Capítulo 1

Ciência de Dados com Reprodutibilidade usando Jupyter

João Felipe Pimentel, Gabriel P. Oliveira, Mariana O. Silva,
Danilo B. Seufitelli e Mirella M. Moro

Abstract

Data Science has become a trending research topic in Computer Science due to the growing interest in extracting knowledge from different data sources. In such a context, Jupyter Notebook has consolidated itself as one of the main tools used by data scientists to perform exploratory data analysis in a fast and straightforward way, with a high potential for code reproduction. Hence, this JAI aims to present Jupyter with reproducibility for developing Data Science projects. The content is tailored for students and professionals with some programming experience. In particular, we first introduce Jupyter and its general use to develop solutions for Data Science. Then, we present Jupyter advanced topics and address ways to promote open science. Finally, this JAI overviews Data Science with Jupyter Notebooks by combining concepts and theoretical foundations with practical examples and real-world data.

Resumo

Ciência de Dados tornou-se um tópico de pesquisa emergente na Ciência da Computação devido ao crescente interesse em extrair conhecimento de diferentes fontes de dados. Nesse contexto, o Jupyter Notebook vem se consolidando como uma das principais ferramentas utilizadas por cientistas de dados para realizar análises exploratórias de dados de forma rápida e direta, com alto potencial de reprodução de código. Dessa forma, o objetivo deste capítulo é apresentar o Jupyter com reprodutibilidade para a realização de projetos em Ciência de Dados. O conteúdo é organizado para estudantes e profissionais com alguma experiência em programação. Em particular, primeiro apresentamos o Jupyter e seu uso geral para desenvolver soluções para Ciência de Dados. Em seguida, apresentamos tópicos avançados do Jupyter e abordamos maneiras de promover a ciência aberta. Para concluir, este JAI apresenta uma visão geral de Ciência de Dados com Jupyter Notebooks combinando conceitos e fundamentos teóricos com exemplos práticos e dados do mundo real.

1.1. Introdução

A Ciência de Dados tem como principal objetivo extrair informações úteis de dados. No processo de extração dessas informações, dados brutos com diversos formatos e estruturas são obtidos de diversas fontes, pré-processados para atingir um nível esperado de integridade, e finalmente analisados através da extração de estatísticas, visualizações e técnicas de mineração de dados e aprendizado de máquina para que as informações desejadas sejam obtidas. Todo esse processo deve ser feito com rigorosidade para que as informações sejam obtidas com qualidade e reprodutibilidade. Ou seja, após a extração das informações, é importante que seja possível reproduzir a análise e chegar a resultados próximos para dados semelhantes. Essa característica ajuda a validar que a informação extraída está correta e possibilita expandir a análise para novos dados e resultados.

O uso de Jupyter Notebooks para publicar pesquisas reprodutíveis é defendido por combinar texto de relatório com código executável [Kluyver et al. 2016]. Apesar do formato em si não garantir a reprodutibilidade [Pimentel et al. 2019], a documentação de processos com resultados pode ser bem valiosa para tal objetivo, e seu formato possibilita e facilita a análise exploratória de dados [Perkel 2018].

Este capítulo está organizado da seguinte forma. A Seção 1.2 introduz o conjunto de dados utilizado ao longo deste capítulo. A Seção 1.3 apresenta os conceitos iniciais de Jupyter. As Seções 1.4, 1.5 e 1.6 apresentam, respectivamente, a preparação de dados, métodos importantes e visualização do Jupyter para Ciência de Dados com Python. A Seção 1.7 resume aspectos avançados do Jupyter para a sua extensão. A Seção 1.8 apresenta como fazer ciência aberta com o Jupyter, compartilhando notebooks e resultados e garantindo a reprodutibilidade dos mesmos. Por fim, a Seção 1.9 apresenta considerações finais. A Figura 1.1 apresenta uma visão geral do capítulo com as ferramentas e conceitos que são discutidos em cada seção.

Figura 1.1. Conteúdo do curso.

1.2. Conjunto de Dados

Esta seção apresenta o conjunto de dados que guia o restante deste capítulo (Seções 1.3 a 1.7). Um conjunto único facilita o entendimento dos tópicos e reduz o tempo de familiarização com os dados em si. Usar dados reais em um projeto de Ciência de Dados também enriquece o aprendizado, pois dados sintéticos podem não ser fieis aos desafios das etapas

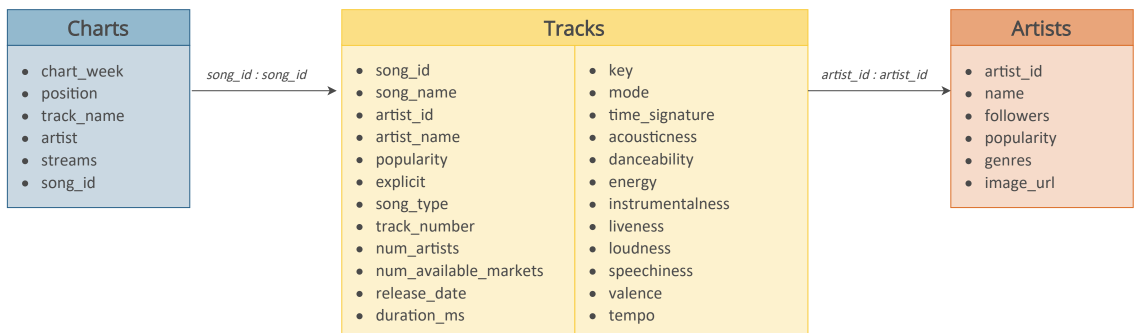


Figura 1.2. Esquema do conjunto de dados de sucesso musical utilizado como estudo de caso ao longo das Seções 1.3 a 1.7.

de tratamento e processamento dos dados [Igual and Seguí 2017, Skiena 2017]. Aqui, o conjunto de dados refere-se a sucesso na indústria da música, uma das mais dinâmicas e importantes no cenário do entretenimento mundial. Especificamente, utilizamos dados provenientes do Spotify,[1] o serviço de *streaming* de áudio mais popular do mundo, que reúne mais de 345 milhões de usuários em 178 países e territórios.[2]

A Figura 1.2 apresenta o esquema do conjunto de dados. Seguindo a metodologia de [Oliveira et al. 2020], os dados são obtidos a partir das paradas semanais de sucesso do Spotify durante 2020. Tais paradas são um *ranking* das 200 músicas mais ouvidas durante a semana em questão e são disponibilizadas separadamente para cada mercado em que a empresa atua. Porém, por simplificação, consideramos apenas os dados globais agregados, ou seja, obtidos somando-se os *streams* de todos os mercados. Para cada lista, obtém-se informação de suas músicas, incluindo lista de intérpretes, data de lançamento e características acústicas. Também inclui-se dados sobre artistas que interpretam tais músicas, destacando-se seus gêneros musicais e indicadores de popularidade.

1.3. Jupyter Básico

Jupyter é o sistema mais usado para programação literária interativa [Shen 2014]. O paradigma de programação literária busca ajudar na comunicação de programas através da alternância de texto em linguagem natural formatada, pedaços de código executáveis, e resultados de computações. O texto em linguagem natural é usado tanto para explicar o código quanto para comentar o resultado obtido [Perkel 2018]. A interatividade do Jupyter permite que este paradigma seja usado em tempo real para análises de dados, com o processo sendo documentado durante o desenvolvimento, resultados sendo exibidos de forma instantânea e discutidos imediatamente em linguagem natural.

Jupyter foi desenvolvido como uma evolução do IPython, um sistema de *REPL* (laço de leitura-avaliação-impressão) para Python [Perkel 2021b]. Apesar de suportar várias linguagens de programação, a linguagem para códigos executáveis mais comum em Notebooks (documentos do Jupyter) ainda é Python [Pimentel et al. 2019]. Python é uma linguagem de programação interpretada multiparadigma com suporte a programação

[1] Spotify API: `https://developer.spotify.com/`
[2] Spotify Company Info: `https://newsroom.spotify.com/company-info/`

imperativa, programação funcional e orientação a objetos. Inclui estruturas de dados e módulos em sua instalação, com flexibilidade para manipular as estruturas e integrar-se com ferramentas escritas em outras linguagens. Assim, cientistas de todas as áreas usam esta linguagem em seus experimentos computacionais [Perkel 2018, Pimentel 2021].

Esta seção tem o objetivo de apresentar a estrutura geral de um Notebook (Seção 1.3.1), bem como introduzir uma biblioteca básica amplamente utilizada em Jupyter Notebooks para projetos de Ciência de Dados (Seção 1.3.3). Além disso, abordamos como adicionar dados de diferentes formatos no Jupyter (Seção 1.3.2).

1.3.1. Estrutura Geral de um Notebook

Jupyter armazena Notebooks como documentos JSON com extensão ".ipynb" (para mais detalhes do que é um documento JSON, ver Seção 1.3.3). Esses documentos são compostos por *células* que podem ser de três tipos: uma célula de *código* possui código executável que produz resultados; uma célula de *Markdown* possui texto formatado; e uma célula *bruta* possui texto que não é nem Markdown, nem código executável. Em geral, células brutas são usadas por ferramentas externas que convertem notebooks em outros formatos.

Os resultados de uma execução de uma célula de código são armazenados na própria célula e exibidos interativamente durante a execução do notebook. Jupyter permite a visualização de textos (saída padrão), erros (saída de erros), imagens (PNG, JPG e SVG), HTML com JavaScript e Markdown. Além do resultado da execução, células de código também armazenam um contador de execução, que pode ser usado para indicar aproximadamente a ordem de execução. Por conta da interatividade do Jupyter, um notebook não precisa ser executado inteiramente de cima para baixo – como ocorre normalmente em arquivos de código. Ao invés disso, pode-se executar em qualquer ordem desejada e até mesmo executar uma mesma célula mais de uma vez.

A Figura 1.3 apresenta um exemplo de notebook executado, com duas células de

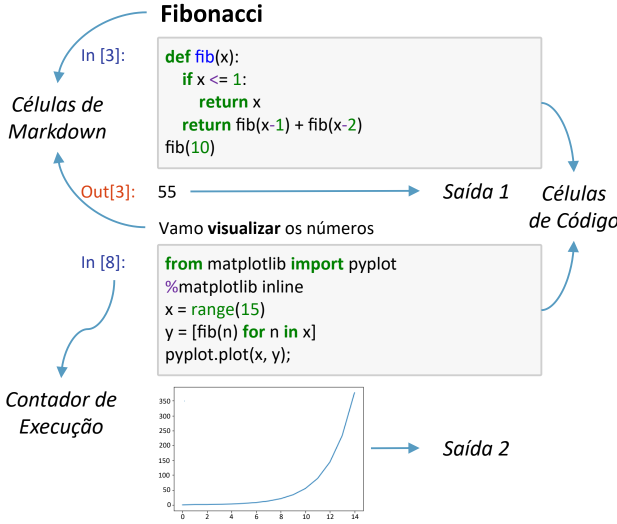


Figura 1.3. Exemplo de notebook executado com Markdown, código e resultados (adaptado de [Pimentel et al. 2019]).

Markdown e duas células de código. Na esquerda das células de código, há contadores de execução que indicam a ordem. Abaixo das células de código, Jupyter exibe os resultados. Perceba que a primeira célula de código retorna um número, identificado por **Out[3]**, e a segunda célula exibe uma imagem sem retorná-la. Nesta imagem, os contadores de execução marcam 3 e 8. Isso significa que ocorreram duas execuções antes da execução da primeira célula de código e quatro execuções entre a primeira célula de código e a segunda. Essas execuções não ficam armazenadas no notebook apesar de poderem contribuir para estado atual da execução.

1.3.2. Introdução ao *pandas*

A biblioteca *pandas*, cujo nome é derivado do termo inglês *Panel Data*, foi criada para a linguagem *Python* e é muito utilizada por cientistas de dados, pois fornece estruturas de dados de alto desempenho e ferramentas para análise de dados em formato tabular e de séries temporais, além de ser fácil de utilizar. Para isso, é necessário importá-la ao projeto incluindo um apelido para referenciá-la (e.g., *pd*), com o trecho de código:

```
import pandas as pd.
```

O principal recurso do *pandas* é o *DataFrame*, um objeto rápido e eficiente para manipulação de dados com indexação integrada. A Figura 1.4 apresenta um *DataFrame*, i.e., uma estrutura tabular com linhas e colunas. As linhas possuem um índice específico para acessá-las, que pode ser um nome ou um valor. Já as colunas, sendo chamadas de *Series*, são um tipo especial de dados que consiste numa lista de vários valores, onde cada valor possui um índice. Portanto, a estrutura de dados do *DataFrame* pode ser entendida como uma planilha, porém muito mais flexível.

pandas oferece muitas funções, incluindo para transformar (facilmente) qualquer conjunto de dados: adicionar ou remover linhas e colunas, redefinir índices, reordenar e renomear colunas. Também permite operações de álgebra relacional (e.g., projeção, junção e concatenação) e funções de limpeza (e.g., preenchimento, substituição ou inserção de valores nulos – *null*). O *pandas DataFrame* possui funções de alto desempenho para agregar, mesclar e juntar conjunto de dados em diferentes formatos: CSV, TXT, MS

	followers	popularity	track_name	chart_position	genres
Tones And I	2664450	83	Dance Monkey	1	australian pop, pop
Maroon 5	29888961	91	Memories	4	pop, pop rock
Post Malone	31726853	94	Circles	6	dfw rap, melodic rap, rap
Dua Lipa	23478007	95	Don't Start Now	7	dance pop, pop, pop dance, uk pop
Shawn Mendes	31997377	91	Señorita	9	canadian pop, pop, post-teen pop, viral pop

Figura 1.4. Estrutura tabular de um *DataFrame* e seus principais componentes.

Excel, SQL e HD5. Para dados incompletos ou não estruturados, oferece tratamento de dados ausentes e alinhamento inteligente de dados. Uma de suas vantagens é a excelente integração com diversas outras bibliotecas do Python para aprendizado de máquina (e.g., *scikitlearn*) e geração de visualizações (e.g., *matplotlib*).

A seguir, estão as principais ferramentas da biblioteca *pandas* alinhadas às necessidades de cientistas de dados: como utilizar as funções do *pandas* para criar e manipular *DataFrames* e *Series*; as principais funções que fornecem uma visão geral dos dados; e funções para manipular e ajustar os dados que facilitam projetos de Ciência de Dados.

1.3.2.1. Criar Objetos do Tipo *Series*

Um objeto do tipo *Series* é um *array* de uma dimensão e possui uma lista de valores de qualquer tipo de dados (e.g., strings, caracteres, booleanos, números e datas). No entanto, não é recomendado colocar tipos diferentes numa mesma *Series*, pois pode-se perder vantagens de desempenho. O exemplo a seguir cria uma *Series* em Python. Como resultado, é possível identificar os valores (bandas e artistas), o tipo (*object*) e os índices. Nota: por legibilidade, a saída dos comandos é mostrada ao lado (e não abaixo).

```
[1]: import pandas as pd
```

```
[2]: # Criando uma Série formada por nomes de bandas e cantores famosos
     artistas_1 = pd.Series(["Beatles", "Michael Jackson",
                             "Rihanna", "Skank"])
     artistas_1
```

```
[2]: 0            Beatles
     1    Michael Jackson
     2            Rihanna
     3              Skank
     dtype: object
```

Toda *Series* possui um índice (*index*) que rotula cada elemento e permite acessar seus valores. O exemplo anterior não tem índice específico, então o *pandas* cria um objeto do tipo *RangeIndex* de 0 até o número de elementos menos um. Por exemplo, **artistas[0]** retorna o valor "**Beatles**". Pode-se criar um índice próprio que não precisa ser numérico e nem exclusivo como o próximo exemplo. Porém, esse recurso requer cuidado, pois a falta de exclusividade dos índices pode ocasionar problemas de manipulação.

```
[4]: # Serie com índices textuais: banda e artista
     artistas = pd.Series(["Beatles", "Michael Jackson", "Rihanna", "Skank"],
                          index=["banda", "artista", "artista", "banda"])
     artistas
```

```
[4]: banda              Beatles
     artista    Michael Jackson
     artista            Rihanna
     banda                Skank
     dtype: object
```

```
[5]: # Como os índices não são exclusivos, o codigo a seguir acessa os valores
     # de todas as 'bandas'
     artistas['banda']
```

```
[5]: banda    Beatles
     banda      Skank
     dtype: object
```

1.3.2.2. Criar Objetos do Tipo *DataFrames*

DataFrames são poderosas estruturas de dados tabulares bidimensionais com tamanho mutável e potencialmente heterogêneos, e são formados por um conjunto de *Series*. Ou

Tabela 1.1. Principais funções de manipulação de estruturas e objetos do *pandas*.

	Função	Descrição
Estruturas	drop()	Remove as linhas ou colunas de *Series*/*DataFrame*, especificado pelo índice ou pelo eixo. Retorna o objeto sem as linhas ou colunas especificadas.
	df['Coluna'] = 'valor'	Adiciona uma nova coluna em um *DataFrame*.
	df.columns = ['coluna 1', ..., 'coluna n']	Renomeia as n colunas de um *DataFrame*. O nome das novas n colunas deverá ser representada como uma lista.
	append()	Concatena duas ou mais *Series* ou *DataFrames*.
Objetos	shape	Retorna o número de linhas e/ou colunas de uma *Series* ou *DataFrame*.
	index	Informações dos índices dos objetos, tais como nomes e tipos.
	columns	Retorna o nome de todas as colunas de um *DataFrame*.
	dtypes	Retorna o nome e tipo de dados de todas as colunas de um *DataFrame*.

seja, as *Series* são colunas das tabelas *DataFrames*. Para criar um *DataFrame*, é necessário instanciar um objeto a partir da classe *DataFrame* do *pandas*. Como as *Series*, os *DataFrames* têm índices, mas também possuem colunas identificadas por nome. Se não explicitar os nomes das colunas, o *pandas* cria um *RangeIndex* de 0 até o número de colunas menos um. Pode-se criar um *DataFrame* de diversas maneiras. Os comandos a seguir mostram duas formas bastante comuns: a partir de uma lista de elementos; e a partir de um dicionário, onde a chave é o nome da coluna, e o valor, uma lista de objetos.

[6]:
```
# Construindo um dataframe a partir de uma lista de elementos
df = pd.DataFrame([
    ["Coldplay", 29397183], ["Katy Perry", 17784767], ["Adele", 26296798]])
df
```

[6]:

	0	1
0	Coldplay	29397183
1	Katy Perry	17784767
2	Adele	26296798

[7]:
```
# Construindo um dataframe a partir de um dicionário
data = {'Artista': ["Coldplay", "Katy Perry", "Adele"],
        'Followers': [29397183, 17784767, 26296798]}
df = pd.DataFrame(data)
df
```

[7]:

	Artista	Followers
0	Coldplay	29397183
1	Katy Perry	17784767
2	Adele	26296798

O acesso aos dados do *DataFrame* é feito de várias formas discutidas mais adiante. De modo simplificado, pode-se acessar uma coluna em *DataFrame* com o comando: **df['coluna']**. Sem índices, **df[0]** acessa os valores da primeira coluna; com índice, a mesma coluna é acessada utilizando **df['Artista']**.

1.3.2.3. Manipulação de *Series* e *DataFrames*

As funções para manipular *Series* e *DataFrames* são praticamente as mesmas. Porém, nem todas são aplicáveis a ambos objetos devido às limitações lógicas (e.g., funções para linhas e colunas não se aplicam a objetos *Series*). O *pandas* disponibiliza muitas funções em sua documentação.[3] A Tabela 1.1 destaca algumas das principais e suas aplicações. Algumas dessas são apresentados nas seções a seguir no contexto do estudo de caso.

[3]Link para acesso a documentação do *pandas*: `https://pandas.pydata.org/docs/`

1.3.3. Importação de Dados

Ciência de Dados extrai informações úteis dos dados. Porém, muitos projetos obrigam seus cientistas a reunir uma miscelânea de padrões de fontes de dados, seja em CSV, JSON, SQL, ou outro formato. Assim, é crucial saber lidar com os principais formatos de dados em Python, bem como realizar tal tarefa eficientemente; afinal, os dados são o ponto de partida de análises estatísticas, aplicação de técnicas de aprendizado de máquina, entre outras. O Python oferece bibliotecas e pacotes para os principais formatos *open data* e proprietários. Portanto, esta seção aborda a importação dos seguintes formatos de dados: CSV, TSV, XLS, JSON, SQL e XML. Embora não exaustiva, é raro um projeto de Ciência de Dados não lidar com ao menos um desses formatos.

Arquivos Texto. Arquivos de texto são amplamente utilizados em projetos de Ciência de Dados. Porém, existem formatos de arquivos textuais distintos, tais como CSV e TSV. Por definição, o CSV é um formato de arquivo tabular cujos valores são separados por vírgulas. A biblioteca *pandas* possui a função **read_csv()** para a leitura deste formato de arquivos. O exemplo a seguir mostra a importação de arquivos CSV em Python.

[2]:
```
# A função read_csv é utilizada
# para ler arquivos texto
artistas = pd.read_csv(
    'spotify_artists.csv')
artistas
```

[2]:

	name	followers	popularity	genres
0	Coldplay	29397183	90.0	['permanent wave', 'pop']
1	Ninho	4239063	84.0	['french hip hop', 'pop urbaine']
2	KEVVO	167419	75.0	['perreo', 'reggaeton', 'reggaeton flow', 'tra...
3	Olivia Rodrigo	1134117	89.0	['alt z', 'pop', 'post-teen pop']
4	Harry Styles	13439256	91.0	['pop', 'post-teen pop']

No entanto, os arquivos CSV demandam cuidado, pois geralmente causam conflitos devido à necessidade de cercar quebras de linha, aspas duplas e vírgulas no conteúdo de campos utilizando aspas duplas. Vírgulas literais são muito comuns em dados de texto, por exemplo, dados financeiros ($R\$120,38$). Nesse caso, em arquivos CSV, tais valores precisam ser escapados, normalmente utilizando aspas ("$R\$120,38$"). Outra opção é utilizar arquivos separados por outros delimitadores, tais como a tabulação.

Arquivos de valores separados por tabulação (TSV) também possuem formato de texto simples para armazenar dados em uma estrutura tabular (linhas e colunas). Analogamente, cada linha da tabela corresponde a uma linha do arquivo, e cada valor de campo de um registro (coluna) é separado por um caractere de tabulação (TAB). A biblioteca *pandas* oferece duas funções para a leitura de arquivos TSV: **read_table()** e **read_csv()** com **t** de delimitador, como mostram os comandos a seguir.

[3]:
```
# A função read_table é utilizada
# para ler arquivos .tsv
charts = pd.read_table(
    'spotify_charts.tsv')
charts
```

[3]:

	chart_week	position	track_name	artist	streams
0	2020-07-16	200	Como Llora	Juanfran	4534139
1	2020-07-23	1	ROCKSTAR (feat. Roddy Ricch)	DaBaby	35694103
2	2020-07-23	2	Savage Love (Laxed - Siren Beat)	Jawsh 685	33897676
3	2020-07-23	3	Blinding Lights	The Weeknd	30485774
4	2020-07-23	4	Watermelon Sugar	Harry Styles	26680752

[4]:
```
# Alternativamente, pode-se utilizar o
# parâmetro delimiter para separar com \t
charts = pd.read_csv(
    'spotify_charts.csv', delimiter='\t')
charts
```

[4]:

	chart_week	position	track_name	artist	streams
0	2020-07-16	200	Como Llora	Juanfran	4534139
1	2020-07-23	1	ROCKSTAR (feat. Roddy Ricch)	DaBaby	35694103
2	2020-07-23	2	Savage Love (Laxed - Siren Beat)	Jawsh 685	33897676
3	2020-07-23	3	Blinding Lights	The Weeknd	30485774
4	2020-07-23	4	Watermelon Sugar	Harry Styles	26680752

Com TAB como separador entre os campos, um campo não pode conter um TAB. Porém, os TABs geralmente não aparecem nos itens de dados, portanto, isso raramente é uma restrição. TSV é um formato de arquivo simples com amplo suporte, muito usado para mover dados tabulares entre diferentes aplicações que oferecem suporte ao formato.

Planilhas Excel. A biblioteca *pandas* permite a importação de arquivos Excel (xls e xlsx) em Python. Similar aos exemplos anteriores, utiliza-se a função `read_excel('arquivo.xlsx')`. Observe que para versão anterior do Excel, pode ser necessário usar a extensão de arquivo 'xls'. Para a importação de uma planilha específica em um mesmo arquivo, é preciso utilizar o parâmetro `sheet_name`. Por exemplo, suponha que tenhamos um arquivo chamado 'dados.xlsx'. Neste arquivo, ente suas diversas abas, temos uma chamada *Streams*. Para importar apenas o conteúdo da aba *Streams* do arquivo dados.xlsx, programa-se: `df = pd.read_excel ('dados.xlsx', sheet_name='Streams')`. Caso o `sheet_name` não seja especificado, importa-se a primeira aba.

JavaScript Object Notation (JSON). Um arquivo JSON armazena estruturas de dados simples e objetos no formato *JavaScript Object Notation*, um formato padrão de intercâmbio de dados. Esses arquivos são leves, textuais, legíveis por humanos e editáveis com editor de texto. Arquivos JSON representam dados com o conceito de chave e valor: cada valor tem uma chave que descreve seu significado. Por exemplo, a *chave:valor* `artista:'Michael Jackson'` representa o artista 'Michael Jackson'. Para importar arquivos JSON, o *pandas* tem a função `read_json()`, com funcionamento similar às anteriores.

Extensible Markup Language (XML). XML é uma linguagem de marcação para representar e distribuir dados semiestruturados. É semelhante ao JSON, porém os dados são organizados entre elementos (ou *tags*) que definem semântica ao valor, sempre com uma *tag* inicial e *tag* final (`<email>email@example.com</email>`). Arquivos XML são úteis para lidar com bases de dados pequenas, médias, ou em atividades que precisam de um esquema de dados flexível. Em Python, existem bibliotecas para importar e manipular XML, tais como a *DOM* (*Document Object Model*), *SAX* (*Simple API for XML*) e *etree.ElementTree*. Esta última é mais comum, pois possui implementação simples e bom desempenho. O comando `parse()` faz a leitura do arquivo XML, e o comando `findall()` retorna os elementos de uma *tag* específica. Assim, é possível manipular os arquivos XML e construir *DataFrames* em formatos tabulares para utilizar todas as funções do *pandas*.

Structured Query Language (SQL). A importação de dados a partir de um banco de dados SQL requer etapas adicionais em comparação aos outros formatos, pois o *pandas* não suporta a integração com bancos de dados de forma nativa e depende de bibliotecas de terceiros para estabelecer a conexão. Assim, após a conexão com o banco de dados, é possível trabalhar diretamente com *pandas* usando a função `read_sql_query()`. Existem várias bibliotecas que fazem a integração do Python com os Sistemas Gerenciadores de Banco de Dados (SGBDs). Uma delas é a biblioteca *sqlalchemy*, responsável por criar uma *engine* de conexão com diferentes SGBDs, e.g., MySQL, Oracle, Postgres e Sqlite. Além da *engine*, é preciso importar um *driver* de conexão, o qual é específico para cada SGBD (e.g., para MySQL, usa-se o comando `!pip install pymysql`).

1.4. Preparação de Dados para Ciência

Em Ciência de Dados, dados de qualidade são pré-requisito para pesquisas válidas, descobertas significativas, modelos de Aprendizado de Máquina, entre outros. Porém, no mundo real, dados brutos costumam ser incompletos, ruidosos, inconsistentes e, às vezes, estão em formato inutilizável. Portanto, antes de alimentá-los a modelos (e outras etapas de pesquisa), é fundamental averiguar a integridade de dados e identificar possíveis problemas. Este processo é denominado pré-processamento de dados.

Essencialmente, preparar dados significa adequá-los para servirem de entrada nos processos da pesquisa. Existem muitas técnicas de pré-processamento que, geralmente, acontecem em etapas organizadas nas seguintes categorias: Limpeza de Dados, Integração de Dados, Transformação de Dados, e Redução de Dados. As etapas do pré-processamento não são mutuamente exclusivas e são altamente dependentes do conjunto de dados; ou seja, podem trabalhar em conjunto, mas não são obrigatórias.

Em particular, a Limpeza de dados pode remover ruído e corrigir inconsistências nos dados. A Integração de dados mescla dados de várias fontes em um armazenamento de dados coerente, como um armazém de dados. Transformações de dados, como normalização, podem melhorar a precisão e eficiência de algoritmos que envolvem medições de distância. Então, a Redução de dados pode diminuir o tamanho dos dados agregando ou eliminando recursos redundantes. Através de exemplos com dados reais, esta seção define e descreve cada uma dessas etapas técnicas.

1.4.1. Limpeza de Dados

A Limpeza de dados é o processo de detecção e correção de registros incorretos ou corrompidos. Refere-se à identificação e, em seguida, a substituição, modificação ou exclusão de partes incompletas, imprecisas ou irrelevantes. Em geral, a Limpeza de dados leva a uma sequência de tarefas que visam melhorar a qualidade dos dados. Algumas dessas tarefas incluem não só lidar com dados ausentes e duplicados, mas também remover dados ruidosos, inconsistentes e outliers. Esta seção apresenta cada uma dessas tarefas através de uma versão aleatoriamente modificada das tabelas do estudo de caso.

Dados ausentes. Representam um obstáculo para a criação da maioria dos modelos de Aprendizado de Máquina e outras análises. Portanto, é necessário identificar campos para os quais não há dados e, em seguida, compensá-los adequadamente. Dados ausentes podem ocorrer quando nenhuma informação é fornecida para um ou mais registros (ou atributos inteiros) da base de dados. Em um *pandas DataFrame*, os dados ausentes são representados como `None` ou `NaN` (*Not a Number*), embora `NaN` seja o marcador de valor ausente padrão por razões de velocidade e conveniência computacional.

Após importar pacotes necessários e carregar o conjunto de dados, inicia-se o processo de Limpeza de dados. Para facilitar a detecção, o *pandas* fornece a função `isna()` para identificar valores ausentes em um *DataFrame*. Esta função retorna uma matriz *booleana* indicando se cada elemento correspondente está faltando (`True`) ou não (`False`). O exemplo a seguir apresenta o use desta função no *DataFrame* `df` e seleciona as três primeiras linhas por meio da função `head(3)`. Esta seleção foi feita neste capítulo por questões de legibilidade.

```
[3]: # Esta célula identifica valores ausentes no Dataframe
     # e exibe as três primeiras linhas
     pd.set_option('display.max_columns', 5)
     df.isna().head(3)
```

[3]:

	artist_id	name	...	genres	image_url
0	False	False	...	False	False
1	False	False	...	False	False
2	False	False	...	False	False

3 rows × 6 columns

Segundo o exemplo anterior, o conjunto de dados está aparentemente completo. Porém, tal resposta não é suficiente para descartar a hipótese de que existem dados ausentes. Para uma melhor averiguação, pode-se resumir cada coluna no *DataFrame booleano* somando os valores **False=0** e **True=1**. Tal processo retorna o número total de valores ausentes. Também pode-se dividir cada valor pelo número total de linhas no conjunto de dados, resultando na porcentagem de tais ausências, conforme o exemplo a seguir.

```
[4]: # Calcula o total e a % de valores ausentes
     num_ausentes = df.isna().sum()
     porc_ausentes = df.isna().sum() * 100 / len(df)
     # DataFrame com as informações computadas acima
     df_ausentes = pd.DataFrame({
         'Coluna': df.columns,
         'Dados ausentes': num_ausentes,
         'Porcentagem': porc_ausentes
     })
     df_ausentes
```

[4]:

	Coluna	Dados ausentes	Porcentagem
artist_id	artist_id	0	0.00
name	name	0	0.00
followers	followers	0	0.00
popularity	popularity	62	9.92
genres	genres	40	6.40
image_url	image_url	10	1.60

O *DataFrame* resultante contém 62 popularidades, 40 listas de gêneros e dez url de imagens ausentes, resultando em 9,9%, 6,4% e 1,6% de registros de cada coluna, respectivamente. Após essa identificação, é necessário tratar esses dados. A abordagem mais simples é eliminar todos os registros que contenham valores ausentes. No *pandas*, o método **dropna()** permite analisar e descartar linhas/colunas com valores nulos. O parâmetro **axis** determina a dimensão em que a função atuará: **axis = 0** remove todas as linhas que contêm valores nulos, e **axis = 1** remove colunas, conforme o seguinte exemplo.

```
[5]: # Elimina linhas com valores ausentes
     novo_df = df.dropna(axis=0)
     print(f"""\
     Nº de linhas do DF original: {len(df)}
     Nº de linhas do DF novo: {len(novo_df)}
     Nº de linhas com pelo menos 1 valor ausente: {
     (len(df) - len(novo_df))}""")

     Nº de linhas do DF original: 625
     Nº de linhas do DF novo: 529
     Nº de linhas com pelo menos 1 valor ausente: 96
```

```
[6]: # Eliminando colunas com valores ausentes
     novo_df = df.dropna(axis=1)
     print(f"""\
     Nº de colunas do DF original: {len(df.columns)}
     Nº de colunas do DF novo: {len(novo_df.columns)}
     Nº de colunas com pelo menos 1 valor ausente: {
     (len(df.columns) - len(novo_df.columns))}""")

     Nº de colunas do DF original: 6
     Nº de colunas do DF novo: 3
     Nº de colunas com pelo menos 1 valor ausente: 3
```

Os exemplos anteriores removeram as linhas/colunas onde pelo menos um elemento está faltando. Ambas abordagens são particularmente vantajosas para amostras de grande volume de dados, onde os valores podem ser descartados sem distorcer significativamente a interpretação. No entanto, apesar de ser uma solução simples, ela ainda apresenta o risco de perder dados potencialmente úteis.

Uma alternativa mais confiável para lidar com dados ausentes é a imputação. Em vez de descartar tais dados, a imputação procura substituir seus valores por outros. Nessa abordagem, os valores ausentes são inferidos a partir dos dados existentes. Existem várias maneiras de imputar os dados, sendo a imputação por valor constante ou por estatísticas básicas (média, mediana ou moda) as mais simples. No exemplo a seguir, os valores de popularidade ausentes são substituídos pela média das popularidades existentes por meio da função **fillna**.

[7]:
```
# Substituindo NaNs pela média de valores presentes
copia_df = df.copy()
copia_df['popularity'].fillna(copia_df['popularity'].mean(), inplace=True)
copia_df.sample(2)
```

[7]:

	artist_id	name	...	genres	image_url
376	2kqUKsTuEj1IPbm6BSn1AU	Rich Music LTD	...	['latin', 'reggaeton', 'trap latino']	https://i.scdn.co/image/7eff816abb6777c0d9efdd...
600	5QdEbQJ3ylBnc3gsIASAT5	G Herbo	...	['chicago drill', 'chicago rap', 'drill', 'rap...	https://i.scdn.co/image/8363009cbb612741bfc343...

2 rows × 6 columns

Também existem várias técnicas de imputação avançadas cuja escolha depende da utilização dos dados, por exemplo, depende de um modelo de aprendizado de máquina para inserir e avaliar com precisão os dados ausentes. A imputação múltipla e modelos preditivos podem ser mais precisos, e assim são mais comuns do que métodos mais simples. No entanto, não existe uma maneira ideal de compensar a ausência, pois cada estratégia possui um desempenho melhor ou pior dependendo do conjunto de dados.

Dados ruidosos. São dados que fornecem informações adicionais, mas sem sentido, chamados de ruído. Geralmente são gerados por alguma falha na coleta de dados, erros de entrada de dados, entre outros. Dados com ruído podem prejudicar resultados de análises e de modelos, como os de aprendizado de máquina, por exemplo. Tal problema pode ser solucionado a partir de diferentes abordagens, incluindo o método de Binning, Regressão e algoritmos de agrupamento de dados (*Clustering*) [Igual and Seguí 2017, Skiena 2017].

Para reduzir os efeitos de pequenos erros de observação, utiliza-se o método de Binning, que é uma técnica de suavização de dados. Os dados originais são divididos em segmentos de tamanhos iguais (*bins*) e, em seguida, são substituídos por um valor geral calculado para cada intervalo. Cada segmento é tratado separadamente, onde a substituição de valores pode ser realizada através de valores médios ou limites. No *pandas*, o método Binning usa as funções **qcut()** e **cut()**, que parecem iguais, mas são diferentes.

De acordo com a documentação do *pandas*, **qcut()** é uma função de discretização baseada em quantis: ela procura dividir os dados em *bins* usando percentis com base na distribuição da amostra. A maneira mais simples de usá-la é definir o número de quantis e deixar que o *pandas* descubra como dividir os dados. O exemplo a seguir discretiza a variável **followers** de duas maneiras diferentes: criando cinco *bins* de mesmo tamanho, e configurando três quantis rotulados como “alto”, “médio” e “baixo”.

[8]:
```
# Discretizando em 5 intervalos de tamanhos iguais
df['qcut_1'] = pd.qcut(df['followers'], q=5)
# Discretizando usando três quantis
df['qcut_2'] = pd.qcut(df['followers'],
  q=[0,.3,.7,1], labels=["baixo", "médio", "alto"])
df.head(3)
```

[8]:

	artist_id	name	...	qcut_1	qcut_2
0	4gzpq5DPGxSnKTe4SA8HAU	Coldplay	...	(6020134.2, 77681514.0]	alto
1	6Te49r3A6f5BilgBRxH7FH	Ninho	...	(2373906.0, 6020134.2]	alto
2	4QrBoWLm2WNlPdbFhmlaUZ	KEVVO	...	(134539.2, 790144.4]	baixo

3 rows × 8 columns

Por outro lado, pode-se utilizar a função **cut()** para segmentar e ordenar os dados em *bins*. Enquanto **qcut()** calcula o tamanho de cada *bin*, garantindo que a distribuição dos dados nos compartimentos seja igual, a função **cut()** define bordas exatas dos compartimentos. Ou seja, não há garantia sobre a distribuição de itens em cada *bin*. O exemplo a seguir corta os dados da variável *followers* em quatro *bins* de tamanhos iguais. Desta forma, se você deseja uma distribuição igual dos valores em cada compartimento, use **qcut()**. Caso contrário, se você quiser definir seus próprios intervalos numéricos de categorias, use a função **cut()**.

[9]:
```python
# Discretizando em 4 bins
df['cut_1'] = pd.cut(
    df['followers'],
    bins=4)
df.head(3)
```

[9]:

	artist_id	name	...	qcut_2	cut_1
0	4gzpq5DPGxSnKTe4SA8HAU	Coldplay	...	alto	(19420416.75, 38840782.5]
1	6Te49r3A6f5BilgBRxH7FH	Ninho	...	alto	(-77630.463, 19420416.75]
2	4QrBoWLm2WNlPdbFhmlaUZ	KEVVO	...	baixo	(-77630.463, 19420416.75]

3 rows × 9 columns

Outliers. São amostras de dados notoriamente diferentes da tendência central. São geralmente criados por erros de coleta ou entrada de dados, e podem facilmente produzir valores discrepantes interferindo na qualidade de análises. A maneira mais simples de identificar outliers é observar os valores máximos e mínimos em cada variável para ver se eles estão muito fora da curva normal. O exemplo a seguir utiliza a função **describe()** para gerar estatísticas do conjunto de dados, incluindo os valores máximos e mínimos.

[12]:
```python
pd.set_option('display.max_columns', 10)

df.describe().round().transpose()
```

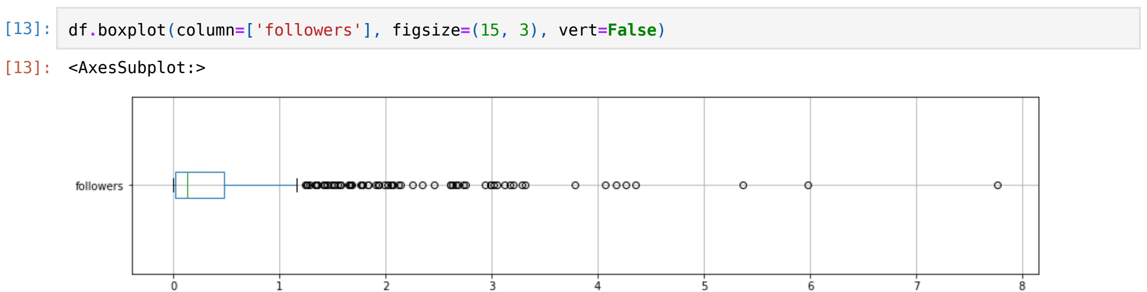
[12]:

	count	mean	std	min	25%	50%	75%	max
followers	625.0	4471046.0	8194928.0	51.0	222726.0	1256547.0	4809711.0	77681514.0
popularity	563.0	78.0	10.0	0.0	72.0	79.0	84.0	100.0

Para o recurso **followers**, o valor máximo é 77,7 milhões de seguidores, enquanto o quartil de 75% é apenas 48 milhões. Portanto, artistas com mais de 77 milhões de seguidores podem ser outliers. Essa verificação geral é melhor realizada através da representação gráfica dos dados numéricos através de seus quartis. Para isso, pode-se utilizar *boxplots*,[4] onde os valores discrepantes são plotados como pontos individuais. O gráfico do exemplo a seguir mostra a distribuição da variável **followers**.

[13]:
```python
df.boxplot(column=['followers'], figsize=(15, 3), vert=False)
```

[13]: <AxesSubplot:>



Esse exemplo ilustra que existem inúmeros pontos (outliers) entre aproximadamente 12 a 78 milhões (observe que o eixo x está em dezenas de milhões). Embora tenha sido fácil detectar tais valores discrepantes, é preciso determinar as soluções adequadas para tratá-los. Assim como no caso de dados ausentes, o tratamento de outliers depende muito do conjunto de dados e do objetivo do projeto. Soluções possíveis incluem manter, ajustar ou apenas remover os dados discrepantes.

Uma técnica comum para a remoção de outliers é o método de Z-score, que considera como outliers e remove valores a uma determinada quantidade de desvios padrões da média. A quantidade desses desvios pode variar conforme o tamanho da amostra. No exemplo a seguir, para identificar e remover os outliers da coluna **followers**, usou-se os z-scores de seus registros com a quantidade de desvios configurada para três. Para a obtenção dos z-scores, foi usado o módulo **stats** da biblioteca *SciPy*.

[4]Descrevemos com detalhes o que são *boxplots* na Seção 1.5

```python
[14]: import numpy as np
      from scipy import stats
      z_scores = stats.zscore(df['followers']) # Z-scores dos valores da coluna 'followers'
      abs_z_scores = np.abs(z_scores) # Obtendo os valores absolutos
      novo_df = df[abs_z_scores < 3] # Filtra por desvios padrões < 3
      print(f'{(len(df) - len(novo_df))} foram outliers removidos')
```

```
18 foram outliers removidos
```

Dados duplicados. Aparecem em muitos contextos, especialmente durante a entrada ou coleta de dados. Por exemplo, ao usar um *web scraper*, ele pode coletar a mesma página web mais de uma vez ou as mesmas informações de duas páginas diferentes. Independente da causa, a duplicação de dados pode levar a conclusões incorretas, onde algumas observações podem ser consideradas mais comuns do que realmente são. O exemplo a seguir mostra quantas linhas estão duplicadas em cada coluna do conjunto de dados.

```python
[15]: # Para cada coluna,
      for coluna in df.columns:
          # Seleciona linhas duplicadas
          duplicatas_df = df[df.duplicated(coluna)]

          print(f"Total de linhas duplicadas "
                f"na {coluna}: {len(duplicatas_df)}")
```

```
Total de linhas duplicadas na artist_id: 0
Total de linhas duplicadas na name: 1
Total de linhas duplicadas na followers: 0
Total de linhas duplicadas na popularity: 569
Total de linhas duplicadas na genres: 136
Total de linhas duplicadas na image_url: 10
Total de linhas duplicadas na qcut_1: 620
Total de linhas duplicadas na qcut_2: 622
Total de linhas duplicadas na cut_1: 621
```

No exemplo, apenas duas colunas do *DataFrame* não possuem duplicatas: **artist_id** e **followers**. Além disso, nota-se que existem duas cópias do nome de um mesmo artista. Essa duplicidade de dados é a categoria mais simples de duplicatas: são cópias exatamente iguais de um mesmo registro. Para resolver, basta identificar os valores idênticos e removê-los. O *pandas* fornece o método **drop_duplicates()** que retorna um novo *DataFrame* com linhas duplicadas removidas, como no exemplo a seguir.

```python
[16]: novo_df = df.drop_duplicates() # Remove as linhas duplicadas
      duplicatas_df = novo_df[novo_df.duplicated()] # calcula o total de linhas duplicadas
      print(f"Total de linhas duplicadas: {len(duplicatas_df)}")
```

```
Total de linhas duplicadas: 0
```

Com apenas uma lista de registros com duplicatas, a melhor e mais simples solução é geralmente a remoção. Porém, com dados tabulares, a melhor solução é remover os dados duplicados considerando os identificadores exclusivos. Por exemplo, **artist_id** é a coluna de identificadores únicos, facilitando verificar se o registro duplo pode ser descartado, conforme o seguinte.

```python
[17]: pd.set_option('display.max_columns', 5)
      # Extrai o nome duplicado
      nome_duplicado = df[df.duplicated(['name'])].name

      # Linhas onde 'name' é igual ao nome duplicado
      df.loc[df['name'].isin(nome_duplicado)]
```

[17]:

	artist_id	name	...	qcut_2	cut_1
88	30vzHJVtI7JWEKqk40eOdz	Niack	...	baixo	(-77630.463, 19420416.75]
249	5uYe4bcAXIMP7d4mrOYkML	Niack	...	baixo	(-77630.463, 19420416.75]

2 rows × 9 columns

Os resultados mostram dois artistas de nome "Niack" com identificadores diferentes; logo, não descartáveis. Existem outras formas complexas, onde mais de um registro é associado à mesma observação, porém seus valores não são completamente idênticos.

Por exemplo, nomes próprios com e sem abreviação ou omissão de algum dos sobrenomes. Essa duplicação parcial é mais difícil de identificar, e uma solução comum é utilizar funções de similaridade de strings.

A biblioteca Python *FuzzyWuzzy* usa a distância de Levenshtein para calcular as diferenças entre duas strings. Essa distância é calculada a partir da contagem de operações de inserção, remoção ou substituição necessárias para transformar uma string em outra. A seguir, as funções **ratio()** e **partial_ratio()** encontram cópias não idênticas de nomes de artistas. O código retorna duas prováveis duplicações parciais: na primeira, os dois nomes identificam duas artistas distintas, não se podendo removê-las; na segunda, "Red Velvet" denomina um grupo feminino sul-coreano, mas o nome "Red Velvet - Irene & Seulgi" foi cadastrado na plataforma Spotify para representar a primeira subunidade do grupo (composto por Irene & Seulgi).

```
[18]: from fuzzywuzzy import fuzz
      from itertools import combinations

      combinacoes = combinations(df.name, 2) # gera todas as combinações
      for nome_1, nome_2 in list(combinacoes): # para cada tupla de nomes,
          partial_ratio = fuzz.partial_ratio(nome_1, nome_2) # similaridade parcial
          ratio = fuzz.ratio(nome_1, nome_2) # similaridade simples

          # Se os nomes forem parcialmente iguais, porém não identicos,
          if partial_ratio == 100 and ratio < 100 and ratio > 50:
              print(f'{nome_1} ({partial_ratio}) | {nome_2} ({ratio})')
```

```
The Blessed Madonna (100) | Madonna (54)
Red Velvet - IRENE & SEULGI (100) | Red Velvet (54)
```

1.4.2. Integração de Dados

Geralmente é necessário extrair dados a partir de várias fontes. A etapa seguinte do pré-processamento é combinar dados dessas diferentes fontes para obter uma estrutura unificada com informações mais significativas e valiosas. Por exemplo, dados básicos das músicas estão na tabela M, e os detalhes de tais músicas na tabela D. Então, uma análise mais robusta sobre as músicas requer reunir todos os dados possíveis de M e D em um único lugar, processo denominado Integração de dados.

O *pandas* permite mesclar e juntar múltiplas tabelas utilizando operações de junção. Para exemplificar, dividimos a tabela *Hits* em dois *DataFrames*, **df1** com título da música, identificador e nome de seus artistas, e **df2** com popularidade e data de lançamento das músicas. Em ambas as tabelas, a coluna **song_id** está presente, para identificar cada uma das músicas, conforme a seguir.

df1	song_id	song_name	artist_id	artist_name
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	['Trevor Daniel']
1	3BYIzNZ3t9lRQCACXSMLrT	Venetia	['4O15NlyKLIASxsJ0PrXPfz']	['Lil Uzi Vert']
2	1g3J9W88hTG173ySZR6E9S	Tilidin Weg	['1aS5tqEs9ci5P9KD9tZWa6']	['Bonez MC']

df2	song_id	popularity	release_date
0	2rRJrJEo19S2J82BDsQ3F7	77	2020-03-26
1	3BYIzNZ3t9lRQCACXSMLrT	66	2020-03-06
2	1g3J9W88hTG173ySZR6E9S	13	2020-07-30

A função **merge()** implementa vários tipos de junção: um-para-um (1:1), um-para-muitos (1:N) e muitos-para-muitos (N:N). O tipo de junção depende da organização dos conjuntos de dados de entrada.

Junções um-para-um (1:1). A junção um-para-um é talvez o tipo mais simples de fusão, muito semelhante à concatenação de colunas. Por exemplo, considere os dois *DataFrames*

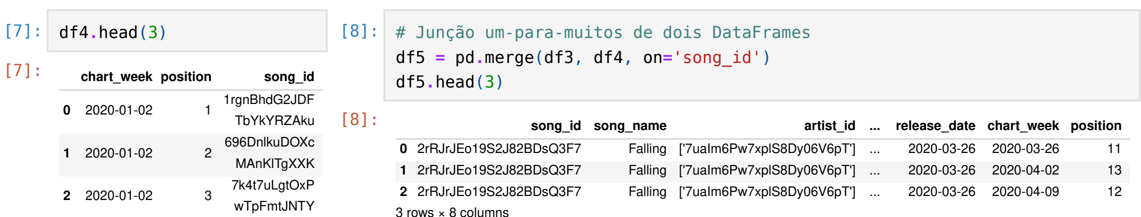
df1 e **df2**. O resultado da seguinte junção é um *DataFrame* que combina as duas entradas, baseado nos valores comuns da coluna **song_id**. Observe que a ordem das entradas em cada coluna não é necessariamente mantida. Ou seja, a ordem da coluna **song_id** difere entre as duas tabelas, e a função **merge()** as considera corretamente.

[5]:
```python
# Junção um-para-um
df3 = pd.merge(
    df1, df2, on='song_id')
df3.head(3)
```

[5]:

	song_id	song_name	artist_id	artist_name	popularity	release_date
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	['Trevor Daniel']	77	2020-03-26
1	3BYIzNZ3t9lRQCACXSMLrT	Venetia	['4O15NlyKLIASxsJ0PrXPfz']	['Lil Uzi Vert']	66	2020-03-06
2	1g3J9W88hTG173ySZR6E9S	Tilidin Weg	['1aS5tqEs9ci5P9KD9tZWa6']	['Bonez MC']	13	2020-07-30

Junções um-para-muitos (1:N). A junção um-para-muitos é usada quando uma das duas colunas-chave contém mais de uma entrada por registro. Nesse caso, o *DataFrame* resultante preserva tais entradas duplicadas conforme apropriado. Por exemplo, considere a junção da tabela resultante do exemplo anterior com os sucessos do Spotify na tabela *Charts*, na qual os identificadores das músicas podem aparecer mais de uma vez. O resultado do comando a seguir é um único *DataFrame* que combina as duas entradas; porém, ao contrário do exemplo anterior, os dados originais (i.e., do *DataFrame* **df3**) se repetem conforme exigido pelas entradas do *DataFrame* **df4**.



[7]:
```python
df4.head(3)
```

[7]:

	chart_week	position	song_id
0	2020-01-02	1	1rgnBhdG2JDFTbYkYRZAku
1	2020-01-02	2	696DnlkuDOXcMAnKlTgXXK
2	2020-01-02	3	7k4t7uLgtOxPwTpFmtJNTY

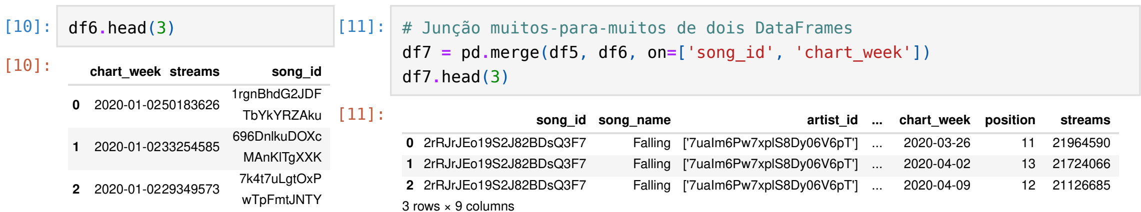
[8]:
```python
# Junção um-para-muitos de dois DataFrames
df5 = pd.merge(df3, df4, on='song_id')
df5.head(3)
```

[8]:

	song_id	song_name	artist_id	...	release_date	chart_week	position
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-03-26	2020-03-26	11
1	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-03-26	2020-04-02	13
2	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-03-26	2020-04-09	12

3 rows × 8 columns

Junções muitos-para-muitos (N:N). Na junção muitos-para-muitos, se a coluna-chave das duas tabelas incluir entradas duplicadas, o resultado é uma fusão muitos-para-muitos. Por exemplo, para criar um relacionamento muitos-para-muitos entre a tabela anterior e outra contendo o número total de streams que cada música atingiu nas semanas dos charts. A coluna-chave do *DataFrame* resultante é a combinação das colunas-chave de cada tabela. Agora, as duas tabelas terão duas colunas-chave: **song_id** e **chart_week**. Após a junção muitos-para-muitos, a coluna-chave do *DataFrame* resultante é expressa pela combinação dos valores de **song_id** e **chart_week** das tabelas **df5** e **df6**.



[10]:
```python
df6.head(3)
```

[10]:

	chart_week	streams	song_id
0	2020-01-02	50183626	1rgnBhdG2JDFTbYkYRZAku
1	2020-01-02	33254585	696DnlkuDOXcMAnKlTgXXK
2	2020-01-02	29349573	7k4t7uLgtOxPwTpFmtJNTY

[11]:
```python
# Junção muitos-para-muitos de dois DataFrames
df7 = pd.merge(df5, df6, on=['song_id', 'chart_week'])
df7.head(3)
```

[11]:

	song_id	song_name	artist_id	...	chart_week	position	streams
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-03-26	11	21964590
1	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-04-02	13	21724066
2	2rRJrJEo19S2J82BDsQ3F7	Falling	['7ualm6Pw7xplS8Dy06V6pT']	...	2020-04-09	12	21126685

3 rows × 9 columns

1.4.3. Transformação de Dados

Em geral, após a Integração de dados, colunas com diferentes tipos de dados podem ser geradas. Na Transformação de dados, o principal objetivo é transformar tais dados de diferentes formatos em um formato suportado pelo processo de pesquisa, e.g., modelos e algoritmos. A Transformação de dados possui etapas em comum com a Limpeza de

dados, incluindo técnicas de remoção de ruído e discretização de dados. Assim, a seguir, apresentamos as demais etapas envolvidas nesta fase de pré-processamento.

Dados não padronizados. Dados coletados de diferentes fontes podem reunir dados heterogêneos, não padronizados e em diferentes escalas. O reescalonamento de dados não é uma etapa obrigatória, mas uma boa prática, pois atributos em um padrão auxiliam o desempenho de modelos, algoritmos e afins. Existem duas técnicas de reescalonamento que transformam características para uma escala, magnitude ou intervalo. A seguir, descrevemos e exemplificamos cada uma delas utilizando o conjunto de atributos numéricos que descrevem as músicas do Spotify.

Normalização de dados reescalona atributos numéricos em um intervalo de 0 a 1, alterando os valores das colunas numéricas, sem distorcer as diferenças nos intervalos de valores. No exemplo a seguir, subtrai-se o valor mínimo de cada coluna e divide-se pelo intervalo, aplicando e escala *min-max* do *pandas* através dos métodos **min()** e **max()**.

[3]:
```
df_norm = df.copy()  # cópia do DataFrame
for coluna in df_norm.columns:  # Para cada coluna,
    df_norm[coluna] = (  # Normalização min-max
        df_norm[coluna] - df_norm[coluna].min()
    ) / (df_norm[coluna].max() - df_norm[coluna].min())
df_norm.head(3)
```

[3]:

	popularity	track_ number	num_ artists	...	speechiness	valence	tempo
0	0.793814	0.310345	0.0	...	0.015335	0.212314	0.506887
1	0.680412	0.448276	0.0	...	0.176348	0.558386	0.606828
2	0.134021	0.000000	0.0	...	0.243727	0.143312	0.393380

3 rows × 17 columns

Padronização de dados reescalona a distribuição de cada atributo para média igual a zero e desvio padrão igual a um. Cada valor padronizado é calculado subtraindo a média do recurso correspondente e dividindo pelo desvio padrão. O exemplo a seguir transforma os atributos do *DataFrame*, subtraindo a média de cada coluna e dividindo pelo desvio padrão, usando os métodos **mean()** e **std()** do *pandas*.

[4]:
```
df_padron = df.copy()  # cópia do DataFrame
for coluna in df_padron.columns:  # Para cada coluna,
    df_padron[coluna] = (  # Padronização
        df_padron[coluna] - df_padron[coluna].mean()
    ) / (df_padron[coluna].std())
df_padron.head(3)
```

[4]:

	popularity	track_ number	num_ artists	...	speechiness	valence	tempo
0	0.448125	1.007112	-0.644346	...	-0.802165	-1.217354	0.122480
1	-0.164943	1.806576	-0.644346	...	0.462704	0.221431	0.658761
2	-3.118819	-0.791681	-0.644346	...	0.992015	-1.504228	-0.486598

3 rows × 17 columns

Engenharia de Características. (*Feature Engineering*) Consiste em criar ou remover características em um conjunto de dados para melhorar o desempenho de modelos de Aprendizado de Máquina, por exemplo. Muitas técnicas de Transformação e Limpeza de dados também são *Feature Engineering*, incluindo métodos de reescalonamento, discretização e redução de dimensionalidade. Então, a seguir, discutimos duas técnicas ainda não abordadas neste capítulo.

Dados categóricos são variáveis que só podem assumir um número limitado e, geralmente, fixo de valores possíveis. Por exemplo, as músicas do Spotify contêm dados categóricos para informar se as músicas são explícitas. Ou seja, cada unidade de observação (i.e., música) é atribuída a um determinado grupo ou categoria nominal. Muitos algoritmos e modelos comuns em Ciência de Dados (e.g., aprendizado de máquina) têm dificuldades em processar dados categóricos, necessitando alguma forma de conversão para valores numéricos. Uma das abordagens mais simples e comuns é converter variáveis categóricas em *dummies* ou indicadores, utilizando a função **get_dummies()** do *pandas*, conforme o exemplo a seguir.

```
[7]: # Convertendo em dummies
df_dummies = pd.get_dummies(df)
df_dummies.head()
```

[7]:

	explicit_False	explicit_True	song_type_Collaboration	song_type_Solo
0	1	0	0	1
1	0	1	0	1
2	1	0	0	1
3	1	0	0	1
4	0	1	1	0

```
[8]: df_dummies.dtypes
```

```
[8]: explicit_False             uint8
explicit_True              uint8
song_type_Collaboration    uint8
song_type_Solo             uint8
dtype: object
```

Variáveis de data são um tipo especial de dados categóricos. Embora uma data forneça apenas um momento específico no calendário, quando pré-processada corretamente, ela pode enriquecer muito o conjunto de dados. Os exemplos a seguir convertem datas em formatos amigáveis, extraindo recursos e criando novas variáveis que podem ser usadas na análise de um modelo. Especificamente, extraímos recursos da data de lançamento das músicas do Spotify, utilizando métodos do *pandas*.

```
[10]: # Convertendo a data de lançamento em 'datetime'
data['release_date'] = pd.to_datetime(
    data['release_date'])
data.dtypes
```

```
[10]: song_id                 object
song_name               object
release_date    datetime64[ns]
dtype: object
```

```
[11]: data['day'] = data['release_date'].dt.day  # dia
data['month'] = data['release_date'].dt.month  # mês
data['year'] = data['release_date'].dt.year  # ano
data.head(3)
```

[11]:

	song_id	song_name	release_date	day	month	year
0	2rRJrJEo19S2J82BDsQ3F7	Falling	2020-03-26	26	3	2020
1	3BYIzNZ3t9lRQCACXSMLrT	Venetia	2020-03-06	6	3	2020
2	1g3J9W88hTG173ySZR6E9S	Tilidin Weg	2020-07-30	30	7	2020

Dados desbalanceados. São uma ocorrência comum em domínios reais, especialmente em modelos de classificação. Dados estão desbalanceados quando existe uma desproporção de observações de cada classe. Tal desbalanceamento não gera um erro imediato ao construir e executar um modelo. Porém, os resultados podem ser ilusórios, dado que a maioria das técnicas de Aprendizado de Máquina, por exemplo, funcionam melhor quando o número de amostras em cada classe é equilibrado. O exemplo a seguir usa características acústicas das músicas do Spotify para prever se uma música é uma colaboração ou não. Aqui, modelamos o problema como uma classificação binária: cada música recebe o valor 1 se for uma colaboração, ou 0 caso contrário.

```
[13]: data.head(3)
```

[13]:

	song_type	duration_ms	key	...	speechiness	valence	tempo
0	Solo	159381	10	...	0.0364	0.236	127.087
1	Solo	188800	9	...	0.1750	0.562	142.933
2	Solo	180950	10	...	0.2330	0.171	109.090

3 rows × 14 columns

```
[14]: data['song_type'] = [int(x == 'Collaboration') for x in data.song_type] # Mapeia 'song_type' em 0 e 1
data['song_type'].value_counts() # qtd de cada classe
```

```
[14]: 0    752
1    532
Name: song_type, dtype: int64
```

A função **value_counts()** computa os valores únicos de cada classe. Como resultado, apenas cerca de 41,4% das observações (i.e., 532 músicas) são colaborações. Portanto, ao prever a classe 0, músicas solo, obtém-se precisão de 58,6%. A seguir, testamos esse cenário criando um modelo linear de regressão logística, usando o módulo **sklearn.linear_model** e a classe **LogisticRegression**.

```
[15]: import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
y = data.song_type # treino (X)
X = data.drop('song_type', axis=1) # teste (y)
logr = LogisticRegression().fit(X, y) # treinando o modelo de regressão logística
pred_y_0 = logr.predict(X)
print(accuracy_score(pred_y_0, y)) # acurácia do modelo
print(np.unique(pred_y_0)) # classes previstas pelo modelo
```

```
0.5856697819314641
[0]
```

O resultado anterior indica que o modelo apresentou uma precisão geral de 58,6%; ou seja, como consequência dos dados desbalanceados, o modelo prevê apenas uma classe. Em outras palavras, ele ignora a classe minoritária (i.e., colaborações) em favor da classe majoritária (i.e., músicas solo). A seguir, aplicamos a técnica de *Downsampling*, que remove aleatoriamente observações da classe majoritária para evitar que seu sinal domine o algoritmo de aprendizagem: (1) separa-se as observações de cada classe em diferentes *DataFrames*; (2) realiza-se nova amostragem da classe majoritária sem substituição, definindo o número de amostras para corresponder ao da classe minoritária; e (3) concatena-se o *DataFrame* da classe minoritária com o *DataFrame* resultante.

```
[16]: from sklearn.utils import resample
df_majority = data[data.song_type == 0] # classe majoritária
df_minority = data[data.song_type == 1] # classe minoritária
df_majority_downsampled = resample(
    df_majority, replace=False,  # amostra sem substituição
    n_samples=532,  # para corresponder à classe minoritária
    random_state=123)  # garantindo reprodutibilidade
df_downsampled = pd.concat([df_majority_downsampled, df_minority])
df_downsampled.song_type.value_counts()
```

```
[16]: 1    532
0    532
Name: song_type, dtype: int64
```

Apesar de o novo *DataFrame* ter menos observações do que o original, a proporção das duas classes está balanceada. Pode-se, então, treinar novamente o modelo de regressão logística. Conforme o exemplo a seguir, após o balanceamento, o modelo prevê duas classes com menor precisão, mas com métrica de avaliação mais significativa.

```
[17]: y = df_downsampled.song_type # treino (X)
X = df_downsampled.drop('song_type', axis=1) # teste (y)
logr = LogisticRegression().fit(X, y) # Treinando o modelo de regressão logística
pred_y_1 = logr.predict(X)
print(np.unique(pred_y_1)) # acurácia do modelo
print(accuracy_score(y, pred_y_1)) #classes previstas pelo modelo
```

```
[0 1]
0.49154135338345867
```

1.4.4. Redução de Dados

Gerenciar e processar dados requer tempo, esforço e recursos, especialmente com grandes volumes de dados de várias fontes e em diferentes formatos. Para enfrentar tais desafios, técnicas de Redução de dados são aplicadas, auxiliando na análise de dados com alta dimensionalidade. Embora essenciais, essas técnicas geralmente são complexas, pois exigem amplo conhecimento para a escolha adequada de qual técnica utilizar. Assim, por simplicidade, essa seção apresenta apenas uma técnica, citando o texto [Igual and Seguí 2017] para as demais.

A Análise de Componentes Principais (PCA) é um dos métodos mais simples e, de longe, o mais comum para a redução da dimensionalidade. PCA é um algoritmo não supervisionado[5] que cria combinações lineares dos atributos originais, classificadas em ordem de sua variância explicada. O próximo exemplo utiliza a biblioteca *scikit-learn* para importar o módulo **sklearn.decomposition** e a classe **PCA** para extrair os dois componentes principais (**n_components = 2**) do nosso conjunto de dados.

[5]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
# Padronizando os dados de treino
X = StandardScaler().fit_transform(X)
# Calculando os dois componentes principais
pca_resultado = PCA(n_components=2)
df_pcs = pd.DataFrame(pca_resultado.fit_transform(X), columns=['PC1', 'PC2'])
df_pcs.tail()
```

[5]:

	PC1	PC2
1279	-1.850735	-1.341044
1280	5.250560	1.087556
1281	0.449023	-1.070109
1282	-0.780147	0.100485
1283	-2.275789	0.858764

O *DataFrame* resultante apresenta os valores dos dois componentes principais para todas as 1283 amostras. O conjunto de dados foi padronizado, utilizando a classe **StandardScaler**; do contrário, os atributos em maior escala dominariam os novos componentes principais. Após a extração dos componentes principais, o método fornece a quantidade de informações ou variação que cada componente principal mantém após projetar os dados em um subespaço de dimensão inferior. O primeiro componente principal detém 17,3% das informações, enquanto o segundo apenas 9% das informações. Ou seja, ao reduzir a dimensionalidade do conjunto de dados para duas dimensões, 73,7% das informações originais foram perdidas, conforme mostrado a seguir.

[6]:
```
print('Variação explicada por componentes principais: {}'.format(
    pca_resultado.explained_variance_ratio_))
```

```
Variação explicada por componentes principais: [0.17349178 0.08857436]
```

1.5. Ciência de Dados Básica

Após o pré-processamento dos dados descrito na seção anterior, a sequência natural é realizar a Análise Exploratória dos Dados (*Exploratory Data Analysis* – EDA). Esta etapa consiste em entender melhor um conjunto de dados através de estatísticas descritivas e de gráficos visuais, sendo comum em Ciência de Dados para levantar hipóteses e responder questões sobre os dados em si. Apesar de relativamente simples, essa análise é importante e pode ser demorada, pois é conectada com a etapa de pré-processamento, podendo exigir reexecução de passos para assegurar melhor qualidade de dados.

Uma das principais vantagens de utilizar o Jupyter em projetos de Ciência de Dados é justamente a sua estrutura em células. Tal vantagem é evidenciada nas etapas de análise exploratória e visualização de dados (Seção 1.6), pois a fácil visualização em formato tabular ou gráfico permite a cientistas uma rápida análise descritiva de dados. Assim, a depender dos resultados, é possível corrigir, descartar ou gerenciar os dados de forma apropriada. Além disso, não existe uma única sequência de passos para descrever e lidar com dados. Cabe a cientistas de dados escolher as melhores técnicas para a tarefa de análise, a partir de um entendimento prévio e do arcabouço prático disponível.

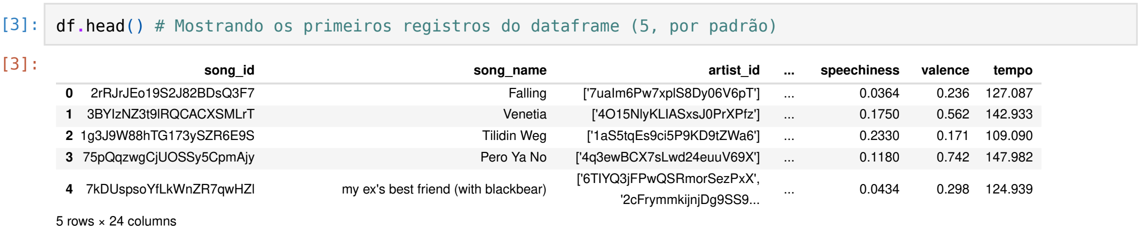
[5] Algoritmos não-supervisionados visam extrair padrões de dados não-rotulados (e.g., definir grupos de instâncias a partir de características comuns). Para mais detalhes, ver [Igual and Seguí 2017, Skiena 2017].

Mesmo sem sequência fixa para análise exploratória, esta seção apresenta os passos comumente utilizados. Os principais tópicos abordados incluem a exploração e descrição inicial de um *DataFrame*, estatísticas básicas para variáveis numéricas e categóricas, além da análise de distribuições e de correlação de valores. Aqui, é utilizada a tabela *Tracks* do conjunto de dados da Seção 1.2, que contém informações relevantes sobre as canções que entraram na parada global de sucesso do Spotify no ano de 2020.

1.5.1. Exploração Inicial

A etapa de exploração inicial é simples, mas importante para cientistas se familiarizarem com o conteúdo de seus dados. A seção anterior focou no pré-processamento de dados para deixá-los prontos para as primeiras análises, conforme descrito a seguir.

Amostras de Dados. Uma das primeiras ações sobre um novo conjunto de dados é verificar uma amostra de suas instâncias, para entender o conteúdo das colunas e o tipo de informação em cada uma delas. O *pandas* possui funções que permitem visualizar tais amostras. Utilizá-las em um ambiente Jupyter facilita ainda mais a compreensão, pois permite uma visualização simples e amigável do *DataFrame* ao final de cada célula. Um exemplo é a função **head()** para visualizar as primeiras linhas de um *DataFrame*.



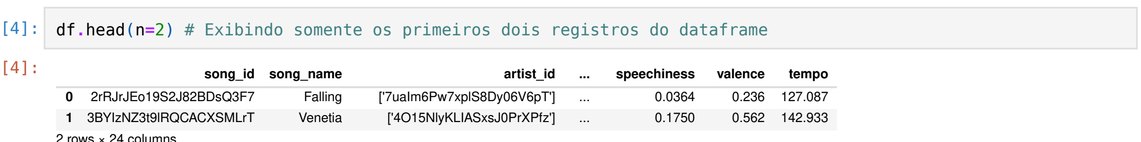
```
[3]: df.head() # Mostrando os primeiros registros do dataframe (5, por padrão)
```

[3]:

	song_id	song_name	artist_id	...	speechiness	valence	tempo
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7uaIm6Pw7xplS8Dy06V6pT']	...	0.0364	0.236	127.087
1	3BYIzNZ3t9lRQCACXSMLrT	Venetia	['4O15NlyKLIASxsJ0PrXPfz']	...	0.1750	0.562	142.933
2	1g3J9W88hTG173ySZR6E9S	Tilidin Weg	['1aS5tqEs9ci5P9KD9tZWa6']	...	0.2330	0.171	109.090
3	75pQqzwgCjUOSSy5CpmAjy	Pero Ya No	['4q3ewBCX7sLwd24euuV69X']	...	0.1180	0.742	147.982
4	7kDUspsoYfLkWnZR7qwHZl	my ex's best friend (with blackbear)	['6TIYQ3jFPwQSRmorSezPxX', '2cFrymmkijnjDg9SS9...	...	0.0434	0.298	124.939

5 rows × 24 columns

Por padrão, **head()** exibe as cinco primeiras linhas do *DataFrame*, mas também permite usar o parâmetro **n** para personalizar a saída. O próximo exemplo mostra somente os dois primeiros registros. Esta função limita apenas as linhas, e então todas as colunas do *DataFrame* são mostradas no resultado.



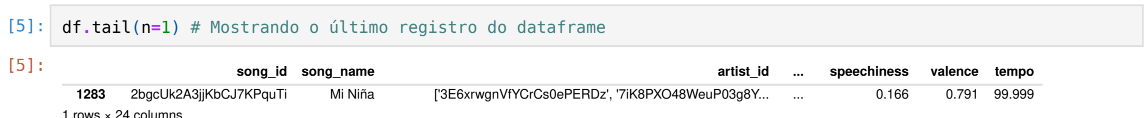
```
[4]: df.head(n=2) # Exibindo somente os primeiros dois registros do dataframe
```

[4]:

	song_id	song_name	artist_id	...	speechiness	valence	tempo
0	2rRJrJEo19S2J82BDsQ3F7	Falling	['7uaIm6Pw7xplS8Dy06V6pT']	...	0.0364	0.236	127.087
1	3BYIzNZ3t9lRQCACXSMLrT	Venetia	['4O15NlyKLIASxsJ0PrXPfz']	...	0.1750	0.562	142.933

2 rows × 24 columns

De forma similar, o *pandas* também oferece a função **tail()**, que exibe as últimas linhas do *DataFrame*. Esta função também pode ter a saída personalizada com o parâmetro **n**, e, portanto, é uma importante ferramenta na inspeção manual dos registros.



```
[5]: df.tail(n=1) # Mostrando o último registro do dataframe
```

[5]:

	song_id	song_name	artist_id	...	speechiness	valence	tempo
1283	2bgcUk2A3jjKbCJ7KPquTi	Mi Niña	['3E6xrwgnVfYCrCs0ePERDz', '7iK8PXO48WeuP03g8Y...	...	0.166	0.791	99.999

1 rows × 24 columns

Entretanto, tais funções só permitem analisar as extremidades do *DataFrame*, e uma inspeção mais aprofundada sobre os outros registros pode ser necessária para um melhor entendimento dos dados. Dessa forma, o *pandas* também oferece a função **sample()**, que retorna de fato uma amostra aleatória dos registros do *DataFrame*. Esta função

possui mais parâmetros disponíveis do que as anteriores, o que oferece um maior poder de análise para cientistas. O próximo exemplo usa os parâmetros **n**, para a quantidade de linhas amostradas, e **random_state**, que é uma semente de um gerador de números aleatórios que mantém o resultado caso a célula seja executada várias vezes.

```
[6]: df.sample(n=3, random_state=7) # Mostrando uma amostra aleatória de n=3 linhas do DataFrame
```

[6]:

	song_id	song_name	artist_id	...	speechiness	valence	tempo
1245	0PQsrLxPbOBBwwmXCnGvcF	Diamonds (with Normani)	['181bsRPaVXVlUKXrxwZfHK', '2cWZOOzeOm4WmBJRnD...	...	0.0873	0.488	94.012
992	1bRpSCFv6P2OUhciByeRYR	Jangueo	['2DspEsT7UXGKd2VaaedgG4', '11YLRSsZA3YVuQQtHX...	...	0.0536	0.723	103.994
1045	4r9jkMEnArtWGH2rL2FZl0	A Tu Merced	['4q3ewBCX7sLwd24euuV69X']	...	0.0568	0.887	92.023

3 rows × 24 columns

Dimensões e tipos de variáveis. Além de amostras, conhecer informações como as dimensões e os tipos das colunas de um *DataFrame* é igualmente útil. Para saber as dimensões de um *DataFrame* (ou qualquer objeto *pandas*, como uma *Series*), basta acessar o atributo **shape**. Para um *DataFrame*, este atributo contém uma tupla com dois valores, informando o número de linhas e colunas, respectivamente, como a seguir.

```
[7]: print('Dimensões do DataFrame completo: ', df.shape)
     print('Dimensões da coluna song_id, que é uma Series: ', df['song_id'].shape)

Dimensões do DataFrame completo:  (1284, 24)
Dimensões da coluna song_id, que é uma Series:  (1284,)
```

O *pandas* possui outras funções com informações gerais sobre um *DataFrame*. No próximo exemplo, a função **info()** retorna, para cada coluna, a quantidade de valores não-nulos, a existência desses valores, bem como o tipo de dado armazenado naquela coluna (*dtype*). O tipo de dados é essencialmente a construção interna que a linguagem usa para entender como armazenar e manipular os dados. A Tabela 1.2 apresenta a lista de todos os tipos de dados suportados pelo *pandas*.

```
[8]: df.info()  # Verificando tipos das variáveis

RangeIndex: 1284 entries, 0 to 1283
Data columns (total 24 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   song_id                1284 non-null   object
 1   song_name              1284 non-null   object
 2   artist_id              1284 non-null   object
 3   artist_name            1284 non-null   object
 4   popularity             1284 non-null   int64
 5   explicit               1284 non-null   bool
 6   song_type              1284 non-null   object
 7   track_number           1284 non-null   int64
 8   num_artists            1284 non-null   int64
 9   num_available_markets  1284 non-null   int64
 10  release_date           1284 non-null   object
 11  duration_ms            1284 non-null   int64

 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 12  key                    1284 non-null   int64
 13  mode                   1284 non-null   int64
 14  time_signature         1284 non-null   int64
 15  acousticness           1284 non-null   float64
 16  danceability           1284 non-null   float64
 17  energy                 1284 non-null   float64
 18  instrumentalness       1284 non-null   float64
 19  liveness               1284 non-null   float64
 20  loudness               1284 non-null   float64
 21  speechiness            1284 non-null   float64
 22  valence                1284 non-null   float64
 23  tempo                  1284 non-null   float64
dtypes: bool(1), float64(9), int64(8), object(6)
memory usage: 232.1+ KB
```

Conhecer os tipos de dados é crucial em projetos de Ciência de Dados. Se o *pandas* acusa um tipo string onde se espera um valor numérico, tal coluna pode conter ruído. Assim, é necessário voltar à etapa de pré-processamento e realizar a remoção de ruídos antes de prosseguir para análises mais complexas (ver Seção 1.4).

Tabela 1.2. Lista dos tipos de dados (dtypes) suportados pelo *pandas* e sua descrição.

Tipo	Descrição
object	*String* ou uma mistura de valores numéricos e não-numéricos
int64	Valores inteiros
float64	Valores com ponto flutuante
bool	Valores booleanos (**True**/**False**)
datetime64	Valores com data e hora
timedelta[ns]	Diferença entre valores **datetime64**
category	Lista finita de categorias (*string*)

1.5.2. Estatísticas Básicas para Dados Numéricos

Em geral, variáveis de conjuntos de dados podem ser classificadas como *categóricas* ou *numéricas/quantitativas*. Dados categóricos são aqueles obtidos a partir de observações nominais (i.e., classes), enquanto as variáveis numéricas são derivadas de medições quantitativas em uma escala numérica. Por exemplo, a cor de uma régua é um atributo categórico, e seu valor sempre será obtido de uma lista de categorias bem definidas (i.e., cores). Já o seu comprimento é um atributo numérico, medido com valores inteiros. Para cada tipo, existem um conjunto de estatísticas descritivas que permitem sumarizar essas informações de uma forma mais legível e próxima para as pessoas analistas.

Para variáveis numéricas, a análise exploratória pode considerar métodos de agregação sobre a população (i.e., conjunto de todas as instâncias do conjunto de dados). Exemplos incluem média, mediana, desvio padrão, soma, valores máximos e mínimos. Todas estão disponíveis de forma simples e prática pelo *pandas*, e cientistas não precisam gastar tempo ou linhas de código para obtê-las. O próximo exemplo mostra a média[6] de todas as variáveis numéricas de um *DataFrame* com um único comando.

```
[9]: df.mean() # Exibindo a média das variáveis numéricas
```

```
popularity                 68.959502     danceability           0.697962
explicit                    0.445483     energy                 0.627272
track_number                4.961059     instrumentalness       0.011885
num_artists                 1.598910     liveness               0.180263
num_available_markets     157.330997     loudness              -6.415536
duration_ms            196804.523364     speechiness            0.124298
key                         5.331776     valence                0.511828
mode                        0.564642     tempo                123.467972
time_signature              3.966511     dtype: float64
acousticness                0.243906
```

Nesse exemplo, todas as variáveis exibidas são do tipo **int64** ou **float64**, com a exceção de **explicit** (informa se a letra de uma música contém termos explícitos), cujo tipo é **bool**. Neste caso, o próprio *pandas* converte esses valores booleanos para inteiros (i.e., 0 para **False** e 1 para **True**), permitindo então o cálculo da média. Assim como a média, também existem funções no *pandas* para outros métodos de agregação para variáveis numéricas. A Tabela 1.3 apresenta as principais delas acompanhadas de sua descrição.

As funções de agregação do *pandas* também podem ser aplicadas a colunas individuais de um *DataFrame*, isto é, objetos do tipo *Series*. A seguir, são exibidas algumas

[6] Aqui média é com o comando **mean**, mas representa o que em outros softwares é calculado com **average**. Não confundir com a mediana, cujo comando em Python é **median**.

Tabela 1.3. Principais funções de agregação disponibilizadas pelo *pandas*.

Função	Descrição	Função	Descrição	Função	Descrição
count	Número de observações	**max**	Valor máximo	**sem**	Erro padrão da média
sum	Soma de valores	**mode**	Moda	**skew**	Assimetria (*Skewness*)
mean	Média dos valores	**abs**	Valor absoluto	**kurt**	Curtose (*Kurtosis*)
mad	Desvio absoluto médio	**prod**	Produto de valores	**quantile**	Quantis (valor em %)
median	Valor mediano	**std**	Desvio padrão	**cumsum**	Soma cumulativa
min	Valor mínimo	**var**	Variância	**cumprod**	Produto cumulativo

características da duração das músicas, medidas em milissegundos na coluna **duration_ms**.

[10]:
```
print('Registros não-nulos:', df['duration_ms'].count())
print('Valor máximo:', df['duration_ms'].max())
print('Valor mínimo:', df['duration_ms'].min())
print('Média:', df['duration_ms'].mean())
print('Mediana:', df['duration_ms'].median())
print('Desvio padrão:', df['duration_ms'].std())
print('Variância:', df['duration_ms'].var())
```

```
Registros não-nulos: 1284
Valor máximo: 484146
Valor mínimo: 30133
Média: 196804.52336448597
Mediana: 193683.0
Desvio padrão: 44185.523479052776
Variância: 1952360485.1179242
```

Percentis. O *pandas* também oferece a funcionalidade de calcular percentis, que são pontos estabelecidos em uma função de distribuição de probabilidade de uma variável aleatória. A função **quantile()** permite o cálculo de um ou mais percentis a partir do parâmetro 'q', que aceita valores entre 0 e 1 (o 50° percentil, com $q = 0,5$ é a mediana). A seguir, o 10° percentil da variável **duration_ms** é calculado, ou seja, o valor abaixo do qual se encontram 10% das instâncias dessa variável (ordenadas da menor para a maior). O comando seguinte mostra como efetuar o cálculo de percentis para todo o *DataFrame*.

[11]:
```
# Cálculo do 10º percentil
#   (q=0.1) de duration_ms
df['duration_ms'].quantile(
    q=0.1
)
```

[11]: 150759.9

[12]:
```
# 10º e 90º percentis para variáveis numéricas
df.quantile(q=[0.1, 0.9])
```

[12]:

	popularity	explicit	track_number	...	speechiness	valence	tempo
0.1	54.0	0.0	1.0	...	0.0346	0.2076	86.3606
0.9	85.0	1.0	13.0	...	0.2900	0.8227	167.9285

2 rows × 18 columns

Por fim, o *pandas* oferece uma função que combina os principais agregadores utilizados nesta seção. A função **describe()** permite verificar, para as colunas numéricas do *DataFrame*, a quantidade de valores não-nulos (**count**), a média, desvio padrão, valores mínimos, máximo, bem como os percentis 25°, 50° (mediana) e 75°.

[13]:
```
df.describe() # Principais informações descritivas para o DataFrame
```

[13]:

	popularity	track_number	num_artists	...	speechiness	valence	tempo
count	1284.000000	1284.000000	1284.000000	...	1284.000000	1284.000000	1284.000000
mean	68.959502	4.961059	1.598910	...	0.124298	0.511828	123.467972
std	17.942532	5.003355	0.929485	...	0.109577	0.226580	29.547921
min	0.000000	1.000000	1.000000	...	0.023200	0.036000	46.718000
25%	65.000000	1.000000	1.000000	...	0.045800	0.341000	99.028500
50%	73.000000	3.000000	1.000000	...	0.078550	0.508500	122.066000
75%	80.000000	8.000000	2.000000	...	0.173250	0.680250	144.843500
max	97.000000	30.000000	9.000000	...	0.884000	0.978000	205.272000

8 rows × 17 columns

1.5.3. Estatísticas Básicas para Dados Categóricos

Para dados categóricos, uma das principais estratégias descritivas é a contagem de instâncias pertencentes a cada categoria. Assim, cientistas de dados podem entender melhor a

frequência de ocorrência de cada classe em um determinado conjunto de dados. O *pandas* oferece a função **value_counts**, que retorna as classes existentes ordenadas pela quantidade de elementos. Essa função tem um parâmetro *booleano* **normalize** que, caso verdadeiro, retorna a frequência (porcentagem) de ocorrência de cada classe. Seguindo as análises sobre o *DataFrame* com os dados de músicas, a coluna **song_type** informa o tipo de música com relação à quantidade de artistas (solo ou colaboração). O exemplo a seguir aplica **value_counts** nessa coluna com o parâmetro **normalize**.

```
[14]: # Contagem de valores para o tipo de música
      df['song_type'].value_counts()
```

```
[14]: Solo             752
      Collaboration    532
      Name: song_type, dtype: int64
```

```
[15]: # Frequência dos tipos de música
      df['song_type'].value_counts(normalize=True)
```

```
[15]: Solo             0.58567
      Collaboration    0.41433
      Name: song_type, dtype: float64
```

Assim, descobre-se que mais da metade das músicas no conjunto de dados são canções solo (752 ou 58,6%), enquanto as colaborações representam aproximadamente 41,4% (532 canções). Esse resultado pode ser exibido de uma forma mais legível apenas adicionando uma função à contagem de valores. De forma gráfica, utilizando o comando **.plot.barh()**, transformamos a contagem/frequência em um gráfico de barras horizontal simples, conforme exemplo a seguir. Aqui, o parâmetro **figsize** foi usado para legibilidade.

```
[16]: # Frequência em um gráfico de barras
      (
      df['song_type'].value_counts(normalize=True)
      .plot.barh(figsize=(7, 1.25))
      )
```

```
[16]: <AxesSubplot:>
```


1.5.4. Distribuições

Estatísticas básicas para variáveis numéricas como média, mediana e variância são um bom ponto de partida para compreender conjuntos de dados. No entanto, elas não são suficientes para descrever totalmente esses dados, pois dados completamente diferentes podem ter a mesma média, por exemplo. Dessa forma, é necessário aprofundar a análise olhando para a distribuição, que de forma geral apresenta a frequência com que os valores aparecem no conjunto de dados. Nesta seção, são apresentadas duas das principais formas de se visualizar distribuições de valores: *boxplots* e histogramas.

***Boxplots*.** Também conhecidos como *diagramas de caixa*, são uma forma padronizada e visual de sumarizar a distribuição de uma variável através de seus quartis: primeiro (Q1), segundo (Q2, mediana) e terceiro (Q3). Além deles, um *boxplot* também exibe outras informações importantes, como o intervalo interquartil (IQR) e *outliers*. As principais definições necessárias para entender um *boxplot* são as seguintes:

- *Mediana* (Q2, 50º percentil): o valor “do meio” do conjunto de dados;
- *Primeiro quartil* (Q1, 25º percentil): valor abaixo do qual se encontram os primeiros 25% dos valores do conjunto de dados (ordenados do menor para o maior);
- *Terceiro quartil* (Q3, 75º percentil): valor abaixo do qual se encontram os primeiros 75% dos valores do conjunto de dados (ordenados do menor para o maior); e
- *Intervalo interquartil* (IQR): intervalo entre o 25º e o 75º percentis.

Então, um *boxplot* para uma variável consiste em uma caixa cujas extremidades vão do primeiro ao terceiro quartil (Q1 a Q3), com uma linha indicando a mediana (Q2). Além disso, este tipo de gráfico contém linhas (“bigodes”) que saem das extremidades da caixa e possuem o comprimento de $1,5 * IQR$. Todos os valores acima ou abaixo dos “bigodes” são classificados como *outliers*, e, portanto, são visualizados como círculos.

O comando **.plot.box()** exibe um *boxplot* para uma ou mais colunas de um *DataFrame* (ou uma *Series*). O exemplo a seguir mostra *boxplots* para as colunas **danceability** e **acousticness**, features do Spotify para a probabilidade de uma música ser dançante e acústica, respectivamente. Assim como **figsize**, o parâmetro **vert** foi usado para legibilidade.

```
[18]: # Distribuição das colunas
      #  'danceability' e 'acousticness'

      df[[
          'danceability',
          'acousticness'
      ]].plot.box(vert=False, figsize=(7, 1.65))
```

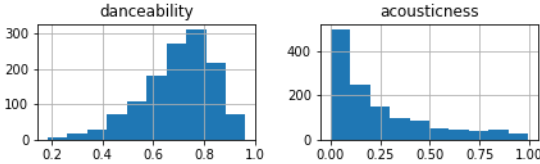
[18]: <AxesSubplot:>

Com esse exemplo, percebe-se que os valores de **acousticness** estão em sua maioria abaixo de 0,5, e poucos outliers com valores altos (círculos fora dos quartis). Além disso, a maioria dos valores de **danceability** se encontra em um patamar acima de 0,5, ou seja, a maioria das músicas do nosso conjunto de dados possui um alto valor para essa feature e, portanto, são bastante dançantes. Também são poucas as músicas com valores baixos para essa métrica (outliers). Como nosso conjunto de dados compreende músicas que entraram nas paradas de sucesso do Spotify em 2020, pode-se afirmar que a maioria dos hits são dançantes e não-acústicos (geralmente gravadas em estúdio).

Histogramas. Em linhas gerais, um histograma é uma representação da distribuição de dados numéricos, mostrando a frequência dos valores que ocorrem em um conjunto de dados. O intervalo em que os valores ocorrem é dividido em partes iguais (*bins*) cujo tamanho pode ser especificado pela pessoa que analisa os dados, e a quantidade de valores que estão em cada uma dessas partes é representada por barras. Diferentemente dos gráficos de barras para dados categóricos, histogramas não possuem espaços entre as barras por representarem valores em um espectro contínuo.

No *pandas*, um histograma de valores é facilmente construído adicionando o comando **.hist()** em um *DataFrame* ou *Series*. Assim como para os *boxplots*, os histogramas a seguir mostram a distribuição de valores das colunas **danceability** e **acousticness** do conjunto de músicas das paradas globais do Spotify. Observe que as mesmas conclusões dos *boxplots* podem ser obtidas a partir dos histogramas, por serem duas formas diferentes de visualizar a mesma informação, i.e., a distribuição dos valores para um atributo.

```
[20]: # Histograma de valores de
      #  'danceability' e 'acousticness'

      df[[
          'danceability',
          'acousticness'
      ]].hist(figsize=(7, 1.65));
```



1.5.5. Correlações

Correlação é uma métrica que informa o grau em que duas ou mais variáveis estão relacionadas entre si. A correlação entre variáveis pode ser medida por um coeficiente, cujo valor varia entre -1 e 1. Uma correlação é dita totalmente positiva quando o coeficiente é igual a 1, e totalmente negativa quando igual a -1. Coeficientes iguais a zero informam que não existe correlação explícita entre as variáveis consideradas. Existem várias formas de se calcular coeficientes de correlação, dentre as quais pode-se eleger duas principais:

- *Pearson* (r): mede a correlação linear (de primeira ordem) entre variáveis;
- *Spearman* (ρ): mede correlação monotônica entre variáveis, i.e., utiliza a ordem dos dados ao invés dos valores em si.

Para calcular correlações, o *pandas* dispõe da função **corr()**, que pode ser facilmente aplicada tanto para *Series* quanto para *DataFrame*. O exemplo a seguir exibe o valor do coeficiente de Pearson para as colunas **energy** e **loudness**, que medem a intensidade de uma música e a altura medida em decibéis (dB), respectivamente.

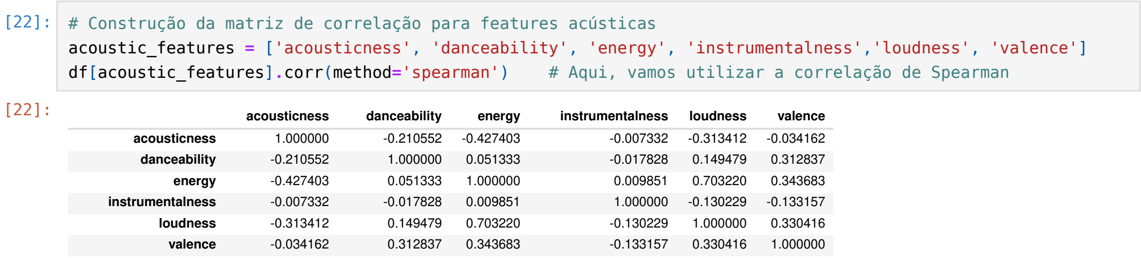
```
[21]: # Correlação de Pearson entre as colunas energy e loudness
      df['energy'].corr(other=df['loudness'], method='pearson')

[21]: 0.737109548990312
```

A partir de tal resultado, pode-se dizer que a correlação linear entre essas duas variáveis é forte e positiva, pois o valor de $0,737$ é muito próximo de 1. Ou seja, à medida que o valor de **energy** aumenta, o valor de **loudness** também cresce, e vice-versa. É importante notar que correlação nem sempre implica causalidade, e então não se pode afirmar que uma coisa acontece devido à outra, mas que ambas ocorrem juntas.

Além disso, cientistas de dados frequentemente calculam correlações entre mais de duas variáveis para ter uma visão ampla de algum fenômeno. Para isso, **matrizes de correlação** são bastante úteis, pois permitem uma fácil e rápida visualização em única estrutura de dados. Em uma matriz de correlação M, as linhas e colunas são as variáveis para as quais se deseja observar a correlação, e os valores m_{ij} representam a correlação entre as variáveis i e j. A diagonal principal dessas matrizes é sempre igual a 1, pois a correlação entre uma variável e ela mesma é total. O próximo exemplo mostra a matriz de correlação entre algumas *features* acústicas das músicas, i.e., obtidas do áudio da mesma.

```
[22]: # Construção da matriz de correlação para features acústicas
      acoustic_features = ['acousticness', 'danceability', 'energy', 'instrumentalness','loudness', 'valence']
      df[acoustic_features].corr(method='spearman')    # Aqui, vamos utilizar a correlação de Spearman
```

[22]:



	acousticness	danceability	energy	instrumentalness	loudness	valence
acousticness	1.000000	-0.210552	-0.427403	-0.007332	-0.313412	-0.034162
danceability	-0.210552	1.000000	0.051333	-0.017828	0.149479	0.312837
energy	-0.427403	0.051333	1.000000	0.009851	0.703220	0.343683
instrumentalness	-0.007332	-0.017828	0.009851	1.000000	-0.130229	-0.133157
loudness	-0.313412	0.149479	0.703220	-0.130229	1.000000	0.330416
valence	-0.034162	0.312837	0.343683	-0.133157	0.330416	1.000000

Nessa matriz, utiliza-se a correlação de Spearman entre as colunas do *DataFrame*. O tipo de correlação pode ser definido através do parâmetro **method**. Além dos coeficientes de Pearson e Spearman, a função **corr()** do *pandas* também permite calcular a correlação de Kendall (τ), que também é baseada na ordem das variáveis.

De modo geral, a análise exploratória dos dados possui grande interseção com a etapa de visualização de dados, pois muitas ferramentas de análise produzem gráficos como resultado (histogramas, gráficos de caixa (*boxplots*), dentre outros). No entanto, o objetivo dessa seção foi apresentar as principais funções do *pandas* e outras bibliotecas que auxiliam na exploração dos dados, ou seja, o foco aqui reside na análise de tais dados. Mais informações sobre os diferentes tipos de visualizações e como personalizá-las são abordadas na próxima seção.

1.6. Visualização de Dados Científicos

Na Ciência de Dados, a Visualização de dados é uma fase importante que envolve a representação gráfica de informações. Através de elementos visuais, ela permite a atribuição de sentido e a comunicação de tendências e padrões nos dados. É um meio de comunicação de resultados de pesquisa, bem como uma plataforma de análise e exploração de dados. Portanto, saber como criar visualizações de dados ajuda cientistas a resolver problemas e analisar os objetos de um estudo de forma detalhada e significativa.

Existem vários métodos e abordagens para a criação de recursos visuais com base na natureza e na complexidade dos dados. Em particular, o Jupyter Notebook é uma ótima ferramenta para a exploração de dados e a criação de visualizações. Ao mesmo tempo, associado ao suporte simplificado do Jupyter, a linguagem Python oferece uma variedade de bibliotecas e módulos de visualização de dados. Matplotlib e Seaborn são duas das bibliotecas de visualização mais populares do Python. Ambas funcionam muito bem para a tarefa de visualização, sendo mais poderosas quando comparadas com os recursos adicionais de plotagem incorporados do *pandas*, apresentadas na seção anterior.

Diante da variedade de opções e métodos, esta seção fornece uma visão geral de como visualizar conjuntos de dados no Jupyter Notebook com a ajuda das bibliotecas Matplotlib e Seaborn para demonstrar os recursos de plotagem do Python. Para melhor compreensão e organização, descrevemos um conjunto básico de recursos visuais agrupados não pelo tipo de dados visualizados, mas pelo tipo de mensagem ou informação que transmitem. Assim, esta seção é dividida em quatro categorias de visualização: Quantidade, Distribuição, Correlação e Evolução.

1.6.1. Quantidade

Ao visualizar quantidades, o interesse está em analisar a magnitude de algum conjunto de números. Por exemplo, pode-se querer visualizar o número total de *streams* de diferentes tipos de música, ou o número total de artistas para cada tipo de gênero musical. Em todos esses casos, existe um conjunto de categorias (por exemplo, tipos de música e gêneros musicais) e um valor quantitativo para cada categoria (número total de *streams*). A visualização mais utilizada para esse cenário é o gráfico de barras.

Gráfico de barras simples. Em um gráfico de barras, cada entidade da variável categórica é representada como uma barra, e o tamanho da barra representa seu valor numérico. Para exemplificar o conceito de gráfico de barras, considere o total de *streams* no Spotify para as músicas solo e colaborativas mais populares em 2020. Esse tipo de dado é comumente visualizado com barras horizontais ou verticais. O exemplo a seguir ilustra como construir um gráfico de barras simples usando Matplotlib (à esquerda) e Seaborn (à

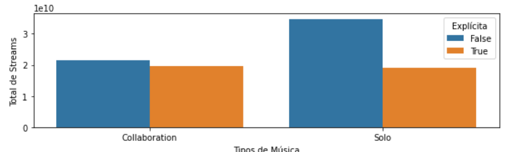
direita). Em ambos os gráficos, para cada tipo de música, existe uma barra que começa em zero e se estende até a quantidade de *streams*.

```
[5]: fig, ax = plt.subplots()
     ax.barh(y=df['Tipos de Música'],
         width=df['Total de Streams'])
     ax.set_xlabel('Total de Streams')
     ax.set_ylabel('Tipos de Música')
     fig.set_size_inches(7, 1.5)
```

```
[6]: sns.barplot(data=df,
         x="Total de Streams",
         y="Tipos de Música",
         order=['Solo', 'Collaboration'],
         color='#3274A1')
     plt.gcf().set_size_inches(7, 1.5)
```

Gráfico de barras agrupadas. O exemplo anterior mostra como uma quantidade varia em relação a uma variável categórica. Porém, frequentemente, é necessário analisar duas variáveis categóricas ao mesmo tempo. Por exemplo, agora pretende-se saber o total de *streams* de músicas explícitas ou não, além do tipo da música. Tal informação é melhor visualizada com um gráfico de barras agrupadas. Esse tipo de gráfico contém um grupo de barras em cada posição ao longo do eixo x, determinado por uma variável categórica, e mais barras dentro de cada grupo conforme a outra variável categórica. As legendas do eixo e do gráfico distinguem os grupos, como o exemplo a seguir.

```
[9]: sns.barplot(
         x="Tipos de Música", y="Total de Streams",
         hue="Explícita",
         data=df)
     plt.gcf().set_size_inches(10, 2.5)
```



1.6.2. Distribuição

Visualização de distribuições facilita entender como uma determinada variável é distribuída em um conjunto de dados. Por exemplo, para visualizar a distribuição da popularidade ou número de seguidores de artistas de sucesso, pode-se utilizar histogramas. Além de histogramas, existem visualizações alternativas como gráficos de densidade e *boxplots*.

Histograma. É uma representação gráfica da distribuição de uma variável numérica, no qual a variável é dividida em barras, e o número de observações por barra é representado pela sua altura. Considere a distribuição da popularidade de artistas das músicas de 2020. Para cada artista, o próximo exemplo mostra a distribuição da variável **popularity** dividida por intervalos de tamanho igual a dez, usando Matplotlib e Seaborn.

```
[4]: fig, ax = plt.subplots()
     ax.hist(df["Popularidade"], bins=10)
     ax.set_xlabel('Popularidade');
     plt.gcf().set_size_inches(12, 2.5)
```

```
[5]: sns.histplot(df["Popularidade"],
         kde=False,
         bins=10);
     plt.gcf().set_size_inches(12, 2.5)
```

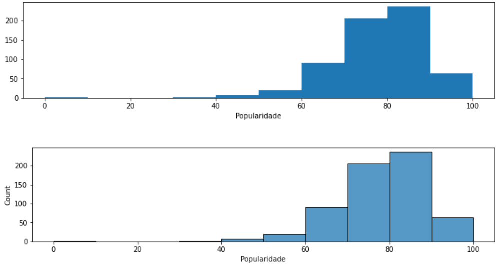
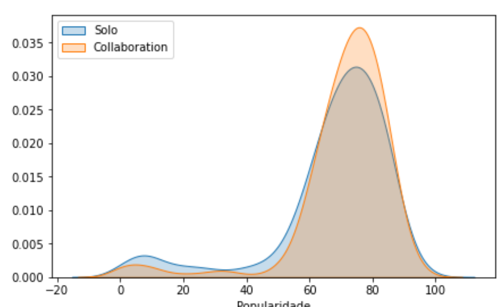


Gráfico de densidade. Geralmente, histogramas são mais utilizados para visualizar uma única distribuição. Porém, ao visualizar mais de uma distribuição simultaneamente, gráficos de densidade são mais recomendados. Em um gráfico de densidade, a distribuição é visualizada através de uma curva contínua. Essa curva precisa ser estimada a partir dos dados, e o método mais comum para tal é chamado de estimativa de densidade de kernel. O exemplo a seguir utiliza o kernel gaussiano através do método **seaborn.kdeplot()** para estimar a distribuição da popularidade conforme o tipo de música. As cores das curvas indicam se a distribuição é de músicas solo ou colaborações.

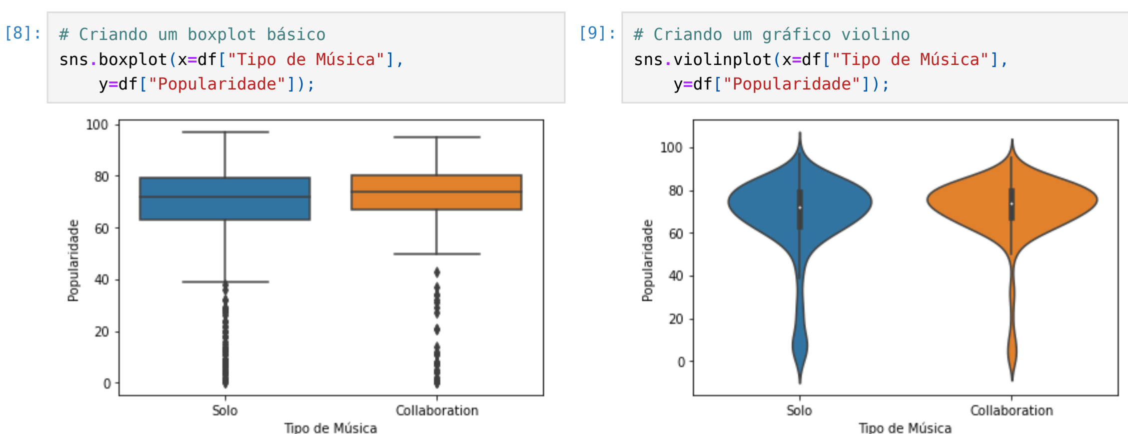
```
[7]: # Plotando duas distribuições em uma mesma figura
fig = sns.kdeplot(df.loc[
    df['Tipo de Música'] == 'Solo', 'Popularidade'
], shade=True, label='Solo')
fig = sns.kdeplot(df.loc[
    df['Tipo de Música'] == 'Collaboration',
    'Popularidade'
], shade=True, label='Collaboration')
plt.xlabel("Popularidade")
plt.ylabel("")
plt.legend(loc='upper left')
plt.gcf().set_size_inches([7, 4.25])
```



***Boxplots* e Gráficos Violino.** Assim como os gráficos de densidade, *boxplots* são uma ótima maneira de visualizar distribuições. Em um *boxplot*, os dados são divididos em quartis, fornecendo um bom resumo da distribuição de variáveis numéricas. Esse tipo de visualização tem a vantagem de ocupar pouco espaço, o que é útil ao comparar distribuições entre muitos grupos ou conjuntos de dados. Uma desvantagem dos *boxplots* é que ao resumir as distribuições, informações podem ser perdidas. O *boxplot* do exemplo a seguir (à esquerda), aparentemente, permite concluir que as colaborações são em média mais populares do que as músicas solo. No entanto, o gráfico não ilustra a distribuição subjacente de pontos em cada categoria ou o número de observações. Uma alternativa é utilizar gráficos violino (à direita), que descrevem a distribuição permitindo uma compreensão mais profunda sobre a mesma. Os gráficos a seguir utilizam os métodos **seaborn.boxplot()** e **seaborn.violinplot()** para melhor visualização das distribuições do exemplo anterior.

```
[8]: # Criando um boxplot básico
sns.boxplot(x=df["Tipo de Música"],
    y=df["Popularidade"]);
```

```
[9]: # Criando um gráfico violino
sns.violinplot(x=df["Tipo de Música"],
    y=df["Popularidade"]);
```



1.6.3. Correlação

Correlação permite analisar como duas variáveis se relacionam entre si, por exemplo, entre popularidade e duração da música. Por exemplo, podemos querer visualizar a relação entre a popularidade e alguma feature acústica de música de sucesso. Para representar graficamente a relação de duas dessas variáveis, geralmente se utiliza um gráfico de dispersão. Ele pode ser útil também para conjuntos de dados com alta dimensionalidade, como exemplificado na Seção 1.5. Esta seção explica melhor o gráfico de dispersão básico e inclui algumas variações, como os gráficos de bolhas e correlogramas. Todos os exemplos desta seção consideram o conjunto de dados *Hits*.

Gráfico de dispersão. Permite estudar a relação entre duas variáveis. Para cada ponto de dados, o valor de sua primeira variável é representado no eixo x e a segunda no eixo y. Os dois gráficos a seguir exemplificam a dispersão sobre a popularidade em relação à duração e à energia das músicas, respectivamente, utilizando o método **matplotlib.pyplot.scatter()**. Note que **subplots()** permite colocar gráficos lado-a-lado na mesma saída.

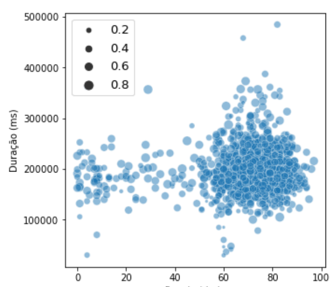
[3]:
```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
# popularidade vs. duração
ax[0].scatter(
    x = df_hits['popularity'],
    y = df_hits['duration_ms'])
ax[0].set_xlabel("Popularidade")
ax[0].set_ylabel("Duração (ms)")
# popularidade vs. energia
ax[1].scatter(
    x = df_hits['popularity'],
    y = df_hits['energy'])
ax[1].set_xlabel("Popularidade")
ax[1].set_ylabel("Energia");
```

Geralmente, linhas ou curvas são ajustadas dentro do gráfico de dispersão para auxiliar a análise. O método **seaborn.regplot()** permite criar os mesmos gráficos de dispersão, mas, agora, com uma reta vermelha representando o ajuste linear.

[4]:
```
# Gráfico de dispersão com o ajuste linear
fig, ax = plt.subplots(ncols=2,figsize=(10,5))
line_kws = {"color":"r","alpha":0.7}
sns.regplot(x='popularity', y='duration_ms',
            data=df_hits, ax=ax[0],
            line_kws=line_kws)
ax[0].set(xlabel='Popularidade',
          ylabel='Duração (ms)')
sns.regplot(x='popularity', y='energy',
            data=df_hits, ax=ax[1],
            line_kws=line_kws)
ax[1].set(xlabel='Popularidade',
          ylabel='Energia');
```

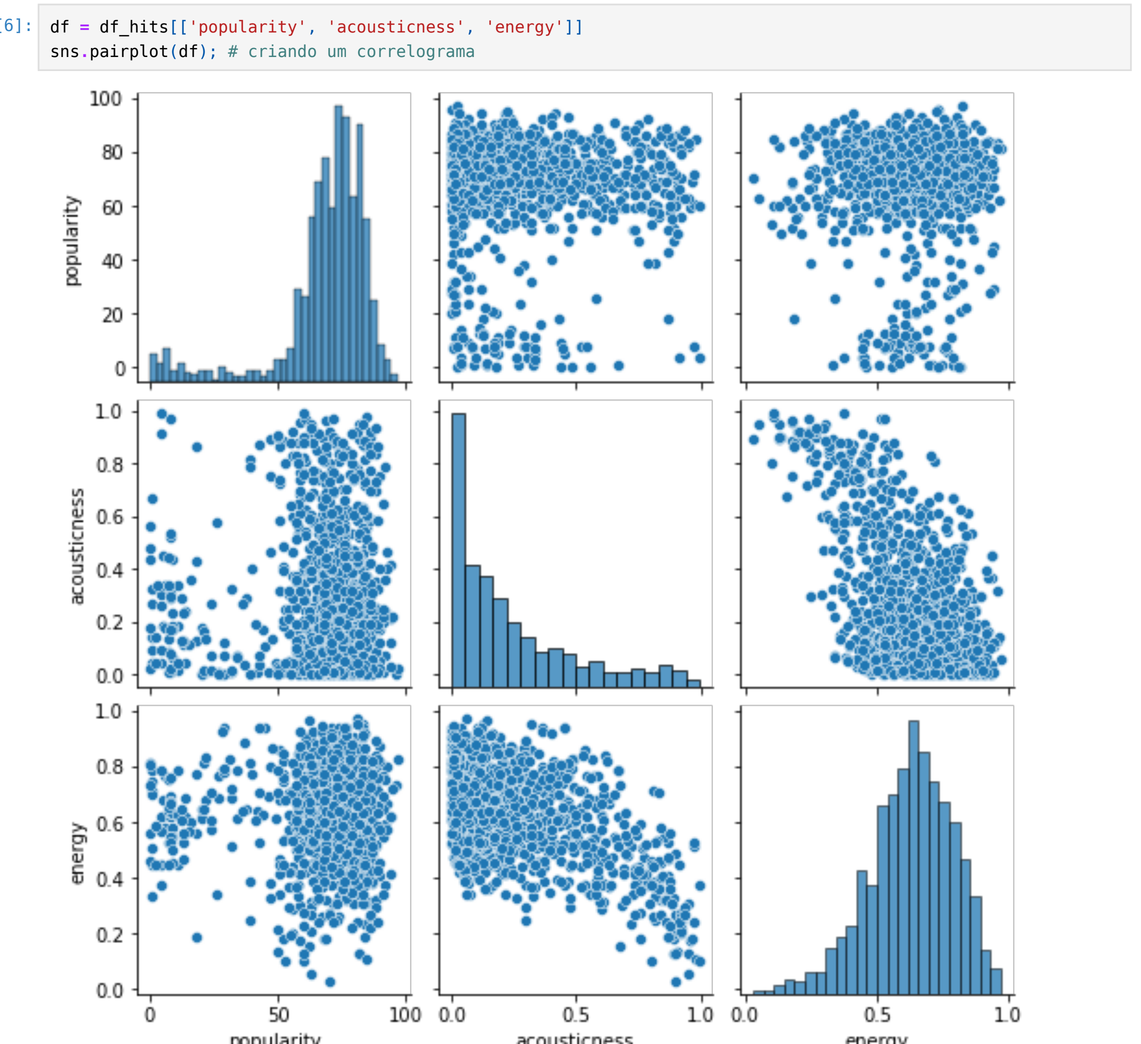
Gráfico de bolhas. É um gráfico de dispersão com três dimensões definidas por variáveis numéricas como entrada: duas estão nos eixos x e y, e a terceira no tamanho dos pontos. O código a seguir usa a função **seaborn.scatterplot()** para analisar a popularidade das músicas em relação a sua duração e nível de energia. O parâmetro **size** controla o tamanho do círculo de acordo com uma variável numérica (dado **energy**).

[5]:
```python
fig, ax = plt.subplots(figsize=(5, 5))
sns.scatterplot(
    data=df_hits,
    x="popularity",
    y="duration_ms",
    size="energy",
    alpha=0.5,
    sizes=(5, 100))
plt.legend(loc='upper left', fontsize=13)
ax.set_xlabel('Popularidade')
ax.set_ylabel('Duração (ms)');
```



Correlogramas. Ou matriz de correlação, permite analisar a relação entre pares de variáveis numéricas de um conjunto de dados. Correlogramas são muito úteis na Análise Exploratória, pois permitem visualizar as relações de todo o conjunto de dados de uma só vez. A relação entre cada par de variáveis é geralmente mostrada com um gráfico de dispersão, enquanto a diagonal representa a distribuição de cada variável, usando um histograma ou um gráfico de densidade. No próximo exemplo, visualizamos três variáveis utilizando a função **seaborn.pairplot()**: **popularity**, **acousticness** e **energy**.

[6]:
```python
df = df_hits[['popularity', 'acousticness', 'energy']]
sns.pairplot(df); # criando um correlograma
```



1.6.4. Evolução

A visualização de uma evolução permite analisar uma tendência nos dados ao longo de intervalos de tempo – uma série temporal. Por exemplo, pode-se visualizar a evolução de músicas solo vs. colaborações ou a média de *streams* das músicas mais populares em 2020. Nesses casos, gráficos de linha e de área são frequentemente utilizados, sendo semelhantes a um gráfico de dispersão, porém os pontos de medição são ordenados e unidos com segmentos de linha reta. O exemplo a seguir mostra um gráfico de linha e outro de área para visualizar a evolução da média de *streams* no ano de 2020. O código utiliza as funções **matplotlib.pyplot.plot()** e **fill_between()**, respectivamente.

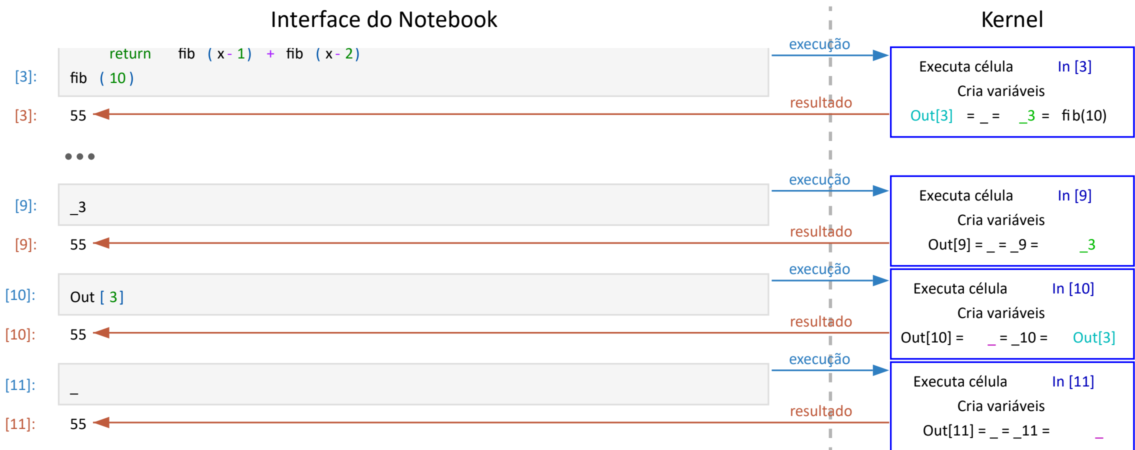
```
[5]: fig, ax = plt.subplots(1, 2, figsize=(15, 5))
     ax[0].plot('chart_week', 'streams', '-', data=df) # gráfico de dispersão
     ax[0].set_ylabel("Média de streams")
     ax[1].plot('chart_week', 'streams', '-', data=df) # gráfico de área
     ax[1].fill_between('chart_week', 'streams', data=df)
     ax[1].set_ylabel("Média de streams");
```


1.7. Jupyter Avançado

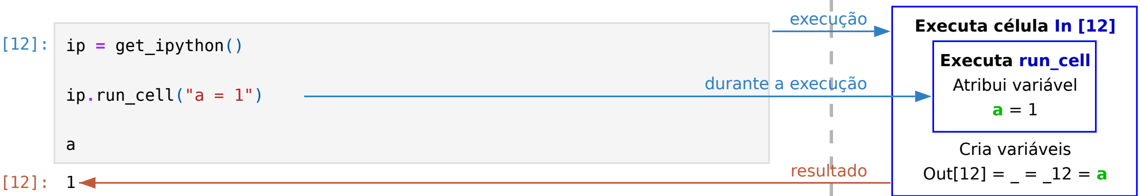
Além do já apresentado, o Jupyter possui características avançadas que permitem tanto um uso mais aprofundado de sua interatividade quanto de suas visualizações para análise de dados. Essas características estão presentes em diversas bibliotecas existentes e também podem ser usadas para estender o Jupyter. Esta seção tem objetivo de se aprofundar em características avançadas do Jupyter que permitam a criação de extensões.

1.7.1. IPython

A execução de códigos no Jupyter acontece através de um *kernel*. Ao executar um código na interface web, o Jupyter envia uma mensagem para o *kernel*, que trata da execução, mantém o ambiente de execução, e devolve o resultado em algum formato suportado. Existem *kernels* de diversas linguagens para Jupyter e até mesmo linguagens com múltiplas implementações de *kernel*. Para Python, o *kernel* mais usado é o IPython, que fornece um superconjunto da linguagem Python com operações orientadas ao Jupyter e visualizações de diferentes formatos. Após a execução de cada célula, o IPython popula variáveis para permitir que resultados de células anteriores sejam acessados e visualizados. O exemplo a seguir continua o exemplo da Figura 1.3, mostrando a execução de células no *kernel* e a população de variáveis. Perceba que é possível acessar o resultado da célula 3 tanto pela variável **_3** (célula **[9]**) quanto pelo acesso **Out[3]** (célula **[10]**). Além disso, se a célula tiver acabado de ser executada, é possível acessar seu resultado pela variável **_**, como ocorre na última célula da figura.



O *kernel* também pode ser diretamente acessado através da função **get_ipython** para a realização de funções mais avançadas. O próximo exemplo apresenta uma célula que executa essa função para obter o objeto do *kernel* e executa a função **run_cell** para executar uma célula diretamente no *kernel* sem uma célula associada no notebook. Esta célula define a variável **a** com valor 1. Por fim, a célula original termina sua execução acessando o valor de **a** recentemente definido.



Além de controlar a execução do código Python em notebooks e de criar variáveis após a execução de células, o IPython também estende a linguagem para adicionar expressões *bang*, "mágicas" de linha e de célula e consultas à documentação, como exemplificado no próximo notebook. Este notebook utiliza todos esses recursos para: listar arquivos no diretório de *datasets* (*bang* na célula **[2]**); verificar o histórico de execução de células (mágica de linha na célula **[3]** – perceba que o notebook começa pela célula **[2]**); mostrar a quantidade de linhas em cada arquivo do *dataset* (atribuição de *bang* na célula **[4]**); criar links em HTML para os arquivos do *dataset* (mágica de célula em **[5]**); e consultar a documentação da função **len** (interrogação na célula **[6]**). Apesar de ser uma extensão à sintaxe padrão do Python, todas essas operações são transformadas em código Python pelo IPython imediatamente antes da execução.

Uma *expressão bang* é uma expressão que executa um comando no sistema e retorna a saída padrão do comando como resultado. Ela pode ser usada com uma exclamação precedendo o resto do comando na linha. Por exemplo, a expressão **!ls** usada na célula **[2]** do exemplo anterior retorna todos os arquivos do diretório especificado em um sistema Unix. Note que **ls** é um comando que não existe normalmente no Windows. Portanto, a execução dessa célula em um sistema operacional Windows resultaria em erro. Para executar essa operação, o IPython transforma **!ls** em **get_ipython().system('ls')**.

```
[2]: !ls ../dataset/
```

```
spotify_artists_info_complete.tsv  spotify_hits_dataset_complete.tsv
spotify_charts_complete.tsv
```

```
[3]: %history -n
```

```
   1: !ls
   2: !ls ../dataset/
   3: %history -n
```

```
[4]: files = !ls ../dataset/  # Atribui lista de arquivos a variável files
     for name in files:  # Percorre arquivos, pegando nome e número de linhas
         print(name, '--', len(open("../dataset/" + name).readlines()))
```

```
spotify_artists_info_complete.tsv -- 626
spotify_charts_complete.tsv -- 10401
spotify_hits_dataset_complete.tsv -- 1285
```

```
[5]: %%html
     <a href="../dataset/spotify_artists_info_complete.tsv">artists</a>
     <a href="../dataset/spotify_charts_complete.tsv">charts</a>
     <a href="../dataset/spotify_hits_dataset_complete.tsv">hits</a>
```

artists charts hits

```
[6]: len?
```

```
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

Uma *mágica* do IPython tem o objetivo de modificar a forma de executar operações. Existem dois tipos de mágicas: mágica de linha e de célula. Uma mágica de linha altera o restante da linha e é usada de forma semelhante a expressões *bang*, com o símbolo % precedendo o nome da mágica. Por exemplo, a mágica **%history -n** usada na célula **[3]** do exemplo anterior exibe o histórico de células executadas com a numeração de células. Perceba que o resultado também apresenta o histórico da célula **[1]**, que já não existe mais no notebook. Para executar essa mágica de linha, o IPython transforma **%history -n** em **get_ipython().run_line_magic('history', '-n')**.

Expressões *bang* e mágicas de linha também podem ser atribuídas a variáveis do Python, como ocorre na segunda linha da célula **[4]** do exemplo anterior. Para essa atribuição, é necessário retornar um valor ao invés de simplesmente imprimir na tela. Portanto, o IPython transforma **files = !ls** em **files = get_ipython().getoutput('ls')**. O objeto retornado por **getoutput** funciona como uma lista de linhas, mas possui métodos especiais de visualização e transformação para o uso no notebook.

Uma mágica de célula altera a célula inteira e deve ser usada no topo da célula com %% precedendo o nome da mágica. A mágica **%%html** permite escrever código HTML que é exibido diretamente no navegador, mesmo que a linguagem do notebook seja Python. Na célula **[5]** do exemplo anterior, essa mágica é usada para criar links para os arquivos do *dataset*, que são normalmente carregados pela aplicação do notebook. Para a executar esta célula, o IPython a transforma em **get_ipython().run_cell_magic('html', '', '<a>...\n')**.

O IPython oferece diversas mágicas de linha e de célula e também permite definir novas mágicas como apresentado na Seção 1.7.2. Para obter uma lista de mágicas disponíveis, é possível usar a mágica **%lsmagic**. A Tabela 1.4 apresenta as 10 mágicas de linha e as 10 mágicas de célula mais usadas em notebooks no GitHub. Para construir

Tabela 1.4. **Mágicas mais usadas em um conjunto de 198.374 notebooks.**

Mágica	Notebooks	Descrição
%matplotlib	156.353	Configura matplotlib para funcionar interativamente
%load_ext	14.216	Carrega uma extensão IPython pelo nome do módulo
%autoreload	12.920	Recarrega módulos automaticamente
%pylab	8.183	Carrega numpy e matplotlib
%time	5.172	Mede a duração da execução de uma expressão
%config	4.805	Configura o IPython
%pinfo	3.253	Acessa a documentação de um objeto
%run	2.474	Executa um arquivo Python dentro do IPython
%timeit	2.318	Tira a mediana de várias execuções de %time
%reload_ext	1.635	Recarrega uma extensão IPython
%%time	13.129	O mesmo que %time, mas para a célula
%%html	2.944	Exibe célula como HTML
%%writefile	2.937	Escreve o conteúdo de uma célula em um arquivo
%%bash	2.850	Executa a célula com bash em um subprocesso
%%timeit	2.628	O mesmo que %timeit, mas para a célula
%%javascript	2.447	Executa a célula em Javascript no navegador
%%sql	1.822	Realiza consulta SQL
%%R	1.285	Executa a célula com R em um subprocesso
%%capture	1.131	Captura stdout, stderr e IPython display da execução
%%cython	1.117	Compila e importa todas as funções definidas em cython

essa tabela, foram usados dados de um estudo que coletou 1.450.071 de notebooks do GitHub até abril de 2018 [Pimentel et al. 2021]. Apesar de ter coletado essa quantidade inicial de notebooks, após a seleção de apenas notebooks de repositórios que continuavam existindo em julho de 2020, o descarte de duplicatas, e o descarte de notebooks com erros de sintaxe ou escritos outras linguagens, a quantidade total de notebooks em Python usados para a análise foi 886.668. Desses, 173.256 notebooks utilizam mágicas de linha e 33.541 notebooks utilizam mágicas de célula. Perceba na tabela que a mágica de linha mais usada é a `%matplotlib`, que serve para configurar a biblioteca *matplotlib* para funcionar interativamente no notebook. Já a mágica de célula mais usada é a `%%time`, que serve para medir o tempo de execução da célula.

Por fim, o IPython estende a sintaxe do Python para permitir o acesso a documentações e definições de funções e classes através dos símbolos ? e ??, respectivamente. Por exemplo, o comando `len?` da célula **[6]** do exemplo anterior apresenta a função *len* com sua assinatura, documentação e tipo. Esses símbolos podem ser usados com qualquer função, classe, módulo, ou variável do Python e com qualquer mágica do IPython.

Além de executar células e de estender a sintaxe para melhorar a interatividade, o *kernel* também tem a função de fornecer métodos para visualizações ricas no Jupyter. As seções anteriores apresentaram diversos gráficos e tabelas em notebooks como visualizações ricas. Dentre os formatos personalizados, é possível exibir imagens em PNG, JPEG, SVG e PDF, equações em LaTeX, texto formatado com Markdown, Javascript executável no navegador e tabelas em HTML. No IPython, objetos são exibidos em duas situações: quando a função `display` é invocada, ou quando o objeto é o resultado da célula (i.e., úl-

tima expressão). O formato da visualização rica depende dos métodos implementados para isso em cada objeto, como apresentado na próxima seção.

1.7.2. Estendendo o IPython

Esta seção mostra como é possível estender o IPython tanto para definir novas mágicas quanto para definir novas visualizações para cada tipo de objeto.

Mágicas. Para construir novas mágicas, é possível definir uma classe de mágicas em que cada método é uma mágica de linha ou de célula. Esses métodos recebem os argumentos da mágica como parâmetro **line** e o código da célula como parâmetro **cell**, e retornam o resultado da operação. O código a seguir mostra o exemplo de uma mágica **%%artist** que exibe um tocador do Spotify para um registro de artista (i.e., a banda Coldplay).

```
[1]: import pandas as pd
     dfa = pd.read_csv("../dataset/spotify_artists_info_complete.tsv", sep="\t")
     dfa.loc[0]
```

```
[1]: artist_id                                  4gzpq5DPGxSnKTe4SA8HAU
     name                                                     Coldplay
     followers                                                29397183
     popularity                                                     90
     genres                                  ['permanent wave', 'pop']
     image_url      https://i.scdn.co/image/4ffd6710617d289699cc0d...
     Name: 0, dtype: object
```

```
[2]: from IPython.core.magic import Magics, magics_class, line_magic
     EMBED_URL = ('<iframe src="https://open.spotify.com/embed/{type_}/{id}" width="{w}" height="{h}"'
                  ' frameborder="0" allowtransparency="true" allow="encrypted-media"></iframe>')
     @magics_class  # Define classe de mágicas
     class SpotifyMagics(Magics):
         @line_magic  # Define mágica de lina
         def artist(self, line):  # Executa mágica de célula no kernel (self.shell) para exibir HTML
             return self.shell.run_cell_magic("html", '', EMBED_URL.format(id=line, type_="artist", w=360, h=180))
     get_ipython().register_magics(SpotifyMagics)  # Cadastra mágica
```

```
[3]: %artist 4gzpq5DPGxSnKTe4SA8HAU
```

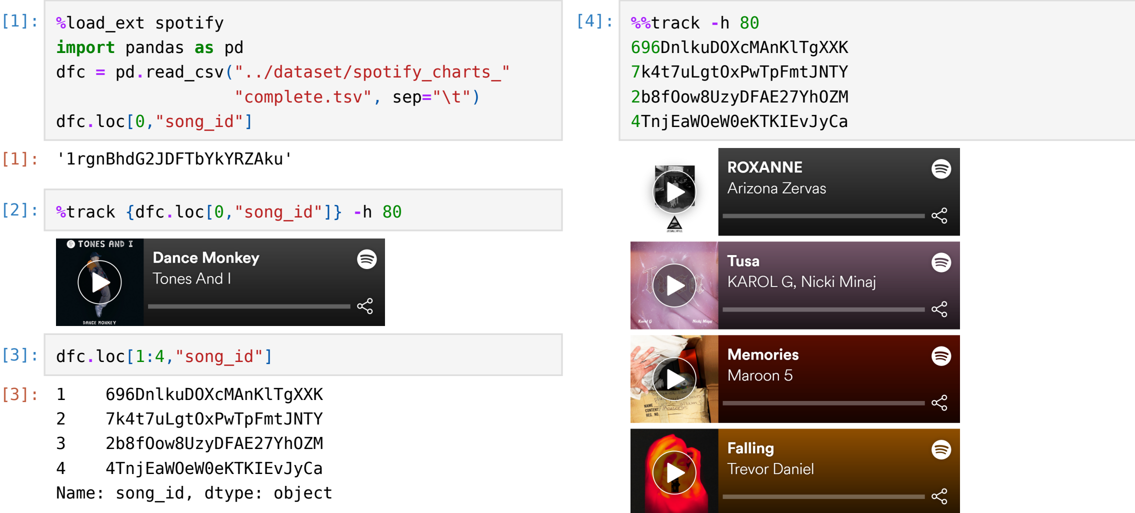
Considerando as células do código anterior, a primeira importa a biblioteca *pandas*, o arquivo de artistas e a informação do artista na posição 0. Nessa célula o dado mais relevante é o identificador do artista (i.e., **artist_id**). Em seguida, a segunda célula define uma mágica de linha com o nome **%artist** que executa a mágica de célula **%%html** para exibir um IFrame com o tocador do spotify, usando o identificador do artista passado como parâmetro para a mágica (i.e., argumento **line** do método **artist**). Para definir essa mágica de linha, a célula decorou o método **artist** com **line_magic** (i.e., precede o comando), decorou a classe **SpotifyMagics** com **magics_class**, e registrou a(s) mágica(s) dessa classe no kernel com a função **register_magics**. Por fim, a terceira célula do notebook utilizou a mágica definida passando o id do artista obtido na primeira célula e exibiu um tocador. Com esse tocador, é possível ouvir as músicas dos artistas que foram analisados anteriormente.

Essa mágica está restrita a um único notebook. Para reusá-la em mais de um notebook, cria-se uma extensão para o IPython em um módulo externo: um arquivo Python com o código da mágica e com uma função **load_ipython_extension** encarregada de registrá-la. O próximo exemplo define um módulo IPython no arquivo **spotify.py** para a mágica **%artist** e uma nova mágica **%track**, que atua como mágica de linha e como mágica de célula para exibir tocadores de música. Para isso, essa mágica foi decorada com **line_cell_magic**.

```python
from IPython.core.magic import Magics, magics_class, line_magic, line_cell_magic
from IPython.core.magic_arguments import parse_argstring, magic_arguments, argument
from IPython.display import HTML
EMBED_URL = ('<iframe src="https://open.spotify.com/embed/{type_}/{id}" width="{w}" height="{h}"'
             ' frameborder="0" allowtransparency="true" allow="encrypted-media"></iframe>')

# Define classe de mágicas
@magics_class
class SpotifyMagics(Magics):
    # Mágica artist definida anteriormente
    @line_magic
    def artist(self, line):
        return HTML(EMBED_URL.format(id=line, type_="artist", w=360, h=180))

    # Nova mágica track
    @magic_arguments()  # Define argumentos para mágica
    @argument("song_ids", nargs="*", help="Lista de músicas")
    @argument("-w", "--width", type=int, default=300, help="Largura")
    @argument("-h", "--height", type=int, default=80, help="Altura")
    @line_cell_magic  # Define mágica de linha e de célula
    def track(self, line, cell=""):
        # Carrega argumentos
        args = parse_argstring(self.track, line)
        # Usa lista de ids do parâmetro ou linhas da célula
        ids = args.song_ids or cell.split("\n")
        # Retorna tocadores separados por <br>
        return HTML("<br>".join(
            EMBED_URL.format(id=song_id, type_="track", w=args.width, h=args.height)
            for song_id in ids if song_id  # Um tocador para cada música
        ))

# Carrega extensão
def load_ipython_extension(kernel):
    kernel.register_magics(SpotifyMagics)  # Cadastra mágicas
```

Outra diferença da mágica **%track** para a mágica **%artist** está na definição de argumentos. A mágica **%track** aceita uma lista de músicas e o tamanho do tocador como argumentos. Esses argumentos são configurados pelos decoradores **magic_arguments** e **argument**. Além disso, eles são usados pela função **parse_argstring**. Por fim, ao invés de usar a mágica de célula **%%html** para exibir o IFrame do tocador, a função **track** usa diretamente um objeto do tipo **HTML**, que define a própria forma de visualização HTML. Apesar desta alteração não ser estritamente necessária, ela foi realizada em ambas as funções para permitir o uso destas mágicas na extensão de visualizações ricas, como a seguir.

Para carregar a extensão, usa-se a mágica **%load_ext spotify**, como no próximo exemplo. A primeira célula carrega a extensão e os dados, e a segunda usa a mágica de linha **%track** para exibir o tocador de uma música. Essa música é definida por um código Python que é executado por conta das chaves nos argumentos. Caso os argumentos fossem passados sem as chaves, seriam interpretados como texto. A última célula possui a mágica de célula **%%track**, que cria tocadores para cada linha da célula.



Visualizações ricas. Como mencionado, o Jupyter suporta a visualização de textos, erros, imagens, HTML com Javascript e Markdown. Porém, o Python em si só possui a representação textual de seus objetos. Para aproveitar as visualizações ricas do Jupyter, pode-se usar a função **display** do IPython, passando o valor a ser exibido e o tipo da exibição (i.e., SVG, PNG, JPEG, HTML, Javascript, LaTeX). Também pode-se definir métodos especiais na forma **_repr_<tipo>_** em classes para retornarem cada tipo de visualização. Por exemplo, a definição do método **_repr_png_** permite retornar o conteúdo binário de uma imagem PNG para exibição no Jupyter. Ainda, é possível definir um único método **_repr_mimebundle_** para retornar todas as visualizações suportadas pelo objeto.

O próximo exemplo define esses métodos em duas classes, **Artist** e **Track**. Essas classes recebem como parâmetro de inicialização uma linha dos *Dataframes* instanciados anteriormente (i.e., **dfa** e **dfc**). A classe **Artist** define o método **__repr__** para visualização padrão no Python, o método **_repr_html_** para visualização do tocador em HTML, e o método **_repr_markdown_** para visualização de texto formatado em Markdown. Já a classe **Track** define um único método **_repr_mimebundle_**, que retorna visualizações nesses mesmos formatos.

Por padrão, o Jupyter exibe apenas um dos formatos disponíveis seguindo a ordem HTML, Markdown, LaTeX, SVG, imagem, JavaScript, texto. Note que as células **[4]** e **[8]** desse exemplo exibiram a visualização HTML dos objetos **artist** e **track** ao retornarem esses objetos. Contudo, é possível forçar a visualização de outros formatos usando a função **display**. As células **[3]** e **[6]** permitem visualizar esses objetos como Markdown.

Note que a função **_repr_mimebundle_** de **Track** define um formato adicional, **trk.spotify+json**, que a célula **[7]** tenta visualizar. Esse formato adicional não é suportado pelo Jupyter por padrão, portanto não é exibido. Entretanto, é possível criar extensões para a interface do Jupyter – tanto do Notebook quanto do Lab – para suportar novos formatos.[7] A criação de extensões para a interface requer a definição de arquivos externos relativamente grandes e está fora do escopo deste capítulo.

[7] https://github.com/jupyterlab/mimerender-cookiecutter-ts

```
[2]: class Artist:
         def __init__(self, row):
             self.row = row
         def __repr__(self):
             return str(self.row['name'])
         def _repr_html_(self):
             html = %artist {self.row.artist_id}
             return html.data
         def _repr_markdown_(self):
             return f"# {self.row['name']}"
     artist = Artist(dfa.loc[0])
```

```
[3]: display(artist, include=["text/markdown"])
```

Coldplay

```
[4]: artist
```

Músicas mais tocadas de Coldplay

1 Yellow 4:26
2 The Scientist 5:09
3 Something Just Like This 4:07

```
[5]: class Track:
         def __init__(self, row):
             self.row = row
         def _repr_mimebundle_(self, **kwargs):
             row = self.row
             html = %track {row.song_id}
             return {
               'text/markdown': f"{row.artist} - "
                                f"*{row.track_name}*",
               'text/html': html.data,
               'trk.spotify+json': row.to_json(),
               'text/plain': row.track_name
             }
     track = Track(dfc.loc[0])
```

```
[6]: display(track, include=["text/markdown"])
```

Tones And I - *Dance Monkey*

```
[7]: display(track, include=["trk.spotify+json"])
```

```
[8]: track
```

Dance Monkey
Tones And I

1.7.3. Widgets

Finalmente, uma outra forma de interagir com o Jupyter é a partir de *widgets* interativos. *Widgets* utilizam as visualizações ricas do IPython e do Jupyter para interatividade e podem ser usados para fazer formulários, dashboards e até mesmo variar rapidamente parâmetros de funções em análises.

O próximo exemplo mostra um formulário para consultar diversos percentis de uma coluna de um *DataFrame* de forma interativa. Tal formulário é criado em cima da função **percentil**, que recebe a coluna e o percentil como parâmetros e imprime qual é o valor determinado para o percentil. Para transformar essa função em um formulário composto de *widgets*, usa-se o decorador **@interact** da biblioteca **ipywidgets**. Esse decorador tenta criar um formulário com um *widget* para cada argumento da função decorada, identificando o tipo mais provável dos atributos padrões.

```
[2]: from ipywidgets import interact, interact_manual, IntSlider
     @interact(q=IntSlider(value=50, min=0, max=100))
     def percentil(column="duration_ms", q=10):
         print(f"Percentil {q} de {column} é {df[column].quantile(q=q/100)}")
```

```
column  duration_ms
     q  ----o----   50
Percentil 50 de duration_ms é 193683.0
```

Note que foi criado um *widget* de texto para o argumento **column** que está com um valor texto como padrão. Além disso, o decorador **@interact** aceita parâmetros próprios para definir formatos diferentes para determinados argumentos. O código do exemplo define o parâmetro **q** como controlado por um *widget* de controle deslizante, **IntSlider**, com valor inicial 50, mínimo 0 e máximo 100.

Ao executar a célula, o formulário é exibido com diversos *widgets* e quem estiver usando o notebook pode interagir com eles, deslizando o controle do argumento **q** ou

escrevendo um nome de coluna diferente para o argumento **column**. Assim que qualquer valor é alterado, a função **percentil** é executada com os valores atualizados, e o resultado da execução imediatamente atualizado. Isso permite maior interatividade na análise.

Em algumas situações, a função computada pelo formulário é demorada e não deseja-se executá-la automaticamente a cada pequena alteração em valores. Para utilizar *widgets* nesse caso, é possível substituir o decorador **interact** por **interact_manual**. Esse decorador cria um *widget* de botão no final do formulário para executar a função decorada apenas quando todos os parâmetros forem ajustados.

Nem sempre uma única função decorada é o suficiente para construir a interatividade desejada em formulários e *dashboards*. O próximo exemplo apresenta um formulário em que a pessoa que está interagindo com o notebook deve primeiro escolher uma música do conjunto de dados e depois uma artista que esteja associada à música. Com esse objetivo, a célula **[2]** define dois campos de seleção com busca (**Combobox cs** para música e **ca** para artista), dois campos de exibição para tocadores (**Output out_song** para música e **out_artist** para artista), e uma caixa vertical para a exibição do formulário como um único *widget* (**VBox**). Este *widget* não é exibido de imediato.

[2]:
```python
from ipywidgets import Combobox, Output, VBox
# Cria items para comboboxes
songs = df.song_name.to_list()
artists = []
# Cria form com 2 campos e 2 saídas
cs = Combobox(description="Música", options=songs)
out_song = Output()
ca = Combobox(description="Artista", options=[""])
out_artist = Output()
widget = VBox([cs, out_song, ca, out_artist])
```

[3]:
```python
# Chama função ao alterar a musica
@cs.observe
def change_song(w):
    global artists
    # Limpa saídas e seleção ca
    out_song.clear_output()
    out_artist.clear_output()
    ca.value = ""
    artists = []
    # Seleciona linha da música
    if cs.value not in songs:
        return
    index = songs.index(cs.value)
    row = df.loc[index]
    # Exibe música
    with out_song:
        html = %track {row.song_id} -w 360
        display(html)
    # Atualiza opções de artista
    artists = eval(row.artist_id)
    ca.options = tuple(eval(row.artist_name)+[""])
```

[4]:
```python
# Chama função ao alterar o artista
@ca.observe
def change_artist(w):
    # Limpa saída de artista
    out_artist.clear_output()
    # Seleciona artista
    v = ca.value
    if not v or v not in ca.options:
        return
    index = ca.options.index(v)
    # Exibe artista
    with out_artist:
        html = %artist {artists[index]}
        display(html)
```

[5]:
```python
widget
```

Música Happy Xmas (War is Over) ▼

Happy Xmas (War Is Over) - Rem...
John Lennon

Artista Yoko One ▼

Músicas mais tocadas de Yoko Ono
1 Instant Karma! (We All Shine On) - Re... 3:20
2 Happy Xmas (War Is Over) - Remaster... 3:34
3 Give Peace A Chance - Remastered 2... 4:54

A célula **[3]** define uma função para observar mudanças de valores no campo **cs** (i.e., música). Toda vez que uma música é selecionada, a função **change_song** é chamada. Essa função apaga o conteúdo dos campos de exibição **out_song** e **out_artist** utilizando o método **clear_output** e verifica se o nome da música existe no *DataFrame* usado. Se existir, a função exibe o tocador da música utilizando a mágica **%track** no gerenciador de contexto **with out_song**, e atualiza o *Combobox* **ca** com a lista de artistas da música.

De forma semelhante, a célula **[4]** define uma função para observar mudanças de valores no campo **ca** (i.e., artista). Quando o campo de artista é alterado, a função **change_artist** é chamada e apaga o campo de exibição de artista e verifica a existência de artista no campo de seleção. Se existir, a função usa a mágica **%artist** para exibir o tocador de artista no local apropriado através do gerenciador de contexto **with out_artist**. Por fim, a célula **[5]** retorna o *widget* resultante e permite a interação com os campos definidos.

1.8. Ciência Aberta com Jupyter

Por ser usado em experimentos e análises de dados, existe a necessidade de que notebooks do Jupyter sejam reprodutíveis e estejam disponíveis em plataformas abertas para consulta e reprodução de experimentos [Perkel 2021a]. Esta seção trata tanto do compartilhamento quanto do desenvolvimento de notebooks reprodutíveis.

1.8.1. Compartilhamento de Notebooks

Arquivos de notebooks possuem código e resultados de execução de notebook. Por conterem código, é natural que se usem plataformas de compartilhamento e versionamento de código, como GitHub. Essas plataformas utilizam o git ou outro sistema de versionamento para gerenciar diferentes versões do código e chegam a implementar a visualização de notebooks em suas interfaces web.

Apesar de permitirem o armazenamento de arquivos de notebooks, ferramentas como git podem não ser a escolha ideal para experimentos científicos por diversos motivos. Primeiro, elas tentam realizar operações de *diff* em arquivos de texto para reduzir a sobrecarga do armazenamento de versões e permitir o desenvolvimento paralelo. Como notebooks são armazenados em arquivos de texto (JSON) que também possuem resultados binários (imagens) dentro, a realização do *diff* pode falhar e induzir ao erro no uso habitual de notebooks. Ferramentas como *nbdime*[8] podem reduzir o problema com operações de *diff* e *merge*, mas exigem que cientistas as usem ativamente por fora do uso padrão do git. Segundo, experimentos científicos costumam ter grandes conjuntos de dados de entrada que precisam ser compartilhados para garantir a reprodutibilidade. Por ser projetado para fazer versionamento de código, o git não lida bem com um volume grande de dados e plataformas como GitHub restringem o tamanho de arquivos e de repositórios, inviabilizando o uso. Por fim, não existe uma garantia de que páginas do GitHub se manterão sem mudanças ao longo do tempo.

Dessa forma, ao citar que um repositório possui dados do experimento, existe a possibilidade do repositório versionado de código evoluir e deixar de representar o experimento citado em um artigo. Esse problema pode ser amenizado com a citação de uma versão específica, mas requer um trabalho a mais de cientistas marcarem a versão.

Por conta dessas limitações, repositórios de acesso aberto como figshare[9] e Zenodo[10] foram criados especificamente para permitir o compartilhamento de dados científicos, notebooks e outros artefatos digitais. Apesar de permitirem o versionamento dos dados, essas plataformas não foram projetadas para o desenvolvimento contínuo de ver-

[8]https://nbdime.readthedocs.io/en/latest/
[9]https://figshare.com/
[10]https://zenodo.org/

sões. Ao invés disso, elas promovem o compartilhamento de versões finais dos experimentos, com a indicação de licenças apropriadas e documentação de como reproduzir. Após a confirmação do envio, essas plataformas geram números de DOI que podem ser citados em artigos com a garantia de que não ocorrerão mudanças.

1.8.2. Reprodução e Boas Práticas

A análise exploratória em notebooks ajuda a entender melhor os dados e fazer mudanças parciais sem ter que reexecutar toda a análise. Por outro lado, esse modo de trabalhar em notebooks gera células fora de ordem e estados ocultos de execução que podem comprometer a reprodutibilidade de experimentos. Além disso, a falta de inclusão dos dados de entrada em repositórios (ver seção anterior) e falta de rastreio de bibliotecas usadas no momento do desenvolvimento também podem comprometer a reprodutibilidade.

Esta parte do capítulo tem como base um estudo de reprodutibilidade que coletou mais de um milhão de notebooks do GitHub e observou taxas baixas de reprodução ao tentar executar notebooks com código Python em diferentes ambientes e diferentes ordens [Pimentel et al. 2021]. A taxa de execuções que foram capazes de executar todas as células variou de 22,57% a 26,09%. Já a taxa de execuções que produziram os resultados próximos aos armazenados nos notebooks variou de 4,9% a 15,04%, após normalizações de resultados. Esse estudo evidenciou que não basta usar notebooks para fazer Ciência de Dados com reprodutibilidade. É necessário ter um trabalho organizado e seguir boas práticas como as propostas a seguir [Pimentel et al. 2019].

1. **Usar títulos descritivos com um conjunto restrito de caracteres (A-Z a-z 0-9 . _ -) e células Markdown com títulos mais detalhados no corpo.** No Jupyter, o nome de um arquivo de notebook representa seu título. Esse título muitas vezes é importante para determinar não só o assunto do notebook como também a ordem de execução dos notebooks em projetos com muitos arquivos. Portanto, recomenda-se o uso de títulos descritivos para que isso seja possível. Contudo, alguns sistemas operacionais não suportam todo tipo de caractere em seus sistemas de arquivo. Por exemplo, o uso do caractere de dois pontos pode causar problemas no Windows e no MacOS. Dessa forma, para garantir a reprodutibilidade em diversos sistemas, recomenda-se apenas o uso de um conjunto restrito de caracteres no título, como o conjunto definido no guia de nomes de arquivos completamente portáteis do POSIX [Lewine 1991]. O título completo do notebook pode ser escrito dentro do documento através de células Markdown.

2. **Prestar atenção ao final do notebook e verificar se células do final foram executadas e se não cabe nenhuma célula de Markdown para concluir a análise.** Durante a análise de notebooks do GitHub [Pimentel et al. 2021], observou-se que o início do notebook costuma receber mais atenção, incluindo células de Markdown e células com resultados de execuções. Isso ocorre por essa parte estar sempre sendo revisitada para a inclusão de novos *imports* e reexecução de partes do notebook. Já o final de notebooks costuma ter mais células de código sem execução, menos células de Markdown, e poucos notebooks selecionados para a análise qualitativa possuíam células de Markdown com conclusões dos resultados das análises, apesar de um número considerável ter definido o que se desejava observar no início. Portanto, recomenda-se dar mais atenção ao final do notebook, para que a qualidade seja mantida do início ao fim.

3. **Abstrair funções, classes e módulos a partir do código e testá-los.** Por mais genérico que seja, o código de uma célula só costuma ser acessível no próprio notebook. Dessa forma, para reusar um determinado código, muitas vezes é necessário copiar e colar sua definição de um arquivo para outro, prejudicando o reuso e levando códigos a evoluírem de maneira independente sem compartilharem melhorias (i.e., é possível que uma versão do código receba uma correção de defeito e a outra receba uma funcionalidade nova). Além disso, a execução de testes automatizados em notebooks pode ser problemática por quebrar a narrativa que está sendo contada pela programação literária. Assim, recomenda-se a abstração de códigos em funções, classes e módulos que possam ser importados por diversos notebooks e testados externamente. Um exemplo de abstração foi apresentado na Seção 1.7.2 deste capítulo, ao apresentar a extração de uma classe de mágicas para um módulo externo. Essa abstração permitiu o reuso das mesmas mágicas em diferentes notebooks, sem a necessidade de copiar e colar definições de um notebook para outro. Por questão de escopo, não foram definidos testes automatizados para o módulo extraído, mas o uso de um módulo externo permite que isso seja feito de forma transparente sem quebrar a narrativa do notebook.
4. **Declarar dependências em arquivos apropriados e fixar a versão de módulos.** O estudo de reprodutibilidade de notebooks observou que muitos notebooks falhavam em executar até fim por falta da instalação de módulos [Pimentel et al. 2021]. Além disso, o estudo identificou que arquivos *requirements.txt* eram menos suscetíveis a falhas do que os outros formatos avaliados, *setup.py* e *Pipfile*. Dessa forma, recomenda-se definir dependências explicitamente em arquivos *requirement.txt* com suas respectivas versões fixas. Arquivos *setup.py* e *Pipfile* também podem ser usados se as pessoas que estão desenvolvendo o projeto considerarem mais apropriados ou com o aumento da maturidade de arquivos *Pipfile* – que ainda eram bem recentes quando o estudo de reprodutibilidade foi feito. Por exemplo, para este capítulo, foi usada a versão 1.1.3 do *pandas*. Para favorecer a reprodutibilidade, um arquivo *requirements.txt* poderia ter a linha `pandas==1.1.3`. Além desta declaração, também seria necessário declarar versões do *IPython* (7.19.0), *numpy* (1.19.2), *scipy* (1.5.2), *scikit-learn* (0.24.2), *fuzzywuzzy* (0.18.0), *matplotlib* (3.3.2), *seaborn* (0.11.0), *ipywidgets* (7.5.1), entre outras.
5. **Usar um ambiente limpo para testar dependências e verificar se todas estão declaradas corretamente.** Muitos projetos presumem um conjunto de dependências no sistema e não definem versões explícitas para elas em arquivos de dependências. O estudo inicial de reprodutibilidade [Pimentel et al. 2019] observou que instalar dependências em ambientes limpos falhou mais do que usar ambientes com anaconda, um gerenciador de pacotes que vem com uma distribuição Python e diversas bibliotecas científicas instaladas. De forma semelhante, a extensão do estudo [Pimentel et al. 2021] observou que usar ambientes inchados de dependências falhavam menos por *ImportError* e mais por atualizações em módulos. Dessa forma, recomenda-se configurar um ambiente limpo para testar as dependências dos notebooks antes de suas publicações, para verificar se todas as dependências estão declaradas corretamente.
6. **Colocar importações de módulos no início do notebook.** Essa recomendação é semelhante à recomendação da PEP 8 de se colocar *imports* no topo de arquivos [van Rossum et al. 2001]. Em notebooks, além dessa recomendação ajudar a organizá-los, pode ajudar na verificação de *imports* discutida anteriormente. Se todos os coman-

dos de *import* estiverem juntos, a verificação do ambiente pode ser feita de forma simplificada executando apenas o início do notebook. Note, entretanto, que isso não garante a reprodutibilidade do notebook, visto que versões diferentes de módulos podem permitir a execução do comando de *import*, mas gerar resultados diferentes.

7. **Usar caminhos relativos para acessar dados no repositório.** O estudo de reprodutibilidade [Pimentel et al. 2021] observou que o acesso a arquivos também foi uma causa comum de erros. Em muitas situações, isso aconteceu por arquivos de dados não estarem disponíveis no repositório; e em outras, ocorreu por notebooks tentarem acessar arquivos usando caminhos absolutos no sistema de arquivos. Usar caminhos relativos e disponibilizar dados de análise podem minimizar esse problema.

8. **Re-executar notebooks do início ao fim antes de enviá-los para algum repositório.** Como observado na análise [Pimentel et al. 2021], muitos notebooks possuem células fora de ordem e saltos na execução. Essas características podem induzir a problemas por estados ocultos de execução e reduzir a reprodutibilidade de notebooks. A Figura 1.5 apresenta três situações que ocorrem durante o desenvolvimento de notebooks que podem gerar problemas de estados ocultos. A re-execução de notebooks do início ao fim ajuda a restaurar contadores de execução e a verificar a existência de estados ocultos. Dessa forma, recomenda-se que a prática seja adotada antes do envio de notebooks a repositórios.

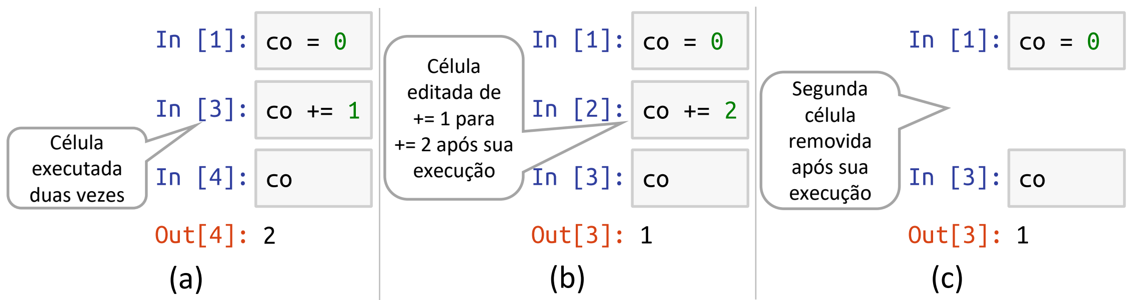


Figura 1.5. Três tipos de estados ocultos: (a) Re-execução; (b) célula editada; (c) célula removida (adaptado de [Pimentel et al. 2021])

Seguir algumas dessas boas práticas manualmente pode exigir um esforço cognitivo de estar sempre organizando o notebook. Algumas ferramentas foram propostas para diminuir esse esforço e melhorar a qualidade e reprodutibilidade de notebooks: Julynter [Pimentel et al. 2021], NBSafety [Macke et al. 2020], ReproZip [Chirigati et al. 2016], nbgather [Head et al. 2019] e Osiris [Wang et al. 2020].

Julynter [Pimentel et al. 2021] é uma extensão para Jupyter Lab feita com o objetivo de oferecer *linting* para notebooks. Essa ferramenta utiliza informações já coletadas durante a execução pelo kernel do IPython e as combina com análises estáticas de código-fonte para exibir recomendações em tempo de desenvolvimento para quem estiver escrevendo o notebook. As recomendações são diretamente baseadas nas recomendações descritas anteriormente, e sete das oito boas práticas estão cobertas pelas recomendações do Julynter. Apesar de suportar diversas recomendações, muitas não são completas o suficiente para todo tipo de problema. Por exemplo, o Julynter consegue detectar dependências

expressas por *imports* no Python, mas não consegue detectar o uso de programas externos. Além disso, o Julynter só pode fazer recomendações durante o desenvolvimento. Dessa forma, não é muito útil resolver a ordem de execução após fechar a sessão para auxiliar a reprodução ou para ajudar a execução de notebooks desenvolvidos por outras pessoas em outros momentos.

NBSafety [Macke et al. 2020] funciona de forma semelhante ao Julynter, coletando execuções de células durante o desenvolvimento, analisando o código estaticamente e apresentando recomendações para resolver problemas de estados ocultos e ordem de execução de células. Diferente do Julynter que possui recomendações variadas para diversas situações, NBSafety direciona as recomendações apenas a essas questões da execução e apresenta mais detalhes sobre elas. Além disso, NBSafety funciona tanto no Jupyter Notebook quanto no Jupyter Lab, enquanto que Julynter só funciona no Jupyter Lab.

Para tratar a questão de dependências, o ReproZip [Chirigati et al. 2016] captura dependências automaticamente durante a execução. O ReproZip é capaz de capturar não só bibliotecas escritas em Python, mas também programas externos e dados usados pelos notebooks. Após a execução de um notebook, o ReproZip cria pacotes com essas dependências que podem ser usados para reprodução em outros ambientes.

Em relação à ordem de execução e à presença de estados ocultos, tanto nbgather [Head et al. 2019] quanto o kernel do noWorkflow [Pimentel 2021] são capazes de coletar todas as execuções de células que ocorrem durante o desenvolvimento para a reconstrução de um notebook organizado posteriormente. Essas ferramentas também possuem recursos para limpar a execução, removendo células do histórico que não contribuem para o resultado final. A nbgather realiza essa operação de forma estática, observando nomes que aparecem em cada célula executada. Já o noWorkflow faz a coleta de forma dinâmica, identificando dependências reais que existem entre variáveis.

Por fim, muitas vezes essas ferramentas não foram usadas durante o desenvolvimento e é necessário reproduzir notebooks seguindo a ordem correta. Para resolver esses problemas, Osiris [Wang et al. 2020] busca reordenar as células de um notebook para uma ordem reprodutível. Para isso, Osiris analisa a ocorrência de definição e uso de nomes em cada célula do notebook e reordena a execução de forma a minimizar a presença de células fora de ordem. Osiris atinge uma taxa de reprodução de 82,23% em notebooks que não possuem problemas de dependência.

1.9. Considerações Finais

A necessidade de transformar dados em informações úteis de forma fácil e rápida tornou a Ciência de Dados um dos tópicos mais relevantes não só em Computação mas também nas mais diversas Ciências. O principal objetivo dessa área de pesquisa interdisciplinar é extrair conhecimento não-trivial a partir de dados brutos. Através de técnicas e algoritmos de mineração de dados e aprendizado de máquina, é possível detectar padrões e obter *insights* valiosos sobre informações múltiplas, de forma com que o valor agregado gerado permita responder perguntas e solucionar problemas. Nesse amplo e relevante contexto, este capítulo apresentou a ferramenta Jupyter com reprodutibilidade para a realização de projetos em Ciência de Dados. As seções foram organizadas por ordem de complexidade, iniciando nos conceitos básicos, detalhando o tratamento de dados necessário para deixá-

los prontos para realizar ciência, a qual se baseia em diferentes métodos de análise e visualização, e finalizando com conceitos avançados de Jupyter e reprodutibilidade.

De fato, notebooks Jupyter são tão flexíveis que permitem construir processamentos muito mais complexos em cima da base abordada neste capítulo. Por exemplo, o artigo [Fangohr et al. 2021] introduz uma edição especial de cinco artigos convidados a partir de apresentações na JupyterCon[11] 2020. Além do primeiro artigo com os criadores do Jupyter, os demais abordam pesquisas em astrofísica e geociências, bem como aplicações relevantes de simulação matemática e visualização de estruturas moleculares.

Com sua grande amplitude de utilização, vários aspectos sobre Jupyter e Ciência de Dados não foram abordados neste capítulo. Um aspecto fundamental para qualquer ciência é a atualização de dados que interfere em cálculos, análises e resultados. Atualmente, existem trabalhos em andamento para tornar notebooks mais reativos e interativos, assim como planilhas de cálculo [Perkel 2021a]. Outra frente de trabalho busca transformar notebooks em aplicativos Web que sejam leves, interativos, de código aberto e reprodutíveis [Clarke et al. 2021]. Os chamados *Appyters* constituem workflows reutilizáveis baseados na Web e incluem pipelines de aprendizado de máquina, entre outras funções destinadas às áreas de biomedicina e bioinformática. Outra questão relevante sempre presente em pesquisa orientada a dados inclui ética no compartilhamento e processamento de dados. Como exemplo, o trabalho [Peisert 2021] discute como ambientes de execução confiável (do inglês *trusted execution environments*) permitem dados sensíveis serem analisados sem ter de confiar na administração do sistema ou provedora de computadores.

Jupyter também tem muito a contribuir para a educação em seus diversos níveis. Por exemplo, o artigo [Willis et al. 2020] mostra como notebooks Jupyter podem auxiliar a desenvolver habilidades de comunicação através da escrita científica. De maneira mais ampla, o artigo [Manzoor et al. 2020] aponta que corrigir e avaliar (dar notas) para trabalhos escolares entregues em notebooks Jupyter são tarefas complexas de serem realizadas manualmente. Então, apresenta a implementação de um autocorretor para tais trabalhos, com as vantagens de feedback imediato para estudantes e diminuição considerável da carga de trabalho docente.

Os benefícios do Jupyter para educação são muito claros. Porém, como a sua utilização para fins educacionais ainda está nos anos iniciais, existem muitos pontos a serem melhorados, como discutido em [Johnson 2020]. Entre as principais críticas apontadas em tal estudo estão comportamento imprevisível, dificuldade de replicação, permissibilidade de práticas fracas de programação, e abertura para possíveis problemas de segurança. Este capítulo está em sincronia com as duas primeiras críticas ao discutir conceitos de reprodutibilidade e boas práticas na seção anterior.

Disponibilidade de dados e código. O código-fonte do capítulo inteiro, bem como o conjunto de dados utilizado estão disponíveis no repositório `https://github.com/opgabriel/jai2021-jupyter`.

Agradecimentos. Agradecemos a Juliana Freire, Leonardo Murta e Vanessa Braganholo por colaborações em trabalhos anteriores de onde obtivemos dados sobre o uso de Note-

[11] https://jupytercon.com

books e boas práticas para reprodutibilidade e qualidade. Este trabalho foi parcialmente financiado por Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq, Fundação de Amparo à Pesquisa do Estado de Minas Gerais – FAPEMIG e Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES.

Referências

[Chirigati et al. 2016] Chirigati, F., Rampin, R., Shasha, D., and Freire, J. (2016). Reprozip: Computational reproducibility with ease. In *SIGMOD*, pages 2085–2088.

[Clarke et al. 2021] Clarke, D. J. et al. (2021). Appyters: Turning jupyter notebooks into data-driven web apps. *Patterns*, 2(3):100213.

[Fangohr et al. 2021] Fangohr, H., Kluyver, T., and DiPierro, M. (2021). Jupyter in computational science. *Computing in Science Engineering*, 23(2):5–6.

[Head et al. 2019] Head, A., Hohman, F., Barik, T., Drucker, S. M., and DeLine, R. (2019). Managing messes in computational notebooks. In *CHI*, pages 1–12.

[Igual and Seguí 2017] Igual, L. and Seguí, S. (2017). *Introduction to Data Science - A Python Approach to Concepts, Techniques and Applications*. Undergraduate Topics in Computer Science. Springer.

[Johnson 2020] Johnson, J. W. (2020). Benefits and pitfalls of jupyter notebooks in the classroom. In Khazanchi, D., Siy, H. P., Grispos, G., and Setor, T. K., editors, *SIGITE*, pages 32–37.

[Kluyver et al. 2016] Kluyver, T. et al. (2016). Jupyter notebooks - a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90.

[Lewine 1991] Lewine, D. (1991). *POSIX programmers guide*. "O'Reilly Media, Inc.".

[Macke et al. 2020] Macke, S., Gong, H., Lee, D. J.-L., Head, A., Xin, D., and Parameswaran, A. (2020). Fine-grained lineage for safer notebook interactions. *arXiv preprint arXiv:2012.06981*.

[Manzoor et al. 2020] Manzoor, H., Naik, A., Shaffer, C. A., North, C., and Edwards, S. H. (2020). Auto-grading jupyter notebooks. In *SIGCSE*, pages 1139–1144.

[Oliveira et al. 2020] Oliveira, G. P., Silva, M. O., Seufitelli, D. B., Lacerda, A., and Moro, M. M. (2020). Detecting collaboration profiles in success-based music genre networks. In *ISMIR*, pages 726–732.

[Peisert 2021] Peisert, S. (2021). Trustworthy scientific computing. *Communications of the ACM*, 64(5):18–21.

[Perkel 2018] Perkel, J. M. (2018). Why Jupyter is data scientists' computational notebook of choice. *Nature*, 563:145–146.

[Perkel 2021a] Perkel, J. M. (2021a). Reactive, reproducible, collaborative: computational notebooks evolve. *Nature*, 593:156–157.

[Perkel 2021b] Perkel, J. M. (2021b). Ten computer codes that transformed science. *Nature*, 589:344–348.

[Pimentel 2021] Pimentel, J. F. (2021). *Provenance from Script*. PhD thesis, Universidade Federal Fluminense.

[Pimentel et al. 2019] Pimentel, J. F., Murta, L., Braganholo, V., and Freire, J. (2019). A large-scale study about quality and reproducibility of jupyter notebooks. In *MSR*, pages 507–517.

[Pimentel et al. 2021] Pimentel, J. F., Murta, L., Braganholo, V., and Freire, J. (2021). Understanding and improving the quality and reproducibility of jupyter notebooks. *Empirical Software Engineering*, 26(4):65.

[Shen 2014] Shen, H. (2014). Interactive notebooks: Sharing the code. *Nature*, 515(7525):151–152.

[Skiena 2017] Skiena, S. (2017). *The Data Science Design Manual*. Texts in Computer Science. Springer.

[van Rossum et al. 2001] van Rossum, G., Warsaw, B., and Coghlan, N. (2001). Pep 8: style guide for python code. `https://www.python.org/dev/peps/pep-0008/`. Accessed: 2019-10-01.

[Wang et al. 2020] Wang, J., Tzu-Yang, K., Li, L., and Zeller, A. (2020). Assessing and Restoring Reproducibility of Jupyter Notebooks. In *ASE*, pages 138–149.

[Willis et al. 2020] Willis, A., Charlton, P., and Hirst, T. (2020). Developing students' written communication skills with jupyter notebooks. In *SIGCSE*, pages 1089–1095.