INTELIGÊNCIA ARTIFICIAL APLICADA - MACHINE LEARNING

AULA 2

Prof. Antonio Willian Sousa

CONVERSA INICIAL

Nesta aula, apresentaremos os formatos de intercâmbio de dados comumente utilizados, as bibliotecas de cálculo numérico para lidar com os problemas de aprendizagem de máquina, como trabalhar com dados tabulares e visualizá-los. Também falaremos sobre as bases de dados utilizadas para aprendizagem de máquina e a sua importância.

TEMA 1 – FORMATOS DE INTERCÂMBIO DE DADOS

Nesta seção, apresentaremos os formatos de intercâmbio de dados mais comuns na área de aprendizagem de máquina. Abordaremos a sua organização interna e como podemos trabalhar com eles em Python. Também serão apresentados outros formatos que podem ser utilizados e como podemos lidar com eles.

Os dados utilizados em aprendizagem de máquina são comumente referenciados como *datasets* (“base de dados” ou “conjunto de dados”). Esses dados podem se apresentar das mais diferentes formas, desde arquivo em formato texto puro até estruturas de bancos de dados mais complexos, estruturados e não estruturados. Eles podem ou não apresentar os dados de forma completa, fazendo com que dentro dos processos de preparação dos dados para entregar aos processos de aprendizagem a avaliação da qualidade dos dados seja um dos passos mais importantes. Como citamos anteriormente, a qualidade dos seus modelos está intrinsecamente relacionada à qualidade dos dados que lhe são entregues.

Os *datasets* comumente virão organizados ou armazenados em uma das seguintes formas:

- arquivos texto puro;
- arquivos de planilha;
- bases de dados estruturadas (SQL);
- bases de dados não estruturadas (NoSQL);
- dados multimídia (imagens, vídeos, áudio etc.);
- arquivos serializados.

As diferentes maneiras como os dados são armazenados e apresentam algo em comum é pelo fato de, praticamente, todas as aplicações de preparação,

importação, exportação e manipulação apresentarem métodos para conversão e importação de dados em formatos textuais. Um dos principais motivos para isso é a facilidade de lidar com arquivos textos em diferentes plataformas e o fato de arquivos e textos apresentarem alto grau de compressão quando submetidos a algoritmos de compactação. Dentre os formatos de arquivos textuais, os arquivos do tipo *Comma-separated values* (CSV) figuram como um dos mais utilizados.

1.1 Arquivos CSV

O formato CSV geralmente é utilizado para armazenamento de dados em forma de tabela. Na Figura **Erro! Fonte de referência não encontrada.**, podemos ver um exemplo de como os dados são teoricamente organizados. No entanto, essa representação não é fiel à forma como os dados realmente estão armazenados, com a forma de armazenamento mais próxima do real sendo algo semelhante à Figura **Erro! Fonte de referência não encontrada.**.

Figura 1 – Exemplo de dados tabulares em formato texto

```
idade,nacionalidade,altura,peso,sexo,gosta de futebol,salario
22,0,1.7,75,0,1,2500
52,1,1.75,80,0,1,2500
31,1,1.5,65,1,1,4500
65,1,1.95,86,0,0,6500
17,0,1.81,95,0,1,1500
54,0,1.65,80,1,0,3500
30,0,1.9,105,1,1,1200
25,1,1.85,65,1,1,2245
49,0,1.71,75,0,0,25000
26,1,1.81,80,1,1,8000
```

Fonte: Sousa, 2020.

Figura 2 – Exemplo de dados tabulares armazenados em CSV

idade	nacionalidade	altura	peso	sexo	gosta de futebol	salario
22	0	1.70	75	0	1	2500
52	1	1.75	80	0	1	2500
31	1	1.50	65	1	1	4500
65	1	1.95	86	0	0	6500
17	0	1.81	95	0	1	1500
54	0	1.65	80	1	0	3500
30	0	1.90	105	1	1	1200
25	1	1.85	65	1	1	2245
49	0	1.71	75	0	0	25000
26	1	1.81	80	1	1	8000

Fonte: Sousa, 2020.

Quando trabalhamos com arquivos do tipo CSV, devemos nos atentar para algumas particularidades, como a presença dos cabeçalhos de cada coluna no próprio arquivo, sendo preciso armazenar essa informação inicialmente para saber a que se refere cada coluna do arquivo. Outra característica desses arquivos são o fato de os arquivos CSV utilizarem o caractere vírgula (,) como separador padrão. Assim, quando o valor de alguma coluna apresentar um caractere igual ao separador, é necessário indicar uma forma de delimitar cada campo. Comumente se utiliza o caractere aspas duplas (“) como delimitador. Ainda que esses arquivos utilizem a vírgula como seu separador padrão, atualmente o termo se refere a todo um conjunto de arquivos que utilizem um caractere como separador dos campos e um outro caractere como delimitador.

Um tipo de arquivo comum, semelhante ao CSV, é padrão TSV (*tab-separated values*) que utiliza a tabulação como o caractere de separação. O uso de diferentes separadores pode ser útil em situações nas quais é necessário utilizar determinado caractere como separador. No caso do Brasil, o nosso separador decimal é a vírgula, assim o número 1,70, ao ser representado em um arquivo CSV, deveria vir armazenado em uma das formas indicadas no Quadro 1 a seguir.

Quadro 1 – Formas de armazenamento de numeral decimal representado em um arquivo CSV

formato: **csv** separador: **,** delimitador: **”**	“22”,”0”,”**1,70**”,”75”
formato: **tsv** separador: **tabulação**	220**1,70**75

Fonte: Sousa, 2020.

A linguagem Python possui um módulo nativo que permite a manipulação de arquivos em formato CSV e outros similares. O seguinte trecho de código mostra como abrir um arquivo com o nome dados_clientes.csv em formato CSV por meio de objeto que lê o arquivo linha por linha e imprime os dados nele armazenados:

```
import csv
with open("dados_clientes.csv",delimiter=',') as f:
  data = csv.reader(f)
  #Itera as linhas do arquivo
  for linha in data:
    print(str(linha))
```

Isso gera o seguinte resultado:

```
['idade', 'nacionalidade', 'altura', 'peso', 'sexo', 'gosta de futebol', 'salario']
['22', '0', '1.7', '75', '0', '1', '2500']
['52', '1', '1.75', '80', '0', '1', '2500']
['31', '1', '1.5', '65', '1', '1', '4500']
['65', '1', '1.95', '86', '0', '0', '6500']
['17', '0', '1.81', '95', '0', '1', '1500']
['54', '0', '1.65', '80', '1', '0', '3500']
['30', '0', '1.9', '105', '1', '1', '1200']
['25', '1', '1.85', '65', '1', '1', '2245']
['49', '0', '1.71', '75', '0', '0', '25000']
['26', '1', '1.81', '80', '1', '1', '8000']
```

Para manipular arquivos com outros separadores e delimitadores, é possível especificar os caracteres que se deseja utilizar. Esses formatos de arquivos podem facilmente ser utilizados tanto para exportação quanto importação de dados. Comumente, esse formato é utilizado para bases de dados relacionais ou dados tabulares como planilhas. Para outras formas de estruturação de dados, o formato JavaScript Object Notation (JSON) pode ser utilizado, sendo considerado um padrão mais adequado para esse tipo de operação.

1.2 Arquivos JSON

JSON é um formato leve para intercâmbio de informações inspirado na sintaxe da linguagem JavaScript, mas atualmente se trata de um formato independente de linguagem com a sua evolução desvinculada da linguagem JavaScript. Uma das principais razões da adoção do JSON para intercâmbio de informações é o fato de ele ser legível para humanos e fácil para os programas gerarem.

Uma das facilidades para a manipulação do JSON por linguagens de programação é a sua estrutura ser facilmente associada a estruturas de dados existentes nas mais diversas linguagens. Com o padrão tendo como base duas estruturas: uma coleção de pares nome-valor e uma lista ordenada de valores.

A coleção de pares nome-valor pode facilmente ser associada com estruturas como objetos, registros, dicionários e outros. Já a lista ordenada de valores pode facilmente ser relacionada com estruturas como vetores, matrizes e listas. Um exemplo da notação utilizada em arquivos JSON, representando dois registros do exemplo anterior, utilizado com arquivos CSV, será da seguinte forma:

```
[{"nome": "Marcel da Silva", "idade": 22,
 "nacionalidade": "brasileiro", "altura": 1,70,
 "peso": 75, "sexo": "masculino",
 "gosta de futebol?": true, "salario": 2500},
 {"nome": "Ruan del Silva", "idade": 52,
 "nacionalidade": "mexicano", "altura": 1,75,
 "peso": 80, "sexo": "masculino",
 "gosta de futebol?": true, "salario": 2500}]
```

Pela organização do padrão, podemos notar os pares ordenados, por exemplo, o par “idade”: 22, que em conjunto com os demais dados de um registro de um cliente forma a lista delimitada por um par de chaves ({}) – object. Já a lista ordenada desses registros está delimitada por um par de colchetes ([]) – array. Se observarmos atentamente, notaremos que os valores de cada atributo são armazenados de maneira que valores numéricos não são armazenados com delimitador, enquanto valores que sejam associados a variáveis do tipo string, sim. Podemos notar ainda que os valores de atributos que possam ser representados por valores booleanos também são armazenados com o seu tipo de valor indicado. Por exemplo, o par “gosta de futebol?”: true. Isso ocorre porque o padrão JSON possui tipos de dados que determinam o tipo do dado armazenado.

Como o JSON possui, além das suas estruturas internas, tipos de dados, quando os dados são exportados para o formato é preciso fazer uma conversão dos dados de origem para os seus equivalentes no JSON. O contrário também acontece no momento em que precisamos carregar os dados em formato JSON em estruturas de dados de uma linguagem de programação ou em suas

bibliotecas. A esses dois processos, respectivamente, damos o nome de serialização e desserialização. Os processos recebem esse nome por conta da conversão dos dados em uma série de *bytes*, sendo um processo comum a outros formatos de dados. Esse processo de equivalência não é igual para toda linguagem de programação e depende de como a linguagem implementa o tratamento de arquivos JSON. Contudo, o resultado do arquivo gerado, por conta da sua padronização, permite a comunicação entre diferentes linguagens. No Quadro 2 a seguir apresentamos a equivalência entre os tipos de dados do Python e do JSON nos processos de serialização e desserialização.

Quadro 2 – Equivalência entre os tipos de dados do Python e do JSON nos processos de serialização e desserialização

Python (serialização)	→ **JSON** →	**Python (desserialização)**
dict	object	dict
list, tuple	array	list
str	string	str
int, long, float	number	int, float
True, False	true, false	True, False
None	null	None

Fonte: Sousa, 2020.

O Python possui um módulo nativo que permite trabalhar com JSON para execução de serialização e desserialização de dados. O código a seguir ilustra como os dados podem ser armazenados em um arquivo e recuperados dele:

```
import json
with open("dados_clientes.json", "r") as arquivo_json:
  # Itera as linhas do arquivo
  dados_clientes = json.load(arquivo_json)
```

Uma grande vantagem do uso do JSON, além das facilidades na utilização dos tipos de dados e representação similar a estruturas comuns nas linguagens de programação, é a sua flexibilidade de representação. É possível utilizar as suas estruturas de forma aninhada e em diferentes conformações, permitindo representar objetos complexos e realizar o intercâmbio de informações. Essa

flexibilidade faz com que o JSON seja utilizado não apenas em aprendizagem de máquina, mas em diversas outras áreas, como destaque para a comunicação entre programas e APIs REST.

1.3 Outros formatos

Os formatos JSON e CSV não são os únicos utilizados para intercâmbio de dados. Muitos outros também apresentam características e qualidades que os tornam mais adequados para problemas distintos. Entre esses formatos podemos destacar o XML (eX*tensible Markup Language*).

O XML é uma linguagem de marcação genérica criada com um conjunto de regras para descrever documentos de maneira que essa definição seja legível para humanos e computadores. Assim, JSON é utilizado para transferência de dados por meio de redes de informação e, apesar de ter sido criada para descrição de documentos, a linguagem pode ser usada para descrever as mais variadas estruturas de dados, sendo muito utilizada na comunicação com *web services*.

Por conta das definição e estruturação padronizada do XML, a forma como os dados estão representados nos documentos XML podem ser validados por meio de documentos do tipo DTD (*Document Type Definition*) sendo possível validar se determinado documento está de acordo com um conjunto de definições, evitando erros e permitindo padronização e validação no intercâmbio de dados.

A representação de um registro de cliente em formato XML poderia ter a seguinte representação:

```
<?xml version="1.0" ?>
<cliente>
    <campo name="Nome" tipo="str">Marcel da Silva</campo>
    <campo name="Idade" tipo="int">22</campo>
    <campo name="Nacionalidade" tipo="str">brasileiro</campo>
    <campo name="Altura" tipo="float">1,70</campo>
    <campo name="Peso" tipo="float">75</campo>
    <campo name="Sexo" tipo="str">masculino</campo>
    <campo name="Gosta de Futebol?" tipo="bool">True</campo>
    <campo name="Salario" tipo="float">2500</campo>
</cliente>
```

Observando a representação dos dados em formato XML, podemos notar que, diferentemente do JSON, o foco é maior na descrição dos dados que no processo de conversão deles entre os dados em formato texto e as estruturas

de dados em memória. Quando trabalhamos com dados para aprendizagem é comum encontrarmos arquivos em JSON, CSV ou XML. O Python, assim como diversas linguagens, possui bibliotecas para manipulação desse tipo de arquivos. Conhecer o funcionamento, a estrutura e métodos de manipulação desses formatos é muito importante tanto para aprendizagem de máquina quanto para o desenvolvimento de programas que lidam com dados em geral.

TEMA 2 – PROCESSAMENTO NUMÉRICO

Nesta seção, apresentaremos a biblioteca NumPy e o seu funcionamento. Essa biblioteca, seus objetos e métodos são de grande importância para o desenvolvimento de sistemas de aprendizagem de máquinas, pois, como vimos anteriormente, nessa área trabalha-se constantemente com vetores numéricos de altas dimensões e boa parte dos cálculos são efetuados de forma matricial para obter maior eficiência.

2.1 Biblioteca NumPy

Apesar de a linguagem Python apresentar vantagens para escrita de códigos e poder ser considerada como a linguagem mais comum em projeto de aprendizagem de máquina e inteligência artificial, ela é uma linguagem interpretada, o que impacta diretamente a execução de códigos que exijam alto desempenho. Para suprir a necessidade de desempenho de velocidade de execução, especialmente de cálculos de computação científica, a biblioteca NumPy foi criada.

NumPy, assim como todas as outras que aprenderemos nesta aula, não é uma biblioteca nativa, sendo necessária a sua instalação após a configuração do ambiente de desenvolvimento. A instalação é simples, como pode ser visto no exemplo a seguir:

```
# create the virtual environment
python3 -m venv
# activate the virtual environment
source venv/bin/activate
# upgrade pip package manager
pip3 install --upgrade pip
#install numpy
pip3 install numpy
```

O principal objeto em NumPy é o ndarray, um array de **n** dimensões contendo elementos do mesmo tipo. A obrigatoriedade do mesmo tipo de dados no ndarray exige atenção ao seu uso, pois não podemos ter dados do tipo int e float no mesmo array, ainda que estes sejam tipos numéricos. Esses dados se distribuem pelas dimensões dos arrays que no NumPy são referenciadas como axes, sendo importante conhecer os objetos da biblioteca, como construí-los e inspecioná-los. O trecho de código e a sua saída, mostrados a seguir, exemplificam cada uma dessas tarefas.

```
import numpy as np

# array de 1 dimensão (axis)
array_1_dim = np.array([1, 2, 3])
# array de 2 dimensões (axis)
array_2_dim = np.array([[1, 2, 3],
       [4, 5, 6]])
# array de 2 dimensões (axis)
array_3_dim = np.array([[(1, 2, 3),
        (4, 5, 6)],
       [(1, 2, 3),
        (4, 5, 6)]])
# Verificando as dimensões dos arrays
print("array_1_dim: {} dimensão, formato {}, {} elementos \
 do tipo {}".format(array_1_dim.ndim,array_1_dim.shape,
        array_1_dim.size,array_1_dim.dtype))
print("array_2_dim: {} dimensões,{} formato,{} elementos \
  do tipo {}".format(array_2_dim.ndim, array_2_dim.shape,
        array_2_dim.size,array_2_dim.dtype))
print("array_3_dim: {} dimensões,{} formato,{} elementos \
  do tipo {}".format(array_3_dim.ndim, array_3_dim.shape,
        array_3_dim.size, array_3_dim.dtype))
```

array_1_dim: 1 dimensão,(3,) shape,3 elementos do tipo int64
array_2_dim: 2 dimensões,(2, 3) shape,6 elementos do tipo int64
array_3_dim: 3 dimensões,(2, 2, 3) shape,12 elementos do tipo int64

Perceba que os arrays são criados com base em listas aninhadas, mas o último (array_3_dim) possui, além de listas, tuplas como os elementos mais internos. Quaisquer um desses tipos pode ser utilizado na criação de ndarrays, atentando sempre para o formato (shape) e para o tipo que, no exemplo anterior, aparece como int64; mas outros tipos (int8,int16,int32,float) também podem ser

utilizados a depender da precisão necessária. Esse cuidado é necessário, pois problemas de conversão podem acontecer quando se utiliza dados de fontes distintas e diversas bibliotecas, inclusive nos casos em que a conversão é feita de forma implícita. Por isso, a checagem dos tipos de dados ou a especificação explícita no momento da criação dos dados é de grande importância.

Além dos processos de criação, diversas operações podem ser feitas com NumPy. Devemod conhecer as operações mais importantes, como manipulação e cálculos envolvendo arrays, mas para os propósitos da aprendizagem de máquina não há necessidade de se aprofundar nos métodos algébricos e numéricos utilizados. Contudo, os seus conhecimentos acerca dos fundamentos de álgebra linear lhe serão necessários para guiá-lo aos resultados esperados.

2.2.1 Exemplos de uso

Apresentaremos aqui algumas operações que podem ser efetuadas utilizando o NumPy na manipulação de arrays e efetuando cálculos. No NumPy, é possível além de criar arrays por meio de estruturas de dados do Python, criar matrizes com propósitos específicos. Veja os exemplos a seguir:

```
1_dim_zeros = np.zeros(2, dtype=np.int32)
2_dim_zeros = np.zeros((3,2),dtype=np.int32)
```

1_dim_zeros: 1 dimensões,(2,) shape,2 elementos do tipo int32
2_dim_zeros: 2 dimensões,(3, 2) shape,6 elementos do tipo int16

No processo de criação dos arrays, note que o tipo (dytpe) especificado é diferente do objeto int padrão do Python. Por conta disso, sempre que arrays NumPy forem criados devemos utilizar o tipo de acordo com o conjunto de tipos da biblioteca (numpy.dtypes). A documentação da biblioteca pode fornecer todos os tipos disponíveis de acordo com a versão. Note também que, na criação do array de duas dimensões, o formato é passado por meio da tupla (3,2), a qual pode ser do tamanho que se desejar criar o array. Diferentes tipos de arrays podem ser criados das seguintes maneiras:

- np.zeros(3): cria um array de 1 dimensão com 3 elementos preenchidos com zeros;
- np.ones(3): cria um array de 1 dimensão com 3 elementos preenchidos com uns;

- np.arange(0, 6, 2): cria um array de 1 dimensão com os números entre 0 e 6 contados de 2 em 2;
- np.empty([2, 2]): cria um arrray vazio de 2 dimensões;
- np.linspace(1,100, num=13): cria uma array de dimensão 1 com 10 elementos igualmente espaçados no intervalo de 1 a 100;
- np.full([3,3,3], 5): cria um array de 3 dimensões com 9 elementos todos preenchidos com o valor 5;
- np.diag([1,2,3], k=0): cria uma matriz diagonal 3x3 com 1,2,3 como elementos da diagonal;
- np.identity(3): cria uma matriz identidade de 3x3;
- np.eye(5, k=1): cria uma matriz com uma diagonal preenchida somente com 1 e o restante com 0;
- np.random.rand(3,3): cria uma matriz 3x3 com valores randômicos.

Além dos métodos de criação do NumPy, é importante conhecer dois processos denominados *vectorization* e *broadcasting*. *Vectorization* é uma maneira de ganhar *performance*, evitando *loops*, executando operações por meio de vetores. Já o *broadcasting* é a expansão de um elemento menor para execução de cálculo para um elemento maior. Assim, se multiplicarmos uma matriz por um único número, o NumPy automaticamente multiplicará todos os elementos da matriz por aquele número.

Para exemplificar o funcionamento de *vectorization* e *broadcasting*, imaginemos que há um vetor de características que define uma pessoa e desejamos compará-lo com todos os outros vetores de todas as pessoas inseridas na sua aplicação. A maneira mais intuitiva de fazer isso seria por meio de um *loop* sobre todos os vetores de características e um *loop* interno comparando os elementos um a um. Ao executar esse cálculo de forma vetorial, podemos criar uma matriz em que o nosso vetor de características se repete, e depois comparar uma matriz com a outra. Todavia, se considerarmos o processo de *broadcasting*, podemos comparar diretamente a matriz contendo todos os vetores de características com o vetor do cliente específico. Veja o código a seguir, que executa os dois processos.

```
import numpy as np
cliente_01 = [22,0,1.70,75,0,1,2500]
todos_clientes = [[52,1,1.75,80,0,1,2500],
    [31,1,1.50,65,1,1,4500],
```

```
    [65,1,1.95,86,0,0,6500],
    [17,0,1.81,95,0,1,1500],
    [54,0,1.65,80,1,0,3500],
    [30,0,1.90,105,1,1,1200],
    [25,1,1.85,65,1,1,2245],
    [49,0,1.71,75,0,0,25000],
    [26,1,1.81,80,1,1,8000]]
# Comparação do vetor de 1 cliente com todos os demais
array_todos = np.array(todos_clientes)

# Matriz com a mesma quantidade de linhas de array_todos
# mas com a mesma linha repetida
array_cli_01 = np.array([cliente_01 for _ in range(vetor_todos.shape[0])])
array_comparacao = array_todos - array_cli_01

# Comparação por broadcasting
vetor_cli_01 = np.array(cliente_01)
array_comparacao = array_todos - vetor_cli_01
```

Nos dois casos, o valor resultante será o mesmo:

```
[[ 30. 1. 0.05  5. 0. 0.   0. ]
 [  9. 1. -0.2 -10. 1. 0. 2000. ]
 [ 43. 1. 0.25 11. 0. -1. 4000. ]
 [ -5. 0. 0.11 20. 0. 0. -1000. ]
 [ 32. 0. -0.05  5. 1. -1. 1000. ]
 [  8. 0. 0.2  30. 1. 0. -1300. ]
 [  3. 1. 0.15 -10. 1. 0. -255. ]
 [ 27. 0. 0.01  0. 0. -1. 22500. ]
 [  4. 1. 0.11  5. 1. 0. 5500. ]]
```

Outras operações e comparações são possíveis, entre elas, a filtragem de informações das matrizes. Da matriz obtida por comparação entre todos os clientes, se quisermos extrair apenas aqueles cujo salário (último item/coluna do vetor de características) seja menor que o do cliente de referência, o código a seguir executado sobre a matriz resultante da comparação nos fornecerá essa informação.

```
idx_salarios_menores = [ idx for idx,vec in      enumerate(array_comparacao) if vec[0] < 0]
```

Esse código traz como resultado **[3, 5, 6]**, que são as linhas cujos valores e comparação nos trouxeram os valores negativos. Essa lista pode ser aplicada diretamente sobre a matriz com todos os vetores de características para nos fornecer apenas aquelas nas quais estamos interessados.

```
todos_clientes[idx_salarios_menores]
```

Isso nos fornece uma nova matriz apenas com as informações que queremos:

```
[[ 17.  0.  1.81 95.  0.  1.  1500. ]
 [ 30.  0.  1.9  105.  1.  1.  1200. ]
 [ 25.  1.  1.85 65.  1.  1.  2245. ]]
```

Esse tipo de operação é bem comum, principalmente para coleta dos resultados de comparação com informações retornadas pelos modelos de aprendizagem.

A intenção desta seção é introdutória em relação ao funcionamento da biblioteca NumPy. Há muitas outras funções disponíveis e aconselhamos o estudo mais aprofundado de alguns métodos que podem ser úteis em aprendizagem de máquina. Contudo, com a utilização de biblioteca cria-se mais familiaridade e prática.

2.2 Outras bibliotecas

A biblioteca NumPy é uma das muitas opções que podem ser utilizadas para cálculo computacional e científico. Uma outra biblioteca Python bem conhecida é SciPy, que é voltada especificamente para estudos científicos e que utiliza algumas funções do NumPy. Assim, podemos usar de forma indistinta as duas.

As funcionalidades do NumPy permitem a criação e execução de procedimentos essenciais para a criação de modelos e algoritmos de aprendizagem de máquina. Por exemplo, a definição e o treinamento de redes neurais, por meio de bibliotecas específicas com o Scikit-Learn, uma biblioteca específica para processos e métodos de aprendizagem de máquina que também utiliza o NumPy como base.

Algumas bibliotecas – hoje em dia bastante conhecidas como Tensorflow e Pytorch – essencialmente fazem o trabalho que a combinação NumPy e Scikit-Learn faz, permitindo o recebimento de dados de entrada, criação, treinamento e refinamento de modelos, bem como a disponibilização destes. Entretanto, antes de iniciarmos os trabalhos relacionados aos modelos de aprendizagem, precisamos de uma maneira prática de lidar com os dados, após a sua carga, entender o seu comportamento e como estes podem ser organizados para facilitar o trabalho em aprendizagem de máquina.

TEMA 3 – MANIPULANDO DADOS TABULARES

Assim como na seção da biblioteca NumPy, nosso objetivo nesta seção será apresentar o funcionamento básico da biblioteca Pandas para uso em aprendizagem de máquina. Não esgotamos o assunto, que é muito vasto, e como sempre encorajamos a busca de mais referências e textos sobre o assunto.

3.1 Biblioteca Pandas

O Pandas é uma biblioteca de código aberto para análise de dados em Python. Ela foi desenvolvida por Wes McKinney, em 2008, e vem sendo melhorada desde então. Entre as vantagens que ela oferece, estão a capacidade de lidar com dados de tipos de diferentes, carregar e importar de diferentes fontes de dados, filtragem, separação, agrupamento e ordenação de dados, além de ser facilmente integrável com outros bibliotecas do ecossistema Python, como SciPy e Scikit-Learn.

A principal estrutura de dados do Pandas para uso em aprendizagem de máquina e que abordaremos aqui é o DataFrame. Ele representa uma estrutura semelhante à um array de duas dimensões, com as suas colunas contendo dados de tipos heterogêneos, sendo muitas vezes comparado com planilhas eletrônicas. Ele possui índices distintos para colunas e para linhas, como também pode ser criado de formas distintas, como mostramos no código a seguir:

```
import pandas as pd
dados_clientes = {"idade": [22, 52, 31, 65, 17],
      "nacionalidade": ["brasileiro","estrangeiro",
          "estrangeiro", "estrangeiro",
        "brasileiro"],
      "altura": [1.7, 1.75, 1.5, 1.95, 1.81],
      "peso": [75.0, 80.0, 65.0, 86.0, 95.0],
      "sexo": ["feminino", "masculino", "feminino",
         "masculino", "feminino"],
      "gosta de futebol": [True, True, True,
 False, True],     "salario": [2500, 2500, 4500, 6500, 1500]}
df_clientes = pd.DataFrame(data=dados_clientes,
index=["Ana","Beto","Bia","Carlos","Carol"])
```

O resultado será um DataFrame com a seguinte apresentação.

Figura 3 – Apresentação do DataFrame obtido como resultado

	idade	nacionalidade	altura	peso	sexo	gosta de futebol	salario
Ana	22	brasileiro	1.70	75.0	feminino	True	2500
Beto	52	estrangeiro	1.75	80.0	masculino	True	2500
Bia	31	estrangeiro	1.50	65.0	feminino	True	4500
Carlos	65	estrangeiro	1.95	86.0	masculino	False	6500
Carol	17	brasileiro	1.81	95.0	feminino	True	1500

Fonte: Sousa, 2020.

Um DataFrame pode ser construído de diversas maneiras, utilizando diferentes entradas de dados. A Quadro 3 a seguir traz algumas das possibilidades.

Quadro 3 – DataFrame: tipo de dado x comportamento

TIPO DE DADO	COMPORTAMENTO
Dicionário de listas e tuplas	Cada objeto iterável dentro do dicionário se torna uma coluna e a chave se torna o rótulo da coluna
Dicionário de dicionários	Cada dicionário se torna uma coluna e as chaves dos dicionários internos se tornam os rótulos das linhas
Lista de dicionários, lista ou tuplas	Cada item se torna uma linha do DataFrame
Outro DataFrame	Cria um DataFrame igual ou com novos índices, se estes forem passados
Array NumPy	Cria um DataFrame de forma semelhante à quando passamos uma lista de listas

Fonte: Sousa, 2020.

Cada DataFrame pode armazenar dados de diferentes tipos e podemos acessar os valores de uma coluna ou item da mesma forma como referenciamos os índices de um dicionário. É possível o uso de métodos de filtragem e seleção, como:

- selecionar apenas uma coluna:

```
df_clientes["idade"]
```

```
Ana   22
Beto  52
Bia   31
Carlos 65
```

```
Carol  17
Name: idade, dtype: int64
```

- selecionar apenas uma linha:

```
df_clientes.loc["Ana"]
```

```
idade         22
nacionalidade   brasileiro
altura         1.7
peso           75
sexo         feminino
gosta de futebol     True
salario        2500
Name: Ana, dtype: object
```

- filtrar todas as pessoas com idade maior que 32 anos e exibir apenas determinadas colunas:

```
df_filtro = df_clientes.loc[:,["idade","sexo","salario"]]

df_filtro[[df_filtro.salario > 3000]]
```

```
        idade      sexo  salario
Bia     31   feminino   4500
Carlos  65 masculino   6500
```

O Pandas possui diversos outros métodos para manipular dados e carregá-los, permitindo inclusive carregar os dados por meio de arquivos do tipo CSV e JSON. Essas funcionalidades em conjunto com os outros métodos que vimos anteriormente serão muito úteis tanto para importação quanto para exportação de dados. Contudo, ainda necessitamos de uma ferramenta que nos permita avaliar a distribuição dos dados, por meio da exploração de diferentes visualizações.

TEMA 4 – VISUALIZAÇÃO DE DADOS

Nesta seção, apresentaremos uma biblioteca do Python utilizada para gerar visualizações, geralmente referenciadas como plotagens. Essas visualizações permitem encontrar grupos ou valores discrepantes, bem como prover *insights* sobre os dados.

4.1 Biblioteca matplotlib

A biblioteca matplotlib possibilita gerar visualizações de diferentes tipos de dados no Python, podendo ser utilizada em diferentes sistemas operacionais e gerar diferentes tipos de arquivos de saída, como pdf, jpeg, png, svg, entre

outros. Uma grande facilidade do uso dessa biblioteca é a fácil integração com NumPy e Pandas.

4.2 Exemplos de uso

Em aprendizagem de máquina, especialmente nas fases de preparação dos dados, é preciso conhecê-los, para que assim seja possível gerar novas informações sobre eles e, dessa maneira, obter modelos mais assertivos e com melhor desempenho. Para ilustrar o funcionamento das bibliotecas vistas até agora, apresentamos a seguir códigos de carga de dados por meio de um arquivo CSV, gerando um DataFrame e a plotagem de diferentes visualizações desses dados.

```
# Importação das bibliotecas
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Carrega os dados do dataframe de um arquivo csv
df_clientes = pd.read_csv("dados_clientes.csv")
```

Figura 4 – Plotagem 1

	nome	idade	nacionalidade	altura	peso	sexo	gosta de futebol	salario
0	Aldo	22	0	1.70	75	0	1	2500
1	Beto	52	1	1.75	80	0	1	2500
2	Bia	31	1	1.50	65	1	1	4500
3	Carlos	65	1	1.95	86	0	0	6500
4	Diego	17	0	1.81	95	0	1	1500
5	Daina	54	0	1.65	80	1	0	3500
6	Carol	30	0	1.90	105	1	1	1200
7	Fabia	25	1	1.85	65	1	1	2245
8	Felix	49	0	1.71	75	0	0	25000
9	Gaia	26	1	1.81	80	1	1	8000

Fonte: Sousa, 2020.

```
# Prepara as figuras com 4 gráficos
fig = plt.figure()
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)
```

```
ax4 = fig.add_subplot(2,2,4)

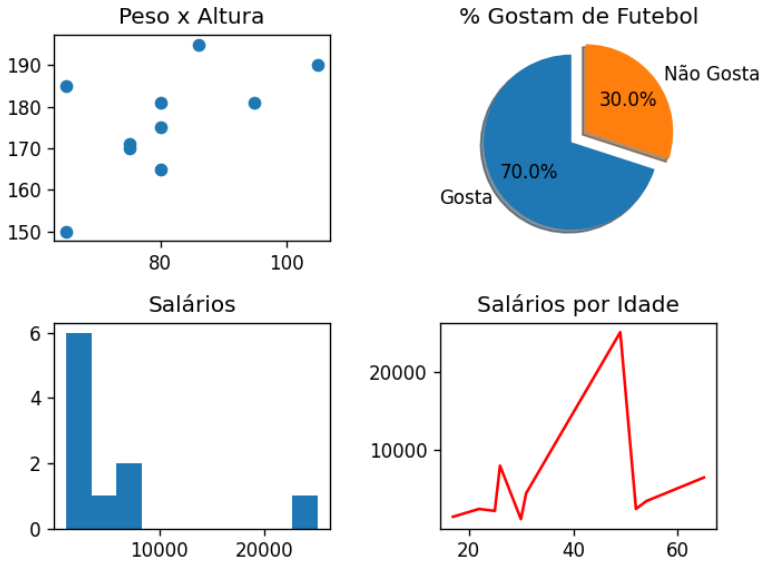
# Plota distribuição das idades dos clientes
ax1.set_title("Peso x Altura")
idades_clientes = df_clientes["idade"].values
pesos_clientes = df_clientes["peso"].values
alturas_clientes = df_clientes["altura"].values*100
cores = np.random.rand(df_clientes["idade"].nunique())
ax1.scatter(pesos_clientes,alturas_clientes)

# Plota gosto por futebol
ax2.set_title("% Gostam de Futebol")
gosta_futebol = df_clientes["gosta de futebol"]
labels = 'Gosta', 'Não Gosta'
sizes = df_clientes["gosta de futebol"].value_counts()
explode = (0.1, 0.1)
ax2.pie(sizes, explode=explode, labels=labels, autopct='%1.1f% %', shadow=True,
startangle=90)
ax2.axis('equal')

# Plota a histograma de salários
ax3.set_title("Salários")
salarios_clientes = df_clientes["salario"].values
nomes_clientes = df_clientes.index
ax3.hist(salarios_clientes)

# Plota idades x salários dos clientes
ax4.set_title("Salários por Idade")
df_clientes = df_clientes.sort_values("idade")
idades_clientes = df_clientes["idade"].values
salarios_clientes = df_clientes["salario"].values
ax4.plot(idades_clientes, salarios_clientes, "r-")
plt.subplots_adjust(wspace=0.4,hspace=0.4)
plt.show()
```

Figura 5 – Plotagem 2



Fonte: Sousa, 2020.

A visualização dos dados sob diferentes perspectivas é uma poderosa ferramenta para buscar correlações entre eles, avaliar possibilidades de agrupamentos, detectar *outliers* e avaliar o balanceamento entre suas diferentes categorias. Nesta seção, nosso objetivo era mostrar como a integração entre Pandas e Matplotlib pode ser útil no processo de exploração e preparação dos dados, antes de iniciarmos os treinamentos de modelos de inteligência.

4.3 Outras bibliotecas

Com a evolução da biblioteca Matplotlib, outras bibliotecas baseadas nela surgiram. Um exemplo é a biblioteca Seaborn, que apresenta funcionalidades semelhantes, mas algumas facilidades para construção e apresentação dos gráficos. Além da Seaborn, outras bibliotecas como Bokeh e Plotly são comumente utilizadas para os mesmos fins, cada uma delas com as suas vantagens e desvantagens a depender do problema que se queira resolver. Recomendamos conhecer inicialmente o funcionamento básico da Matplotlib, pois isso servirá como referência para quaisquer outras que venhamos a utilizar.

TEMA 5 – BASES DE DADOS DE APRENDIZAGEM

Nas seções anteriores, apresentamos algumas ferramentas para utilização nos processos de exploração e preparação dos dados, os quais devem ser executados antes mesmo de podermos iniciar o treinamento dos modelos de inteligência, e que ajudarão na definição do método e dos atributos que serão utilizados.

Agora que sabemos quais ferramentas precisamos utilizar para manipular os dados, vamos conversar sobre os dados em si e como eles podem ser obtidos, além de fontes de dados que são fornecidas gratuitamente para problema de aprendizagem de máquina que são conhecidos e que podem ser expandidos ou refinados para outros problemas.

5.1 Bases de dados

As bases de dados ou *datasets* se referem ao conjunto de dados fornecidos aos algoritmos de aprendizagem de máquina para que estes aprendam padrões sobre esses dados. Um conceito que devemos sempre ter em mente é que a qualidade dos seus dados tem grande impacto sobre a qualidade do seu modelo.

Um *dataset* pode ser armazenado em diversos formatos e estruturas, sendo muito comum a sua distribuição em formatos como JSON e CSV. Assim, para compreendermos melhor um *dataset*, é preciso conhecer o que é instância. A instância é uma referência a um registro ou uma linha de dados de um *dataset*, contendo atributos comuns. Essa descrição é parecida com o vetor de características. Comumente, cada linha de um *dataset* traz um vetor de características e as classes associado a cada instância.

Antes de serem repassados para os algoritmos de aprendizagem, as instâncias de um *dataset* são divididos em conjuntos menores: *dataset* de treinamento (utilizado pelo algoritmo para aprender padrões) e *dataset* de teste ou validação (utilizado para avaliar o quão bem o modelo está interpretando os dados fornecidos). Esse processo de fornecimento de dados é dinâmico e, mesmo após a obtenção de um modelo, pode ser melhorado por meio da adição de novos dados ou de uma melhora aplicada sobre os dados anteriormente fornecidos.

Os dados fornecidos nos *datasets* podem ser fornecidos em formatos distintos:

- **dados numéricos** – dados quantitativos representando medidas como altura, peso, idade etc.;
- **dados categóricos** – dados não numéricos que definem características descritivas como gênero, etnia, naturalidade ou quaisquer outros rótulos que permitam descrever uma instância do nosso *dataset*;
- **dados de séries temporais** – dados numéricos coletados em intervalos de tempo consistentes, que permitem comparar os dados diária, mensal ou anualmente;
- **dados textuais** – conjunto de palavras, frases, parágrafos ou documentos que permitam gerar informação sobre os textos.

A obtenção de *datasets* pode ser feita em portais específicos como o *Dataset Research*, do Google, o *Microsoft Research Open Data*, o *Open Data on AWS*, da Amazon, e o *UCI Machine Learning Repository*. Este último trata-se de um repositório da Universidade da Califórnia, contendo bases voltadas para problemas e estudos acadêmicos, mas que, assim como todos os demais, podem ser usados livremente para quaisquer outros propósitos.

Ao buscar um *dataset*, ou até mesmo depois de obtê-lo, devemos nos atentar para as seguintes características:

- **tipo de tarefa** – para qual tarefa originalmente aquele *dataset* foi concebido, como classificação, regressão, agrupamento ou outro tipo;
- **tipos dos atributos** – categórico, numérico, misto;
- **tipos de dados** – multivariado, univariado, sequencial, série temporal, texto, entre outros;
- **quantidade de atributos**;
- **quantidade de instâncias**;
- **formato dos dados**.

As informações sobre o *dataset* permitem entender melhor como utilizar os dados e também como obter os subconjuntos de treinamento, teste e validação a serem utilizados no treinamento dos modelos.

5.2 Bases comuns

Nos repositórios de dados de aprendizagem de máquina, a quantidade e variedade de *datasets* é imensa, com novos sendo construídos e disponibilizados. Todavia, alguns são bastante conhecidos na área, seja pela sua facilidade didática ou por conta da qualidade e quantidade de dados. O Quadro 4 a seguir traz alguns *datasets* que comumente são utilizados no ensino de aprendizagem de máquina.

Quadro 4 – Alguns *datasets* comumente são utilizados no ensino de aprendizagem de máquina

DATASET	**DESCRIÇÃO**	**TAREFAS**
Iris Flower	Contém 150 instâncias de 3 variações da planta Iris, com valores de 4 atributos.	Classificação
Wine Quality	Contém 4.898 instâncias de tipos de vinhos com notas de 0 a 10 para a qualidade, com valores de 11 atributos.	Classificação; Regressão
Car Evaluation	Contém 1.728 instâncias de carros, com 6 atributos e 4 classes distintas de aceitabilidade dos modelos de carro.	Classificação
Boston House Pricing	Contém 506 instâncias, com 13 atributos que descrevem residências, com o valor de venda de cada uma, e a sua vizinhança.	Regressão
MNIST	Contém 60.000 instâncias de dígitos manuscritos, suas imagens distribuídas em 10 classes (0-9).	Classificação
ImageNet	Contém mais de 14 milhões de instâncias de imagens, distribuídas em mais de 20.000 categorias.	Reconhecimento de objeto; Classificação de imagens

Fonte: Sousa, 2020.

Uma das primeiras avaliações que necessitará realizar ao se deparar com um problema a ser resolvido com aprendizagem de máquina é avaliar a disponibilidade de dados. Caso já exista um *dataset* que se aplica ao seu problema, pegue-o e faça uma análise da sua estrutura, de modelos que possam existir disponíveis e quais os resultados obtidos utilizando os *dataset* que obtiver.

Podemos notar que a quantidade de instâncias e de classes nos *datasets* pode variar de dezenas a milhares, o que torna custoso o problema de treinar os modelos computacionalmente, exigindo muito espaço de armazenamento e alto poder de processamento principalmente se ainda for necessário acrescentar seus próprios dados. Todo esse processo é facilitado pelas ferramentas sobre as quais conversamos nesta aula. A seguir, vamos apresentar a carga de um *dataset* e a sua separação nos conjuntos de treinamento e de testes, bem como uma visualização dos dados das instâncias desse *dataset*.

5.3 Trabalhando os dados

Para ilustrar como trabalhar com um *dataset*, utilizaremos o Iris, que contém 3 classes com 4 atributos que identificam tipos diferentes da planta íris por meio de informações de medidas das flores. A indicação da classe de cada instância desse *dataset* é feita por meio do nome da planta (iris-setosa, iris-versicolor, iris-virginica). Como necessitamos que esse valor seja numérico, algumas funcionalidade do Pandas podem nos ajudar com a conversão de cada categoria em um valor numérico. Após a obtenção do *dataset*, podemos visualizar as suas classes e os seus atributos.

```
df_irisdata = pd.read_csv("iris.data",
    names=['sepal length', 'sepal width',
'petal length', 'petal width',          'class'])

df_irisdata['class'] = df_irisdata['class'].astype('category')
df_irisdata['class_number'] = df_irisdata['class'].cat.codes
```

Figura 6 – Classes e atributos do *dataset*

	sepal length	sepal width	petal length	petal width	class	class_number
0	5.1	3.5	1.4	0.2	Iris-setosa	0
1	4.9	3.0	1.4	0.2	Iris-setosa	0
2	4.7	3.2	1.3	0.2	Iris-setosa	0
3	4.6	3.1	1.5	0.2	Iris-setosa	0
4	5.0	3.6	1.4	0.2	Iris-setosa	0
...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica	2
146	6.3	2.5	5.0	1.9	Iris-virginica	2
147	6.5	3.0	5.2	2.0	Iris-virginica	2
148	6.2	3.4	5.4	2.3	Iris-virginica	2
149	5.9	3.0	5.1	1.8	Iris-virginica	2

Fonte: Sousa, 2020.

Também podemos executar operações para prover informações gerais sobre o *dataset*.

```
df_irisdata.describe()
```

Figura 7 – Informações gerais sobre o *dataset*

	sepal length	sepal width	petal length	petal width	class_number
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667	1.000000
std	0.828066	0.433594	1.764420	0.763161	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Fonte: Sousa, 2020.

As informações sobre o *dataset* nos permitem avaliar a distribuição dos dados, pois como todas as contagens de atributos e de classes resultam no mesmo valor, podemos perceber que não há valores faltantes nas características. Assim, o valor do desvio-padrão (std) de cada um dos atributos

pode nos indicar quais deles possibilitam uma melhor separação entre as classes. Valores altos de desvio-padrão indicam que os valores daquele atributo específico para as diversas instâncias se distribuem em faixa grande de valores eu torno do valor da média. Já os valores muito baixos tendem a dificultar a separação das instâncias por conta da proximidade dos valores, da mesma forma que os valores mínimo e máximo nos permitem ter uma noção da amplitude de faixas de valores.

Uma vez que temos nossos dados carregados e analisados, necessitamos dividir em subconjuntos para treinamento e testes. Essa divisão deve levar em consideração as quantidades de cada classe que o *dataset* contêm. Idealmente, deveríamos ter quantidades iguais ou aproximadas de cada classe e o *dataset* poderia ser considerado balanceado. Todavia, em geral, as instâncias não se distribuem igualmente entre as classes, e em casos mais extremos, uma classe ou um pequeno número delas contém um número elevado de instâncias, tornando o *dataset* desbalanceado ou alta desbalanceado, o que pode afetar seriamente o desempenho dos modelos de aprendizagem, ainda que existam métodos para treinar os modelos utilizando *dataset* desbalanceados.

Um fator que deve ser considerado na divisão do *dataset* em subconjuntos é a variabilidade das instâncias, uma vez que, por meio de diferentes instâncias que lhe são apresentadas, o modelo aprende a reconhecer as que se afastem muito dos valores médios, adquirindo uma característica referenciada como poder ou capacidade de generalização. Essa característica permite que o modelo, partindo dos dados que lhe foram fornecidos durante o treinamento, reconheça uma nova instância que nunca tenham lhe apresentado.

A capacidade de generalização é muito importante e deve ser monitorada durante todo o treinamento, do contrário, podemos ter modelos que conseguem identificar com 100% de certeza determinadas classes e não reconhecer nenhuma outra. Para evitar esse tipo de ocorrência, é fundamental analisar os dados e dividi-los bem antes de repassar ao algoritmo de aprendizagem.

Todos esses conceitos de divisão dos subconjuntos com garantia de variabilidade como forma de obter capacidade de generalização pode parecer complexo, mas com o uso das bibliotecas Python todo o processo é simplificado.

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

```
iris_X, iris_y = datasets.load_iris(return_X_y=True)
X_train,X_test,y_train,y_test = train_test_split(iris_X, iris_y, random_state=0)
```

O *dataset* iris é carregado diretamente para um conjunto de arrays NumPy por meio da chamada datasets.load_iris, que carrega todos os vetores de características em iris_X e todos os rótulos indicando as classes de cada instância em iris_y. Essa é uma notação bem comum no uso do Python para aprendizagem de máquina, em que a matriz contendo os valores do atributos é carregada em uma variável com uma indicação da letra X maiúsculo, e as classes com seus valores numéricos são carregadas em um vetor com uma indicação da letra minúscula **y**. Em seguida, a chamada train_test_split divide cada uma das estruturas carregadas anteriormente nos dois subconjuntos de treinamento (X_train, y_train) e de teste (X_test, y_test), realizando um embaralhamento das instâncias e escolhendo 75% destas para compor a base de treino e 25% para compor a base de testes. Assim, busca-se garantir variabilidade na escolha das instâncias visando à capacidade de um modelo que venha a ser treinado com esse *dataset*.

FINALIZANDO

Nesta aula, apresentamos algumas das ferramentas com as quais realizaremos as etapas de aquisição, preparação e exploração dos dados, que são uma fase essencial antes de aplicarmos os algoritmos de aprendizagem de máquina. Conhecemos os formatos CSV e JSON, que comumente são utilizados para transferência de dados. Conhecemos, ainda, a biblioteca NumPy para cálculos vetorizados e manipulação de matrizes e vetores, que são operações comuns no dia a dia do profissional de aprendizagem de máquina. Vimos também a biblioteca Pandas e como ela pode facilitar a carga, a organização e a exploração de dados, bem como foi apresentada a biblioteca Matplotlib, utilizada para visualização de dados. Essas bibliotecas, em conjunto com o conhecimento sobre os *datasets* de aprendizagem, nos deixarão prontos para iniciarmos os primeiros processos de escolha e treinamento de algoritmos de aprendizagem. Caso algum conceito não tenha ficado claro, faça uma releitura mais atenta do material, pois esses conceitos são de suma importância para as nossas próximas etapas.

REFERÊNCIAS

CUESTA, H. **Practical data analysis**. Birmingham: Packt Publishing Ltd, 2013.

GÉRON, A. **Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow**: Concepts, tools, and techniques to build intelligent systems. Sebastopol: O'Reilly Media, 2019.

MCKINNEY, W. **Python para análise de dados**: tratamento de dados com Pandas, NumPy e IPython. São Paulo: Novatec, 2019.

RICHERT, W. **Building machine learning systems with Python**. Birmingham: Packt Publishing Ltd, 2013.

RUSSEL, S. J.; NORVIG, P. **Inteligência Artificial**: uma abordagem moderna. 3. ed. Tradução de Regina Célia Simille. Rio de Janeiro: Elsevier, 2013.