Pandas Python

Data Wrangling para Ciência de Dados

Casa do Código

EDUARDO CORRÊA

Sumário

- ISBN
- Prefácio
- 1. Muito prazer, biblioteca pandas
- 2. A estrutura de dados Series
- 3. A estrutura de dados DataFrame
- 4. Conhecendo os seus dados
- 5. Combinando DataFrames
- 6. Transformação e limpeza de DataFrames
- 7. Um pouco de Machine Learning

ISBN

Impresso e PDF: 978-85-7254-048-3

EPUB: 978-85-7254-049-0

MOBI: 978-85-7254-050-6

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio

Ciência de dados (*data science*) é um processo que emprega técnicas estatísticas e computacionais para analisar grandes bases de dados, procurando extrair delas conhecimento útil para empresas, instituições científicas, governos e demais organizações.

Tipicamente, os projetos de ciência de dados são divididos em quatro macroetapas de execução. A primeira consiste simplesmente na definição do problema que será resolvido (por exemplo, criar um sistema para caracterizar o perfil dos clientes de uma empresa). A segunda é a etapa de pré-processamento, onde as bases de dados relevantes (base de vendas, base de clientes etc.) devem ser reunidas e adequadamente formatadas. Na terceira etapa, um algoritmo é aplicado sobre os dados pré-processados, com o objetivo de extrair um modelo estatístico ou de *Machine Learning*. Este modelo tem por objetivo identificar padrões de relacionamento entre os itens de dados (por exemplo, o algoritmo pode gerar um modelo que revela as características mais comuns dos compradores de cada produto). Por fim, na quarta etapa, os especialistas da empresa avaliam os resultados gerados pelo modelo, procurando determinar a relevância e validade deles.

De acordo com a literatura, a etapa de pré-processamento de dados (segunda etapa) costuma ser a mais trabalhosa em qualquer projeto relacionado à ciência de dados, ocupando tipicamente 80% do tempo consumido. É nesta fase que são realizadas as tarefas de **seleção**, **limpeza** e **transformação** dos dados que serão utilizados pelo algoritmo de Machine Learning / Estatística. O objetivo da seleção de dados é coletar e reunir todos os dados que sejam relevantes para a resolução do problema de ciência de dados definido (por exemplo, combinar dados dos sistemas corporativos da empresa com dados disponibilizados na internet). Limpeza, significa eliminar sujeira e informações irrelevantes. Por fim, transformação consiste em converter os dados de origem para um outro formato, mais adequado para ser usado pelo algoritmo. As atividades de

seleção, limpeza e transformação de dados são comumente referenciadas como atividades de *Data Wrangling*, *Data Munging* ou *Data Preparation*.

A biblioteca pandas (*Python Data Analysis Library*) foi especialmente projetada para oferecer o suporte ao processo de Data Wrangling. Trata-se de um software livre, do tipo open source, que ao longo dos últimos anos se consolidou como a biblioteca para ciência de dados mais utilizada no ambiente Python. As funcionalidades oferecidas pela pandas consistem basicamente em uma combinação de técnicas eficientes para processamento de vetores e matrizes, com um conjunto de funções específicas para a manipulação de dados tabulares, que se assemelham muito às oferecidas pelo Excel e pela famosa linguagem SQL. Alguns exemplos:

- Importar, de forma direta e padronizada, dados estruturados de diferentes tipos de fontes, tais como: arquivos texto (CSV, JSON etc.), bancos de dados e planilhas eletrônicas.
- Combinar de forma inteligente registros provenientes de diferentes bases de dados (operação conhecida como `merge` ou `join`);
- Produzir resultados agregados e tabulações (`group by`);
- Limpar e transformar bases de dados (ex.: aplicação de filtros sobre linhas e colunas de tabelas, ordenação e ranqueamento dos dados, tratamento de dados ausentes etc.).
- Produzir diferentes tipos de gráficos a partir de dados armazenados em colunas de tabelas.

Qual o objetivo deste livro?

Este livro aborda a pandas sob uma perspectiva profissional, explicando como utilizá-la para resolver problemas práticos e, muitas vezes, difíceis de Data Wrangling. O livro combina a teoria com um projeto prático, que envolve o uso da pandas como ferramenta para

viabilizar a execução das atividades de seleção, estudo, limpeza e transformação de uma base de dados real que contém informações detalhadas sobre diversos países (variando desde a população e extensão de cada país até as características de suas bandeiras, entre outras informações). O projeto mostrará o passo a passo para realizar o pré-processamento desta base de dados, que será então utilizada como fonte para a criação de um modelo de Machine Learning, mais especificamente, um modelo de classificação de dados.

Público-alvo

- Estudantes e profissionais envolvidos com ciência de dados (independente da área ou nível de experiência);
- Usuários do Excel que pretendem migrar para o Python;
- Pessoas com conhecimento de SQL que pretendem trabalhar com Python.

Sobre o autor

Eduardo Corrêa Gonçalves cursou Doutorado em Ciência da Computação pela UFF (2015) com período sanduíche na University of Kent, no Reino Unido. Também cursou Mestrado (2004) e Graduação (1999) em Ciência da Computação pela UFF. Possui certificação Oracle Database SQL Certified Expert (OCE). Atualmente, trabalha como administrador de banco de dados no Instituto Brasileiro de Geografia e Estatística (IBGE) e também atua como professor colaborador na Escola Nacional de Ciências Estatísticas (ENCE-IBGE). Suas áreas de interesse são: Banco de Dados, Algoritmos, Processamento de Linguagem Natural e Python.

Agradecimentos

Inicialmente, agradeço a todos os alunos e professores da ENCE por terem contribuído com valiosas sugestões, críticas e ideias interessantes para este livro. É por isso que sempre fico feliz em lecionar disciplinas relacionadas à ciência de dados, banco de dados e Python para a graduação em Estatística!

Agradeço aos meus pais, Ana e Joaquim, pelo apoio de sempre. Obrigado, também, queridos Amanda, Inês e Antonio!

Mais do que tudo, agradeço a Glauce, o amor de minha vida, por seu carinho, compreensão e suporte ao longo dos últimos dez anos.

CAPÍTULO 1
Muito prazer, biblioteca pandas

Você já deve ter ouvido falar que, nos últimos anos, muitas empresas têm empregado a ciência de dados (*data science*) com o propósito de alcançar um melhor posicionamento no mercado.

Mas o que é exatamente ciência de dados? Quais as suas aplicações práticas? Que tipo de conhecimento posso obter em processos de ciência de dados? Este capítulo responde estas questões e explica por que a pandas se tornou uma das ferramentas mais utilizadas em projetos de ciência de dados.

1.1 O que é ciência de dados?

De uma maneira simples, é possível definir a ciência de dados como um processo que emprega técnicas estatísticas e computacionais para resolver o problema da descoberta de **conhecimento valioso** em grandes bases de dados. A figura 1.1 ilustra a ideia.

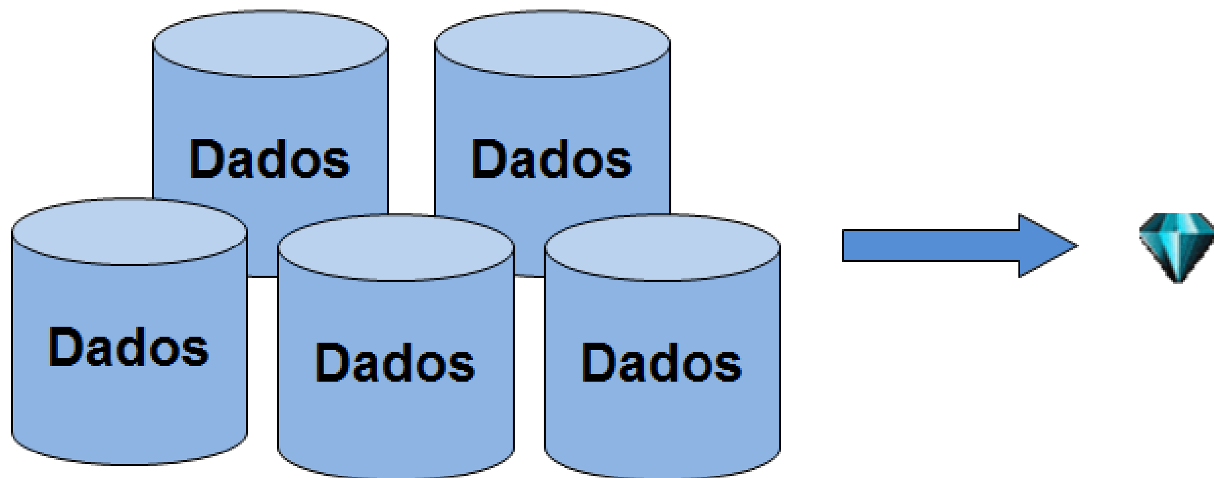

Figura 1.1: Ciência de dados: um pequeno, porém valioso diamante de conhecimento é extraído a partir de uma grande montanha de dados

A ciência de dados baseia-se na utilização de **algoritmos**, que são capazes de vasculhar grandes bases de dados de modo eficiente (ou seja, com certa rapidez) com o intuito de extrair **modelos** que descrevem padrões de relacionamento interessantes escondidos dentro da montanha de dados.

Como surgiu a ciência de dados?

De acordo com estudos recentes (veja mais em https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html), o volume total de dados digitais disponíveis no planeta é igual a 33ZB (33 zetabytes, ou seja, 33 x 10^{21} bytes), podendo atingir a marca de 175ZB em 2025. Estamos falando não apenas de **dados estruturados** (tabelas, planilhas etc.) mas, principalmente, de **dados não estruturados** (documentos, vídeos, fotos etc.). Diante desta situação, é bastante natural que cientistas e analistas de negócio alimentem o seguinte desejo: por que não tentamos analisar estes dados para que novas informações sejam descobertas e utilizadas de forma estratégica para a tomada de decisões?

A ideia é excelente, no entanto, a sua implementação não é nada trivial, uma vez que, na prática, é comum encontrar empresas que mantêm bancos de dados com bilhões ou trilhões de registros. Além disso, esses bancos são normalmente compostos por centenas ou milhares de atributos que precisam ser simultaneamente considerados durante o processo de análise. Desta maneira, o uso de métodos estatísticos ou computacionais tradicionais torna-se inviável: eles não são escaláveis para Big Data e tampouco são capazes de lidar com dados não estruturados. Este cenário motivou o surgimento da Ciência de Dados, uma nova linha de pesquisa que combina ideias da Estatística e da Ciência da Computação com o intuito de resolver o problema da descoberta de conhecimento em grandes bases de dados.

1.2 Quais os problemas resolvidos pela ciência de dados?

Os tópicos a seguir introduzem os três principais tipos de problemas que podem ser resolvidos através do emprego de técnicas de ciência de dados.

Descoberta de padrões frequentes

Neste tipo de problema, o objetivo é descobrir combinações de itens que ocorrem com frequência significativa e acima da que é esperada em uma base de dados. Considere, por exemplo, um banco de dados que registra as vendas efetuadas por uma loja que comercializa roupas através da internet. Um algoritmo de descoberta de padrões frequentes poderia analisar esta base e revelar o seguinte padrão:

"A compra do produto sapatênis aumenta em 5 vezes a chance de um cliente comprar o produto camisa polo"

Na prática, os padrões frequentes são muito utilizados para implementar **sistemas de recomendação** — sistemas que, de maneira autônoma, sugerem produtos/serviços para usuários, com o intuito de ajudá-los a tomar decisões como "qual produto comprar?", "qual música ouvir?", "qual hotel reservar?" etc.

Classificação

Classificação é um processo que consiste em treinar um programa de computador para que ele seja capaz de atribuir automaticamente as classes de um objeto. Os programas para filtragem de spam representam um exemplo de classificador bastante conhecido e bem-sucedido. A partir da análise do assunto e do texto de uma mensagem, o filtro de spam utiliza um algoritmo classificador para identificar automaticamente se ela deve ser classificada como "spam" ou "normal":

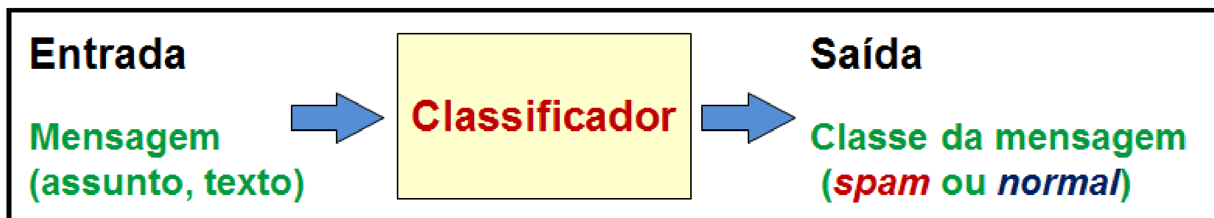

Figura 1.2: Classificador de spam

Nos dias atuais, técnicas de classificação vêm sendo exploradas pelas empresas para resolver diversos problemas importantes, tais como: detecção de fraudes (identificar se uma transação de cartão de crédito é "fraudulenta", "genuína"), classificação de músicas em emoções (por exemplo: "animada", "relaxante", "triste"), sugestão de tags (atribuir pequenas informações textuais a vídeos ou imagens), entre outras.

Determinação de agrupamentos

O objetivo da tarefa de determinação de agrupamentos (*clustering*) é dividir automaticamente um conjunto de objetos em grupos

(*clusters*) de acordo com algum tipo de relacionamento de similaridade existente. A organização dos objetos em cada cluster deve ser feita de forma que haja:

- Alta similaridade entre os objetos pertencentes a um mesmo cluster.
- Baixa similaridade entre os elementos que pertencem a clusters diferentes.

Considere, por exemplo, uma base de dados que contém a idade e o salário de diferentes pessoas entrevistadas por uma pesquisa. A partir dessa base, um algoritmo para a determinação de agrupamentos seria capaz de descobrir os três clusters destacados na figura a seguir (considere que cada ponto representa uma pessoa).

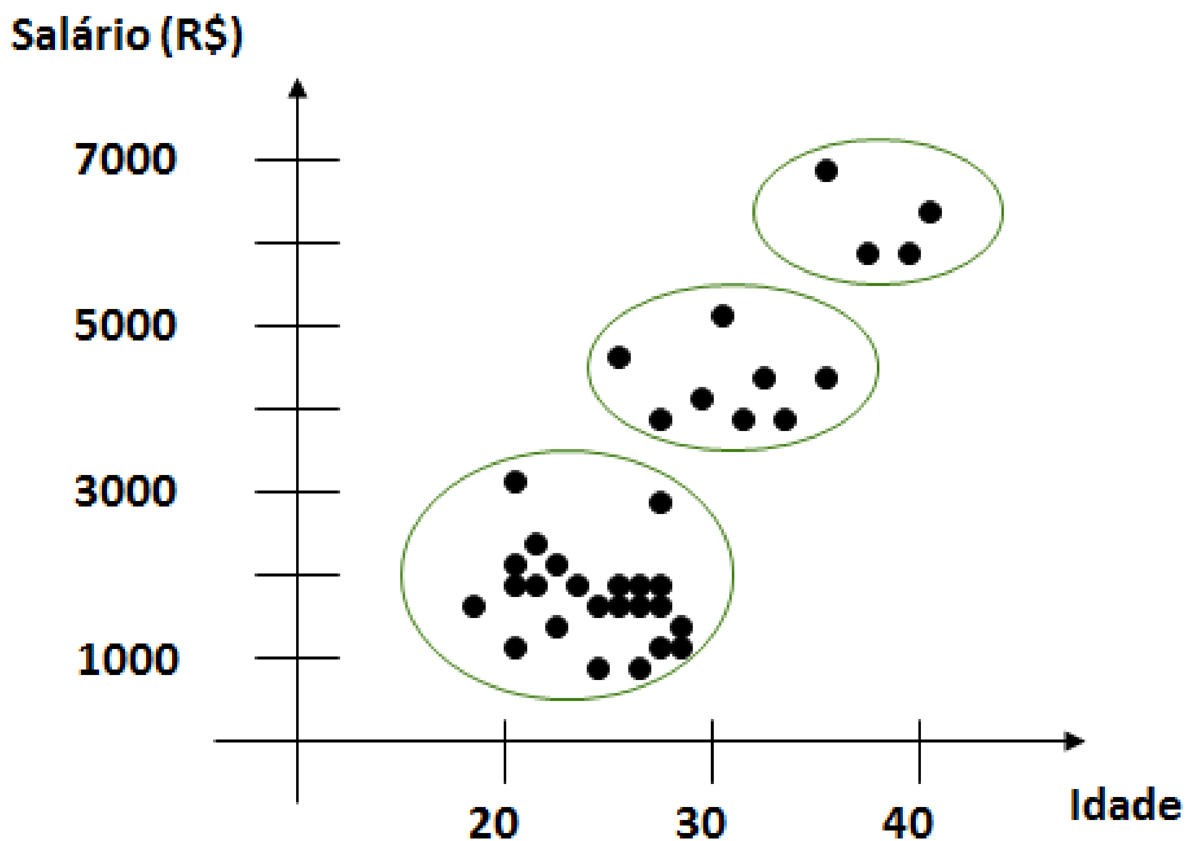

Figura 1.3: Determinação de agrupamentos

O cluster com maior número de objetos é formado por jovens com baixa renda. O segundo é formado por pessoas com idade por volta de 30 anos e renda média. Já o último cluster, que possui menos objetos, é formado por pessoas de renda alta e idade superior a 35 anos.

1.3 Como funciona a ciência de dados na prática?

Em geral, um processo de ciência de dados possui diferentes etapas de execução, conforme mostra o esquema a seguir.

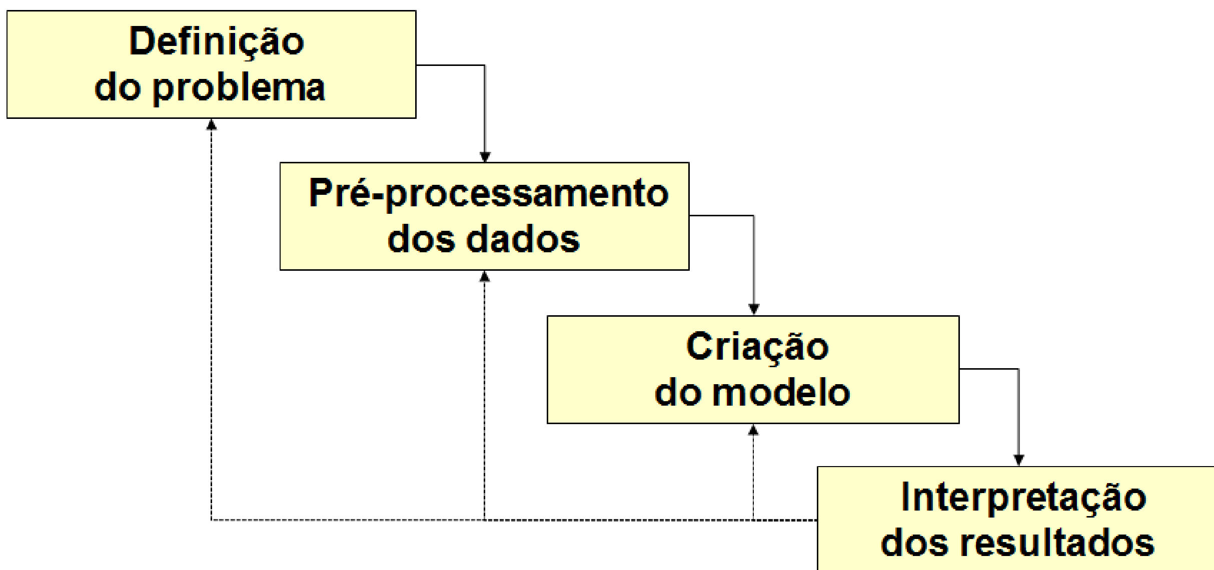

Figura 1.4: Etapas de um típico processo de ciência de dados

Etapa 1 — Definição do problema

Nesta etapa, o objetivo é realizar a modelagem do problema que a empresa deseja resolver utilizando a perspectiva da ciência de dados.

Por exemplo, suponha que o Ministério da Saúde deseje tratar o seguinte problema: "Como reduzir a incidência de doenças cardiovasculares nos brasileiros?". Neste caso, a tradução para um problema de ciência de dados seria: "Quais as características dos brasileiros que possuem doenças cardiovasculares?". Os responsáveis pelo projeto poderiam definir que, como produto final, o processo de ciência de dados gerasse um modelo de classificação capaz de prever a probabilidade de uma pessoa contrair uma doença cardiovascular em função de diversos fatores, como idade, peso, hábitos alimentares, fatores hereditários etc.

Etapa 2 — Pré-processamento dos dados

É subdividida em duas fases: (i) seleção de dados; (ii) limpeza e transformação dos dados. As atividades executadas em ambas as fases são conhecidas como atividades de *data wrangling*, ou *data munging*, que em uma tradução livre significa algo como "brigar com os dados". (Veja mais em https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#7b9e4e6d6f63). Nos parágrafos que se seguem você verá que o verbo "brigar" não é empregado por acaso! A etapa de pré-processamento é, sem dúvida, a mais difícil e trabalhosa em qualquer projeto relacionado à ciência de dados.

Começaremos falando da fase de seleção de dados. Normalmente, os dados de uma empresa se encontram espalhados em diferentes bancos de dados. O sistema de RH possui a sua própria base, o sistema de faturamento possui outra, os dados de uso da internet costumam estar armazenados dentro de arquivos de um ou mais servidores Web (que podem até mesmo não estar fisicamente localizados dentro da empresa). O objetivo da fase de seleção é coletar e reunir todos os dados que sejam relevantes para a resolução do problema de ciência de dados definido.

Se, por exemplo, uma empresa deseja criar um modelo para identificar as características dos consumidores que compram um

determinado produto nas transações realizadas pela internet, poderia ser necessário selecionar as fontes de dados mostradas na figura a seguir:

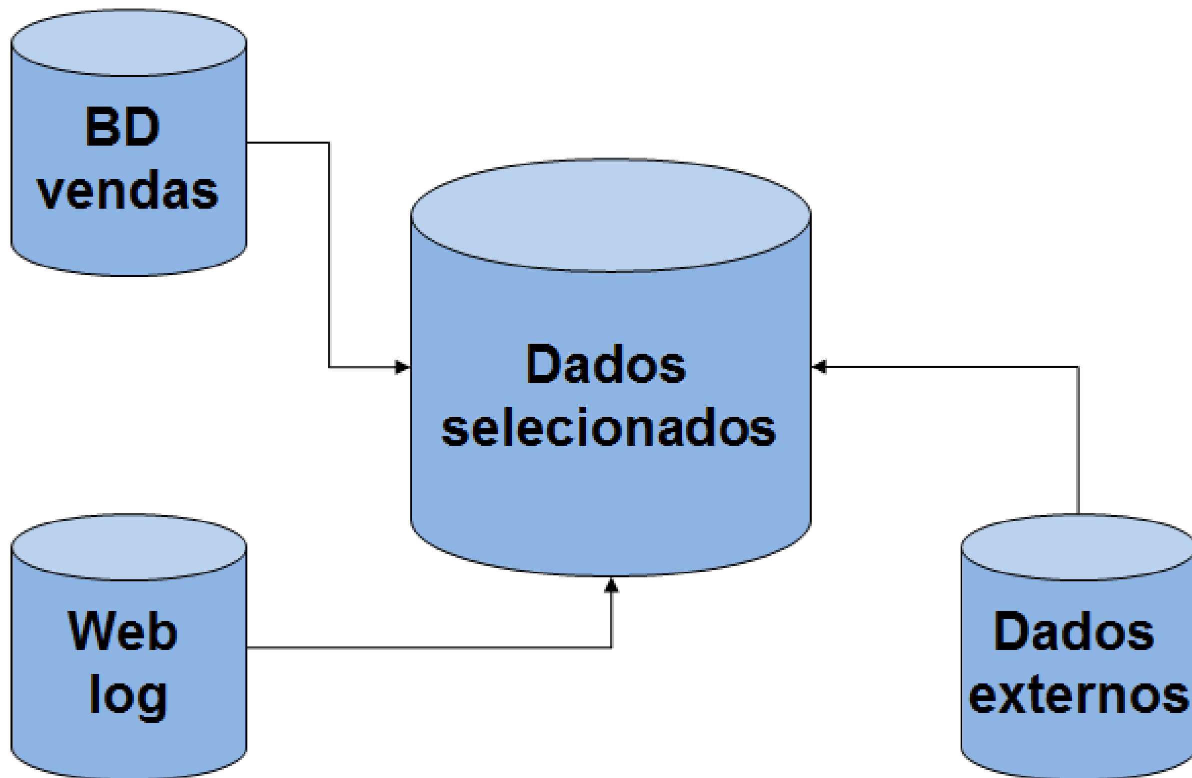

Figura 1.5: Seleção de dados: a primeira etapa da fase de pré-processamento

- Dados do sistema de vendas: base que contém os dados demográficos do cliente (nome, sexo, endereço) e as informações das compras (produtos comprados, forma de pagamento, valor gasto).
- Log de uso do servidor Web: arquivo que armazena a sequência de páginas visitadas pelos clientes em cada sessão.
- Dados externos: muitas vezes pode ser interessante juntar também dados de fontes externas. Por exemplo, incorporar dados do IBGE para obter informações como a renda média e o número de habitantes das cidades dos clientes.

Retornando ao exemplo do Ministério da Saúde ("identificação do perfil das pessoas doenças cardiovasculares"), a fase de seleção

poderia exigir que dados de pessoas atendidas por diversos hospitais da rede pública e privada fossem reunidos. Veja o tamanho do trabalho!

Uma vez realizada a seleção dos dados de interesse, normalmente será preciso realizar a limpeza e a transformação deles. Limpeza significa eliminar sujeira e informações irrelevantes. Transformação consiste em converter os dados de origem para um outro formato, mais adequado para ser usado pelo algoritmo de ciência de dados. Alguns exemplos de situações práticas onde é necessário aplicar estes procedimentos:

- *Múltiplos cadastramentos de um mesmo objeto*. Em uma base de dados que registra as vendas de produtos, muitas vezes existem clientes com o mesmo nome e endereço, mas com identificação diferente. Veja o exemplo na tabela a seguir, onde o cliente "Eduardo Corrêa" está cadastrado duas vezes.

ID Cliente	Nome	Logradouro	Sexo
4003	Eduardo Corrêa	Av. Alfa, 500	M
4006	Glauco Rocha	Rua do Sol, 1234	M
4009	Eduardo Corrêa	Av. Alfa, 500	M
4015	Amanda Pereira	Rua ABC, 999	F
4027	Antônia Silva	Av. Utopia, s/n	F

- *Falta de uniformidade entre as diferentes bases reunidas no processo de seleção*. Por exemplo: alguns arquivos fonte onde atributo Sexo é representado por `M` e `F` e outros onde é Sexo é representado por `1` e `0`.

- *Valores com frequência acima da expectativa*. Por exemplo: muitas pessoas que se cadastraram em um sistema Web declararam morar no bairro 'Água Rasa'. Na verdade,

escolheram a primeira opção do combo no formulário de cadastramento!

- *Dados contraditórios*. Por exemplo: idade 18 anos e escolaridade Doutorado.
- *Valores nulos*. Por exemplo: cliente não preencheu o valor de sua idade. Neste caso será preciso adotar alguma técnica para resolver o problema, como utilizar o valor mais frequente ou médio.
- *Transformar uma variável para formato mais adequado*. Por exemplo: transformar `Data de nascimento` em `Idade`.
- *Normalização de dados*. Por exemplo: mapear todos os valores numéricos para um número entre 0 e 1, para assegurar que os números grandes não dominem os números pequenos durante a execução de um algoritmo de ciência de dados.
- *Agrupar variáveis*. Muitas vezes podem existir muitos valores distintos para uma variável. Estes valores podem ser agrupados em diferentes conjuntos. Por exemplo, a partir da variável `CEP`, poderia ser criada a variável `RegiãoCidade`. No caso da cidade do Rio de Janeiro, a variável `RegiãoCidade` possuiria apenas 4 valores: Centro, Sul, Norte e Oeste.

A FASE MAIS TRABALHOSA

De acordo com a literatura, a fase de pré-processamento de dados costuma ser a mais trabalhosa em qualquer projeto relacionado à ciência de dados, ocupando tipicamente 80% do tempo consumido (https://www.infoworld.com/article/3228245/the-80-20-data-science-dilemma.html).

Etapa 3: Criação do modelo

Trata-se da etapa "nobre" do processo, onde um algoritmo é aplicado sobre os dados que foram selecionados, limpos e transformados na etapa anterior. Como resultado, será extraído um modelo representando padrões de relacionamento identificados no conjunto de dados.

Etapa 4: Interpretação dos resultados

Nesta etapa, os especialistas da empresa avaliam o modelo extraído na etapa anterior. Muitas vezes isso é feito por um time multidisciplinar. Por exemplo, estatísticos podem determinar a validade do modelo. Em seguida, eles se reúnem com as pessoas com conhecimento do domínio do problema (médicos, no caso do exemplo do Ministério da Saúde) para interpretar os resultados considerados válidos e avaliá-los quanto ao seu grau de utilidade.

De acordo com a avaliação, será possível definir o quanto o processo de ciência de dados satisfez o problema definido. Por exemplo: o modelo de classificação gerado consegue prever com elevada acurácia a probabilidade de uma pessoa contrair doenças cardiovasculares? Ainda é preciso aprimorá-lo de alguma forma? A incorporação de uma nova variável poderá aumentar a eficácia do modelo?

Muitas vezes, a primeira rodada de um processo de ciência de dados não é suficiente para produzir um modelo satisfatório. Tornar-se-á, então, necessário refazer alguma etapa do processo. Pode ser preciso retornar para a Etapa 2 e selecionar novas fontes de dados ou realizar novas transformações sobre os dados selecionados. A ciência de dados é, por natureza, um processo cíclico, que usualmente necessita de algumas iterações até que o melhor resultado seja obtido.

1.4 E onde entra a pandas nessa história?

A biblioteca **pandas**, *Python Data Analysis Library* (https://pandas.pydata.org), é um software livre, do tipo *open source*, que ao longo dos últimos anos se consolidou como a biblioteca para ciência de dados mais utilizada no ambiente Python. Trata-se essencialmente de uma biblioteca para *Data Wrangling*, ou seja, uma biblioteca especialmente projetada para oferecer suporte às atividades de seleção, exploração, integração, limpeza e transformação de dados. As funcionalidades oferecidas pela biblioteca consistem basicamente em técnicas para a manipulação de dados tabulares, que se assemelham muito às oferecidas pelo Excel e pela linguagem SQL. Alguns exemplos:

- Importar de forma padronizada dados estruturados de diferentes tipos de fontes, tais como: arquivos texto (CSV, JSON, XML), bancos de dados (relacionais e NoSQL), planilhas eletrônicas etc.;
- Combinar de forma inteligente registros provenientes de diferentes bases de dados (operação conhecida como `merge` ou `join`);
- Produzir resultados agregados e tabulações (`group by`);
- Limpar e transformar bases de dados (ex.: discretização de variáveis, aplicação de filtros sobre linhas e colunas de tabelas, ordenação dos dados, tratamento de dados ausentes etc.).

A biblioteca pandas torna o processo de manipulação de dados tabulares mais rápido, simples e eficaz, acrescentando ao Python duas estruturas de dados (EDs) extremamente poderosas e flexíveis: **Series** e **DataFrame**,

[BR]	[FR]	[UK]	[IT]	[US]
Real	Euro	Libra	Euro	Dólar

Series

	Nome	Peso	Altura	Idade
M01	Ana	95	193	49
M08	David	94	197	52
M13	José	87	202	23
M14	Marina	48	159	34
M20	Renata	33	144	9

Data Frame

Figura 1.6: Estruturas de dados da biblioteca pandas: Series e DataFrame

- Uma Series é uma espécie de array que pode ser indexado através de rótulos em vez de números. No exemplo da figura anterior, mostramos uma Series que armazena dados de países. O rótulo é a sigla do país e o valor o nome de sua moeda oficial. O capítulo 2 deste livro dedica-se às Series.
- A estrutura DataFrame serve para representar dados tabulares similares aos de uma planilha Excel ou de uma tabela de banco de dados relacional. O DataFrame mostrado na figura anterior mantém informações básicas de cinco pessoas. Os DataFrames são o tema principal deste livro! Mostraremos como criar, popular, transformar, limpar e explorar DataFrames nos capítulos 3 a 6.

1.5 Projeto prático — apresentação

Neste livro, pretendemos ensinar o leitor a utilizar a pandas através de uma abordagem que combina a teoria com um projeto prático. O projeto em questão envolve o uso da pandas como ferramenta principal para realizar a seleção, limpeza e transformação de dados provenientes de uma base de dados chamada **flags dataset** (https://archive.ics.uci.edu/ml/datasets/Flags). Esta base mantém informações sobre várias nações e suas bandeiras (por exemplo: cores e figuras presentes nas bandeiras).

Mostraremos o passo a passo para realizar o pré-processamento da base `flags`, que será então utilizada como fonte para a criação de um modelo de classificação de dados. O objetivo do modelo será inferir as cores da bandeira de um país em função de suas características. Maiores detalhes serão apresentados ao longo dos próximos capítulos.

Mas, antes de prosseguir, é necessário primeiro ter certeza de que você o instalou localmente a pandas em seu ambiente Python. Caso você tenha instalado o Python a partir de alguma distribuição voltada para ciência de dados como a **Anaconda** (https://www.anaconda.com/distribution) ou **WinPython** (http://winpython.sourceforge.net), então você não precisa se preocupar, pois a pandas já estará automaticamente disponível. Já se você instalou o Python a partir da **distribuição oficial** (https://www.python.org), será preciso instalar a pandas em seu computador. A forma simples para fazer isso é através do utilitário `pip`. Para usá-lo, é preciso abrir uma janela de terminal (no caso do Windows, "prompt de comandos") e digitar:

```
pip install -U pandas
```

A biblioteca pandas será então baixada e instalada. Caso ela já esteja instalada, será atualizada por uma versão mais nova (se houver). Antes de utilizá-la em qualquer programa Python, é sempre preciso importá-la através do comando `import`. Neste livro, adotaremos a notação apresentada a seguir para realizar essa tarefa, em que o apelido `pd` é atribuído para a pandas:

```python
import pandas as pd
```

Agora que você já sabe o que é pandas, conhece a sua importância e, até mesmo, instalou a biblioteca em seu ambiente Python, que tal iniciarmos a nossa viagem? Os próximos capítulos deste livro foram cuidadosamente preparados para que você se transforme em uma verdadeira fera na utilização desse pacote!

CAPÍTULO 2
A estrutura de dados Series

Neste capítulo, iniciamos a nossa jornada pelo mundo da programação pandas apresentando a ED **Series**, estrutura básica e fundamental do pacote. Se você pretende utilizar a pandas de forma profissional, é muito importante que você aprenda a trabalhar muito bem com esta ED, compreendendo as suas propriedades e dominando os tipos de operações que podem ser realizadas sobre dados que estejam nela estruturados. O capítulo aborda os seguintes temas:

- Criação e propriedades básicas
- Técnicas para consulta e modificação de dados
- Computação vetorizada
- Índices datetime
- Indexação hierárquica

2.1 Como criar Series?

Series é uma ED composta por um vetor de dados e um vetor associado de rótulos, este último chamado de índice (*index*). Ou seja: **Series = vetor de dados + vetor de rótulos**.

A figura 2.1 mostra dois exemplos de Series. Considere que ambas armazenam dados provenientes de um sistema acadêmico utilizado por uma instituição de ensino hipotética. A primeira Series chama-se `notas`, cujos rótulos são inteiros e os dados valores reais (notas médias dos alunos de um dado curso). A segunda chama-se `alunos`, cujos rótulos são strings (matrículas dos alunos do curso) e os dados também strings (nomes).

[0]	[1]	[2]	[3]	[4]	
7.6	5.0	8.5	9.5	6.4	notas

[M02]	[M05]	[M13]	[M14]	[M19]	
Bob	Dayse	Bill	Cris	Jimi	alunos

Figura 2.1: Dois exemplos de Series contendo dados de alunos.

O programa a seguir apresenta o código para criar ambas as Series utilizando o método construtor padrão. Tanto neste exemplo, como nos demais a serem apresentados ao longo do livro, mostra-se primeiro o código do programa e imediatamente depois, o resultado de sua execução. Todos os exemplos foram elaborados para funcionar de forma independente, então nenhum programa depende da execução de algum outro que tenha sido apresentado previamente. Desta forma, para testar o código de qualquer programa exemplo em seu computador, basta que você o digite e execute no ambiente Python de sua preferência.

```
#P01: Hello Series!
import pandas as pd

#cria a Series notas
notas = pd.Series([7.6, 5.0, 8.5, 9.5, 6.4])

#cria a Series alunos
lst_matriculas = ['M02','M05','M13','M14','M19']
lst_nomes = ['Bob','Dayse','Bill','Cris','Jimi']
alunos = pd.Series(lst_nomes,index=lst_matriculas)

#imprime as duas Series
print(notas); print("---------------"); print(alunos)
```

Veja a saída do programa:

```
0    7.6
1    5.0
2    8.5
3    9.5
4    6.4
dtype: float64
---------------
M02      Bob
M05    Dayse
M13     Bill
M14     Cris
M19     Jimi
dtype: object
```

Observe que, para criar a Series `notas`, foi necessário apenas definir uma **lista** com a relação de notas finais. Uma vez que não especificamos um índice para esta Series, a pandas utilizará inteiros de 0 a N-1 (onde N é o tamanho da lista, neste exemplo, N=5). Por outro lado, ao criar `alunos`, além da lista de valores (nomes dos alunos), foi também especificada uma lista de índices (matrículas dos alunos), passada para o método construtor com o uso do parâmetro `index`.

Existem outras formas para criar Series. Um exemplo é apresentado no trecho de código seguinte, onde a Series é criada a partir de um **dicionário**. Neste caso, as chaves do dicionário são automaticamente transformadas em rótulos.

```
dic_alunos = {'M02':'Bob','M05':'Dayse','M13':'Bill',
              'M14':'Cris','M19':'Jimi'}
alunos = pd.Series(dic_alunos)
```

Listas e dicionários são EDs nativas da linguagem Python, ou seja, elas fazem parte da linguagem padrão (ao contrário da Series, que é específica do pacote pandas). Se você tem pouca experiência em Python e não tem intimidade com essas estruturas, não se preocupe, pois aqui vai uma breve explicação sobre ambas.

Uma lista é uma **sequência ordenada de elementos**, cada qual associado a um número responsável por indicar a sua posição (índice). O primeiro índice de uma lista é sempre 0, o segundo, 1, e assim por diante. Para criar uma lista, basta especificar uma sequência de valores entre colchetes `[ ]`, onde os valores devem estar separados por vírgula. Por exemplo, `['BR','FR','US']` é a lista com as siglas dos países Brasil (posição 0), França (posição 1) e Estados Unidos (posição 2).

Por sua vez, um dicionário é uma ED em que os **elementos são pares chave:valor**. A chave (*key*) identifica um item e o valor armazena seu conteúdo. Qualquer valor armazenado pode ser recuperado de forma extremamente rápida através de sua chave. Para criar um dicionário, você deve utilizar chaves `{ }` e, dentro delas, especificar uma relação de elementos do tipo par *chave:valor*, separados por vírgula. Por exemplo, `{'BR':'Real', 'FR': 'Euro', 'US':'Dólar'}` é um dicionário onde a chave é a sigla de um país e o valor, o nome de sua moeda.

Existem duas diferenças fundamentais entre os dicionários e as listas. A primeira é que, em uma lista, os índices que determinam a posição dos elementos precisam ser inteiros, enquanto em um dicionário os índices podem ser não apenas inteiros, mas também de qualquer tipo básico, como strings. A segunda diferença encontra-se no fato de que em um dicionário não existe o conceito de ordem, ou seja, ele é uma coleção não ordenada de pares chave:valor.

A partir dessas definições, podemos concluir que a Series une "o melhor dos dois mundos". Isso porque, assim como uma lista, a Series armazena uma sequência ordenada de elementos; no entanto, ao mesmo tempo, permite que cada elemento também seja acessado de forma simples e rápida através de um rótulo (similar ao acesso por *keys* inerente aos dicionários).

Quais as propriedades elementares das Series?

Você deve ter notado que, além de ter mostrado os índices e valores de cada Series, o comando `print()` do programa anterior também exibiu o `dtype` delas. Trata-se de uma das propriedades básicas das Series, que corresponde ao tipo dos elementos do vetor de dados. O `dtype` de `notas` é `float64`, que é utilizado para armazenar números reais com dupla precisão (64 bits). Por sua vez, o `dtype` de `alunos` é `object`, utilizado pela pandas para armazenar dados alfanuméricos (strings). Vale a pena deixar claro que o vetor de dados de uma Series sempre conterá valores do mesmo tipo, ou seja, com o mesmo `dtype`.

A tabela a seguir apresenta os cinco principais `dtypes` da pandas. Na primeira coluna mostramos o nome do `dtype`, na segunda, o tipo de dado que ele armazena e na terceira, o nome do tipo equivalente no Python padrão.

dtype	Utilização	Tipo Python
int64	números inteiros	int
float64	números reais	float
bool	True/False	bool
object	texto	str
datetime64	data/hora	datetime

É necessário ressaltar que `dtype` não é a mesma coisa que tipo do objeto (*type*)! O tipo de qualquer objeto Python representa a classe deste objeto e pode ser sempre obtido com o uso da função `type()`. Se você aplicar a função `type()` sobre qualquer Series, sempre receberá como resposta o tipo `pandas.core.series.Series`. Outra consideração importante é que toda Series está associada a outras propriedades elementares além do `dtype`. São elas:

- `values`: vetor de dados;
- `index`: vetor de rótulos;
- `name`: nome do vetor de dados;

- `size` : tamanho da Series (número de elementos);
- `index.name` : nome do vetor de rótulos;
- `index.dtype` : `dtype` do vetor de rótulos.

No programa seguinte, apresentamos a forma para trabalhar com essas propriedades.

```
#P02: Propriedades básicas das Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#atribui nomes p/ os vetores de dados e rótulos
alunos.name = "alunos"
alunos.index.name = "matrículas"

#recupera e imprime as propriedades
print(alunos)
print('-----------')
tamanho = alunos.size
dados   = alunos.values
rotulos = alunos.index
alunos_tipo = type(alunos)
alunos_dtype = alunos.dtype
alunos_idx_dtype = alunos.index.dtype

print('número de elementos: ', tamanho)
print('vetor de dados: ', dados)
print('vetor de rótulos: ', rotulos)
print('tipo (type): ', alunos_tipo)
print('dtype da Series:', alunos_dtype)
print('dtype do vetor de rótulos:', alunos_idx_dtype)
```

Saída do programa:

```
matrículas
M02      Bob
M05    Dayse
```

```
M13     Bill
M14     Cris
M19     Jimi
Name: alunos, dtype: object
-----------
número de elementos:  5
vetor de dados:  ['Bob' 'Dayse' 'Bill' 'Cris' 'Jimi']
vetor de rótulos:  Index(['M02', 'M05', 'M13', 'M14', 'M19'],
dtype='object', name='matrículas')
tipo (type):  <class 'pandas.core.series.Series'>
dtype da Series: object
dtype do vetor de rótulos: object
```

Inicialmente, o programa mostra como nomear tanto o vetor de dados como o de rótulos da Series `alunos`, utilizando as propriedades `name` e `index.name`, respectivamente. Veja que quando mandamos imprimir a Series estes nomes passam a ser exibidos. Em seguida, mostramos como recuperar e imprimir as demais propriedades.

2.2 Técnicas para consulta e modificação de dados

No início do capítulo, vimos que a Series é uma ED que mistura características das listas e dos dicionários. Sendo assim, não é surpreendente que a forma para consultar e modificar dados de Series seja bem parecida com a utilizada por estas EDs. As próximas subseções abordam este tema.

Indexação

Utilizamos colchetes `[ ]` para indexar (acessar e obter) elementos de uma Series. A pandas permite o emprego de três diferentes técnicas: indexação tradicional, fatiamento e indexação booleana.

Nos parágrafos a seguir, essas técnicas são explicadas através de exemplos baseados nas Series mostradas na figura 2.1.

Vamos começar pela **indexação tradicional**, a mais simples de todas. Esta técnica de indexação deve ser utilizada quando você quiser recuperar apenas **um elemento** da Series. Para fazer a indexação tradicional, você deve especificar um número inteiro que corresponde ao índice do elemento que você deseja acessar. Se você especificar um número negativo, a pandas fará a indexação de trás para frente, isto é, -1 recupera o último elemento, -2, o penúltimo etc. Caso a sua Series possua um vetor de rótulos associado (como é o caso de `alunos`), você também poderá indexá-la pelos rótulos (como 'M13', 'M02' etc.). Na tabela a seguir, são apresentados diversos exemplos:

Exemplo	Resultado	Explicação
`alunos[0]`	Bob	primeiro aluno
`alunos[1]`	Dayse	segundo aluno
`alunos['M14']`	Cris	aluno de matrícula 'M14'
`alunos[alunos.size-1]`	Jimi	último aluno
`alunos[-1]`	Jimi	outra forma de pegar o último aluno

A técnica de indexação baseada em **fatiamento** (*slicing*) deve ser utilizada quando você quiser recuperar **mais de um elemento** da Series. Você pode fatiar de duas diferentes formas:

- Por intervalos (*ranges*) definidos por dois pontos `:`;
- Por listas.

A tabela seguinte apresenta a sintaxe da operação de fatiamento, onde as cinco primeiras linhas referem-se ao fatiamento por

intervalos e a última, ao fatiamento com listas. Na notação utilizada, considere que `s` é o nome de uma Series.

Sintaxe	Explicação
`s[i:j]`	do elemento de índice `i` ao de índice `j-1`
`s[i:]`	do elemento de índice `i` até o último
`s[:j]`	do primeiro elemento ao de índice `j-1`
`s[-k:]`	últimos `k` elementos
`s[i:j:k]`	do elemento `i` ao `j-1`, utilizando o passo `k`
`s[[lista]]`	todos os elementos especificados na lista

Veja alguns exemplos:

Exemplo	Resultado
`alunos[0:2]`	{M02:Bob, M05:Dayse}
`alunos[2:4]`	{M13:Bill, M14:Cris}
`alunos[:2]`	{M02:Bob, M05:Dayse}
`alunos[2:]`	{M13:Bill, M14:Cris, M19:Jimi}
`alunos[-2:]`	{M14:Cris, M19:Jimi}
`alunos[1:5:2]`	{M05:Dayse, M14:Cris}
`alunos[[2,0,4]]`	{M13:Bill, M02:Bob, M19:Jimi}
`alunos[['M13','M02','M19']]`	{M13:Bill, M02:Bob, M19:Jimi}

Caso você esteja em dúvida com relação aos três últimos exemplos (que são realmente menos intuitivos!), aqui vai uma explicação detalhada. Quando fazemos `alunos[1:5:2]`, estamos pedindo o seguinte para a pandas: a partir do elemento de índice `i=1` até o elemento de índice anterior a `j=5` (ou seja, elemento de índice 4),

recupere todos os elementos pulando os índices de 2 em 2 (`k=2`). Por este motivo, o resultado inclui `M05:Dayse` (índice 1) e também `M14:Cris` (índice 3).

Já no exemplo `alunos[[2,0,4]]` (fatiamento com lista), estamos pedindo para a pandas recuperar os elementos de índice 2, 0 e 4, **nesta ordem**. De maneira análoga, `alunos[['M13','M02','M19']]` (outro exemplo de fatiamento através de lista) faz com que sejam recuperados os alunos de rótulo 'M13', 'M02' e 'M19', nesta ordem. Veja que a lista especificada deve possuir colchetes, isto é, o certo é usar a notação `alunos[[2,0,4]]` e não `alunos[2,0,4]` .

Uma importante diferença entre as operações de indexação e fatiamento diz respeito ao tipo do resultado retornado por cada uma das operações:

- A indexação tradicional sempre retorna um único elemento, cujo tipo será o tipo básico Python correspondente ao `dtype` do vetor de dados. Por exemplo, `alunos[0]` retorna uma string (tipo str) e `notas[0]` , um float.
- A operação de fatiamento sempre retorna uma Series, ou seja, um objeto do tipo `pandas.core.series.Series` .

Para encerrar a subseção, vamos falar sobre a **indexação booleana**, a terceira e última técnica para indexar Series. No presente momento, realizaremos apenas uma introdução ao assunto, que será revisitado e mais bem detalhado em capítulos posteriores. Neste modo de indexação, subconjuntos de dados são selecionados com base nos **valores da Series** (valores do vetor de dados) e não em seus rótulos/índices.

No próximo programa, apresentamos um exemplo em que a indexação booleana é aplicada para determinar os nomes de todos os alunos com nota igual ou superior a 7.0. Explicações detalhadas são apresentadas após o código.

```
#P03: Indexação booleana
import pandas as pd
```

```
#cria as Series "notas" e "alunos"
notas = pd.Series([7.6, 5.0, 8.5, 9.5, 6.4])
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#obtém os índices dos alunos aprovados
idx_aprovados = notas[notas >= 7].index

#imprime os alunos aprovados
print('relação de alunos aprovados:')
print('---------------------------')
print(alunos[idx_aprovados])
```

Saída:

```
relação de alunos aprovados:
---------------------------
M02    Bob
M13    Bill
M14    Cris
dtype: object
```

Como esse programa funciona? O maior segredo está no comando `idx_aprovados = notas[notas >= 7].index` . Ele é responsável por gerar um vetor de índices (objeto do tipo `pandas.core.indexes.numeric.Int64Index`) que armazenará os índices de todos os elementos com valor igual ou superior a 7.0 na Series `notas` . Sendo assim, o comando retorna o vetor `[0, 2, 3]` . Ao utilizarmos esse vetor em `alunos[idx_aprovados]` , executa-se a operação de fatiamento de `alunos` , recuperando os seus elementos 0, 2 e 3 (Bob, Bill e Cris, os alunos que tiraram mais de 7.0).

Busca

O próximo programa mostra como verificar se determinados rótulos ou valores estão presentes em uma Series.

```python
#P04: Busca em Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#testa se rótulos fazem parte de uma Series
tem_M13 = 'M13' in alunos
tem_M99 = 'M99' in alunos
print("existe o rótulo 'M13'? -> ",tem_M13)
print("existe o rótulo 'M99'? -> ",tem_M99)
print('---------------------------')

#testa se valor faz parte de uma Series
tem_Bob = alunos.isin(['Bob'])
print("existe o valor 'Bob'")
print(tem_Bob)
```

Resultado:

```
existe o rótulo 'M13'? ->  True
existe o rótulo 'M99'? ->  False
---------------------------
existe o valor 'Bob'
M02     True
M05    False
M13    False
M14    False
M19    False
dtype: bool
```

Para testar se um rótulo existe em uma Series (mais precisamente, se faz parte de seu vetor de rótulos), a coisa é bem simples: basta utilizar o operador `in` do Python padrão (que retornará `True` ou `False`).

Se, por outro lado, o que você deseja é verificar se um ou mais valores estão em uma Series (mais precisamente, se fazem parte de seu vetor de dados), precisará utilizar o método `isin()` da pandas.

Este método recebe como entrada uma lista contendo um ou mais valores (em nosso exemplo, utilizamos uma lista contendo apenas um valor, 'Bob'). Como saída, é gerada uma Series com `dtype bool` que terá o valor `True` para todos os rótulos associados a valores da lista. Veja que em nosso exemplo, apenas 'M2' recebeu `True`, pois este é o rótulo de `alunos` que possui o valor 'Bob'.

Modificação

No próximo exemplo, apresentamos a forma básica para inserir, modificar e excluir elementos.

```
#P05: Inserindo, Alterando e Removendo elementos de Series
import pandas as pd

#cria a Series "alunos"
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

print('Series original:')
print(alunos)

#insere o aluno de matrícula M55, Rakesh
alunos['M55'] = 'Rakesh'

#altera os nomes Bill, Cris e Jimi para Billy, Cristy e Jimmy
alunos['M13'] = 'Billy'
alunos[['M14','M19']] = ['Cristy','Jimmy']

#remove o aluno de matrícula M02 (Bob)
alunos = alunos.drop('M02')

print('---------------------------')
print('Series após as alterações:')
print(alunos)
```

Saída:

```
Series original:
M02      Bob
M05    Dayse
M13     Bill
M14     Cris
M19     Jimi
dtype: object
---------------------------
Series após as alterações:
M05     Dayse
M13     Billy
M14    Cristy
M19     Jimmy
M55    Rakesh
dtype: object
```

Nesse exemplo, primeiro criamos `alunos` com os mesmos dados dos exemplos anteriores. Em seguida, inserimos um novo aluno de matrícula `M55`, denominado `Rakesh`. Para tal, basta utilizar um comando de atribuição comum: `alunos['M55'] = 'Rakesh'`. A modificação também é feita de forma parecida. Podemos modificar um elemento por vez (`alunos['M13'] = 'Billy'`) ou mais de um, neste caso, usando uma lista de rótulos e uma de valores, como foi feito em `alunos[['M14','M19']] = ['Cristy','Jimmy']`. Por fim, mostramos como remover elementos com o uso do método `drop()` da pandas: `alunos = alunos.drop('M02')`. Vale ressaltar que não apenas a modificação, mas também a inclusão e remoção suportam o uso de listas para que seja possível inserir ou remover muitos elementos de uma vez.

Modificação de índices

Além de alterar valores, podemos alterar os índices de uma Series, utilizando a propriedade `index`. Por exemplo, se o comando adiante for executado, o primeiro rótulo de alunos será alterado para 'M91', o segundo, para 'M92', e assim sucessivamente.

```
alunos.index = ['M91','M92','M93','M94','M95']
```

2.3 Computação vetorizada

É possível utilizar a tradicional estrutura `for … in` para iterar sobre uma Series (ou seja, para percorrer "de cabo a rabo" o vetor de dados ou o de rótulos):

```
#P06: Iteração
import pandas as pd
alunos = pd.Series({'M02':'Bob','M05':'Dayse','M13':'Bill',
                    'M14':'Cris','M19':'Jimi'})

#itera sobre os dados (nomes dos alunos)
for aluno in alunos: print(aluno)

#itera sobre os índices (matrículas)
for indice in alunos.index: print(indice)
```

Entretanto, em grande parte das situações práticas, não precisaremos fazer uso desse recurso. Isso porque, em geral, as operações da pandas podem ser executadas através do mecanismo conhecido como **computação vetorizada** (*vectorization*). Neste processo, as operações são realizadas sobre cada elemento da Series automaticamente, sem a necessidade de programar um laço com o comando `for`. Alguns exemplos:

- Se `s` é uma Series com valores numéricos, e fazemos `s * 2`, obteremos como resultado uma Series que conterá todos os elementos de `s` multiplicados por 2.

- Ao efetuarmos uma soma de duas Series `s1` e `s2`, teremos como resultado uma nova Series em que o valor de um rótulo (ou índice) `i` será igual a `s1[i] + s2[i]`. E da mesma forma ocorrerá para todos os demais rótulos.

O programa a seguir demonstra estes conceitos na prática. Uma outra novidade introduzida no exemplo é a importação da biblioteca **NumPy** e a utilização de um de seus métodos, denominado `sqrt()` (explicações detalhadas são apresentadas após o exemplo).

```
#P07: Operações aritméticas com computação vetorizada
import pandas as pd
import numpy as  np

#cria as Series s1 e s2
s1 = pd.Series([2,4,6])
s2 = pd.Series([1,3,5])
print('s1:'); print(s1)
print('s2:'); print(s2)

#efetua as operações aritméticas
print('----------------------------')
print('s1 * 2')
print(s1 * 2)
print('----------------------------')
print('s1 + s2')
print(s1 + s2)
print('----------------------------')
print('raiz quadrada dos elementos de s1')
print(np.sqrt(s1))   #com a Numpy!
```

Observe com calma os resultados gerados:

```
s1:
0    2
```

```
1    4
2    6
dtype: int64
s2:
0    1
1    3
2    5
dtype: int64
---------------------------
s1 * 2
0     4
1     8
2    12
dtype: int64
---------------------------
s1 + s2
0     3
1     7
2    11
dtype: int64
---------------------------
raiz quadrada dos elementos de s1
0    1.414214
1    2.000000
2    2.449490
dtype: float64
```

Neste programa, primeiro realizamos a importação das bibliotecas pandas (como sempre, apelidada de `pd`) e da biblioteca `numpy` (que recebe o apelido de `np`). Logo depois declaramos duas Series `s1` e `s2`. Então, três operações são efetuadas:

- Multiplicação de todos os elementos de `s1` por 2, obtida por `s1 * 2`;
- Soma de `s1` e `s2`, obtida por `s1 + s2`. Veja que a soma é feita entre pares de elementos de `s1` e `s2` que ocupam a mesma posição (compartilham o mesmo índice);
- Raiz quadrada dos elementos de `s1`, obtida por `np.sqrt(s1)`.

Conforme mencionado anteriormente, o cálculo da raiz quadrada foi feito com o uso de um método da NumPy, uma biblioteca direcionada para a computação de alto desempenho sobre arrays (vetores e matrizes). A NumPy é considerada a "pedra fundamental" da computação científica em Python pelo fato de as suas propriedades e métodos terem sido utilizados como base para o desenvolvimento de diversas outras bibliotecas importantes para ciência de dados, como a nossa querida pandas.

De fato, uma Series é na verdade um array NumPy de dados associado a um array de rótulos. Por este motivo, normalmente é possível aplicar qualquer função matemática disponibilizada pela NumPy sobre os dados de uma Series. Para consultar uma relação completa das funções NumPy, veja https://docs.scipy.org/doc/numpy/reference/ufuncs.html.

Se você estiver utilizando uma distribuição Python voltada para ciência de dados, não precisará se preocupar em instalar a NumPy. Caso contrário, poderá instalar a biblioteca através do utilitário `pip` (apresentado no capítulo anterior):

```
pip install -U numpy
```

O valor NaN

A figura a seguir mostra duas Series, denominadas `verde` e `azul`. Considere que a primeira indica se a cor verde faz parte das bandeiras do Brasil (BR), França (FR), Itália (IT) e Reino Unido (UK), onde o valor 1 representa sim e 0, não. Por sua vez, a Series `azul` indica se a cor azul faz parte das bandeiras dos mesmos países anteriores com o acréscimo da Argentina (AR).

[BR]	[FR]	[IT]	[UK]	
1	0	1	0	verde

[AR]	[BR]	[FR]	[IT]	[UK]	
1	1	1	0	1	azul

Figura 2.2: Series verde e azul, ambas contendo informações sobre cores de bandeiras de diferentes nações.

O programa a seguir examina o comportamento da pandas ao realizar uma operação aritmética (soma) envolvendo estas duas Series **que não possuem os rótulos inteiramente compatíveis**.

```
#P08: o valor NaN
import pandas as pd

verde = pd.Series({'BR':1, 'FR': 0, 'IT':1, 'UK': 0})
azul = pd.Series({'AR':1, 'BR':1, 'FR': 1, 'IT':0, 'UK': 1})

soma = verde + azul
print("soma:")
print(soma)
print('---------------------------')

print("isnull(soma):")
print(pd.isnull(soma))
```

Aqui está o resultado:

```
soma:
AR    NaN
BR    2.0
FR    1.0
IT    1.0
UK    1.0
dtype: float64
```

```
---------------------------
isnull(soma):
AR     True
BR    False
FR    False
IT    False
UK    False
dtype: bool
```

Como você deve ter notado, o resultado da soma para os rótulos existentes nas duas Series ('BR', 'FR', 'IT' e 'UK') ficou perfeitinho. No entanto, para o rótulo 'AR', que só existe em `azul`, o resultado da soma aparece como **`NaN`** (*not a number*). Mas o que é isso? Bem, `NaN` é o conceito utilizado pela pandas para marcar valores nulos/ausentes (*missing*) ou desconhecidos.

Em nosso exemplo, como 'AR' existe em `azul`, mas não tem um correspondente em `verde`, a pandas interpreta que o valor de 'AR' em `verde` é desconhecido. E o resultado de qualquer operação aritmética envolvendo um número conhecido e um desconhecido resultará sempre em desconhecido (ou seja, em `NaN`). Parece estranho, mas veja como faz sentido. Se eu fizer para você a seguinte pergunta: "Qual o resultado da soma do número 1 com outro número desconhecido". Certamente, você me responderia: "Eu não sei! Se o segundo valor é desconhecido, como posso responder?". E a pandas faz da mesma maneira: **1 + desconhecido = desconhecido**.

No mesmo programa, também aproveitamos para apresentar mais um método da pandas, denominado `isnull()`. Este método recebe como entrada uma Series `s`, gerando como saída uma outra Series com `dtype` `bool`, que indica quais dos rótulos de `s` estão associados a valores nulos (`NaN`). Existe também o método `notnull()` que faz o oposto, retornando `False` para todos os valores nulos.

Os exemplos apresentados evidenciaram as vantagens da computação vetorizada em processos de Data Wrangling. Por exemplo, no programa recém-apresentado, utilizamos apenas uma

linha de código para implementar um processo de transformação de dados (criamos uma Series a partir de duas outras), ao fazermos `soma = verde + azul`. De maneira análoga, apenas uma linha de código foi suficiente para testarmos por valores ausentes na Series `soma`: `pd.isnull(soma)`.

2.4 Índices datetime

Como o seu próprio nome indica, a estrutura Series foi originalmente projetada para lidar com **séries temporais** (https://www.datacamp.com/community/tutorials/time-series-analysis-tutorial), um dos temas mais estudados no campo da Estatística. Uma série temporal consiste em uma série de dados coletados em incrementos sucessivos de tempo ou algum outro tipo de indicador de sequência. Um exemplo bem simples é apresentado na figura a seguir: uma Series em que cada elemento representa um determinado dia do ano (rótulo) acompanhado da temperatura máxima registrada no dia em questão em uma determinada cidade (dado).

[10/02/2019]	[11/02/2019]	[12/02/2019]	[13/02/2019]	[14/02/2019]	[15/02/2019]
31	35	34	28	27	27

Figura 2.3: Exemplo simples de série temporal.

O programa a seguir mostra como podemos criar essa série temporal, configurando explicitamente o vetor de índices para que ele possua o **tipo data** (`dtype` `datetime64`). Afinal de contas, essa é a semântica correta da nossa série de registro temperaturas!

```
#P09: Índices datetime
import pandas as pd

#(1)-cria a série temporal
```

```
dias = ['10/02/2019', '11/02/2019','12/02/2019','13/02/2019',
        '14/02/2019','15/02/2019']
temp_max = [31,35,34,28,27,27]

serie_temporal = pd.Series(temp_max,index=dias)

#(2)-converte o tipo do índice para datetime e imprime a série
serie_temporal.index = pd.to_datetime(serie_temporal.index,
                                      format='%d/%m/%Y')
print(serie_temporal)
```

Esta é a saída do programa:

```
2019-02-10    31
2019-02-11    35
2019-02-12    34
2019-02-13    28
2019-02-14    27
2019-02-15    27
dtype: int64
```

Veja que no programa os índices foram originalmente passados como uma lista de strings: `dias = ['10/02/2019', '11/02/2019','12/02/2019','13/02/2019','14/02/2019','15/02/2019']`. O truque para transformar os valores deste índice para `datetime` consistiu no emprego do método `to_datetime()` com dois parâmetros:

- O nome do vetor de índices a ser convertido (em nosso caso, `serie_temporal.index`).
- `format='%d/%m/%Y'`: o parâmetro `format` é utilizado para que possamos informar a pandas o formato da data (**date string**). Para indicar este formato, foi preciso utilizar as máscaras `%d` (dia), `%m` (mês) e `%Y` (ano com quatro dígitos) combinadas com o caractere `/`. No mundo Python, essas máscaras são chamadas de diretivas (*directives*). A lista completa das diretivas pode ser consultada em http://strftime.org.

2.5 Indexação hierárquica

A indexação hierárquica é um recurso oferecido pela pandas para permitir que você trabalhe com mais de um **nível de indexação**. Para que o conceito fique claro, o exemplo desta seção mostra como utilizar este recurso para criar uma Series com informações sobre os nomes das moedas dos cinco países mostrados na figura a seguir. Desta vez, a Series poderá ser indexada não apenas pela sigla do país, mas também pelo nome de seu continente.

[América] [AR]	[América] [BR]	[Europa] [FR]	[Europa] [IT]	[Europa] [UK]
Peso	Real	Euro	Euro	Libra

Figura 2.4: Série com dois níveis de indexação.

```
#P10: Índexação hierárquica
import pandas as pd

moedas  = ['Peso', 'Real', 'Euro', 'Euro', 'Libra']
paises = [['América','América','Europa','Europa','Europa'],
          ['AR','BR','FR','IT','UK']]


paises = pd.Series(moedas, index=paises)

print(paises)                    #imprime toda a Series
print('-----------------------')
print(paises['América'])         #{AR: Peso, BR:Real}
print('-----------------------')
print(paises[:,'IT'])            #{Europa: Euro}
print('-----------------------')
print(paises['Europa','IT'])     #Euro
```

Saída:

```
América  AR      Peso
         BR      Real
Europa   FR      Euro
         IT      Euro
         UK     Libra
dtype: object
----------------------
AR    Peso
BR    Real
dtype: object
----------------------
Europa    Euro
dtype: object
----------------------
Euro
```

Neste programa, a Series `paises` foi criada com um vetor de rótulos indexado em dois níveis. No primeiro nível temos o nome do continente ('América' ou 'Europa') e, no segundo, a sigla do país ('AR', 'BR', 'FR', 'IT' e 'UK'). Com isso, torna-se possível indexar os dados de três formas: apenas pelo nível 1, apenas pelo nível 2 ou por ambos. E foi exatamente o que fizemos no programa:

- `paises['América']` : este é um exemplo de indexação pelo nível 1 (nome do continente). O resultado é uma Series contendo as siglas e moedas dos países da América.

- `paises[:,'IT']` : aqui foi realizada a indexação pelo nível 2 (sigla do país). Como resultado, é retornada uma Series com um único elemento, em que o rótulo é o nome do continente e o valor é o nome da moeda da Itália.

- `paises['Europa','IT']` : este é um exemplo de indexação por ambos os níveis. Como resultado, será retornado 'Euro' (valor string), pois este é o valor do elemento cujo nível 1 do índice é 'Europa' e o nível 2 'IT'.

2.6 Projeto prático — mais detalhes

Em um dos exemplos deste capítulo (mais especificamente, o exemplo que abordou o valor `NaN`), trabalhamos com duas Series contendo dados sobre as cores das bandeiras de países. Isso não foi por acaso! Conforme adiantamos no capítulo 1, ao longo deste livro trabalharemos em um projeto prático que envolverá a criação de um classificador a partir de uma base de dados contendo informações sobre diferentes nações. Chegou a hora de fornecermos mais alguns detalhes sobre este projeto (embora a gente só vá começar mesmo a meter a mão na massa a partir do próximo capítulo).

Classificação (mais informações em https://doi.org/10.5753/sbc.7.3) é o problema de ciência de dados que tem o seguinte objetivo: treinar um programa de computador para que ele seja capaz de atribuir automaticamente classes para objetos cujas classes sejam desconhecidas. Um exemplo bem simples e moderno de classificador, porém muito legal para explicar o conceito, é o de um classificador de fotos. Considere um programa que receba como entrada a fotografia do rosto de uma pessoa e que seja capaz de determinar automaticamente se o rosto pertence a um adulto ou a uma criança. Neste caso, o objetivo do programa é associar uma classe ("Adulto" ou "Criança") a um objeto (a fotografia de um rosto). Por isso, ele é um classificador!

No projeto deste livro, estamos propondo o seguinte problema de ciência de dados: criar um classificador que vai tentar "adivinhar" as cores presentes nas bandeiras de um país em função de suas características. O conjunto de cores (classes) possíveis é: {"amarelo", "azul", "branco", "laranja", "preto", "vermelho" e "verde"}. A base de dados que utilizaremos será a `flags dataset` (https://archive.ics.uci.edu/ml/datasets/Flags), uma base pública muito utilizada em artigos sobre algoritmos de Machine Learning.

Para conseguirmos construir o nosso classificador, precisaremos executar a seleção, limpeza e a transformação de diversos atributos

desta base, e faremos isso sempre com o auxílio da biblioteca pandas. No entanto, para que realmente consigamos iniciar este projeto, é preciso primeiro conhecer a principal ED oferecida pela pandas: o DataFrame. Esta ED é tão importante, mas tão importante, que será coberta nos quatro próximos capítulos do livro. Então, avance para a próxima página, pois agora a coisa vai esquentar de vez!

CAPÍTULO 3
A estrutura de dados DataFrame

DataFrame é a ED pandas utilizada para representar dados tabulares em memória, isto é, dados dispostos em **linhas** e **colunas**. Trata-se da ED mais importante para ciência de dados, responsável por disponibilizar um amplo e sofisticado conjunto de métodos para a importação e o pré-processamento de grandes bases de dados. São tantos métodos, que precisaremos de quatro capítulos para apresentá-los! Para começar, o presente capítulo cobre os seguintes tópicos:

- Como criar DataFrames?
- Técnicas para consulta e modificação de dados
- Trabalhando com arquivos

Adicionalmente, na seção final do capítulo veremos como empregar DataFrames para importar e explorar a base de dados **flags**, iniciando o processo de construção do classificador de cores de bandeiras.

3.1 Como criar DataFrames?

O DataFrame é uma ED especialmente projetada para tornar o processo de manipulação de **dados tabulares** mais rápido, simples e eficaz. A figura a seguir apresenta um exemplo: o DataFrame `paises`, que possui cinco linhas e quatro colunas e armazena informações sobre cinco diferentes nações.

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

Figura 3.1: Exemplo de DataFrame contendo dados de países.

Em sua aparência, o DataFrame é igual a uma planilha Excel, uma vez que possui linhas e colunas. No entanto, considerando a forma como a pandas organiza os DataFrames, cada coluna é, na verdade, uma Series. Mais especificamente, o DataFrame é um **dicionário de Series**, todas do mesmo tamanho (`size`). Tanto as suas linhas como as suas colunas podem ser indexadas e rotuladas.

No DataFrame `paises` as linhas são indexadas pela sigla do país. Este DataFrame possui as seguintes colunas: `nome` (nome do país), `continente` (nome do continente onde se localiza o país), `extensão` (extensão territorial em milhares de quilômetros quadrados) e `corVerde` (indica se a cor verde está presente na bandeira do país; o valor 1 representa sim e 0, não). O programa a seguir, apresenta o código para criar este DataFrame utilizando o método construtor padrão.

```
#P11: Hello DataFrame!
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
```

```
                    'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]
         }

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#imprime o DataFrame
print(paises)
```

Veja a saída do programa:

```
           nome continente  extensao  corVerde
AR    Argentina    América      2780         0
BR       Brasil    América      8511         1
FR       França     Europa       644         0
IT       Itália     Europa       301         1
UK  Reino Unido     Europa       244         0
```

Há várias formas para criar DataFrames. Neste exemplo, fizemos a partir de um **dicionários de listas** chamado `dados`. Observe que cada chave do dicionário foi transformada em uma coluna. As listas associadas às diferentes chaves possuem todas o mesmo tamanho (cinco elementos) e foram utilizadas para estabelecer os dados de cada coluna. Veja ainda que, em nosso exemplo, além de passar os dados do DataFrame, também criamos uma lista de índices de linhas (`siglas`), que foi passada para o método construtor com o uso do parâmetro `index`.

Todo DataFrame é um objeto do tipo `pandas.core.frame.DataFrame` que possui as seguintes propriedades básicas:

- `shape` : formato do DataFrame, ou seja, o seu número de linhas (`shape[0]`) e de colunas (`shape[1]`);
- `index` : lista com os rótulos das linhas;
- `columns` : lista com os rótulos das colunas;
- `dtypes` : retorna uma Series com os `dtypes` de cada coluna;

- `index.dtype` : `dtype` dos rótulos das linhas.

O programa listado a seguir mostra como trabalhar com estas propriedades.

```
#P12: Propriedades básicas dos DataFrames
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#recupera e imprime as propriedades
print('-----------')
num_linhas = paises.shape[0]
num_colunas = paises.shape[1]
indices = paises.index
colunas = paises.columns
paises_tipo = type(paises)
paises_dtypes = paises.dtypes
paises_idx_dtype = paises.index.dtype

print('número de linhas: ', num_linhas)
print('número de colunas: ', num_colunas)
print('rótulos das linhas: ', indices)
print('rótulos das colunas: ', colunas)
print('tipo (type): ',paises_tipo)
print('dtypes das colunas:\n', paises_dtypes)
print('dtype dos rótulos das linhas:', paises_idx_dtype)
```

Saída do programa:

```
número de linhas:  5
número de colunas:  4
rótulos das linhas:  Index(['AR', 'BR', 'FR', 'IT', 'UK'], dtype='object')
rótulos das colunas:  Index(['nome', 'continente', 'extensao',
'corVerde'], dtype='object')
tipo (type):  <class 'pandas.core.frame.DataFrame'>
dtypes das colunas:
 nome          object
continente    object
extensao       int64
corVerde       int64
dtype: object
dtype dos rótulos das linhas: object
```

O SHAPE DE UM DATAFRAME

A forma de trabalhar com as propriedades dos DataFrames é muito parecida com a das Series. No entanto, a maneira de obter o número de linhas e colunas, com o uso da propriedade `shape` , pode causar alguma confusão. O `shape` de um DataFrame corresponde ao seu formato e é representado como uma tupla bidimensional (uma tupla é uma ED praticamente igual a uma lista). O número de linhas é armazenado em `shape[0]` e o de colunas, em `shape[1]` .

3.2 Técnicas para consulta e modificação de dados

Indexação

Utilizamos colchetes `[ ]` para indexar elementos de um DataFrame. Assim como ocorre com as Series, é possível empregar três técnicas de indexação: indexação tradicional, fatiamento e indexação booleana. Nesta seção, abordaremos as duas primeiras, deixando a indexação booleana para a seção final do capítulo.

No quadro adiante, mostramos um resumo das opções disponíveis para indexação básica de células (operações que retornam um único elemento do DataFrame). Na notação utilizada, considere que `d` é o nome de um DataFrame em memória. Veja que a pandas disponibiliza quatro diferentes métodos para esta operação: `iloc()`, `iat()`, `loc()` e `at()`. Os dois primeiros são utilizados para indexação baseada na posição da linha (número inteiro), enquanto os dois últimos são para acessar linhas através de seus rótulos.

Sintaxe	Explicação
`d.iloc[i][j]`	retorna o valor da célula que ocupa a linha i, coluna j
`d.iat[i,j]`	retorna o valor da célula que ocupa a linha i, coluna j
`d.iloc[i]['col']`	retorna o valor da célula que ocupa a linha i, coluna denominada 'col'
`d.loc['idx'][j]`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna j
`d.loc['idx']['col']`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna denominada 'col'
`d.at['idx','col']`	retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna denominada 'col'

Algumas observações importantes:

- No vocabulário adotado pela pandas, o termo **index** é sempre utilizado para índices das linhas, enquanto o termo **column** é utilizado para os índices das colunas.
- Não esqueça que a primeira coluna está na posição 0, a segunda, na posição 1 etc. Da mesma forma, a primeira linha está na posição 0, a segunda, na posição 1 etc.

Veja a seguir alguns exemplos de indexação de células do DataFrame `paises`:

Exemplo	Resultado
`paises.iloc[1][0]`	Brasil
`paises.iat[1,0]`	Brasil
`paises.iloc[3]['corVerde']`	1
`paises.loc['IT'][1]`	Europa
`paises.loc['FR']['nome']`	França
`paises.at['FR','nome']`	França

O fatiamento de DataFrames pode ser realizado com o uso dos métodos `iloc()` (por posição) e `loc()` (por rótulo). Na tabela a seguir, são apresentadas as diferentes sintaxes que podem ser empregadas para fatiar colunas ou linhas inteiras. Em todos os casos, o resultado será retornado em objeto do tipo Series.

Sintaxe	Explicação
`d['col']`	retorna a coluna de nome 'col' (toda a coluna)
`d.col`	outra forma para retornar a coluna de nome 'col'
`d.loc['idx']`	retorna a linha associada ao índice de rótulo 'idx' (linha inteira)
`d.iloc[i]`	retorna a linha que ocupa a posição i (linha

Sintaxe	Explicação
	inteira)

Observe os exemplos:

Exemplo	Resultado
`paises['extensao']`	{AR: 2780, BR: 8511, FR: 644, IT: 301, UK: 244}
`paises.corVerde`	{AR: 0, BR: 1, FR: 0, IT: 1, UK: 0}
`paises.loc['BR']`	{nome: Brasil, continente: América, extensao: 8511, corVerde: 1}
`paises.iloc[2]`	{nome: França, continente: Europa, extensao: 644, corVerde: 0}

Agora serão apresentados exemplos de fatiamentos capazes de "recortar" pedaços do DataFrame que sejam diferentes de toda uma linha ou coluna. Neste caso, as "fatias" de linhas e colunas desejadas devem ser definidas dentro de colchetes `[ ]`, com o uso do operador dois-pontos (`:`) e separadas por vírgula (`,`). Primeiro deve-se definir a fatia de linhas e depois a de colunas. Para que o conceito fique claro, as figuras seguintes apresentam, de maneira visual, o resultado obtido por três diferentes operações de fatiamento sobre o DataFrame `paises` :

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

paises.iloc[:3, :2]

Figura 3.2: Primeiro exemplo de fatiamento.

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

paises.loc[['BR','IT'],'corVerde']

Figura 3.3: Segundo exemplo de fatiamento.

	nome	continente	extensão	corVerde
AR	Argentina	América	2780	0
BR	Brasil	América	8511	1
FR	França	Europa	644	0
IT	Itália	Europa	301	1
UK	Reino Unido	Europa	244	0

paises.iloc[-2:,1:3]

Figura 3.4: Terceiro exemplo de fatiamento.

Explicação:

- `paises.iloc[:3, :2]` : retorna uma fatia contendo as três primeiras linhas e duas primeiras colunas.
- `paises.loc[['BR','IT'],'corVerde']` : para as linhas rotuladas com `'BR'` e `'IT'` , retorna o valor armazenado na coluna `corVerde` .
- `paises.iloc[-2:,1:3]` : retorna uma fatia contendo as duas últimas linhas e as colunas de índice 1 e 2.

Você deve ter notado que a técnica para fatiar DataFrames é muito similar à que é empregada no fatiamento de Series. O que muda é apenas o fato de que você precisa especificar uma fatia para linhas e outra para colunas. Com relação ao tipo de objeto retornado, temos que o resultado do fatiamento de um DataFrame será sempre um objeto do tipo DataFrame, exceto quando a operação resultar em uma única linha ou uma única coluna retornada. Neste caso, o tipo de objeto resultante é uma Series.

Busca

O próximo programa mostra:

1. Como verificar se um determinado rótulo de linha ou coluna existe em um DataFrame;
2. Como verificar se um determinado valor está armazenado em alguma coluna do DataFrame.

```
#P13: Busca em DataFrames
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#testa se um dado rótulo de linha existe
tem_BR = 'BR' in paises.index
tem_US = 'US' in paises.index
print("existe o rótulo 'BR? -> ",tem_BR)
print("existe o rótulo 'US'? -> ",tem_US)
print('---------------------------')

#testa se um dado rótulo de coluna existe
tem_corVerde = 'corVerde' in paises.columns
tem_corAzul = 'corAzul' in paises.columns
print("existe o rótulo 'corVerde? -> ",tem_corVerde)
print("existe o rótulo 'corAzul'? -> ",tem_corAzul)
print('---------------------------')

#testa se valor faz parte de uma coluna
```

```
tem_Brasil = paises['nome'].isin(['Brasil'])
print("existe o valor 'Brasil' na coluna 'nome'?")
print(tem_Brasil)
```

Saída do programa:

```
existe o rótulo 'BR? ->  True
existe o rótulo 'US'? ->  False
---------------------------
existe o rótulo 'corVerde? ->  True
existe o rótulo 'corAzul'? ->  False
---------------------------
existe o valor 'Brasil' na coluna 'nome'?
AR    False
BR     True
FR    False
IT    False
UK    False
Name: nome, dtype: bool
```

Para verificar se um rótulo de linha ou de coluna existe em um DataFrame, você deve aplicar o operador `in` sobre a lista de rótulos de linha ou de colunas, respectivamente (propriedades `index` e `columns`).

Se você quiser testar se um valor está armazenado em uma coluna, precisará utilizar o método `isin()` . Em nosso exemplo, passamos para o método uma lista com um único elemento (`'Brasil'`) e mandamos checar a coluna `nome` . Como resultado, o método indicou `True` para a sigla `BR` .

Modificação

Este tópico apresenta as técnicas básicas para inserir e alterar linhas de um DataFrame e para modificar o conteúdo de alguma célula. Em capítulos posteriores voltaremos a abordar o assunto, introduzindo técnicas mais avançadas, capazes de realizar a transformação de valores de colunas inteiras.

```python
#P14: Modificação de DataFrame
import pandas as pd

#cria o DataFrame
dados = {'nome': ['Argentina','Brasil','França','Itália',
                  'Reino Unido'],
         'continente': ['América','América','Europa','Europa',
                        'Europa'],
         'extensao': [2780,8511,644,301,244],
         'corVerde': [0,1,0,1,0]}

siglas = ['AR','BR','FR','IT','UK']

paises = pd.DataFrame(dados,index=siglas)

#insere o país Japão (JP)
paises.loc['JP']= {'nome': 'Japão',
      'continente': 'Ásia',
      'extensao': 372,
      'corVerde': 0}

#altera a extensão do Brasil
paises.at['BR','extensao']=8512

#remove a Argentina e o Reino Unido
paises = paises.drop(['AR','UK'])

print('DataFrame após as alterações:')
print(paises)
```

A seguir, a saída do programa, que imprime `paises` após três diferentes operações de modificação: inserção do país Japão, alteração da extensão territorial do Brasil (de 8511 para 8512) e remoção dos países associados às siglas `'AR'` e `'UK'`.

```
DataFrame após as alterações:
      nome continente  extensao  corVerde
BR  Brasil    América      8512         1
FR  França     Europa       644         0
```

```
IT   Itália     Europa       301         1
JP    Japão       Ásia       372         0
```

- Para inserir o Japão, bastou indicar os dados desse país em um dicionário e realizar a atribuição com o uso do método `loc()`. Caso o DataFrame não possuísse rótulos de linha, seria preciso utilizar o método `iloc()` com o índice da linha a ser inserida.
- Para alterar uma célula, no caso, a extensão do Brasil, utilizou-se um comando de atribuição simples que empregou o método `at()` com a indicação dos rótulos de linha e coluna da célula a ser alterada. Caso o DataFrame não possuísse rótulos de linha, seria preciso utilizar o método `iat()` com o índice da linha a ser alterada.
- Por fim, para remover linhas, basta utilizar o método `drop()` indicando a lista de rótulos de linha a serem removidos.

3.3 Trabalhando com arquivos

Tipicamente, os dados dos DataFrames são obtidos a partir de arquivos ou tabelas de banco de dados. Nesta seção, serão apresentadas as técnicas básicas para importar dados destas fontes para DataFrames pandas. Os exemplos utilizam pequenas bases de dados que se encontram disponíveis no **repositório de bases de dados** de nosso livro. O endereço do repositório é: https://github.com/edubd/pandas.

Importação de arquivo CSV

Podemos realizar a leitura de arquivos CSV (*comma-separated values* — valores separados por vírgula) e de outros tipos de arquivos baseados em **caracteres delimitadores** utilizando o método `read_csv()`. Um exemplo de arquivo deste tipo é `paises.csv`, listado a seguir (note que o arquivo contém os mesmos dados usados nos exemplos anteriores):

```
sigla,nome,continente,extensao,corVerde
AR,Argentina,América,2780,0
BR,Brasil,América,8511,1
FR,França,Europa,644,0
IT,Itália,Europa,301,1
UK,Reino Unido,Europa,244,0
```

Suponha que este arquivo esteja armazenado na pasta `C:\bases` de seu computador (utilizaremos essa suposição para todos os exemplos de agora em diante — modifique o código se você preferir usar outra pasta). A seguir, apresenta-se um programa que importa o conteúdo do arquivo para um DataFrame, definindo ainda que a coluna `sigla` será utilizada como índice de linha.

```
#P15: Importação de CSV padrão para um DataFrame
import pandas as pd

paises = pd.read_csv("c:/bases/paises.csv",index_col="sigla")
print(paises)
```

Simples, não? A saída do programa demonstra que a importação foi perfeita:

```
              nome  continente  extensao  corVerde
sigla
AR       Argentina     América      2780         0
BR          Brasil     América      8511         1
FR          França      Europa       644         0
IT          Itália      Europa       301         1
UK     Reino Unido      Europa       244         0
```

Nesse caso, o programa ficou bem pequeno porque o formato do arquivo de entrada estava idêntico ao que método `read_csv()` espera como padrão: os dados estão separados por vírgula e o arquivo possui cabeçalho. O único parâmetro que utilizamos foi `index_col` , para permitir que a coluna `sigla` fosse transformada no índice de linha.

O método `read_csv()` é extremamente flexível, possuindo uma série de parâmetros que podem ser utilizados para permitir a importação de arquivos CSV estruturados de diferentes maneiras. Os principais parâmetros são relacionados e explicados a seguir:

- `sep` : caractere ou expressão regular utilizada para separar campos em cada linha;
- `skiprows` : número de linhas no início do arquivo que devem ser ignoradas;
- `skip_footer` : número de linhas no final do arquivo que devem ser ignoradas;
- `encoding` : padrão de codificação do arquivo. A codificação default da pandas é **utf-8**. Se o seu arquivo estiver codificado no formato ANSI, você deverá utilizar `encoding='ansi'` . Para obter uma lista completa dos encodings, consulte https://docs.python.org/3/library/codecs.html#standard-encodings;
- `header` : número da linha que contém o cabeçalho (default=0). Se não houver cabeçalho, deve-se especificar `header=None` ou passar uma lista de nomes através do parâmetro `names` ;
- `names` : permite especificar uma lista de nomes para as colunas;
- `index_col` : permite que uma das colunas seja transformada em índice de linhas;
- `na_values` : sequência de valores que deverão ser substituídos por `NA` . Útil para transformação de dados;
- `thousands` : definição do separador de milhar, por exemplo `.` ou `,` ;
- `squeeze` : caso o arquivo CSV possua apenas uma coluna, é possível fazer com que ele seja importado para uma Series em vez de um DataFrame, bastando para isso especificar `squeeze=True` .

A seguir, serão apresentados exemplos que demonstram a utilização destes parâmetros. Vamos começar pelos mais utilizados na prática: `sep` e `names` . Será apresentado um programa capaz de importar e formatar o arquivo `notas.csv` , cujo conteúdo armazena as matrículas

de quatro alunos e as notas por eles obtidas em duas diferentes provas.

```
M0012017;9.8;9.5
M0022017;5.3;4.1
M0032017;2.5;8.0
M0042017;7.5;7.5
```

Veja que o arquivo usa ponto e vírgula (`;`) como separador e não possui uma linha de cabeçalho. Para importá-lo e definir uma linha de cabeçalho, basta fazer da seguinte forma:

```
#P16: Importação de CSV sem cabeçalho e com ";" como separador
import pandas as pd

notas = pd.read_csv("c:/bases/notas.csv", sep=";",
                     names=['matricula','nota1','nota2'])
print(notas)
```

Saída:

```
  matricula  nota1  nota2
0  M0012017    9.8    9.5
1  M0022017    5.3    4.1
2  M0032017    2.5    8.0
3  M0042017    7.5    7.5
```

O parâmetro `sep` foi utilizado para definir `;` como separador, enquanto `names` foi empregado para definir os cabeçalhos das colunas. Desta vez, não utilizamos o parâmetro `index_col` e, por consequência, os índices de linha foram definidos como números inteiros. Para que fosse possível colocar matrículas como índices de linha, bastaria ter feito: `index_col='matricula'` .

Para demonstrar a utilização do parâmetro `squeeze` , considere o arquivo `gols.txt` , que armazena informações sobre uma sequência de sete jogos realizados por um time de futebol no mês de junho de 2019. Para cada jogo, registra-se a sua data de realização e o

número de gols que o time marcou (observe que as duas informações são separadas por um espaço em branco).

```
dia gols
05/06/2019 1
09/06/2019 0
16/06/2019 5
19/06/2019 2
23/06/2019 1
27/06/2019 3
30/06/2019 0
```

Veja que esses dados correspondem a uma sequência de valores registrados em incrementos de tempo. Ou seja, eles representam uma série temporal. Sendo assim, nada mais justo do que importá-los para uma Series em vez de um DataFrame. Esta operação é bem simples com a pandas:

```
#P17: Importação de arquivo com série temporal
import pandas as pd

#importa o arquivo para uma Series
serie_gols = pd.read_csv("c:/bases/gols.txt", sep=" ",
                     squeeze=True, index_col=0)

#converte o tipo do índice para datetime e imprime a série
serie_gols.index = pd.to_datetime(serie_gols.index,
                                  format='%d/%m/%Y')
print(serie_gols)
```

Saída:

```
dia
2019-06-05    1
2019-06-09    0
2019-06-16    5
2019-06-19    2
2019-06-23    1
2019-06-27    3
```

```
2019-06-30    0
Name: gols, dtype: int64
```

Desta vez, três parâmetros foram utilizados no método `read_csv()` :

- `sep=" "` : para indicar que o separador é espaço em branco.
- `squeeze=True` : para retornar uma Series em vez de um DataFrame, já que, por padrão, o método `read_csv()` sempre retorna um DataFrame.
- `index_col=0` : para indicar que a primeira coluna do arquivo (`dia`) deverá ser usada para formar o vetor de rótulos da Series. Por consequência, os gols serão usados no vetor de rótulos.

> **MÉTODO `READ_TABLE()`**
>
> A pandas disponibiliza ainda um outro método para a leitura de arquivos separados por delimitador, denominado `read_table()` . Este método possui os mesmos parâmetros de `read_csv()` . A única diferença entre os dois métodos é que o `read_csv()` tem a vírgula `,` como separador padrão, enquanto `read_table()` utiliza a tabulação `\t` .

Importação de planilha Excel

Considere a planilha Excel `capitais.xlsx` , mostrada na figura a seguir. Ela contém a relação das capitais dos estados brasileiros localizados nas regiões Sul, Sudeste e Norte. Para cada capital, indica-se o seu nome (coluna A), região (coluna B) e a população estimada de acordo com o IBGE (coluna C). Os dados foram obtidos no site https://cidades.ibge.gov.br.

	A	B	C
1	capital	região	população
2	Belém	Sudeste	1446042
3	Belo Horizonte	Sudeste	2513451
4	Boa Vista	Norte	326419
5	Curitiba	Sul	1893977
6	Florianópolis	Sul	477798
7	Macapá	Norte	465495
8	Manaus	Norte	2094391
9	Palmas	Norte	279856
10	Porto Alegre	Sul	1481019
11	Porto Velho	Norte	511219
12	Rio Branco	Norte	377057
13	São Paulo	Sudeste	12038175
14	Rio de Janeiro	Sudeste	6498837
15	Vitória	Sudeste	359555

Figura 3.5: Planilha Excel com dados de capitais do Brasil.

Podemos importar esta planilha para um DataFrame pandas de maneira trivial, utilizando o método `read_excel()`.

```
#P18: Importação de planilha Excel
import pandas as pd

cidades = pd.read_excel("c:/bases/capitais.xlsx")
print(cidades)
```

Saída do programa:

```
           capital   região  população
0   Belém            Sudeste    1446042
1   Belo Horizonte   Sudeste    2513451
2   Boa Vista          Norte     326419
3   Curitiba             Sul    1893977
4   Florianópolis        Sul     477798
5   Macapá             Norte     465495
6   Manaus             Norte    2094391
7   Palmas             Norte     279856
8   Porto Alegre         Sul    1481019
9   Porto Velho        Norte     511219
10  Rio Branco         Norte     377057
11  São Paulo        Sudeste   12038175
12  Rio de Janeiro   Sudeste    6498837
13  Vitória          Sudeste     359555
```

Importação de arquivo JSON

JSON (*JavaScript Object Notation*) é um modelo para armazenamento e transmissão de informações no formato texto. Apesar de muito simples, é o mais utilizado por aplicações Web devido à sua capacidade de estruturar informações de forma compacta e autodescritiva. A listagem a seguir apresenta um exemplo de arquivo no formato JSON. O arquivo, denominado `notas.json`, contém as matrículas de quatro alunos e as notas por eles obtidas em duas provas (os dados são os mesmos do arquivo `notas.csv`, apresentado previamente, porém agora estruturados no formato JSON).

```
[
    {
```

```
        "matricula": "M0012017",
        "notas": [9.8, 9.5]
    },
    {
        "matricula": "M0022017",
        "notas": [5.3, 4.1]
    },
    {
        "matricula": "M0032017",
        "notas": [2.5, 8.0]
    },
    {
        "matricula": "M0042017",
        "notas": [7.5, 7.5]
    }
]
```

O arquivo lembra um pouco um dicionário, no sentido de que também é formado por chaves e valores. Neste exemplo, as chaves são `matricula` e `notas`. A primeira está associada a um valor String (matrícula do aluno). Já a segunda chave é um pouco mais interessante, pois está associada a um **vetor** de números reais, contendo duas notas. É exatamente aí que está o poder do JSON! Ao contrário do formato CSV, o JSON consegue representar não apenas dados tabulares, mas também dados complexos, como vetores, listas, hierarquias etc.

Neste livro, não apresentaremos uma explicação detalhada sobre o padrão JSON, pois trata-se de um assunto bastante abrangente e que não é específico da linguagem Python. No entanto, se você não conhece muito sobre o tema, a sugestão é consultar o tutorial introdutório disponibilizado em https://www.devmedia.com.br/json-tutorial/25275. O que será coberto aqui é a receita básica para importar dados de arquivos JSON para DataFrames pandas. Essa receita consiste em dois passos:

- Importar o arquivo JSON para um objeto em memória, utilizando o pacote `json` do Python padrão.

- Transferir os dados do objeto para um DataFrame.

```
#P19: Importação de arquivo JSON
import pandas as pd
import json

#(1)-Importa o arquivo JSON para memória
with open("c:/bases/notas.json") as f:
    j_notas = json.load(f)

#(2)-Transfere as informações para um DataFrame
notas = pd.DataFrame(j_notas,
                     columns=['matricula','notas'])

print(notas)
```

Saída do programa:

```
  matricula       notas
0  M0012017  [9.8, 9.5]
1  M0022017  [5.3, 4.1]
2  M0032017  [2.5, 8.0]
3  M0042017  [7.5, 7.5]
```

Conforme mencionado anteriormente, o processo tem dois passos:

- Na primeira parte temos a "receita de bolo" para a leitura de arquivos JSON, que consiste em utilizar o método `load()` do pacote `json`. Ao ser chamado, este método faz com que o arquivo JSON inteiro seja carregado para um objeto em memória (que no nosso programa, foi chamado de `j_notas`).
- Na segunda parte, transfere-se o conteúdo do objeto JSON em memória para um DataFrame. É preciso passar duas informações para o construtor padrão: o objeto JSON (`j_notas`) e a relação de chaves do objeto que desejamos mapear para colunas no DataFrame (utilizando o parâmetro `columns`).

Veja que, como resultado, foi gerado um DataFrame que é "fiel" ao formato original do arquivo JSON. Ele possui duas colunas,

`matricula`, do tipo String, e `notas`, do tipo **lista de `floats`**. Interessante, não é? Um DataFrame suporta a definição de colunas que armazenam tipos complexos, como listas e outras EDs.

Gravação de arquivos

Se você quiser gravar o conteúdo de um DataFrame em memória para um arquivo CSV, deverá utilizar o método `to_csv()`.

```
#P20: Salva o conteúdo de um DataFrame em um CSV.
import pandas as pd

#cria o DataFrame
dados = {'codigo': [1001,1002,1003,1004,1005],
         'nome': ['Leite','Café','Biscoito','Chá',
                  'Torradas']
         }

produtos = pd.DataFrame(dados)

#salva o seu conteúdo para um arquivo
produtos.to_csv("C:/bases/produtos.csv", sep="\t", index=False)
```

Neste exemplo, realiza-se a gravação do arquivo `produtos.csv` a partir do conteúdo do DataFrame `produtos`. Nos parâmetros do método `to_csv()`, a tabulação `\t` foi adotada como caractere delimitador (parâmetro `sep`) e a opção `index=False` foi utilizada para evitar que os rótulos dos índices (números inteiros) fossem gravados no arquivo. O formato do arquivo produzido é este:

```
codigo    nome
1001    Leite
1002    Café
1003    Biscoito
1004    Chá
1005    Torradas
```

Se você quiser exportar o DataFrame para uma planilha Excel, poderá utilizar o método `to_excel()`. Sua forma básica de utilização

é bem simples, como mostra-se a seguir:

```
produtos.to_excel("C:/bases/produtos.xlsx", index=False)
```

Para obter a relação com as opções avançadas do método `to_excel()`, consulte o link: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html.

3.4 Projeto prático — importação e filtragem do dataset flags

Agora que você sabe o básico sobre DataFrames, chegou a hora de aplicar o conhecimento adquirido em nosso projeto prático. Nesta seção, construiremos um programa que importará a base de dados `flags` para um DataFrame. Em seguida, a técnica de indexação booleana será empregada para que seja possível determinar quais são os países que, além do Brasil, possuem presentes em sua bandeira nacional as cores verde, amarelo, azul e branco.

Como já foi introduzido nos capítulos anteriores, a base de dados `flags` mantém informações sobre nações e suas bandeiras. Mais especificamente, trata-se de um arquivo CSV contendo 194 linhas, onde cada linha armazena os dados de um país específico. Cada país é descrito por nada menos do que 30 atributos, que são relacionados e descritos a seguir. Optamos por manter os nomes originais dos atributos (nomes em Inglês), uma vez que esta base é muito utilizada em artigos sobre Machine Learning escritos neste idioma.

1. `name` : nome do país.
2. `landmass` : código do continente.
    - 1=América do Norte e América Central;
    - 2=América do Sul;

    - 3=Europa;
    - 4=África;
    - 5=Ásia;
    - 6=Oceania.
3. `zone` : quadrante geográfico.
    - 1=NE;
    - 2=SE;
    - 3=SW;
    - 4=NW.
4. `area` : área em milhares de quilômetros quadrados.
5. `population` : população em milhões de pessoas.
6. `language` : idioma predominante.
    - 1=Inglês;
    - 2=Espanhol;
    - 3=Francês;
    - 4=Alemão;
    - 5=Eslavo;
    - 6=Outro idioma Indo-Europeu;
    - 7=Chinês;
    - 8=Árabe;
    - 9=Japonês/Turco/Finlandês/Húngaro;
    - 10=Outro.
7. `religion` : religião predominante.
    - 0=Católica
    - 1=Outra Religião Cristã
    - 2=Muçulmana;
    - 3=Budista;
    - 4=Hindu;
    - 5=Étnica;
    - 6=Marxista;
    - 7=Outra.
8. `bars` : número de barras verticais na bandeira.
9. `stripes` : número de barras horizontais na bandeira.
10. `colours` : número de cores distintas presentes na bandeira.
11. `red` : 1, caso a bandeira possua a cor vermelha ou 0, caso não possua.

12. `green` : idem para a cor verde.
13. `blue` : idem para a cor azul.
14. `gold` : idem para as cores dourada ou amarela.
15. `white` : idem para a cor branca.
16. `black` : idem para a cor preta.
17. `orange` : idem para as cores laranja ou marrom.
18. `mainhue` : cor predominante da bandeira.
19. `circles` : quantidade de círculos.
20. `crosses` : quantidade de cruzes verticais.
21. `saltires` : quantidade de cruzes diagonais.
22. `quarters` : quantidade de divisões.
23. `sunstars` : quantidade de desenhos de sol ou estrela.
24. `crescent` : 1 se existe desenho de lua crescente, 0 caso contrário.
25. `triangle` : idem para desenho de triângulo.
26. `icon` : idem para imagem de objeto (por exemplo, um barco).
27. `animate` : idem para imagem de ser vivo ou algo relacionado a um ser vivo (por exemplo, uma águia, árvore ou mão).
28. `text` : idem para letras ou texto na bandeira (ex: "ordem e progresso").
29. `topleft` : cor predominante do canto superior esquerdo.
30. `botright` : cor predominante do canto inferior direito.

O repositório original da base, que contém a documentação e os dados propriamente ditos, pode ser acessado em: https://archive.ics.uci.edu/ml/machine-learning-databases/flags/flag.names. Nele, a versão disponibilizada do dataset `flags` está sem cabeçalho. Por isso, preparamos uma versão com cabeçalho e a disponibilizamos no repositório de arquivos do livro. O nome do arquivo é `flags.csv`. Sugerimos que você baixe esse arquivo para alguma pasta de sua máquina para que possa utilizá-lo neste e nos diversos outros exemplos que serão apresentados em nosso livro.

Finalmente chegou a hora! No próximo exemplo, apresentamos um programa que importa a base de dados `flags` e descobre os nomes

dos países que têm as cores verde, amarelo, azul e branco entre as cores de suas bandeiras.

```
#P21: Quais os países que também têm verde, amarelo, azul
#     e branco entre as cores de sua bandeira nacional?
import pandas as pd

#-------------------------------------------------------
#(1)-Importa a base de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')

#-------------------------------------------------------
#(2)-alguns métodos para obter informações básicas
#-------------------------------------------------------
#imprime as primeiras linhas
print('head():'); print(flags.head())
print('------------------------------------------------')

#imprime as últimas linhas
print('tail():'); print(flags.tail())
print('------------------------------------------------')

#-------------------------------------------------------
#(3)-quem tem verde, amarelo, azul e branco na bandeira
#-------------------------------------------------------
#separa as cores
verde = flags['green']
amarelo = flags['gold']
azul = flags['blue']
branco = flags['white']
soma = verde + amarelo + azul + branco

#gera vetor booleano com True para quem tem as 4 cores
tem_todas = (soma==4)

#imprime os nomes dos países com as quatro cores
print('países com verde, amarelo, azul e branco:')
print(flags[tem_todas.values]['name'])
```

E aqui está o resultado produzido pelo programa:

```
head():
              name  landmass  zone ... text  topleft  botright
0      Afghanistan         5     1 ...    0    black     green
1          Albania         3     1 ...    0      red       red
2          Algeria         4     1 ...    0    green     white
3   American-Samoa         6     3 ...    0     blue       red
4          Andorra         3     1 ...    0     blue       red

[5 rows x 30 columns]
-----------------------------------------------
tail():
              name  landmass  zone ... text  topleft  botright
189  Western-Samoa         6     3 ...    0     blue       red
190     Yugoslavia         3     1 ...    0     blue       red
191          Zaire         4     2 ...    0    green     green
192         Zambia         4     2 ...    0    green     brown
193       Zimbabwe         4     2 ...    0    green     green

[5 rows x 30 columns]
-----------------------------------------------
países com verde, amarelo, azul e branco:
17                       Belize
19                      Bermuda
23                       Brazil
24         British-Virgin-Isles
26                     Bulgaria
33               Cayman-Islands
34     Central-African-Republic
48                     Dominica
56           Falklands-Malvinas
57                         Fiji
71                         Guam
78                    Hong-Kong
116                  Montserrat
135                     Parguay
139                    Portugal
142                     Romania
```

```
151            Soloman-Islands
158                  St-Helena
161                 St-Vincent
176        Turks-Cocos-Islands
182            US-Virgin-Isles
187                  Venezuela
Name: name, dtype: object
```

O programa é dividido em três partes. Na primeira, a base `flags.csv` é importada para um DataFrame chamado `flags`. Como o arquivo CSV original possui cabeçalho e dados separados por vírgula, não foi preciso utilizar nenhum parâmetro do método `read_csv()` para importá-lo.

Na segunda parte, empregamos uma abordagem simples para checar se o arquivo foi importado com sucesso. Ela baseia-se na utilização dos métodos `head()` e `tail()`. O método `head()` serve para exibir as cinco primeiras linhas do DataFrame, enquanto `tail()` faz o mesmo para as cinco últimas linhas. Ambos os métodos também informam a quantidade de colunas do DataFrame. De acordo com a saída apresentada, foi possível conferir que o DataFrame resultante da importação do arquivo `flags.csv` foi gerado com 30 colunas (o que confere com o original). O método `head()` nos informa que que o primeiro país (índice 0) é "Afghanistan", enquanto `tail()` nos informa que o último (índice 193) é "Zimbabwe" (logo o DataFrame foi gerado com 194 linhas, conferindo com o arquivo-fonte). Simples e prático, não é mesmo?

Na terceira e mais importante parte do programa, são identificados todos os países que têm verde, amarelo, azul e branco entre as cores da bandeira nacional. O programa funciona da seguinte forma. Primeiro utilizamos o fatiamento para gerar quatro Series, chamadas `verde`, `amarelo`, `azul` e `branco`, contendo todos os dados das colunas relacionadas a estas respectivas cores (ou seja, a coluna completa com 194 elementos). Lembre-se de que essas colunas são binárias (para cada país, o valor 1 é armazenado se a cor faz parte da bandeira ou 0 caso contrário). Em seguida, utilizamos a

computação vetorizada para gerar uma Series chamada `soma`, cujo conteúdo corresponde à soma `verde + amarelo + azul + branco`. Desta forma, `soma` conterá o valor 4 apenas nos índices associados aos países que possuem as quatro cores. Logo depois, está o comando fundamental do programa: `tem_todas = (soma==4)`. Este comando é responsável por executar o teste lógico `soma==4` (o valor de soma é 4?) sobre todas as linhas de `soma`. Como resultado, gera-se uma Series booleana chamada `tem_todas`, que possuirá o valor `True` associado a todas as linhas de `soma` que possuam o valor 4.

Depois disso tudo fica fácil: basta fazer um fatiamento do DataFrame `flags` utilizando `tem_todas`. O fatiamento funciona da seguinte forma: utilizamos `tem_todas` para fatiar as linhas, fazendo com que só as linhas associadas ao valor `True` sejam impressas. E utilizamos `['name']` para fatiar colunas, ou seja, para retirar apenas esta coluna do DataFrame original. Ao observar a saída, descobrimos que há 22 países com verde, amarelo, azul e branco nas cores da bandeira (Brasil e mais 21 países).

Embora simples (tirando os comentários, não são nem 15 linhas de código), o exemplo evidencia o poder da biblioteca pandas para processar bases de dados. No próximo capítulo, vamos nos aprofundar neste tema, aprendendo a explorar bases de dados através da produção de estatísticas.

CAPÍTULO 4
Conhecendo os seus dados

Agora que você já sabe como importar diferentes tipos de arquivos para DataFrames, é natural que esteja ansioso para aplicar algum algoritmo sobre seus dados e, com isso, construir modelos de Machine Learning capazes de resolver muitos problemas de sua empresa e de seus clientes. Entretanto, em qualquer projeto de ciência de dados de médio ou grande porte, é preciso deixar os seus dados **prontos** antes de aplicar qualquer algoritmo, o que significa efetuar a limpeza e transformação deles.

O primeiro passo para a execução dessa tarefa consiste em realizar a **análise exploratória de dados**, que nada mais é do que **estudar a sua base de dados** e, assim, compreender as suas principais características. O objetivo é descobrir respostas para perguntas como: quantos atributos existem na base de dados? Quais os seus tipos? Como os seus valores estão distribuídos? Será que existe algum *outlier* (valor fora do padrão)?

Este capítulo aborda a análise exploratória de dados na pandas. Mostraremos as ferramentas básicas que a biblioteca oferece para que você possa obter a maior quantidade de informação possível de seus dados. Os seguintes tópicos são abordados:

- Tipos de atributos
- Estatísticas básicas
- Ranqueamento e ordenação
- Tabulações
- Produção de gráficos
- Detecção de outliers

4.1 Tipos de atributos

Considere o DataFrame `pme` (pesquisa mensal de emprego), apresentado na próxima figura. Suponha que ele armazena os dados de seis pessoas entrevistadas por uma pesquisa sobre renda e mercado de trabalho.

renda	*empregos*	*sexo*	*escolaridade*
6,46	1	F	pós-graduação
1,50	1	M	fundamental
0,00	0	F	médio
2,57	1	M	médio
9,90	2	M	superior
6,22	3	F	médio

Figura 4.1: DataFrame contendo os dados dos entrevistados em uma pesquisa sobre a renda dos trabalhadores.

Como aprendemos no último capítulo, qualquer DataFrame organiza dados em linhas e colunas. Cada linha é utilizada para armazenar um **objeto** da base de dados enquanto as colunas são utilizadas para representar os **atributos** que descrevem as características dos objetos. Desta forma, no DataFrame `pme`, cada objeto corresponde a uma pessoa entrevistada, que é descrita por quatro atributos: `renda` (renda mensal do entrevistado em quantidade de salários mínimos), `empregos` (número total de empregos que a pessoa possui), `sexo` ("F" ou "M") e `escolaridade` (último nível de escolaridade inteiramente concluído). Cada um destes atributos é inerentemente

diferente dos outros três, embora alguns compartilhem características similares. Nos próximos parágrafos, examinaremos mais a fundo esta questão, ou seja, nos dedicaremos a identificar os diferentes tipos de atributos e entender as suas principais características.

Vamos começar pelo atributo `renda`, que é **numérico**, isto é, um atributo que pode armazenar uma ampla gama de valores numéricos sobre os quais podemos executar operações como soma, subtração, obtenção de valor médio etc. Ao seu lado, temos o atributo `empregos` que também é numérico, possuindo entretanto características diferentes: ele pode armazenar apenas números inteiros (0, 1, 2, ...). Sendo assim, `empregos` é classificado como um atributo numérico do tipo **discreto**, enquanto `renda` é do tipo **contínuo**. Basicamente, um atributo discreto é aquele que pode assumir um conjunto finito ou infinito contável de valores. E qualquer atributo numérico que não seja discreto é contínuo. Nos DataFrames Python, os atributos discretos são armazenados em colunas cujo `dtype` é `int64` enquanto os contínuos ocupam colunas `float64`.

O terceiro atributo do DataFrame é `sexo`, que pode armazenar apenas as strings "F" e "M". Devido ao fato de esses valores serem categorias, `sexo` é chamado de atributo **categórico**, cujos níveis (*levels*) ou categorias são {"F", "M"}. Ao seu lado, temos o atributo `escolaridade`, cujos níveis são {"fundamental", "médio", "superior", "pós-graduação"}. Observe que este atributo também é categórico, porém os seus níveis possuem uma ordenação natural (por exemplo, "médio" está acima de "fundamental"). Um atributo com estas propriedades é classificado como **ordinal**. Já o atributo `sexo`, cujos níveis não possuem nenhum tipo de ordenação especial, é classificado como do tipo **nominal**. Nos DataFrames, os atributos categóricos, sejam ordinais ou nominais, são normalmente armazenados em colunas cujo `dtype` é `object`. Entretanto, também é possível estruturar dados categóricos em colunas `int64`, bastando para isso mapear as categorias para conjuntos de códigos. Por exemplo, no caso do atributo `escolaridade`, os códigos 1, 2, 3 e 4

poderiam ser utilizados para representar as categorias "fundamental", "médio", "superior" e "pós-graduação", respectivamente.

A figura a seguir apresenta uma taxonomia que caracteriza os tipos de atributos.

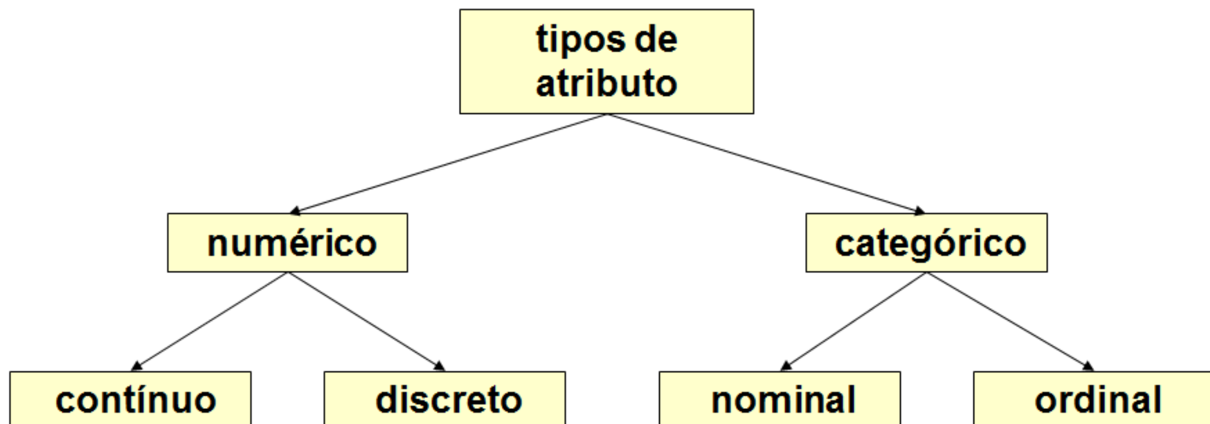

Figura 4.2: Categorização dos diferentes tipos de atributos.

Identificar corretamente os diferentes tipos de atributos de uma base de dados é uma tarefa importante. Isso porque existem medidas e operações diferentes para estudar cada um. Por exemplo, faz sentido utilizar a medida conhecida como moda (valor mais frequente) para estudar atributos categóricos e discretos, mas ela não ajuda muito no estudo de um atributo contínuo. Abordaremos esse assunto em detalhes no próximo tópico. Por enquanto, apresentaremos apenas um programa simples que mostra como percorrer os atributos de um DataFrame e identificar os seus `dtypes` .

```
#P22 Tipos de atributos
import pandas as pd

#cria o DataFrame "pme"
dados = {"renda": [6.46, 1.50, 0.00,
                   2.57, 9.90, 6.22],
         "empregos": [1,1,0,1,2,3],
         "sexo": ["F","M","F","M","M","F"],
         "escolaridade": ["pós-graduação","fundamental",
```

```
                                 "médio","médio",
                                 "superior","médio"]
            }

pme = pd.DataFrame(dados)

#imprime o nome de cada atributo e seu dtype
print("\natributos e seus dtypes:")
print("------------------------")
print(pme.dtypes)
```

Saída:

```
atributos e seus dtypes:
------------------------
renda            float64
empregos           int64
sexo              object
escolaridade      object
dtype: object
```

Neste programa, após criar o DataFrame com os dados da `pme`, fizemos uso da propriedade `dtypes` para investigar o tipo de cada atributo. Recorde que essa propriedade, introduzida no capítulo anterior, retorna uma Series contendo o nome e o `dtype` de cada atributo do DataFrame. Conforme discutimos neste tópico, os diferentes tipos de atributos estarão normalmente associados a `dtypes` específicos da pandas. Veja que, no DataFrame `pme`, os atributos `renda` (contínuo), `empregos` (discreto), `sexo` (nominal) e `escolaridade` (ordinal) foram automaticamente estruturados pela pandas em colunas `float64`, `int64`, `object` e `object`, respectivamente, o que corresponde a um casamento perfeito entre tipos de atributos e `dtypes` pandas.

UM POUCO DE NOMENCLATURA

Os diferentes livros de Estatística, Inteligência Artificial e Ciência de Dados muitas vezes empregam diferentes termos para os conceitos de objeto e atributo. Um objeto (linha do DataFrame) pode também ser chamado de observação, instância, exemplo, registro, tupla ou *data point*. Por sua vez, um atributo (coluna do DataFrame) pode também ser chamado de variável, *feature*, dimensão ou *data field*. Muitas vezes, um conjunto de atributos que descreve um dado objeto é chamado de *attribute vector* ou *feature vector*.

4.2 Estatísticas básicas

As EDs da pandas oferecem diversos métodos para a computação de estatísticas básicas, capazes de determinar as propriedades elementares dos atributos. Alguns dos métodos são empregados para computar medidas de tendência central (como a média aritmética), outros para calcular medidas de dispersão (como o desvio padrão) e há ainda os que são capazes de produzir gráficos (como os *boxplots*). Os próximos tópicos abordam estes assuntos.

Medidas de tendência central

Uma medida de tendência central é aquela capaz de representar o que é **médio** ou **típico** do conjunto de dados. No caso da pandas, você pode entender o termo "conjunto de dados" como a distribuição de valores de um atributo de DataFrame ou de uma Series. Neste tópico, serão apresentadas três medidas de tendência central: **média**, **mediana** e **moda**.

Para introduzir essas medidas, faremos uso de exemplos baseados no DataFrame da próxima figura. Ele registra as infrações cometidas

e punições atribuídas para diversos jogadores que foram expulsos em partidas de um campeonato de futebol, onde a punição é dada em número de jogos do campeonato em que o punido deverá ficar ausente. Observe que os jogadores "Marcelo" e "Pedro" foram expulsos mais de uma vez durante o campeonato.

nome	infracao	punicao
Marcelo	FALTA VIOLENTA	4
Pedro	RECLAMACAO	1
Marcelo	FALTA COMUM	3
Adriano	RECLAMACAO	2
Mauro	FALTA COMUM	4
Pedro	FALTA VIOLENTA	4
Marcelo	FALTA COMUM	2

Figura 4.3: DataFrame com as punições dadas aos jogadores expulsos em um campeonato.

Vamos começar pela **média**, uma medida bem conhecida por qualquer pessoa. Trata-se da soma de cada valor do conjunto de dados, dividida pelo número de observações do conjunto. Em nosso exemplo, os jogadores foram, em média, punidos por 2.86 jogos, como mostra o cálculo a seguir:

média = (4 + 1 + 3 + 2 + 4 + 4 + 2) / 7 = 2,86

Agora vamos falar da **mediana**. A ideia desta medida é separar a distribuição em duas partes iguais. A mediana corresponderá ao

ponto do meio. Em nosso exemplo, se ordenarmos as punições de forma ascendente, teremos o seguinte conjunto de valores: {1, 2, 2, 3, 4, 4, 4}. Nesse caso, o valor 3 é a mediana, pois é ele quem separa a distribuição em duas partes: {1, 2, 2} e {4, 4, 4}. Caso o conjunto de dados possua um número par de valores, você precisará somar os dois "casos médios" e dividir o resultado por dois. Por exemplo, no conjunto de dados {3, 4, 8, 11} a mediana é obtida por (4+8) / 2 = 6.

Já a **moda** consiste no valor mais frequente em uma distribuição. Por exemplo, pesquisas recentes indicam que, na área de ciência de dados, há mais programadores Python do que adeptos de qualquer outra linguagem de programação. Desta forma, Python é a moda entre as linguagens para ciência de dados (veja https://www.kdnuggets.com/2019/03/typical-data-scientist-2019.html). No caso de nosso DataFrame exemplo, a moda é 4, pois a maioria das punições dadas aos jogadores expulsos (3 casos) foi igual a 4 jogos.

Na pandas, os métodos `mean()`, `median()` e `mode()` são utilizados para calcular, respectivamente, os valores da média, mediana e moda. O programa a seguir exemplifica a utilização destes métodos sobre o DataFrame com os dados do campeonato.

```
#P23 Estatísticas básicas: medidas de tendência central
import pandas as pd

#cria o DataFrame
dados = {"jogador": ["Marcelo",
                     "Pedro",
                     "Marcelo",
                     "Adriano",
                     "Mauro",
                     "Pedro",
                     "Marcelo"],
         "infracao": ["FALTA VIOLENTA",
                      "RECLAMACAO",
                      "FALTA COMUM",
```

```
                      "RECLAMACAO",
                      "FALTA COMUM",
                      "FALTA VIOLENTA",
                      "RECLAMACAO"],
         "punicao": [4,1,3,2,4,4,2]
         }

df = pd.DataFrame(dados)

#calcula as medidas de tendência central
print("média:", df['punicao'].mean())
print("mediana:", df['punicao'].median())
print("moda: ", df['punicao'].mode().values)
```

Saída:

```
média: 2.857142857142857
mediana: 3.0
moda:  [4]
```

Veja que, no exemplo, a forma de utilizar os métodos `mean()` e `median()` foi bastante direta: bastou escolher a coluna-alvo (`df['punicao']`) e depois "plugar" o nome do método. Isso porque esses métodos retornam um valor escalar (um número). Já o método `mode()` é um pouquinho mais chato para se trabalhar, pois ele sempre retorna uma Series em vez de um escalar. Isso porque uma distribuição pode ser unimodal (possui uma moda), bimodal (possui duas modas) ou multimodal (apresenta várias modas). Em nosso exemplo, temos apenas uma moda cujo valor é 4; no entanto, mesmo nesse caso o método `mode()` retorna uma Series. Por isso, nosso programa imprime apenas o `values` dessa Series, que contém o valor escalar 4.

Para finalizar, no seguinte quadro comparativo são apresentados os tipos de atributos que podem ser utilizados por cada uma das três medidas.

Medida	Tipos de atributo
média	contínuo e discreto
mediana	contínuo e discreto
moda	discreto, nominal e ordinal

Medidas de variabilidade

Como acabamos de mostrar, média, mediana e moda são medidas que têm por objetivo sintetizar em um único número o que é típico (ou médio) em um conjunto de dados. Porém, em muitas situações práticas estas medidas conseguem fornecer apenas um quadro incompleto dos seus dados. Para exemplificar o que acabamos de afirmar, considere o DataFrame mostrado a seguir, que, assim como no exemplo anterior, mostra as infrações cometidas por diversos jogadores. A diferença é que agora temos duas colunas chamadas `juiz_A` e `juiz_B` que servem para apresentar as punições que foram atribuídas por dois diferentes juízes do tribunal desportivo (neste caso, a punição final poderia ser determinada, por exemplo, considerando a maior das penas).

nome	infracao	juiz_A	juiz_B
Marcelo	FALTA VIOLENTA	4	2
Pedro	RECLAMACAO	1	1
Marcelo	FALTA COMUM	3	4
Adriano	RECLAMACAO	2	1
Mauro	FALTA COMUM	4	1
Pedro	FALTA VIOLENTA	4	5
Marcelo	FALTA COMUM	2	6

Figura 4.4: DataFrame com as punições aplicadas por dois juízes.

Se formos calcular a média de jogos atribuída nas penas de ambos os juízes, o resultado será de 2,86 jogos tanto para o Juiz A como para o Juiz B. Sendo assim, seria possível concluir que os juízes têm um comportamento similar, ou seja, que eles utilizam critérios parecidos para tomar decisões? Ou será que, de fato, o uso da medida da média isoladamente não foi capaz de apresentar um quadro real da situação?

Para obter as respostas a estas perguntas, o melhor a fazer é utilizar uma medida de variabilidade, como será demonstrado nos próximos parágrafos. Serão apresentadas três medidas deste tipo: **amplitude**, **variância** e **desvio padrão**.

Amplitude é uma medida rápida da variabilidade. Ela consiste na diferença entre o mais alto e o mais baixo valor de um determinado conjunto de dados. Em nosso exemplo, a amplitude do Juiz A é igual a 3, pois a sua maior punição foi de 4 jogos e a menor de 1 jogo (4

– 1 = 3). Já a amplitude do Juiz B é igual a 5, pois a sua maior punição foi de 6 jogos e a menor de 1 jogo (6 – 1 = 5). Com isto, já fica claro que a distribuição das punições do Juiz B apresenta uma maior variabilidade do que a do Juiz A.

A amplitude tem a vantagem de ser simples e rápida de calcular. Porém, tem a desvantagem de depender apenas de dois valores de toda a distribuição (o menor valor e o maior valor). Por isso, existirá sempre o risco de ela ser influenciada por um único valor extremo e muito diferente dos demais. A solução é adotar medidas que levem em conta todos os valores da distribuição, como é o caso da variância e do desvio padrão.

Para entendermos a variância, inicialmente precisamos apresentar o conceito de **desvio**, que consiste na distância de um valor arbitrário ao valor médio de um conjunto de dados. Retornando aos dados de nosso DataFrame exemplo, podemos computar os desvios para cada valor dos atributos `juiz_A` e `juiz_B` da forma mostrada a seguir:

```
Desvios - Juiz A
Desvio Pena 1: (4 - 2,86) =  1,14
Desvio Pena 2: (1 - 2,86) = -1,86
Desvio Pena 3: (3 - 2,86) =  0,14
Desvio Pena 4: (2 - 2,86) = -0,86
Desvio Pena 5: (4 - 2,86) =  1,14
Desvio Pena 6: (4 - 2,86) =  1,14
Desvio Pena 7: (2 - 2,86) = -0,86

Desvios - Juiz B
Desvio Pena 1: (2 - 2,86) = -0,86
Desvio Pena 2: (1 - 2,86) = -1,86
Desvio Pena 3: (4 - 2,86) =  1,14
Desvio Pena 4: (1 - 2,86) = -1,86
Desvio Pena 5: (1 - 2,86) = -1,86
Desvio Pena 6: (5 - 2,86) =  2,14
Desvio Pena 7: (6 - 2,86) =  3,14
```

Uma situação que pode ser facilmente observada é que, considerando os valores absolutos (ou seja, ignorando o sinal de negativo), os desvios associados aos *scores* do Juiz B são, em geral, maiores do que os do Juiz A. A variância é uma medida esperta, que não apenas leva esse fato em consideração, como faz isso utilizando todos os *scores* de desvio. Mais especificamente, ela realiza a soma dos quadrados dos desvios e divide o resultado por N-1, que corresponde ao número total de *scores* menos 1 (em nosso exemplo, temos N = 7, logo N – 1 = 6). A ideia de elevar ao quadrado é usada somente para eliminar os sinais negativos de alguns desvios. Veja a seguir o exemplo do cálculo da variância para os juízes A e B:

```
Cálculo da variância das penas do Juiz A
(4 - 2,86)² =  1,30
(1 - 2,86)² =  3,46
(3 - 2,86)² =  0,02
(2 - 2,86)² =  0,74
(4 - 2,86)² =  1,30
(4 - 2,86)² =  1,30
(2 - 2,86)² =  0,74
------------------
soma        =  8,86

variância = (8,86 / 6) = 1,48


Cálculo da variância das penas do Juiz B
(2 - 2,86)² =  0,74
(1 - 2,86)² =  3,46
(4 - 2,86)² =  1,30
(1 - 2,86)² =  3,46
(1 - 2,86)² =  3,46
(5 - 2,86)² =  4,58
(6 - 2,86)² =  9,86
------------------
soma        = 26,86
```

```
variância = (26,86 / 6) = 4,48
```

A variância considera todos os valores da distribuição, oferecendo uma vantagem sobre amplitude, que considera somente dois valores. No entanto, um problema associado a esta medida é a sua interpretação difícil. Como os valores dos desvios são elevados ao quadrado, a unidade original de medida acaba sendo alterada. Em nosso exemplo é alterada de "número de jogos" para "número de jogos ao quadrado". Ou seja, o valor 4,48 para a variância significa "punição de 4,48 jogos ao quadrado".

Corrigir esse problema é muito simples: basta utilizar a medida de desvio padrão. Essa medida corresponde a nada mais do que a raiz quadrada da variância. Ela é usada exatamente para colocar o valor da variabilidade na unidade original. Ou seja, o desvio padrão das penas do Juiz A é dado por raiz de 1,48 = 1,22 jogo. Para o Juiz B temos raiz de 4,48 = 2,12 jogos. A interpretação que podemos dar para estes valores é a seguinte: em média, as penas atribuídas pelo Juiz A se afastam da média por apenas 1,22 jogo. Já no caso do Juiz B, em média, suas penas se afastam da média por mais de 2 jogos (2,12). Isso indica claramente que as penas do Juiz B apresentam maior variabilidade. De fato, se você voltar a observar com calma o DataFrame de nosso exemplo, notará que o Juiz A parece não se importar com jogadores reincidentes, enquanto o Juiz B tende a ser mais rigoroso com aqueles que já haviam sido punidos anteriormente.

Na pandas, podemos calcular a amplitude com o uso dos métodos `min()` e `max()`, que obtêm os valores mínimo e máximo de uma distribuição, respectivamente. Já a variância pode ser computada com o uso do método `var()`, enquanto, para o desvio padrão, utiliza-se o método `std()`. Todos eles são demonstrados no próximo programa.

```
#P24 Estatísticas básicas: medidas de variabilidade
import pandas as pd
```

```python
#cria o DataFrame
dados = {"jogador": ["Marcelo",
                     "Pedro",
                     "Marcelo",
                     "Adriano",
                     "Mauro",
                     "Pedro",
                     "Marcelo"],
         "infracao": ["FALTA VIOLENTA",
                      "RECLAMACAO",
                      "FALTA COMUM",
                      "RECLAMACAO",
                      "FALTA COMUM",
                      "FALTA VIOLENTA",
                      "RECLAMACAO"],
         "juiz_A": [4,1,3,2,4,4,2],
         "juiz_B": [2,1,4,1,1,5,6]
         }

df = pd.DataFrame(dados)

#calcula as medidas de variabilidade
print("Juiz A:")
print("-------")
print("amplitude:", df['juiz_A'].max() - df['juiz_A'].min())
print("variância:", df['juiz_A'].var())
print("desvio padrão:", df['juiz_A'].std())

print("\nJuiz B")
print("-------")
print("amplitude:", df['juiz_B'].max() - df['juiz_B'].min())
print("variância:", df['juiz_B'].var())
print("desvio padrão:", df['juiz_B'].std())
```

Resultados gerados pelo programa:

```
Juiz A:
-------
amplitude: 3
```

```
variância: 1.476190476190476
desvio padrão: 1.2149857925879117

Juiz B
-------
amplitude: 5
variância: 4.476190476190476
desvio padrão: 2.1157009420498154
```

Boxplot

Boxplot é uma técnica gráfica popularmente utilizada para exibir vários aspectos de um conjunto de dados. Para explicar as suas características e apresentar a maneira como podemos gerar este tipo de gráfico na pandas, utilizaremos os dois conjuntos de dados a seguir. Considere que eles armazenam os resultados de um experimento que mediu o tempo de espera, em minutos, de dez diferentes homens e mulheres que discaram para uma linha telefônica de um serviço falso de telepromoções (suponha que o atendente os pedia para aguardar e nunca mais retornava).

```
homens   = [4, 2, 7, 3, 1, 4, 2, 4, 8, 1]
mulheres = [5, 4, 6, 5, 4, 2, 6, 6, 4, 3]
```

A partir desses conjuntos de dados, podemos obter as seguintes estatísticas:

- Para os homens, a média de espera é 3.6 minutos, a mediana 3.5, amplitude 7.0 e desvio padrão 2.37.
- Para as mulheres, tem-se que o tempo médio de espera é de 4.5 minutos, a mediana 4.5, amplitude 4.0 e desvio padrão 1.35.
- Sendo assim, podemos concluir que as mulheres que participaram do experimento foram mais pacientes em termos de tempo médio e mediano. Já os homens acusaram níveis mais diversos de paciência.

Através de um boxplot você consegue visualizar em um mesmo gráfico a média, mediana, desvio padrão e amplitude dos homens e mulheres, o que facilita a tarefa de comparar o comportamento dos dois grupos (afinal de contas, comparar gráficos é uma tarefa mais agradável do que comparar uma série de números). O exemplo a seguir mostra como podemos utilizar o método `boxplot` da pandas para este fim. Explicações detalhadas são apresentadas logo após o programa.

```
#P25 Boxplot - comparação de distribuições
import pandas as pd

df = pd.DataFrame({
        "homens":[4, 2, 7, 3, 1, 4, 2, 4, 8, 1],
        "mulheres": [5, 4, 6, 5, 4, 2, 6, 6, 4, 3]
        })

boxplot = df.boxplot(column=['homens','mulheres'],
                     showmeans=True)
```

Gráfico gerado:

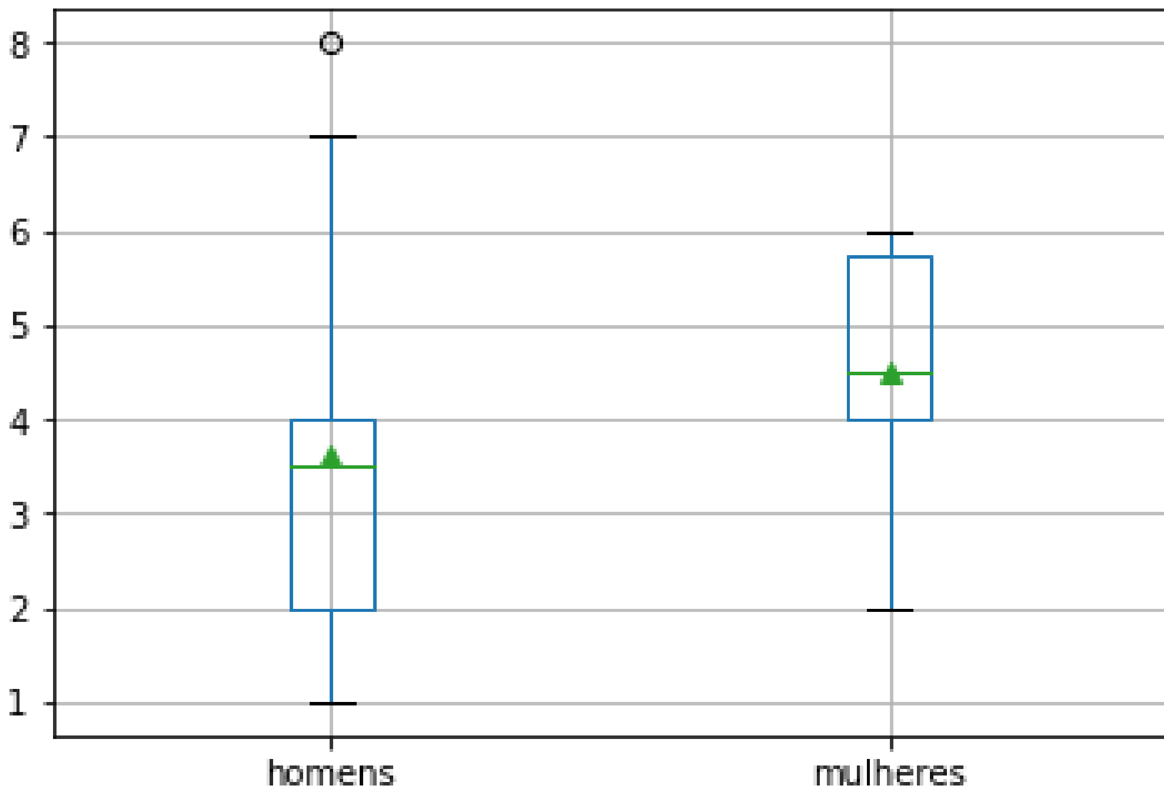

Figura 4.5: Boxplot - tempos de espera dos homens e das mulheres.

Para gerar o boxplot, basta utilizar o método `boxplot()` , especificando uma relação de colunas numéricas (neste exemplo `homens` e `mulheres`). Um boxplot será produzido para cada coluna. Mas como interpretar esse gráfico? A explicação é dada a seguir:

- A caixa retangular dentro do gráfico representa o intervalo entre o primeiro quartil (Q1, linha inferior) e o terceiro quartil (Q3, linha superior). De uma maneira simplificada, podemos dizer que Q1 corresponde à mediana dos valores compreendidos entre o menor valor e a mediana do conjunto de dados. Já Q3 corresponde à mediana dos valores compreendidos entre a mediana e o maior valor do conjunto de dados.
- A discreta reta através da caixa mostra a mediana (3.5 para os homens e 4.5 para as mulheres). A mediana também é chamada de Q2.

- O pequeno triângulo no meio da caixa é a média (3.6 para os homens e 4.5 para as mulheres). É importante observar que, para que a média seja exibida, torna-se preciso utilizar o parâmetro `showmeans=True` .
- As duas linhas fora da caixa (chamadas de *whiskers*) são utilizadas para indicar que os valores localizados fora delas podem ser considerados *outliers* (valores suspeitos).

Ainda neste capítulo, falaremos sobre os outliers e mostraremos outros tipos de gráfico que podem ser gerados com a pandas, tais como gráfico de barras e histogramas.

Eixos

Os exemplos anteriores mostraram como selecionar colunas específicas de um DataFrame para produzir estatísticas sobre elas. No entanto, se você desejar, também poderá produzir estatísticas para **todas** as colunas ou linhas de um DataFrame. Para demonstrar essa funcionalidade da pandas, utilizaremos um exemplo envolvendo o DataFrame `notas` , apresentado na figura seguinte. Considere que ele armazena as notas de quatro alunos ("A1", "A2", "A3" e "A4") em três diferentes provas ("P1", "P2" e "P3"). Ou seja, neste DataFrame cada linha corresponde a um aluno e cada coluna a uma prova.

	P1	P2	P3
A1	9,8	7,2	8,0
A2	5,3	4,0	3,5
A3	5,5	8,1	7,2
A4	7,0	7,5	6,5

Figura 4.6: Notas de quatro alunos em três provas.

Antes de apresentar o exemplo, é importante introduzir o conceito de **eixo** (*axis*). Trata-se, basicamente, de uma propriedade que podemos especificar para decidir se queremos produzir estatísticas por coluna (`axis=0` , que corresponde ao modo padrão) ou por linha (nesse caso, especifica-se `axis=1`). No programa a seguir, essas propriedades foram empregadas para que fosse possível computar automaticamente as médias de cada aluno e de cada prova.

```
#P26 Estatísticas sobre colunas e linhas de um DataFrame
import pandas as pd

#cria uma DataFrame com as notas de 4 alunos em 3 provas
notas = pd.DataFrame({"A1":[9.8, 7.2, 8.0],
                      "A2": [5.3, 4.0, 3.5],
                      "A3": [5.5, 8.1, 7.2],
                      "A4": [7.0, 7.5, 6.5]},
                      index=["P1","P2","P3"])

#imprime o DataFrame
print("\nnotas finais: ")
print('------------------------------')
print(notas)
```

```
#computa e imprime as estatísticas por aluno e prova
print('\nmédia de cada aluno:')
print('-----------------------------')
print(notas.mean())

print('\nmaior nota de cada aluno:')
print('-----------------------------')
print(notas.max())

print('\nmédia de cada prova:')
print('-----------------------------')
print(notas.mean(axis=1))

print('\nmaior nota de cada prova:')
print('-----------------------------')
print(notas.max(axis=1))
```

Saída:

```
notas finais:
-----------------------------
     A1   A2   A3   A4
P1  9.8  5.3  5.5  7.0
P2  7.2  4.0  8.1  7.5
P3  8.0  3.5  7.2  6.5

média de cada aluno:
-----------------------------
A1    8.333333
A2    4.266667
A3    6.933333
A4    7.000000
dtype: float64

maior nota de cada aluno:
-----------------------------
A1    9.8
A2    5.3
A3    8.1
```

```
A4    7.5
dtype: float64

média de cada prova:
-----------------------------
P1    6.9
P2    6.7
P3    6.3
dtype: float64

maior nota de cada prova:
-----------------------------
P1    9.8
P2    8.1
P3    8.0
dtype: float64
```

Quando fazemos `notas.mean()` , a pandas computa de forma automática a média dos valores de todas as colunas ("A1", "A2", "A3" e "A4"), o que corresponde à média de cada aluno. Da mesma forma `notas.max()` computa o maior valor armazenado em cada coluna. Por sua vez, o comando `notas.mean(axis=1)` faz com que sejam geradas as médias dos valores armazenados em cada linha ("P1", "P2" e "P3"), o que corresponde à média de cada prova. De maneira análoga, `notas.max(axis=1)` retorna as notas máximas de cada prova. Em resumo: o padrão da pandas é computar estatísticas por colunas. Para computar por linhas, você deve utilizar o parâmetro `axis=1` .

4.3 Ranqueamento e ordenação

Uma das provas de natação mais apreciadas pelo público é a dos 50m nado livre. Nos Jogos Olímpicos do Rio de Janeiro, em 2016, a

final desta prova na categoria masculino, foi considerada uma das mais empolgantes. A figura a seguir mostra o resultado final dos oito nadadores que a disputaram. Para cada nadador, apresenta-se o número raia na qual ele realizou a prova, o seu nome, nacionalidade e tempo obtido.

raia	*nadador*	*nacionalidade*	*tempo*
1	Simonas Bilis	Lituânia	22.08
2	Benjamin Proud	Grã-Bretanha	21.68
3	Anthony Ervin	Estados Unidos	21.40
4	Florent Manaudou	França	21.41
5	Andriy Hovorov	Ucrânia	21.74
6	Nathan Adrian	Estados Unidos	21.49
7	Bruno Fratus	Brasil	21.79
8	Brad Tandy	África do Sul	21.79

Figura 4.7: DataFrame contendo o resultado da final dos 50m nado livre, masculino, nas Olimpíadas do Rio de Janeiro.

Note que o DataFrame está ordenado pelo número da raia, o que atrapalha a identificação do vencedor da prova! No próximo programa, mostramos como os métodos `sort_values()` e `rank()` podem ser utilizados para nos ajudar a resolver este problema. O primeiro método serve para ordenar o DataFrame e o segundo para produzir um ranqueamento.

```
#P27 Métodos sort_values() e rank()
import pandas as pd

#1-cria o DataFrame da prova de 50m
dados = {"nadador": ["Simonas Bilis",
```

```
                    "Benjamin Proud",
                    "Anthony Ervin",
                    "Florent Manaudou",
                    "Andriy Hovorov",
                    "Nathan Adrian",
                    "Bruno Fratus",
                    "Brad Tandy"],
        "nacionalidade": ["Lituânia",
                          "Grã-Bretanha",
                          "Estados Unidos",
                          "França",
                          "Ucrânia",
                          "Estados Unidos",
                          "Brasil",
                          "África do Sul"],
        "tempo": [22.08,
                  21.68,
                  21.40,
                  21.41,
                  21.74,
                  21.49,
                  21.79,
                  21.79]
        }

raias = list(range(1,9))

prova50m = pd.DataFrame(dados, index=raias)
prova50m.index.name = 'raia'

#2-ordena pelo tempo de forma crescente
prova50m = prova50m.sort_values(by="tempo")
print("* * resultado final ordenado por tempo:")
print(prova50m)

#3 - gera os rankings
resultado_por_raia = prova50m['tempo'].rank(method="min")
print("\n* * posição de cada nadador (por raia):")
print(resultado_por_raia)
```

Observe com atenção a saída do programa:

```
* * resultado final ordenado por tempo:
               nadador   nacionalidade  tempo
raia
3        Anthony Ervin  Estados Unidos  21.40
4     Florent Manaudou          França  21.41
6        Nathan Adrian  Estados Unidos  21.49
2       Benjamin Proud    Grã-Bretanha  21.68
5       Andriy Hovorov         Ucrânia  21.74
7         Bruno Fratus          Brasil  21.79
8           Brad Tandy   África do Sul  21.79
1        Simonas Bilis        Lituânia  22.08

* * posição de cada nadador (por raia):
      tempo
raia
3       1.0
4       2.0
6       3.0
2       4.0
5       5.0
7       6.0
8       6.0
1       8.0
```

Agora uma explicação detalhada sobre o programa. Mais uma vez, ele está dividido em três partes. A primeira trata simplesmente de criar e imprimir o DataFrame. O comando `prova50m.index.name = 'raia'` é usado para atribuir o rótulo `raia` à coluna que armazena o índice.

A segunda parte apresenta o método `sort_values()`. Este método serve para ordenar o DataFrame por uma ou mais colunas, que devem ser especificadas no parâmetro `by`. Neste exemplo, `by="tempo"` foi utilizado para gerar uma ordenação ascendente pela coluna `tempo`. Uma observação muito importante é que se você quiser mudar de fato o DataFrame (ou seja, ordená-lo de verdade

em vez de apenas exibir as suas linhas ordenadas), precisará fazer uso de um comando de **atribuição**:

```
prova50m = prova50m.sort_values(by="tempo")
```

Alguns comentários adicionais sobre o método `sort_values()` :

- A ordenação default é ascendente. Para ordenar de forma descendente, você deve utilizar o parâmetro `ascending=False` .
- Para ordenar por mais de uma coluna, é preciso especificá-las em uma lista. Por exemplo, `by=["tempo","nacionalidade"]` faria com que a pandas realizasse a ordenação por tempo (primeiro critério) e, em casos de empate, por país (segundo critério). Da mesma maneira, para indicar explicitamente o tipo de ordenação de cada campo, use uma lista, como `ascending=[True,False]` .

A terceira e última parte do programa apresenta o método `rank()` que serve para gerar uma Series contendo ranking de valores:

```
resultado_por_raia = prova50m['tempo'].rank(method="min")
```

- O comando apresentado cria o ranqueamento com base nos valores da coluna `tempo` . Veja que a saída do programa mostra que o nadador da raia 3 ficou em primeiro no ranqueamento (menor tempo), o da raia 4 em segundo e assim por diante.
- O parâmetro `method="min"` serve para especificar o procedimento a ser adotado para o tratamento de empates. Neste exemplo, os nadadores das raias 7 e 8 empataram em sexto lugar e `method="min"` faz com que a posição 6 seja atribuída para ambos. Outros valores possíveis seriam `method="max"` (7 seria atribuído para ambos), `method="average"` (colocaria 6.5 em ambos).
- Para gerar um ranqueamento por ordem decrescente de valores, basta empregar o parâmetro `ascending=False` .

4.4 Produzindo tabulações

Esta seção apresenta duas diferentes técnicas para tabular dados a partir de DataFrames.

Domínio e tabelas de frequência

Considere que o DataFrame `vendas`, apresentado na figura a seguir, armazene informações resumidas sobre as vendas efetuadas por uma loja de departamentos da cidade de Belo Horizonte. Cada linha refere-se a uma venda específica (identificada por um `id`), onde se registram o sexo e bairro do cliente (observe que o cliente da transação com `id=5` não possui o bairro cadastrado), o valor da venda e o cartão de crédito utilizado no pagamento.

id	sexo	bairro	valor	cartao
1	F	Belverde	150.00	Master
2	M	Belverde	35.00	Visa
3	F	Savassi	80.00	Visa
4	F	Anchieta	250.00	Amex
5	F		9.90	Elo
6	M	Savassi	25.00	Master

Figura 4.8: DataFrame contendo dados de clientes de uma loja.

Neste DataFrame, `valor` é o único atributo numérico, enquanto `sexo`, `bairro` e `cartao` são categóricos. No programa a seguir introduzimos dois métodos da pandas muito utilizados quando desejamos estudar atributos categóricos como os que acabamos de mencionar. São eles:

- `unique()` : retorna o **domínio** de um atributo do DataFrame, isto é, todas as categorias distintas que ele assume.
- `value_counts()` : gera uma **tabela de frequências** simples para o atributo.

```
#P28 Métodos unique() e value_counts()
import pandas as pd

#1-cria o DataFrame
dados = {"sexo": ["F","M","F","F","F","M"],
         "bairro": ["Belverde",
                    "Belverde",
                    "Savassi",
                    "Anchieta",
                    None,
                    "Savassi"],
         "valor": [150.00,
                   35.00,
                   80.00,
                   250.00,
                   9.90,
                   25.00],
         "cartao": ["Master",
                    "Visa",
                    "Visa",
                    "Amex",
                    "Elo",
                    "Master"]}

id_clientes = [1,2,3,4,5,6]

vendas = pd.DataFrame(dados, index=id_clientes)
```

```
#2-retorna o domínio dos atributos categóricos
print('Domínio dos atributos categóricos:')
print('---------------------------------')
print('sexo:', vendas['sexo'].unique())
print('bairro:', vendas['bairro'].unique())
print('cartao:', vendas['cartao'].unique())

#3-retorna as frequências dos valores de cada coluna
print("\n")
print('Tabelas de frequência:')
print('---------------------------------------')
print("\n1-sexo:")
print(vendas['sexo'].value_counts())
print("\n2-bairro:")
print(vendas['bairro'].value_counts())
print("\n3-cartão:")
print(vendas['cartao'].value_counts())
```

Esta é a saída produzida pelo programa:

```
Domínio dos atributos categóricos:
---------------------------------
sexo: ['F' 'M']
bairro: ['Belverde' 'Savassi' 'Anchieta' None]
cartao: ['Master' 'Visa' 'Amex' 'Elo']


Tabelas de frequência:
---------------------------------------

1-sexo:
F    4
M    2
Name: sexo, dtype: int64

2-bairro:
Belverde    2
Savassi     2
Anchieta    1
Name: bairro, dtype: int64
```

```
3-cartão:
Visa      2
Master    2
Elo       1
Amex      1
Name: cartao, dtype: int64
```

O programa é dividido em três partes. Na primeira, declara-se o DataFrame `vendas` com dados idênticos aos da figura apresentada. Embora já estejamos bastante acostumados com esta operação (criar DataFrames), é importantíssimo destacar uma novidade presente nesse programa: o emprego do valor especial `None` para representar que o bairro do cliente de `id=5` é desconhecido. Esta abordagem é comumente utilizada em situações em que o valor de um atributo (seja ele categórico ou numérico) é desconhecido, ausente ou não aplicável para um dado objeto.

Na segunda parte do programa utilizamos o método `unique()` sobre cada um dos atributos categóricos. Esse método retorna um array que armazena todas as categorias distintas do atributo escolhido. Por exemplo, no DataFrame `vendas`, temos seis linhas, mas apenas duas categorias distintas para o atributo `sexo`: {"F", "M"}. Para o atributo `Bairro` as categorias são {"Belverde", "Savassi", "Anchieta", `None`} (veja que `None` é apresentado, para que fique claro que em ao menos uma linha do DataFrame possui valor ausente para o atributo). Por fim, as categorias de `cartao` são {"Master", "Visa", "Amex", "Elo"}. Embora muito simples, o método é uma verdadeira "mão na roda", especialmente quando estamos trabalhando com uma base de dados muito volumosa, e não temos qualquer conhecimento prévio a respeito do conteúdo por ela armazenado.

Na terceira parte do programa, apresentamos o método `value_counts()` que faz a mesma coisa que o `unique()`, acrescido de um bônus e tanto: além de apresentar as categorias distintas do atributo, o método computa o número de ocorrências de cada uma. Ou seja, ele é capaz de gerar uma tabela de frequências. Por

exemplo, considerando novamente o atributo `sexo`, temos que a categoria "F" possui frequência 4, enquanto "M" tem frequência 2. Analogamente, o mesmo tipo de tabela de frequência é computada para os atributos `bairro` e `cartao`. Um detalhe importante sobre o `value_counts()` é o fato de que ele retorna a tabela de frequências em uma Series.

Agregações

Agregação é uma operação que visa computar estatísticas para **grupos de linhas** de um DataFrame. Os grupos de linhas são definidos a partir de valores de um ou mais atributos categóricos, enquanto as estatísticas são computadas sobre atributos numéricos. Na biblioteca pandas, agregações são produzidas com o uso do método `group_by()`. O próximo programa mostra como produzir resultados agregados por `bairro` a partir do DataFrame `vendas`. Mais especificamente, o programa gera uma tabulação que apresenta o total de clientes por bairro, o valor total das vendas por bairro e o valor médio das vendas por bairro.

```
#P29 Agregação com o método `group_by()`
import pandas as pd

#1-cria o DataFrame
dados = {"sexo": ["F","M","F","F","F","M"],
         "bairro": ["Belverde",
                    "Belverde",
                    "Savassi",
                    "Anchieta",
                    None,
                    "Savassi"],
         "valor": [150.00,
                   35.00,
                   80.00,
                   250.00,
                   9.90,
                   25.00],
         "cartao": ["Master",
```

```
                    "Visa",
                    "Visa",
                    "Amex",
                    "Elo",
                    "Master"]}

id_clientes = [1,2,3,4,5,6]

vendas = pd.DataFrame(dados, index=id_clientes)

#2-Gera uma variável "grouped"
#  onde a chave é "bairro" e a medida "valor"
grupo_valor_bairro = vendas['valor'].groupby(vendas['bairro'])

#3-Computa agregados a partir da variável gerada
print('- quantidade de clientes, por bairro:\n
',grupo_valor_bairro.count())
print('--------------------------------------')
print('- valor total das vendas, por bairro:\n ',grupo_valor_bairro.sum())
print('--------------------------------------')
print('- valor médio das vendas, por bairro:\n
',grupo_valor_bairro.mean())
```

Observe cuidadosamente a saída produzida para, em seguida, ler a explicação do programa:

```
- quantidade de clientes, por bairro:
  bairro
Anchieta    1
Belverde    2
Savassi     2
Name: valor, dtype: int64
--------------------------------------
- valor total das vendas, por bairro:
  bairro
Anchieta    250.0
Belverde    185.0
Savassi     105.0
Name: valor, dtype: float64
--------------------------------------
```

```
- valor médio das vendas, por bairro:
  bairro
Anchieta    250.0
Belverde     92.5
Savassi      52.5
Name: valor, dtype: float64
```

O segredo do programa está na seguinte linha de código, que apresenta uma das possíveis formas de utilização do método `groupby()`:

```
grupo_valor_bairro = vendas['valor'].groupby(vendas['bairro'])
```

Essa linha é responsável por criar um **objeto grouped** ou **objeto GroupBy** chamado `grupo_valor_bairro`. Este objeto não armazena o resultado de nenhum cálculo, mas apenas gera estruturas internas que vão facilitar a produção de resultados agregados da coluna `valor` (atributo numérico) por `bairro` (atributo categórico, este último denominado de **atributo chave do grupo**). A partir de `grupo_valor_bairro` será possível produzir diferentes tipos de agregados de `valor` por `bairro`. No exemplo apresentado, utilizamos os métodos `count()`, `sum()` e `mean()` para obter as estatísticas desejadas. Mas é possível utilizar qualquer um dos métodos relacionados em https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats.

O exemplo que acabamos de mostrar produziu uma **tabulação simples**. Se você quiser agregar por mais de um atributo categórico (gerar uma **tabulação cruzada**), basta especificar estes atributos em uma lista, e passá-la como parâmetro para o método `groupby()`. Por exemplo, para criar um objeto `grouped` cruzando o valor da compra por sexo e bairro, você deve utilizar a sintaxe a seguir:

```
grupo_valor_sexo_bairro = vendas['valor'].groupby([vendas['sexo'],
vendas['bairro']])`
```

4.5 Gráficos

Neste capítulo, já apresentamos a forma para criar boxplots na pandas. Agora mostraremos como produzir outros tipos gráficos com a utilização do módulo `plot`. Mais especificamente, mostraremos as técnicas básicas para a produção de gráficos de linhas, gráficos de barras e histogramas.

Gráfico de linhas

Esse é o mais simples dos gráficos, aquele que você certamente cansou de fazer em suas provas na época de escola. No exemplo a seguir, mostramos como plotar o gráfico da função `y = 2x + 3`, para `x` variando de 1 a 10:

```
#P30 Gráfico de uma função
import pandas as pd

df = pd.DataFrame({"x": list(range(1,11))})
df["y"]=(df.x * 2) + 3
lines = df.plot.line(x="x",y="y", legend=False)
```

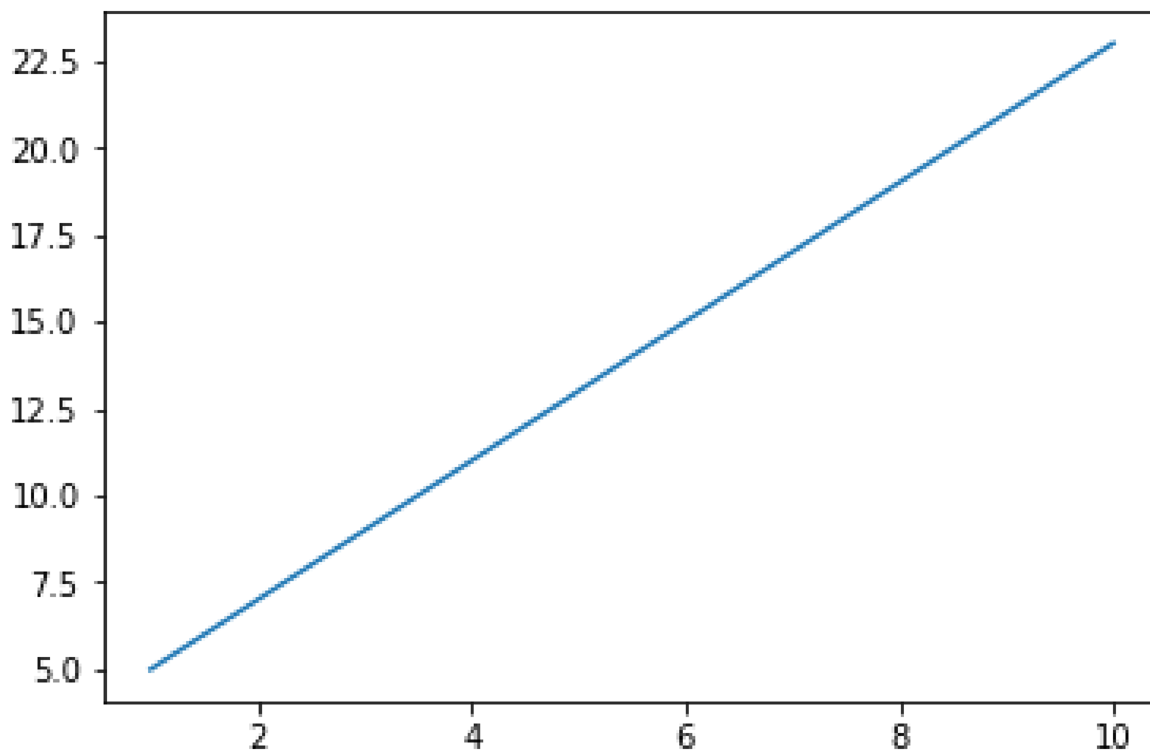

Figura 4.9: Gráfico da função y = 2x + 3.

Apesar de bem pequeno, o programa apresenta novidades bastante úteis. Ele funciona da seguinte forma:

- A primeira linha cria um DataFrame chamado `df` com 10 linhas e apenas a coluna `x`, com valor variando de 1 na primeira linha até 10 na última. Esses valores são produzidos com o uso da instrução `list(range(1,11))` (`list()` é uma função que gera uma lista de valores e `range()` a função que especifica quais serão os valores, neste caso o conjunto de números de 1 até o número anterior a 11).
- Em seguida, o comando `df["y"]=(df.x * 2) + 3` realiza uma operação muito interessante. Ele emprega a técnica de computação vetorizada, introduzida no capítulo 2, para acrescentar a coluna `y` como uma segunda coluna em `df`. Veja como é simples: basta realizar uma atribuição com o operador `=`, indicando no lado esquerdo que `y` é o nome da nova coluna

(`df["y"]=`) e jogando a fórmula `2x+3` no lado direito (`(df.x * 2) + 3`). Se você estiver em dúvida, acrescente o comando `print(df)` para visualizar o conteúdo do DataFrame.
- Por fim, utilizamos o método `plot.line()` para gerar um gráfico de linhas, onde os valores do eixo X serão obtidos a partir da coluna `x` de `df` (parâmetro `x="x"`) e os valores do eixo Y obtidos a partir da coluna `y` (parâmetro `y="y"`).

Vamos agora apresentar um segundo exemplo, desta vez bem mais relacionado à ciência de dados. Ele baseia-se nos dados do DataFrame `pof` (figura seguinte). Este DataFrame apresenta dados reais provenientes de uma pesquisa sobre os hábitos de consumo das famílias brasileiras (para maiores informações, consulte: https://www.researchgate.net/publication/303721710_Construcao_de_um_Data_Mart_para_a_Analise_dos_Habitos_de_Compra_das_Familias_Brasileiras). Os valores de cada célula indicam o percentual estimado de famílias que adquiriu cada um dos três produtos apresentados (óleo de soja, frango congelado e melancia) nos meses selecionados de um dado ano (3=março, 6=junho, 9=setembro e 12=dezembro). Por exemplo, o valor 71.15 na primeira célula indica que, de acordo com a pesquisa, estima-se que 71,15% das famílias brasileiras adquiriram óleo de soja no mês de março do ano em que a pesquisa foi realizada.

Mês	Óleo de Soja	Melancia	Frango Congelado
3	71.15	10.52	47.41
6	66.93	15.68	23.68
9	67.91	14.95	38.36
12	67.66	26.23	17.08

Figura 4.10: DataFrame contendo dados de clientes de uma loja.

O programa a seguir mostra como produzir um gráfico de linhas que demonstra a evolução das vendas dos três produtos ao longo dos meses.

```
#P31 Gráfico de linhas
import pandas as pd

dados = {"óleo de soja": [71.15, 66.93, 67.91, 67.66],
         "melancia": [10.52, 15.68, 14.95, 26.23],
         "frango congelado": [47.17, 23.68, 38.36, 17.08]
         }

df = pd.DataFrame(dados,index=[3,6,9,12])

lines = df.plot(kind="line",
                style=[".-","+-","*-"],
                ylim=(0,100), xlim=(3,12))
```

Aqui está o gráfico produzido:

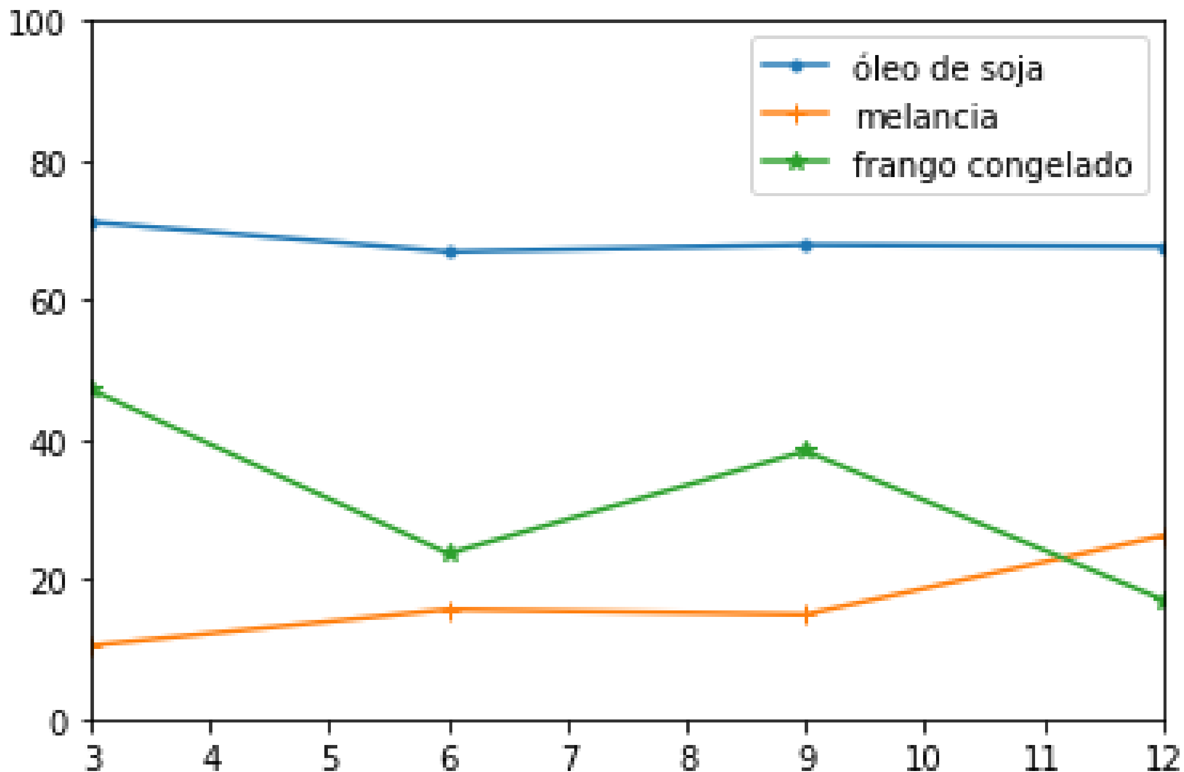

Figura 4.11: Evolução das compras por período do ano.

Antes de explicar o código, vamos falar do resultado apresentado no gráfico. Neste gráfico, o valor representado no eixo Y indica a porcentagem de famílias que adquiriram cada um dos produtos nos diferentes meses da pesquisa. É possível observar o padrão de vendas de cada produto ao longo dos meses é bastante distinto. A venda do produto óleo de soja mantém-se quase estável (variando entre 67% e 71%) em qualquer época do ano. O item melancia, por sua vez, foi adquirido por cerca de 26% das famílias entrevistadas no mês de dezembro. Mas, nos meses anteriores, uma porcentagem menor de famílias adquiriu este produto. Com relação ao produto frango congelado, o gráfico indica que no mês de março ocorreu uma grande procura pelo item. As compras caem em junho, sobem em setembro e caem de novo em dezembro.

Com relação ao código, veja que dessa vez utilizamos o método `df.plot()` de uma maneira diferente, especificando quatro

parâmetros:

- `kind="line"` : para informar que desejávamos produzir um gráfico de linhas. Na verdade a pandas oferece duas maneiras distintas para produzir seus gráficos. Ou você utiliza o nome de um método específico para gerar o tipo de gráfico desejado (por exemplo, `plot.line()`) ou utiliza o parâmetro `kind` para indicar o tipo (por exemplo, `plot(kind="line")`).
- `style=[".-","+-","*-"]` : para alterar o estilo do gráfico. Neste caso, indicamos que o gráfico contivesse pontos ligados por segmentos de reta (`".-"`) para os valores do óleo de soja, o sinal de adição ligado por segmentos de reta para os dados da melancia (`"+-"`) e asteriscos ligados por segmentos de reta (`"*-"`) para os dados do frango congelado. Para cada um dos três produtos, os símbolos `.` , `+` ou `*` marcam um dado do DataFrame, ou seja o valor do percentual comprado do um produto em um mês.
- `ylim(0,100)` : para indicar a escala do eixo Y (0 a 100).
- `xlim(3,12)` : para indicar a escala do eixo X (3 a 12).

Existem diversos outros parâmetros que você pode definir para configurar os seus gráficos. Para uma lista completa, consulte: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html#pandas.DataFrame.plot.

Agora, atenção para a explicação a seguir. O programa que acabamos de apresentar gerou três diferentes gráficos em um só (um para cada produto), que compartilham os mesmos valores para o eixo Y (mês do ano). Por este motivo, foi preciso definir a coluna do mês como índice do DataFrame no momento de sua criação. Esta é a forma padrão que utilizamos na pandas para poder gerar gráficos contendo diversos resultados. Por isso, muitas vezes precisamos criar um DataFrame temporário resultante da transformação de um DataFrame com os dados originais para conseguir gerar os gráficos que desejamos. Não se preocupe, pois o capítulo 6 deste livro é inteiramente dedicado às operações para

transformação de DataFrames (mas já adiantaremos o tema no último exemplo deste capítulo, como você logo verá!).

Gráfico de barras

Este é, sem dúvida, um dos tipos de gráfico com o qual mais estamos acostumados. Mesmo as pessoas que não possuem qualquer conhecimento de Estatística conseguem interpretá-lo sem maiores problemas. Neste tópico, apresentaremos dois diferentes programas para a produção de gráfico de barras. O primeiro utiliza o conjunto de dados seguinte. Suponha que ele armazene a quantidade de alunos que se inscreveram em três diferentes cursos de verão de uma universidade.

```
teatro    = 70 alunos
escultura = 25 alunos
pintura   = 50 alunos
```

Para produzir um gráfico de barras com esses dados, basta utilizar o método `plot` e especificar o parâmetro `kind="bar"` (para gráfico de barras verticais) ou `kind="barh"` (para gráfico de barras horizontais):

```
#P32 Gráfico de barras vertical
import pandas as pd

df = pd.DataFrame([70, 25, 50],
                  index=["teatro", "escultura", "pintura"])

barras = df.plot(kind="bar", legend=False)
```

Veja o resultado: os índices foram automaticamente utilizados para formar as legendas do eixo Y.

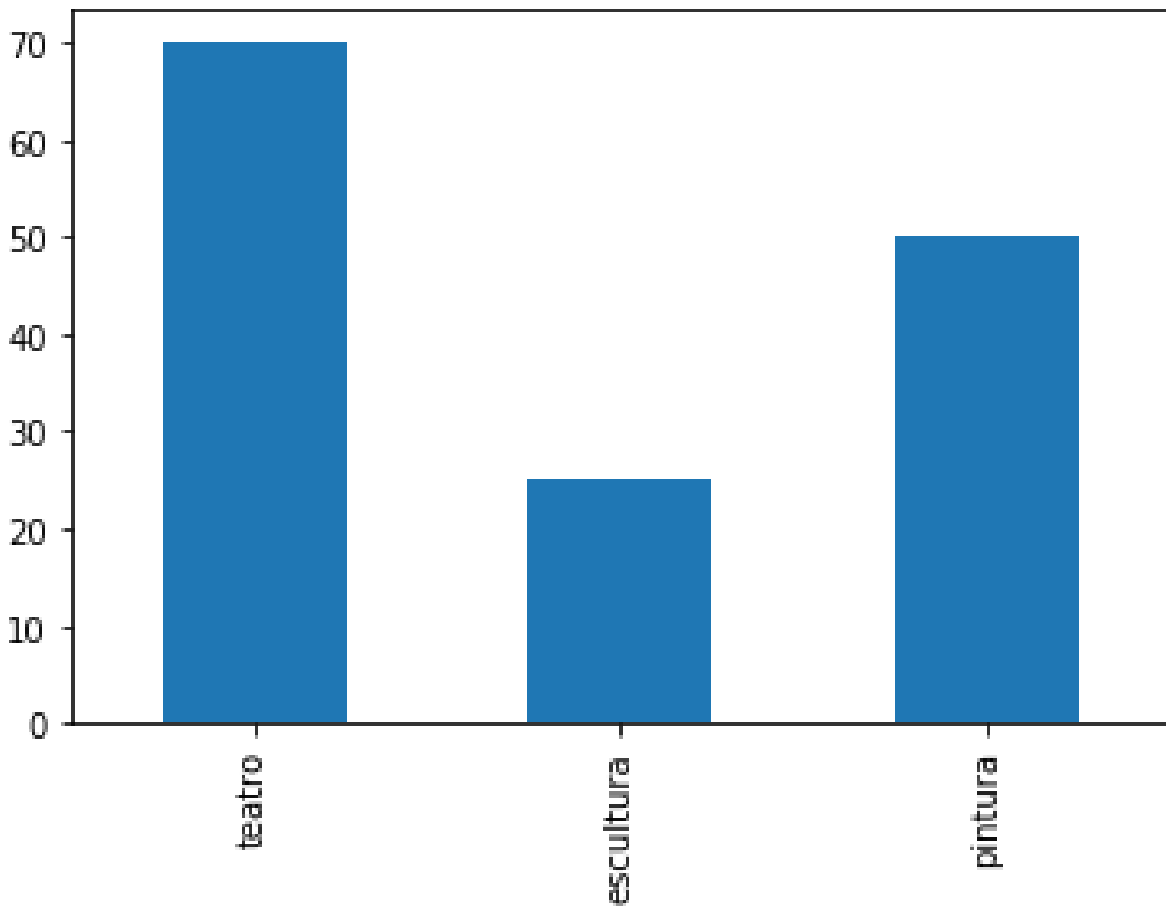

Figura 4.12: Gráfico de barras simples.

Suponha que, desta vez, o seu conjunto de dados possua informações sobre a quantidade de homens e mulheres inscritos em cada curso:

```
teatro    = 40 mulheres e 30 homens
escultura = 10 mulheres e 15 homens
pintura   = 30 mulheres e 20 homens
```

No programa a seguir, mostramos como produzir um gráfico de barras agrupadas que mostra o número de homens e mulheres inscritos em cada curso.

```
#P33: gráfico de barras agrupadas
import pandas as pd
```

```
df = pd.DataFrame({"mulheres": [40,10,30],
                   "homens": [30,15,20]},
                  index=["teatro", "escultura", "pintura"])

barras = df.plot(kind="bar", legend=True, color=["yellow", "blue"])
```

Neste programa, utilizamos o parâmetro `color=["yellow", "blue"]` para definir as cores das barras de cada sexo (amarelo para mulheres e azul para homens) e também `legend=True` para habilitar as legendas. Desta forma, o resultado produzido é o seguinte:

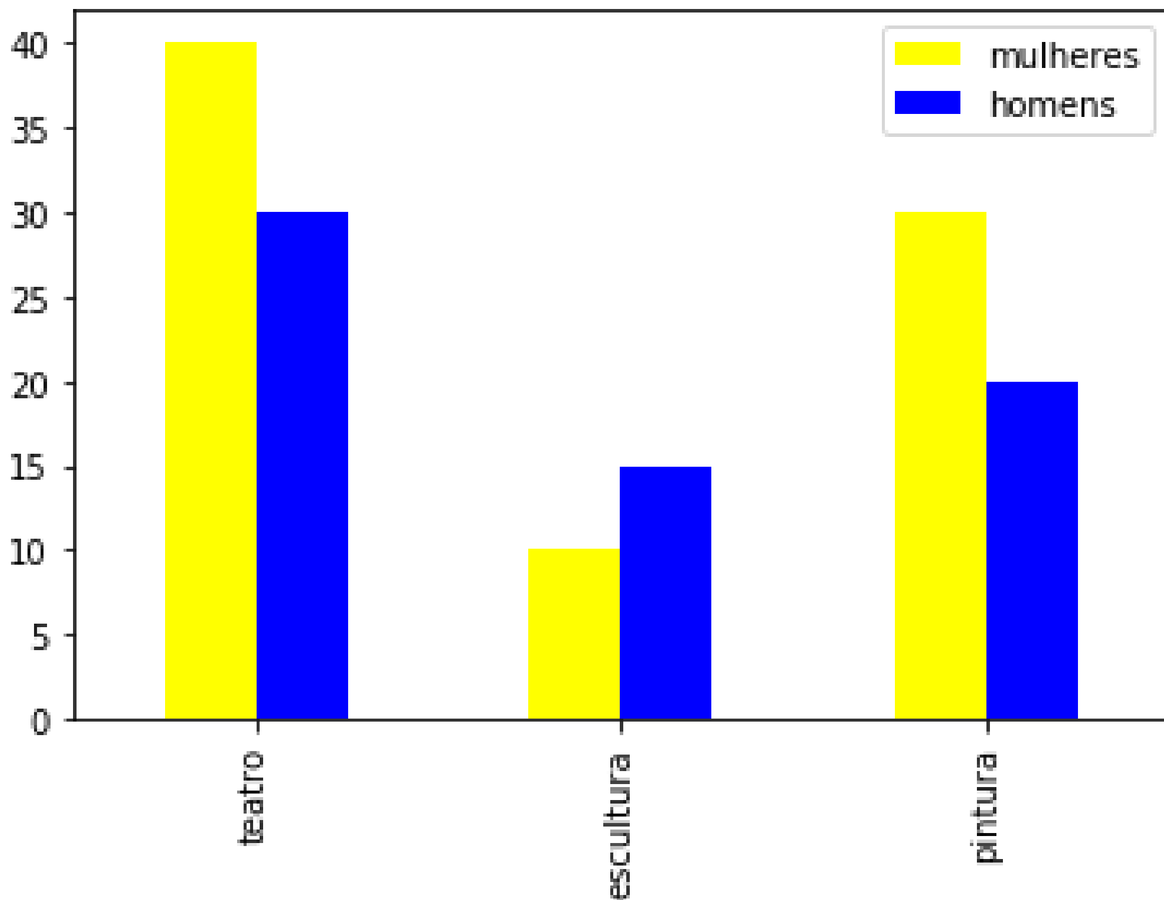

Figura 4.13: Gráfico de barras agrupadas.

Histograma

O histograma é um gráfico utilizado para sumarizar a distribuição de um dado atributo numérico `x`. Neste tipo de gráfico, a faixa de valores de `x` é particionada em subfaixas consecutivas, denominadas *bins* ou *buckets*. O número de bins de um histograma pode ser especificado pelo usuário ou definido automaticamente pela pandas.

O programa a seguir mostra como produzir um histograma com três bins a partir de um conjunto de dados que armazena o número de horas que 30 diferentes estudantes passaram se preparando para uma prova.

```
#P34: Histograma
import pandas as pd

df = pd.DataFrame({
     "tempo": [4, 5, 1, 7, 7, 8, 6, 6, 5,
               2, 5, 8, 7, 1, 6, 3, 4, 8,
               5, 7, 4, 6, 3, 6, 2, 6, 8]
     })

hist = df.plot(kind="hist", bins=3)
```

Aqui está o histograma produzido a partir das 30 observações:

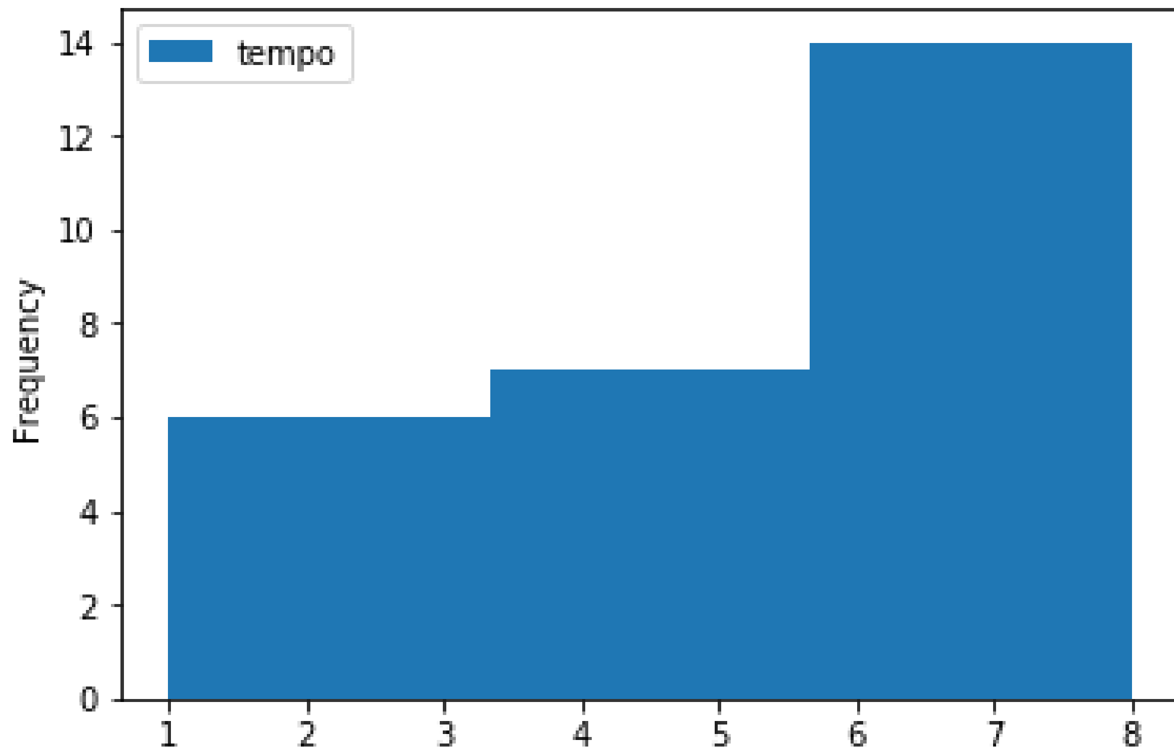

Figura 4.14: Histograma.

No histograma, o eixo X representa os bins (faixas de valores do atributo), enquanto o eixo Y indica a frequência de cada faixa. No nosso exemplo, criamos 3 bins, ou seja, definimos 3 faixas de valores para o histograma com larguras iguais. Ao observar o gráfico gerado, é possível notar que a faixa contendo o tempo entre 6 e 8 horas é a mais frequente.

PANDAS *VERSUS* MATPLOTLIB

A funcionalidade de produção de gráficos da pandas não foi criada com o objetivo de gerar gráficos com qualidade de produção (aqueles que podem aparecer em um relatório ou no site de sua empresa).

Na realidade, os gráficos da pandas têm o propósito de servir como ferramenta de apoio para o processo de estudo de uma base de dados. Para criar de gráficos profissionais, existe uma outra biblioteca mais recomendada, chamada Matplotlib (https://matplotlib.org).

4.6 Detecção de outliers

Outliers são dados anormais que podem ser reais ou, no caso mais frequente, gerados devido a algum erro (seja um erro na coleta dos dados, um erro de digitação etc.). Outliers costumam exercer influência negativa na qualidade dos resultados de um processo de ciência de dados. Por esta razão, torna-se preciso identificá-los na etapa de estudo da base de dados.

Uma vez que os outliers tenham sido identificados, há diversas maneiras para tratá-los. A mais simples consiste em tão somente removê-los da base de dados antes do processo de análise. Por exemplo, considerando uma base contendo dados de clientes, você poderia ordená-la pela renda de forma decrescente, e remover os 0,5% primeiros e também os 0,5% últimos registros. Dessa forma, você eliminaria as pessoas com renda extremamente alta ou extremamente baixa.

O exemplo do parágrafo anterior representa, obviamente, uma maneira muito simplória para lidar com o tratamento de outliers. Na

prática, existem diversas técnicas estatísticas, algumas bastante sofisticadas, para identificar e tratar outliers. Nesta seção, será apresentada uma técnica básica, porém eficaz e muito comumente utilizada (ao menos como técnica preliminar) quando lidamos com bases reais. Ela se chama *interquartile range* (IQR).

Antes de apresentar a técnica IQR, é preciso definir o conceito de **quartil**. Tratam-se de três *data points* (Q1, Q2, Q3 — já apresentados na seção em que introduzimos os boxplots) que dividem um conjunto de dados ordenados em quartos, isto é, em quatro pedaços, cada um com 25% das observações:

- Q1 (1º quartil): 25% das observações abaixo, 75% acima.
- Q2 (2º quartil): 50% das observações abaixo, 50% acima (ou seja, Q2 é a mediana do conjunto de dados).
- Q3 (3º quartil): 75% das observações abaixo, 25% acima.

A medida do IQR consiste na diferença entre o terceiro e o primeiro quartis da distribuição (Q3 – Q1). A técnica para detecção de outliers baseada no IQR funciona da seguinte forma: todos os pontos localizados a uma distância de 1,5 x IQR acima do terceiro quartil ou abaixo do primeiro quartil são listados como candidatos a outliers. Como estes pontos se localizam muito distantes de Q1 ou de Q3, há uma grande chance real de eles realmente representarem dados anormais.

Uma das maiores vantagens do método IQR é o fato de que ele torna possível a visualização dos outliers em um boxplot, como mostra o exemplo a seguir. Neste programa, utilizaremos uma versão ligeiramente modificada do conjunto de dados contendo o número de horas de estudo de 30 alunos — utilizado no recém-apresentado exemplo sobre histograma. A modificação em questão é a seguinte: considere que, acidentalmente, o último valor foi digitado como 18 horas em vez de 8 horas, que seria o valor correto. Ou seja, considere que um outlier foi introduzido devido a um erro de digitação. No programa a seguir, veremos que o método IQR é capaz de colocar esse dado suspeito em destaque no boxplot.

```
#P35: IQR + Boxplot
import pandas as pd

df = pd.DataFrame({
     "tempo": [4, 5, 1, 7, 7, 8, 6, 6, 5,
               2, 5, 8, 7, 1, 6, 3, 4, 8,
               5, 7, 4, 6, 3, 6, 2, 6, 18]
     })

boxplot = df.boxplot(column=['tempo'],
                     showmeans=True)
```

Diagrama gerado:

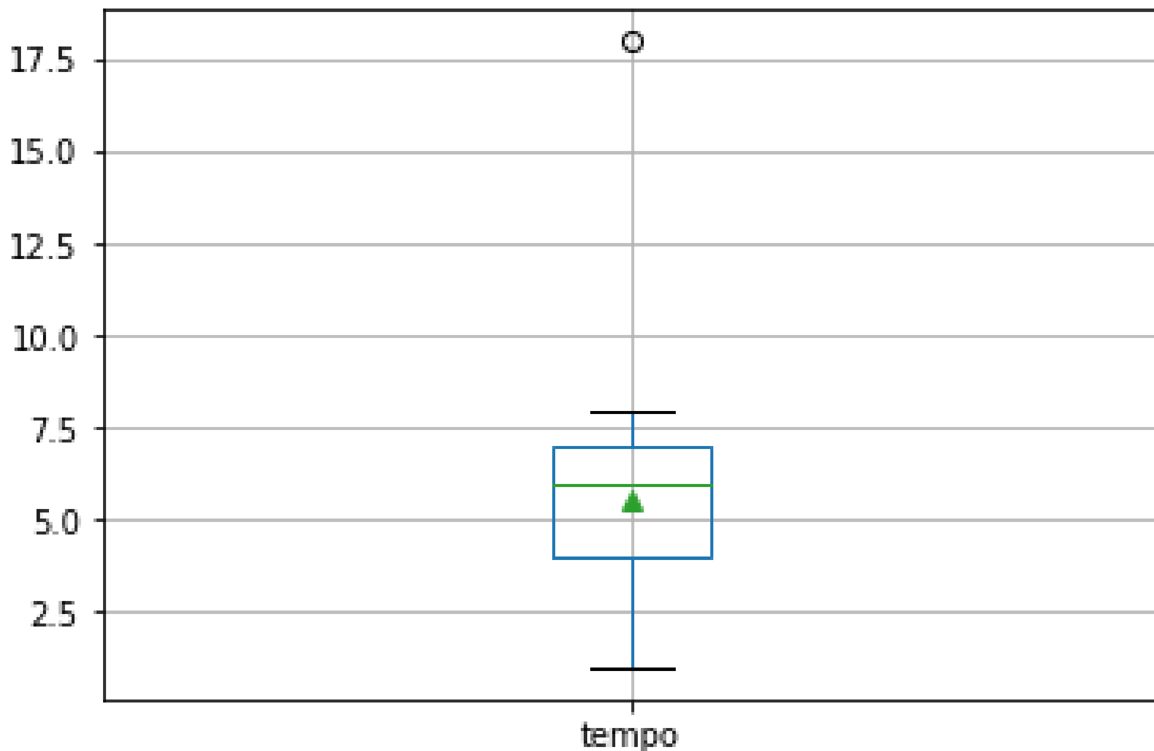

Figura 4.15: Histograma.

Vamos à explicação. Em um boxplot, Q1 e Q3 representam, respectivamente, a linha inferior e a linha superior da caixa retangular, enquanto Q2 é a reta através da caixa. Os whiskers (as duas linhas fora da caixa) marcam os valores mais extremos

compreendidos entre 1,5 x IQR abaixo do primeiro quartil e acima do terceiro quartil. Os casos fora dessa faixa são considerados outliers e são plotados individualmente, fora dos whiskers. Veja que é isto que acontece com o valor 18 do conjunto de dados (trata-se da bolinha branca localizada bem no topo do gráfico).

4.7 Projeto prático — estudando o dataset flags

Agora chegou o momento de aplicarmos tudo o que foi aprendido ao longo do capítulo sobre uma base de dados real: `flags` , a base de nosso projeto prático. Nesta seção, faremos isso através de três diferentes programas que executarão as seguintes tarefas, respectivamente:

1. Obter as propriedades de todos os atributos da base `flags` ;
2. Gerar uma tabela de frequência a partir um conjunto de atributos selecionados;
3. Gerar um gráfico de barras para um conjunto de atributos selecionados.

Antes de tudo, vamos refrescar sua memória sobre a base `flags` . Trata-se de um arquivo CSV contendo 194 linhas e 30 colunas, sendo que cada linha armazena os dados de um determinado país e de sua bandeira nacional.

A seguir, apresentamos o primeiro programa para estudar a base, que tem por objetivo listar as propriedades básicas de todos os atributos. Para cada atributo, o programa verifica se o tipo é numérico ou categórico e apresenta um breve resumo de seu conteúdo. Por exemplo: lista as categorias no caso de atributos categóricos ou computa medidas de tendência central e dispersão para os numéricos. Comentários detalhados são apresentados após o código.

```python
#P36: Estudando a base de dados flags
#     I. Propriedades básicas de cada atributo
import pandas as pd

#-------------------------------------------------------
#(1)-Importa a base de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')

#-------------------------------------------------------
#(2)-Obtém as propriedades básicas de cada atributo
#-------------------------------------------------------
i=0
for c in flags.columns:
    i +=1
    att = flags[c]                         #atributo
    att_dtype = att.dtype                  #dtype
    att_tam_dominio =  att.unique().size #tamanho do domínio
    att_tem_nulo = any(att.isnull())     #possui valor nulo?
    if (att_tam_dominio < 8):
        print("("+str(i)+") atributo:", c, "\t",
              "dtype:", att_dtype, "\t",
              "nulos: ", att_tem_nulo, "\n",
              "domínio:", att.unique())
    else:
        if (att_dtype=='object'):
            print("("+str(i)+") atributo:", c, "\t",
                  "dtype:", att_dtype, "\t",
                  "nulos: ", att_tem_nulo, "\n",
                  "domínio (primeiros elementos):", att.unique()[0:8])
        else:
            print("("+str(i)+") atributo:", c, "\t",
                  "dtype:", att_dtype, "\t",
                  "nulos: ", att_tem_nulo, "\n",
                  "min: ", att.min(), "\t",
                  "max: ", att.max(), "\t",
                  "média: ", round(att.mean(),2), "\t",
                  "d.p.: ", round(att.std(),2))
```

A seguir, apresenta-se a parte inicial da saída gerada pelo programa. Como esta saída é muito grande, apenas os 5 primeiros resultados são apresentados.

```
(1) atributo: name       dtype: object    nulos:  False
 domínio (primeiros elementos): ['Afghanistan' 'Albania' 'Algeria'
'American-Samoa' 'Andorra' 'Angola'
 'Anguilla' 'Antigua-Barbuda']
(2) atributo: landmass   dtype: int64     nulos:  False
 domínio: [5 3 4 6 1 2]
(3) atributo: zone       dtype: int64     nulos:  False
 domínio: [1 3 2 4]
(4) atributo: area       dtype: int64     nulos:  False
 min:  0         max:  22402     média:  700.05          d.p.:  2170.93
(5) atributo: population         dtype: int64     nulos:  False
 min:  0         max:  1008      média:  23.27   d.p.:  91.93
 ...
```

O programa é dividido em duas partes, sendo que a primeira é trivial: consiste em uma única linha, onde o método `read.csv()` é empregado para importar o arquivo `flags.csv` para um DataFrame pandas.

Já a segunda parte é grande e bem mais interessante. É nela que se realiza o estudo dos atributos da base de dados. Esse trecho de código funciona da seguinte maneira. Primeiro implementamos `for c in flags.columns:` para gerar um laço sobre o **nome** de todas as colunas. Embora laços sejam menos utilizados em programas de ciência de dados, existem situações onde o recurso é necessário e essa é exatamente uma delas! Dentro do laço, temos inicialmente os seguintes comandos:

- `att = flags[c]` : recupera o atributo de nome `c` ;
- `att_dtype = att.dtype` : recupera o `dtype` do atributo;
- `att_tam_dominio = att.unique().size` : utiliza o método `unique()` para gerar um vetor contendo o domínio (valores ou categorias distintas) do atributo. Então o tamanho (número de elementos) deste conjunto é retornado com a propriedade `size` .

- `att_tem_nulo = any(att.isnull())` : a função `any()` é uma função do Python padrão que recebe como entrada uma relação de valores (em uma Series, lista ou outra ED) e retorna `True` (valor escalar) caso ao menos um dos elementos da relação seja `True` . (mais informações em: https://www.programiz.com/python-programming/methods/built-in/any). Neste exemplo, como estamos aplicando `any()` sobre `att.isnull()` , o método retornará `True` caso o atributo possua ao menos uma linha com valor nulo (por curiosidade, nenhum dos atributos da base `flags` possui valor nulo).

Após obter esses dados, o programa segue o seu processamento e imprime as propriedades básicas do atributo. De acordo com o domínio do atributo 3 diferentes tipos de informação poderão ser exibidos:

- Caso o domínio do atributo possua 8 ou menos elementos, então são impressos todos os seus valores ou categorias.
- Caso o domínio do atributo possua mais de 8 elementos e o seu `dtype` seja `object` , são impressas apenas as suas 8 primeiras categorias.
- Caso o domínio do atributo possua mais de 8 elementos e o seu `dtype` não seja `object` (ou seja, o atributo é numérico), são impressos os valores mínimo, máximo, médio e o desvio padrão.

Será agora apresentado o segundo programa para estudar a base `flags` . Este programa explora especificamente os atributos que indicam as cores presentes na bandeira de cada país. Ele utiliza o método `value_counts()` para computar uma tabela de frequências para cada cor (ou seja, para cada cor, o programa vai obter o total de bandeiras em que a mesma ocorre).

```
#P37: Estudando a base de dados flags
#     II. Cores e suas frequências nas bandeiras
import pandas as pd

#-------------------------------------------------------
```

```
#(1)-Importa a base de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')

#-------------------------------------------------------
#(2)-Gera a tabela de frequências
#-------------------------------------------------------
df_cores=pd.DataFrame()
for c in flags.columns:
    if c in ['red','green','blue','gold','white','black','orange']:
        df_cores[c]=flags[c].value_counts()

print(df_cores)
```

E aqui está a saída. Podemos ver que vermelho (`red`) é a cor mais frequente, ocorrendo em 153 das 194 bandeiras. A segunda mais frequente é a cor branca (`white`), presente em 146 bandeiras. E a menos frequente, a cor laranja (`orange`) que está em apenas 26 bandeiras.

```
   red  green  blue  gold  white  black  orange
1  153     91    99    91    146     52      26
0   41    103    95   103     48    142     168
```

O programa funciona da seguinte maneira. Após a importação do arquivo `flags.csv` para o DataFrame `flags` , a segunda parte do programa começa com a geração de um DataFrame vazio, denominado `df_cores` . Então, mais uma vez programamos um laço para iterar sobre todos os nomes de colunas de `flags` . Caso a coluna corrente seja uma das colunas referentes às cores, o programa computa a frequência dos valores `0` (ausência da cor na bandeira) ou `1` (presença da cor). Esse resultado é então adicionado como uma coluna em `df_cores` . Ao final do processamento, `df_cores` conterá 7 colunas e duas linhas, armazenando o número de ocorrências de cada cor.

E agora que tal produzir gráficos de barras horizontais a partir dessa tabela de frequências? É exatamente isso que o que o programa

seguinte faz.

```
#P38: Estudando a base de dados flags
#     Gráficos de barras com as frequências das cores
import pandas as pd

#-------------------------------------------------------
#(1)-Importa a base de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')

#-------------------------------------------------------
#(2)-Gera a tabela de frequências
#-------------------------------------------------------
df_cores=pd.DataFrame()
for c in flags.columns:
    if c in ['red','green','blue','gold','white','black','orange']:
        df_cores[c]=flags[c].value_counts()

#-------------------------------------------------------
#(3)-Gera os gráficos de barras
#-------------------------------------------------------
lst_cores = [['red','beige'],
             ['green','beige'],
             ['blue','beige'],
             ['gold','beige'],
             ['whitesmoke','beige'],
             ['black','beige'],
             ['orange','beige']]

df_cores.plot(kind='barh',
              subplots=True,
              figsize=(8,25),
              color = lst_cores)
```

Eis o belo gráfico produzido:

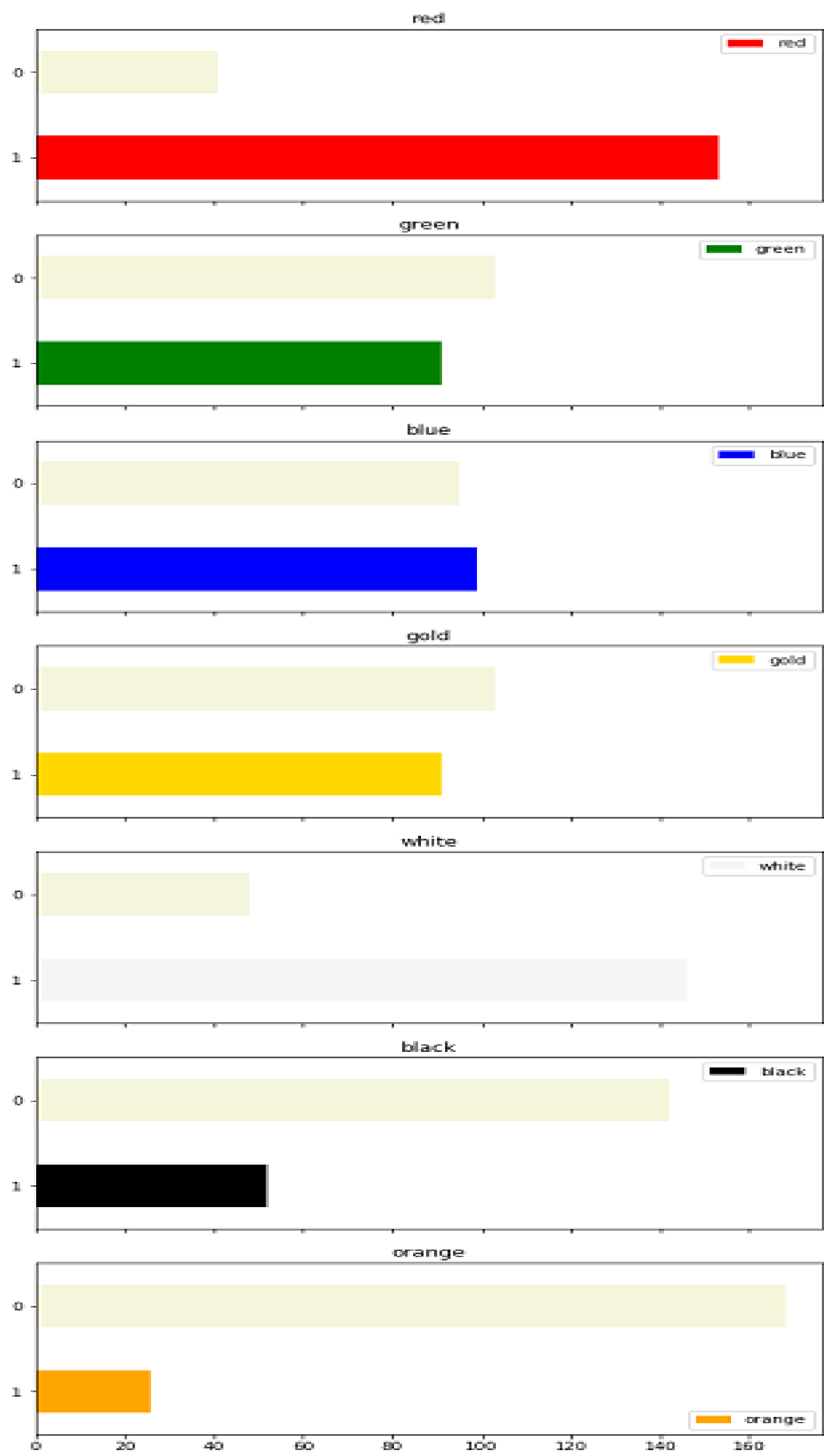
red
0
1
red
green
0
1
green
blue
0
1
blue
gold
0
1
gold
white
0
1
white
black
0
1
black
orange
0
1
orange
0
20
40
60
80
100
120
140
160

Figura 4.16: Gráfico de barras com as frequências de cada cor.

Neste programa, utilizamos o método `plot()` sobre `df_cores` com os seguintes parâmetros:

- `kind='barh'` : define que o tipo de gráfico é de barras horizontais.
- `subplots=True` : para permitir gerar vários gráficos (um para cada cor) dentro de uma única figura.
- `figsize=(8,25)` : altura e largura do gráfico.
- `color = lst_cores` : define as cores de cada gráfico. Veja que estamos utilizando sempre a cor bege (`beige`) na barra referente ao valor 0 (ausência de cor) e a própria cor para a barra que representa o valor 1 (presença da cor). Para uma lista com os nomes de cores que podem ser utilizados em programas, consulte https://matplotlib.org/3.1.0/gallery/color/named_colors.html.

Para encerrar o capítulo, dois comentários importantes. Primeiro, é necessário observar que, além dos diversos métodos que apresentamos ao longo deste capítulo, existem outros disponíveis na pandas para a computação de medidas estatísticas e a geração de gráficos. Para obter a relação completa dos mesmos, consulte o endereço https://pandas.pydata.org/pandas-docs/stable/reference/frame.html.

Por fim, como segundo comentário, é válido mencionar que todos estes métodos podem ser aplicados não apenas sobre colunas de DataFrames (como fizemos ao longo de todo o capítulo), mas também sobre Series. Afinal de contas, cada coluna de um DataFrame é uma Series (o DataFrame é um dicionário de Series).

Capítulo 5
Combinando DataFrames

A etapa de **seleção de dados** é quase sempre necessária em qualquer processo de ciência de dados. É nela que todos os dados relevantes para a resolução do problema são coletados e **combinados** em um único DataFrame. Trata-se de uma tarefa desafiadora, uma vez que os dados relevantes costumam estar estruturados em formatos heterogêneos: tabelas de bancos de dados, arquivos texto (CSV, JSON etc.), planilhas Excel, páginas da internet, arquivos de log, entre outras possibilidades.

Felizmente a pandas simplifica muito o processo de coleta de dados, pois permite a importação de virtualmente qualquer tipo de arquivo para DataFrames (como foi visto no capítulo 3). A biblioteca vai ainda mais além, oferecendo ferramentas para possibilitar a combinação de DataFrames. Este capítulo tem como objetivo principal apresentar as três diferentes abordagens que podem ser empregadas para combinar DataFrames:

- Concatenação
- Operações de conjunto
- Junção

5.1 Concatenação

Concatenar dois ou mais DataFrames pandas consiste na operação de empilhar um "no topo" do outro, gerando um DataFrame único como resultado. Vamos apresentar essa operação de forma prática. Suponha uma empresa que possua três lojas, digamos A, B e C. Considere que em todas as segundas-feiras, cada uma das lojas deve enviar para a matriz um arquivo padronizado contendo o

consolidado das vendas do final de semana. Por exemplo, considere que os arquivos seguintes foram os últimos enviados por cada loja:

Arquivo da Loja A:

```
A sex 7500
A sab 9500
A dom 8200
```

Arquivo da Loja B:

```
B sex 5100
B sab 8250
B dom 9900
```

Arquivo da Loja C:

```
C sab 7500
C dom 11800
```

Veja que todos os arquivos possuem os mesmos três atributos, especificados na mesma ordem: identificação da loja, dia da semana e valor consolidado das vendas. Observe ainda que o arquivo da loja C possui apenas duas linhas (imagine que a loja não abriu na sexta-feira). No programa a seguir, os dados apresentados serão inicialmente estruturados em três DataFrames distintos que, em seguida, serão concatenados para um único DataFrame maior através do método `concat()`:

```
#P39: Concatenação de DataFrames
import pandas as pd

#(1)-Cria os DataFrames com as vendas de cada loja
lojaA = pd.DataFrame({"loja":["A", "A", "A"],
                      "dia": ["sex", "sab", "dom"],
                      "valor": [7500, 9500, 8200]}
                      )

lojaB = pd.DataFrame({"loja":["B", "B", "B"],
                      "dia": ["sex", "sab", "dom"],
```

```
                      "valor": [5100, 8250, 9900]}
                      )

lojaC = pd.DataFrame({"loja":["C", "C"],
                      "dia": ["sab", "dom"],
                      "valor": [7500, 11800]}
                      )

#(2)-Concatena tudo em um único DataFrame
lojasABC = pd.concat([lojaA,lojaB,lojaC], ignore_index=True)

print(lojasABC)
```

Saída:

```
  loja  dia  valor
0    A  sex   7500
1    A  sab   9500
2    A  dom   8200
3    B  sex   5100
4    B  sab   8250
5    B  dom   9900
6    C  sab   7500
7    C  dom  11800
```

Neste exemplo, os DataFrames `lojaA` (com 3 linhas), `lojaB` (com 3 linhas) e `lojaC` (com 2 linhas) foram concatenados para um único DataFrame `lojasABC` contendo 3 + 3 + 2 = 8 linhas. A utilização do método `concat()` para efetuar esta operação é bastante simples: basta especificar os arquivos a serem concatenados em uma lista (`[lojaA,lojaB,lojaC]`) para que isso seja feito na ordem indicada. Em nosso exemplo, também fizemos uso do parâmetro `ignore_index=True` , que, apesar de opcional, é muito útil. Ele foi utilizado para forçar a geração de novos índices, variando de 0 a 7, no DataFrame `lojasABC` . Se o parâmetro não for especificado, a pandas trabalhará apenas com os índices dos DataFrames originais (0, 1 e 2), associando mais de uma linha a cada um destes três índices.

A concatenação é uma operação que faz mais sentido quando os DataFrames de origem possuem a mesma definição ou **esquema**, isto é, contêm os mesmos atributos, com os mesmos nomes e tipos, especificados na mesma ordem. No entanto, nada impede que DataFrames não inteiramente compatíveis sejam concatenados. No exemplo a seguir, mostramos o que acontece quando se realiza a concatenação de dois DataFrames que não possuem nenhum atributo em comum:

```
#P40: Concatenação de DataFrames Incompatíveis
import pandas as pd

#(1)-Cria dois DataFrames com definição diferente
d1 = pd.DataFrame({"carro":["Hyundai", "Renault", "Fiat"]})
d2 = pd.DataFrame({"animal":["Capivara", "Bem-te-vi"]})

#(2)-Concatena os DataFrames
d3 = pd.concat([d1,d2], ignore_index=True, sort=False)

print(d3)
```

A seguir, o resultado da concatenação:

```
     carro     animal
0  Hyundai        NaN
1  Renault        NaN
2     Fiat        NaN
3      NaN   Capivara
4      NaN  Bem-te-vi
```

Neste exemplo, primeiro criamos dois DataFrames, `d1` com uma única coluna chamada `carro` e `d2` com uma única coluna chamada `animal`. Como resultado da concatenação, o DataFrame produzido fica definido com essas mesmas duas colunas, fazendo uso do valor `NaN` para permitir o alinhamento dos dados. O parâmetro `sort=False` foi utilizado para fazer com que, no DataFrame resultante, a pandas não tente reordenar os atributos (isto é, para definir o primeiro atributo de `d3` como `carro`, vindo `d1`, e o segundo como `animal`, vindo de `d2`).

5.2 Operações de conjunto

União, interseção e diferença

Em algumas situações práticas, pode ser interessante tratar um DataFrame como um **conjunto matemático**, ou seja, como uma coleção de elementos do mesmo tipo. Neste caso, cada linha do DataFrame passa a ser considerada um elemento do conjunto. Mas por que isso é útil? A explicação é simples. Como sabemos, problemas de ciência de dados tipicamente exigem a análise de diversas bases de dados, muitas vezes oriundas de diferentes fontes. Em alguma etapa da análise, pode ser necessário comparar o conteúdo de dois ou mais DataFrames para determinar se eles possuem elementos (linhas) em comum.

Por exemplo, suponha que você recebeu dois enormes arquivos contendo e-mails de alunos de cursos EAD. Um dos arquivos contém os e-mails dos alunos que realizaram o curso de SQL e o outro dos alunos do curso de Python. Imagine que você precise de respostas para as seguintes questões:

1. Quais são os alunos distintos, considerando ambas as listas?
2. Quais são os alunos que fizeram ambos os cursos?
3. Quais os alunos que realizaram o curso de SQL, mas não o de Python (e vice-versa)?

Neste caso, as respostas podem ser obtidas de maneira trivial, se você importar os arquivos para dois diferentes DataFrames e, então, considerar que estes DataFrames são conjuntos matemáticos. Bastará então aplicar as operações básicas de união, interseção e diferença para obter as respostas para as questões (1), (2) e (3), respectivamente. Na pandas, estas operações de conjunto são disponibilizadas através dos métodos relacionados na tabela seguinte:

Operação	Método(s)
união	`concat()` + `drop_duplicates()`
interseção	`merge()`
diferença	`isin()` + indexação booleana

No próximo programa, a utilização destes métodos é demonstrada. Uma explicação detalhada é apresentada logo após a listagem do código.

```
#P41: Operações de conjunto
import pandas as pd

#(1)-Cria dois DataFrames com e-mails
df_sql = pd.DataFrame({"email":["rakesh@xyz.com",
                                "ecg@acmecorpus.com"]})

df_python = pd.DataFrame({"email":["ana@xyz.com",
                                   "jonas@acmecorpus.com",
                                   "rakesh@xyz.com"]})

#(2)-Efetua as operações de conjunto

#2.1 União (relação de alunos distintos)
alunos = pd.concat([df_sql, df_python], ignore_index=True)
alunos = alunos.drop_duplicates()

#2.2 Interseção (quem fez ambos os cursos)
sql_e_python = df_sql.merge(df_python)

#2.3 Diferença (quem fez só SQL e quem fez só Python)
so_sql = df_sql[df_sql.email.isin(df_python.email)==False]
so_python = df_python[df_python.email.isin(df_sql.email)==False]

#(3)-Imprime os resultados
print('------------------------------')
print('Alunos Distintos:')
print(alunos)
```

```
print('------------------------------')
print('Alunos cursaram SQL e Python:')
print(sql_e_python)
print('------------------------------')
print('Alunos cursaram apenas SQL:')
print(so_sql)
print('------------------------------')
print('Alunos cursaram apenas Python:')
print(so_python)
```

Veja o resultado:

```
Alunos Distintos:
                  email
0         rakesh@xyz.com
1     ecg@acmecorpus.com
2            ana@xyz.com
3   jonas@acmecorpus.com
------------------------------
Alunos cursaram SQL e Python:
            email
0  rakesh@xyz.com
------------------------------
Alunos cursaram apenas SQL:
                email
1  ecg@acmecorpus.com
------------------------------
Alunos cursaram apenas Python:
                  email
0            ana@xyz.com
1   jonas@acmecorpus.com
```

Nesse programa, inicialmente criamos dois DataFrames, `df_sql` (e-mails dos alunos do curso de SQL) e `df_python` (e-mails dos alunos do curso de Python). Após terem sido criados, estes DataFrames são submetidos às operações de conjunto na segunda parte do programa.

Para realizar a operação de união (parte 2.1 do programa), utilizamos nosso conhecido método `concat()` seguido do método

`drop_duplicates()`. Como sabemos, o método `concat()` serve para concatenar DataFrames. Sendo assim, o comando `alunos = pd.concat([df_sql, df_python], ignore_index=True)` simplesmente gera um DataFrame com 2 + 3 = 5 linhas:

```
0        rakesh@xyz.com
1    ecg@acmecorpus.com
2           ana@xyz.com
3  jonas@acmecorpus.com
4        rakesh@xyz.com
```

Veja que existem duas linhas referentes ao e-mail "rakesh@xyz.com", pois esse é o único aluno que realizou os dois cursos (seu e-mail está contido nos dois DataFrames). Para manter apenas uma ocorrência deste e-mail, basta aplicar o método `drop_duplicates()` - que, como o seu próprio nome sugere, serve para remover linhas duplicadas. Desta forma, o comando `alunos = alunos.drop_duplicates()` produz como resultado o conjunto de todos os e-mails distintos:

```
0        rakesh@xyz.com
1    ecg@acmecorpus.com
2           ana@xyz.com
3  jonas@acmecorpus.com
```

Para realizar a operação de interseção (parte 2.2 do programa), utilizamos o método `merge()`. Trata-se de um poderoso método da pandas que é empregado principalmente para a implementar a operação de junção de DataFrames, como veremos em detalhes ainda neste capítulo. Por ora, basta saber que, quando este método é utilizado sem a especificação de nenhum parâmetro, em uma operação que envolva dois DataFrames compatíveis (com a mesma definição), o resultado produzido consiste na interseção dos conjuntos:

```
0  rakesh@xyz.com
```

Por fim, vamos falar da operação de diferença, que tem o código um pouco mais complicado. A diferença entre dois conjuntos `R` e `S` é a

operação que tem por objetivo é determinar os elementos de `R` que não fazem parte de `S`. Na pandas, é preciso utilizar o método `isin()` (apresentado no capítulo 2) em conjunto com a indexação booleana (também introduzida no capítulo 2) para que seja possível realizar essa operação. Tudo começa com o comando seguinte:

```
df_sql.email.isin(df_python.email)
```

Este comando produz uma Series booleana contendo o valor `False` associado a todos os e-mails do DataFrame `df_sql` que não façam parte de `df_python`. Isto é: ele vai associar `True` ao **índice** do e-mail "rakesh@xyz.com" e `False` para o índice de "ecg@acmecorpus.com".

```
0     True
1    False
```

De posse desse resultado, basta aplicar a indexação booleana sobre `df_sql` para que seja possível obter apenas os e-mails que foram associados ao valor `False` (ou seja, quem fez o curso de SQL, mas não fez o de Python). É exatamente isto que é feito no comando:

```
so_sql = df_sql[df_sql.email.isin(df_python.email)==False]`
```

Este comando gera o seguinte resultado:

```
1  ecg@acmecorpus.com
```

A obtenção dos alunos que cursaram Python, mas não SQL, é realizada de forma análoga, através do comando `so_python = df_python[df_python.email.isin(df_sql.email)==False]`. Ele produz o seguinte resultado:

```
0            ana@xyz.com
1  jonas@acmecorpus.com
```

Um detalhe muito importante sobre as operações de conjunto é que elas só fazem sentido quando utilizadas sobre DataFrames que possuam a mesma definição. Como sabemos, só podemos fazer a união, interseção e diferença de conjuntos matemáticos que

possuam o mesmo tipo de elemento (não faz sentido fazer a interseção, união ou diferença de um conjunto de e-mails com um conjunto de carros, por exemplo).

Comparando dois DataFrames

Uma outra operação simples - porém muitas vezes necessária, especialmente quando estamos lidando com bases de dados muito volumosas - consiste na comparação de dois DataFrames para determinar se eles são idênticos (armazenam o mesmo conteúdo) ou não. Neste caso, você deve utilizar o método `equals()`, conforme apresentado no exemplo a seguir. Este método retorna `True` quando os dois DataFrames envolvidos no teste forem iguais ou `False` se eles tiverem qualquer diferença, por mínima que seja.

```
#P42: Comparação de DataFrames
import pandas as pd

#(1)-Cria três DataFrames de Filmes
filmes1 = pd.DataFrame({"titulo":["O Filho da Noiva",
                                  "La La Land"],
                        "ano":[2001,
                               2017]})

filmes2 = pd.DataFrame({"titulo":["Noel, Poeta da Vila",
                                  "La La Land"],
                        "ano":[2007,
                               2017]})

filmes3 = pd.DataFrame({"titulo":["O Filho da Noiva",
                                  "La La Land"],
                        "ano":[2001,
                               2017]})

#(2)-Verifica quais DataFrames são iguais
#   (possuem o mesmo conteúdo)
print('filmes1 é igual à filmes2 -> ', filmes1.equals(filmes2))
print('filmes1 é igual à filmes3 -> ', filmes1.equals(filmes3))
```

Saída:

```
Filmes1 é igual à Filmes2 ->  False
Filmes1 é igual à Filmes3 ->  True
```

5.3 Junção

A operação de junção (*join* ou *merge*) é talvez a mais importante dentre as operações para combinação de DataFrames. As próximas seções introduzem os três tipos de junção diretamente disponibilizados pela pandas: junção natural, interna e externa.

Junção natural

A operação de junção natural realiza o **casamento** (*match*) de linhas de um DataFrame `R` com as linhas de outro DataFrame `S`. No entanto, a combinação é feita de forma **seletiva**. Isso significa que, por padrão, a junção natural combina apenas as linhas de `R` e `S` que coincidem em quaisquer atributos que são comuns a ambos os DataFrames. Para que o conceito fique claro, um exemplo é apresentado na figura a seguir.

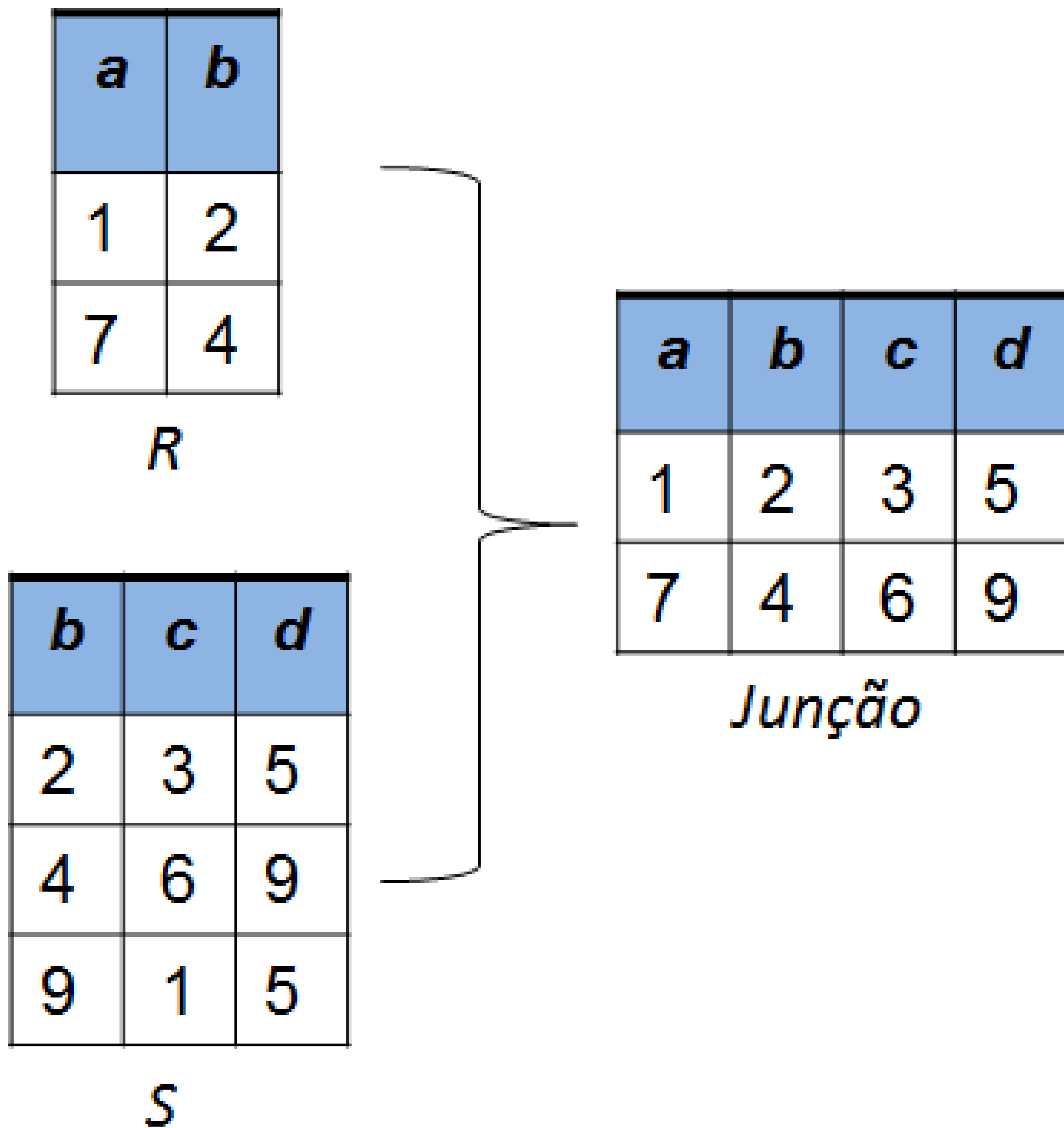

Figura 5.1: Operação de junção natural entre os DataFrames R e S.

Ao observar a figura, podemos constatar que o único atributo comum a `R` e `S` é o atributo `b`. Portanto, para uma linha de `R` poder casar com uma linha de `S`, é preciso que ambas possuam o mesmo valor para o atributo `b`. Em nosso exemplo, a primeira linha de `R` casa com a primeira de `S`, pois ambas compartilham o mesmo valor (valor 2) para o seu atributo em comum `b`. Esse casamento

gera a primeira linha do resultado da junção: (1, 2, 3, 5). O processo é ilustrado na figura seguinte.

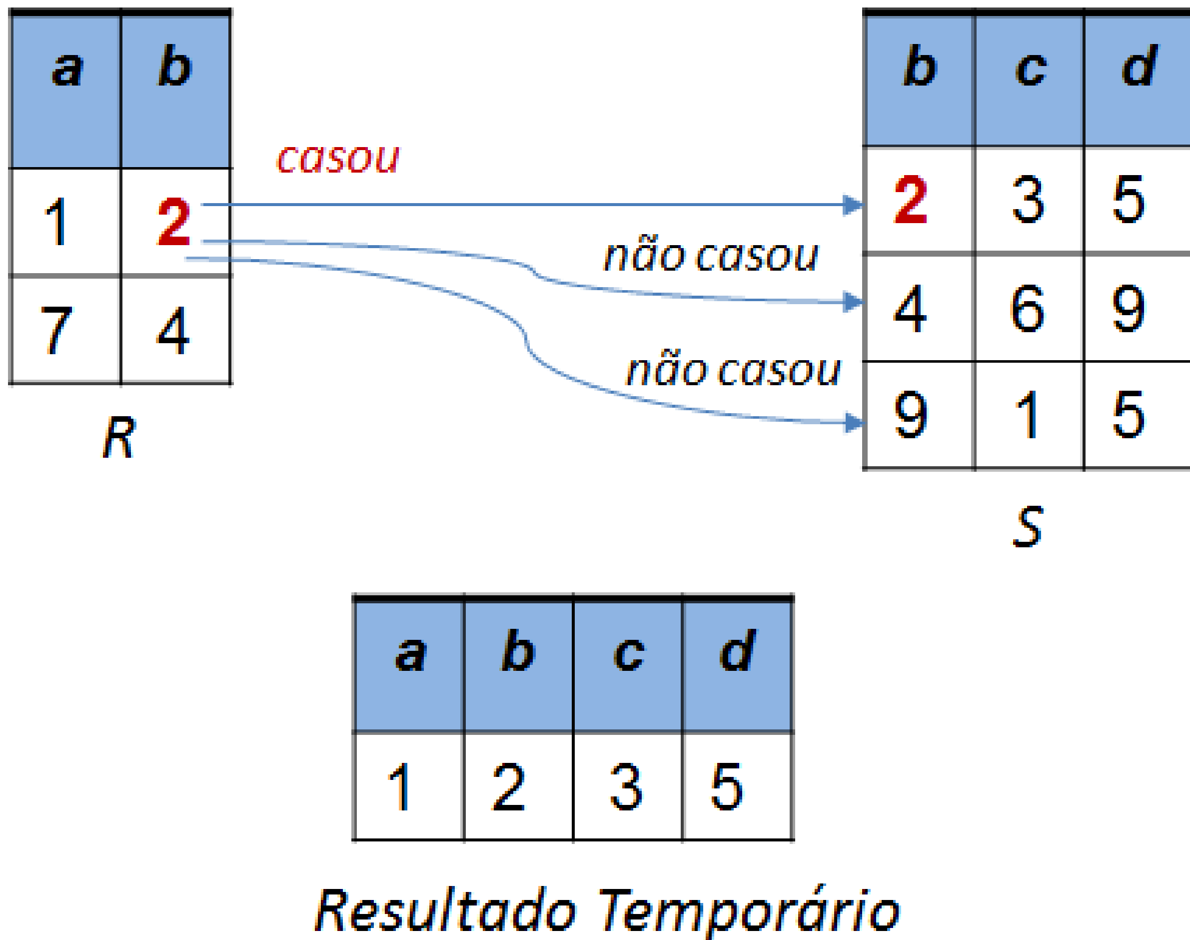

Figura 5.2: Etapa 1 da junção natural entre os DataFrames R e S: casamento da primeira linha de R.

Já a segunda linha de `R` casa com a segunda linha de `S`, uma vez que elas compartilham o mesmo valor (valor 4) para o atributo em comum `b`. Esse casamento gera a segunda linha do resultado, (7, 4, 6, 9), conforme ilustrado na figura seguinte. Este é o resultado final da operação de junção, uma vez que todas as linhas de `R` já foram comparadas com todas as de `S`.

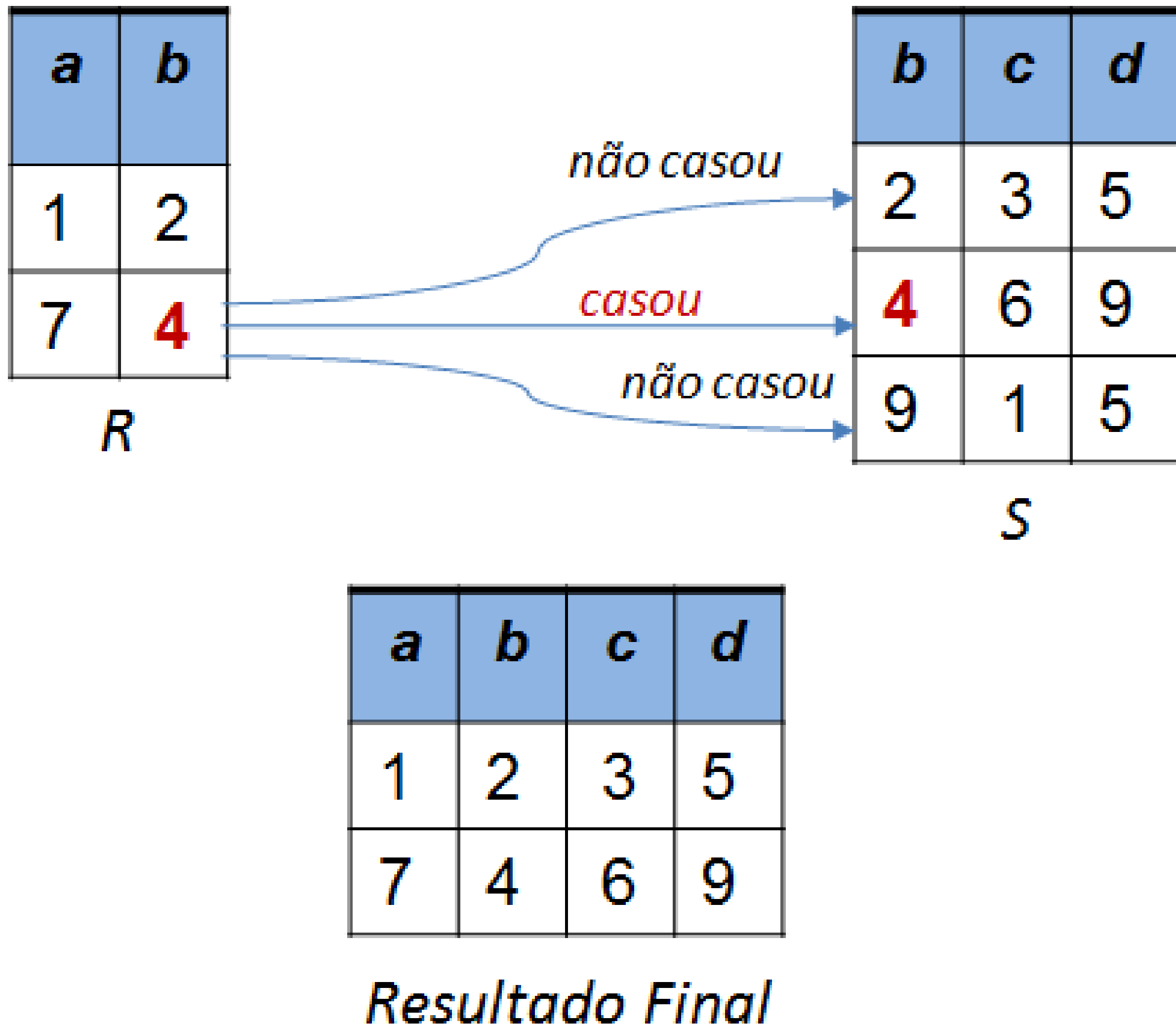

Figura 5.3: Etapa 2 da junção natural entre os DataFrames R e S: casamento da segunda e última linha de R).

Observações:

- No exemplo apresentado, a terceira linha de `S` não casa com nenhuma linha de `R`, pois não há nenhuma linha em `R` com o valor 9 para o atributo em comum `b`. Desta forma, a terceira linha de `S` não tem nenhum efeito no resultado final da operação de junção (ela não é levada para o resultado).
- O atributo comum aos DataFrames é chamado de **atributo de ligação** ou **chave de ligação** ou, simplesmente, **chave** (*key*).

- Enquanto a concatenação produz como resultado um DataFrame mais "comprido" (com linhas empilhadas) a junção produz um DataFrame mais "largo" (que possui atributos de dois DataFrames).

O método `merge()` é utilizado na pandas para implementar todos os tipos de operação de junção. O programa a seguir reproduz o exemplo da junção natural entre os DataFrames `R` e `S` que acabamos de apresentar.

```
#P43: Junção Natural
import pandas as pd

#(1)-Cria os DataFrames R e S
R = pd.DataFrame({"a":[1,7],
                  "b":[2,4]})

S = pd.DataFrame({"b":[2,4,9],
                  "c":[3,6,1],
                  "d":[5,9,5]})



#(2)-Efetua a operação de junção
juncao_natural = pd.merge(R, S)

print('------------------------------')
print("R:")
print(R)
print('------------------------------')
print("S:")
print(S)
print('------------------------------')
print("junção natural entre R e S:")
print(juncao_natural)
```

Resultado:

```
R:
   a  b
```

```
0  1  2
1  7  4
------------------------------
S:
   b  c  d
0  2  3  5
1  4  6  9
2  9  1  5
------------------------------
junção natural entre R e S:
   a  b  c  d
0  1  2  3  5
1  7  4  6  9
```

Conforme mostrado no programa, a sintaxe do método `merge()` para realizar a junção interna de `R` e `S` é trivial. Basta especificar um par de DataFrames como parâmetros, que o método `merge()` se encarrega de identificar, sozinho, qual é o atributo comum a ambos e, assim, processar a junção. Veja que no resultado final o atributo `b` (o único comum aos dois esquemas) aparece apenas uma vez.

Entretanto, também é possível indicar explicitamente qual é o atributo de ligação, bastando para isso utilizar o parâmetro `on`.

```
pd.merge(R, S, on="b")
```

Esse parâmetro é útil em situações em que os DataFrames que são alvo da operação de junção possuam mais de um atributo de mesmo nome, sendo que apenas um deles representa a chave de ligação. Além disso, o parâmetro `on` torna o programa mais legível, pois deixa o atributo de ligação explícito para qualquer pessoa que venha a examinar o código.

Junção interna

Na junção natural, para um atributo ser considerado comum a `R` e `S`, ele precisa ter o **mesmo nome** nestes dois DataFrames. No entanto, isso nem sempre acontece quando estamos lidando com bases de dados reais. Na figura a seguir, um exemplo que ilustra

esse tipo situação é apresentado. Temos dois DataFrames chamados `depto` (departamento) e `emp` (empregados), que possuem o **id do departamento** como atributo de ligação. No entanto, este atributo de ligação possui o nome diferente em cada DataFrame. Veja que ele se chama `id` em `depto` e `idDepto` no DataFrame `emp` .

depto

Id	NomDepto	Local
D1	Compras	SP
D2	RH	RJ
D3	TI	RJ
D4	Vendas	SP

emp

Num	Nome	Salario	IdDepto
3199	Ana	1600	D2
3269	David	2975	D3
3555	José	1500	
3788	Marina	5000	D2
3844	Luís	3000	D4

Figura 5.4: DataFrames com dados de Departamentos (depto) e Empregados (emp).

Felizmente, a pandas também permite com que seja efetuada a **junção interna** (*inner join*) de DataFrames, um tipo que não exige que a chave de ligação possua nome igual nos DataFrames cuja junção será feita.

Na junção interna, você deve indicar explicitamente a **condição de junção** para o método `merge()`. Isso significa especificar quais são os atributos de ligação de cada DataFrame, utilizando para tal os parâmetros `left_on` e `right_on`. A seguir, o uso destes parâmetros é demonstrado em um programa que implementa a junção interna com o objetivo de combinar os dados dos funcionários e com os dados de seus departamentos.

```
#P44: Junção Interna
import pandas as pd

#(1)-Cria os DataFrames depto e emp
dic_depto = {"id":["D1","D2","D3","D4"],
             "nomDepto": ["Compras","RH","TI","Vendas"],
             "local":["SP","RJ","RJ","SP"]
             }

dic_emp = {"num":[3199,3269,3555,3788,3844],
           "nome": ["Ana","David","José","Marina","Luís"],
           "salario":[1600,2975,1500,5000,3000],
           "idDepto": ["D2","D3",None,"D2","D4"]
           }


depto = pd.DataFrame(dic_depto)
emp = pd.DataFrame(dic_emp)


#(2)-Efetua a operação de junção
juncao_interna = pd.merge(emp, depto, left_on="idDepto", right_on="id")

print('-------------------------------')
print("depto:")
print(depto)
```

```
print('-----------------------------')
print("emp:")
print(emp)
print('-----------------------------')
print("junção interna:")
print(juncao_interna)
```

Aqui está a saída gerada. Todos os quatro empregados com departamento cadastrado tiveram os seus dados corretamente combinados com os dados de seus respectivos departamentos:

```
-----------------------------
depto:
   id nomDepto local
0  D1  Compras    SP
1  D2       RH    RJ
2  D3       TI    RJ
3  D4   Vendas    SP
-----------------------------
emp:
    num    nome  salario idDepto
0  3199     Ana     1600      D2
1  3269   David     2975      D3
2  3555    José     1500    None
3  3788  Marina     5000      D2
4  3844    Luís     3000      D4
-----------------------------
junção interna:
    num    nome  salario idDepto  id nomDepto local
0  3199     Ana     1600      D2  D2       RH    RJ
1  3788  Marina     5000      D2  D2       RH    RJ
2  3269   David     2975      D3  D3       TI    RJ
3  3844    Luís     3000      D4  D4   Vendas    SP
```

Conceitualmente, a junção interna funciona da mesma forma que a junção natural, ou seja, ela combina uma linha de um DataFrame `R` com uma linha de um DataFrame `S` apenas quando há coincidência nos valores das colunas especificadas na condição de junção. Para

realizar a junção interna entre dois DataFrames, basta seguir sempre a mesma receita:

- Especificar os nomes dos DataFrames envolvidos na junção separados por vírgula, dentro do método `merge()` (por exemplo, `depto,emp`).
- Utilizar o parâmetro `left_on` indicando o nome do atributo de ligação no DataFrame que foi especificado à esquerda (em nosso exemplo, `emp` foi especificado antes de `depto`, logo `emp` é considerado o DataFrame à esquerda).
- Utilizar o parâmetro `right_on` indicando o nome do atributo de ligação no DataFrame que foi especificado à direita.

Conforme mostra a saída do programa, cada linha produzida como resultado na junção interna entre `emp` e `depto` contém todos os atributos de `emp` e todos os de `dept`, incluindo as chaves de ligação `idDepto` e `id`. Para selecionar apenas os atributos de interesse, basta aplicar a técnica de fatiamento, introduzida no capítulo 3. No exemplo a seguir, mostra-se como empregar o fatiamento para manter apenas os atributos `num`, `nome` e `nomDepto` (esta técnica também é chamada de **projeção de atributos**, como veremos no próximo capítulo):

```
fatia = juncao_interna[['num','nome','nomDepto']]
print(fatia)
```

```
   num    nome nomDepto
0  3199     Ana       RH
1  3788  Marina       RH
2  3269   David       TI
3  3844    Luís   Vendas
```

Junção externa

Você deve ter percebido que em nossa base de dados exemplo existe um empregado sem departamento cadastrado ("José") e também um departamento que não possui nenhum funcionário ("D1"). As linhas referentes ao empregado e departamento citados

acabaram não fazendo parte do resultado produzido pela pela junção interna.

Mas e se houvesse a necessidade de efetuar um estudo envolvendo qualquer empregado, possuindo ele departamento ou não? Ou um relatório que abrangesse qualquer departamento, mesmo aqueles que ainda não tenham nenhum funcionário alocado? Nesse caso, seria preciso utilizar outro tipo de junção, conhecida como **junção externa**.

As junções externas conectam linhas de dois DataFrames de uma forma mais inclusiva: elas produzem os mesmos resultados da junção interna **acrescidos** de linhas não casadas de um ou ambos os DataFrames. Existem três tipos de junção externa: *left join* (a mais usada), *right join* e *full join*.

A operação de *left join* retorna todas as linhas do DataFrame especificado à esquerda no comando `merge()`, mesmo que não exista casamento (valor equivalente) no DataFrame à direita. Para que o conceito fique claro, vamos retornar para o nosso exemplo envolvendo os DataFrames `emp` e `depto`. O empregado "José" não possui um departamento cadastrado (o atributo `idDepto` possui valor `None` para a linha referente a este empregado em `emp`). Se desejarmos produzir um DataFrame contendo todos os empregados combinados com os dados de seus respectivos departamentos, mas que também inclua os empregados sem departamento cadastrado, é preciso fazer uso do *left join* como mostrado a seguir:

```
j_esq = pd.merge(emp, depto, how="left", left_on="idDepto", right_on="id")
```

O resultado da operação é apresentado adiante:

	num	nome	salario	idDepto	id	nomDepto	local
0	3199	Ana	1600	D2	D2	RH	RJ
1	3269	David	2975	D3	D3	TI	RJ
2	3555	José	1500	None	NaN	NaN	NaN
3	3788	Marina	5000	D2	D2	RH	RJ
4	3844	Luís	3000	D4	D4	Vendas	SP

- O parâmetro `how="left"` é usado para indicar à pandas que se deseja realizar o *left join*.
- No resultado, observe que existem quatro linhas idênticas às que foram geradas pela junção interna, referentes aos empregados "Ana", "David", "Marina" e "Luís". A única linha extra é a do empregado "José", onde os dados deste (dados vindos de `emp`) foram combinados com `NaN` (dados que viriam de `depto` caso José tivesse algum departamento cadastrado).

A junção do tipo *right join* retorna todas as linhas do DataFrame à direita, mesmo que não exista casamento (valor equivalente) no DataFrame à esquerda. Desta forma, para produzir um DataFrame com todos os empregados combinados com os dados de seus departamentos, mas incluindo os empregados sem departamento cadastrado, seria necessário montar o `merge()` da forma indicada a seguir, invertendo a posição de `emp` e `depto` .

```
j_dir = pd.merge(depto, emp, how="right", left_on="id",
right_on="idDepto")
```

Resultado do *right join*:

```
    id nomDepto local   num    nome  salario idDepto
0   D2       RH    RJ  3199     Ana     1600      D2
1   D2       RH    RJ  3788  Marina     5000      D2
2   D3       TI    RJ  3269   David     2975      D3
3   D4   Vendas    SP  3844    Luís     3000      D4
4  NaN      NaN   NaN  3555    José     1500    None
```

- O parâmetro `how="right"` é usado para indicar à pandas que se deseja realizar o *right join*.
- Embora o resultado seja conceitualmente equivalente ao do *left join*, uma diferença é que no DataFrame resultante os dados vindos de `depto` são colocados antes daqueles vindos de `emp` , uma vez que `depto` foi especificado antes de `emp` no `merge()` .

Para terminar, vamos apresentar o *full join*, que junta em uma única operação os resultados do *left join* e do *right join*. Esse tipo de

junção externa retorna todas as linhas do DataFrame à esquerda e todas as linhas do DataFrame à direita devidamente combinadas, de acordo com a especificação da condição de junção. Caso existam linhas no DataFrame à esquerda que não casem com qualquer linha do DataFrame à direita, elas serão levadas para o resultado final. E caso existam linhas no DataFrame à direita que não casem com qualquer linha do DataFrame à esquerda, elas também serão levadas para o resultado final.

```
j_full = pd.merge(emp, depto, how="outer", left_on="idDepto",
right_on="id")
```

Aqui está o resultado produzido:

```
      num    nome  salario idDepto   id nomDepto local
0  3199.0     Ana   1600.0      D2   D2       RH    RJ
1  3788.0  Marina   5000.0      D2   D2       RH    RJ
2  3269.0   David   2975.0      D3   D3       TI    RJ
3  3555.0    José   1500.0    None  NaN      NaN   NaN
4  3844.0    Luís   3000.0      D4   D4   Vendas    SP
5     NaN     NaN      NaN     NaN   D1  Compras    SP
```

- O parâmetro `how="outer"` é usado para indicar à pandas que se deseja realizar o *full join* (esse tipo de junção é também conhecida como *full outer join*).
- No resultado gerado, veja que foram incluídos tanto o empregado sem departamento ("José"), como o departamento que não está associado a nenhum empregado ("D1").

JOIN() *VERSUS* MERGE()

Além do `merge()`, a pandas oferece outro método para junção de DataFrames, denominado `join()`. Entretanto, esse método é um pouco mais simples (possui menos parâmetros) e costuma ser mais utilizado para realizar a junção através de **índices** em vez de colunas. Para maiores informações consulte a documentação oficial da pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.join.html.

5.4 Projeto prático — combinando os datasets flags e countries

Nesta seção, veremos como realizar a junção da base de dados `flags` com uma outra base de dados, montada com dados provenientes do arquivo `countries-of-the-world` (https://www.kaggle.com/fernandol/countries-of-the-world). O nosso objetivo será descobrir quais são os países comuns a estas duas fontes de informação.

Como você já sabe, `flags` é a base de dados de nosso projeto prático, que armazena dados reais de países e suas bandeiras. Nos últimos capítulos, demonstramos a aplicação de diversos conceitos importantes sobre Data Wrangling - desde a computação vetorizada até a indexação booleana - utilizando exemplos que exploraram esta base de dados. Ela também será utilizada em um exemplo de Machine Learning a ser apresentado no capítulo final do livro.

Apesar de `flags` ser muito interessante, ela é uma base de dados bem antiga, tendo sido originalmente criada para auxiliar o treinamento de algoritmos de Inteligência Artificial ainda na década de 1980. Considerando este fato, nesta seção faremos uma comparação dos países presentes na `flags` com aqueles constantes

na base `countries-of-the-world`, uma vez que esta última armazena informações atuais. Você verá que, com o uso da operação de junção, é muito fácil combinar essas bases e determinar as divergências entre ambas. O exemplo utiliza os arquivos `flags.csv` (nosso velho conhecido) e `countries.csv` (base montada a partir de `countries-of-the-world`), ambas disponibilizadas no **repositório de bases de dados** de nosso livro. O endereço do repositório é: https://github.com/edubd/pandas. A base `countries.csv` possui apenas três atributos: `country` (nome do país), `pop_country` (população em número de pessoas) e `area_country` (área do país em quilômetros quadrados). Os seus dez primeiros registros são apresentados a seguir:

```
country,pop_country,area_country
Afghanistan,31056997,647500
Albania,3581655,28748
Algeria,32930091,2381740
American-Samoa,57794,199
Andorra,71201,468
Angola,12127071,1246700
Anguilla,13477,102
Antigua-Barbuda,69108,443
Argentina,39921833,2766890
Armenia,2976372,29800
...
```

A chave de ligação entre `flags.csv` e `countries.csv` é o nome do país (atributo denominado `name` em `flags` e `country` em `countries-of-the_world`). Através dessa chave, o programa a seguir combina as bases para atingir os seguintes objetivos:

- Verificar quais países fazem parte de ambas as bases, usando a junção interna;
- Verificar quais países fazem parte apenas de `flags.csv`, utilizando o *left join*;
- Verificar quais países fazem parte apenas de `countries.csv`, utilizando o *right join*;

É importante observar que não podemos resolver essas questões utilizando diretamente as operações de conjunto, pois o par de bases de dados envolvido não é compatível. Então, vamos mostrar uma maneira alternativa, que envolve o uso do método `merge()` .

```
#P45: Comparação entre as bases flags.csv e countries.csv
import pandas as pd

#-------------------------------------------------------
#(1)-Importa as bases de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')
countries = pd.read_csv('c:/bases/countries.csv')

num_linhas_flags = flags.shape[0]
num_linhas_countries = countries.shape[0]

#-------------------------------------------------------
#(2)-Realiza a junção entre os DataFrames
#-------------------------------------------------------

#2.1 - Junção Interna: países comuns a ambas as bases
ambas = pd.merge(flags, countries,
                 how="inner",
                 left_on="name", right_on="country")

num_linhas_ambas = ambas.shape[0]

#2.2 - Left Join: países apenas em flags
so_flags = pd.merge(flags, countries,
                 how="left",
                 left_on="name", right_on="country")

so_flags = so_flags[pd.isnull(so_flags['country'])==True]
num_linhas_so_flags = so_flags.shape[0]

#2.3 - Right Join: países apenas em countries
so_countries = pd.merge(flags, countries,
                 how="right",
```

```
                 left_on="name", right_on="country")

so_countries = so_countries[pd.isnull(so_countries['name'])==True]
num_linhas_so_countries = so_countries.shape[0]

#-------------------------------------------------------
#(3)-Imprime os resultados
#-------------------------------------------------------
print('- Num países em "flags":', num_linhas_flags)
print('- Num países em "countries":', num_linhas_countries)
print('- Num países em ambas:', num_linhas_ambas)
print('- Num países só em "flags":', num_linhas_so_flags)
print('- Num países só em "countries":', num_linhas_so_countries)
print('------------------------------')
print("países só em flags:")
print(so_flags['name'])
print('------------------------------')
print("países só em countries:")
print(so_countries['country'])
```

O resultado mostra que a maioria dos países é comum a ambas as bases. No entanto, também existem países que estão armazenados em apenas uma delas (alguns resultados foram suprimidos para que a listagem não ficasse muito longa):

```
- Num países em "flags": 194
- Num países em "countries": 227
- Num países em ambas: 177
- Num países só em "flags": 17
- Num países só em "countries": 50
------------------------------
países só em flags:
9                Argentine
56      Falklands-Malvinas
65             Germany-FRG
88             Ivory-Coast
92               Kampuchea
103               Malagasy
109               Marianas
```

```
127                  Niue
135               Parguay
155           South-Yemen
163               Surinam
182       US-Virgin-Isles
184                  USSR
186          Vatican-City
189         Western-Samoa
190            Yugoslavia
191                 Zaire
Name: name, dtype: object
------------------------------
países só em countries:
177               Armenia
178                 Aruba
179            Azerbaijan
180               Belarus
181    Bosnia-Herzegovina
182              Cambodia
183   Congo-Repub. of the
184         Cote d'Ivoire
185               Croatia
186           East Timor
187               Eritrea
188               Estonia
189            Gaza Strip
...    ...
225             West Bank
226        Western Sahara
Name: country, dtype: object
```

O programa funciona da seguinte forma. Na primeira parte, apenas realizou-se a importação das duas bases utilizando o método `read_csv()`. Já a segunda parte é a que realiza as operações de junção interna e externa para comparar as bases.

Na seção 2.1 do código, a junção interna foi efetuada para que os países presentes nos dois DataFrames fossem devidamente combinados em um novo DataFrame chamado `ambas`.

```
ambas = pd.merge(flags, countries,
                 how="inner",
                 left_on="name", right_on="country")
```

Veja que a condição de junção é baseada no nome do país. Neste exemplo, o uso do parâmetro `how=inner` é opcional, uma vez que, por padrão, o método `merge()` realiza a junção interna.

Ainda na seção 2.1, logo depois de realizar a junção interna, recuperamos o número de linhas que foram incluídas no DataFrame `ambas` empregando o comando: `num_linhas_ambas = ambas.shape[0]`. O total de linhas é 177, ou seja, este é o número de países em ambas as bases.

Na seção 2.2, o *left join* foi efetuado em conjunto com a indexação booleana para que fosse possível determinar os países presentes apenas em flags:

```
so_flags = pd.merge(flags, countries,
                 how="left",
                 left_on="name", right_on="country")

so_flags = so_flags[pd.isnull(so_flags['country'])==True]
```

A operação de *left join* gerou o DataFrame `so_flags`, contendo os países comuns a ambos os DataFrames e também todas as linhas do DataFrame `flags`, mesmo quando não existe casamento (país equivalente) no DataFrame `countries`. Em seguida, a indexação booleana foi efetuada para manter apenas as linhas com valor `null` para o atributo `country`. Isto é: ficamos apenas com as linhas que vieram somente de `flags`.

A seção 2.3 é análoga. Nela, o *right join* e a indexação booleana são empregados para determinar os países que estão apenas no DataFrame `countries` (aqueles que ficam com valor `null` para o atributo `name` depois da execução do *right join*).

```
so_countries = pd.merge(flags, countries,
                 how="right",
```

```
                      left_on="name", right_on="country")

so_countries = so_countries[pd.isnull(so_countries['name'])==True]
```

Na terceira parte do programa, todos os resultados computados são impressos, incluindo a listagem completa de países que fazem parte apenas de `flags` e dos que fazem parte apenas de `countries`.

E agora que você já sabe selecionar e combinar bases de dados, além de ter aprendido as técnicas básicas para estudar atributos (produção de gráficos e estatísticas), está faltando aprender sobre apenas mais duas atividades importantes de Data Wrangling: a limpeza e a transformação de dados. Mas não se preocupe, pois estes são exatamente os assuntos cobertos em nosso próximo capítulo.

CAPÍTULO 6
Transformação e limpeza de DataFrames

A transformação e a limpeza de DataFrames são duas atividades de pré-processamento relacionadas, que possuem grande importância para a ciência de dados. Isso porque, em geral, os dados utilizados em processos de ciência de dados são coletados a partir de diferentes fontes, variando desde os bancos de dados de sistemas corporativos até dados obtidos na internet. Desse modo, é muito comum ocorrerem situações em que os dados de uma das fontes não sejam compatíveis com os da outra (elas podem, por exemplo, adotar códigos distintos para representar uma mesma informação). Além disso, dados do mundo real costumam apresentar diferentes tipos de inconsistências, tais como valores ausentes, informações incompletas ou valores de atributos fora do padrão exigido por um algoritmo de ciência de dados. O objetivo das técnicas de transformação e limpeza de dados é exatamente combater estes problemas, ou seja, padronizar os dados e dar algum tipo de tratamento para os valores ausentes ou incompletos.

Este capítulo apresenta um estudo das principais funcionalidades oferecidas pela pandas para a limpeza e transformação de dados, cobrindo os seguintes tópicos:

- Seleção e projeção
- Modificação
- Formatação
- Discretização
- Normalização

Com o objetivo de facilitar a apresentação dos conceitos, todos os exemplos apresentados no capítulo fazem uso da base de dados `flags` — nossa base de dados com informações sobre nações e suas bandeiras. Adicionalmente, na seção final do capítulo apresenta-se um programa que aplica um conjunto de técnicas de transformação

de dados sobre esta base com o intuito de padronizar os seus atributos.

6.1 Seleção e projeção

A seleção e a projeção são as duas operações mais simples e populares para transformação de dados. Ambas são utilizadas para extrair partes de um DataFrame baseando-se no emprego da técnica de fatiamento (já apresentada em diversos exemplos de capítulos anteriores deste livro). As próximas seções explicam como trabalhar com estas operações na pandas.

Seleção

A seleção consiste na operação que gera um novo DataFrame a partir da **extração de algumas linhas** de interesse de um outro DataFrame. Essa operação também é conhecida como **filtragem** de linhas.

Embora os DataFrames possam ser filtrados com o emprego de diferentes técnicas, a mais utilizada na prática é a indexação booleana, provavelmente por ser mais intuitiva. Nesta abordagem, passamos uma Series de elementos booleanos (`True` / `False`) para o DataFrame, com o objetivo de obter apenas as linhas associadas ao valor `True` . No programa a seguir, a técnica é empregada para listar os países da Oceania que fazem parte da base `flags` .

```
#P46: Seleção
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Seleciona apenas as linhas dos países da oceania
v = (flags['landmass']==6)
```

```
flags_oceania = flags[v]

#3-imprime os países da oceania
print(flags_oceania)
```

O resultado indica que há 20 países da Oceania presentes em `flags`:

```
                 name  landmass  zone    ...     botright
3      American-Samoa         6     3    ...     red
10          Australia         6     2    ...     blue
41       Cook-Islands         6     3    ...     blue
57               Fiji         6     2    ...     blue
61   French-Polynesia         6     3    ...     red
71               Guam         6     1    ...     red
82          Indonesia         6     2    ...     white
94           Kiribati         6     1    ...     blue
109          Marianas         6     1    ...     blue
113        Micronesia         6     1    ...     blue
119             Nauru         6     2    ...     blue
123       New-Zealand         6     2    ...     blue
127              Niue         6     3    ...     gold
134  Papua-New-Guinea         6     2    ...     black
137       Philippines         6     1    ...     red
151    Soloman-Islands        6     2    ...     green
172             Tonga         6     2    ...     red
177            Tuvalu         6     2    ...     blue
185           Vanuatu         6     2    ...     green
189     Western-Samoa         6     3    ...     red
```

Agora a explicação sobre o programa. Na base de dados `flags`, o atributo `landmass` é utilizado para indicar o continente de cada país (recorde que a relação completa dos atributos de `flags` foi apresentada na última seção do capítulo 3). Este atributo utiliza o código `6` para representar o continente Oceania. Com isto, para filtrar as linhas dos países da Oceania foi preciso escrever somente dois comandos:

- `v = (flags['landmass']==6)`: gera uma Series chamada `v`, de elementos booleanos, que associará o valor `True` às linhas onde

`landmass` possui o valor 6. Veja que o teste de igualdade é realizado com o uso do operador `==` (dois símbolos de `=` e não apenas um!);

- `flags_oceania = flags[v]` : comando de atribuição que gera o DataFrame `flags_oceania`. Este conterá apenas as linhas em que `v` é `True`.

Simples, não? É importante dizer que os dois comandos poderiam ser condensados em um único, conforme indicado a seguir:

```
flags_oceania = flags[flags['landmass']==6]
```

Alguns programadores preferem fazer a filtragem em dois passos, pois assim o código fica mais legível. De maneira oposta, há quem ache melhor implementar tudo em uma única linha, para tornar o programa mais conciso. O importante é que, independente da abordagem adotada, a seleção funcionará do mesmo jeito!

A tabela a seguir apresenta os **operadores de comparação** que podem ser utilizados para a construção de condições de filtragem. Note que o operador `==` ("igual a") foi utilizado em nosso último exemplo.

Operador	Significado
`==`	igual a
`!=`	diferente
`>`	maior do que
`>=`	maior ou igual a
`<`	menor do que
`<=`	menor ou igual a

A próxima tabela apresenta os **operadores lógicos** disponíveis na pandas. Eles são empregados para permitir a elaboração de

condições de filtragem complexas, envolvendo mais de um teste lógico.

Operador	Significado
`&`	AND — retorna `True` se todas as condições avaliadas forem verdadeiras.
|	OR — retorna `True` se ao menos uma das condições avaliadas forem verdadeiras.
`!`	NOT — retorna `True` se a condição seguinte for `False`.

A seguir, são apresentados e comentados diversos exemplos de utilização prática para os operadores lógicos e de comparação. Substitua a condição `(flags['landmass']==6)` por cada um dos exemplos adiante para que você possa executá-los e verificar os resultados retornados.

- `(flags['landmass']!=6`): países que **não** são da Oceania;
- `(flags['colours'] <=2)` : países com no máximo 2 cores em sua bandeira;
- `(flags['language'] ==1) | (flags['language'] ==4)` : países cujo idioma predominante é o inglês ou o alemão;
- `(flags['landmass'] ==6) & (flags['area'] >200)` : países da Oceania, com área acima de 200 mil quilômetros quadrados.
- `(flags['landmass'] ==6) & (flags['area'] >200) & (flags['language'] ==1)` : países da Oceania, com área acima de 200 mil quilômetros quadrados, em que o idioma predominante é o inglês.

Projeção

A projeção é a operação que gera um novo DataFrame a partir da **extração de algumas colunas** de um outro DataFrame. É uma operação ainda mais simples do que a seleção, uma vez que para implementá-la basta especificar uma lista de nomes de atributos.

No exemplo a seguir, a projeção é utilizada para extrair apenas as colunas `name`, `colours`, `language`, `landmass` e `area` dos países da Oceania com área acima de 200 mil quilômetros quadrados.

```
#P47: Projecao
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Seleciona apenas as linhas dos países da oceania
#  com área acima de 200 mil quilômetros quadrados
v =(flags['landmass'] ==6) & (flags['area'] >200)
df = flags[v]

#3-Projeta apenas as colunas "name", "colours", "language", "landmass" e
"area"
df = df[['name','colours','language','landmass','area']]

#4-Imprime o resultado
print(df)
```

Saída:

```
                 name  colours  language  landmass  area
10          Australia        3         1         6  7690
82          Indonesia        2        10         6  1904
123       New-Zealand        3         1         6   268
134  Papua-New-Guinea        4         1         6   463
137       Philippines        4        10         6   300
```

Veja que no programa apresentado, primeiro executamos uma operação de seleção (passo 2) e depois a projeção (passo 3). No entanto, é importante deixar claro que isto não é obrigatório. Se você quiser apenas projetar (sem selecionar) não há qualquer problema. Observe ainda que o DataFrame resultante da projeção conterá as colunas na ordem especificada na lista.

FATIAMENTO

A seleção e a projeção nada mais são do que casos particulares de emprego da técnica de fatiamento de DataFrames, introduzida no capítulo 3. No entanto, pelo fato de serem muito intuitivas e bastante utilizadas na prática, elas costumam ser referenciadas por seus nomes (seleção ou projeção em vez de fatiamento).

Recriando os índices

Você deve ter percebido que a operação de seleção gera um novo DataFrame com elementos que **preservam os índices** do DataFrame original. Por exemplo, no último programa apresentado, que executou a seleção das linhas correspondentes aos países da oceania com área acima de 200 mil quilômetros quadrados, obtivemos um DataFrame `df` com 5 elementos de índices `[10,82,123,134,137]` em vez de `[0,1,2,3,4]`.

Entretanto, é muito simples recriar (ou resetar) os índices no DataFrame resultante da seleção. Para tal, basta fazer uso do método `reset_index()`, conforme exemplificado a seguir:

```
#P48: Método `reset_index()`
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Seleciona apenas as linhas dos países da oceania
#  com área acima de 200 mil quilômetros quadrados
v =(flags['landmass'] ==6) & (flags['area'] >200)
df = flags[v]

#3-Projeta apenas as colunas "name", "colours", "language",
#  "landmass" e "area"
df = df[['name','colours','language','landmass','area']]
```

```
#4-reseta os índices
df = df.reset_index(drop=True)

#5-Imprime o resultado
print(df)
```

O parâmetro `drop=True` é utilizado para evitar que a pandas gere uma coluna extra com o índice anterior. O resultado do programa é apresentado a seguir:

```
               name  colours  language  landmass  area
0         Australia        3         1         6  7690
1         Indonesia        2        10         6  1904
2       New-Zealand        3         1         6   268
3  Papua-New-Guinea        4         1         6   463
4       Philippines        4        10         6   300
```

Para maiores informações sobre o método `reset_index()`, consulte o link a seguir: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html.

6.2 Modificação de dados

Nesta seção, abordaremos as três operações para modificar o conteúdo de um DataFrame: **inserção**, **atualização** e **exclusão**. Embora este tema já tenha sido introduzido no capítulo 3, chegou a hora de complementá-lo, através da apresentação de técnicas mais avançadas.

Inserção

Inserção consiste na operação de inserir um ou mais objetos (novas linhas) em um DataFrame. Na pandas, essa tarefa é realizada de maneira trivial: basta especificar os dados dos objetos em um novo DataFrame e depois utilizar o método `concat()` para concatená-lo ao DataFrame alvo da modificação. No exemplo a seguir, mostramos como o processo é feito para inserir os dados do país "East Timor" (Timor Leste) no DataFrame `flags` :

```
#P49: Inserção
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Cria um DataFrame com os dados do novo país
df_novo = pd.DataFrame(
        {
        'name': 'East Timor',
        'landmass': 5,
        'area': 15,
        'language': 10,
        }, index=[0])

#3-Insere o novo país em flags
flags = pd.concat([flags,df_novo], ignore_index=True, sort=False)

#4-Imprime o DataFrame alterado
print(flags[['name',
            'landmass',
            'area',
            'language',
            'religion'
            ]])
```

O resultado é apresentado a seguir (nesta listagem e na maioria das demais apresentadas ao longo do capítulo, serão mostradas apenas as três primeiras e as três últimas linhas produzidas no resultado):

```
                 name  landmass  area  language  religion
0         Afghanistan         5   648        10       2.0
1             Albania         3    29         6       6.0
2             Algeria         4  2388         8       2.0
...
192            Zambia         4   753        10       5.0
193          Zimbabwe         4   391        10       5.0
194       East Timor          5    15        10       NaN
```

Neste exemplo, após a importação da base `flags`, criou-se um novo DataFrame, denominado `df_novo`. Para simplificar o exemplo, fornecemos dados de apenas quatro atributos (`name`, `landmass`, `area` e `language`) e de um único objeto ("East Timor"). No entanto, poderíamos ter especificado quantos objetos ou atributos desejássemos. De posse de um DataFrame alvo (`flags`) e de um DataFrame de transações (`df_novo`), basta utilizar o nosso conhecido método `concat()` para concatenar (inserir) os dados do DataFrame de transações ao DataFrame alvo. O parâmetro `ignore_index=True` foi utilizado para forçar a atribuição de um novo índice para o "East Timor". Por sua vez, o parâmetro `sort=False` foi utilizado para fazer com que "East Timor" fosse inserido no final de flags (como último registro).

No passo final do programa, utilizamos a operação de projeção para imprimir cinco atributos escolhidos de `flags`. Veja que o atributo `religion`, que não foi especificado para o novo objeto "East Timor", fica automaticamente com o valor `NaN` após a inserção.

Atualização

Atualização consiste na operação de alterar os valores de alguns atributos de um objeto existente. Para alterar um atributo específico de uma linha **através de seu índice**, basta utilizar o método `at()`. No exemplo a seguir, o valor do atributo `religion` é alterado para 0 na linha referente ao Timor Leste (índice=194).

```
flags.at[194,'religion']=0
```

Entretanto, utilizar o valor do índice para guiar a operação de modificação nem sempre é possível ou simples. Por exemplo, a posição do "Timor Leste" pode facilmente mudar de 194 para um outro valor caso objetos sejam removidos ou inseridos em `flags` e, em seguida, o DataFrame seja reindexado.

Uma alternativa mais viável é realizar a alteração tendo por base um atributo que funcione como uma **chave de identificação** para os registros da base de dados. No caso de `flags`, a chave é o atributo `name`, pois é este o atributo que identifica unicamente cada país. No programa a seguir, mostramos como aplicar o método `loc()` para efetuarmos uma atualização com base no atributo `name`.

```
#P50: Atualização
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Cria um DataFrame com os dados do novo país
df_novo = pd.DataFrame(
        {
        'name': 'East Timor',
        'landmass': 5,
        'area': 15,
        'language': 10,
        }, index=[0])

#3-Insere o novo país em flags
flags = pd.concat([flags,df_novo], ignore_index=True, sort=False)

#4-Atualiza o valor do atributo "religion"
#  (não espeficado no passo 2)
flags.loc[flags['name']=='East Timor','religion']=0

#5-Imprime o DataFrame alterado
print(flags[['name',
            'landmass',
            'area',
```

```
            'language',
            'religion'
            ]])
```

Aqui está a saída. Veja que agora o valor de `religion` é `0.0`:

```
                name  landmass  area  language  religion
0        Afghanistan         5   648        10       2.0
1            Albania         3    29         6       6.0
2            Algeria         4  2388         8       2.0
...
192           Zambia         4   753        10       5.0
193         Zimbabwe         4   391        10       5.0
194       East Timor         5    15        10       0.0
```

A atualização através do método `loc()` foi feita através da seguinte sintaxe:

```
loc(teste, atributo)=valor
```

Primeiro especifica-se um teste lógico para identificar as linhas que serão atualizadas. Em nosso exemplo, o teste especificado foi `flags['name']=='East Timor'`, que retorna `True` para uma única linha. Após o teste lógico, deve-se indicar o atributo (ou lista de atributos) que serão modificados. Em nosso exemplo, apenas `religion`. Por fim, você deve especificar o valor a ser atribuído (ou lista de valores) após o sinal de `=`.

Exclusão

Exclusão consiste na operação de remover um ou mais objetos de um DataFrame. Se você conhece de antemão os índices dos objetos a serem excluídos, então a operação pode ser feita de maneira trivial, com o uso do método `drop()`. Por exemplo, para remover os objetos de índice 1, 5 e 193 de `flags` e depois resetar os índices do

DataFrame, basta utilizar o método `drop()` seguido do `reset_index()`, da forma mostrada a seguir:

```
flags = flags.drop([1,5,193])
flags = flags.reset_index(drop=True)
```

Entretanto, no caso mais comum — onde objetos são excluídos com base no valor de um ou mais atributos — utilizamos o método `loc()`. Porém, é preciso muita atenção, pois devemos criar um teste lógico para determinar os objetos que deverão **permanecer no DataFrame** e não os que vão sair. Observe o exemplo a seguir, que remove todos os países que não possuem as cores verde, amarelo, azul e branco em suas bandeiras (só os que possuem todas elas é que **não** serão excluídos).

```
#P51: Exclusão
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-Mantém apenas os países com verde, amarelo,
#  azul e branco na bandeira
flags = flags.loc[(flags['green']==1) &
                  (flags['gold']==1) &
                  (flags['blue']==1) &
                  (flags['white']==1)]

#3-Imprime o DataFrame alterado
print(flags[['name',
            'green',
            'gold',
            'blue',
            'white']])
```

Saída:

```
                       name  green  gold  blue  white
17                   Belize      1     1     1      1
19                  Bermuda      1     1     1      1
```

```
23                          Brazil      1     1     1      1
24          British-Virgin-Isles      1     1     1      1
26                        Bulgaria      1     1     1      1
33                  Cayman-Islands      1     1     1      1
34    Central-African-Republic      1     1     1      1
48                        Dominica      1     1     1      1
56              Falklands-Malvinas      1     1     1      1
57                            Fiji      1     1     1      1
71                            Guam      1     1     1      1
78                       Hong-Kong      1     1     1      1
116                     Montserrat      1     1     1      1
135                        Parguay      1     1     1      1
139                       Portugal      1     1     1      1
142                        Romania      1     1     1      1
151                Soloman-Islands      1     1     1      1
158                      St-Helena      1     1     1      1
161                     St-Vincent      1     1     1      1
176            Turks-Cocos-Islands      1     1     1      1
182                US-Virgin-Isles      1     1     1      1
187                      Venezuela      1     1     1      1
```

Neste exemplo, utilizamos o método `loc()` para remover de `flags` todos os países cujo resultado do teste lógico seguinte resulta em `False`:

```
(flags['green']==1) &
(flags['gold']==1) &
(flags['blue']==1) &
(flags['white']==1)
```

Como exercício, compare este programa com o programa apresentado no final do capítulo 3, onde os mesmos países foram identificados com o uso da técnica de computação vetorizada.

6.3 Funções aplicadas sobre colunas

Em processos de transformação de dados, é muito comum que haja a necessidade de realizar o ajuste ou formatação dos dados de uma coluna do DataFrame através da aplicação de uma **função** Python sobre a coluna (técnica conhecida como *apply*). Esta seção apresenta as diversas funcionalidades oferecidas pela pandas para essa finalidade.

Substituindo valores

O método `replace()` pode ser utilizado em situações onde se deseja formatar uma coluna através da substituição de certos valores por um outro conjunto de valores. Por exemplo, no programa a seguir, mostramos como empregar esta técnica para modificar a coluna `green`, trocando o valor 0 por "Não" e o valor 1 por "Sim".

```
#P52: Método replace()
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-replace
flags['green'] = flags['green'].replace([0,1],['Não','Sim'])

#3-Imprime o DataFrame alterado
print(flags[['name','green']])
```

Saída:

```
                     name green
0             Afghanistan   Sim
1                 Albania   Não
2                 Algeria   Sim
...
191                 Zaire   Sim
192                Zambia   Sim
193              Zimbabwe   Sim
```

Na sintaxe do `replace()` primeiro especifica-se a lista de valores que serão substituídos (nesse exemplo, `[0,1]`) e depois a lista com os novos valores (`['Não','Sim']`). Note ainda que a substituição é feita a partir de uma atribuição, isto é a coluna `green` de `flags` , no lado esquerdo da atribuição, recebe o resultado do `replace()` especificado no lado direito:

```
flags['green'] = flags['green'].replace([0,1],['Não','Sim'])
```

Veja que, mais do que ter trocado os valores, o `replace()` foi capaz de mudar o tipo do atributo `green` de numérico discreto para categórico.

Limpeza de dados

O método `replace()` também é muito utilizado para realizar a limpeza de bases de dados. Por exemplo, considere o caso — muito comum! — de uma base de dados CSV que adota um valor **sentinela** (código especial) para representar dados ausentes, por exemplo, 999999 (outros sentinelas muito usados são -1, 999 e -999).

Suponha que esta base de dados tenha sido importada para um DataFrame `df` e que `x` é o nome da coluna que utiliza o valor sentinela. Para substituir todos os valores 999999 por `NaN` em `x` basta utilizar o seguinte comando:

```
df['x'] = df['x'].replace(999999,np.nan)
```

Recorde que `NaN` é o valor adotado pelas bibliotecas pandas e NumPy para representar valores ausentes; no entanto, a constante `np.nan` é, de fato, proveniente da biblioteca NumPy. Desta forma, para que o comando anterior funcione, é preciso realizar a importação da NumPy e da pandas, no início do código do programa, como mostrado a seguir:

```
import numpy as np
import pandas as pd
```

O MÉTODO REPLACE()

O método `replace()` é extremamente rico e possui muitas variações, suportando inclusive o uso de expressões regulares. Consulte a documentação do método para maiores detalhes: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html.

Formatando valores

Considere agora que você não deseja substituir valores de uma coluna, mas sim formatá-los. Por exemplo: converter os nomes de todos os países para caixa alta. Situações deste tipo podem ser resolvidas de maneira extremamente simples com o uso do método `apply()`, que permite que qualquer função do Python seja aplicada de uma só vez sobre todos os valores de uma coluna.

```
#P53: Método apply()
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-apply
flags['name'] = flags['name'].apply(str.upper)

#3-Imprime o DataFrame alterado
print(flags)
```

Nesse exemplo, o método `apply()` foi utilizado para aplicar a função `upper()` da classe `str` (string) sobre a coluna `name`. Como resultado, os nomes dos países foram passados para caixa alta.

```
                 name  landmass  zone    ...    botright
0         AFGHANISTAN         5     1    ...    green
1             ALBANIA         3     1    ...    red
2             ALGERIA         4     1    ...    white
```

```
...
191                ZAIRE         4      2     ...      green
192               ZAMBIA         4      2     ...      brown
193             ZIMBABWE         4      2     ...      green
```

O mais interessante do método `apply()` é que ela aceita o uso de qualquer tipo de função, seja uma função do Python padrão (como `upper()`) seja uma função **definida pelo próprio programador**, como será visto já a partir da próxima seção.

6.4 Discretização

Discretização é a tarefa de transformação de dados que consiste em converter os valores de um atributo contínuo em um conjunto reduzido de valores discretos ou categóricos. Esta seção cobre a abordagem básica para discretização, baseada na criação de uma função customizada.

Criando funções

Uma função é um bloco de código nomeado, que pode receber um ou mais valores como entrada (parâmetros) e que, após realizar algum tipo de processamento envolvendo os parâmetros, retornará um resultado como saída. A palavra reservada `def` é utilizada para a definição de funções no Python. Para criar uma função, você deve iniciar escrevendo `def`, depois indicar o nome da função, os seus parâmetros (caso existam) e, por fim, o bloco de código correspondente ao corpo da função.

Será apresentado um exemplo onde se realiza a definição de uma função que recebe como entrada a área de um país (fornecida em milhares de quilômetros quadrados) e que retorna como saída um

código que classifica a área de acordo com a sua grandeza, considerando a tabela a seguir:

Código	Tamanho da área
A	mais de 1 000 000 km^2
B	de 500 000 km2 a 1 000 000 km^2
C	de 100 000 km2 a 500 000 km^2
D	de 50 000 km2 a 99 000 km^2
E	de 10 000 km2 a 50 000 km^2
F	menos de 10 000 km^2

Observe que se trata de uma função de discretização, pois ela recebe como entrada um valor contínuo (valor da área) e retorna como saída um valor categórico, pertencente ao domínio {"A","B","C","D","F","G","H"}.

```
#p54: Criação e utilização de Função
def classe_area(area):
    if (area < 10) :
        return 'F'
    elif (area >= 10 and area < 50) :
        return 'E'
    elif (area >= 50 and area < 100) :
        return 'D'
    elif (area >= 100 and area < 500) :
        return 'C'
    elif (area >= 500 and area < 1000) :
        return 'B'
    else :
        return 'A'

print('Classe de extensão territorial: ')
print('Brasil     :', classe_area(8515))
print('Moçambique :', classe_area(801))
```

```
print('Portugal   :', classe_area(92))
print('Timor Leste:', classe_area(14))
```

Saída:

```
Classe de extensão territorial:
Brasil      : A
Moçambique : B
Portugal    : D
Timor Leste: E
```

Como você deve ter observado, o programa começa com a definição função `classe_area()` (linhas 2-14). A primeira linha da definição da função, que contém a palavra `def`, é chamada de **cabeçalho** da função, enquanto as linhas seguintes formam o seu **corpo**. O cabeçalho deve terminar com `:` e o corpo deve estar indentado. Uma vez a função tendo sido definida, ela pode ser utilizada em qualquer parte do programa. E é exatamente isto que fizemos! No exemplo, a função `classe_area()` é chamada quatro vezes, para classificar os países Brasil, Moçambique, Portugal e Timor Leste.

Aplicando a função criada

No exemplo a seguir, mostramos como fazer um uso ainda melhor da nossa função personalizada `classe_area()`. Ela será aplicada sobre a coluna `area` de `flags`, possibilitando com que a classe de todos os países dessa base seja determinada.

```
#p55: Aplicando a função classe_area()
import pandas as pd

#1-Define a função
def classe_area(area):
    if (area < 10) :
        return 'F'
    elif (area >= 10 and area < 50) :
        return 'E'
    elif (area >= 50 and area < 100) :
        return 'D'
```

```
    elif (area >= 100 and area < 500) :
        return 'C'
    elif (area >= 500 and area < 1000) :
        return 'B'
    else :
        return 'A'

#2-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#3-aplica a função com apply(), criando novo atributo
flags['classe_area'] = flags['area'].apply(classe_area)

#4-Imprime o DataFrame alterado
print(flags[['name',
             'area',
             'classe_area'
             ]])
```

Saída:

```
                    name   area classe_area
0            Afghanistan    648           B
1                Albania     29           E
2                Algeria   2388           A
...
191                Zaire    905           B
192               Zambia    753           B
193             Zimbabwe    391           C
```

Neste exemplo, utilizamos o método `apply()` para aplicar a nossa função personalizada `classe_area()` :

```
flags['classe_area'] = flags['area'].apply(classe_area)
```

Adicionalmente, veja que o programa também exemplificou um processo **construção de atributo**. Isso porque **criamos um novo atributo** no DataFrame `flags` , denominado `classe_area` , cujo valor foi definido a partir do resultado do `apply()` .

DISCRETIZAÇÃO: TÉCNICAS NÃO-SUPERVISIONADAS

Também é possível realizar a discretização através dos chamados métodos não-supervisionados. Neste tipo de método, a operação de discretização é feita através da aplicação direta de uma fórmula matemática capaz de determinar automaticamente a configuração de cada faixa de valores.

Para maiores informações consulte o link: https://www.saedsayad.com/unsupervised_binning.htm.

6.5 Normalização

Normalizando atributos numéricos

Em qualquer base dados, é comum que diferentes variáveis (atributos) estejam representadas em diferentes unidades de medidas. Veja a seguir as unidades de medidas de três diferentes atributos numéricos presentes na base de dados `flags` :

- `population` : milhões de pessoas
- `area` : milhares de quilômetros quadrados
- `colours` : unidades

No entanto, diversos algoritmos de ciência de dados (especialmente algoritmos para cálculo de similaridade ou distância entre objetos) requerem que todos os atributos numéricos estejam na mesma **escala** para que possam funcionar corretamente, ou seja, que estes atributos estejam **normalizados** dentro de uma faixa de valores comum como `[0.0, 1.0]` ou `[-1, 1]` . Isto deve ser feito para

assegurar que os números grandes não "dominem" os números pequenos durante a análise.

Considere, por exemplo, um DataFrame contendo dois atributos `X` e `Y`, onde a unidade de medida de `X` é milhas náuticas e a de `Y` centímetros. Neste caso, uma diferença de 1 unidade em `X` é equivalente a 185.200 unidades em `Y`! Ao aplicar uma fórmula para medir a distância entre os objetos do DataFrame, o valor de `Y` acabaria sendo o único a contribuir no valor computado.

Existem diversas fórmulas que podem ser utilizadas para mapear todos os valores de um atributo para a faixa `[0.0, 1.0]`. Uma das mais simples é apresentada a seguir. Ela pode ser utilizada sobre qualquer atributo que armazene valores contínuos positivos.

```
valor_normalizado = (x-min(X))/(max(X)-min(X))
```

Nesta fórmula, considere que `x` e o valor real do atributo numérico `X` em uma dada linha do DataFrame. Por sua vez, `min(X)` e `max(X)` consistem, respectivamente, no menor e no maior valor de `X` em toda a base de dados. A fórmula apresentada retornará o valor normalizado de `x`, isto é o valor convertido para algum número localizado entre 0 e 1. No exemplo a seguir, mostra-se uma aplicação prática da fórmula: ela é empregada para computar o valor normalizado do atributo `area` de todos os países de `flags`.

```
#P56: Normalização
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-normaliza a área
area_max = max(flags['area'])
area_min = min(flags['area'])
flags['area_norm'] = (flags['area'] - area_min) / (area_max - area_min)

#3-Imprime o DataFrame alterado
print(flags[['name',
```

```
             'area',
             'area_norm']])
```

Saída:

```
                    name   area  area_norm
0            Afghanistan    648   0.028926
1                Albania     29   0.001295
2                Algeria   2388   0.106598
...
191                Zaire    905   0.040398
192               Zambia    753   0.033613
193             Zimbabwe    391   0.017454
```

Veja que a pandas possibilita com que a normalização seja realizada de uma forma extremamente simples e elegante, através da computação vetorizada:

```
flags['area_norm'] = (flags['area'] - area_min) / (area_max - area_min)
```

A fórmula definida neste comando de atribuição é aplicada sobre toda a coluna `area`, sem a necessidade da implementação de um laço.

Normalizando atributos categóricos

Em uma base de dados, os atributos categóricos costumam apresentar grande variação no número de categorias. Veja a seguir o número de categorias de três diferentes atributos da base de dados `flags`:

- `religion`: 8 categorias
- `language`: 10 categorias
- `red`: 2 categorias (variável binária)

Esta variabilidade também pode atrapalhar alguns algoritmos de ciência de dados e, por isso, um tipo de transformação muitas vezes necessário consiste na conversão de todos os atributos categóricos com mais de 2 categorias (como é o caso de `religion` e `language`)

em `k` atributos binários, onde `k` é o número original de categorias do atributo. Parece complicado, mas o exemplo a seguir deixará tudo mais claro.

Considere o atributo `language`, cujo domínio é: {"1=Inglês", "2=Espanhol", "3=Francês", "4=Alemão", "5=Eslavo", "6=Outro idioma Indo-Europeu", "7=Chinês", "8=Árabe", "9=Japonês/Turco/Finlandês/Húngaro", "10=Outro"}. Como esse atributo possui 10 categorias distintas, então ele será transformado em 10 atributos binários, conforme indicado a seguir:

- `lang_1`: indica se idioma predominante do país é Inglês (0=Não, 1=Sim)
- `lang_2`: Idem para Espanhol
- `lang_3`: Idem para Francês
- `lang_4`: Idem para Alemão
- `lang_5`: Idem para Eslavo
- `lang_6`: Idem para Outro idioma Indo-Europeu
- `lang_7`: Chinês
- `lang_8`: Árabe
- `lang_9`: Japonês/Turco/Finlandês/Húngaro
- `lang_10`: Idem para Outro

No programa a adiante, apresentamos a receita para realizar este tipo de transformação sobre atributos categóricos na pandas. Ela consiste no emprego de dois métodos: `get_dummies()` e `join()`.

```
#P57: Normalização de atributo categórico
import pandas as pd

#1-Importa as bases de dados
flags = pd.read_csv('c:/bases/flags.csv')

#2-realiza a transformação com get_dummies() e join()
dummies = pd.get_dummies(flags['language'], prefix='lg')
flags = flags.join(dummies)
```

```
print(flags[["language",
             "lg_1",
             "lg_2",
             "lg_3",
             "lg_4",
             "lg_5",
             "lg_6",
             "lg_7",
             "lg_8",
             "lg_9",
             "lg_10"
        ]])
```

Veja a saída, onde mostramos o valor original do atributo `language` e os valores dos 10 atributos derivados:

```
                name  language  lg_1  ...   lg_8  lg_9  lg_10
0        Afghanistan        10     0  ...      0     0      1
1            Albania         6     0  ...      0     0      0
2            Algeria         8     0  ...      1     0      0
...
191            Zaire        10     0  ...      0     0      1
192           Zambia        10     0  ...      0     0      1
193         Zimbabwe        10     0  ...      0     0      1
```

Neste programa, o método `get_dummies()` foi utilizado com dois parâmetros:

- `flags['language']` : coluna que será transformada;
- `prefix='lg'` : prefixo dos atributos que serão criados.

Como resultado, foi gerado um novo DataFrame chamado `dummies` , contendo 192 linhas (uma para cada país) e 10 atributos binários, `lg_1` a `lg_10` , um para cada categoria de `language` :

```
     lg_1  lg_2  lg_3  lg_4  lg_5  lg_6  lg_7  lg_8  lg_9  lg_10
0       0     0     0     0     0     0     0     0     0      1
1       0     0     0     0     0     1     0     0     0      0
2       0     0     0     0     0     0     0     1     0      0
...
```

```
191      0      0      0      0      0      0      0      0      0       1
192      0      0      0      0      0      0      0      0      0       1
193      0      0      0      0      0      0      0      0      0       1
```

Em seguida, o método `join()` foi empregado para fazer a junção de `dummies` com `flags`, isto é, para incorporar as novas variáveis binárias ao DataFrame `flags`. Conforme comentado no capítulo anterior, o método `join()` serve para realizar a junção de DataFrames através dos índices dos mesmos. Melhor explicando: quando usamos o método `join()`, os índices são utilizados como chave de ligação para combinar o par de DataFrames.

6.6 Projeto prático — transformando o dataset flags

Aos examinarmos os atributos do dataset `flags`, torna-se possível categorizá-los em quatro grupos distintos, que são relacionados a seguir:

- Grupo 1 — categóricos binários: `red`, `green`, `blue`, `gold`, `white`, `black`, `orange`, `crescent`, `triangle`, `icon`, `animate`, `text`;
- Grupo 2 — categóricos não binários: `name`, `landmass`, `zone`, `language`, `religion`, `mainhue`, `topleft`, `botright`;
- Grupo 3 — numéricos contínuos: `area`, `population`;
- Grupo 4 — numéricos discretos: `bars`, `stripes`, `colours`, `circles`, `crosses`, `saltires`, `quarters`, `sunstars`;

Nesta seção, apresentaremos um programa que será responsável por transformar alguns destes atributos com o objetivo de **uniformizar** a base `flags`. Esse processo de uniformização será empregado para em deixar a base de dados em uma condição ideal para ser utilizada por um algoritmo de classificação, o que ocorrerá no próximo capítulo deste livro. Mais especificamente, o objetivo é:

1. Normalizar todos os atributos numéricos para uma faixa comum: `[0.0, 1.0]`;
2. Converter todos os atributos categóricos com `k > 2` categorias para `k` atributos binários;
3. Remover os atributos que não tomarão parte no processo de classificação (por exemplo: remover o atributo original `language` após o mesmo ter sido transformado em 10 atributos binários).

De acordo com essa explicação, está claro que não será necessário efetuar transformações sobre os atributos do Grupo 1, mas precisaremos manipular todos os atributos dos Grupos 2, 3 e 4. Parece complicado, mas a pandas facilita muito as coisas para os programadores! Veja que com o pequeno programa listado a seguir, torna-se possível dar conta desta tarefa.

```
#p58 - transformação da base flags
import pandas as pd

#-------------------------------------------------------
#(1)-Importa a base de dados
#-------------------------------------------------------
flags = pd.read_csv('c:/bases/flags.csv')

#-------------------------------------------------------
#(2)-Conversão dos atributos do Grupo 2
#    De: Categóricos não binários
#    Para: Categóricos binários
#---------------------------------------------------------
for c in flags.columns:
    if c in ['landmass', 'zone',
             'language', 'religion', 'mainhue',
             'topleft', 'botright']:
        dummies = pd.get_dummies(flags[c], prefix=c)
        flags = flags.join(dummies)

#-------------------------------------------------------
#(3)-Normalização dos atributos dos Grupo 3 e 4
#    De: Numéricos contínuos e discretos
```

```python
#    Para: Numéricos com valores na faixa entre 0 e 1
#--------------------------------------------------------
for c in flags.columns:
    if c in ['area', 'population',
             'bars', 'stripes', 'colours',
             'circles', 'crosses', 'saltires',
             'quarters', 'sunstars']:

        c_max = max(flags[c])
        c_min = min(flags[c])
        flags[c] = (flags[c] - c_min) / (c_max - c_min)

#------------------------------------------------------
#4-Exclusão dos atributos indesejados
#-----------------------------------------------------
flags = flags.drop(columns=['name', 'landmass',
                            'zone', 'language',
                            'religion', 'mainhue',
                            'topleft', 'botright',
                            ])

#------------------------------------------------------
#5-imprime a configuração final de flags
#------------------------------------------------------
#imprime as primeiras linhas
print('head():'); print(flags.head())
print('-----------------------------------------------')

#imprime as últimas linhas
print('tail():'); print(flags.tail())
print('-----------------------------------------------')

#------------------------------------------------------
#6-Salva o dataset alterado para um arquivo
#------------------------------------------------------
flags.to_csv("C:/bases/flags_transf.csv", sep=",", index=False)
```

Como saída, gera-se o arquivo CSV `flags_transf.csv` , com 194 linhas e 73 colunas. Uma parte das linhas e colunas deste arquivo é

apresentada na figura a seguir.

```
flags_transf.csv
1  area,population,bars,stripes,colours,red,green,blue,gold,whi
2  0.028925988751004373,0.015873015873015872,0.0,0.214285714285
3  0.0012945272743505045,0.002976190476190476,0.0,0.0,0.2857142
4  0.10659762521203464,0.01984126984126984,0.4,0.0,0.2857142857
5  0.0,0.0,0.0,0.0,0.5714285714285714,1,0,1,1,1,0,1,0.0,0.0,0.0
6  0.0,0.0,0.6,0.0,0.2857142857142857,1,0,1,1,0,0,0,0.0,0.0,0.0
7  0.05566467279707169,0.006944444444444444,0.0,0.1428571428571
8  0.0,0.0,0.0,0.07142857142857142,0.2857142857142857,0,0,1,0,1
9  0.0,0.0,0.0,0.07142857142857142,0.5714285714285714,1,0,1,1,1
10 0.12396214623694313,0.027777777777777776,0.0,0.2142857142857
11 0.12396214623694313,0.027777777777777776,0.0,0.2142857142857
12 0.3432729220605303,0.01488095238095238,0.0,0.0,0.28571428571
13 0.00374966520846353,0.007936507936507936,0.0,0.2142857142857
14 0.0008481385590572271,0.0,0.0,0.21428571428571427,0.28571428
15 4.463887152932774e-05,0.0,0.0,0.0,0.14285714285714285,1,0,0,
16 0.0063833586286938665,0.08928571428571429,0.0,0.0,0.14285714
17 0.0,0.0,0.6,0.0,0.2857142857142857,0,0,1,1,0,1,0,0.0,0.0,0.0
18 0.00138380501740916,0.00992063492063492,0.6,0.0,0.2857142857
19 0.001026694045174538,0.0,0.0,0.14285714285714285,1.0,1,1,1,1
20 0.005044192482814035,0.002976190476190476,0.0,0.0,0.14285714
21 0.0,0.0,0.0,0.0,0.7142857142857143,1,1,1,1,1,1,0,0.25,0.5,1.
22 0.0020980269618784035,0.000992063492063492,0.0,0.0,0.4285714
23 0.04905811981073119,0.005952380952380952,0.0,0.2142857142857
24 0.026783322917596643,0.000992063492063492,0.0,0.357142857142
25 0.3799660744576377,0.11805555555555555,0.0,0.0,0.42857142857
26 0.0,0.0,0.0,0.0,0.7142857142857143,1,1,1,1,1,0,1,0.0,0.5,1.0
27 0.00026783322917596643,0.0,0.0,0.0,0.42857142857142855,1,0,0
28 0.004954914739755379,0.008928571428571428,0.0,0.214285714285
29 0.0122310507990358,0.006944444444444444,0.0,0.14285714285714
30 0.030265154896884208,0.034722222222222224,0.0,0.0,0.28571428
31 0.0012498884028211766,0.003968253968253968,0.0,0.0,0.2857142
```

Figura 6.1: Base flags após a sua transformação.

O programa é dividido em 6 partes e utiliza algumas das técnicas que foram introduzidas ao longo deste capítulo, sem qualquer tipo de modificação. Aplicamos diretamente e tudo funcionou! A seguir, a explicação relativa a cada parte do programa.

- A primeira parte consiste simplesmente na importação do dataset `flags`.

- Na segunda parte, fazemos a conversão de todos os atributos do Grupo 2. Isto é: cada atributo categórico com `k>2` categorias é transformado em `k` atributos binários. Por exemplo, `language`, é transformado em `language_1` ... `language_10`. Para tal, bastou empregar o método `get_dummies()` em parceria com o método `join()`.
- De maneira análoga, na parte 3, todos os atributos numéricos, sejam contínuos ou discretos, são normalizados para valores entre 0 e 1. Utilizamos a mesma fórmula que foi apresentada no tópico sobre normalização.
- A parte 4 é bastante interessante. Ela tem por objetivo eliminar do DataFrame todos os atributos indesejados. Por exemplo, depois de gerar os atributos `language1` ... `language10`, não faz mais nenhum sentido continuar com o atributo original `language`. Então, utilizamos o método `drop()` passando uma lista contendo `language` e todos os demais atributos que se tornaram inúteis após a transformação da base. É importante comentar que aproveitamos para também eliminar o atributo `name`, que é a chave identificadora dos registros de `flags` (conforme comentado no capítulo 1 deste livro, os atributos chave não são úteis para a construção de modelos de Machine Learning).
- Na parte 5, imprime-se a nova configuração de `flags`.
- Por fim, na parte 6, o arquivo transformado é salvo para um CSV chamado `flags_transf.csv`.

E assim finalizamos nosso capítulo. Mais do que isso, concluímos a apresentação das funcionalidades da biblioteca pandas. Ao longo do livro, foram cobertas todas as atividades importantes de *Data Wrangling*: importação de dados para Series e DataFrames; estudo das características dos atributos através da produção de índices estatísticos e gráficos; combinação de bases de dados; Por fim, neste capítulo, limpeza e transformação de dados.

Para encerrar o livro da melhor forma possível, o próximo capítulo conclui o nosso projeto prático com um pouquinho de Machine Learning. Nós veremos como é possível executar um processo de

classificação em Python. Para isso, precisaremos de apenas três ferramentas em mãos: a nossa querida pandas, o pacote de Machine Learning `scikit-learn` e a base de dados `flags_transf.csv` (versão transformada da base `flags`). Está pronto para aprender sobre classificação com Python? Se a resposta for positiva, basta avançar para a próxima página...

Capítulo 7
Um pouco de Machine Learning

Machine Learning é o campo da ciência que se preocupa em investigar como computadores podem aprender a executar tarefas baseando-se no uso de dados. Os processos de Machine Learning são realizados por programas de computador que aprendem a reconhecer padrões complexos e conseguem tomar decisões inteligentes através de análise de dados.

Classificação é provavelmente a tarefa de Machine Learning mais utilizada em projetos de ciência de dados. Nesta tarefa, o objetivo é realizar a associação automática de objetos a uma ou mais classes, pertencentes a um conjunto predefinido de classes. Existem muitas aplicações modernas e importantes para a classificação, variando desde a análise de sentimentos (classificar a opinião expressa em um texto como "positiva", "negativa" ou "neutra") até a detecção de fraudes (classificar uma transação financeira como "legal" ou "suspeita").

Neste capítulo final de nosso livro, você aprenderá a criar um classificador na linguagem Python. Mais especificamente, um classificador capaz de inferir as cores presentes na bandeira de um país em função de suas características. Para tal, utilizaremos a pandas trabalhando em conjunto com a **scikit-learn**, a principal biblioteca para Machine Learning do ambiente Python. A base de dados utilizada no projeto será a `flags_transf.csv`, a versão transformada da base de dados `flags`.

O capítulo enfatiza a parte prática, mas também cobre um pouco de teoria sobre classificação. Ele está dividido da seguinte forma:

- O que é classificação?
- Como criar um classificador?
- Como avaliar um classificador?

- Biblioteca scikit-learn
- Criando um classificador em Python

7.1 O que é classificação?

A tarefa de classificação possui o seguinte objetivo: a partir de um banco de dados contendo objetos pré-classificados (objetos cuja classe é conhecida), construir um modelo que seja capaz de classificar automaticamente novos objetos (objetos cuja classe é desconhecida) em função de suas características. O modelo criado é chamado de **modelo de classificação** ou **classificador**.

Existem inúmeras aplicações práticas para a classificação, tanto no âmbito das pesquisas científicas quanto dentro das empresas. A seguir são apresentados alguns exemplos. Alguns deles são bem conhecidos — você perceberá que já teve contato com classificadores várias vezes em sua vida, mesmo sem saber disso!

- Os programas de filtragem de spam empregam algoritmos de classificação. A partir do assunto e do texto (corpo) da mensagem, o classificador de spam decide se a mensagem é "normal" ou um "spam";
- Diversos sites e portais de notícias utilizam a classificação para realizar a atribuição automática de tópicos (*Topic Tagging*). Nesta aplicação, textos de reportagens ou artigos são automaticamente classificados em um número fixo de categorias (por exemplo: "meio ambiente", "política", "ciência");
- Uma utilização mais científica pode ser encontrada na área de Bioinformática. Dentre as diversas aplicações para classificadores nesta área, temos a identificação automática da classe de proteínas, o que possibilita descobrir quais são suas funções;

Em alguns tipos de aplicação podemos associar apenas um rótulo de classe ao objeto alvo da classificação. Por exemplo, na filtragem de spam, um e-mail recebido deve ser classificado como "normal" ou "spam", mas, evidentemente nunca pode ser classificado com ambos os rótulos. Por isso, esse tipo de aplicação é chamado de classificação monorrótulo (*single-label classification*).

Em outros tipos, como na atribuição de tópicos, o objeto a ser classificado pode receber uma ou mais classes. Neste caso, trata-se de uma aplicação de classificação multirrótulo (*multi-label classification*). O classificador que construiremos neste capítulo será do tipo multirrótulo, pois trata-se de um modelo para classificar as cores da bandeira de um país (as bandeiras possuem diversas cores).

7.2 Como criar um classificador?

Para criar um classificador multirrótulo, você precisa ter apenas dois artefatos em mãos: uma boa **base de dados de treinamento** e um bom **algoritmo de classificação**. A seguir, detalhamos as características de cada um destes artefatos.

Bases de dados de treinamento

A base de dados de treinamento é simplesmente uma base composta por objetos já classificados, de acordo com as classes predefinidas. Por exemplo, para gerar um classificador de spam, você precisará de uma base de dados de treinamento contendo mensagens que são normais e outras que são spam.

Tecnicamente falando, uma base de dados de treinamento deve ser formada por uma coleção de registros do tipo `(X, Y)`, onde:

- `X` representa um conjunto de atributos denominados atributos preditivos. Estes atributos representam as características do objeto.
- `Y` representa o conjunto com os atributos classe (também chamados de **rótulos de classe**).

Para que o conceito fique claro, vamos apresentar dois exemplos, um relacionado à classificação monorrótulo e outro, à classificação multirrótulo.

- Classificação monorrótulo — base de treinamento para gerar um classificador de spam: neste caso, `X` conteria o texto de cada e-mail, enquanto `Y` consiste em um único atributo categórico — que poderia, por exemplo, se chamar `tipo_email` — que pode registrar apenas duas categorias: "normal" ou "spam".
- Classificação multirrótulo — base de treinamento para gerar um classificador de cores de bandeiras: no caso da base `flags`, o conjunto `X` é composto pelos atributos que registram informações como extensão territorial, linguagem, continente etc. (na verdade, todos os atributos, exceto os sete que registram as cores da bandeira). O conjunto `Y` é formado por sete atributos classe: `red`, `green`, `blue`, `gold`, `white`, `black` e `orange`.

Algoritmos de classificação

O objetivo de um algoritmo de classificação é realizar o aprendizado de um modelo capaz de processar a base de dados de treinamento, mapeando o conjunto de atributos preditivos `X` para um ou mais rótulos de classe `Y`.

Como já foi mencionado, este modelo é chamado de classificador. Após ser criado, o classificador será colocado em produção com o objetivo de classificar novos objetos, isto é, objetos cuja classe é desconhecida. Por exemplo, novos e-mails que chegam a uma caixa postal serão classificados como "normal" ou "spam" pelo classificador de spam.

Ao longo das últimas décadas, diversos algoritmos de classificação foram projetados por pesquisadores e empresas de todas as partes do mundo. Dentre os mais popularmente utilizados, podemos citar k-vizinhos mais próximos, árvores de decisão, redes neurais e Naïve Bayes, apenas para mencionar alguns.

Neste capítulo, mostraremos como criar um classificador em Python utilizando o algoritmo dos k-vizinhos mais próximos (*k-Nearest Neighbours*, abreviado como k-NN). Este algoritmo é muito intuitivo, executando a tarefa de classificação a partir de dois passos bem simples:

1. **Treinamento do classificador**. Consiste apenas em importar a base de dados de treinamento para a memória.
2. **Classificação de um novo objeto**. Suponha que temos um novo objeto para ser classificado, chamado `novoObj`. O algoritmo então vai procurar os `k` objetos mais similares a `novoObj` na base de dados de treinamento, que são os chamados "vizinhos mais próximos". As classes mais frequentes entre os `k` vizinhos mais próximos serão atribuídas para `novoObj`.

Para que o conceito fique claro, a figura seguinte apresenta um exemplo que ilustra como seria a classificação de cores de bandeiras através do k-NN. Para simplificar, considere que a base de treinamento utilizada contém apenas 6 objetos (isto é, apenas os dados de seis países) e que o valor de `k` é 3 (é importante mencionar que o valor de `k` é um parâmetro de entrada que pode ser especificado pelo usuário da ferramenta de ciência de dados).

Suponha que desejamos classificar as cores da bandeira de um novo objeto chamado `novoPaís`. Nesse caso, como usuários do sistema de classificação, nós conheceríamos os valores do conjunto `X` de atributos preditivos de `novoPaís` (extensão, população, idioma etc.). No entanto, as cores de sua bandeira (conjunto `Y`) nos seriam desconhecidas. Utilizaríamos o classificador exatamente para descobri-las, assim como o classificador de spam é utilizado para descobrir se um novo e-mail é "normal" ou "spam".

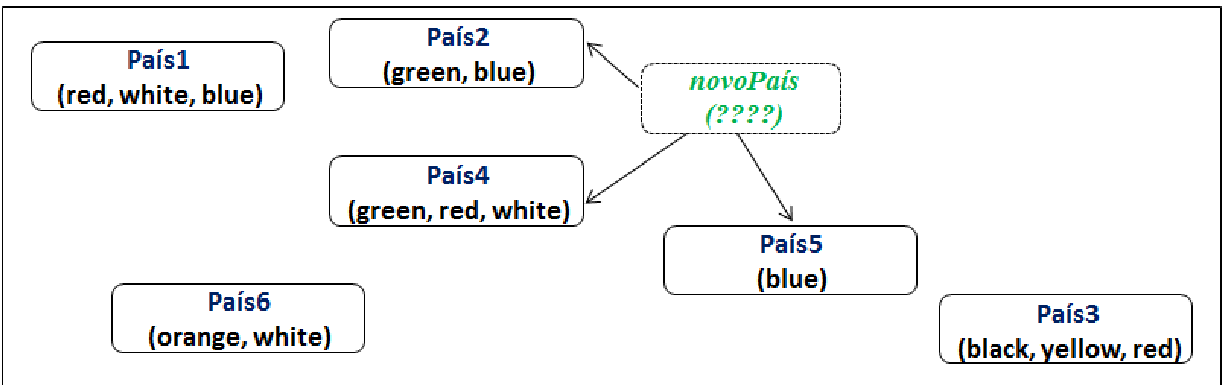

Figura 7.1: Algoritmo de classificação k-NN.

No exemplo da figura, veja que os 3 países mais similares a `novoPaís` são `País2`, `País4` e `País5`. Estes são os 3 vizinhos mais próximos de `novoPaís`. Observe ainda que as classes de `País2` são `green` e `blue`; as de `País4` são `green`, `red` e `white`; e as de `País5`, apenas `blue`. Sendo assim, as classes atribuídas para `novoPaís` seriam `green` e `blue`, pois estas classes apareceram em 2 dentre os 3 vizinhos mais próximos de `novoPaís`.

Como você dever ter percebido, o algoritmo k-NN é muito simples. Entretanto, computar a similaridade ou distância entre objetos é algo bem mais difícil e foge do escopo deste livro. Uma apresentação detalhada de alguns métodos para este fim pode ser obtida em livros acadêmicos sobre Machine Learning e Data Mining (como por exemplo http://www.mmds.org).

A biblioteca scikit-learn disponibiliza cerca de vinte diferentes medidas de distância para os seus usuários (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html). Apenas a título de curiosidade, o exemplo apresentado na seção final deste capítulo fará uso da medida conhecida como **distância euclidiana** para determinar a similaridade/dissimilaridade entre países. Essa é a medida default da scikit-learn.

7.3 Como avaliar um classificador?

Antes de um(a) cientista de dados colocar um modelo de classificação em produção, deverá estar muito seguro(a) de que este apresentará um bom **desempenho preditivo** para classificar novos objetos. Isso significa que o(a) cientista de dados precisará estar confiante de que o classificador acertará a maioria significativa das predições das classes de novos objetos. Afinal de contas, ninguém vai querer usar um detector de spam que classifique muitos e-mails normais como spam e vice-versa. E muito menos, nenhuma companhia de cartão de crédito adotaria um sistema de detecção de fraudes que classificasse muitas transações fraudulentas como normais (imagine o prejuízo!)

Falando de uma forma mais técnica, em qualquer projeto real de classificação existe a necessidade de **estimar a acurácia** do classificador antes que ele seja colocado em produção. Diferentes métodos foram propostos para esta finalidade, conforme indicado na própria documentação da scikit-learn (https://scikit-learn.org/stable/model_selection.html). Para o nosso classificador de cores de bandeiras, utilizaremos um método chamado LOO (*leave-one-out*, que pode ser traduzido como "deixar um de fora"), pois este é considerado o método mais adequado para bases de dados que não contêm um número grande de objetos.

O método LOO executa N iterações, onde N é o número de objetos da base de treinamento. A cada iteração, um dos objetos é removido da base de dados, e o modelo de classificação é construído com os N-1 objetos restantes. Finalizada a sua construção, o modelo é então utilizado para classificar o objeto que foi removido. A classe predita pelo modelo poderá ser então comparada com a classe real do objeto testado. Evidentemente, poderá ocorrer um acerto ou um erro, isto é, o modelo poderá acertar ou errar a classe do objeto testado.

Cada erro ou acerto será devidamente registrado em uma estrutura denominada **matriz de confusão** (não se preocupe, pois essa estrutura será explicada em breve). Ao final do processo, o LOO terá construído e avaliado N versões de classificadores de uma forma justa, pois a cada iteração, o objeto testado não é utilizado na construção do modelo (ou seja, o objeto testado sempre será desconhecido do modelo, como se fosse um objeto novo).

Conforme mencionado no parágrafo anterior, uma matriz chamada de matriz de confusão deverá ser atualizada a cada uma das iterações do LOO. Trata-se de uma matriz quadrada, 2x2, que terá a responsabilidade de registrar todas as classificações corretas e erradas que ocorreram durante todo o processo de LOO. Um exemplo de matriz de confusão é mostrado na figura seguinte. Considere que ela refere-se aos resultados da avaliação do rótulo de classe `green` de nosso classificador de cores de bandeiras.

		Rótulos Preditos	
		green='Não'	green='Sim'
Rótulos Reais	green='Não'	VN	FP
	green='Sim'	FN	VP

Figura 7.2: Matriz de confusão.

Em uma matriz de confusão `mc`, cada célula `mc[i,j]` denota o número de objetos que o classificador associou à classe `i` e que, de fato, pertencem à classe `j`. Sendo assim:

- `mc[1,1]` armazena o total de objetos cujo rótulo era "Não" (neste exemplo, objetos que não possuem a cor `green` na bandeira) e que foram corretamente classificados como "Não". São por isso chamados de Verdadeiro Negativos (VN).
- `mc[1,2]` armazena o total de objetos testados cujo rótulo era "Não" e que foram incorretamente classificados como "Sim". Tratam-se dos Falso Positivos (FP).

- `mc[2,1]` armazena o total de objetos de teste cujo rótulo era "Sim" (neste exemplo, objetos que possuem a cor `green` na bandeira) e que foram incorretamente classificados como "Não". Estes são os Falso Negativos (FN).
- `mc[2,2]` armazena o total de objetos testados cujo rótulo era "Sim" e que foram corretamente classificados como "Sim", chamados de Verdadeiro Positivos (VP).

A cada iteração do LOO, uma célula da matriz será incrementada de acordo com a comparação do rótulo real do objeto testado com o rótulo predito pelo classificador. Tendo sido finalizado o processo de LOO, a matriz de confusão definitiva estará pronta. A partir dela, torna-se possível calcular a medida da **Acurácia**, que computa a proporção de objetos corretamente classificados durante o LOO. Sua fórmula é dada por:

```
Acurácia = (VP + VN) / (VP + FP + FN + VN)
```

Como você deve ter percebido, a diagonal principal da matriz de confusão (células VP e VN) armazena as classificações corretas. Em geral, um classificador considerado efetivo precisa ter não apenas uma acurácia alta, mas um desempenho equilibrado para os tipos de classificação que realiza. Isso significa que não adianta ele ser muito bom para prever o valor de classe "Sim", mas ter o desempenho ruim para o valor de classe "Não" e vice-versa.

É importante ainda observar que, no caso de um processo de classificação multirrótulo, uma estratégia comumente adotada consiste em gerar uma matriz de confusão para cada rótulo separadamente (por exemplo: uma para `green`, outra para `blue`, outra para `red` etc.), para que seja possível verificar o desempenho preditivo do classificador com respeito a cada um dos rótulos. É exatamente isso que faremos em nosso programa-exemplo.

7.4 Biblioteca scikit-learn

A biblioteca scikit-learn é voltada para o desenvolvimento de projetos de Machine Learning em Python. Ela contém uma série de módulos que oferecem algoritmos para a criação e avaliação de modelos de classificação, regressão e *clustering*, além de métodos para realizar a análise de importância de atributos.

Assim como ocorre com a pandas, a scikit-learn é instalada automaticamente por qualquer distribuição python voltada para ciência de dados. Porém, se você estiver utilizando um ambiente onde ela não esteja instalada, poderá utilizar o utilitário `pip` para obtê-la. Basta abrir uma janela de prompt de comandos e digitar:

```
pip install -U scikit-learn
```

Antes de utilizar a scikit-learn em qualquer programa Python, é sempre preciso importá-la através do comando `import`. Tipicamente, todo programa que utiliza a biblioteca faz a sua importação de forma semelhante à mostrada adiante:

```
from sklearn.modulo import algoritmo as apelido
```

Onde:

- `modulo`: consiste em algum dos módulos disponibilizados pela scikit-learn. Normalmente eles são organizados de acordo com o tipo de tarefa a ser executada (classificação, *clustering* etc.) e, às vezes, também considerando uma determinada **família** (princípio matemático) de algoritmos. Por exemplo `sklearn.naive_bayes` é o módulo de algoritmos para classificação da família Naïve Bayes.
- `algoritmo`: indica o algoritmo a ser selecionado dentro do módulo. Por exemplo, o módulo `sklearn.naive_bayes` oferece os algoritmos `GaussianNB`, `MultinomialNB` e `BernoulliNB` (são basicamente, três versões distintas do Naïve Bayes).

- `apelido` : nem sempre utilizado, servindo apenas para simplificar a referência ao algoritmo no corpo do programa. É a mesma coisa que fazemos ao apelidar a pandas de "pd".

7.5 Projeto prático — classificador multirrótulo

Chegou a hora! Nesta seção, apresentamos um programa-exemplo que envolve a utilização das bibliotecas pandas e scikit-learn para criar um classificador multirrótulo de cores de bandeiras. O exemplo aborda tudo o que acabamos de aprender neste capítulo:

- O algoritmo de classificação adotado é o k-NN.
- O desempenho preditivo do classificador é estimando através do método LOO, a partir do qual são produzidas matrizes de confusão para os diferentes rótulos de classe (cores de bandeiras).
- A base de dados de treinamento deste projeto é a `flags_transf.csv` , a versão pré-processada da base de dados `flags` que criamos em nosso último capítulo. Essa base de dados se encontra disponibilizada no **repositório de bases de dados** de nosso livro https://github.com/edubd/pandas.

A listagem a seguir apresenta o programa. A sua explicação detalhada é apresentada posteriormente. Você perceberá que código é relativamente simples para um programa que faz algo tão complexo quanto criar e avaliar um modelo de Machine Learning! Entretanto, há uma explicação. Essa simplicidade se deve principalmente ao fato de que **já estudamos e transformamos a base original** `flags` , deixando-a em um formato perfeito para ser consumida pelo k-NN.

Isto é, o Data Wrangling já foi executado, justamente a parte mais trabalhosa de qualquer processo de ciência de dados (e a nossa pandas foi a única ferramenta utilizada para conduzir essa tarefa).

Daí, se temos a base aprontada, sabemos como funciona o k-NN e também sabemos como funciona o LOO, tudo que precisamos é seguir a receita indicada na documentação da scikit-learn.

```
#P59: Programa final!
#     Cria um classificador k-NN
#     Avalia este classificador com o método LOO

#---------------------------------------------------------
#Parte 1: importação das bibliotecas
#---------------------------------------------------------
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier

#---------------------------------------------------------
#Parte 2: carrega a base de dados de treinamento
#         para um DataFrame
#---------------------------------------------------------
flags = pd.read_csv('c:/bases/flags_transf.csv')

#---------------------------------------------------------
#Parte 3: configura os parâmetros e variáveis auxiliares
#---------------------------------------------------------
#valor de k a ser usado no k-nn
k=3

#nomes dos labels (rótulos de classe)
labels=['red',
        'green',
        'blue',
        'gold',
        'white',
        'black',
        'orange']

#número de labels (=7)
q = len(labels)

#número de registros da base de treinamento (=194)
```

```
N = flags.shape[0]

#------------------------------------------------------------
#Parte 4: cria um classificador com o algoritmo k-NN
#         e faz a estimativa do desempenho preditivo
#         com o método LOO (Leave-One-Out)
#-------------------------------------------------------------

#-------------------------------------------------------------
#4.1 - for j: laço que precorre cada rótulo
#-------------------------------------------------------------
for j in range(0,q):
    print('-------------------------------------------------')
    print("PROCESSANDO O RÓTULO ", labels[j])

    #---------------------------------------------------------
    #4.1.1 instancia uma matriz de confusão para o rótulo j
    #---------------------------------------------------------
    mc = pd.DataFrame({'predito_nao': [0,0],
                       'predito_sim':[0,0]},
                      index = ['real_nao','real_sim'])

    #---------------------------------------------------------
    #4.1.2 divide a base de treinamento verticalmente
    #      em duas partes: X (atributos preditivos) e
    #                      Y (atributo classe)
    #---------------------------------------------------------
    X = flags.drop(columns=labels)
    Y = flags[labels[j]]

    #---------------------------------------------------------
    #4.2 - for i: laço que realiza o LOO para o rótulo j
    #---------------------------------------------------------
    for i in range(0,N):

        #-----------------------------------------------------
        #4.2.1 Separa os dados que serão utilizados para
        #      treinar o modelo
        #-----------------------------------------------------
```

```
    X_treino = X.drop([i])
    Y_treino = Y.drop([i])

    #-----------------------------------------------------
    #4.2.2 Separa o objeto de teste
    #-----------------------------------------------------
    X_teste = X.iloc[[i],:]
    Y_teste = Y.iloc[i]

    #-----------------------------------------------------
    #4.2.3 Treinamento do modelo k-NN com os dados
    #      de treino
    #-----------------------------------------------------
    modelo=KNeighborsClassifier(n_neighbors=k)
    modelo.fit(X_treino,Y_treino)

    #-----------------------------------------------------
    #4.2.4 Teste do modelo k-NN com o objeto de teste
    #-----------------------------------------------------
    pred = modelo.predict(X_teste)

    #-----------------------------------------------------
    #4.2.5 Atualiza a célula adequada da matriz de
    #      confusão em função do resultado do teste
    #-----------------------------------------------------
    if (Y_teste == 0):
        if (pred == 0): mc.iloc[0,0]+=1
        if (pred == 1): mc.iloc[0,1]+=1
    else:
        if (pred == 0): mc.iloc[1,0]+=1
        if (pred == 1): mc.iloc[1,1]+=1

#---------------------------------------------------------
#4.3 - Fim do LOO para o rótulo j:
#      imprime a sua matriz de confusão e acurácia
#---------------------------------------------------------
print(mc)
acuracia = (mc.iloc[0,0] + mc.iloc[1,1]) / N
print('acurácia = ',round(acuracia,2))
```

E aqui está a saída do programa, onde se imprime a matriz de confusão e a acurácia resultantes do processo LOO de cada rótulo de classe:

```
------------------------------------------------
PROCESSANDO O RÓTULO  red
          predito_nao  predito_sim
real_nao           20           21
real_sim           22          131
acurácia =  0.78
------------------------------------------------
PROCESSANDO O RÓTULO  green
          predito_nao  predito_sim
real_nao           88           15
real_sim           17           74
acurácia =  0.84
------------------------------------------------
PROCESSANDO O RÓTULO  blue
          predito_nao  predito_sim
real_nao           64           31
real_sim           28           71
acurácia =  0.7
------------------------------------------------
PROCESSANDO O RÓTULO  gold
          predito_nao  predito_sim
real_nao           81           22
real_sim           37           54
acurácia =  0.7
------------------------------------------------
PROCESSANDO O RÓTULO  white
          predito_nao  predito_sim
real_nao           18           30
real_sim           24          122
acurácia =  0.72
------------------------------------------------
PROCESSANDO O RÓTULO  black
          predito_nao  predito_sim
real_nao          125           17
real_sim           28           24
```

```
acurácia =  0.77
-----------------------------------------------
PROCESSANDO O RÓTULO  orange
          predito_nao  predito_sim
real_nao          163            5
real_sim           20            6
acurácia =  0.87
```

Vamos à explicação do código do programa. Incialmente, observe que ele possui quatro seções distintas, cada qual exemplificando uma parte específica do processo de classificação:

1. Importação das bibliotecas;
2. Importação da base de dados de treinamento;
3. Configuração de parâmetros;
4. Construção e avaliação do modelo de classificação.

Parte 1 — importação das bibliotecas

Em primeiro lugar, realizamos a importação da biblioteca pandas:

```
import pandas as pd
```

Em nosso programa, a pandas foi utilizada para executar diversas funções. A mais importante delas é estruturar a base de dados de treinamento em um DataFrame. Além disso, nós a utilizamos em diversas operações de fatiamento (por exemplo, para separar os conjuntos de atributos `X` e `Y`) e também para implementar a matriz de confusão `mc`. Isso demonstra o quanto a pandas é importante para a ciência de dados! Ao longo dos capítulos deste livro, utilizamos a pandas para implementar todas as atividades de Data Wrangling e mesmo agora, quando já estamos na fase de construção do modelo de Machine Learning, ainda precisamos empregar esta biblioteca!

Depois de importar a pandas, também fizemos a importação da scikit-learn. Para utilizar o k-NN oferecido por esta biblioteca, o comando `import` foi definido da seguinte forma:

```
from sklearn.neighbors import KNeighborsClassifier
```

Este comando faz a importação da classe `KNeighborsClassifier` , que faz parte do módulo `neighbors` da scikit-learn. É esta a classe que contém a implementação do algoritmo k-NN na scikit-learn.

Veja que no processo de importação, apelidamos a pandas de `pd` (como sempre fazemos), mas preferimos não atribuir nenhum apelido para a classe `KNeighborsClassifier` .

Parte 2 — importação da base de dados de treinamento

Esta parte do programa é composta por apenas uma linha, onde importamos a base de dados de treinamento `flags_transf.csv` para um DataFrame pandas denominado `flags` . Trata-se de uma operação simples, porém necessária em qualquer programa que envolva a construção de um classificador. Afinal de contas, para que seja possível construir um classificador, um dos artefatos necessários é exatamente uma base de dados de treinamento (o outro é o algoritmo de classificação).

Parte 3 — configuração de parâmetros

Neste trecho de código, definimos quatro variáveis para armazenar informações utilizadas durante o processo de construção do classificador:

- `k` : valor do número de vizinhos a ser adotado pelo k-NN;
- `labels` : lista que armazena os nomes dos atributos do conjunto `Y` (rótulos de classe);
- `q` : o número de rótulos de classe (nesse caso, igual a 7).
- `N` : o total de objetos da base de treinamento (como sabemos, `flags_transf.csv` possui dados de 194 países).

Parte 4 — construção e avaliação do modelo de classificação

Essa é a parte principal do programa, onde utilizamos o k-NN e a LOO para, respectivamente, criar e avaliar classificadores para cada um dos rótulos de classe.

Este processo é implementado através de dois laços `for` :

- `for j in range(0,q):` : laço externo, utilizado para processar cada rótulo de classe (`red` , `green` , `blue` , `gold` , `white` , `black` e `orange`).
- `for i in range(0,N):` : laço interno, responsável pela implementação da avaliação LOO.

Sendo assim, tudo começa no passo 4.1, onde definimos o laço `for j in range(0,q):` para percorrer toda a lista de rótulos de classe, desde `red` (o primeiro rótulo), até `orange` (o último). Ou seja, este laço realizará 7 iterações.

Então vamos examinar os comandos subsequentes. Ao longo do texto a seguir, reproduziremos o trecho de código e, logo depois, apresentaremos uma explicação sobre o mesmo.

```
#---------------------------------------------------------
#4.1.1 instancia uma matriz de confusão para o rótulo j
#---------------------------------------------------------
mc = pd.DataFrame({'predito_nao': [0,0],
                   'predito_sim':[0,0]},
                  index = ['real_nao','real_sim'])
```

Este comando cria uma matriz de confusão `mc` para o rótulo `j` (rótulo correntemente processado). Ao final do processo de LOO, a matriz estará totalmente preenchida com os resultados completos do teste do rótulo `j` .

```
#---------------------------------------------------------
#4.1.2 divide a base de treinamento verticalmente
#      em duas partes: X (atributos preditivos) e
#                      Y (atributo classe)
#---------------------------------------------------------
```

```
X = flags.drop(columns=labels)
Y = flags[labels[j]]
```

Aqui estamos utilizando toda a flexibilidade oferecida pela funcionalidade de fatiamento da pandas para dividir **verticalmente** o DataFrame `flags` (nossa base de treinamento) em duas partes, `X` e `Y`. A fatia `X` recebe o conjunto de atributos preditivos, ou seja, todos os atributos de `flags`, exceto os 7 rótulos de classe. Já a fatia `Y` recebe uma única coluna, correspondente ao rótulo de classe `j` (rótulo que está sendo correntemente processado no laço `for j`).

Logo depois, o passo 4.2 é iniciado, onde definimos o laço `for i in range(0,N):` para percorrer todos os objetos de treinamento (que acabaram de ser divididos em `X` e `Y`) e executar o processo de LOO. Esse laço realizará N iterações. A cada iteração, um modelo de classificação será construído utilizando-se N-1 objetos (todos, menos o objeto da posição `i`) e testado utilizando um único objeto (o objeto da posição `i`).

Vamos agora examinar detalhadamente os trechos de código subordinados ao laço `for i`. Novamente, nós primeiro reproduzimos o trecho de código e depois apresentamos uma explicação.

```
#-----------------------------------------------------
#4.2.1 Separa os dados que serão utilizados para
#       treinar o modelo
#-----------------------------------------------------
X_treino = X.drop([i])
Y_treino = Y.drop([i])
```

Nesta parte, realizamos o fatiamento dos DataFrames `X` e `Y` com o intuito de separar os objetos que serão utilizados para treinar o modelo. Conforme acabamos de comentar, a cada passo do LOO todos os objetos são utilizados para treinar o modelo, exceto um único objeto `i`.

```
#-----------------------------------------------------
#4.2.2 Separa o objeto de teste
#-----------------------------------------------------
```

```
X_teste = X.iloc[[i],:]
Y_teste = Y.iloc[i]
```

E aqui separamos o objeto que será utilizado para teste na iteração corrente do LOO (objeto da posição `i` na base de dados). Veja que pegamos tanto a parte `X` do objeto (atributos preditivos) como a sua parte `Y` (sua classe real). Precisamos da classe real para compará-la com a classe que será predita pelo modelo de classificação e, assim, realizar a atualização da matriz de confusão (passo 4.2.5 do programa).

É importante observar que a sintaxe `X.iloc[[i],:]` foi utilizada para forçar o tipo de `X_teste` como um DataFrame. Se não tivéssemos feito isso, o fatiamento retornaria uma Series por *default*, já que resulta em apenas uma linha (se estiver em dúvida, reveja o tópico sobre fatiamento do capítulo 3).

```
#-----------------------------------------------------
#4.2.3 Treinamento do modelo k-NN com os dados
#      de treino
#-----------------------------------------------------
modelo=KNeighborsClassifier(n_neighbors=k)
modelo.fit(X_treino,Y_treino)
```

Esta é a parte mais importante de todo o programa, onde criamos o modelo de classificação. Apesar disso, veja como ela é incrivelmente simples, consistindo apenas em duas linhas de código.

Inicialmente, instanciamos um objeto chamado `modelo`, do tipo `kNeighborsClassifier`. É importante chamar a atenção para dois aspectos. O primeiro é que, para utilizar a classe `kNeighborsClassifier`, torna-se preciso preliminarmente fazer o `import` da respectiva classe no scikit-learn, algo que fizemos logo no início de nosso programa (se você for utilizar outro algoritmo de classificação, como o Naïve Bayes, deverá fazer a importação da classe adequada). O segundo aspecto importante é que o `kNeighborsClassifier` foi instanciado com o parâmetro `n_neighbors=k`. Este parâmetro é utilizar para especificar o número de vizinhos a ser

adotado pelo `k-NN`. Desta forma, estamos definindo 3 vizinhos, já que havíamos feito a atribuição `k=3` na parte 2 do programa.

Após instanciarmos o objeto do tipo `kNeighborsClassifier` com o parâmetro adequado, podemos utilizá-lo para efetivamente treinar o modelo de classificação. Para isto, basta fazer uma chamada para o método `fit()`, passando como parâmetros as partes `X` e `Y` da base de treinamento, que neste caso estão respectivamente estruturadas nos DataFrames `X_treino` e `Y_treino`.

A classe `kNeighborsClassifier` aceita outros parâmetros além de `n_neighbors`. Um parâmetro muito importante, por exemplo, é `metrics`, que serve para definir a medida de distância a ser empregada. Para informações completas, consulte a documentação de `kNeighborsClassifier` em https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

```
#---------------------------------------------------
#4.2.4 Teste do modelo k-NN com o objeto de teste
#---------------------------------------------------
pred = modelo.predict(X_teste)
```

Uma vez o modelo tendo sido criado, podemos utilizá-lo para classificar o nosso objeto de teste `i`. Isso é feito com o uso do método `predict()`. Como parâmetro, devemos passar as características (*features*) de `i`, isto é, a sua parte `X`. Em nosso programa, essas características estão armazenadas no DataFrame `X_teste`.

Como resultado, `predict()` vai retornar um valor binário:

- `0`: significa que o objeto de teste `i` **não** está associado à classe `j`. Isto é, o modelo considera que a cor `j` não está presente na bandeira do país `i`.
- `1`: significa que o objeto de teste `i` está sim, associado à classe `j`. Ou seja, para o modelo de classificação, a cor `j` está presente na bandeira do país `i`.

Veja que, em nosso programa, o valor retornado é armazenado em uma variável chamada `pred`.

```
#------------------------------------------------------
#4.2.5 Atualiza a célula adequada da matriz de
#      confusão em função do resultado do teste
#------------------------------------------------------
if (Y_teste == 0):
    if (pred == 0): mc.iloc[0,0]+=1
    if (pred == 1): mc.iloc[0,1]+=1
else:
    if (pred == 0): mc.iloc[1,0]+=1
    if (pred == 1): mc.iloc[1,1]+=1
```

Este é o último trecho de código subordinado ao laço `for i`. Nele, comparamos o valor predito pelo modelo de classificação (`pred`) com o valor real do rótulo de classe na base de dados de treinamento (`Y_teste`). Assim, poderemos descobrir se o teste resultou em um `VP`, `FP`, `FN` ou `VN`, atualizando a célula adequada da matriz de confusão `mc`.

```
#----------------------------------------------------------
#4.3 - Fim do LOO para o rótulo j:
#      imprime a sua matriz de confusão e acurácia
#----------------------------------------------------------
print(mc)
acuracia = (mc.iloc[0,0] + mc.iloc[1,1]) / N
print('acurácia = ',round(acuracia,2))
```

Chegamos aos três comandos finais de nosso programa (passo 4.3). Esses comandos são executados para cada rótulo de classe `j`, depois que os N modelos de classificação foram gerados e testados no processo de LOO. Nesse momento, a matriz de confusão do label `j` estará prontinha. Então, imprimimos essa matriz e também calculamos e imprimimos o valor da acurácia do modelo de classificação.

Análise dos resultados

A base de dados `flags_transf.csv` possui um número muito pequeno de objetos, algo que não favorece o processo de construção do classificador (o algoritmo k-NN fica com pouca informação para aprender a mapear `X` em `Y`). Apesar disso, podemos considerar que a estimativa do desempenho preditivo — computada pelo processo de LOO — revelou que o modelo k-NN conseguiu atingir uma performance surpreendentemente boa para alguns dos rótulos de classe.

Essa conclusão pôde ser obtida levando-se em conta não apenas o valor da acurácia de cada rótulo, mas **principalmente** através da observação da composição da matriz de confusão. Neste sentido, podemos considerar, por exemplo, que o desempenho preditivo do classificador para o rótulo `green` foi o melhor dentre todos os labels. A matriz de confusão resultante do teste LOO deste rótulo é reproduzida a seguir:

```
------------------------------------------------
PROCESSANDO O RÓTULO  green
          predito_nao  predito_sim
real_nao           88           15
real_sim           17           74
acurácia =  0.84
------------------------------------------------
```

Veja que a matriz de confusão está bastante equilibrada. Isso significa que o classificador do rótulo `green` conseguiu se sair bem tanto para a previsão do valor "não", como do valor "sim". No geral, o classificador quase não errou, pois foram poucos casos de `FP` e `FN` em relação ao número total de casos. Em outras palavras, o valor de `VN` é bem maior do que o de `FP` e o valor de `VP` é bem maior do que o de `FN`. A acurácia total foi de 84%.

Por outro lado, podemos considerar que o classificador que apresentou o pior desempenho preditivo foi o do rótulo `orange`, cuja matriz de confusão é apresentada a seguir:

```
-----------------------------------------------
PROCESSANDO O RÓTULO  orange
          predito_nao  predito_sim
real_nao          163            5
real_sim           20            6
acurácia =  0.87
-----------------------------------------------
```

Curiosamente, o valor de acurácia do classificador para o rótulo `orange` é maior do que o obtido para o rótulo `green`. No entanto, esse resultado é ilusório. Ao observar a matriz, é trivial perceber que o classificador teve uma boa taxa de acertos para o valor "não", (seu valor de `VN` é bem maior do que o de `FP`), mas errou quase todas as previsões para o valor "sim" (tem valor de `FN` bem maior do que de `VP`).

A matriz de confusão está bastante desequilibrada e isso não é bom! Na verdade, seu valor alto de Acurácia (87%), foi determinado unicamente pelo fato de a base utilizada ter muito mais objetos "não" (onde o classificador se sai bem) do que "sim" (onde o desempenho do classificador é insatisfatório).

De fato, qualquer algoritmo de classificação teria dificuldades para conseguir um bom desempenho ao classificar o rótulo `orange`, pois há muito poucas bandeiras que possuem essa cor na base de treinamento (apenas 26 dentre os 194 países). O modelo de classificação não teve informação suficiente para aprender as características de um país que possui a cor laranja em sua bandeira.

Sugerimos ao leitor examinar as demais matrizes de confusão, empregando o mesmo método de análise para determinar quais classificadores apresentaram um bom desempenho preditivo. Conforme mencionado anteriormente, só faz sentido colocar um classificador em produção se a estimativa obtida para a sua acurácia for satisfatória. Sobre esse assunto, duas observações importantes precisam ser realizadas:

- A definição de acurácia alta ou satisfatória varia de acordo com o tipo de problema e o nível de exigência da empresa que vai utilizar o classificador.
- Na prática, muitas vezes a primeira rodada de um processo de ciência de dados não é suficiente para produzir um modelo satisfatório para a empresa. Neste caso, será preciso refazer alguma das etapas do projeto. Por exemplo, realizar novas transformações sobre a base de dados; ou tentar obter arquivos externos para combinar com a base, enriquecendo assim o conjunto de informações a serem disponibilizadas para o algoritmo de classificação. A ciência de dados é, por natureza, um processo cíclico, que usualmente necessita de algumas iterações até que o melhor resultado seja obtido.

CLASSIFICAÇÃO MULTIRRÓTULO

Classificação multirrótulo é um tema extremamente rico, que nos últimos anos vem sendo bastante estudado na Ciência da Computação.

Se você gostou do assunto e deseja aprender um pouco mais, uma dica é consultar o tutorial disponibilizado em https://doi.org/10.5753/sbc.7.3.

Conclusões e comentários finais

E assim, chegamos ao fim de nosso livro, onde foram apresentados os conceitos fundamentais e as principais técnicas para Data Wrangling oferecidas pela biblioteca pandas, a mais importante biblioteca para ciência de dados do Python.

A pandas oferece ao usuário técnicas provenientes de diferentes áreas (Estatística, Computação etc.), com o intuito de viabilizar o processo de seleção, estudo, limpeza e transformação de bases de dados estruturadas nos mais diversos formatos.

Ela é incrivelmente flexível e poderosa, o que explica a sua larga utilização por profissionais envolvidos com ciência de dados em todo o mundo.

Aos que desejarem tirar dúvidas ou realizar comentários, sintam-se à vontade para contatar o autor do livro. Um abraço e até a próxima!