamente, a técnica de Euclides consiste nos seguintes passos:

- Dividimos o maior número pelo menor e verificamos se o resto da divisão é zero;
- Em caso afirmativo, o menor número é o MDC;
- Caso contrário, substituímos o maior número pelo menor, o menor número pelo resto da divisão e repetimos o processo.

Repare que a repetição do processo no terceiro ponto caracteriza uma estrutura de repetição. Como exemplos, demonstraremos como calcular o MDC de 144 e 56. O processo é o seguinte:

- Começamos dividindo 144 por 56. Como o resto da divisão é 32, vamos considerar os números 56 e 32 e repetir o processo;
- Dividimos 56 por 32 e temos 24 como resto. Agora, consideramos 32 e 32 e dividimos novamente;
- Na divisão de 32 por 24, chegamos a 8 como resto. Assim, a próxima divisão será 24 por 8;
- Por fim, dividimos 24 por 8 e temos zero como resto. Logo, o MDC é o número 8.

O laço de repetição mais adequado para implementar o algoritmo de Euclides é o **while**, pois não tem como prever quantas divisões precisam ser feitas. A Figura 30 mostra o código do algoritmo de Euclides.

```
print('Informe dois números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))

if n1 < 1 or n2 < 1:
    print('Números inválidos para MDC')
else:
    while True:
        resto = n1 % n2
        print (n1, '/' , n2, '-> resto:', resto)
        if resto == 0:
            break
        n1 = n2
        n2 = resto
    print('O MDC é ', n2)
```

Figura 30 – Soma indefinida de números (exceto negativos)
Fonte: Elaborado pelo Autor.

A condição de parada do laço é **True**. Todavia, dentro do laço, testamos se o resto da última divisão é zero e interrompemos o laço com a instrução **break**. Antes do laço de repetição, usamos uma estrutura de decisão para testar se os números são válidos (positivos maiores que zero). O **print()** dentro do laço não é necessário, ele foi incluído apenas para mostrar as divisões realizadas do processo.

1.2.6 Tratamento de exceções

Considere uma instrução **n = float(input('Informe um número'))**. Estamos pedindo ao usuário para digitar um número. Todavia, se for digitado qualquer texto que não possa ser convertido, para número ocorrerá um erro em tempo de execução. Assim, ainda que o código esteja correto, podem ocorrer situações que causam erros. Esses erros são chamados de exceções e podem ser contornados usando a instrução **try** da linguagem Python (PSF, 2021).

O código da Figura 31 é um exemplo simples de tratamento de exceção. Dentro da cláusula **try**, colocamos as instruções de código que deveriam ser executadas normalmente. Se ocorrer algum erro, o fluxo de execução é direcionado para a cláusula **except** (CEDER, 2018). Execute o código e informe números corretos e incorretos. Você poderá ver que, ao ocorrer um erro, será executada a instrução da cláusula **except**.

```
try:
    print('Informe dois números')
    n1 = float(input('n1: '))
    n2 = float(input('n2: '))
    r = n1 / n2
    print(n1, '/', n2, '=', r)
except:
    print('Ocorreu um erro.')
```

Figura 31 – Tratamento simples de exceção
Fonte: Elaborado pelo Autor.

No código anterior, se ocorrer um erro, a execução será finalizada. Assim, caso o usuário quiser tentar novamente, será preciso executar o código novamente. Uma solução para esse problema é colocar o tratamento de exceção dentro de um laço de repetição. Dessa maneira, se ocorrer um erro, o usuário poderá tentar novamente sem ter que executar todo o código mais uma vez. Essa solução é mostrada na Figura 32. Como foi usado um laço **while True**, temos que incluir a instrução **break** no final da cláusula **try** para, na ausência de erros, finalizar o laço. Por outro lado, se ocorrer algum erro, a cláusula **except** é disparada e, depois de sua execução, o laço continua as repetições.

```
import math
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except:
        print('Ocorreu um erro! Tente novamente.')
print(n1, '/', n2, '=', r)
```

Figura 32 – Tratamento de exceção com repetição
Fonte: Elaborado pelo Autor.

Em nosso código, além do erro de digitação pelo usuário, pode acontecer o erro de divisão por zero na instrução **r = n1 / n2**. A instrução **except** captura todos os tipos de erros. Se for necessário tratar erros diferentes, temos que colocar uma cláusula **except**

para cada tipo de erro. Para descobrir as classes dos erros, podemos usar a função **type()** como mostrado no código da Figura 33.

```
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except Exception as e:
        print('Ocorreu o seguinte erro:', type(e))
print(n1, '/', n2, '=', r)
```

Figura 33 – Descobrindo classes de erros
Fonte: Elaborado pelo Autor.

Se for digitado um número incorretamente, temos a exceção **ValueError**. Se o segundo número for zero, temos a exceção **ZeroDivisionError**. O código da Figura 34 mostra como fazer o tratamento de exceção específico para cada uma dessas classes de erro.

```
while True:
    try:
        print('Informe dois números')
        n1 = float(input('n1: '))
        n2 = float(input('n2: '))
        r = n1 / n2
        break
    except ValueError as e:
        print(e)
        print('Número inválidos! Tente novamente.')
    except ZeroDivisionError as e:
        print(e)
        print('Divisão por zero! Tente novamente.')
print(n1, '/', n2, '=', r)
```

Figura 34 – Tratamento de exceções específicas
Fonte: Elaborado pelo Autor.

A instrução **try** possui também as cláusulas **else** e **finally**. A cláusula **else** pode ser usada para realizar alguma ação quando não ocorrer erros. Já a cláusula **finally** é executa incondicionalmente, ocorrendo erros ou não. A cláusula **finally** é útil para executar ações de limpeza como fechamento de arquivos.

1.2.7 Modularização

A modularização consiste em dividir um algoritmo em partes menores chamadas sub-rotinas ou funções com o intuito de facilitar o desenvolvimento e a manutenção de código (BORGES, 2010). Considerando um problema grande a ser resolvido, as funções representam pequenas partes do problema maior com menor complexidade. Além disso, as funções podem ser trabalhadas de forma independente e a localização de erros se torna mais fácil.

Um código deve ser desenvolvido de forma modular sempre que possível. Assim, se um mesmo trecho de código é executado em pontos diferentes do programa, podemos criar uma função para executar esse código uma única vez e chamar a função quando necessário. As principais vantagens da modularização são as seguintes:

- Evita a reescrita desnecessária de códigos similares;
- Melhora legibilidade do código;
- Permite desenvolvimento em equipe (cada programador cuida de um trecho do código);
- As funções podem ser testadas isoladamente para verificação de erros;
- A manutenção se torna mais fácil, apenas algumas partes podem precisar de alteração.

Funções são trechos de código que executam determinada tarefa ao serem chamados e depois retornam o controle para o ponto em que foram chamados (CEDER, 2018). Apesar de ainda não termos criados nossas próprias funções, já escrevemos diversos códigos utilizando funções prontas como **input()** e **print()**. Quando chamamos uma função devemos informar os parâmetros necessários. Isto significa que se vamos usar uma função **teste()** que recebe um parâmetro **int** e outro **str**, então devemos usar a instrução **teste(a, b)**, onde **a** é do tipo **int** e **b** é do tipo **str**.

A criação de novas funções é feita com a instrução **def**, seguida pelo nome da função e seus parâmetros entre parênteses e dois pontos (**:**). A linha com a instrução **def** é chada de declaração ou cabeçalho da função. Tanto o nome da função e seus parâmetros devem obedecer às mesmas regras dos nomes de variáveis. Após a declaração, vem o chamado corpo da função com suas instruções endentadas. No corpo da função, quando a função precisar tiver que retornar algum valor, é usada a instrução **return**.

O código da Figura 35 contém a função **saudacao()** que não possui parâmetros e não retorna valores. É importante incluir uma linha em branco entre o corpo da função e a próxima instrução para manter uma boa legibilidade no código. É importante frisar que as instruções **print()** da função não caracterizam retorno de valor. Quando uma função retorna um valor, podemos atribuir ser resultado a uma variável. Esse é o caso da função **input()**, por exemplo.

Na última linha do código, acontece chamada à função. Essa chamada direciona o fluxo de execução do código para a primeira instrução da função. Após a execução de todas as linhas da função, o fluxo de execução retorna para a linha onde ocorreu a chamada. Além disso, as funções são executadas somente quando são chamadas. No código da Figura 35, por exemplo, a primeira instrução a ser executada é **saudacao()**.

```
def saudacao():
    print('*******************************')
    print('*          BEM VINDO          *')
    print('*******************************')

saudacao()
```

Figura 35 – Função **saudacao()**
Fonte: Elaborado pelo Autor.

No código da Figura 35 criamos apenas uma função. Porém, para a grande maioria dos problemas precisar escrever códigos com várias funções. Além disso, podemos criar uma função pode chamar outras funções. Uma função pode declarar variáveis para serem utilizadas apenas internamente. Estas variáveis, assim como os parâmetros da função, são chamadas de variáveis locais enquanto as variáveis de fora da função são as variáveis globais. É importante que as variáveis sejam locais sempre que possível.

No código anterior, podemos ver que a função **saudacao()** não recebe nenhum parâmetro e nem dados do usuário. Isso faz com que essa função sempre realize as mesmas ações. Na maioria das vezes, precisamos criar funções parametrizáveis que realizam ações específicas conforme os parâmetros recebidos. Na prática, os parâmetros são como variáveis já inicializadas recebidas pelas funções.

A Figura 36 mostra um código com a função **cubo()**. A função recebe como parâmetro o número **num** e calcula o cubo do mesmo. Contudo, essa função ainda pode ser melhorada. Observe que não foi usada a instrução **return** para retornar o resultado do cálculo. A função está simplesmente escrevendo na tela. Assim, se modificarmos a função para retornar o resultado, teremos uma função mais útil. Isso porque podemos usar a função em qualquer lugar, sem ter que escrever na tela e usar seu resultado da maneira que for mais apropriada.

```
def cubo(num):
    cubo = num * num * num
    print(num, 'ao cubo é', cubo)

n = float(input('Informe um número: '))
calcula_cubo(n)
```

Figura 36 – Função **cubo()** sem retorno
Fonte: Elaborado pelo Autor.

```
def cubo(num):
    return num * num * num

n = float(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
print(n, 'elevado a nona é', cubo(cubo(n)))
```

Figura 37 – Função **cubo()** com retorno
Fonte: Elaborado pelo Autor.

A Figura 37 mostra a modificação da função **cubo()** com o retorno do resultado. Repare que podemos usar diretamente a função dentro do **print()**. Além disso, podemos usar o resultado da função em qualquer expressão numérica, como na instrução **cubo(cubo(n))**.

Na chamada de funções é possível, passar parâmetros em ordem diferente daquela especificada da declaração, desde que existam atribuições aos nomes dos parâmetros. Também podemos criar funções com parâmetros opcionais. Os parâmetros opcionais devem ser os últimos e devem ter um valor padrão já atribuído. Assim, na chamada da função, se o parâmetro opcional não for passado, será usado o valor padrão.

```
def juros(capital, taxa, tempo=12):
    return (capital * taxa * tempo) / 100

print('Cálculo de juros')
cap = float(input('Capital: '))
tax = float(input('Taxa: '))
tem = input('Tempo (deixe em branco para o padrão de 12): ')
if tem == '':
    jur = juros(cap, tax)
else:
    tem = float(tem)
    jur = juros(taxa=tax, capital=cap, tempo=tem)
print('O valor dos juros é', jur)
```

Figura 38 – Código com Nomes de parâmetros e parâmetros opcionais
Fonte: Elaborado pelo Autor.

A Figura 38 exibe um código com parâmetros opcionais e chamada de função usando os nomes dos parâmetros. Na função **juros()**, temos o parâmetro **tempo** como opcional, seu valor padrão é **12**. Na instrução **jur = juros(cap, tax)**, é feita uma chamada de função sem usar os nomes de parâmetros. Nesse caso, temos que manter os parâmetros na ordem correta. Primeiro, o parâmetro **capital** e, depois, o parâmetro **taxa**. Nessa linha o parâmetro opcional **tempo** não foi usado. Já na instrução **jur = juros(taxa=tax, capital=cap, tempo=tem)**, ocorre uma chamada de função usando os nomes dos parâmetros. Observe que eles não estão na mesma ordem da declaração, mas, como usamos os nomes, isso não é um problema.

A recursão ocorre quando uma função chama a si mesma. Na prática, é possível usar laços de repetição para substituir recursões. Contudo, em muitas situações, funções recursivas podem ser mais intuitivas do que laços de repetição. Uma observação importante é que precisamos ter cuidado com a condição de parada, assim como fazemos nos laços de repetição. Caso contrário, podemos cair em uma recursão infinita que a função faz chamadas a ela mesma indefinidamente.

```
def fat(num):
    if num <= 1:
        return 1
    return num * fat(num - 1)

n = int(input('Informe um número: '))
print('O fatorial de', n, 'é', fat(n))
```

Figura 39 – Função recursiva **fat()**
Fonte: Elaborado pelo Autor.

O código da Figura 39 mostra apresenta a função recursiva **fat()**. O **if** faz o teste da condição de parada. Quando o parâmetro de entrada for menor ou igual a um, seu fatorial

será um. Na última linha da função, ocorre a chamada recursiva, usando a equivalência **n! = n * (n-1)!**.

Dependendo da quantidade de código, pode ser interessante a criar módulos para agrupar as funções correlacionadas. Normalmente, os módulos possuem apenas definições de funções ou outras estruturas e o código principal controla o fluxo de execução do código importando os módulos e chamando suas funções.

Para exemplificar a criação de módulos, consideraremos o problema de calcular o cubo e o fatorial de um número. A Figura 40 mostra o módulo com as funções que realizam esses cálculos.

```
def cubo(num):
    return num * num * num

def fat(num):
    if num <= 1:
        return 1
    return num * fat(num - 1)
```

Figura 40 – Módulo **mat.py**
Fonte: Elaborado pelo Autor.

Na Figura 41 temos o código do módulo principal que controla o fluxo de execução e efetua as chamadas as funções do módulo **mat.py**. É importante que ambos módulos sejam salvos na mesma pasta ou diretório. No código, especificamos quais funções deveriam ser importados usando a instrução **from ... import ...**. Poderíamos fazer de outra maneira, fazendo somente a importação do módulo **mat** e chamando as funções no formato **mat.cubo(...)** e **mat.fat(...)**.

```
from mat import cubo, fat

n = int(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
print('O fatorial de', n, 'é', fat(n))
```

Figura 41 – Módulo **principal.py**
Fonte: Elaborado pelo Autor.

A linguagem Python é uma linguagem interpretada, ou seja, não é preciso compilar o código-fonte para gerar arquivos executáveis. No caso do Linux, podemos criar scripts executáveis usando o comentário especial **#!/usr/bin/env python3** na primeira linha do módulo principal. Além disso, temos que dar permissão de execução ao arquivo. No caso do Windows, podemos associar a abertura de arquivos **.py** ao programa **pythonw.exe** disponível na pasta de instalação do Spyder. Assim, os scripts podem ser executados diretamente através do gerenciador de arquivos ou linha de comando. A Figura 42 mostra um script executável juntando os dois módulos do exemplo anterior.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def cubo(num):
    return num * num * num

n = int(input('Informe um número: '))
print(n, 'ao cubo é', cubo(n))
```

Figura 42 – Exemplo de script executável
Fonte: Elaborado pelo Autor.

1.2.8 Decomposição de problemas

A principal vantagem da modularização é a possibilidade de usarmos decomposição de problemas para resolver problemas mais complexos. Assim, podemos quebrar um problema maior em pequenas partes. Cada uma dessas partes pode ser resolvida com uma função específica. Ao final, juntamos as funções para solucionar o problema inicial.

Para exemplificar a decomposição de problemas construiremos uma calculadora de expressões considerando os seguintes pontos:

- A calculadora consiste em um console onde o usuário digita comandos;
- O usuário pode digitar expressões aritméticas para serem calculadas, ver o histórico de expressões ou sair;
- Os comandos **h** e **s** são usados respectivamente para histórico e sair;
- Deve ser realizado tratamento de exceção no cálculo da expressão digitada para garantir que a mesma não possuir erros;
- O histórico deve guardar apenas as expressões sem erros.

A descrição da calculadora pode parecer muito complexa, mas vamos decompô-la em problemas menores para facilitar a implementação. Primeiro construiremos uma função que recebe o texto da expressão e calcula o resultado. Para facilitar um pouco mais a nossa vida, usaremos a função **eval()** do Python. Essa função é capaz de executar um texto como se fosse os códigos em Python. No caso de uma expressão aritmética, a função **eval()** retorna o resultado dessa expressão. Entretanto, se a expressão for possuir algum erro de sintaxe, ocorrerá um erro. Portanto, temos que fazer o tratamento de exceções ao executarmos a função **eval()**.

```
def calcula(expr):
    try:
        return eval(expr)
    except:
        print('Expressão inválida!')
        return None
```

Figura 43 – Função **calcula()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

A Figura 43 exibe o código da função **calcula()**. Usamos a instrução **try** para fazer o tratamento de exceção. No caso de algum erro, mostramos a mensagem de expressão inválida e retornamos o valor **None**. Esse valor é um valor nulo que podemos testar ao receber o resultado da função.

```
def historico(expr, res):
    global HIST
    if res is not None:
        HIST += '\n\n'+ expr
        HIST += '\n' + str(res)
```

Figura 44 – Função **historico()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Agora criaremos uma função para atualizar o histórico da calculadora. Seu código é mostrado na Figura 44. A instrução **global HIST** é usada para podermos alterar a variável global **HIST**. Tal variável será declarada posteriormente para armazenar o histórico. Nesse problema estamos adotando maiúsculas para variáveis globais e minúsculas para variáveis locais. A função **historico()** recebe os parâmetros **expr** (texto da expressão calculada) e **res** (resultado do cálculo). O **if** faz um teste para verificar se a expressão é válida. O teste **res is not None** é usado para verificar se o resultado da expressão (**res**) é diferente de **None**. Dentro do **if** apenas adicionamos a expressão e seu resultado ao histórico. Utilizamos o **'\n'** para separar as linhas do histórico.

Depois de construirmos as duas funções auxiliares, vamos escrever a função principal da calculadora que deverá gerenciar o fluxo de execução e chamar as funções auxiliares quando necessário. A Figura 45 mostra o código dessa função.

```
def principal():
    while True:
        print('Informe a expressão matemática')
        print('(h para histórico, s para sair)')
        expr = input()
        if expr.lower() == 's':
            break
        if expr.lower() == 'h':
            print(HIST, '\n')
        else:
            res = calcula(expr)
            historico(expr, res)
            print(res, '\n')
```

Figura 45 – Função **principal()** da calculadora de expressões
Fonte: Elaborado pelo Autor.

Temos um laço de repetição para que o usuário possa digitar quantas expressões desejar. A expressão **expr.lower() == 's'** testa se o usuário informou o comando **s** e interrompe o laço de repetição. No caso do comando **h**, o teste é feito com a expressão **expr.lower() == 'h'** e apenas mostramos o histórico de cálculos guardado na variável **HIST**. Repare que não precisamos da instrução **global HIST** nessa função porque estamos apenas lendo o conteúdo da variável global.

Se o usuário não informar os comandos **s** ou **h**, partimos para o cálculo da expressão. O resultado da expressão é guardado na variável **res**. Em seguida, chamamos a função de atualizar o histórico. Por fim, mostramos o resultado do cálculo da expressão.

As funções desenvolvidas até agora não são suficientes para que a calculadora funcione. Temos que declarar a variável global **HIST** e chamar a função **principal()**. A Figura 46 exibe o código completo da calculadora de expressões. Os comentários das duas primeiras linhas permitem que o código seja executado na forma de script. A variável global **HIST** é inicializada com um texto vazio. A instrução **if __name__ == '__main__'** utiliza a variável especial **__name__** do Python para verificar se o módulo foi executado como um script. Quando isso acontece o conteúdo dessa variável é **'__main__'**.

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  HIST = ''
5
6  def calcula(expr):
7      try:
8          return eval(expr)
9      except:
10         print('Expressão inválida!')
11         return None
12
13 def historico(expr, res):
14     global HIST
15     if res is not None:
16         HIST += '\n\n'+ expr
17         HIST += '\n' + str(res)
18
19 def principal():
20     while True:
21         print('Informe a expressão matemática')
22         print('(h para histórico, s para sair)')
23         expr = input()
24         if expr.lower() == 's':
25             break
26         if expr.lower() == 'h':
27             print(HIST, '\n')
28         else:
29             res = calcula(expr)
30             historico(expr, res)
31             print(res, '\n')
32
33 if __name__ == '__main__':
34     principal()
```

Figura 46 – Código completo da calculadora de expressões
Fonte: Elaborado pelo Autor.

1.2.9 Coleções

Os tipos de dados básicos permitem armazenar um único valor de um tipo de dado. Entretanto, em muitas situações, precisamos de estruturas de dados mais complexas,

chamadas de coleções de dados, para agrupar diversos elementos de dados. A vantagem dessas estruturas está na facilidade de acesso e manipulação desses elementos de dados. Os principais tipos de coleções de dados são listas, tuplas, dicionários e conjuntos (PSF, 2021).

Para ilustrarmos a necessidade do uso de coleções de dados, vamos considerar o código da Figura 47. O código recebe o preço de 10 produtos e calcula o preço médio dos mesmos. Agora imagine que fosse necessário listar os produtos com preço acima da média. Como fazer isto de uma forma eficiente?

```
soma = 0
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
media = soma / 10
print('O preço médio é', media)
```

Figura 47 – Cálculo do preço médio de produtos
Fonte: Elaborado pelo Autor.

O problema das preços acima da média é que usamos os preços dos produtos para calcular a média e depois precisamos novamente desses preços. Uma solução seria declarar 10 variáveis uma para cada produto. Porém, imagine uma lista de 50 produtos ou mais, a declaração de variáveis individuais não é viável na prática.

Outra solução seria pedir para o usuário para informar os preços, realizar o cálculo da média e solicitar os preços novamente. Essa solução também não é eficiente, pois o usuário precisa digitar a lista de preços duas vezes. Além da sobrecarga do usuário, pode ocorrer uma digitação errada na segunda vez. A melhor solução para o problema é a utilizar uma coleção de dados para guardar os preços de todos os produtos e, depois do cálculo da média, revisitar a coleção para buscar aqueles produtos com preço acima da média.

As listas são estruturas dinâmicas e sequenciais de elementos (CORRÊA, 2020). Por ser dinâmica, podemos alterar a lista com a inclusão ou remoção de elementos. Além disso, por sua característica sequencial, dizemos que os elementos são indexados. Assim, podemos ler ou modificar os elementos da lista usando seu índice. O índice de um elemento é a sua posição dentro da lista, com a numeração começando em 0 (zero). A Figura 48 mostra uma representação de lista de números com suas respectivas posições. O elemento **10**, por exemplo, está na posição **4**.

40	35	61	89	10	52
0	1	2	3	4	5

Figura 48 – Representação de lista de números
Fonte: Elaborado pelo Autor.

Em Python, a declaração de listas é feita colocando os elementos da lista separados por vírgula dentro de colchetes, por exemplo, **minha_lista = [1, 2, 3, 4, 5]**. Também podemos criar listas vazias simplesmente abrindo e fechando colchetes, por exemplo,

lista_vazia = []. Além disso, as listas possuem diversas funções para facilitar sua manipulação. A Figura 49 apresenta algumas dessas funções.

Função	Funcionamento
l.append(x)	Adiciona o elemento **x** no final da lista **l**
l.insert(p, x)	Insere o elemento **x** na posição **p** da lista **l**
l.pop(p)	Remove e retorna o elemento da posição **p** de **l** (se **p** não for informado, a última posição é considerada)
l.clear()	Remove todos os elementos da lista **l**

Figura 49 – Algumas funções de listas
Fonte: Elaborado pelo Autor.

O código da Figura 50 mostra uma possível solução para o problema dos produtos com preço acima da média utilizando listas. Criamos uma lista vazia para guardar os preços dos produtos (**lista_precos**). Dentro do primeiro laço de repetição, os preços são adicionados no final da lista com a função **append()**. O Segundo laço de repetição percorre os preços da lista e mostra apenas aqueles acima da média.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os preços acima da média são:')
for preco in lista_precos:
    if preco > media:
        print(preco)
```

Figura 50 – Utilização de listas para resolver o problema dos preços acima da média
Fonte: Elaborado pelo Autor.

O código anterior mostra apenas os preços acima da média, mas não exibe os produtos com esses preços. Para resolver essa questão, no segundo laço, temos que percorrer as posições da lista usando o range. Essa solução é apresentada na Figura 51. Como estamos varrendo a lista pelos índices, usamos **lista_precos[cont]** para acessar o preço na posição **cont**. O código usa a função **range()** para gerar o intervalo de índices da lista. Uma alternativa é usar a função **enumerate()** que numera os elementos da lista e os retorna junto com seus índices. Essa solução alternativa é mostrada na Figura 52.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont in range(10):
    if lista_precos[cont] > media:
        print('Produto', cont+1)
        print('Preço: ', lista_precos[cont])
```

Figura 51 – Solução mostrando os produtos com preços acima da média
Fonte: Elaborado pelo Autor.

```
soma = 0
lista_precos = []
print('Informe o preço dos produtos')
for cont in range(10):
    mensagem = 'Produto ' + str(cont+1) + ': '
    preco = float(input(mensagem))
    soma += preco
    lista_precos.append(preco)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for cont, preco in enumerate(lista_precos):
    if preco > media:
        print('Produto', cont+1)
        print('Preço: ', preco)
```

Figura 52 – Solução mostrando os produtos com preços acima da média usando **enumerate()**
Fonte: Elaborado pelo Autor.

Em algumas situações, precisamos inicializar listas com certos valores ou com um determinado número de posições. Nesses casos, podemos fazer isso com a função **range()** ou com o operador *. A Figura 53 mostra alguns exemplos de inicialização de listas. A função **list()** converte o intervalo gerado pelo **range(10)** para o formato de lista.

```
# Lista inicializada com números de 0 a 9
lista = list(range(10))
print(lista)

# Lista de 10 posições com valores 0
lista = [0]*10
print(lista)
```

Figura 53 – Exemplos de inicialização de listas com **range()** e *
Fonte: Elaborado pelo Autor.

Outra maneira de inicializar listas é utilizando a compressão. A compressão consiste em escrever um código entre colchetes que gera uma lista de valores. Também podemos ler um texto e quebrá-lo em lista. Nesse caso, podemos quebrar o texto com a função

split() e usar a função **len()** para obter o tamanho da lista. A Figura 54 exemplifica a compressão de listas e a criação de listas a partir de texto.

```
# Lista inicializada com números de 0 a 9
lista = [n for n in range(9)]
print(lista)
# Leitura de string e com split
texto = input('Informe números (separados com espaços): ')
lista = [int(x) for x in texto.split()]
print(lista)
print(len(lista))
```

Figura 54 – Exemplo de compressão e lista a partir de texto
Fonte: Elaborado pelo Autor.

Dica do Professor: A função **len()** é uma função da biblioteca padrão da linguagem Python que retorna a quantidade de elementos de listas ou de outros tipos de dados como texto, por exemplo.

Além da seleção de um único elemento pelo seu índice, as listas permitem diversos outros tipos de seleções. A utilização de índices negativos faz a seleção dos elementos a partir do final da lista. O índice **-1** representa o último elemento, **-2** é o penúltimo elemento, e assim por diante. Também podemos selecionar sub-listas, ou seja, pedaços da lista. No caso das sub-listas usamos a notação de colchetes após a lista indicando a posição inicial e final da sub-lista. A Figura 55 mostra alguns exemplos de sub-listas a partir de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**.

Para verificarmos se um elemento existe dentro de uma lista temos que varrer todos os seus elementos. Uma maneira de fazer isso escrevendo de forma mais concisa é utilizar o operador **in**. Basicamente, a instrução **x in l**, retorna **True** se o elemento **x** existir na lista **l**. De forma análoga, podemos, a expressão **x not in l**, retorna **True**, se o elemento **x** não estiver na lista **l**.

Notação	Resultado	Explicação
l[:]	**[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**	Copia a lista
l[1:]	**[1, 2, 3, 4, 5, 6, 7, 8, 9]**	Todos os elementos a partir do segundo
l[:-1]	**[0, 1, 2, 3, 4, 5, 6, 7, 8]**	Todos os elementos até o penúltimo
l[3:7]	**[3, 4, 5, 6]**	Do quarto ao sétimo elemento

Figura 55 – Seleção de sub-listas de uma lista **l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**
Fonte: Elaborado pelo Autor.

As tuplas são estruturas usadas para agrupar uma quantidade fixa de elementos. A especificação de tuplas é feita escrevendo seus elementos separados por vírgula. Se a tupla tiver um único elemento, é preciso incluir uma vírgula após esse elemento. Normalmente, por uma questão de legibilidade, colocamos essa lista de elementos entre parênteses. As tuplas são recomendadas para agrupar quantidades fixas e pequenas de elementos.

Uma aplicação interessante para tuplas é o agrupamento de dia, mês e ano em uma data. A Figura 56 ilustra essa aplicação. Observe que as tuplas de data são criadas com ano, mês e dia, nessa ordem. Isso possibilita comparar as tuplas diretamente como foi feito no **if**. A comparação é feita considerando o primeiro elemento, depois o segundo e assim por diante.

```
print('Informe as datas')
print('Primeira data')
dia = int(input('Dia: '))
mes = int(input('Mês: '))
ano = int(input('Ano: '))
data1 = (ano, mes, dia)
print('Segunda data')
dia = int(input('Dia: '))
mes = int(input('Mês: '))
ano = int(input('Ano: '))
data2 = (ano, mes, dia)
recente = data1
if data2 > data1:
    recente = data2
print('Data mais recente:', recente)
```

Figura 56 – Utilização de tuplas para representar datas
Fonte: Elaborado pelo Autor.

Cada elemento de uma tupla possui um índice, começando pelo zero (DOWNEY, 2015). Assim, podemos usar a instrução **t[n]** para acessar o elemento da tupla **t** na posição **n-1**. Os índices dos elementos das tuplas representando datas podem vistos na Figura 57.

0	1	2
ano	mes	dia

Figura 57 – Índices dos elementos de uma tupla representando data
Fonte: Elaborado pelo Autor.

Outra peculiaridade das tuplas é a possibilidade de atribuir o valor dos elementos de uma tupla a um conjunto de variáveis em uma única instrução no formato **a, ..., x = t**. Tal instrução é valida desde que o número de variáveis seja igual ao número de elementos da tupla.

Outra aplicação interessante de tuplas é no problema dos preços acima da média visto anteriormente. Além do preço, seria interessante guardar o nome do produto para mostrar quando necessário. Podemos podemos fazer isso agrupando esses dados em uma tupla. A Figura 58 demonstra como podemos implementar o código. Dentro do laço **for**, pegamos o nome e o preço do produto, respectivamente. Em seguida, juntamos os dados em uma tupla e adicionamos essa tupla à lista de produtos. No segundo laço de repetição percorremos os produtos da lista e extraímos seu nome e preço. O **if** é usado para verificar se o preço do produto está acima da média.

```
soma = 0
lista_produtos = []
print('Informe o dados dos produtos')
for cont in range(10):
    print('\nProduto ' , cont+1)
    nome = input('Nome: ')
    preco = float(input('Preço: '))
    produto = (nome, preco)
    soma += preco
    lista_produtos.append(produto)
media = soma / 10
print('O preço médio é', media)
print('Os produtos com preço acima da média são:')
for produto in lista_produtos:
    nome, preco = produto
    if preco > media:
        print('Produto:', nome)
        print('Preço:', preco)
```

Figura 58 – Produtos acima da média usando tuplas
Fonte: Elaborado pelo Autor.

Os conjuntos são coleções não ordenadas de elementos (CORRÊA, 2020). Eles devem ser utilizados quando a existência de um elemento na coleção é mais importante do que a ordem do elemento ou do que a quantidade de vezes que o elemento aparece. Na verdade, os conjuntos em Python, assim como na matemática, não possuem elementos repetidos.

Podemos criar um conjunto com elementos separados por vírgula entre parênteses ou usar o construtor **set()** (TAGLIAFERRI, 2018). No caso do construtor, temos que passar uma coleção de elementos, como um alista, por exemplo. Conjuntos vazios devem ser criados com o construtor **set()** sem sem nenhum parâmetro. Para exemplificar as operações sobre conjuntos, vamos considerar os conjuntos **s1 = {1, 2, 3, 4}**, **s2 = {4, 5, 6}** e **s3 = {4, 5}**. A Figura 59 mostra algumas operações sobre esses conjuntos na linguagem Python e a notação matemática correspondente. Observe que as operações mostradas podem ser feitas com operadores ou funções.

Notação Matemática	Código Python		Resultado
	Operadores	Funções	
$s_1 \cap s_2$	s1 & s2	s1.intersect(s2)	{4}
$s_1 \cup s_2$	s1 | s2	s1.union(s3)	{1, 2, 3, 4, 5, 6}
$s_1 - s_2$	s1 - s2	s1.difference(s2)	{1, 2, 3}
$s_3 \subseteq s_2$	s3 <= s2	s3.issubset(s2)	True
$s_1 \supseteq s_3$	s1 >= s3	s1.issuperset(s3)	False

Figura 59 – Operações sobre conjuntos de dados
Fonte: Elaborado pelo Autor.

Além das operações entre conjuntos podemos verificar se um elemento existe no conjunto como operador **in**. Também podemos adicionar e remover elementos com as operações **add()** e **remove()**, respectivamente.

Um dicionário é um tipo especial de coleção que faz mapeamento ou associação de chave-valor (CORRÊA, 2020). Isso significa que, para cada chave do dicionário, existe um valor associado. A chave deve ser um tipo de dado imutável, mas o valor pode ser qualquer tipo de dado. Os tipos imutáveis são os tipos de dados básicos e outros dados que não podem sofre modificações como, por exemplo, tuplas compostas de dados imutáveis.

A maneira mais comum de criar dicionários com lista de chaves e valores (chave:valor) entre chaves. Um dicionário vazio pode ser criado com **{}** (abre e fecha chaves). Considerando um dicionário **d** e uma chave **c**, podemos consultar o valor para a chave **c** com o comando **d[c]**. Já a adição ou alteração do valor pode ser feita com a instrução **d[c] = v**, onde **v** o valor a ser atribuído à chave. A Figura 60 mostra um exemplo simples de código que cria e manipula dicionários.

```
dic_vazio = {}
dic_letras = {1:'A', 2:'B', 3:'C'}
dic_letras[4] = 'E'
dic_letras[4] = 'D'
print(dic_vazio, dic_letras)
for cont in range(1, 5):
    print(cont, ':', dic_letras[cont])
```

Figura 60 – Exemplo simples com dicionários
Fonte: Elaborado pelo Autor.

Além das manipulações usando chaves e colchetes, os dicionários possuem diversas funções que facilitam seu tratamento. A Figura 61 exibe algumas dessas funções e e seu funcionamento.

Função	Funcionamento
d.clear()	Remove todos os elementos do dicionário **d**
d.copy()	Retorna uma cópia de **d**
d.items()	Retorna as chaves-valores de **d** no formato de tuplas
d.keys()	Retorna uma lista com as chaves de **d**
d.popitem()	Retorna uma tupla (chave, valor) qualquer(se **d** estiver vazio, ocorre um erro)
d.update(d2)	Atualiza os elementos de **d** utilizando os elementos de **d2**
d.values()	Retorna uma lista com os valores de **d**

Figura 61 – Algumas funções de dicionários
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

1.3 Exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários. Em seguida, o código deve recalcular os salários considerando os seguintes aumentos:

- 20% para salários de até R$2.000,00;
- 15% para salários entre R$2.000,00 e R$5.000,00;
- 5% para salários maiores que R$5.000,00;

Por fim, o código deve exibir o total de aumento (total dos salários novos menos o total de salários antigos e os nomes dos funcionários com salários menores que R$2.000,00.

B) Construir um simulador de urna eletrônica. Inicialmente, o simulador deve permitir o cadastro de candidatos. O usuário pode cadastrar quantos candidatos desejar. O cadastro de um candidato envolve um número e seu nome. O número deve ser armazenado em formato textual, mas deve possuir exatamente dois dígitos numéricos. A função **isdigit()** do tipo textual pode ser usada para verificar se o texto é um número válido. Por fim, os candidatos cadastrados devem ser mantidos em um dicionário (número: nome).

Após o cadastro de candidatos, o simulador deve iniciar a votação. O simulador deve permitir uma quantidade indeterminada de votos. Para votar, o usuário deve informar o número do candidato. O sistema deve mostrar o nome do candidato para o usuário confirmar. Números inválidos devem ser computados como votos nulos. Já o texto vazio deve ser contabilizado como voto em branco. A sumarização dos votos deve ser feita usando um dicionário. Ao término da votação, o simulador deve mostrar o total e a porcentagem de votos de cada candidato, nulos e brancos.

1.4 Respostas dos exercícios

Escreva os códigos em Python modularizados e com tratamento de exceções para resolver os problemas a seguir:

A) Desenvolver um algoritmo que construa uma lista de tuplas com nome e salário de funcionários.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

def le_salario():
    while True:
        try:
            return float(input('Salário: '))
        except:
            print('Valor inválido! Informe novamente')

def le_funcionario():
    nome = input('Nome: ')
    salario = le_salario()
    return (nome, salario)

def aumenta_salarios(lista_funcionarios):
    total = 0
    for cont, funcionario in enumerate(lista_funcionarios):
        nome, salario = funcionario
        if salario <= 2000:
            salario *= 1.2
        elif salario <= 5000:
            salario *=1.15
        else:
            salario *=1.05
        total += salario
        lista_funcionarios[cont] = (nome, salario)
    return total

def principal():
    lista_funcionarios = []
    total_antigo = 0
    while True:
        funcionario = le_funcionario()
        lista_funcionarios.append(funcionario)
        total_antigo += funcionario[1]
        resp = input('Continuar (S/N): ').lower().strip()
        if resp != 's':
            break
    total_novo = aumenta_salarios(lista_funcionarios)
    total_aumento = total_novo - total_antigo
    print('Total de aumento:', total_aumento)
    print('Funcionários com salário abaixo de R$2.000,00:')
    for nome, salario in lista_funcionarios:
        if salario < 2000:
```

```
46              print(nome)
47
48  if __name__ == '__main__':
49      principal()
```

B) Construir um simulador de urna eletrônica.

```
 1  #!/usr/bin/python3
 2  # -*- coding: utf-8 -*-
 3
 4  def lista_cand(dict_cand):
 5      print('\n'*100)
 6      print('Candidatos:')
 7      for numero, nome in dict_cand.items():
 8          print(numero, '-', nome)
 9
10  def adiciona_cand(dict_cand):
11      lista_cand(dict_cand)
12      print('\nInforme os dados do novo candidato')
13      num = input('Número: ')
14      nome = input('Nome: ')
15      if len(num) != 2 or not num.isdigit() or num in dict_cand:
16          print('Cadastro inválido!')
17      else:
18        dict_cand[num] = nome
19
20  def pega_voto(dict_cand, dict_votos):
21      while True:
22          lista_cand(dict_cand)
23          print('\nInforme seu voto')
24          voto = input('Número do candidato: ').strip()
25          if voto == '':
26              voto = 'Branco'
27          elif voto in dict_cand:
28              voto = voto + ' - ' + dict_cand[voto]
29          else:
30              voto = 'Nulo'
31          print('Voto:', voto)
32          resp = input('Confirma (S/N): ').lower().strip()
33          if resp == 's':
34              soma_voto(dict_votos, voto)
35              break
36
37  def soma_voto(dict_voto, voto):
38      if voto in dict_voto:
39          dict_voto[voto] += 1
40      else:
41          dict_voto[voto] = 1
42
43  def principal():
44      dict_cand = {}
45      while True:
46          adiciona_cand(dict_cand)
47          resp = input('Continuar cadastros (S/N): ').lower().strip()
48          if resp == 'n':
```

```
49                 break
50         dict_voto = {}
51         total = 0
52         while True:
53             pega_voto(dict_cand, dict_voto)
54             total += 1
55             resp = input('Encerrar votação (S/N): ').lower().strip()
56             if resp == 's':
57                 break
58         print('Resultado da eleição:')
59         for voto in dict_voto:
60             porcent = round(dict_voto[voto] / total * 100, 2)
61             print(voto, ': ', dict_voto[voto],
62                   ' (', porcent, '%', ')', sep='')
63
64     if __name__ == '__main__':
65         principal()
```

1.5 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista aos vídeos “Revisão da Primeira Semana - Parte 1” e “Revisão da Primeira Semana - Parte 2”.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de preparação de dados;
- Desenvolver e entender códigos para preparação de dados.

2.1 Introdução

Uma mineração de dados bem sucedida deve levar em consideração o tipo e a qualidade dos dados. O tipo de dado diz respeito ao formato dos dados e isto pode definir quais técnicas de mineração devem ser usadas. A qualidade é importante porque, embora muitas das técnicas de mineração de dados tolerem certo nível de ruídos, a melhora na qualidade dos dados pode levar a resultados mais promissores.

Em determinadas situações, é preciso modificar os dados brutos para que se tornem mais apropriados para certas técnicas de mineração. Uma preparação de dados bem sucedida envolve o bom conhecimento da base de dados a ser minerada, seus dados, suas características e das possíveis técnicas de preparação de dados que podem ser aplicadas (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018).

2.2 Tipos de dados

Um conjunto de dados pode ser visto como uma coleção de objetos de dados. Os objetos de dados também podem ser chamados de registros, linhas, ponteiros, vetores, eventos, casos, exemplos, observações ou entidades. Os objetos de dados são descritos por um número de atributos. Outros nomes para atributos são variáveis, características, campos ou dimensões. Como exemplo, qualquer tabela ou resultado de uma consulta em um banco de dados é um conjunto de dados (ELMASRI; NAVATHE, 2011).

2.2.1 Atributos

Um atributo é uma característica de um objeto que pode variar com o tempo ou de um objeto para outro, por exemplo, a cor dos olhos ou peso de pessoas. Os valores dos atributos estão relacionados com as seguintes propriedades:

- Igualdade ou distinção (operadores de = e ≠);
- Ordenação (operadores de <, ≤, > e ≥);
- Adição (operadores de + e −).

Com base nessas propriedades os atributos podem ser classificados em tipos conforme mostrado na Figura 62. Os atributos nominais e ordinais são chamados de categóricos ou qualitativos. Os atributos numéricos, por sua vez, também são chamados

de quantitativos. Os atributos também podem ser discretos (inteiros) ou contínuos (valores com casas decimais).

Tipo	Descrição	Operações suportadas	Exemplo
Nominais	Valores nominais	= e ≠	Cidade, Profissão
Ordinais	É possível ordenar os valores	=, ≠, <, ≤, > e ≥	Conceitos, grau de escolaridade
Numéricos	É possível calcular diferenças entre os valores	=, ≠, <, ≤, >, ≥, + e -	Data, temperatura, preço

Figura 62 – Tipos de dados
Fonte: Elaborado pelo Autor.

2.2.2 Conjuntos de dados

Além do conhecimento sobre os tipos de atributos, é importante sabermos com que espécie de conjunto de dados teremos que lidar. As principais características de conjuntos de dados são a dimensão e a dispersão. A dimensão é o número de atributos do conjunto de dados. A dispersão está relacionada com a quantidade de registros com valores diferentes de zero. Quanto menos valores diferentes de zero o conjunto possui, mais disperso ele é.

Em geral, os conjuntos de dados podem ser classificados em três grandes grupos. São eles:

- Dados em registros;
- Dados baseados em grafos;
- Dados ordenados.

Os dados em registros é o grupo mais comum de conjunto de dados. Esse formato consiste em uma coleção de registros, onde cada registro possui um número fixo de atributos. Os dados em registros também podem ser vistos como uma tabela de linhas e colunas.

Quando todos os atributos de um conjunto de dados são numéricos, esse conjunto pode ser interpretado como uma matriz **m x n** (**m** linhas por **n** colunas). As matrizes de dados são interessantes porque podemos aplicar operações de matrizes para transformar os dados. A Figura 63 apresenta um exemplo de matriz de dados.

X	Y	Distância	Peso
10,23	5,27	15,22	1,2
12,65	6,25	16,22	1,1
13,54	7,23	17,34	1,2
14,27	8,43	18,45	0,9

Figura 63 – Exemplo de matriz de dados
Fonte: Elaborado pelo Autor.

A matriz de dados dispersos é um caso especial de matriz de dados onde os atributos são do mesmo tipo e dispersos. Um exemplo interessante de aplicação é a matriz de termos de documentos contendo o número de vezes que o termo aparece no documento. A Figura 64 mostra um exemplo desse tipo de matriz.

	equipe	treinador	jogo	bola	placar	partida	vitória	derrota
Documento 1	3	0	5	2	1	0	3	4
Documento 2	1	3	4	3	0	2	1	2
Documento 3	2	0	0	2	1	1	2	0

Figura 64 – Exemplo de matriz de dados dispersos
Fonte: Elaborado pelo Autor.

Os grafos são representações utilizadas em diversas aplicações. Os conjuntos de dados baseados em grafos pode ser usados para representar relacionamentos entre objetos ou os objetos propriamente ditos. Um exemplo de grafo que representa relacionamento entre objetos é uma base de dados com páginas da Internet e os hyperlinks entre essas páginas. Exemplos de objetos na forma de grafos são estruturas químicas de substâncias.

Nos conjuntos de dados ordenados, os atributos têm relacionamentos que envolvem ordenação no tempo ou espaço. Alguns exemplos são sequências de DNA, séries temporais de ações da bolsa de valores e sequências de posições capturadas por aparelhas de GPS.

2.3 Bibliotecas

Existem diversas bibliotecas úteis para se trabalhar com o pré-processamento de dados. Nesta seção falaremos brevemente das bibliotecas **numpy** e **pandas**.

2.3.1 A biblioteca numpy

Muitos atributos de bases de dados são dados numéricos, como preços, números de vendas, medições de sensores, placares esportivos e assim por diante. A biblioteca **numpy** é especialmente útil para trabalhar com vetores e matrizes de dados numéricos. A

instalação da biblioteca pode ser feita como comando sudo **apt install python3-numpy**, no Linux, ou com o comando **pip install numpy**.

O principal tipo de dado da biblioteca **numpy** é o vetor de dados **ndarray** ou simplesmente **array**. Os vetores permitem representar vetores numéricos de uma ou mais dimensões. Diferentemente das listas padrões da linguagem Python, nos vetores, todos os elementos precisam ser do mesmo tipo. A criação de vetores pode ser feita com a função **array()** da biblioteca **numpy** aplicada sobre uma lista de valores. A Figura 65 mostra um exemplo de código que trabalha com vetores.

```
import numpy as np

a = np.array([1, 2, 3, 4])
print('\nVetor de inteiros\n', a)
print(a.dtype)
a = np.array([1, 2, 3, 4], dtype='float64')
print('\nVetor de flutuantes\n', a)
print(a.dtype)

a.shape = (2, 2)
print('formato:', a.shape)
print('\nVetor bidimensional', a)

print('\nMudando o formato do vetor')
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
a = a.reshape(3, -1)
print('Dimensões:', a.ndim)
print(a)
```

Figura 65 – Exemplo de criação e mudança de formato de vetores
Fonte: Elaborado pelo Autor.

A biblioteca **numpy** possui também as funções **arange()** e **linspace()** para a criação de vetores dentro de um intervalo especificado. Além disso, o atributo **flat** de um vetor retorna um iterador que pode ser usado para percorrer todos os elementos do vetor. A Figura 66 mostra um exemplo de código usando esses recursos.

```
import numpy as np

print('Vetor com arange()')
a = np.arange(0, 2, 0.2)
print(a)

print('\nVetor com linspace()')
a = np.linspace(0, 1, 8)
for n in a.flat:
    print(n)
```

Figura 66 – Exemplo de criação de vetores com intervalos e iteração sobre os elementos
Fonte: Elaborado pelo Autor.

O acesso aos elementos de vetores é feito de forma parecida com o acesso a elementos de lista. Entretanto, no caso dos vetores, devemos informar uma posição ou faixa de posições para cada dimensão do vetor. Lembrando que a indexação dos

elementos sempre começam na posição 0 (zero). A Figura 67 mostra um exemplo de código que demonstra o acesso a posições de um vetor.

```
import numpy as np

print('Matriz 5x5')
a = np.array(range(25)).reshape(5, 5)
print(a)
print('Elemento la linha 2 coluna 3:', a[2, 3])
print('Linhas 2 a 3:\n', a[2:4])
print('Colunas 1 a 2:\n', a[:, 1:3])
print('Recorte (1,3) a (3,5):\n', a[1:3, 3:5])
```

Figura 67 – Exemplo de acesso a posições de vetor
Fonte: Elaborado pelo Autor.

Podemos fazer diversas operações aritméticas entre vetores e entre vetores e valores numéricos. Os vetores também possuem métodos para obtenção de valores mínimos e máximos, média e outras informações. A Figura 68 apresenta um código que realiza algumas dessas operações.

```
import numpy as np

print('Operações com vetores')
a = np.array([3, 4, 8, 2])
b = np.array([5, 7, 4, 1])
print('a = ', a)
print('b = ', b)
print('a + b = ', a + b)
print('a * b = ', a * b)
print('a * 3 = ', a * 3)

print('Mínimo, máximo, média')
a = np.arange(9).reshape(3, 3)
print(a.min(), a.max(), a.mean())

print('Menor linha, menor coluna, média da coluna')
print(a.min(axis=0), a.min(axis=1), a[:,2].min())
```

Figura 68 – Exemplo de operações com vetores
Fonte: Elaborado pelo Autor.

A biblioteca **numpy** conta também com diversas funções para trabalhar com números aleatórios no módulo **numpy.random**. Além, disso é possível lidar com diversas operações com matrizes como multiplicação e transposição. A Figura 69 mostra um exemplo com geração de matrizes com números aleatórios, multiplicação e transposição de matrizes. Em nosso, exemplo usamos a função **rand()** do módulo **random** para gerar duas matrizes de números aleatórios. Depois, multiplicamos tais matrizes com a função **dot()** e, por último, mostramos a matriz transposta do resultado dessa multiplicação.

```
import numpy as np

m1 = np.random.rand(3, 2)
print('Primeira matriz aleatória:\n', m1)
m2 = np.random.rand(2, 4)
print('\nSegunda matriz aleatória:\n', m2)
m3 = np.dot(m1, m2)
print(m3)
m3 = m3.transpose()
print('\nMultiplicação transposta:\n', m3)
```

Figura 69 – Exemplo de geração de matrizes aleatórias, multiplicação e transposição de matrizes
Fonte: Elaborado pelo Autor.

2.3.2 A biblioteca pandas

A biblioteca pandas é um dos principais recursos usados para fazer pré-processamento em bases de dados. Sua instalação, no Linux, pode ser feita com o comando sudo **apt install python3-pandas**. No Windows, podemos usar o comando **pip install pandas**. Os dois tipos de dados básicas da biblioteca são o **Series** e o **DataFrame**. O **Series** é um vetor unidimensional de valores que conta com o atributo **index** para identificar a posição de cada valor.

```
import pandas as pd

precos = pd.Series([10.5, 5.8, 9, 15.3, 21, 11.4, 4, 7])
print('Series com index automático:\n', precos)

precos = pd.Series([10.5, 5.8, 9, 15.3, 21, 11.4, 4, 7],
                   index=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'])
print('\nSeries com index rotulado:\n', precos)

print('\nPreço do produto B:', precos['B'])
print('\nPreço médio:', precos.mean())

print('\nPreços com 20% de aumento:\n', precos * 1.2)
```

Figura 70 – Exemplo com Series
Fonte: Elaborado pelo Autor.

A Figura 70 mostra um exemplo usado vetores de **Series** para guardar uma lista de preços. O primeiro vetor foi criado com **index** automático, assim cada preço fica associado a uma posição, começando com 0 (zero). No segundo, vetor o **index** é rotulado com os nomes dos produtos. Podemos usar os valores do **index** para acessar o preço do produto na posição desejada.

O objeto **Series**, assim como os vetores da biblioteca **numpy**, conta com diversas funções aplicáveis a lista de valores guardados como média, somatório, contagem e outros. Em nosso exemplo, usamos o método **mean()** para calcular o preço médio. Podemos também fazer operações aritméticas envolvendo todos os valores da lista, como o aumento em 20% nos preços mostrado no exemplo.

O **DataFrame** é uma estrutura bidimensional de dados semelhante a uma matriz. Cada coluna de um **DataFrame** é uma **Series**. A Figura 71 mostra um exemplo de código

que trabalha com **DataFrame**. Criamos o **DataFrame dados** a partir de um dicionário representando as colunas da matriz. Usamos as propriedades **dtypes** e **columns** para mostrar os tipos e nomes de cada coluna, respectivamente. Podemos ter acesso a **Series** que representa uma coluna, colocando o nome dessa coluna como uma chave de dicionário para o **DataFrame**. No final, usamos o método **to_csv()** para salvar o DataFrame em arquivo. O parâmetro **index=False** é usado porque não desejamos salvar o índice, apenas os dados.

```
import pandas as pd

dados = pd.DataFrame(
        {'Produto': ['A', 'B', 'C', 'D', 'E'],
         'Preço': [10.5, 5.8, 9, 15.3, 21],
         'Estoque': [120, 100, 50, 150, 204]})
print('Produtos:\n', dados)
print('\nTipos de dados:\n', dados.dtypes)
print('\nColunas:', dados.columns)
print('\nEstoque:\n', dados['Estoque'])
print('\nSalvando dados em arquivo')
dados.to_csv('produtos.csv', index=False)
```

Figura 71 – Exemplo de criação de **DataFrame** a partir de dicionário
Fonte: Elaborado pelo Autor.

A biblioteca pandas possui o método **read_csv()** que permite a importação de dados de arquivos para um **DataFrame**. A Figura 72 mostra um exemplo de uso dessa função além de operações de ordenação e filtragem. A ordenação é feita como método **sort_values()**. Em nosso, exemplo usamos o parâmetro **ascending=True** para usar a ordem decrescente. Por padrão, todas as operações em **DataFrames** retornam um **DataFrame** como resposta, o parâmetro **inplace=True** é usado para que a operação de ordenação altere o **DataFrame** atual. Esse parâmetro está disponível em muitos outros métodos de manipulação do **DataFrame**.

```
import pandas as pd

dados = pd.read_csv('produtos.csv')
print('\nDados lidos de arquivo:\n', dados)
dados.sort_values(by='Estoque', ascending=False, inplace=True)
print('\nProdutos ordenados pelo estoque:\n', dados)
dados.reset_index(inplace=True)
dados.drop(columns='index', inplace=True)
print('\nProdutos ordenados com índice atualizado:\n', dados)
print('\nTerceiro produto:\n', dados.loc[2])
print('\nPosições com estoque maior que 100:\n', dados['Estoque'] > 100)
print('\nProdutos com estoque alto:\n',
      dados[dados['Estoque'] > 100])
print('\nProdutos com estoque e preço altos:\n',
      dados[(dados['Estoque'] > 100) & (dados['Preço'] > 20)])
```

Figura 72 – Importação de dados de arquivos, ordenação e filtragem
Fonte: Elaborado pelo Autor.

Quando reordenamos os dados, os valores dos índices modificam conforme a posição que a linha fica na ordenação. Podemos usar o método **reset_index()** para

reiniciar os índices considerando as novas posições da ordenação. O detalhe é que esse método cria a coluna **index** com o valor do índice antigo. Assim, usamos o método **drop()** para apagar essa coluna que foi criada.

A propriedade **loc** pode ser usada para acessar uma determinada linha pelo seu índice. Quando fazemos uma comparação do tipo **dados['Estoque'] > 100**, recebemos uma lista de valores booleanos com o resultado da comparação em cada linha da coluna. Podemos usar o resultado dessa operação para filtrar os dados, ou seja, somente as linhas onde a comparação resultou em **True** serão retornadas. Também podemos combinar diversas comparações durante a filtragem.

Tanto os **DataFrames** quanto as **Series** também possuem recursos para plotagem de gráficos. O gráfico padrão pode ser plotado com o método **plot()**. Além disso, o atributo **plot** possui diversos outros tipos de gráficos. A Figura 73 mostra um exemplo de código que faz três tipos de plotagem de gráficos. Nesse exemplos, utilizamos a biblioteca **seaborn** que pode ser instalada como comando **pip install seaborn**. A função **load_dataset()** da biblioteca é usada para retornar um **DataFrame** com a base de dados. Em nosso exemplo, utilizamos a base de dados **tips** contendo informações sobre rotas de taxistas. A Figura Figura 74 exibe os três gráficos gerados pelo código.

```
import matplotlib.pyplot as plt
from seaborn import load_dataset

dados = load_dataset('tips')
dados['total_bill'].plot()
plt.show()
plt.clf()
dados['total_bill'].plot.hist()
plt.show()
plt.clf()
dados.plot.scatter(x='total_bill', y='tip')
```

Figura 73 – Código para plotagem com **DataFrames** e **Series**
Fonte: Elaborado pelo Autor.

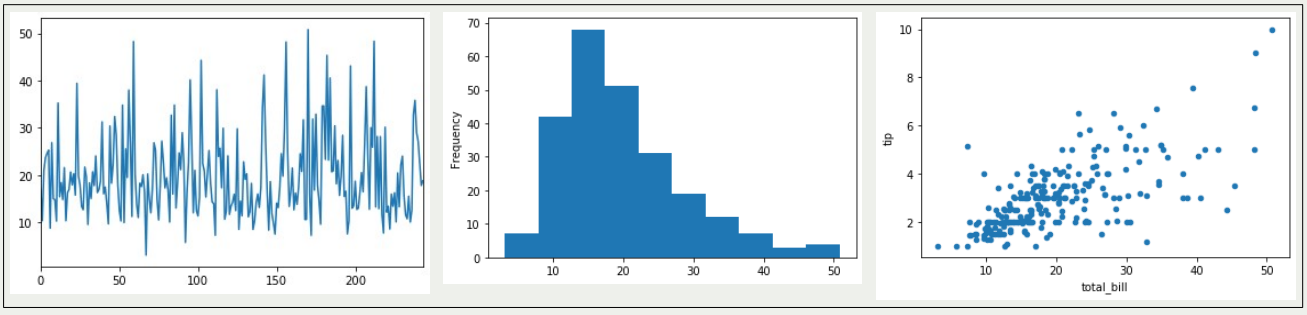


Figura 74 – Gráficos gerados pelo código para plotagem com **DataFrames** e **Series**
Fonte: Elaborado pelo Autor.

2.4 Pré-processamento de dados

O pré-processamento dos dados consiste na aplicação de alguma transformação sobre os mesmos com o objetivo de melhorar o resultado da mineração de dados. As

principais técnicas de pré-processamento são agregação, amostragem, redução de dimensionalidade, criação de recursos, normalização e discretização.

2.4.1 Agregação

A agregação consiste em combinar vários registros em um único registro. Os conjuntos de dados agregados são menores e, com isto, consumem menos memória e menos tempo de processamento. A agregação serve também para se ter uma visão de mais alto nível dos dados. Por outro lado, uma desvantagem da agregação é a potencial perda de detalhes.

A Figura 75 mostra um código para fazer agregação sobre a base de dados **flights** disponível na biblioteca **seaborn**. Essa base de dados contém o número de voos mensais nos EUA de 1949 a 1960. A agregação é feita usando o método **groupby** do **DataFrame**.

```
import seaborn as sb

voos = sb.load_dataset('flights')
print(voos.head())
voos_anual = voos.groupby('year').sum()
print(voos_anual.head())
```

Figura 75 – Exemplo de agregação de dados
Fonte: Elaborado pelo Autor.

Em nosso exemplo, fazemos uma agregação utilizando o método **sum()** para somar os voos por ano. A Figura 76 mostra o resultado da execução do código. Além da soma, é possível fazer a agregação com média, moda e outras funções.

```
(144, 3)
   year month  passengers
0  1949   Jan         112
1  1949   Feb         118
(12, 2)
   year  passengers
0  1949        1520
1  1950        1676
```

Figura 76 – Resultado do exemplo de agregação de dados
Fonte: Elaborado pelo Autor.

2.4.2 Amostragem

O tipo mais básico de amostragem é a *amostragem aleatória simples* onde há uma probabilidade igual de selecionar qualquer item. Durante a seleção dos elementos, podemos manter ou remover o elemento selecionado da base de dados. Quando removemos o elemento, estamos fazendo a *amostragem sem substituição*.

Quando quando mantemos o elemento selecionado na base de dados original, estamos realizando a *amostragem com substituição*. Dessa forma, podem ocorrer elementos repetidos na amostra. A amostragem com substituição é indicada para bases de dados com poucos elementos.

A amostragem aleatória simples pode não ser adequada quando a população possui poucos objetos de um determinado tipo, levando até à inexistência desse tipo na amostra. Uma solução para este problema é utilizar a *amostragem estratificada* que traz objetos de todos os grupos da população de forma proporcional.

```
import pandas as pd
from seaborn import load_dataset


dados = load_dataset('penguins')
# Amostragem sem substituição
print('Amostragem sem substituição')
amostra = dados.sample(frac=0.1)
print(amostra.index.value_counts().head())

# Amostragem com substituição
print('Amostragem com substituição')
amostra = dados.sample(n=50, replace=True)
print(amostra.index.value_counts().head())

# Amostragem estratificada
print('Amostragem estratificada')
print('Contagem da base de dados original:')
contagem = dados['species'].value_counts()
print(contagem)
amostra = pd.DataFrame()
for n in range(len(contagem)):
    especie = contagem.index[n]
    quantidade = int(contagem[n] * 0.1)
    amostra_especie = dados[dados['species'] == especie
                            ].sample(n=quantidade)
    amostra = amostra.append(amostra_especie)
print('Contagem da amostra')
print(amostra['species'].value_counts())
```

Figura 77 – Exemplo de amostragem de dados
Fonte: Elaborado pelo Autor.

A Figura 77 mostra um exemplo de código para realização de amostragem na base de dados *penguins* disponível na biblioteca **seaborn**. As amostragens são realizadas com o método **sample()**. Na primeira delas, informamos o parâmetro **frac=0.1** para obtermos uma amostra com 10% de registros da base original. Após a realização da amostragem fazemos uma contagem de valores como método **value_counts()** no atributo **amostra.index**. Aplicamos o **head()** nessa contagem de valores para mostrar apenas 5 primeiros resultados.

Na segunda amostragem, além do parâmetro **n=50** para definir o tamanho da amostra, usamos o parâmetro **replace=True** para realizar a amostragem com substituição. Na Figura 78, podemos ver o resultado da execução do código. Observe que, na segunda amostragem, alguns valores do **index** aparecem com contagem 2 indicando que houve repetição.

Por último, demostramos como pode ser feita uma amostragem estratificada. A coluna **species** indica a espécie de cada pinguim da base de dados. Assim, começamos

fazendo uma contagem sobre os valores dessa coluna. Depois, para cada valor contado, filtramos a base de dados pela espécie e fazemos uma amostra com 10% do valor contado. Nós juntamos as amostragens feita com aca espécie usando o método **append()** do **DataFrame** amostras para chegar na amostra com todas as espécies.

```
Amostragem sem substituição
127    1
79     1
88     1
215    1
86     1
dtype: int64
Amostragem com substituição
3      2
153    2
167    2
63     1
80     1
dtype: int64
Amostragem estratificada
Contagem da base de dados original:
Adelie       152
Gentoo       124
Chinstrap     68
Name: species, dtype: int64
Contagem da amostra
Adelie       15
Gentoo       12
Chinstrap     6
Name: species, dtype: int64
```

Figura 78 – Resultado do exemplo de amostragem de dados
Fonte: Elaborado pelo Autor.

Em geral, a amostragem é usada quando precisamos de uma base de dados menor do que a original. Um dos problemas relacionados a amostragem é definir o tamanho da amostra. Uma amostra muito pequena pode não ser representativa e uma amostra muito grande perde o objetivo da amostragem (redução do tamanho da base de dados original). O tamanho da amostra deve ser representativo e pequeno.

2.4.3 Redução de dimensionalidade

Conjuntos de dados com um grande número de dimensões podem causar problemas nas técnicas de mineração de dados. Dentre os problemas causados, podem ocorrer a geração de modelos de baixa precisão, alto consumo de memória e tempo para execução da mineração. A ocorrência desses problemas causados pelo grande número de atributos é chamada de *maldição da dimensionalidade*. Desta maneira, foram criados métodos para reduzir a dimensionalidade de conjuntos de dados sem que os elementos percam sua representatividade.

A redução da dimensionalidade pode ser feita com a transformação do conjunto de atributos em um conjunto menor (projeção de características) ou com a seleção de um subconjunto menor de atributos (seleção de características).

2.4.3.1 Projeção de características

Uma das técnicas mais usadas para projeção de características é a Análise de Componentes Principais ou *Principal Component Analysis (PCA)*. O PCA utiliza transformações lineares para construir um conjunto menor de atributos a partir dos atributos originais. O inconveniente é que os atributos gerados não têm um significado como nos atributos reais. Por exemplo, peso, altura têm significado, mas não é possível dizer o que significa um valor qualquer de um atributo PCA.

```
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.decomposition import PCA

# Criação do conjunto de dados
X, Y = make_classification(n_samples=1000, n_informative=10,
                           n_redundant=10, random_state=42)
# Classificador Árvore de decisão
arvore = DecisionTreeClassifier(random_state=42, max_depth=7)
# Classificação dos dados originais
arvore.fit(X, Y)
acuracia = arvore.score(X, Y)
print('Número de atributos originais:', X.shape[1])
print('Acurárcia (dados originais):', acuracia)
# Aplicação do PCA
pca = PCA(n_components=10)
X = pca.fit_transform(X, Y)
# Classificação dos dados PCA
arvore.fit(X, Y)
acuracia = arvore.score(X, Y)
print('\nNúmero de atributos PCA:', X.shape[1])
print('Acurárcia (dados PCA):', acuracia)
```

Figura 79 – Exemplo de aplicação de PCA sobre uma base de dados
Fonte: Elaborado pelo Autor.

A Figura 79 demostra a aplicação do PCA sobre uma base de dados. Em nosso, exemplo, utilizamos a biblioteca **sklearn** que pode ser instalada com o comando **sudo apt python3-sklearn**, no Linux, ou com o comando **pip intall sklearn**, no Windows. A biblioteca **sklean** possuir implementações de diversos algoritmos para mineração de dados. Chamamos a função **make_classification()** do módulo **datasets** para criar uma base de dados fictícia com 1.000 registros e 20 atributos. Dos 20 atributos, 10 são significantes e 10 são redundantes. O parâmetro **random_state=42** é usado para que sempre sejam gerados os mesmos dados e possamos reproduzir o mesmo experimento.

Após a criação da base de dados, usamos um algoritmo de árvore de decisão para classificar os dados (veremos com mais detalhes os algoritmos de classificação nos próximos capítulos). O método **fit()** faz o treinamento com os dados e o método **score()** retorna a acurácia da classificação do algoritmo. Basicamente, a acurácia é a porcentagem de classificações corretas do algoritmo. Depois da classificação sobre os dados originais, aplicamos a técnica de PCA na base de dados e fazemos a classificação novamente. A aplicação da técnica é realizada por meio do método **fit_transform()** do objeto da classe **PCA**.

A Figura 80 mostra o resultado da execução do código. Podemos observar que a acurácia antes do PCA era de 94.1% (0.941) e depois do PCA passou para 95.2% (0.925). Isso mostra que os atributos redundantes estavam prejudicando a eficácia do algoritmo de classificação. Com a aplicação do PCA, ocorre a redução dessa interferência e, consequentemente, o algoritmo tem maior taxa de acertos.

```
Número de atributos originais: 20
Acurárcia (dados originais): 0.941

Número de atributos PCA: 10
Acurárcia (dados PCA): 0.952
```

Figura 80 – Resultado do exemplo de aplicação de PCA
Fonte: Elaborado pelo Autor.

2.4.3.2 Seleção de características

Dependendo da base de dados e do nível de conhecimento disponível, é possível descartar atributos que sejam redundantes ou irrelevantes. Os atributos redundantes possuem informações que já estão disponíveis em outros atributos como, por exemplo, o valor total de um produto que é calculado plea multiplicação da quantidade pelo valor unitário. Os atributos irrelevantes são aqueles que não possuem informações úteis como CPF ou código de clientes.

Alguns atributos podem ser identificados como redundantes ou irrelevantes de forma trivial. Contudo, podem existir outros atributos redundantes que não podem ser detectados tão facilmente. Assim, é interessante utilizar técnicas de seleção de características mais elaboradas para chegarmos a um subconjunto menor e sem grande perda de representatividade. O ideal seria testar todas as combinações possíveis de atributos, mas, na prática, isso pode não ser viável devido ao grande número de combinações possíveis.

As técnicas de seleção de características podem ser classificadas como abordagem interna, abordagem de filtro e abordagem de envoltório. Na abordagem interna, a seleção de atributos ocorre naturalmente como parte do algoritmo de mineração. A abordagem de filtro utiliza alguma técnica independente da mineração para selecionar os melhores atributos. A Abordagem de envoltório testa um algoritmo de mineração em diferentes combinações de atributos para selecionar aquela na qual o algoritmo teve melhor resultado. Como ainda não estudamos algoritmos de mineração de dados, vamos mostrar um exemplo de abordagem de filtro.

Um dos algoritmos mais utilizados na abordagem de filtro é a seleção dos k atributos com maior pontuação (top-k atributos). Em geral, essa pontuação pode ser calculada analisando informações estatísticas da base de dados como a correlação. A Figura 81 mostra um exemplo de seleção de atributos utilizando a técnica do top-k atributos. O resultado da execução do código pode ser visto na Figura 82. Podemos observar que a acurácia era de 94.1% passou para 95% após a seleção dos atributos.

```
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_selection import SelectKBest

# Criação do conjunto de dados
X, Y = make_classification(n_samples=1000, n_informative=10,
                           n_redundant=10, random_state=42)


# Classificador Árvore de decisão
arvore = DecisionTreeClassifier(random_state=42, max_depth=7)
# Classificação dos dados originais
arvore.fit(X, Y)
acuracia = arvore.score(X, Y)
print('Número de atributos originais:', X.shape[1])
print('Acurárcia (dados originais):', acuracia)
# Aplicação da seleção de atributos
selecionador = SelectKBest(k=10)
X = selecionador.fit_transform(X, Y)
# Classificação dos dados após seleção
arvore.fit(X, Y)
acuracia = arvore.score(X, Y)
print('\nNúmero de atributos selecionados:', X.shape[1])
print('Acurárcia (após seleção):', acuracia)
```

Figura 81 – Exemplo de aplicação de seleção de atributos
Fonte: Elaborado pelo Autor.

```
Número de atributos originais: 20
Acurárcia (dados originais): 0.941

Número de atributos selecionados: 10
Acurárcia (após seleção): 0.95
```

Figura 82 – Resultado da seleção de atributos
Fonte: Elaborado pelo Autor.

2.4.4 Criação de recursos

A criação de recursos visa criar novos atributos a partir dos atributos existentes. A ideia é que o número de atributos seja menor para termos as mesmas vantagens da redução de dimensionalidade. As principais metodologias empregadas são a extração de características, o mapeamento de dados para um novo espaço dimensional e a construção de características.

A extração de características tem como objetivo criar um novo conjunto de atributos para representar os atributos originais dos objetos. Esta metodologia é amplamente utilizada quando precisamos trabalhar com imagens. Dessa maneira, uma imagem contendo milhares ou milhões de pixeis pode ser representa de outras formas como, por exemplo, um histograma de cores.

O mapeamento para novo espaço é muito utilizado em padrões periódicos que podem conter quantidades significativa de ruídos difíceis de serem detectado. Um exemplo

de mapeamento e a transformada de Fourier aplicada em dados que representam sinais sonoros.

Em determinadas situações, a criação de novos atributos pode facilitar o processo de mineração de dados. Considere, por exemplo, uma mineração de dados em uma base de dados contendo informações de saúde de pessoas com o objetivo de detectar doenças relacionadas à obesidade. Um atributo de índice de massa corporal (IMC) que combina o peso e a altura das pessoas pode facilitar muito essa tarefa.

2.4.5 Normalização

Alguns algoritmos de mineração de dados tendem a ter uma acurácia melhor quando quando as faixas de valores dos atributos de entrada não é muito grande. Isso ocorre principalmente nos algoritmos que usam algum tipo de medida de distância para comparar objetos. Por exemplo, um peso entre 60 e 80 kg vai interferir mais na comparação de dois objetos do que uma altura entre 1,5 e 1,8 m. Portanto, em determinadas situações é interessante normalizar os dados antes do processo de mineração. A normalização coloca todas os atributos dentro de uma mesma faixa de valores, normalmente entre 0 e 1 (zero e um). A Figura 83 mostra um exemplo de normalização.

Em nosso exemplo de normalização, utilizamos um base de dados sobre vinhos (*wine*) na qual as faixas de valores são muito discrepantes. Isso pode ser visto no resultado da execução do código mostrado na Figura 84. Para a classificação dos dados, usamos o algoritmo dos k vizinhos mais próximos ou *k nearest neighbors* (KNN). Tal algoritmo compara a distância entre os objetos e considera a classe predominante dos k vizinhos mais próximos. Como o KNN utiliza uma medida de distância, sua acurácia acaba sendo prejudicada por dados não normalizados.

A normalização foi feita com a função **normalize()** do módulo **preprocessing** da biblioteca **sklearn**. Utilizamos o parâmetro axis=0 porque queremos fazer a normalizadão por coluna. Após a normalização a acurácia do algoritmo aumenta consideravelmente de 78,65% para 97,75%.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import preprocessing
from sklearn import datasets

# Base de dados de vinhos
dados = datasets.load_wine()
X = dados.data
Y = dados.target
# Classificador KNN
knn = KNeighborsClassifier()
# Classificação dos dados originais
knn.fit(X, Y)
acuracia = knn.score(X, Y)
print('Dados originais:')
print('Valor mínimo:', X.min())
print('Valor máximo:', X.max())
print('Acurárcia:', acuracia)
# Normalização
X = preprocessing.normalize(X, axis=0)
# Classificação dos dados após seleção
knn.fit(X, Y)
acuracia = knn.score(X, Y)
print('\nDados normalizados:')
print('Valor mínimo:', X.min())
print('Valor máximo:', X.max())
print('Acurárcia:', acuracia)
```

Figura 83 – Exemplo de normalização de atributos
Fonte: Elaborado pelo Autor.

```
Dados originais:
Valor mínimo: 0.13
Valor máximo: 1680.0
Acurárcia: 0.7865168539325843

Dados normalizados:
Valor mínimo: 0.011273438211330495
Valor máximo: 0.17520791358804919
Acurárcia: 0.9775280898876404
```

Figura 84 – Resultado da normalização de atributos
Fonte: Elaborado pelo Autor.

2.4.6 Discretização

Alguns algoritmos de mineração de dados podem requerer ou funcionar melhor com dados discretizados. A discretização é a transformação de atributos contínuos em atributos discretos. Podem ser usadas estratégias baseadas em intervalos uniformes, em quantidade de elementos ou em centroides. Na discretização baseada em intervalos uniformes, todos os intervalos tem o mesmo tamanho. A estratégia baseada em quantidade faz com que todos os intervalos tenham o mesmo número de elementos. Já a estrategia baseada em centroides, também chamada de *k-means*, considera pontos que representam as médias dos intervalos para agrupar os elementos mais próximos dessas médias.

Na Figura 85, mostramos um exemplo de discretização na base de dados *wine* e usamos o algoritmo KNN para comparar a acurácia antes e depois da discretização. A discretização é feita por meio da classe **KBinsDiscretizer**. Em nosso exemplo, definimos que a discretização deve considerar 4 (quatro) intervalos através do parâmetro **n_bins**. A estratégia utilizada pode ser informada por meio do parâmetro **strategy** na instanciação da classe. Os possíveis valores são:

- **'uniform'** (baseada em intervalos uniformes);
- **'quantile'** (baseada em quantidade), valor padrão se nenhuma for especificada;
- **'kmeans'** (baseada em centroides).

O resultado da execução do código é mostrado na Figura 86. Podemos observar que, após a discretização, a acurácia passou de 78,65% para 96,72%.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import KBinsDiscretizer
from sklearn import datasets

# Base de dados de vinhos
dados = datasets.load_wine()
X, Y = dados.data, dados.target
# Classificador KNN
knn = KNeighborsClassifier()
# Classificação dos dados originais
knn.fit(X, Y)
acuracia = knn.score(X, Y)
print('Dados originais:')
print('Acurárcia:', acuracia)
# Discretização
discretizador = KBinsDiscretizer(n_bins=4)
X = discretizador.fit_transform(X)
# Classificação dos dados discretizados
knn.fit(X, Y)
acuracia = knn.score(X, Y)
print('\nDados discretizados:')
print('Acurárcia:', acuracia)
```

Figura 85 – Exemplo de discretização
Fonte: Elaborado pelo Autor.

```
Dados originais:
Acurárcia: 0.7865168539325843

Dados discretizados:
Acurárcia: 0.9662921348314607
```

Figura 86 – Resultado da exemplo de discretização
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

2.5 Exercícios

A) Implemente uma função para fazer amostragem com a mesma quantidade de elementos para cada classe:

- A função deve receber como parâmetros um **DataFrame** com os dados de entrada, o nome do atributo de classe e a quantidade de registros amostrados para cada classe;

- Crie um **DataFrame** vazio para guardar o resultado e utilize o método **value_counts()** para fazer a contagem dos valores no atributo classe;

- Para cada valor de classe faça uma amostragem com a quantidade solicitada recebida como parâmetro e adicione do **DataFrame** de resultado. No momento da amostragem, verifique se a quantidade de valores da classe é maior do que a quantidade solicitada. Em caso negativo, a amostragem deve ser feita com substituição;

- Utilize a base de dados **penguins** da biblioteca **seaborn** para comparar a contagem de classes nos dados originais e na amostragem realizada com a função implementada.

B) Escreva um código para testar diferentes estratégias e números de intervalos para um classificador:

- Crie uma função para discretizar e retornar a acurácia de um classificador. A função deve receber como parâmetros o classificador, os *arrays* de dados e de classe, o número de intervalos e a estratégia;

- Implemente a função principal para carregar a base de dados de vinhos (**wine**) da biblioteca **skearn**. Mostrar a acurácia do algoritmo nos dados originais e mostrar as acurácias variando as estratégias e números de intervalos;

- Considere os números de intervalos de 2 a 7 e o algoritmo KNN para os testes.

2.6 Respostas dos exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Implemente uma função para fazer amostragem com a mesma quantidade de elementos para cada classe:

```python
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import pandas as pd
from seaborn import load_dataset

def amostragem_classes(dados, atributo_classe, quantidade):
    '''Amostragem com quantidade por classe'''
    # Contagem das classes
    contagem_classes = dados[atributo_classe].value_counts()
    # Dataframe vazio para guardar amostras
    amostras = pd.DataFrame()
    # Para cada classe na contagem
    for classe in contagem_classes.index:
        # Filtra os dados pela classe
        dados_classe = dados[dados[atributo_classe] == classe]
        if len(dados_classe) > quantidade:
            # Amostragem sem substituição
            amostras_classe = dados_classe.sample(n=quantidade)
        else:
            # Amostragem com substituição
            amostras_classe = dados_classe.sample(n=quantidade,
                                                  replace=True)
        # Inclui amostras da classe no resultado
        amostras = amostras.append(amostras_classe)
    return amostras

# Teste com a base de dados penguins
dados = load_dataset('penguins')
atributo_classe = 'species'
print('Dados originais:\n', dados[atributo_classe].value_counts(),
      sep='')
amostra = amostragem_classes(dados, atributo_classe, 100)
print('\nAmostra:\n', amostra[atributo_classe].value_counts(),
      sep='')
```

B) Escreva um código para testar diferentes estratégias e números de intervalos para um classificador:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import KBinsDiscretizer
from sklearn import datasets


def discretiza_classifica(classificador, X, Y, num_intervalos,
                          estrategia):
    # Cria discretizador
    discretizador = KBinsDiscretizer(n_bins=num_intervalos,
                                     strategy=estrategia)
    # Discretiza
    X_discretizado = discretizador.fit_transform(X)
    # Classifica
    classificador.fit(X_discretizado, Y)
    return classificador.score(X_discretizado, Y)

def principal():
    # Base de dados de vinhos
    dados = datasets.load_wine()
    X, Y = dados.data, dados.target
    # Listas de números de intervalos e de estratégias
    lista_intervalos = [n for n in range(2, 8)]
    lista_estrategias = ['uniform', 'quantile', 'kmeans']
    # Classificador KNN
    knn = KNeighborsClassifier()
    knn.fit(X, Y)
    print('Acurácias obtidas')
    print('Dados originais:', knn.score(X, Y))
    # Percorre lista de números de intervalos e de estratégias
    for estrategia in lista_estrategias:
        for num_intervalos in lista_intervalos:
            acuracia = discretiza_classifica(knn, X, Y,
                                             num_intervalos,
                                             estrategia)
            print('Estrategia ', estrategia, ' (', num_intervalos,
                  '): ', acuracia, sep='')

if __name__ == '__main__':
    principal()
```

2.7 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista aos vídeos “Revisão da Segunda Semana - Parte 1” e “Revisão da Segunda Semana - Parte 2”.

Nos encontramos na próxima semana.

Bons estudos!

Objetivos
- Conhecer os conceitos básicos de técnicas de classificação de dados;
- Desenvolver e entender códigos para classificação de dados.

3.1 Introdução

A classificação consiste em identificar uma classe ou categoria para um objeto. Alguns exemplos de aplicações da classificação são detecção de mensagens de spam, classificação de clientes em perfis de crédito, identificação de tipos de galáxias, etc. Os dados usadas para classificação, normalmente, são conjuntos de registros sendo que cada registro tem um atributo discreto que identifica sua classe.

O objetivo da classificação é construir um modelo com base em objetos previamente classificados. Essa fase é chamada de treinamento. O modelo construído pode então ser usado para descrever o conjunto de objetos ou para prever a classe de novos objetos. No caso da previsão, é importante que o modelo seja genérico o suficiente para atingir uma boa taxa de acerto.

Os erros cometidos na classificação são divididos em erros de treinamento e erros de generalização. Os erros de treinamento são as classificações incorretas realizadas durante o treinamento. Já os erros de generalização dizem respeito à classificações erradas sobre registros nunca vistos anteriormente.

Um bom modelo de classificação deve ter poucos erros de treinamento e, principalmente poucos erros de generalização. Quando um modelo possui baixo erro de treinamento e alto erro de generalização, dizemos que ocorreu superajustamento ou *overfiting*.

3.2 Avaliação de classificadores

Em geral, é importante medir o desempenho de algoritmos de classificação no conjunto de testes para se obter uma estimativa imparcial do erro de generalização. A precisão ou a taxa de erro calculada a partir do conjunto de testes também pode ser utilizada para comparar diferentes classificadores.

3.2.1 Estratégias de avaliação

As duas estratégias básicas para avaliar o desempenho de métodos de classificação são a divisão e a validação cruzada. A estratégia de divisão, também chamada de *holdout* ou *split-sample*, divide a base de dados em uma partição de treinamento e outra de teste. Normalmente, a partição de treinamento é maior (cerca de 2/3 ou 70% do total de registros). O algoritmo faz o treinamento na partição maior e

contabiliza a taxa de acerto classificando os registros da partição de teste. Como cada registro já possui sua classe, é possível verificar se o classificador acertou.

A Figura 87 mostra um exemplo comparando a acurácia de um classificador sem e com a divisão da base de dados em partições de treino e de teste. Utilizamos a base de dados de vinhos da **sklearn** e o classificador de árvore de decisão.

```
from sklearn.datasets import load_wine
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

# Carrega a base de dados
X, Y = load_wine(return_X_y=True)
# Cria classificador de árvore de decisão
arvore = DecisionTreeClassifier(random_state=42)
# Calcula acurácia sem divisão da base de dados
arvore.fit(X, Y)
acuracia = arvore.score(X, Y)
print('Acurácia sem divisão:', acuracia)
# Divisão da base de dados
X_treino, X_teste, Y_treino, Y_teste = \
    train_test_split(X, Y, train_size=0.7, random_state=42)
# Acurácia com divisão da base em treino e teste
arvore.fit(X_treino, Y_treino)
acuracia = arvore.score(X_teste, Y_teste)
print('Acurácia após divisão:', acuracia)
```

Figura 87 – Código para comparação de acurácia sem e com divisão da base de dados
Fonte: Elaborado pelo Autor.

Na primeira utilização do classificador, fazemos o treinamento e teste em toda a base de dados. Com isso, atingimos uma acurácia de 100% (1.0). Contudo, sabemos que esse valor pode ser enganoso porque o classificador foi treinado e testado com os mesmos dados. Portanto, repetimos o treinamento e teste com a base de dados dividida em treinamento e teste. A Figura 88 mostra as acurácias dos dois experimentos. Podemos observar que utilizando a divisão da base de dados, a acurácia caiu para 96,3% (0.96296...) mostrando um valor mais condizente do algoritmo quando consideramos a classificação de dados nunca vistos.

```
Acurácia sem divisão: 1.0
Acurácia após divisão: 0.9629629629629629
```

Figura 88 – Resultado da comparação de acurácia sem e com divisão da base de dados
Fonte: Elaborado pelo Autor.

A divisão da base de dados foi realizada com a função **train_test_split()** do módulo **model_selection**. Essa função recebe os dados (**X**) e classes (**Y**) e o tamanho da partição de dados de treinamento (**train_size**) que definimos como 70% (0.7) em relação ao tamanho da base de dados original. Nós também definimos o parâmetro **random_state** para tornar o experimento reprodutível, uma vez que a função **train_test_split()** embaralha os dados para fazer o particionamento.

Quando temos uma base de dados pequena e retiramos 1/3 ou 30% dos dados, podem sobrar poucos dados para treinamento levando a um modelo de baixa precisão.

Uma alternativa para essa situação é usar uma técnica de validação cruzada. Basicamente, na validação cruzada, a base de dados é particionada em subconjuntos mutuamente exclusivos para que alguns subconjuntos sejam usados para treinamento e os demais para teste. Dizemos que conjuntos são mutuamente exclusivos quando os elementos de um conjunto não aparecem em nenhum dos outros conjuntos.

O tipo de validação cruzada mais utilizado é o *k-fold* que divide a base de dados em k subconjuntos mutualmente exclusivos. O experimento de classificação é então executado k vezes de forma que, a cada execução, k-1 subconjuntos são usados para treinamento e um é usado para teste. A acurácia é a média da acurácia de todas as execuções.

```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, cross_val_score

# Carrega a base de dados
X, Y = load_wine(return_X_y=True)
# Cria classificador de árvore de decisão
arvore = DecisionTreeClassifier(random_state=42)
# Divisão da base de dados
X_treino, X_teste, Y_treino, Y_teste = \
    train_test_split(X, Y, train_size=0.7, random_state=42)
# Acurácia com divisão da base em treino e teste
arvore.fit(X_treino, Y_treino)
acuracia = arvore.score(X_teste, Y_teste)
print('Acurácia com divisão:', acuracia)
# Acurácia com validação cruzada
acuracia = np.mean(cross_val_score(arvore, X, Y, cv=10))
print('Acurácia com validação cruzada:', acuracia)
```

Figura 89 – Código para comparação de acurácia com divisão da base de dados e com validação cruzada
Fonte: Elaborado pelo Autor.

A Figura 89 mostra um exemplo onde comparamos a acurácia com divisão da base de dados e a acurácia com validação cruzada *k-fold*. Nesse exemplo usamos um classificador de árvore de decisão (**DecisionTreeClassifier**). A validação cruzada é feita com a função **cross_val_score()** do módulo **model_selection** que recebe como parâmetros o classificador (**arvore**), dados (**X**), classe dos dados (**Y**) e a quantidade de partições (**cv**). A função faz o particionamento dos dados, executa o classificador múltiplas vezes e retorna uma lista com a acurácia de cada execução. Para chegarmos à acurácia final, usamos a função **mean()** da biblioteca **numpy** para calcular a média sobre a lista das acurácias recebida.

```
Acurácia com divisão: 0.9629629629629629
Acurácia com validação cruzada: 0.865032679738562
```

Figura 90 – Resultado da comparação de acurácia com divisão da base de dados e com validação cruzada
Fonte: Elaborado pelo Autor.

Na Figura 90, podemos observar que acurácia da validação cruzada é menor em relação à acurácia com divisão da base de dados. A validação cruzada *k-fold* é uma das técnicas mais utilizada para avaliar classificadores porque podemos obter uma acurácia muito próxima da acurácia real do algoritmo para dados nunca vistos.

3.2.2 Métricas de avaliação

Uma importante ferramenta para analisar classificadores é a chamada matriz de confusão que sumariza como os registros foram classificados. O número de classificações corretas fica na diagonal principal, enquanto o número de classificações incorretas fica nas demais posições.

		Classificado como	
		spam	*não spam*
Real	*spam*	537	43
	não spam	16	5834

Figura 91 – Matriz de confusão de classificação de *spam*
Fonte: Elaborado pelo Autor.

A Figura 91 exibe uma matriz de confusão para uma classificação de registros representando mensagens de *spam*. A partir dessa matriz podemos constatar as seguintes informações:

- Verdadeiros positivos (VP): registros de *spam* classificados como *spam* (537);
- Verdadeiros negativos (VN): registros não *spam* classificados como não *spam* (5834);
- Falsos positivos (FP): registros não *spam* classificados como *spam* (16);
- Falsos negativos (FN): registros de *spam* classificados como não *spam* (43).

A matriz de confusão também pode ser gerada para classificações envolvendo mais de uma classe (multi-classe). Como exemplo vamos considerar o código da Figura 92 que faz a classificação e extrai a matriz de confusão para a base de dados de vinhos.

```
from sklearn.datasets import load_wine
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_predict
from sklearn.tree import DecisionTreeClassifier

# Carrega base de dados
dados = load_wine()
X, Y = dados.data, dados.target
# Cria classificador de árvore de decisão
arvore = DecisionTreeClassifier(max_depth=2, random_state=42)
# Classificação com validação cruzada
previsoes = cross_val_predict(arvore, X, Y, cv=10)
# Matriz de confusão
matriz = confusion_matrix(Y, previsoes)
for cont, linha in enumerate(matriz):
    print(linha, ':', dados.target_names[cont])
```

Figura 92 – Código para extrair matriz de confusão
Fonte: Elaborado pelo Autor.

O resultado da execução do código para extrair a matriz e confusão é mostrado na Figura 93. A diagonal principal mostra os números de classificações corretas. Para cada

classe, se somarmos os valores da coluna excepto o valor da diagonal, teremos o número de falsos positivos. A soma das linhas funciona de forma análoga para os falsos negativos. Como exemplo, para a classe **class_1** temos 9 falsos positivos (4 + 5) e 14 falsos negativos (10 + 5). Isso significa que 9 registros foram classificados como **class_1**, mas são de outras classes, e 15 registros foram classificados como outras classes, mas deveriam ser classificados como **class_1**.

```
[52  4  3] : class_0
[10 56  5] : class_1
[ 1  5 42] : class_2
```

Figura 93 – Resultado do código para extrair matriz de confusão
Fonte: Elaborado pelo Autor.

Para os experimentos realizados até o momento, comparamos os classificadores usando apenas a medida de acurácia. O cálculo da mesma é feito conforme a Equação (1). Na prática a acurácia pode ser calculada dividindo-se a quantidade de acertos pelo número total de registros.

$$acurácia = \frac{VP + VN}{VP + FP + VN + FN} \quad (1)$$

A acurácia não recomendada para banco de dados desbalanceados, ou seja, quando a distribuição das classes não é uniforme. Nesses cenários, o valor da acurácia pode não representar corretamente a realidade. Outras medidas interessantes a serem consideradas são a revocação (ou *recall*), a precisão e a *f1-score* (também conhecida como *f-score* ou *f-measure*).

A precisão e a revocação são interessantes para medir o quanto o classificador acertou cada classe individualmente. A precisão analisa o total de acertos considerando falsos positivos enquanto revocação analisa o total de acertos com respeito aos falsos negativos, conforme mostrados nas Equações (2) e (3).

$$precisão = \frac{VP}{VP + FP} \quad (2)$$

$$revocação = \frac{VP}{VP + FN} \quad (3)$$

A *f1-score* representa a média harmônica da precisão e revocação. Assim, para aumentarmos a *f1-score*, tanto a precisão quanto a revocação precisam ser melhoradas. A Equação (4) apresenta a fórmula da *f1-score*.

$$f1\text{-}score = \frac{2 \times precisão \times revocação}{precisão + revocação} \quad (4)$$

Para exemplificar, vamos calcular as quatro medidas descritas para a matriz de confusão da Figura 91. O resultado é o seguinte:

- Acurácia: (537 + 5834) / (537 + 16 + 5834 + 43) ≈ 0,9908 (99,08%)
- Precisão: 537 / (537 + 16) ≈ 0,9711 (97,11%)
- Revocação: 537 / (537 + 43) ≈ 0,9259 (92,59%)

- f1-score: (2 × 0,9711 × 0,9259) / (0,9711 + 0,9259) ≈ 0,9480 (94,80%)

Nós podemos também calcular as medidas estudadas para uma classe específica. Considere, por exemplo, a classe **class_0** da da matriz de confusão mostrada na Figura 93. Para a referida classe, temos 52 verdadeiros positivos (VP), 98 (56 + 42) verdadeiros negativos (VN), 11 (10 + 1) falsos positivos (FP) e 7 (4 + 3) falsos negativos (FN).

O cálculo das medidas para essa classe pode ser feito da seguinte maneira:

- Acurácia: (52 + 98) / (52 + 11 + 98 + 7) ≈ 0,8923 (89,23%)
- Precisão: 52 / (52 + 11) ≈ 0,8254 (82,54%)
- Revocação: 52 / (52 + 7) ≈ 0,8814 (88,14%)
- f1-score: (2 × 0,8254 × 0,8814) / (0,8254 + 0,8814) ≈ 0,8525 (85,25%)

```
from sklearn.datasets import load_wine
from sklearn.metrics import precision_recall_fscore_support
from sklearn.model_selection import cross_val_predict
from sklearn.tree import DecisionTreeClassifier

# Carrega base de dados
data = load_wine()
X, Y = data.data, data.target
# Classificador e classificação
arvore = DecisionTreeClassifier(max_depth=2, random_state=42)
previsoes = cross_val_predict(arvore, X, Y, cv=10)

# Medidas para cada classe
precicao, revocacao, f1score, suporte = \
    precision_recall_fscore_support(Y, previsoes)
for posicao, classe in enumerate(data.target_names):
    print('\nClasse:', classe, '(suporte:', suporte[posicao], ')')
    print('- Precisão:', round(precicao[posicao], 4))
    print('- Revocação:', round(revocacao[posicao], 4))
    print('- F1-score:', round(f1score[posicao], 4))

# Média simples das medidas
precicao, revocacao, f1score, _ = \
    precision_recall_fscore_support(Y, previsoes, average='macro')
print('\nGeral (média simples):')
print('- Precisão:', round(precicao, 4))
print('- Revocação:', round(revocacao, 4))
print('- F1-score:', round(f1score, 4))

# Média ponderada das medidas
precicao, revocacao, f1score, _ = \
    precision_recall_fscore_support(Y, previsoes, average='weighted')
print('\nGeral (média ponderada):')
print('- Precisão:', round(precicao, 4))
print('- Revocação:', round(revocacao, 4))
print('- F1-score:', round(f1score, 4))
```

Figura 94 – Código para medidas de avaliação para cada classe e a média das medidas
Fonte: Elaborado pelo Autor.

O cálculo das medidas para uma classificação como um todo é obtido por meio da média das medidas de cada classe. O código da Figura 94 mostra um exemplo de código que calcula as medidas de cada classe é a média da classificação para a base de dados de vinhos.

```
Classe: class_0 (suporte: 59 )
- Precisão: 0.8254
- Revocação: 0.8814
- F1-score: 0.8525

Classe: class_1 (suporte: 71 )
- Precisão: 0.8615
- Revocação: 0.7887
- F1-score: 0.8235

Classe: class_2 (suporte: 48 )
- Precisão: 0.84
- Revocação: 0.875
- F1-score: 0.8571

Geral (média simples):
- Precisão: 0.8423
- Revocação: 0.8484
- F1-score: 0.8444

Geral (média ponderada):
- Precisão: 0.8438
- Revocação: 0.8427
- F1-score: 0.8422
```

Figura 95 – Resultado do código para extrair matriz de confusão
Fonte: Elaborado pelo Autor.

O cálculo das medidas para cada classe foi realizado com a função **precision_recall_fscore_support()**. Quando passamos apenas os parâmetros **Y** (valor real de cada classe) e **previsoes** (valor predito para cada classe), essa função retorna vetores com o valor de medida de cada classe. As medidas retornadas são precisão, revocação, *f1-score* e suporte. O suporte é apenas a contagem de cada valor de classe. Depois de calcularmos as medidas, fazemos um laço de repetição para exibir os valores para cada classe.

Quando desejamos obter a média das medidas para toda a classificação, podemos utilizar incluir o parâmetro **average** na função **precision_recall_fscore_support()**. Se o parâmetro tiver o valor 'macro', o resultado será a média simples das medidas. Por outro lado, se **average='weighted'**, a função retorna a média ponderada das medidas. Usamos uma variável anônima (_) para receber o valor do suporte, porque ele retornado como **None** no caso das médias e, dessa forma, não tem aplicação prática. Para o cálculo, individual das medidas de precisão, revocação e *f1-score*, podem ser usadas, respectivamente, as funções **precision_score()**, **recall_score()**, **f1_score()** do módulo **sklearn.metrics**.

3.3 Árvores de decisão

A ideia básica de uma árvore de decisão é quebrar um problema maior e mais complexo em problemas menores que tenham complexidade menor. Essa ideia é aplicada de forma recursiva sobre o conjunto de dados até que a árvore de decisão seja construída. A Figura 96 mostra um exemplo de árvore de decisão usada para classificar flores da família íris. O nó do topo é chamado de raiz, os retângulos representam nós folhas com as classes e as elipses representam nós internos com atributos.

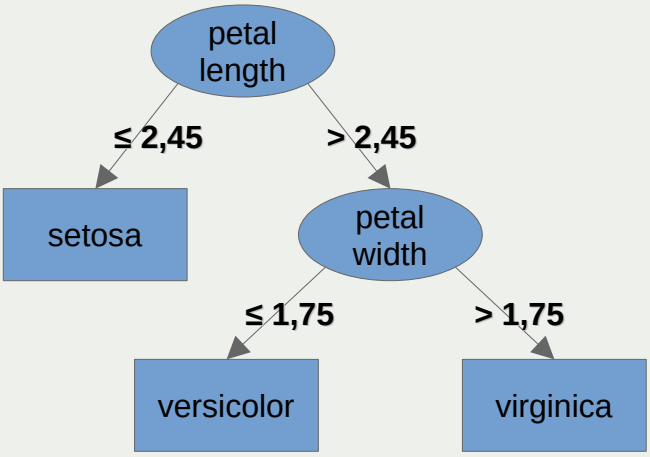


Figura 96 – Exemplo de árvore de decisão para classificar flores íris
Fonte: Elaborado pelo Autor.

A classificação de um registro começa pela raiz. Para cada no interno, tomamos o atributos correspondente no registro e seguimos as setas conforme o valor desse atributo no registro. Ao chegamos em uma folha encontramos a classe do registro. Como exemplo vamos classificar um registro com os seguintes atributos:

- **sepal length**: 7;
- **sepal width**: 3.2;
- **petal length**: 4.7;
- **petal width**: 1.4.

Começando pela raiz, observamos que **petal length** (4,7) é maior que 2,45 e descemos para o nó do atributo **petal width**. Nesse momento, percebemos que **petal width** (1.4) é menor ou igual a 1,75 e chegamos à classe **versicolor**, obtendo a classificação do registro.

Dependendo da quantidade de registros na base de dados, pode existir um enorme número de possíveis árvores de decisão. Para resolver esse problemas, os pesquisadores da área já desenvolveram diversos algoritmos eficientes. Na prática, a grande maioria dos métodos já criados segue a ideia do geral do método de Hunt.

Um dos algoritmos baseados no método de Hunt é o ID3, sua ideia geral é mostrada na Figura 97. O parâmetro **N** é a raiz da árvore criada inicialmente com todos os registros e o parâmetro **A** é o conjunto de atributos a serem considerados. Na primeira chamada do algoritmo, o conjunto **A** contém todos os atributos exceto o atributo de classe.

```
ArvoreID3(N, A):
 1) Se todos os registros de N são da mesma classe C, então:
 2)   Transforme N em nó folha rotulado com c
 3) Senão se A = {}, então:
 4)   Transforme N em nó folha rotulado com a clase mais frequente
 4) Senão:
 5)   X = Ganho(N, A)
 6)   Rotula N com atributo X
 7)   Para cada valor v de X:
 8)     Crie nó filho F com ramo rotulado com v
 9)     Associa a F o conjunto D' de registros com X = v
10)     Chama ArvoreID3(F, A - {X})
```

Figura 97 – Algoritmo de Hunt para construção de árvore de decisão
Fonte: Elaborado pelo Autor.

O principal detalhe do algoritmo **ArvoreID3** é a função **Ganho()** que calcula uma medida de ganho de informação para os atributos de **A** e retorna aquele com maior ganho. A medida de ganho de informação serve para selecionar o atributo que melhor particiona os registros de um nó. Alguns exemplos de medidas são o índice de Gini e a entropia.

```
from matplotlib import pyplot as plt
from sklearn import datasets
from sklearn import tree

# Carrega base de dados
dados = datasets.load_iris()
X, Y = dados.data, dados.target
# Cria arvore
arvore = tree.DecisionTreeClassifier(max_depth=2, random_state=42)
arvore.fit(X, Y)
# Imprime árvore
print(tree.export_text(arvore))
# Árvore como figura
plt.figure(figsize=(12, 8))
tree.plot_tree(arvore,
               feature_names=dados.feature_names,
               class_names=dados.target_names,
               filled=True)
```

Figura 98 – Código para criar e mostrar árvore de decisão
Fonte: Elaborado pelo Autor.

A Figura 98 mostra um exemplo de criação de uma árvore de decisão para a base de dados íris. Depois da criação da árvore, utilizamos a função **export_text()** do módulo **tree** para mostrar um representação textual da árvore na tela. Em seguida, usamos a função **plot_tree()** do mesmo módulo para mostrar a representação gráfica da árvore no formato de uma figura. Como a função **plot_tree()** usa o módulo **pytplot** da biblioteca **matplotlib**, nós importamos também esse módulo para aumentar o tamanho da figura passando o parâmetro **figsize** para a função **figure()**.

A Figura 99 mostra o resultado da execução do código para criar e mostrar a árvore de decisão. Infelizmente, a representação textual, não mostra nem os nomes dos atributos, nem os nomes das classes. Por outro lado, a representação gráfica mostra, além desses dados, outras informações interessantes como o valor da medida de ganho de informação

(**gini**), a quantidade de registros analisados em cada nó (**samples**), a quantidade de registros de cada classe (**value**) e a classe mais frequente (**class**).

```
|--- feature_2 <= 2.45
|   |--- class: 0
|--- feature_2 >  2.45
|   |--- feature_3 <= 1.75
|   |   |--- class: 1
|   |--- feature_3 >  1.75
|   |   |--- class: 2
```

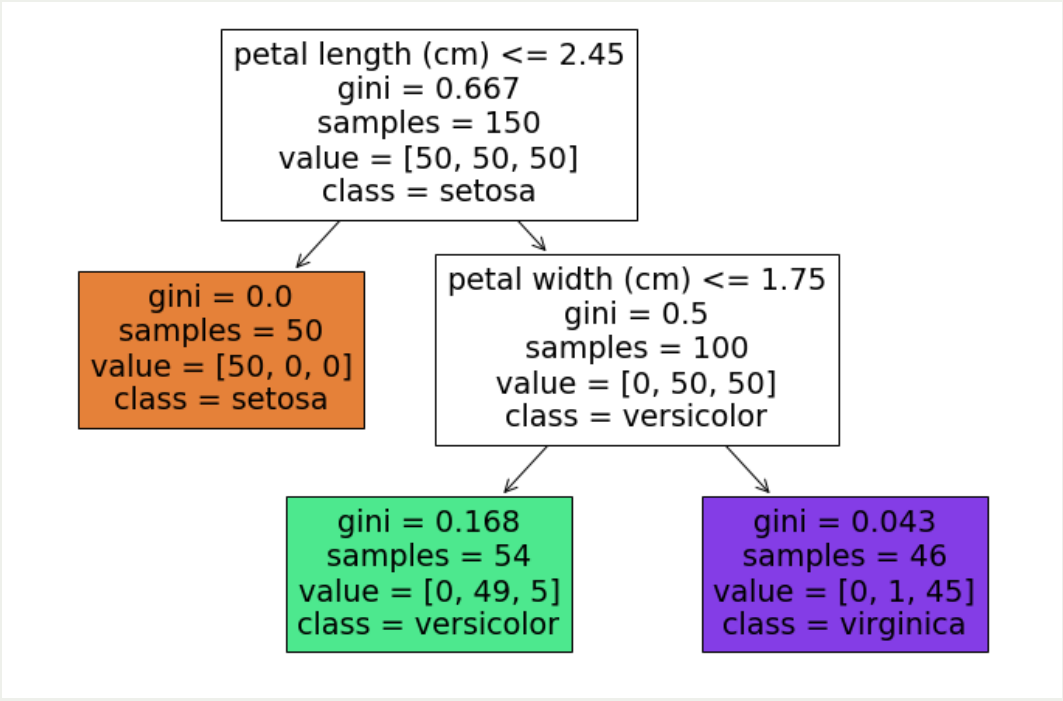


Figura 99 – Resultado da execução do código para criar e mostrar árvore de decisão
Fonte: Elaborado pelo Autor.

```
from sklearn import datasets
from sklearn import tree

# Carrega base de dados
dados = datasets.load_iris()
X, Y = dados.data, dados.target
# Cria árvore
arvore = tree.DecisionTreeClassifier(max_depth=2, random_state=42)
arvore.fit(X, Y)
# Faz classificação de registro
registro = [7, 3.2, 4.7, 1.4]
num_classe = arvore.predict([registro])[0]
classe = dados.target_names[num_classe]
print('Classificando registro:', registro)
print('Classe:', classe)
```

Figura 100 – Código para criar árvore de decisão e classificar um registro
Fonte: Elaborado pelo Autor.

Depois de criarmos uma árvore, podemos classificar novos registros usando o método **predict()**. A Figura 100 mostra um exemplo de código que realiza essa tarefa. O

resultado da execução do código pode ser visto na Figura 101. O método **predict()** recebe uma vetor de registros para classificação e retorna um vetor com as classes de cada registro recebido. Como estamos classificando um único registro, devemos incluí-lo em uma lista (**[registro]** como parâmetro do método) e pegar a primeira classe do resultado (**[0]** no resultado do método).

```
Classificando registro: [7, 3.2, 4.7, 1.4]
Classe: versicolor
```

Figura 101 – Execução do código para criar árvore de decisão e classificar um registro
Fonte: Elaborado pelo Autor.

A medida de ganho de informação utilizada por padrão é o índice de Gini (**'gini'**). Além dela, a classe **DecisionTreeClassifier** permite a utilização da entropia (**'entropy'**) com a utilização do parâmetro **criterion** na instanciação da classe. O cálculo da entropia é um pouco mais lento, mas, em geral, a árvore de decisão tende a apresentar resultados mais interessantes.

```
from sklearn import datasets
from sklearn import tree
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import precision_recall_fscore_support

dados = datasets.load_wine()
X, Y = dados.data, dados.target
lista_medidas = ['gini', 'entropy']
for medida in lista_medidas:
    arvore = tree.DecisionTreeClassifier(random_state=42,
                                         criterion=medida)
    previsoes = cross_val_predict(arvore, X, Y, cv=10)
    precicao, revocacao, f1score, _ = \
        precision_recall_fscore_support(Y, previsoes,
                                        average='weighted')
    print('Medida:', medida)
    print('- Precisão:', round(precicao, 4))
    print('- Revocação:', round(revocacao, 4))
    print('- F1-score:', round(f1score, 4))
```

Figura 102 – Código para comparação de medidas de ganho de informação
Fonte: Elaborado pelo Autor.

A Figura 102 apresenta um código que compara o desempenho das medidas de ganho. O resultado do código é apresentado na Figura 103. Como podemos, observar a árvore que utilizou a entropia, obteve melhor desempenho.

```
Medida: gini
- Precisão: 0.8656
- Revocação: 0.8652
- F1-score: 0.8646
Medida: entropy
- Precisão: 0.9173
- Revocação: 0.9157
- F1-score: 0.916
```

Figura 103 – Execução do código para comparação de medidas de ganho de informação
Fonte: Elaborado pelo Autor.

O parâmetro **criterion** é chamado de hiperparâmetro do algoritmo de classificação. Cada algoritmo de classificação pode possuir hiperparâmetros diferentes que podem afetar seu funcionamento e, principalmente, seu desempenho. No caso da árvore de decisão, um hiperparâmetro muito importante é a profundidade máxima da árvore, definida pelo parâmetro **max_depth** na instanciação da classe. A Figura 104 mostra um experimento de teste na variação da profundidade máxima da árvore de decisão sobre uma base de dados de dígitos manuscritos digitalizados.

```
from sklearn import datasets
from sklearn import tree
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import f1_score

dados = datasets.load_digits()
X, Y = dados.data, dados.target
print('Diferentes profundidades:')
for profundidade in range(1, 16):
    arvore = tree.DecisionTreeClassifier(random_state=42,
                                         criterion='entropy',
                                         max_depth=profundidade)
    previsoes = cross_val_predict(arvore, X, Y, cv=10)
    f1score = f1_score(Y, previsoes, average='weighted')
    print('Profundidade', profundidade, ':', round(f1score, 4))
```

Figura 104 – Experimento comparando a profundidade máxima de uma árvore de decisão
Fonte: Elaborado pelo Autor.

No experimento, variamos a profundidade da árvore de decisão de 1 a 15. A Figura 105 mostra o resultado do experimento. Podemos notar que o *f1-score* aumenta gradualmente até a profundidade 11, chegando a 81,7%. Depois disso, ele cai para 81,41% e mantém esse mesmo *f1-score* para as demais profundidades. Isso acontece porque ocorre um *overfitting*, a árvore se especializa muito nos registros treinados, mas começa a errar mais novos registros. Assim, quando trabalhamos com árvores de decisão, é importante limitar a profundidade máxima para evitar esse tipo de problema.

```
F1-score em diferentes profundidades:
Profundidade 1 : 0.1182
Profundidade 2 : 0.3064
Profundidade 3 : 0.4738
Profundidade 4 : 0.6411
Profundidade 5 : 0.7413
Profundidade 6 : 0.7757
Profundidade 7 : 0.8147
Profundidade 8 : 0.8123
Profundidade 9 : 0.8123
Profundidade 10 : 0.8109
Profundidade 11 : 0.817
Profundidade 12 : 0.8141
Profundidade 13 : 0.8141
Profundidade 14 : 0.8141
Profundidade 15 : 0.8141
```

Figura 105 – Resultado do experimento comparando a profundidade máxima de uma árvore de decisão
Fonte: Elaborado pelo Autor.

A construção de árvores de decisão não requer grande conhecimento sobre o conjunto de dados. Contudo, a implementação disponível na biblioteca **sklearn** não trabalha diretamente com dados nominais. As técnicas desenvolvidas para a construção de árvores de decisão não apresentam grande custo computacional, mesmo quando temos grandes conjuntos de treinamento.

Outra vantagem das árvores de decisão é a classificação rápida de novos registros. Além disso, as árvores de decisão são relativamente simples de serem interpretadas. por fim, a presença de atributos redundantes não afeta negativamente a precisão das árvores de decisão. Esses atributos acabam sendo ignorados.

3.4 Vizinhos mais próximos

Os classificadores podem ser divididos em ávidos e preguiçosos. Os classificadores ávidos constroem um modelo com base no conjunto de treinamento e, posteriormente, utilizam o modelo construído para classificar novos registros. Os classificadores preguiçosos, por sua vez, não criam um modelo e classificam novos registros confrontando-os com os dados de treinamento. As árvores de decisão são um exemplo de classificadores ávidos.

O exemplo mais comum de classificador preguiçoso é o classificador de vizinhos mais próximos que compara um novo registro aos registros de treinamento mais semelhantes para fazer a classificação. Um classificador de vizinho mais próximo representa cada registro como um ponto de dados em um espaço *d*-dimensional, onde *d* é o número de atributos. Calculamos a proximidade de um novo registro em relação aos demais pontos de dados no conjunto de treinamento usando uma medida de distância. Os *k*-vizinhos mais próximos ou *k-nearest neighbors* (KNN) de um registro **z** referem-se aos *k* pontos que estão mais próximos de **z**. A classe do novo registro é a classe predominante nos *k*-vizinhos.

A distância mais utilizada nos algoritmos KNN é a distância euclidiana. Sua fórmula é mostrada na Equação (5), onde **X** e **Y** são os registros para cálculo da distância, X_i e Y_i são os atributos de **X** e **Y**, respectivamente, e **n** é o número de atributos. No caso de atributos não numéricos, podemos considerar 1 (um) para valores distintos e 0 (zero) para valores iguais.

$$distância(X,Y)=\sqrt{\sum_{i=1}^{n}(X_i-Y_i)^2} \quad (5)$$

Dependendo do número de vizinhos considerados, o resultado da classificação pode ser diferente. Considere como exemplo as classificações feitas na Figura 106 e na Figura 107. Na Figura 106, consideramos 3 mais próximos vizinhos e novo registro (ponto azul) é classificado como amarelo. Por outro lado, na Figura 107, pegamos os 5 vizinhos mais próximos e o novo registro é classificado como vermelho.

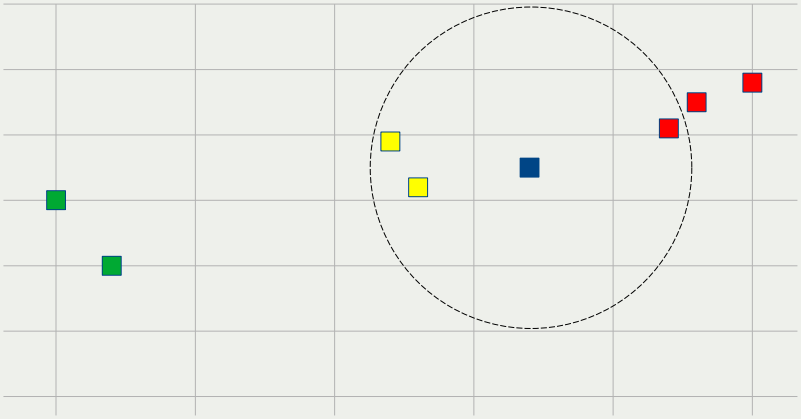


Figura 106 – Classificação considerando 3 vizinhos mais próximos
Fonte: Elaborado pelo Autor.

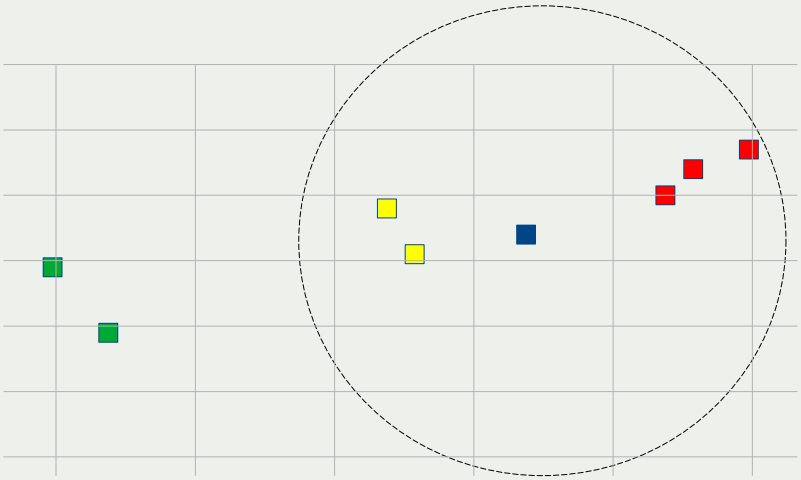


Figura 107 – Classificação considerando 5 vizinhos mais próximos
Fonte: Elaborado pelo Autor.

Ao utilizarmos um algoritmo KNN é importante fazer alguns experimentos variando o número de vizinhos mais próximos para encontramos o valor mais adequado. A Figura 108 mostra um código que executa esse experimento para duas bases de dados.

```
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import f1_score

def varia_k(X, Y):
    for k in range(1, 16, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        previsoes = cross_val_predict(knn, X, Y, cv=10)
        f1score = f1_score(Y, previsoes, average='weighted')
        print('K =', k, ':', round(f1score, 4))

dados = datasets.load_digits()
X, Y = dados.data, dados.target
print('Base de dados digits, f1-score variando k:')
varia_k(X, Y)

dados = datasets.load_iris()
X, Y = dados.data, dados.target
print('Base de dados iris, f1-score variando k:')
varia_k(X, Y)
```

Figura 108 – Experimento com algoritmo KNN variando o número de vizinhos mais próximos
Fonte: Elaborado pelo Autor.

```
Base de dados digits, f1-score
variando k:
K = 1 : 0.975
K = 3 : 0.9767
K = 5 : 0.9711
K = 7 : 0.9677
K = 9 : 0.965
K = 11 : 0.9638
K = 13 : 0.9632
K = 15 : 0.9621

Base de dados iris, f1-score
variando k:
K = 1 : 0.96
K = 3 : 0.9667
K = 5 : 0.9666
K = 7 : 0.9666
K = 9 : 0.9733
K = 11 : 0.9667
K = 13 : 0.98
K = 15 : 0.9733
```

Figura 109 – Resultado do experimento com algoritmo KNN variando o número de vizinhos mais próximos
Fonte: Elaborado pelo Autor.

O resultado da execução do experimento variando o número de vizinhos mais próximos (**k**) é apresentado na Figura 109. Podemos observar que, para a base de dados **digits**, o melhor valor de k foi 3 enquanto que, para a base de dados **iris**, o melhor valor para k foi 13.

Os classificadores preguiçosos, como o KNN, não exigem a construção de modelos, mas classificar um registro pode ser computacionalmente caro. Por outro, os classificadores ávidos consomem um tempo razoável na construção de modelos, mas a classificação é mais rápida. O KNN costuma funcionar bem para grandes conjuntos de dados que possuem poucos atributos.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

3.5 Exercícios

Escreva os códigos em Python para resolver os problemas a seguir:

A) Experimento com classificadores KNN e Árvore de decisão

- Escreva um código para realizar experimentos com os classificadores KNN e árvore de decisão sobre a base de dados de câncer de pulmão disponibilizada pela biblioteca **sklearn**. A base de dados pode ser carregada com a função **load_breast_cancer()** do módulo **datasets**;

- Na execução dos classificadores utilize a validação cruzada com 10 partições e os avalie utilizando a medida *f1-score*;

- Crie a função **experimento_knn(atributos, classe)** para testar o algoritmo KNN variando o número de vizinhos mais próximos (**k**) com números ímpares de 1 a 29. A função deve mostrar o *f1-score* de cada execução e retornar o **k** com melhor resultado;

- Implemente a função **experimento_arvore(atributos, classe)** para testar o algoritmo de árvore de decisão variando a medida de ganho de informação (índice de Gini e entropia) e a profundidade máxima da árvore. Quando à profundidade, devem ser considerados valores de 1 até o número de atributos da base de dados acrescido de 1. A função deve retornar os hiperparâmetros que obtiveram o melhor resultado;

- Faça também uma função principal para chamar as funções de experimento.

B) Classificador de flores íris

- Crie um programa classificador de flores íris que utiliza uma árvore de decisão treinada com a base de dados iris disponível na biblioteca **sklearn**. A base de dados pode ser carregada com a função **load_iris()** do módulo **datasets**.

- Escreva a função **cria_arvore()** para carregar a base de dados, criar e treinar a árvore de decisão. A função deve também salvar a árvore em arquivo usando a função **dump()** da biblioteca **pickle**. No final, a função deve retornar a árvore criada;

- Implemente a função **carrega_arvore()** para carregar a árvore de arquivo, se o mesmo existir, ou chamar a função **cria_arvore()**, em caso negativo. A existência pode ser verificada com a função **isfile()** do módulo **path** da biblioteca **os**. O carregamento da árvore em arquivo pode ser feito com a função **load()** da biblioteca **pickle**. No final, a função deve retornar a árvore carregada ou criada;

- Faça a função classifica() que deve pegar os valores para os atributos de uma flor íris com o usuário e classificar essa flor usando a árvore de decisão. Os atributos de uma flor iris são comprimento e largura da sépala (em cm) e comprimento e largura da pétala (em cm).

- Crie também uma função **principal()** com um laço de repetição que deve ser executado até que o usuário deseje sair. A cada repetição a função deve mostrar as opções (classificar ou sair), pegar a resposta do usuário e fazer a ação escolhida.

3.6 Respostas dos exercícios

A) Experimento com classificadores KNN e Árvore de decisão

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''Experimentos com classificadores KNN e árvore de decisão'''

from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import f1_score
from sklearn.tree import DecisionTreeClassifier

def experimento_knn(atributos, classe):
    '''Experimento com algoritmo KNN'''
    best_score = float('-inf')
    best_k = None
    print('Testando hipermarâmetros:')
    for k in range(1, 30, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        previsoes = cross_val_predict(knn, atributos, classe, cv=10)
        f1score = f1_score(classe, previsoes, average='weighted')
        print(round(f1score, 4), ' (k: ', k, ')', sep='')
        if f1score > best_score:
            best_score = f1score
            best_k = k
    return best_score, best_k

def experimento_arvore(atributos, classe):
    '''Experimento com árvore de decisão'''
    prof_max = atributos.shape[1]
    lista_medidas = ['gini', 'entropy']
    best_score = float('-inf')
    best_medida = None
    best_profundidade = None
    print('Testando hipermarâmetros:')
    for medida in lista_medidas:
        for profundidade in range(2, prof_max+2):
            arvore = DecisionTreeClassifier(random_state=42,
                                            max_depth=profundidade,
                                            criterion=medida)
            previsoes = cross_val_predict(arvore, atributos,
                                          classe, cv=10)
            f1score = f1_score(classe, previsoes,
                               average='weighted')
            print(round(f1score, 4), '(Medida:', medida,
                  ', Profundidade:', profundidade, ')')
            if f1score > best_score:
                best_score = f1score
                best_medida = medida
                best_profundidade = profundidade
    return best_score, best_medida, best_profundidade
```

```
51
52  def principal():
53      '''Função principal'''
54      dados = datasets.load_breast_cancer()
55      atributos, classe = dados.data, dados.target
56      print('\nExperimento com KNN')
57      f1score_knn, k = experimento_knn(atributos, classe)
58      print('\nExperimento com árvore de decisão')
59      f1score_arvore, medida, profundidade = \
60          experimento_arvore(atributos, classe)
61      print('\nMelhor resultado para KNN:')
62      print('K:', k)
63      print('F1-score:', round(f1score_knn, 4))
64      print('\nMelhor resultado para árvore de decisão:')
65      print('Medida:', medida)
66      print('Profundidade:', profundidade)
67      print('F1-score:', round(f1score_arvore, 4))
68
69  if __name__ == '__main__':
70      principal()
```

B) Classificador de flores íris

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''Classificador de flores iris'''

import os
import pickle
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier

ARQUIVO = 'arvore.pickle'

def linha(caractere='-'):
    '''Escreve linha de caracteres'''
    print(caractere * 50)

def cria_arvore():
    '''Cria árvore e salva em arquivo'''
    # Carrega dados
    dados = datasets.load_iris()
    atributos, classe = dados.data, dados.target
    # Cria e treina árvore
    arvore = DecisionTreeClassifier(random_state=42, max_depth=3)
    arvore.fit(atributos, classe)
    # Atribui nomes das classes na árvore
    arvore.classes_ = dados.target_names
    # Salva árvore em arquivo
    pickle.dump(arvore, open(ARQUIVO, 'wb'))
    return arvore

def carrega_arvore():
    '''Carrega árvore salva em arquivo'''
    if os.path.isfile(ARQUIVO):
        # Carrega árvore já criada em arquivo
        return pickle.load(open(ARQUIVO, 'rb'))
    return cria_arvore()

def classifica(arvore):
    '''Classifica um registro'''
    print('Informe os dados')
    sepala_comp = float(input('Comprimento da sépala (cm): '))
    sepala_larg = float(input('Largura da sépala (cm): '))
    petala_comp = float(input('Comprimento da pétala (cm): '))
    petala_larg = float(input('Largura da pétala (cm): '))
    # Registro a ser classificado
    registro = [[sepala_comp, sepala_larg, petala_comp,
                 petala_larg]]
    # Classificação
    classe = arvore.predict(registro)[0]
    linha()
    print('\nClasse do registro:', classe)
    linha()
    input()
```

```
54
55  def principal():
56      '''Função principal'''
57      arvore = carrega_arvore()
58      while True:
59          linha('=')
60          print('Classificador de iris')
61          linha('=')
62          print('C) Classificar um flor')
63          print('S) Sair')
64          linha()
65          resp = input('Informe sua opção: ')
66          if len(resp) >= 1 and resp[0].lower() == 'c':
67              classifica(arvore)
68          if len(resp) >= 1 and resp[0].lower() == 's':
69              break
70
71  if __name__ == '__main__':
72      principal()
```

3.7 Revisão

Antes de prosseguirmos para a próxima o próximo conteúdo, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista aos vídeos “Revisão da Terceira Semana - Parte 1” e “Revisão da Terceira Semana - Parte 2”.

Nos encontramos na próxima semana.

Bons estudos!

Semana 4 - Agrupamento de dados

Objetivos
- Conhecer os conceitos básicos de agrupamento de dados;
- Desenvolver e entender códigos para agrupamento de dados.

4.1 Introdução

O agrupamento de dados, ou *clustering*, é uma tarefa de mineração de dados que tem como objetivo juntar objetos semelhantes em agrupamentos, também chamados de grupos ou *clusters*. O agrupamento de dados é uma tarefa de aprendizado não-supervisionado, já que os *clusters* representam classes que não estão definidas no início do processo de aprendizagem. A classificação, por outro lado, é uma tarefa de aprendizado supervisionado, uma vez que o conjunto de treinamento já possui os objetos previamente classificados em suas classes.

De posse dos agrupamentos dos dados, podemos tentar analisar os agrupamentos e atribuir classe aos mesmos. Outra análise interessante é analisar os pontos que estão distantes dos agrupamentos para detectar anomalias ou *outliers.* Quando trabalhamos com dados bi ou tridimensionais, podemos até plotar gráficos para visualizar os agrupamentos. No caso de dados com mais dimensões, a análise pode ser feita com ferramentas estatísticas que analisam a distribuição dos dados (SHALEV-SHWARTZ; BEN-DAVID, 2014).

Existem diversas categorias para as técnicas de agrupamento de dados. Neste livro vamos focar em algoritmos de agrupamento baseados em centroides, em densidade e algoritmos hierárquicos. Os algoritmos baseados em centroides utilizam elementos que representam a média dos elementos do grupo para representar o centroide daquele grupo. Os elementos são realocados entre os grupos até que os centroides ficam estabilizados.

Nos algoritmos de agrupamento baseados em densidade, os grupos são regiões densas (com alta concentração de pontos), separadas por regiões de baixa densidade (com poucos pontos). Os algoritmos hierárquicos trabalham com partições aninhadas de dados utilizando as abordagens aglomerativa ou divisiva. Na abordagem aglomerativa, o algoritmo inicia com um grupo para cada elemento da base de dados e, sucessivamente, vai juntando esses grupos. Na abordagem divisiva, o algoritmo começa com um único grupo maior contendo todos os elementos e o divide várias vezes para obter grupos menores.

4.2 Algoritmo k-means

O algoritmo k-means é uma técnica baseada em centroides que recebe como parâmetro o número de agrupamentos que devem ser construídos (**k**). O algoritmo começa com a seleção aleatória de **k** elementos da base de dados para serem os centroides iniciais. Cada elemento é associado ao centroide mais próximo. Em seguida, os elementos

são movidos de um grupo para outro com o objetivo de melhorar a qualidade dos agrupamentos. Um detalhe importante é que, dependendo da escolha do centroides iniciais, o algoritmo pode não obter um bom resultado. No entanto, a implementação do algoritmo disponível na biblioteca **sklearn** utiliza algumas estratégias para evitar esse tipo de problema.

A melhora da qualidade dos agrupamentos é feita com base nos erros quadráticos. O erro quadrático de um grupo é a soma das distâncias entre os elementos e o centroide do grupo. O cálculo do erro quadrático é dada pela fórmula da Equação (6), onde $\mathbf{x_i}$ é um elemento do grupo e **m** é o centroide do grupo contendo **n** elementos. Durante a execução, o algoritmo k-means tenta minimizar essa soma.

$$Erro\,Quadrático = \sum_{i=1}^{n} distância(x_i, m) \quad (6)$$

Durante a execução, o algoritmo k-means tenta minimizar a soma dos erros quadráticos de todos os grupos. A execução é interrompida quando essa soma estabiliza e não a mais mudanças de elementos entre grupos.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

def executa_kmeans(X, k):
    # Inicializa algoritmo k-means com 4 grupos
    kmeans = KMeans(n_clusters=k, random_state=42)
    # Executa o algoritmo e armazena o grupo de cada elemento y_grupos
    y_grupos = kmeans.fit_predict(X)
    # Plota os elementos em seus grupos
    plt.scatter(X[:, 0], X[:, 1], c=y_grupos)
    plt.show()

# Gera dados para agrupamento
X, _ = make_blobs(n_samples=200, random_state=42, centers=4)
executa_kmeans(X, 3)
executa_kmeans(X, 4)
```

Figura 110 – Exemplo de execução do algoritmo k-means com plotagem do resultado
Fonte: Elaborado pelo Autor.

A Figura 110 mostra um exemplo de execução do algoritmo k-means. Utilizamos a função **make_blobs()** para criar uma base de dados sintética com 4 (quatro) agrupamentos. Em seguida, chamamos a função **executa_kmeans()** para executar o algoritmo k-means considerando 3 grupos e, depois, considerando 4 grupos. Usamos a função **scatter()** do módulo **matplotlib.plyplot** para mostrar como ficam dispostos nos grupos.

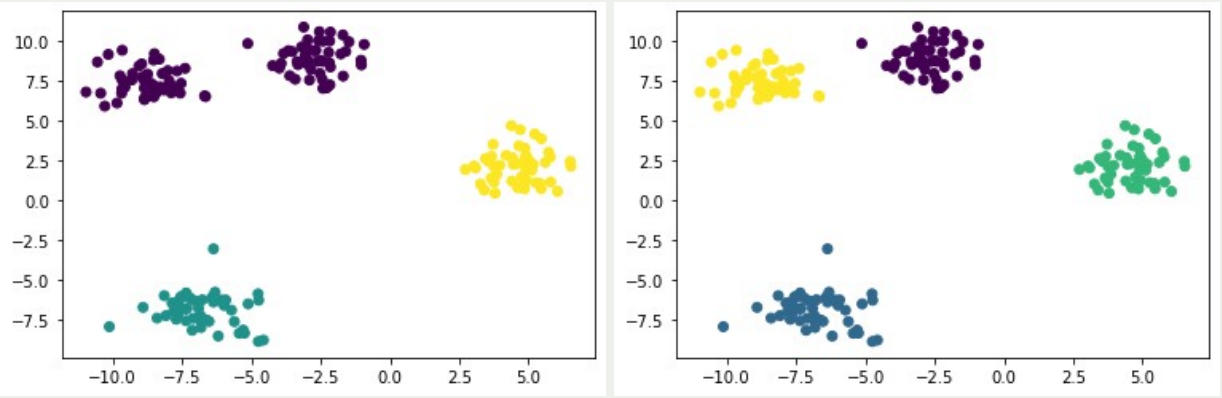


Figura 111 – Gráficos da execução do algoritmo k-means com 3 e 4 grupos
Fonte: Elaborado pelo Autor.

A Figura 111 apresenta os gráficos com o resultado das duas execuções do algoritmo k-means. Os elementos com a mesma cor estão no mesmo grupo. Podemos observar visualmente que o conjunto de dados possui quatro grupos. Assim, a execução do k-means considerando 3 grupos não faz um agrupamento correto. Essa é uma das desvantagens do k-means, temos que informar o número de grupos que devem ser encontrados.

Uma das maneiras de definirmos o número de grupos para um conjunto de dados é por meio do método cotovelo, também chamado de método joelho ou *elbow method*. A ideia é executar o k-means com diferentes quantidades de grupos para para definir qual é a melhor. Quando aumentamos o número de grupos, as diferenças nas distâncias dos membros de um grupo para seu centro diminuem, até chegar a zero quando cada agrupamento possuir um único elemento.

Quando executamos o algoritmo k-means da biblioteca **sklearn**, a soma das distâncias dos membros dos grupos, chamada de inércia, fica guardada no atributo **inertia_**. Podemos plotar uma curva com o valor da inércia para cada quantidade de grupos. Se traçarmos uma reta entre as extremidades dessa curva, a quantidade de grupos ideal será o ponto mais distante da reta.

O código da Figura 112 utiliza a classe **KneeLocator** da biblioteca **kneed** para encontrar encontrar o ponto de cotovelo para a nossa base de dados. Na criação do objeto **knee**, devemos informar os parâmetros **curve='convex'** e **direction='decreasing'** por se tratar de uma curva convexa decrescente. A Figura 113 mostra o gráfico gerado com a execução do código. Para nosso exemplo, o número ideal de grupos encontrado foi 4 (quatro).

```python
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from kneed import KneeLocator

# Cria dados sintéticos
X, _ = make_blobs(n_samples=200, random_state=42, centers=4)
# lista_x com número de grupos (k) e lista_y com as inércias
lista_x = []
lista_y = []
# Laço variando k
for k in range(2, 11):
    # Executa kmeans para k atual
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    # Adiciona inércia e k às listas
    lista_x.append(k)
    lista_y.append(kmeans.inertia_)
# Localiza cotovelo
knee = KneeLocator(lista_x, lista_y, curve='convex',
                   direction='decreasing')
knee.plot_knee()
melhor_k = lista_x[knee.knee-2]
print('Melhor k:', melhor_k)
```

Figura 112 – Código para cálculo de distância, cálculo do ponto de cotovelo e plotagem do gráfico
Fonte: Elaborado pelo Autor.

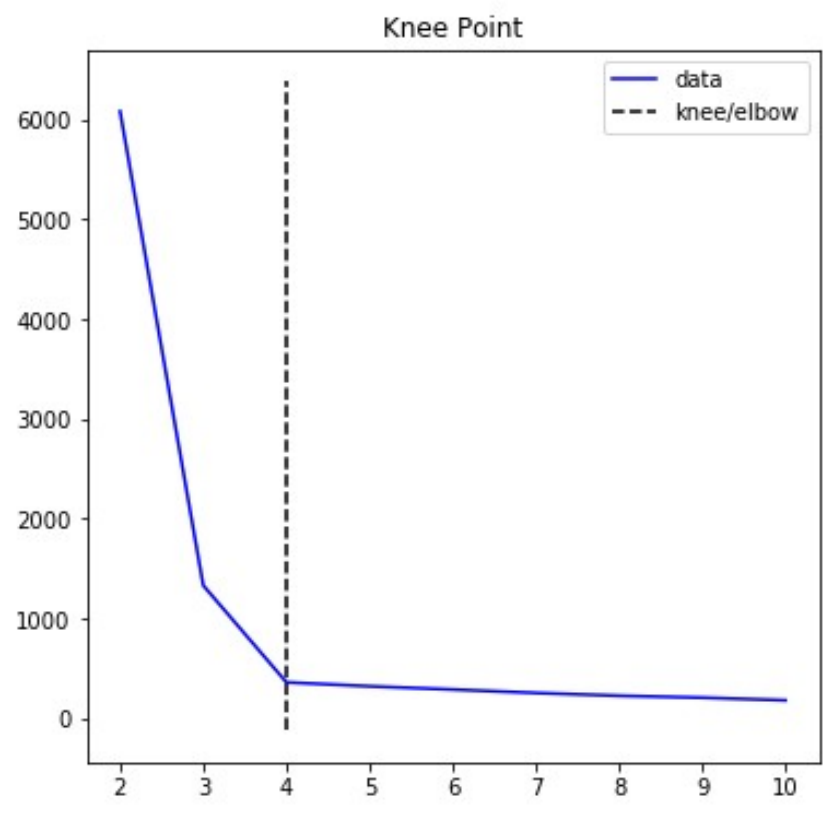


Figura 113 – Gráficos do ponto de cotovelo
Fonte: Elaborado pelo Autor.

4.3 Algoritmo DBSCAN

O algoritmo k-means funciona bem para grupos esféricos e bem formados, mas pode ter problemas com agrupamentos em outros tipos de formatos. Como exemplo, considere o código da Figura 114 que usa o k-means para agrupar dados em formato de lua. Podemos ver no resultado exibido na Figura 115 que, mesmo informando o número correto de agrupamentos, o k-means não funciona muito bem. Para esse tipo de conjunto de dados, é mais interessante utilizar outro tipo de algoritmo como o k-means.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn import datasets

X, _ = datasets.make_moons(n_samples=500, noise=0.1,
                           random_state=42)
kmeans = KMeans(n_clusters=2, random_state=42)
Y = kmeans.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=Y)
plt.show()
```

Figura 114 – Código para executar k-means sobre dados em formato de lua
Fonte: Elaborado pelo Autor.

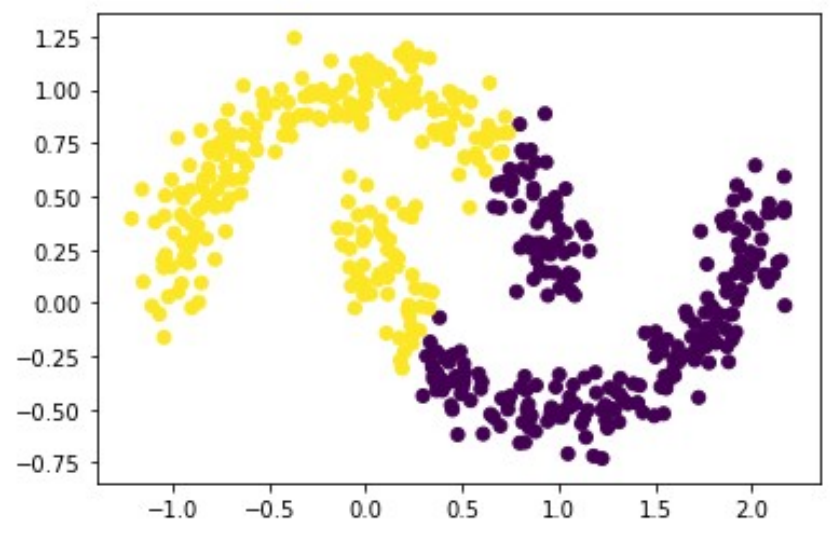


Figura 115 – Execução do k-means sobre dados em formato de lua
Fonte: Elaborado pelo Autor.

O algoritmo DBSCAN é um algoritmo de agrupamento de dados baseado em densidade, seu nome é a sigla de *density-based spatial clustering of applications with noise* que significa "agrupamento espacial baseado em densidade de aplicações com ruído". Basicamente, os agrupamentos são gerados com base em um número mínimo de vizinhos dentro da região delimitada por uma distância máxima entre pontos.

A Figura 116 demonstra como os agrupamentos são formados pelo algoritmo DBSCAN. Os pontos vermelhos são chamados de pontos centrais porque sua vinhança possui o número mínimo de vizinhos (quatro em nosso exemplo). Os pontos amarelos são chamados de pontos de borda. Apesar de não possuírem o número mínimo de vizinhos, os pontos de borda estão dentro da vizinhança de algum ponto central. O ponto azul é um

ruído porque não possui a vizinhança mínima e não está dentro da vizinhança de um ponto central. Na figura, a região contendo os pontos vermelhos e amarelos é considerada uma redição de alta densidade.

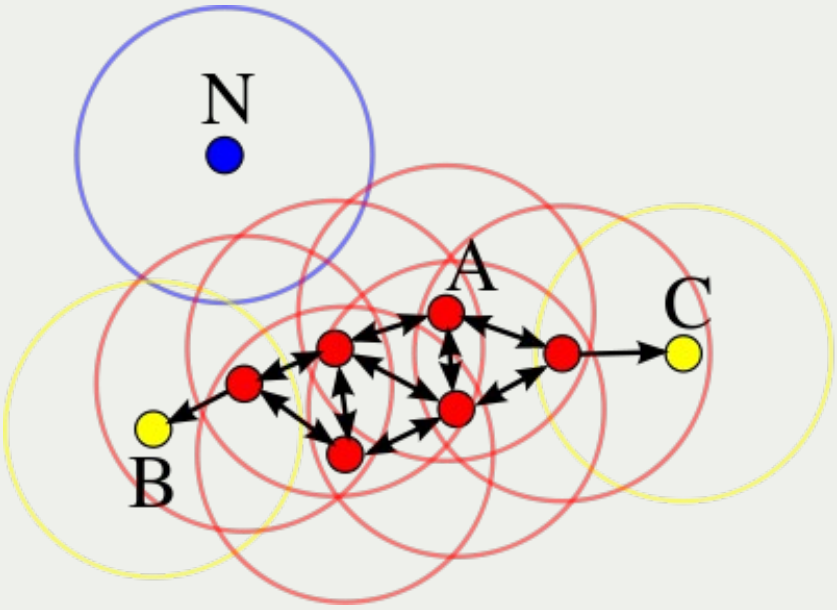


Figura 116 – Agrupamento feito pelo DBSCAN
Fonte: (WIKIPÉDIA, 2022b)

O código da Figura 117 mostra uma tentativa de execução do DBSCAN para agrupar dados em formato de lua. Contudo, como mostrado na Figura 118, o algoritmo não consegue um bom resultado usando os valores padrões para seus parâmetros **eps=0.5** e **min_samples=5**. O parâmetro **eps** é a distância máxima entre vizinhos e o parâmetro **min_samples** o número mínimo de pontos para a vizinhança a ser considerada densa.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans, DBSCAN
from sklearn import datasets

X, _ = datasets.make_moons(n_samples=500, noise=0.1,
                           random_state=42)
dbscan = DBSCAN()
Y = dbscan.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=Y)
plt.show()
```

Figura 117 – Código para executar DBSCAN sobre dados em formato de lua
Fonte: Elaborado pelo Autor.

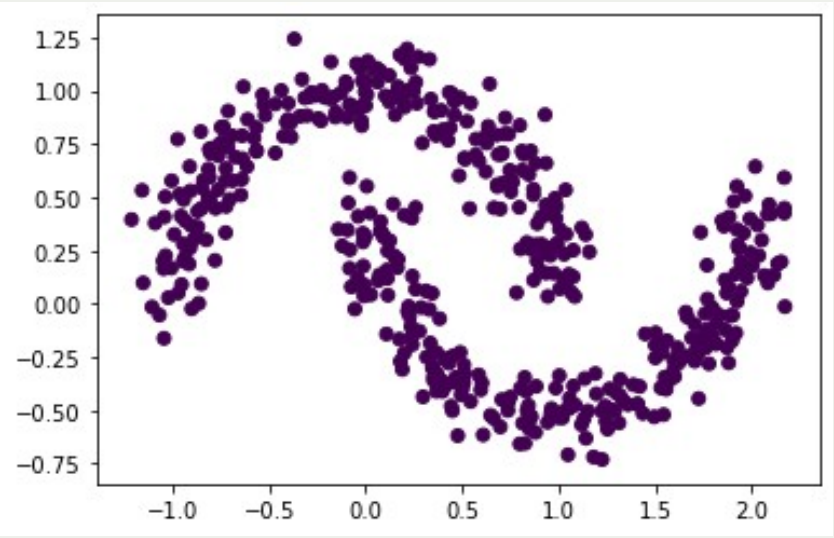


Figura 118 – Execução do DBSCAN sobre dados em formato de lua
Fonte: Elaborado pelo Autor.

A definição dos melhores valores para os parâmetros **eps** e **min_samples** não é uma tarefa muito simples. Contudo, existem algumas técnicas que podem ajudar. Para encontrar um **eps** interessante, podemos utilizar como limites a distância mínima e distância máxima para os k vizinhos mais próximos e, depois, testar valores dentro desses limites. O código da Figura 119 mostra como podemos gerar uma lista de possíveis valores de **eps** para teste.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score
from sklearn.cluster import DBSCAN
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors

def gera_lista_eps(X, min_k, max_k):
    # Calcula distâncias para os vizinhos mais próximos
    knn = NearestNeighbors(max_k).fit(X)
    lista_dist, _ = knn.kneighbors(X)
    # Considera apenas distâncias maiores do que zero
    lista_dist = lista_dist[lista_dist > 0]
    # Distâncias mínima e máxima
    min_dist, max_dist = np.min(lista_dist), np.max(lista_dist)
    # Retorna faixa de valores entre as distâncias mínima e máxima
    return np.linspace(min_dist, max_dist,
                       num=X.shape[0]//min_k)
```

Figura 119 – Código para gerar os possíveis valores de eps
Fonte: Elaborado pelo Autor.

A técnica proposta depende fortemente do número de **k** vizinhos mais próximos considerados. Em nossos experimentos vamos considerar um mínimo (**min_k**) de **L+1** e um máximo (**max_k**) de **(L+1)*3** vizinhos, onde **L** é o número de atributos da base de dados. Em nosso código, utilizamos a classe **NearestNeighbors** do módulo **neighbors** para calcular as distâncias de cada ponto para o maior número de vizinhos (**max_k**). As distâncias iguais a zero são ignoradas. No final, usamos a menor e a maior de distância

para gerar uma lista de possíveis valores de **eps** com a função **linspace()** da biblioteca **numpy**. A quantidade de valores da lista é calculada dividindo o número de registros da base de dados (**X.shape[0]**) pelo número mínimo de vizinhos (**min_k**). Os valores da lista são divididos em intervalos iguais.

Após gerarmos uma lista de valores de **eps**, devemos fazer um experimento para verificar qual desses valores proporciona um agrupamento mais interessante dos dados. Em nossos, experimentos vamos avaliar a qualidade dos agrupamentos com a medida de silhueta. O valor da silhueta representa o quão semelhante um objeto é ao seu próprio agrupamento em comparação com outros agrupamentos. O valor da silhueta varia de -1 a +1, sendo que os valores mais altos indicam que o agrupamento está com melhor qualidade. Uma observação importante é que a silhueta só pode ser calculada quando temos mais de um agrupamento de dados.

A Figura 120 apresenta a função **busca_eps()** que procura pelo melhor valor para eps considerando a medida da silhueta. Observe que, no algoritmo **DBSCAN**, variamos o valor de **eps** e mantemos **min_samples=max_k**. Valores maiores de **min_simples**, em geral, são melhores quando estamos procurando o melhor para **eps**.

```
def busca_eps(X):
    # Valores mínimos e máximos para k
    min_k, max_k = X.shape[1] + 1, (X.shape[1] + 1) * 3
    # Gera faixa de valores de eps para teste
    lista_eps = gera_lista_eps(X, min_k, max_k)
    # Melhores silhueta e eps
    melhor_sil = float('-inf')
    melhor_eps = np.mean(lista_eps)
    for eps_atual in lista_eps:
        # Executa DBSCAN para cada valor
        dbscan = DBSCAN(eps=eps_atual, min_samples=max_k)
        pred = dbscan.fit_predict(X)
        # Testa encontrou mais de um agrupamento
        if len(np.unique(pred)) > 1:
            # Calcula silhueta
            sil_atual = silhouette_score(X, pred)
            # Verifica se a silhueta mellhorou
            if sil_atual > melhor_sil:
                melhor_sil = sil_atual
                melhor_eps = eps_atual
    # Retorna eps relacionado com melhor silhueta
    return melhor_eps
```

Figura 120 – Função **busca_eps()**
Fonte: Elaborado pelo Autor.

Depois de encontrarmos o melhor valor para **eps**, podemos realizar outro experimento para tentar descobrir o melhor valor para o parâmetro **min_samples**. A Figura 121 mostra o código da função **busca_k()** responsável por esse experimento. Observe que, agora, utilizamos o valor de **(L+1)*2** para número máximo de vizinho. Normalmente, no experimento de busca pelo melhor valor de **min_samples**, não é recomendável utilizar valores maiores do que esse. O resultado da execução do código é mostrado na Figura 122. Assim, obtermos o valor de aproximado de **0.119** para o **eps** e o valor de **3** para o **min_samples**.

```
def busca_k(X, eps):
    # Valores mínimos e máximos para k
    min_k, max_k = X.shape[1] + 1, (X.shape[1] + 1) * 2
    # Melhores silhueta e k
    melhor_sil = float('-inf')
    melhor_k = int((min_k + max_k) / 2)
    for k_atual in range(min_k, max_k+1):
        # Executa DBSCAN variando k
        dbscan = DBSCAN(eps=eps, min_samples=k_atual)
        pred = dbscan.fit_predict(X)
        if len(np.unique(pred)) > 1:
            sil_atual = silhouette_score(X, pred)
            # Atualiza silhueta e k
            if sil_atual > melhor_sil:
                melhor_sil = sil_atual
                melhor_k = k_atual
    return melhor_k

X, _ = datasets.make_moons(n_samples=500, noise=0.1,
                           random_state=42)
melhor_eps = busca_eps(X)
melhor_k = busca_k(X, melhor_eps)
print('Melhor eps:', melhor_eps)
print('Melhor k:', melhor_k)
```

Figura 121 – Função **busca_k()** e chamada para cálculo dos valores
Fonte: Elaborado pelo Autor.

```
Melhor eps: 0.119
Melhor k: 3
```

Figura 122 – Resultado da execução do código para busca de **eps** e **min_samples**
Fonte: Elaborado pelo Autor.

Com a obtenção dos melhores valores para os parâmetros do algoritmo, podemos fazer uma nova execução e analisar o gráfico da plotagem dos grupos. A Figura 123 mostra o código para fazer essa execução.

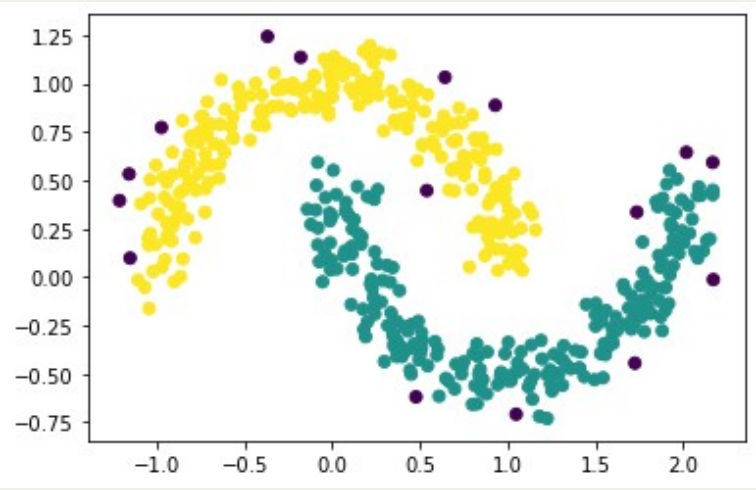
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import datasets

X, _ = datasets.make_moons(n_samples=500, noise=0.1,
                           random_state=42)
dbscan = DBSCAN(eps=0.119, min_samples=3)
Y = dbscan.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=Y)
plt.show()

grupos = np.unique(Y)
grupos = grupos[grupos != -1]
ruidos = Y[Y == -1]
print('Agrupamentos: ', len(grupos))
print('Ruídos:', len(ruidos))
```

Figura 123 – Código para execução do **DBSCAN** com parâmetros otimizados
Fonte: Elaborado pelo Autor.

A Figura 124 mostra o resultado da execução do **DBSCAN** com os parâmetros otimizados. Podemos notar que, dessa vez, o algoritmo conseguiu agrupar melhor os dois agrupamentos em formato de lua. Os agrupamentos encontrados estão coloridos em amarelo e verde. Os pontos roxos são os ruídos, ou seja, aqueles pontos que não foram incluídos em nenhum agrupamento.




```
Agrupamentos:  2
Ruídos: 16
```

Figura 124 – Execução do **DBSCAN** com parâmetros otimizados
Fonte: Elaborado pelo Autor.

É importante mencionar que, mesmo com os valores encontramos para os parâmetros nos experimentos executados, ainda podemos fazer ajustes nos mesmos para tentar melhorar o resultado. Com um valor de **eps** igual **0.2**, por exemplo, conseguimos dois agrupamentos sem nenhum ruído. Esses ajustes manuais dependem de um bom conhecimento da base de dados e análise dos agrupamentos encontrados.

4.4 Algoritmos hierárquicos

Conforme já mencionamos, os algoritmos hierárquicos podem usar abordagens aglomerativa ou divisiva. Na prática, a abordagem aglomerativa é mais utilizada, então vamos focar nessa metodologia utilizando a classe **AgglomerativeClustering** do módulo **sklearn.cluster**.

Os principais parâmetros do algoritmo **AgglomerativeClustering** são o número de grupos (**n_clusters**) e o critério de ligação (**linkage**). O critério de ligação determina qual distância usar entre grupos com possibilidade de junção. O algoritmo irá mesclar os pares de agrupamento que minimizem este critério. Os valores mais interessantes para o parâmetro **linkage** são **'single'** (simples) e **'ward'** (similaridade).

No critério de ligação simples, a distância é definida como a distância mínima entre um objeto de um agrupamento e um objeto no outro agrupamento. O critério de similaridade tem como base a variância dos dois agrupamentos. Em vez de medir a distância diretamente, ele analisa a distância entre os pontos antes e depois da fusão. A ligação simples é indicada para manipular formas não elípticas, mas é bastante sensível a ruídos. Já a ligação por similaridade é menos prejudicada por ruídos e funciona melhor para agrupamentos esféricos e elípticos.

```
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn import datasets

def executa_aglomerativo(X):
    # Executa algoritmo
    agg = AgglomerativeClustering()
    y_grupos = agg.fit_predict(X)
    # Plota resultado
    plt.scatter(X[:, 0], X[:, 1], c=y_grupos)
    plt.show()

# Cria bases de dados
X1, _ = datasets.make_moons(n_samples=500, noise=0.1,
                            random_state=42)
X2, _ = datasets.make_blobs(n_samples=200, random_state=42,
                            centers=4)
X3, _ = datasets.make_blobs(n_samples=1500, random_state=1,
                            centers=5, cluster_std=0.6)
# Executa o algoritmo para cada base de dados
for cont, X in enumerate([X1, X2, X3]):
    executa_aglomerativo(X)
    print('X', cont+1, sep='')
```

Figura 125 – Execução do **AgglomerativeClustering** com valor padrão nos parâmetros
Fonte: Elaborado pelo Autor.

O algoritmo **AgglomerativeClustering** usa por padrão **n_clusters=2** e **linkage='ward'**. O código da Figura 125 mostra a execução do algoritmo usando os parâmetros com valor padrão sobre três bases de dados distintas. A Figura 126 mostra o resultado da execução do código. Podemos observar que o algoritmo não obteve um bom resultado para nenhuma das três bases de dados, pois o número de agrupamentos

(n_**clusters**) não foi adequado para as bases **X2** e **X3** e o critério de ligação (**linkage**) não foi adequado para a base de dados **X1**.

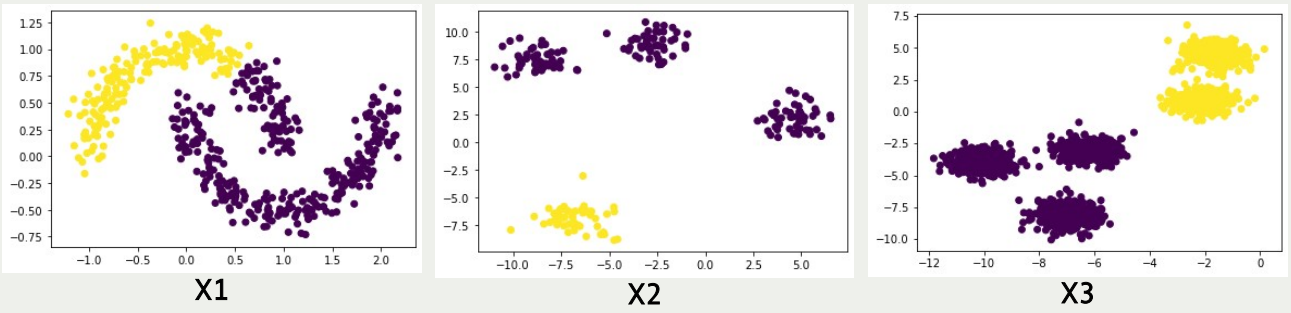


Figura 126 – Resultado da execução do **AgglomerativeClustering** com valor padrão nos parâmetros
Fonte: Elaborado pelo Autor.

Para melhorar o resultado do algoritmo vamos utilizar novamente a medida da silhueta, mas, dessa vez, temos que analisar o número de agrupamentos e o critério de ligação. Inicialmente, variar o critério de ligação e número de agrupamento para calcular a silhueta para cada variação. A Figura 127 exibe o código para realizar essa tarefa.

```
import numpy as np
from sklearn.metrics import silhouette_score
from sklearn.cluster import AgglomerativeClustering
from sklearn import datasets

def lista_silhueta(X, max_k):
    for link_atual in ['single', 'ward']:
        for k_atual in range(2, max_k+1):
            agg = AgglomerativeClustering(n_clusters=k_atual,
                                          linkage=link_atual)
            pred = agg.fit_predict(X)
            if len(np.unique(pred)) > 1:
                sil_atual = silhouette_score(X, pred)
            else:
                sil_atual = float('nan')
            print(link_atual, ', ' , k_atual, ', ', sil_atual, sep='')

# Cria bases de dados
X1, _ = datasets.make_moons(n_samples=500, noise=0.1,
                            random_state=42)
X2, _ = datasets.make_blobs(n_samples=200, random_state=42,
                            centers=4)
X3, _ = datasets.make_blobs(n_samples=1500, random_state=1,
                            centers=5, cluster_std=0.6)
# Executa o algoritmo para cada base de dados
for cont, X in enumerate([X1, X2, X3]):
    lista_silhueta(X, 6)
    print('X', cont+1, sep='')
```

Figura 127 – Código para cálculo da silhueta variando número de grupos e critério de ligação
Fonte: Elaborado pelo Autor.

O resultado da execução do código para calcular a silhueta variando o número de grupos e o critério de ligação pode ser visto da Figura 128. Para cada critério de ligação, destacamos o melhor valor de silhueta encontrado. Na base **X3**, o maior valor de silhueta

corresponde aos melhores valores para o critério de ligação e número de grupos. No caso da base **X2**, tivemos o mesmo valor de silhueta para os critérios de ligação **'single'** e **'ward'**. Como os agrupamentos estão bem separados nessa base, os dois critérios funcionam bem.

A medida da silhueta não funciona muito bem para agrupamentos em formatos não esféricos como acontece na base de dados **X1**. Assim, todas as silhuetas para o critério de ligação **'ward'** são maiores do que no **'single'**. Contudo, o algoritmo **AgglomerativeClustering** não funciona muito bem em agrupamentos não esféricos utilizando tal critério de ligação. Portanto, simplesmente considerar o maior valor de silhueta não é o ideal para agrupamentos não esféricos.

X1	X2	X3
single, 2, 0.327	single, 2, 0.597	single, 2, 0.679
single, 3, 0.075	single, 3, 0.76	single, 3, 0.651
single, 4, -0.193	single, 4, 0.795	single, 4, 0.492
single, 5, -0.217	single, 5, 0.717	single, 5, 0.344
single, 6, -0.224	single, 6, 0.692	single, 6, 0.511
ward, 2, 0.443	ward, 2, 0.597	ward, 2, 0.679
ward, 3, 0.392	ward, 3, 0.76	ward, 3, 0.651
ward, 4, 0.411	ward, 4, 0.795	ward, 4, 0.704
ward, 5, 0.408	ward, 5, 0.656	ward, 5, 0.732
ward, 6, 0.504	ward, 6, 0.532	ward, 6, 0.641

Figura 128 – Resultado da código para cálculo da silhueta variando número de grupos e critério de ligação
Fonte: Elaborado pelo Autor.

Se observarmos novamente os valores de silhueta na Figura 128 para a base de dados **X1**, percebemos que temos alguns valores negativos e outros positivos no critério de ligação **'single'**. Os valores negativos indicam que o resultado do agrupamento está muito ruim. Porém, se temos valores negativos e positivos nesse critério de ligação, é provável que estejamos lidando com agrupamentos não esféricos e pode ser melhor manter o critério de ligação **'single'**.

Para tentar automatizar a tarefa de seleção de parâmetros para o algoritmo **AgglomerativeClustering**, vamos, inicialmente, criar uma função que calcula a silhueta considerando um critério de ligação e variando o número de grupos. A função irá retornar o número de grupos com maior valor de silhueta e também a contagem de valores de silhueta negativos. A Figura 129 mostra o código com a importação das bibliotecas e a função **busca()** para para buscar número de agrupamentos considerando um critério de ligação.

O código exibe também a função **busca_k_link()** que chama a função **busca()** e decide pelo melhor critério de ligação. O resultado retornado será o número de agrupamentos e o critério de ligação ideais. A última função apresentada no código é a **executa_agg()** que recebe uma base de dados e os parâmetros para o algoritmo **AgglomerativeClustering**. A função executa o algoritmo e plota um gráfico com o resultado do agrupamento.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score
from sklearn.cluster import AgglomerativeClustering
from sklearn import datasets

def busca_cont_neg(X, max_k, link):
    # Inicia melhor k com valor médio
    melhor_k = int((2 + max_k) / 2)
    # Contagem de silhuetas negativas
    cont_neg = 0
    melhor_sil = float('-inf')
    # Varia k (número de grupos)
    for k_atual in range(2, max_k+1):
        # Executa algoritmo de agrupamento
        agg = AgglomerativeClustering(n_clusters=k_atual,
                                      linkage=link)
        pred = agg.fit_predict(X)
        if len(np.unique(pred)) > 1:
            # Calcula silhueta
            sil_atual = silhouette_score(X, pred)
            # Contagem de silhuetas negativas
            if sil_atual < 0:
                cont_neg += 1
            # Atualiza melhores valores
            if sil_atual > melhor_sil:
                melhor_sil = sil_atual
                melhor_k = k_atual
    return melhor_k, melhor_sil, cont_neg

def busca_k_link(X, max_k):
    # Busca melhores valores (single)
    k_single, sil_single, neg_single = busca(X, max_k, 'single')
    # Testa se houveram silhuetas negativas
    if neg_single > 0 and sil_single > 0:
        return k_single, 'single'
    # Busca melhores valores (ward)
    k_ward, _, _ = busca(X, max_k, 'ward')
    return k_ward, 'ward'

def executa_agg(X, k, link):
    # Executa algoritmo de agrupamento
    agg = AgglomerativeClustering(n_clusters=k,
                                  linkage=link)
    y_grupos = agg.fit_predict(X)
    # Plota resultado
    plt.scatter(X[:, 0], X[:, 1], c=y_grupos)
    plt.show()
```

Figura 129 – Função para buscar número de agrupamentos considerando um critério de ligação
Fonte: Elaborado pelo Autor.

A Figura 130 mostra o restante do código para executar o experimento completo nas bases de dados **X1**, **X2** e **X3**. Após a geração dos dados, o laço de repetição chama a função **busca_k_link()** para encontrar os melhores valores de parâmetros. De posse desses valores, chamamos a função **executa_agg()** para execução do algoritmo. O

resultado da execução do código é apresentado na Figura 131. Podemos observar que, com a seleção dos parâmetros o algoritmo obteve um bom resultado para todas as bases de dados.

```
X1, _ = datasets.make_moons(n_samples=500, noise=0.1, random_state=42)
X2, _ = datasets.make_blobs(n_samples=200, random_state=42, centers=4)
X3, _ = datasets.make_blobs(n_samples=1500, random_state=1,
                            centers=5, cluster_std=0.6)

# Testa cada base de dados
for cont, X in enumerate([X1, X2, X3]):
    print('X', cont+1, sep='')
    melhor_k, melhor_link = busca_k_link(X, 10)
    print('Melhor link:', melhor_link)
    print('Melhor k:', melhor_k)
    executa_agg(X, melhor_k, melhor_link)
```

Figura 130 – Experimento para seleção de parâmetros do algoritmo **AgglomerativeClustering**
Fonte: Elaborado pelo Autor.

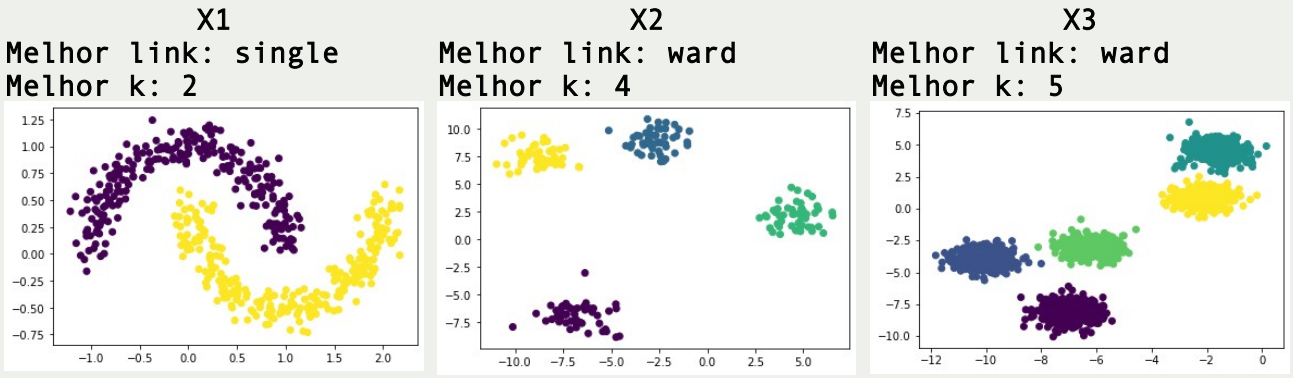


Figura 131 – Resultado do experimento para seleção de parâmetros do algoritmo **AgglomerativeClustering**
Fonte: Elaborado pelo Autor.

Atividade: Resolva os exercícios a seguir. As respostas estão nas próximas páginas, mas é importante que você tente resolver por conta própria e use as respostas apenas para conferência.

4.5 Exercício

```
from sklearn import datasets

X, _ = datasets.make_blobs(n_samples=1200, random_state=11,
                           centers=4,
                           cluster_std=[0.5, 0.7, 1.1, 1.3])
```

Figura 132 – Base sintética para exercício
Fonte: Elaborado pelo Autor.

Considere a base de dados criada pelo código da Figura 132. Escreva um código em Python para para realizar um experimento com essa base de dados usando os algoritmos **KMeans** e **AgglomerativeClustering**. Considere os seguintes pontos:

- Crie a função **busca_k(Algoritmo, X, max_k)** para executar o algoritmo de agrupamento sobre a base de dados **X** variando o número de grupos entre **2** e **max_k**. Para cada execução calcule a silhueta e retorne o número de grupos com maior silhueta;

- Implemente a função executa_k(Algoritmo, X, k) para executar o algoritmo de agrupamento sobre a base de dados X considerando o número de grupos igual a k. Após a execução, a função deve plotar o gráfico com o resultado do agrupamento;

- Faça um experimento com os algoritmos mencionados variando o número de grupos entre 2 e 10 para tentar descobrir o melhor número de grupos. Após o experimento, execute os algoritmos observando o resultado dos mesmos.

4.6 Resposta dos exercício

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score


def busca_k(Algoritmo, X, max_k):
    # Inicia melhor k com valor médio
    melhor_k = int((2 + max_k) / 2)
    melhor_sil = float('-inf')
    # Varia k (número de grupos)
    for k_atual in range(2, max_k+1):
        # Executa algoritmo de agrupamento
        agrupador = Algoritmo(n_clusters=k_atual)
        pred = agrupador.fit_predict(X)
        if len(np.unique(pred)) > 1:
            # Calcula silhueta
            sil_atual = silhouette_score(X, pred)
            # Atualiza melhores valores
            if sil_atual > melhor_sil:
                melhor_sil = sil_atual
                melhor_k = k_atual
    return melhor_k, melhor_sil


def executa_k(Algoritmo, X, k):
    # Executa algoritmo de agrupamento
    agrupador = Algoritmo(n_clusters=k)
    y_grupos = agrupador.fit_predict(X)
    # Plota resultado
    plt.scatter(X[:, 0], X[:, 1], c=y_grupos)
    plt.show()

# Cria dados sintéticos
X, _ = datasets.make_blobs(n_samples=1200, random_state=11,
                           centers=4,
                           cluster_std=[0.5, 0.7, 1.1, 1.3])

Algoritmo = KMeans
melhor_k, silhueta = busca_k(Algoritmo, X, 10)
executa_k(Algoritmo, X, melhor_k)
print('Silhueta:', round(silhueta, 3))

Algoritmo = AgglomerativeClustering
melhor_k, silhueta = busca_k(Algoritmo, X, 10)
executa_k(Algoritmo, X, melhor_k)
print('Silhueta:', round(silhueta, 3))
```

4.7 Revisão

Antes de finalizarmos o curso, é importante que você estude, revise o conteúdo e realize pesquisas sobre os conceitos apresentados para ampliar seus conhecimentos.

Mídia digital: Antes de avançarmos nos estudos, vá até a sala virtual e assista aos vídeos “Revisão da Quarta Semana - Parte 1” e “Revisão da Quarta Semana - Parte 2”.

Bons estudos!

Finalizando o curso

A prática é uma atividade fundamental para um bom aprendizado da lógica de programação e de qualquer linguagem de programação. Uma boa tarefa prática interessante é revisitar os problemas e exercícios estudados e tentar resolvê-los por conta própria. Além disso, você pode se aprofundar mais em diversos conteúdos usando as referencias citadas no livro.

Recomendamos também que você sempre busque pelo constante aprimoramento profissional. Você pode encontrar diversos conteúdos interessantes na Plataforma +IFMG (https://mais.ifmg.edu.br).

Atividade final

Atividade: Para concluir o curso e gerar o seu certificado, vá até a sala virtual e responda ao Questionário “Avaliação final”. Este teste é constituído por 10 perguntas de múltipla escolha, baseadas no conteúdo estudado.

Referências

ALKASEER, K. **Machine Learning and Big Data**: Intuitive ML and big data in C++, Scala, Java and Python. 2016. Disponível em: https://www.kareemalkaseer.com/books/ml

BORGES, L. E. **Python para Desenvolvedores**. 2. ed. Rio de Janeiro: Edição do Autor, 2010. Disponível em: https://ricardoduarte.github.io/python-para-desenvolvedores/

CEDER, N. The **Quick Python Book**. 3. ed. Shelter Island: Manning Publications, 2018. Disponível em: https://livebook.manning.com/book/the-quick-python-book-third-edition/

CORRÊA, E. **Meu primeiro livro de Python**. 2. ed. Rio de Janeiro: Edubd, 2020. Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python

CURRY, E. et al. (Ed.). **Technologies and Applications for Big Data Value**. Gewerbestrasse: Springer Nature, 2022. Disponível em: https://doi.org/10.1007/978-3-030-78307-5

DOWNEY, A. B. **Think Python**: How to Think Like a Computer Scientist. 2. ed. Needham: Green Tea Press, 2015. Disponível em: https://greenteapress.com/wp/think-python-2e/

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson, 2011.

IZBICKI, R.; SANTOS, T. M. dos. **Aprendizado de máquina**: uma abordagem estatística. São Carlos, 2020. Disponível em: http://www.rizbicki.ufscar.br/AME.pdf

KHANNA, R.; AWAD, M. **Efficient Learning Machines**. Berkeley: Apress, 2015. Disponível em: https://doi.org/10.1007/978-1-4302-5990-9

MARÇULA, M.; FILHO, P. A. B. **Informática**: Conceitos e Aplicações. 3. ed. São Paulo: Érica, 2008.

MENEZES, N. N. C. **Introdução à programação com Python**: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of Machine Learning**. 2. ed. Cambridge: MIT Press, 2018. Disponível em: https://cs.nyu.edu/~mohri/mlbook/

PILGRIM, M. **Dive Into Python 3**. New York: Apress, 2009. Disponível em: https://diveintopython3.net/

PYPL. **PYPL**: PopularitY of Programming Language. 2021. Disponível em: https://pypl.github.io/PYPL.html

PYTHON SOFTWARE FOUNDATION (PSF). **Python 3.10.1 documentation**. 2021. Disponível em: https://docs.python.org/

RAMALHO, L. **Python fluente**: programação clara, concisa e eficaz. São Paulo: Novatec, 2015.

SOTO, S. V.; LUNA, J. M.; CANO, A. (Ed.). **Big Data on Real-World Applications**. London: IntechOpen, 2016. Disponível em: https://doi.org/10.5772/61396

SHALEV-SHWARTZ, S.; BEN-DAVID, S. **Understanding machine learning**: From Theory to Algorithms. Cambridge: Cambridge University Press, 2014. 449 p. Disponível em: https://www.cs.huji.ac.il/w~shais/UnderstandingMachineLearning/

SWEIGART, A. **Beyond the basic stuff with python**: best practices for writing clean code. San Francisco: No Starch Press, 2021. Disponível em: https://inventwithpython.com/beyond/

TAGLIAFERRI, L. **How To Code in Python 3**. New York: DigitalOcean, 2018. Disponível em: https://assets.digitalocean.com/books/python/how-to-code-in-python.pdf

TIOBE. **TIOBE Index for December 2021**. 2021. Disponível em: https://www.tiobe.com/tiobe-index/

VELLOSO, F. **Informática**: conceitos básicos. 9. ed. Rio de Janeiro: Elsevier, 2014.

WIKIPÉDIA. **Algoritmo de Euclides**. 2021. Disponível em: https://pt.wikipedia.org/wiki/Algoritmo_de_Euclides

WIKIPÉDIA. **Big Data**. 2022. Disponível em: https://pt.wikipedia.org/wiki/Big_data

WIKIPÉDIA. **DBSCAN**. 2022. Disponível em: https://en.wikipedia.org/wiki/DBSCAN

WIKIPÉDIA. **Python**. 2022. Disponível em: https://pt.wikipedia.org/wiki/Python

Currículo do autor

Marcos Roberto Ribeiro

O autor possui graduação em Ciência da Computação pelo Centro Universitário de Formiga (2005), mestrado em Ciência da Computação pela Universidade Federal de Uberlândia (2008) e doutorado em Ciência da Computação pela Universidade Federal de Uberlândia (2018). Atua como professor em disciplinas de cursos técnicos e superiores do Instituto Federal Minas Gerais - Campus Bambuí desde 2010. As disciplinas ministradas estão relacionadas principalmente com Programação, Banco de Dados e Desenvolvimento de Sistemas.

Currículo Lattes: http://lattes.cnpq.br/8439091552425995

Glossário de códigos QR (Quick Response)

		Mídia digital Apresentação do curso				Mídia digital Revisão da primeira semana - parte 1
		Mídia digital Revisão da primeira semana - parte 2				Mídia digital Revisão da segunda semana - parte 1
		Mídia digital Revisão da segunda semana - parte 2				Mídia digital Revisão da terceira semana - parte 1
		Mídia digital Revisão da terceira semana - parte 2				Mídia digital Revisão da quarta semana - parte 1
		Mídia digital Revisão da quarta semana - parte 2				

Plataforma +IFMG

Formação Inicial e Continuada EaD

A Pró-Reitoria de Extensão (Proex), desde o ano de 2020, concentrou seus esforços na criação do Programa +IFMG. Esta iniciativa consiste em uma plataforma de cursos *online*, cujo objetivo, além de multiplicar o conhecimento institucional em Educação à Distância (EaD), é aumentar a abrangência social do IFMG, incentivando a qualificação profissional. Assim, o programa contribui para o IFMG cumprir seu papel na oferta de uma educação pública, de qualidade e cada vez mais acessível.

Para essa realização, a Proex constituiu uma equipe multidisciplinar, contando com especialistas em educação, *web design*, *design* instrucional, programação, revisão de texto, locução, produção e edição de vídeos e muito mais. Além disso, contamos com o apoio sinérgico de diversos setores institucionais e também com a imprescindível contribuição de muitos servidores (professores e técnico-administrativos) que trabalharam como autores dos materiais didáticos, compartilhando conhecimento em suas áreas de atuação.

A fim de assegurar a mais alta qualidade na produção destes cursos, a Proex adquiriu estúdios de EaD, equipados com câmeras de vídeo, microfones, sistemas de iluminação e isolação acústica, para todos os 18 *campi* do IFMG.

Somando à nossa plataforma de cursos *online*, o Programa +IFMG disponibilizará também, para toda a comunidade, uma Rádio *Web* Educativa, um aplicativo móvel para Android e iOS, um canal no Youtube com a finalidade de promover a divulgação cultural e científica e cursos preparatórios para nosso processo seletivo, bem como para o Enem, considerando os saberes contemplados por todos os nossos cursos.

Parafraseando Freire, acreditamos que a educação muda as pessoas e estas, por sua vez, transformam o mundo. Foi assim que o +IFMG foi criado.

O +IFMG significa um IFMG cada vez mais perto de você!

Professor Carlos Bernardes Rosa Jr.
Pró-Reitor de Extensão do IFMG

Características deste livro:
Formato: A4
Tipologia: Arial e Capriola.
E-book:
1ª. Edição
Formato digital