

Projetos de Ciência de Dados com Python

**Abordagem de estudo de caso para a criação de projetos
de ciência de dados bem-sucedidos usando Python,
pandas e scikit-learn**

Stephen Klosterman

Packt

Novatec

Copyright © Packt Publishing 2019. First published in the english language under the title ‘Data Science Projects with Python – (9781838551025)’

Copyright © Packt Publishing 2019. Publicação original em inglês intitulada ‘Data Science Projects with Python – (9781838551025)’

Esta tradução é publicada e vendida com a permissão da Packt Publishing.

© Novatec Editora Ltda. [2019].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Aldir Coelho Corrêa da Silva

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN do impresso: 978-65-86057-10-2

ISBN do ebook: 978-65-86057-11-9

Histórico de impressões:

Maio/2020 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

GRA20200427

Exploração e limpeza de dados

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Executar operações básicas em Python
- Descrever o contexto empresarial dos dados do estudo de caso e sua adequação à tarefa
- Executar operações de limpeza de dados
- Examinar sínteses estatísticas e visualizar os dados do estudo de caso
- Implementar a codificação one-hot (one-hot encoding) em variáveis categóricas

Este capítulo é uma introdução às operações básicas em Python e lhe mostrará como executar operações relacionadas a dados tal qual a verificação e a limpeza de dados, a conversão de tipos de dados, a verificação de sínteses estatísticas e muito mais.

Introdução

A maioria das empresas possui uma grande riqueza de dados sobre suas operações e clientes. Gerar relatórios sobre esses dados na forma de diagramas descritivos, gráficos e tabelas é uma boa maneira de entender o estado atual dos negócios. No entanto, para fornecermos orientações quantitativas para estratégias empresariais futuras, é preciso ir um pouco além. É nesse momento que as práticas de machine learning e modelagem preditiva entram em cena. Neste livro, mostraremos como passar de análises descritivas a orientações concretas para operações futuras usando modelos preditivos.

Para atingir esse objetivo, apresentaremos algumas das ferramentas de machine learning mais usadas com o emprego de Python e muitos de seus pacotes. Você também conhecerá as habilidades práticas necessárias para executar projetos bem-sucedidos: a curiosidade ao examinar dados e a comunicação com o cliente. O tempo gasto examinando um dataset detalhadamente e verificando criticamente se ele atende com precisão à finalidade pretendida não é desperdício. Aqui você aprenderá várias técnicas para avaliar a qualidade dos dados.

Neste capítulo, após nos familiarizarmos com as ferramentas básicas para a exploração de dados, discutiremos alguns cenários de trabalho típicos mostrando como eles podem ser recebidos. Em seguida, começaremos uma exploração completa do dataset de um estudo de caso e o ajudaremos a aprender como descobrir possíveis problemas, para que, quando você estiver pronto para modelar, prossiga com confiança.

Python e o sistema de gerenciamento de pacotes Anaconda

Neste livro, usaremos a linguagem de programação Python, uma linguagem de ponta para o trabalho com ciência de dados, sendo uma das linguagens de programação cujo crescimento está ocorrendo com mais rapidez. Uma razão normalmente citada para a popularidade do Python é que ele é fácil de aprender. Se você tem experiência no uso de Python, ótimo; porém, se tem experiência no uso de outras linguagens, como C, Matlab ou R, não terá dificuldades para usar Python. É preciso conhecer as estruturas gerais da programação de computadores para aproveitar ao máximo este livro. Alguns exemplos dessas estruturas seriam os loops `for` e as instruções `if` que guiam o **fluxo de controle** de um programa. Independentemente da linguagem que você tem usado, provavelmente está familiarizado com essas estruturas e também as encontrará em Python.

Um recurso-chave do Python, que é diferente de algumas outras linguagens, é o fato de ele ser indexado a partir de zero; em outras palavras, o primeiro elemento de uma coleção ordenada tem índice igual a 0. O Python também dá suporte à indexação negativa, em que o índice 1 referencia o último elemento de uma coleção ordenada e os índices negativos são contados de trás para a frente a partir do fim. O operador slice, `:`, pode ser usado para a seleção de múltiplos elementos de uma coleção ordenada dentro de um intervalo, começando do início ou indo até o fim da coleção.

A indexação e o operador slice

Demonstraremos aqui como a indexação e o operador slice funcionam. Para termos algo para indexar, criaremos uma **lista**, ou seja, uma coleção ordenada **mutável** que pode conter qualquer tipo de dado, inclusive tipos numéricos e strings. A palavra “mutável” significa apenas que os elementos da lista podem ser alterados após serem atribuídos. Para criar os números de nossa lista, que serão inteiros consecutivos, usaremos a função Python interna `range()`. Tecnicamente, a função `range()` cria um **iterador** que converteremos em uma lista usando a função `list()`, mas não se preocupe com esse detalhe. O screenshot a seguir mostra uma lista básica sendo exibida no console:

```
>>> example_list = list(range(1,6))
>>> example_list
[1, 2, 3, 4, 5]
>>> example_list[0]
1
>>> example_list[-1]
5
>>> example_list[-2]
4
>>> example_list[:3]
[1, 2, 3]
>>> example_list[0] = 'a string'
>>> example_list
['a string', 2, 3, 4, 5]
>>> █
```

Figura 1.1 – Criação e indexação de lista.

Algumas coisas que devemos observar na *Figura 1.1*: o ponto final do intervalo está aberto tanto para a indexação de slice quanto para a função `range()`, enquanto o ponto inicial está fechado. Em outras palavras, observe que, quando especificamos o início e o fim de `range()`, o ponto final 6 não foi incluído no resultado, mas o ponto inicial 1 foi. Da mesma forma, ao indexar a lista com o slice `[:3]`, incluímos todos os elementos com índice até 3, porém sem incluir esse último.

Fizemos referência às coleções ordenadas, mas o Python também inclui coleções não ordenadas. Um dos mais importantes chama-se **dicionário**. Um dicionário é uma coleção não ordenada de pares **chave:valor**. Em vez de procurar os valores de um dicionário por índices de números inteiros, você deve procurá-los por chaves, que podem ser números ou strings. Podemos criar um dicionário usando o símbolo de chaves (`{}`) e separando os pares **chave:valor** com vírgulas. O screenshot a seguir é um exemplo de como criaríamos um dicionário com contagens de frutas – ele exibe o número de maçãs e, em seguida, adiciona um novo tipo de fruta e sua contagem:

```
>>> example_dict = {'apples':5, 'oranges':8}
>>> example_dict['apples']
5
>>> example_dict['bananas'] = 13
>>> example_dict
{'apples': 5, 'oranges': 8, 'bananas': 13}
>>>
```

Figura 1.2 – Exemplo de dicionário.

Há várias outras características que distinguem a linguagem Python e só queremos mostrar algumas aqui, sem entrar em muitos detalhes. Na verdade, provavelmente você usará pacotes como o **pandas** (*pandas*) e o **NumPy** (*numpy*) para quase toda a manipulação de dados que fará em Python. O NumPy fornece cálculo numérico rápido com arrays e matrizes, enquanto o pandas oferece vários recursos de preparação e exploração de dados com tabelas chamadas **DataFrames**. No entanto, é bom estar familiarizado com alguns aspectos básicos do Python – a linguagem que dá suporte a tudo isso. Por exemplo, a indexação funciona no NumPy e no pandas da mesma maneira que funciona em Python.

Um dos pontos fortes do Python é que ele é open source e tem uma comunidade de desenvolvedores ativa criando ferramentas extraordinárias. Usaremos várias dessas ferramentas neste livro. Um possível problema do uso de pacotes open source de diferentes colaboradores são as dependências entre os diversos pacotes. Por exemplo, se você quiser instalar o pandas, ele pode depender de uma versão específica do NumPy, que pode ou não ter sido instalada. Os sistemas de gerenciamento de pacotes facilitam a vida nesse aspecto. Quando você instalar um novo pacote por intermédio do sistema de gerenciamento de pacotes, ele assegurará que todas as dependências sejam atendidas. Se não forem, você será solicitado a fazer um upgrade ou a instalar novos pacotes conforme necessário.

Neste livro, usaremos o sistema de gerenciamento de pacotes **Anaconda**, que você precisa instalar. Aqui só utilizaremos Python, mas também é possível executar R com o Anaconda.

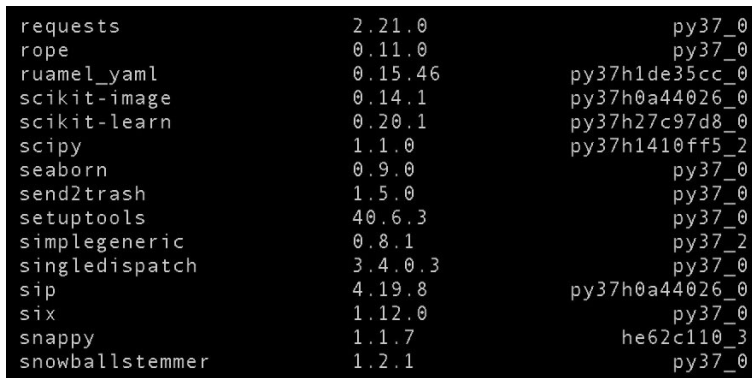
Nota: Ambientes – Se você já instalou e está usando o Anaconda, é recomendável criar um novo ambiente Python 3.x para o livro. Os ambientes são como instalações separadas, em que o conjunto de pacotes instalado pode ser diferente, assim como a versão do Python. Eles são úteis para o desenvolvimento de projetos que precisem ser implantados em diferentes versões da linguagem. Para obter mais informações, consulte <https://conda.io/docs/user-guide/tasks/manage-environments.html>.

Exercício 1: Examinando o Anaconda e familiarizando-se com o Python

Neste exercício, você examinará os pacotes de sua instalação do Anaconda e manipulará um fluxo de controle básico e algumas estruturas de dados do Python, inclusive um loop `for`, e as estruturas `dict` e `list`. Isso confirmará que concluiu as etapas de instalação do prefácio e mostrará como a sintaxe e as estruturas de dados do Python podem ser um pouco diferentes das de outras linguagens de programação que você conhece. Execute as etapas a seguir para fazer o exercício:

Nota: O arquivo do código deste exercício pode ser encontrado aqui: <http://bit.ly/20yag1h>.

1. Abra um Terminal, se estiver usando o macOS ou o Linux, ou uma janela de prompt de comando no Windows. Digite `conda list` na linha de comando. Você deve ver uma saída semelhante à seguinte:

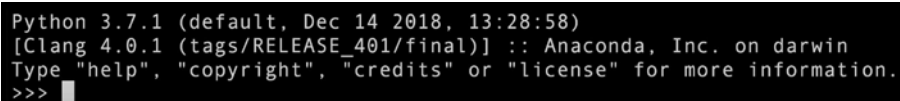


```
requests                2.21.0                py37_0
rope                    0.11.0                py37_0
ruamel_yaml             0.15.46               py37h1de35cc_0
scikit-image            0.14.1                py37h0a44026_0
scikit-learn            0.20.1                py37h27c97d8_0
scipy                   1.1.0                 py37h1410ff5_2
seaborn                 0.9.0                 py37_0
send2trash              1.5.0                 py37_0
setuptools              40.6.3                py37_0
simplegeneric            0.8.1                 py37_2
singledispatch          3.4.0.3               py37_0
sip                     4.19.8                py37h0a44026_0
six                     1.12.0                py37_0
snappy                  1.1.7                 he62c110_3
snowballstemmer         1.2.1                 py37_0
```

Figura 1.3 – Seleção de pacotes de `conda list`.

É possível ver todos os pacotes que estão instalados em seu ambiente. Veja quantos pacotes já vêm com a instalação padrão do Anaconda! Estão incluídos todos os pacotes necessários para a leitura do livro. No entanto, é fácil e descomplicado instalar novos pacotes e atualizar os existentes com o Anaconda; essa é uma das principais vantagens de um sistema de gerenciamento de pacotes.

2. Digite `python` no Terminal para abrir um interpretador Python de linha de comando. Você deve obter uma saída semelhante à seguinte:



```
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figura 1.4 – Linha de comando Python.

Você verá algumas informações sobre sua versão do Python, assim como o prompt de comando da linguagem (`>>>`). Ao digitar algo depois do prompt, estará escrevendo código Python.

Nota: Usaremos o Jupyter Notebook neste livro, mas um dos objetivos do exercício é percorrermos as etapas básicas de criação e execução de programas Python no prompt de comando.

3. Crie um loop `for` no prompt de comando para exibir os valores de 0 a 4 usando o código a seguir:

```
for counter in range(5):  
...     print(counter)  
...
```

Quando você pressionar *Enter* após ver (...) no prompt, deve obter esta saída:

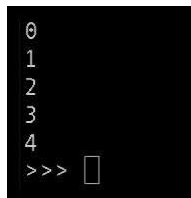


Figura 1.5 – Saída de um loop for na linha de comando.

Observe que, em Python, a abertura do loop `for` é seguida por dois pontos e **o corpo do loop requer recuo**. É normal o uso de quatro espaços no recuo de um bloco de código. Aqui, o loop `for` está exibindo os valores retornados pelo iterador `range()`, que os acessou repetidamente usando a variável `counter` com a palavra-chave `in`.

Nota: Para ver mais detalhes sobre as convenções dos códigos Python, acesse a página a seguir: <https://www.python.org/dev/peps/pep-0008/>.

Agora, retornaremos ao nosso exemplo de dicionário. A primeira etapa é criar o dicionário.

4. Crie um dicionário de frutas (`apples` [maçãs], `oranges` [laranjas] e `bananas` [bananas]) usando o código a seguir:

```
example_dict = {'apples':5, 'oranges':8, 'bananas':13}
```

5. Converta o dicionário em uma lista usando a função `list()`, como mostrado neste fragmento de código:

```
dict_to_list = list(example_dict)  
dict_to_list
```


Quando você executar o código anterior, deve ver a seguinte saída:

```
['apples', 'oranges', 'bananas']
```

Figura 1.6 – Chaves do dicionário convertidas em uma lista.

Observe que, quando essa operação é executada, ao examinarmos o conteúdo, vemos que só as chaves do dicionário foram capturadas. Se quiséssemos os valores, teríamos de especificar isso com o método `.values()` da lista. Observe também que as chaves da lista estão na mesma ordem em que foram escritas na criação do dicionário. No entanto, isso não é garantido, já que os dicionários são tipos de coleção não ordenada.

Algo conveniente que você pode fazer com as listas é acrescentar outras listas a elas com o operador `+`. Como exemplo, na próxima etapa combinaremos a lista de frutas existente com uma lista que só contém mais um tipo de fruta, sobrepondo a variável que armazena a lista original.

6. Use o operador `+` para combinar a lista de frutas existente com uma nova lista contendo apenas uma fruta (`pears` [peras]):

```
dict_to_list = dict_to_list + ['pears']  
dict_to_list
```

```
['apples', 'oranges', 'bananas', 'pears']
```

Figura 1.7 – Aumentando uma lista.

E se quiséssemos classificar nossa lista de tipos de frutas?

O Python fornece uma função `sorted()` interna que pode ser usada para esse fim; ela retornará uma versão classificada da entrada. Em nosso caso, isso significa que a lista de tipos de frutas será classificada alfabeticamente.

7. Classifique a lista de frutas em ordem alfabética usando a função `sorted()`, como mostrado no fragmento a seguir:

```
sorted(dict_to_list)
```

Quando você executar o código anterior, deve ver a seguinte saída:

```
['apples', 'bananas', 'oranges', 'pears']
```

Figura 1.8 – Classificando uma lista.

Por enquanto, já vimos o suficiente de Python. Vamos mostrar-lhe como executar o código deste livro, logo, seu conhecimento de Python deve melhorar ao avançarmos.

Nota: À medida que você aprender mais e inevitavelmente quiser fazer novas experiências, é recomendável consultar a documentação: <https://docs.python.org/3/>.

Diferentes tipos de problemas da ciência de dados

Provavelmente você passará grande parte de seu tempo como cientista de dados preparando dados: descobrindo como obtê-los, obtendo-os, examinando-os, verificando se estão corretos e completos e associando-os a outros tipos de dados. O pandas facilitará esse processo. No entanto, se pretende ser um cientista de dados da área de machine learning, terá de dominar a arte e a ciência da **modelagem preditiva**. Isso significa usar um modelo matemático, ou uma formulação matemática idealizada, para aprender que relacionamentos estão ocorrendo dentro dos dados, na esperança de fazer previsões precisas e úteis quando entrarem novos dados.

Para esse fim, normalmente os dados são organizados em uma estrutura tabular, com **características** e uma **variável de resposta**. Por exemplo, se você quisesse prever o preço de uma casa com base em algumas características dela, como a **área** e o **número de quartos**, esses atributos seriam as características, e o **preço da casa** seria a variável de resposta, também chamada de **variável alvo** ou **variável dependente**, enquanto as características podem ser chamadas de **variáveis independentes**.

Se você tivesse um dataset de 1.000 casas incluindo os valores das características e os preços das casas, poderia dizer que possui 1.000 **amostras** de dados **rotulados**, em que os rótulos são os valores conhecidos da variável de resposta: os preços das diferentes casas. Normalmente, a estrutura de dados tabular é organizada de modo que linhas diferentes são amostras distintas, enquanto as características e a resposta ocupam colunas diferentes, junto com outros metadados como os IDs das amostras, conforme mostrado na Figura 1.9.

ID da casa	Área (em m²)	Número de quartos	Preço (\$)
1	1.500	3	200.000
2	2.500	5	600.000
3	2.000	3	500.000

Figura 1.9 – Dados rotulados (os preços das casas são a variável alvo conhecida).

Problema de regressão

Uma vez que você tiver treinado um modelo para aprender o relacionamento entre as características e a resposta usando os dados rotulados, poderá empregá-lo para

fazer previsões para casas das quais não souber o preço, com base nas informações contidas nas características. O objetivo da modelagem preditiva nesse caso é fazer-mos uma previsão que fique próxima do valor real da casa. Já que estamos prevendo um valor numérico em uma escala contínua, isso se chama **problema de regressão**.

Problema de classificação

Por outro lado, se estivéssemos tentando fazer uma previsão qualitativa da casa, para responder a uma pergunta de resposta **sim** ou **não** como “essa casa estará à venda nos próximos cinco anos?” ou “o proprietário não precisará pagar a hipoteca?”, estaríamos resolvendo o que é conhecido como **problema de classificação**. Nesse caso, esperamos dar a resposta afirmativa ou negativa certa. A Figura 1.10 é um diagrama que ilustra como o treinamento de modelos funciona e quais podem ser os resultados dos modelos de regressão ou classificação:

Fase de treinamento do modelo

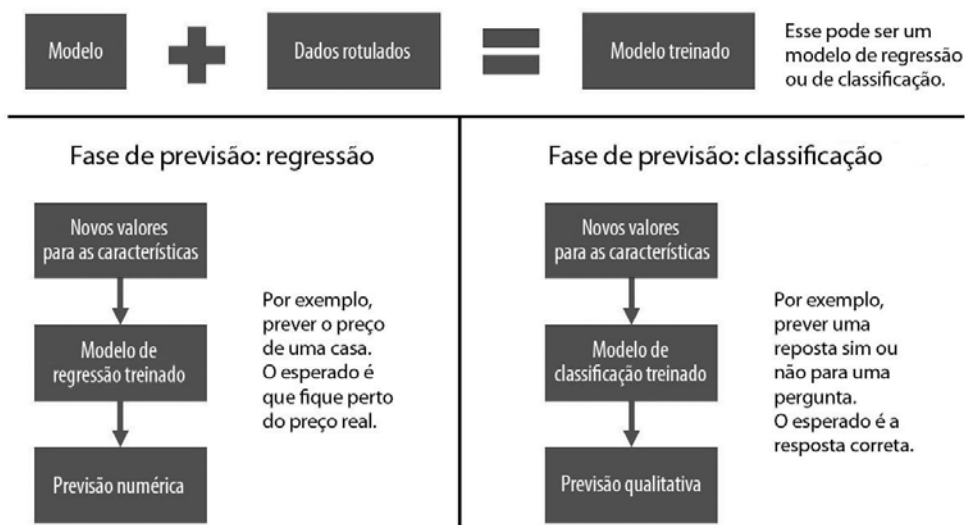


Figura 1.10 – Diagrama de treinamento e previsão de um modelo para a regressão e a classificação.

As tarefas de classificação e regressão são chamadas de **aprendizado supervisionado**, um tipo de problema que depende de dados rotulados. São problemas que demandam a “inspeção” dos valores conhecidos da variável alvo. Por outro lado, também há o **aprendizado não supervisionado**, que está relacionado a perguntas menos limitadas para as quais tentamos encontrar algum tipo de estrutura em um dataset que nem sempre tem rótulos. De um modo geral, qualquer tipo de

problema de matemática aplicada, inclusive de áreas tão variadas quanto **otimização**, **inferência estatística** e **modelagem de séries temporais**, pode ser considerado como uma responsabilidade apropriada para um cientista de dados.

Carregando os dados do estudo de caso com o Jupyter e o pandas

É hora de darmos uma primeira olhada nos dados que usaremos em nosso estudo de caso. A única coisa que faremos nesta seção é verificar se conseguimos carregar os dados corretamente em um **Jupyter Notebook**. As tarefas de examinar os dados e entender o problema que resolveremos com eles virão depois.

O arquivo de dados é uma planilha do Excel chamada `default_of_credit_card_clients__courseware_version_1_13_19.xls`. Recomendamos que você abra assim que puder a planilha no Excel ou no programa de planilhas de sua escolha. Observe o número de linhas e colunas e examine alguns exemplos de valores. Isso o ajudará a saber se conseguiu ou não a carregar corretamente no Jupyter Notebook.

Nota: O dataset pode ser obtido no link a seguir: <http://bit.ly/2HIk5t3>. Essa é uma versão modificada do dataset original, que extraímos do UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: Universidade da Califórnia, School of Information and Computer Science.

O que é um Jupyter Notebook?

Os Jupyter Notebooks são ambientes de codificação interativos que permitem o uso de gráficos e texto inline. São ótimas ferramentas para os cientistas de dados divulgarem e preservarem seus resultados, já que tanto os métodos (código) quanto a mensagem (texto e gráficos) são integrados. O ambiente pode ser considerado como um tipo de página web em que podemos escrever e executar códigos. Na verdade, os Jupyter Notebooks podem ser renderizados como páginas web, e isso ocorre no GitHub. Você pode ver um de nossos exemplos de notebooks em <http://bit.ly/20vndJg>. Examine-o para ter uma ideia do que pode ser feito. Um trecho desse notebook aparece na Figura 1.11, exibindo código, elementos gráficos e texto, que nesse contexto é chamado de **markdown**.

Uma das coisas que é preciso aprender sobre os Jupyter Notebooks é como navegar e fazer edições. Há dois modos disponíveis. Se você selecionar uma célula e pressionar *Enter*, estará no **modo de edição** e poderá editar o texto dessa célula. Se pressionar *Esc*, estará no **modo de comando** e poderá navegar pelo notebook.

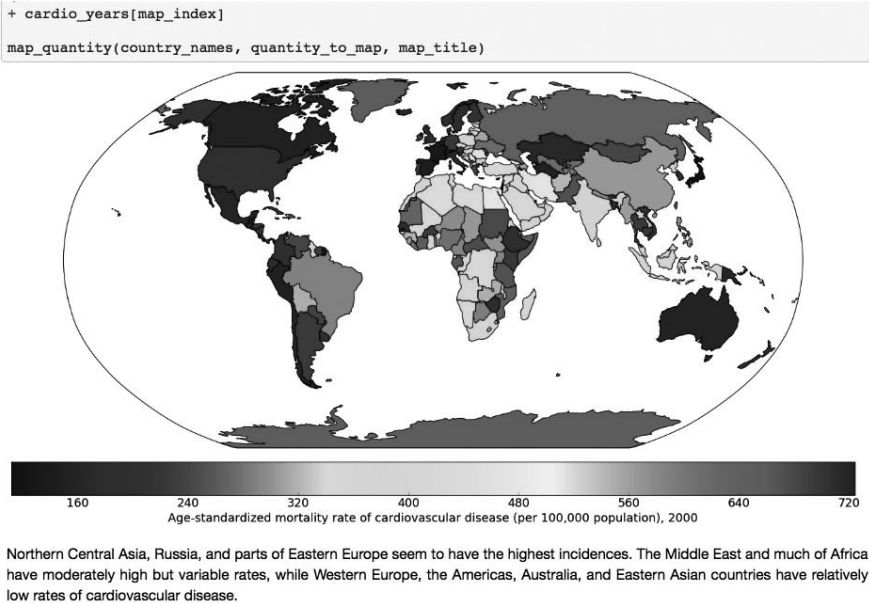


Figura 1.11 – Exemplo de um Jupyter Notebook exibindo código, elementos gráficos e texto markdown.

Quando estiver no modo de comando, poderá usar muitas hotkeys úteis. As setas *para cima* e *para baixo* o ajudarão a selecionar diferentes células e a rolar pelo notebook. Se você pressionar *y* em uma célula selecionada no modo de comando, ele a alterará para uma **célula de código**, na qual o texto é interpretado como código. Pressionar *m* a alterará para uma **célula markdown**. A combinação *Shift + Enter* avalia a célula, renderizando o markdown ou executando o código, conforme o caso.

Nossa primeira tarefa em nosso primeiro Jupyter Notebook será carregar os dados do estudo de caso. Para fazê-lo, usaremos uma ferramenta chamada **pandas**. Não é exagero dizer que o pandas é a ferramenta que se destaca na preparação de dados em Python.

Um DataFrame é uma classe básica do pandas. Falaremos mais sobre o que é uma classe posteriormente, mas você pode considerá-la como um template para uma estrutura de dados, em que a estrutura poderia ser algo como as listas ou os dicionários que já discutimos. No entanto, um DataFrame é muito mais rico em funcionalidade do que essas estruturas. Em muitos aspectos, ele é semelhante às planilhas. Há linhas, que são rotuladas por um índice de linha, e colunas, que em geral recebem rótulos semelhantes a cabeçalhos que podem ser considerados como o índice da coluna. Na verdade, **Index** é um tipo de dado do pandas usado para armazenar os índices de um DataFrame, e as colunas têm o próprio tipo de dado chamado **Series**.

Muitas das coisas que podemos fazer com as planilhas do Excel também podem ser feitas com um DataFrame, como criar tabelas dinâmicas e filtrar linhas. O pandas também inclui uma funcionalidade semelhante à do SQL. Você pode associar diferentes DataFrames, por exemplo. Outra vantagem dos DataFrames é que, uma vez que os dados estão contidos em um deles, temos os recursos de várias funcionalidades do pandas à nossa disposição. A Figura 1.12 é um exemplo de um DataFrame do pandas:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	b730d678-5446	20000	2	2	1	24	2
1	99a3ae70-57a8	120000	2	2	2	26	-1
2	90194006-f94a	90000	2	2	2	34	0
3	19acf9a3-2b01	50000	2	2	1	37	0
4	70d7bc16-a6a4	50000	1	2	1	57	-1

Figura 1.12 – Exemplo de um DataFrame do pandas com um índice de linha inteiro à esquerda e um índice de coluna na forma de strings.

Na verdade, o exemplo da Figura 1.12 são os dados do estudo de caso. Como primeira etapa no Jupyter e no pandas, veremos como criar um Jupyter Notebook e carregar dados com o pandas. Há várias funções convenientes que você pode usar no pandas para explorar seus dados, inclusive `.head()` para ver as primeiras linhas do DataFrame, `.info()` para ver todas as colunas com os tipos de dados, `.columns` para retornar uma lista de nomes de colunas como strings, e outras que aprenderemos nos próximos exercícios.

Exercício 2: Carregando os dados do estudo de caso em um Jupyter Notebook

Agora que você conhece os Jupyter Notebooks, o ambiente no qual escreveremos código, e o pandas, o pacote de preparação de dados, criaremos nosso primeiro Jupyter Notebook. Usaremos o pandas dentro desse notebook para carregar os dados do estudo de caso e os examinaremos brevemente. Execute as etapas a seguir para fazer o exercício:

Nota: Para os Exercícios 2–5 e a Atividade 1, o código e a saída resultante foram carregados em um Jupyter Notebook que pode ser encontrado em <http://bit.ly/2W9cwPH>. Você pode rolar para a seção apropriada dentro do Jupyter Notebook a fim de localizar o exercício ou a atividade que deseja.

1. Abra um Terminal (macOS ou Linux) ou uma janela de prompt de comando (Windows) e digite `jupyter notebook`.

Você verá a interface do Jupyter em seu navegador web. Se ele não abrir automaticamente, copie e cole a URL do terminal para o navegador. Nessa interface, você poderá navegar em seus diretórios começando por aquele em que estava quando iniciou o servidor do notebook.

2. Navegue para um local conveniente para armazenar os materiais deste livro e crie um novo notebook Python 3 a partir do menu **New**, como mostrado na Figura 1.13:



Figura 1.13 – Tela inicial do Jupyter.

3. Faça com que sua primeira célula seja do tipo markdown digitando `m` no modo de comando (pressione `Esc` para entrar no modo de comando) e digite o símbolo de número, `#`, no começo da primeira linha, seguido por um espaço, para o cabeçalho. Crie um título para seu notebook aqui. Nas próximas linhas, insira uma descrição.

Na Figura 1.14 temos um screenshot de um exemplo, incluindo outros tipos de markdown como negrito, itálico e a maneira de escrever texto no estilo código em uma célula markdown:

First Jupyter notebook

Welcome to your first jupyter notebook! The first thing to know about Jupyter notebooks is that there are two kinds of cells. This is a markdown cell.

There are a lot of different ways to mark up the text in markdown cells, including **__bold__** and **italics**.

The next one will be a ``code`` cell.

Figura 1.14 – Célula markdown não renderizada.

Observe que é boa prática criar um título e uma breve descrição para o notebook, a fim de mostrar sua finalidade para os leitores.

4. Pressione *Shift* + *Enter* para renderizar a célula markdown.

Isso também deve criar uma nova célula, que será uma célula de código. Você pode alterá-la para uma célula markdown, já que agora sabe como fazê-lo, pressionando *m*, e transformá-la novamente em uma célula de código pressionando *y*. Sabemos que é uma célula de código porque estamos vendo `In []:` próximo a ela.

5. Digite `import pandas as pd` na nova célula, como mostrado no screenshot a seguir:

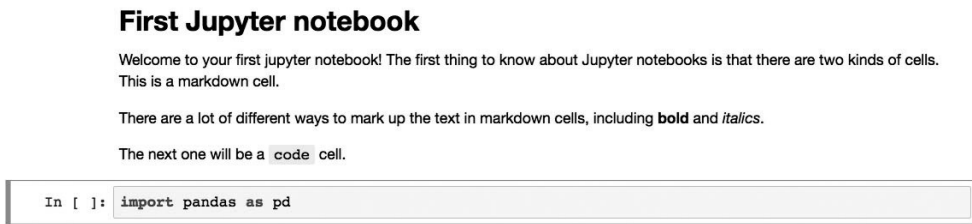


Figura 1.15 – Célula markdown e célula de código renderizadas.

Quando você executar essa célula, a biblioteca pandas será carregada em seu ambiente de computação. É comum a importação com “as” para que seja criado um alias para a biblioteca. Agora usaremos o pandas para carregar o arquivo de dados. Ele está no formato do Microsoft Excel, logo, podemos usar `pd.read_excel`.

Nota: Para obter mais informações sobre todas as opções possíveis para `pd.read_excel`, consulte a documentação a seguir: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html.

6. Importe o dataset, que está no formato do Excel, como um DataFrame usando o método `pd.read_excel()`, como mostrado neste fragmento:

```
df = pd.read_excel('../Data/default_of_credit_card_clients_courseware_
version_1_21_19.xls')
```

Observe que você precisa apontar o leitor do Excel para o local onde o arquivo está localizado. Se ele estiver no mesmo diretório de seu notebook, é possível inserir apenas o nome do arquivo. O método `pd.read_excel` carregará o arquivo do Excel em um `DataFrame`, que chamamos de `df`. Agora o poder do pandas está disponível para nós.

Faremos algumas verificações rápidas nas próximas etapas. Em primeiro lugar, os números de linhas e colunas coincidem com o que vemos ao examinar o arquivo no Excel?

7. Use o método `.shape` para examinar o número de linhas e colunas, como mostrado na linha de código a seguir:

```
df.shape
```

Quando você executar a célula, verá a seguinte saída:

```
In [3]: df.shape
Out[3]: (30000, 25)
```

Figura 1.16 – Verificando a forma de um `DataFrame`.

Ela deve coincidir com o visto na planilha. Caso contrário, será preciso inspecionar as diversas opções de `pd.read_excel` para ver se é necessário ajustar algo.

Com esse exercício, carregamos com sucesso nosso dataset no Jupyter Notebook. Você também pode examinar os métodos `.info()` e `.head()`, que fornecerão informações sobre todas as colunas e mostrarão as primeiras linhas do `DataFrame`, respectivamente. Agora você já está com seus dados no pandas.

Como observação final, embora já deva ter ficado claro, é bom ressaltar que, se você definir uma variável em uma única célula de código, ela ficará disponível em outras células de código dentro do notebook. As células de código de um notebook compartilham o escopo contanto que o kernel esteja sendo executado, como mostrado no screenshot a seguir:

```
In [4]: a = 5
In [5]: a
Out[5]: 5
```

Figura 1.17 – Variável em escopo entre células.

Familiarizando-se com os dados e executando sua limpeza

Suponhamos que estivéssemos examinando os dados pela primeira vez. No trabalho como cientista de dados, há vários cenários a partir dos quais você poderia receber o dataset, por exemplo:

1. Você criou a consulta SQL que gerou os dados.
2. Um colega criou uma consulta SQL para você, com sua participação.
3. Um colega que sabe sobre os dados os forneceu, sem que você pedisse.
4. Você recebeu um dataset sobre o qual pouco sabe.

Nos casos 1 e 2, você participou da geração/extração dos dados. Nesses cenários, entendeu o problema da empresa e encontrou os dados necessários com a ajuda de um engenheiro de dados ou fez sua própria pesquisa e projetou a consulta SQL que os gerou. Com frequência, principalmente quando ganhamos mais experiência na função de cientista de dados, a primeira etapa é reunir-se com o sócio da empresa para entender e aperfeiçoar a definição matemática do problema. Em seguida, você desempenharia um papel-chave na definição do conteúdo do dataset.

Mesmo se você tiver um nível relativamente alto de familiaridade com os dados, sua exploração e a verificação de **sínteses estatísticas** de diferentes variáveis ainda será uma importante primeira etapa. Essa etapa o ajudará a selecionar boas características ou lhe dará ideias de como gerar características novas. No entanto, no terceiro e no quarto casos, dos quais você não participou ou tinha pouco conhecimento sobre os dados, a exploração é ainda mais importante.

Outra etapa inicial importante do processo de ciência de dados é examinar o **dicionário de dados**, o qual, como o termo sugere, é um documento que explica o que o proprietário dos dados acha que deve ser incluído no conjunto, como as definições dos rótulos das colunas. É função do cientista de dados percorrer o conjunto com cuidado para verificar se essas impressões coincidem com o que de fato existe nos dados. Nos casos 1 e 2, provavelmente você terá de criar por conta própria o dicionário de dados, que deve ser considerado documentação essencial do projeto. Nos casos 3 e 4, deve tentar obter o dicionário se possível.

Os dados de estudo de caso que usaremos neste livro são basicamente semelhantes aos do caso 3.

O problema da empresa

Nosso cliente é uma empresa de cartão de crédito. Eles nos trouxeram um dataset que inclui dados demográficos e dados financeiros recentes (últimos seis meses) de uma amostra de 30.000 titulares de contas. Esses dados estão no nível de conta de crédito; em outras palavras, há uma linha para cada conta (você deve sempre esclarecer qual é a definição de linha, em um dataset). As linhas são rotuladas de acordo com se no mês seguinte ao período de dados histórico de seis meses um proprietário de conta ficou inadimplente, ou seja, não fez o pagamento mínimo.

Objetivo

Seu objetivo é desenvolver um modelo que preveja se uma conta ficará inadimplente no próximo mês, de acordo com dados demográficos e históricos. Posteriormente no livro, discutiremos a aplicação prática do modelo.

Os dados já estão preparados e um dicionário de dados está disponível. O dataset fornecido com o livro, `default_of_credit_card_clients__courseware_version_1_21_19.xls`, é uma versão modificada do dataset do UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>. Examine essa página web, que inclui o dicionário de dados.

Nota: O dataset original foi obtido no UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: Universidade da Califórnia, School of Information and Computer Science. Neste livro, modificamos o dataset para que atenda a nossos objetivos. O dataset modificado pode ser encontrado aqui: <http://bit.ly/2HIk5t3>.

Etapas da exploração de dados

Agora que conhecemos o problema da empresa e temos uma ideia do que os dados devem conter, podemos comparar essas impressões com o que estamos vendo realmente nos dados. Nossa função ao explorar dados é percorrê-los tanto diretamente quanto usando resumos numéricos e gráficos e considerar criticamente se fazem sentido e correspondem ao que nos foi dito sobre eles. Algumas etapas úteis para a exploração de dados são:

1. Saber quantas colunas os dados contêm.
Podem ser de características, resposta ou metadados.
2. Quantas linhas (amostras).
3. Que tipos de características existem. Quais são **categóricas** e quais são **numéricas**?

Os valores das características categóricas pertencem a classes discretas como “sim”, “não” ou “talvez”. Normalmente as características numéricas pertencem a uma escala numérica contínua, como as quantias em dólares.

4. Qual é a aparência dos dados segundo essas características.
Para saber isso, você pode examinar o intervalo de valores em características numéricas ou a frequência de classes diferentes em características categóricas, por exemplo.
5. Há dados faltando?

Já respondemos às perguntas 1 e 2 da seção anterior; há 30.000 linhas e 25 colunas. Para responder ao resto das perguntas no próximo exercício, usaremos a ferramenta `pandas` como guia. Começaremos verificando a integridade básica dos dados no exercício a seguir.

Nota: Se compararmos a descrição do dicionário de dados existente no site, observe que X6–X11 chamam-se PAY_1–PAY_6 em nossos dados. Da mesma forma, X12–X17 passam a ser BILL_AMT1–BILL_AMT6 e X18–X23 são PAY_AMT1–PAY_AMT6.

Exercício 3: Verificando a integridade básica dos dados

Neste exercício, executaremos uma verificação básica para saber se o dataset contém o que esperamos e examinaremos se há o número correto de amostras.

Os dados devem ter observações referentes a 30.000 contas de crédito. Embora haja 30.000 linhas, também devemos verificar se existem 30.000 IDs de contas exclusivos. É possível que, se a consulta SQL usada para gerar os dados foi executada com um esquema desconhecido, os valores que deveriam ser exclusivos na verdade não o sejam.

Para examinar isso, podemos verificar se o número de IDs de contas exclusivos é igual ao número de linhas. Execute as etapas a seguir para fazer o exercício:

Nota: O código e o gráfico de saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Examine os nomes das colunas executando o comando a seguir na célula:

```
df.columns
```

O método `.columns` do DataFrame está sendo empregado para examinarmos os nomes de todas as colunas. Você obterá a saída a seguir quando executar a célula:

```
df.columns
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_1',
       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
       'default payment next month'],
      dtype='object')
```

Figura 1.18 – Colunas do dataset.

Como podemos ver, os nomes de todas as colunas estão listados na saída. A coluna de IDs de contas chama-se ID. As outras colunas parecem ser as características, com a última coluna sendo a variável de resposta. Examinaremos resumidamente as informações do dataset que nos foi dado pelo cliente:

LIMIT_BAL: Valor do crédito fornecido (em novos dólares taiwaneses (NT)) inclusive o crédito do consumidor individual e familiar (complementar).

SEX: Gênero (1 = masculino; 2 = feminino).

Nota: Não usaremos os dados de gênero para tomar decisões de solvibilidade devido a considerações éticas.

EDUCATION: Instrução (1 = pós-graduação; 2 = universidade; 3 = ensino médio; 4 = outros).

MARRIAGE: Estado civil (1 = casado; 2 = solteiro; 3 = outros).

AGE: Idade (ano).

PAY_1–Pay_6: Registro de pagamentos passados. Pagamentos mensais passados, registrados de abril a setembro, são armazenados nessas colunas.

PAY_1 representa o status de reembolso em setembro; PAY_2 = status de reembolso em agosto; e assim por diante até PAY_6, que representa o status de reembolso em abril.

A escala de medida do status de reembolso é a seguinte: -1 = pagamento pontual; 1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento; e assim por diante até 8 = atraso de oito meses no pagamento; 9 = atraso de nove meses ou mais no pagamento.

BILL_AMT1–BILL_AMT6: Valor da fatura (em novos dólares taiwaneses).

BILL_AMT1 representa o valor da fatura em setembro; BILL_AMT2 representa o valor da fatura em agosto; e assim por diante até BILL_AMT7, que representa o valor da fatura em abril.

PAY_AMT1–PAY_AMT6: Valor de pagamentos anteriores (novos dólares taiwaneses).

PAY_AMT1 representa o valor pago em setembro; PAY_AMT2 representa o valor pago em agosto; e assim por diante até PAY_AMT6, que representa o valor pago em abril.

Usaremos o método `.head()` na próxima etapa para observar as primeiras linhas de dados.

2. Digite o comando a seguir na célula subsequente e execute-o usando *Shift + Enter*:

```
df.head()
```

Você verá a seguinte saída:

```
df.head()
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	798fc410-45c1	20000	2	2	1	24	2
1	8a8c8f3b-8eb4	120000	2	2	2	26	-1
2	85698822-43f5	90000	2	2	2	34	0
3	0737c11b-be42	50000	2	2	1	37	0
4	3b7f77cc-dbc0	50000	1	2	1	57	-1

5 rows × 25 columns

Figura 1.19 – Método `.head()` de um `DataFrame`.

A coluna ID parece conter identificadores exclusivos. Para verificar se isso ocorre realmente em todo o dataset, podemos contar o número de valores exclusivos usando o método `.nunique()` na série (ou seja, coluna) ID. Primeiro, selecionaremos a coluna usando colchetes.

3. Selecione a coluna alvo (ID) e conte os valores exclusivos usando o comando a seguir:

```
df['ID'].nunique()
```

É possível ver nessa saída que temos 29687 entradas exclusivas:

```
df['ID'].nunique()
```

29687

Figura 1.20 – Descobrindo um problema de qualidade de dados.

4. Execute o comando a seguir para obter o número de linhas do dataset:

```
df.shape
```

Como podemos ver na saída, temos um total de 30.000 linhas no dataset:

```
df.shape
```

(30000, 25)

Figura 1.21 – Dimensões do dataset.

Identificamos que o número de IDs exclusivos é menor do que o número de linhas. Isso significa que o ID não é um identificador exclusivo para as linhas de dados, como pensávamos. Sabemos então que há alguma duplicação de IDs. No entanto, em que quantidade? Um único ID está sendo duplicado várias vezes? Quantos IDs estão sendo duplicados?

Podemos usar o método `.value_counts()` na série ID para começar a responder a essas perguntas. Ele é semelhante a um procedimento **group by/count** em SQL e listará os IDs exclusivos e a frequência com que ocorrem. Executaremos essa operação na próxima etapa e armazenaremos as contagens de valores em uma variável `id_counts`.

5. Armazene as contagens de valores em uma variável definida como `id_counts` e exiba os valores armazenados usando o método `.head()`, como mostrado:

```
id_counts = df['ID'].value_counts()
id_counts.head()
```

Você obterá a seguinte saída:

```
id_counts = df['ID'].value_counts()
id_counts.head()

e50d8395-da32    2
4534975d-bf92    2
a3a5c0fc-fdd6    2
0d66d575-c461    2
fd6033f4-cc72    2
Name: ID, dtype: int64
```

Figura 1.22 – Obtendo contagens de valores de IDs de contas.

Observe que `.head()` retorna as cinco primeiras linhas por padrão. Você pode especificar o número de itens a serem exibidos passando o número desejado nos parênteses, `()`.

6. Exiba o número de entradas duplicadas agrupadas executando outra contagem de valores:

```
id_counts.value_counts()
```

Obteremos esta saída:

```
id_counts.value_counts()

1    29374
2     313
Name: ID, dtype: int64
```

Figura 1.23 – Obtendo contagens de valores das IDs de contas.

Na saída anterior e na contagem de valores inicial, podemos ver que a maioria dos IDs ocorre exatamente uma única vez, como esperado. No entanto, 313 IDs ocorrem duas vezes. Logo, nenhum ID ocorre mais do que duas vezes. De posse dessas informações, estamos prontos para começar a examinar mais detalhadamente esse problema de qualidade de dados e corrigi-lo. Criaremos máscaras booleanas para limpar melhor os dados.

Máscaras booleanas

Para ajudar a limpar os dados do estudo de caso, introduziremos o conceito de **máscara lógica**, também conhecida como **máscara booleana**. Uma máscara lógica é uma maneira de filtrar um array, ou série, obedecendo alguma condição. Por exemplo, podemos usar o operador “igual a” em Python, `==`, para encontrar todos os locais de um array que contêm um valor específico. Outras comparações, como “maior que” (`>`), “menor que” (`<`), “maior ou igual a” (`>=`) e “menor ou igual a” (`<=`), também podem ser usadas. A saída dessa comparação é um array com uma série de valores `True/False`, também conhecidos como valores **booleanos**. Cada elemento da saída corresponderá a um elemento da entrada, com o resultado `True` se a condição for atendida; caso contrário, obteremos `False`. Para ilustrar como funciona, usaremos **dados sintéticos**, ou seja, dados criados para a exploração ou a ilustração de um conceito. Primeiro, importaremos o pacote NumPy, que tem muitos recursos de geração de números aleatórios, e daremos a ele o alias `np`:

```
import numpy as np
```

Agora usaremos o que é chamado de **seed** (semente) no gerador de números aleatórios. Se você definir um seed, obterá os mesmos resultados entre as execuções do gerador. Caso contrário, isso não é garantido. Essa pode ser uma opção útil se você usa números aleatórios em seu trabalho e quer ter resultados consistentes sempre que executar um notebook:

```
np.random.seed(seed=24)
```

Em seguida, geraremos 100 inteiros aleatórios, selecionados entre 1 e 5 (inclusive). Para fazê-lo, podemos usar `numpy.random.randint`, com os argumentos apropriados.

```
random_integers = np.random.randint(low=1,high=5,size=100)
```

Examinaremos os cinco primeiros elementos desse array, com `random_integers[:5]`. A saída deve ser a seguinte:

```
array([3, 4, 1, 4, 2])
```

Figura 1.24 – Inteiros aleatórios.

Suponhamos que quiséssemos conhecer os locais de todos os elementos de `random_integers` igual a 3. Poderíamos criar uma máscara booleana para fazer isso.

```
is_equal_to_3 = random_integers == 3
```

Pela verificação dos primeiros cinco elementos, sabemos que só o primeiro elemento é igual a 3, mas não os outros. Logo, em nossa máscara booleana, esperamos `True` na primeira posição e `False` nas 4 posições seguintes. É isso que ocorre?

```
is_equal_to_3[:5]
```

O código anterior deve fornecer esta saída:

```
array([ True, False, False, False, False])
```

Figura 1.25 – Máscara booleana para os números inteiros.

É isso que esperávamos. Esse exemplo mostra a criação de uma máscara booleana. No entanto, o que mais podemos fazer? Suponhamos que quiséssemos saber quantos elementos são iguais a 3. Nesse caso, você pode fazer a soma de uma máscara booleana, que interpreta `True` como 1 e `False` como 0:

```
sum(is_equal_to_3)
```

Esse código deve fornecer a saída a seguir:

22

Figura 1.26 – Soma da máscara booleana.

Faz sentido, já que, com uma seleção aleatória igualmente provável de 5 valores possíveis, o esperado é que cada valor apareça 20% das vezes. Além de ver quantos valores do array atendem à condição booleana, também podemos usar a máscara booleana para selecionar os elementos do array original que atendem a essa condição. As máscaras booleanas podem ser usadas diretamente para indexar arrays, como mostrado aqui:

```
random_integers[is_equal_to_3]
```

Essa instrução exibe os elementos de `random_integers` que atendem à condição booleana que especificamos. Nesse caso, os 22 elementos iguais a 3:

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

Figura 1.27 – Usando a máscara booleana para indexar um array.

Agora você conhece os aspectos básicos dos arrays booleanos, que são úteis em muitas situações. Especificamente, você pode usar o método `.loc` de DataFrames para indexar suas linhas por uma máscara booleana e as colunas por rótulo. Continuaremos a explorar os dados do estudo de caso com essas habilidades.

Exercício 4: Continuando a verificação da integridade dos dados

Nesse exercício, com nosso conhecimento dos arrays booleanos, examinaremos alguns dos IDs duplicados que descobrimos. No Exercício 3, verificamos que nenhum ID apareça mais de duas vezes. Podemos usar essa informação para localizar os IDs duplicados e examiná-los. Em seguida, tomaremos medidas para remover linhas de qualidade duvidosa do dataset. Execute as etapas a seguir para fazer o exercício:

Nota: O código e o gráfico da saída desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Continuando onde paramos no Exercício 3, queremos os índices da série `id_counts` cuja contagem é 2 para localizar as duplicatas. Atribuiremos os IDs duplicados a uma variável chamada `dupe_mask` e exibiremos os cinco primeiros IDs duplicados usando os comandos a seguir:

```
dupe_mask = id_counts == 2
dupe_mask[0:5]
```

Você obterá a seguinte saída:

```
47d9ee33-0df0    True
db903e22-a55a    True
9878723a-0b58    True
8d3a2576-a958    True
956cbf4a-d24e    True
Name: ID, dtype: bool
```

Figura 1.28 – Máscara booleana para localizar IDs duplicados.

Aqui, `dupe_mask` é a máscara lógica que criamos para armazenar os valores booleanos.

Observe que, na saída anterior, estamos exibindo só as cinco primeiras entradas usando `dupe_mask` para ilustrar o conteúdo desse array. Como sempre, você pode editar os índices nos colchetes (`[]`) para alterar o número de entradas exibidas.

Nossa próxima etapa é usar a máscara lógica para selecionar os IDs que estão duplicados. Os próprios IDs foram incluídos como índice da série `id_count`. Podemos acessar o índice para usar nossa máscara lógica para fins de seleção.

2. Acesse o índice de `id_count` e exiba as cinco primeiras linhas como contexto usando o comando a seguir:

```
id_counts.index[0:5]
```

Com essa instrução, você obterá a seguinte saída:

```
Index(['47d9ee33-0df0', 'db903e22-a55a', '9878723a-0b58', '8d3a2576-a958',
      '956cbf4a-d24e'],
      dtype='object')
```

Figura 1.29 – IDs duplicados.

3. Selecione e armazene os IDs duplicados em uma nova variável chamada `dupe_ids` usando o comando a seguir:

```
dupe_ids = id_counts.index[dupe_mask]
```

4. Converta `dupe_ids` em uma lista e obtenha seu tamanho usando os seguintes comandos:

```
dupe_ids = list(dupe_ids)
len(dupe_ids)
```

Você deve ver esta saída:

313

Figura 1.30 – Saída exibindo o tamanho da lista.

Alteramos a variável `dupe_ids` para uma lista, já que precisaremos dela nessa forma em etapas futuras. A lista tem um tamanho igual a 313, como pode ser visto na saída anterior, que coincide com o número de IDs duplicados que obtivemos na contagem de valores.

5. Verificaremos os dados de `dupe_ids` exibindo as cinco primeiras entradas com o comando a seguir:

```
dupe_ids[0:5]
```

A seguinte saída é obtida:

```
dupe_ids[0:5]
```

```
['47d9ee33-0df0',
 'db903e22-a55a',
 '9878723a-0b58',
 '8d3a2576-a958',
 '956cbf4a-d24e']
```

Figura 1.31 – Criando uma lista de IDs duplicados.

Podemos observar na saída anterior que a lista contém as entradas de IDs duplicados requeridas. Estamos prontos para examinar os dados dos IDs de nossa lista de duplicatas. Especificamente, queremos examinar os valores das características, para ver o que, se houver algo, há de diferente entre essas entradas duplicadas. Usaremos os métodos `.isin` e `.loc` para esse fim.

Usando os três primeiros IDs de nossa lista de duplicatas, `dupe_ids[0:3]`, primeiro queremos encontrar as linhas que contêm esses IDs. Se passarmos essa lista de IDs para o método `.isin` da série `ID`, ele criará outra máscara lógica que poderemos usar no `DataFrame` maior para exibir as linhas que contêm os IDs. O método `.isin` ficará aninhado em uma instrução `.loc` de indexação do `DataFrame` para a seleção do local de todas as linhas que contêm “True” na máscara booleana. O segundo argumento da instrução de indexação `.loc` é `:`, que implica que todas as colunas serão selecionadas. Ao executar as próximas etapas, estaremos basicamente filtrando o `DataFrame` para visualizar todas as colunas dos três primeiros IDs duplicados.

6. Execute o comando a seguir em seu Notebook para pôr em prática o plano que formulamos na etapa anterior:

```
df.loc[df['ID'].isin(dupe_ids[0:3]),:].head(10)
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5
11769	9878723a-0b58	130000	1	5	1	44	0	0	0	0	...	54668	22780
11869	9878723a-0b58	0	0	0	0	0	0	0	0	0	...	0	0
14979	db903e22-a55a	50000	1	2	2	55	2	0	0	0	...	15848	16026
15079	db903e22-a55a	0	0	0	0	0	0	0	0	0	...	0	0
23377	47d9ee33-0df0	550000	2	2	1	32	2	0	0	0	...	548020	530672
23477	47d9ee33-0df0	0	0	0	0	0	0	0	0	0	...	0	0

6 rows x 25 columns

Figura 1.32 – Examinando os dados de IDs duplicados.

O que estamos vendo aqui é que cada ID duplicado parece ter uma linha de dados válidos e outra somente com zeros. Pare um momento e pense o que poderia fazer com essa informação.

Após alguma reflexão, deve ter ficado claro que você deve excluir as linhas só de zeros. Elas podem ter surgido devido a uma condição de associação errada na consulta SQL que gerou os dados. De qualquer forma, uma linha só de zeros definitivamente contém dados inválidos e não faria sentido alguém ter 0 anos, limite de crédito igual a 0 e assim por diante.

Uma abordagem para lidarmos com esse problema seria encontrar as linhas que só têm zeros, exceto na primeira coluna, que tem os IDs. Esses dados seriam inválidos, e, se os excluíssemos, também resolveríamos nosso problema de IDs duplicados. Podemos encontrar as entradas do `DataFrame` que são

iguais a zero criando uma matriz booleana com o mesmo tamanho do DataFrame inteiro, usando a condição “é igual a zero”.

7. Crie uma matriz booleana com o mesmo tamanho do DataFrame inteiro usando `==`, como mostrado:

```
df_zero_mask = df == 0
```

Nas próximas etapas, usaremos `df_zero_mask`, que é outro DataFrame, contendo valores booleanos. O objetivo é criarmos uma série booleana, `feature_zero_mask`, que identifique cada linha em que todos os elementos a partir da segunda coluna (as características e a resposta, mas não os IDs) sejam 0. Para fazê-lo, primeiro temos de indexar `df_zero_mask` usando o método de indexação de inteiros (`.iloc`). Nesse método, passaremos (`:`) para examinar todas as linhas e (`1:`) para examinar todas as colunas a partir da segunda (índice 1). Para concluir, aplicaremos o método `all()` ao longo do eixo da coluna (`axis=1`) e ele retornará `True` somente se todas as colunas dessa linha forem iguais a `True`. Mesmo sendo muita coisa para considerar, é fácil de codificar, como veremos na etapa a seguir.

8. Crie a série booleana `feature_zero_mask`, como mostrado aqui:

```
feature_zero_mask = df_zero_mask.iloc[:,1:].all(axis=1)
```

9. Calcule a soma da série booleana usando este comando:

```
sum(feature_zero_mask)
```

Você deve obter a seguinte saída:

315

Figura 1.33 – Número de linhas somente com zeros exceto pelo ID.

A saída anterior nos diz que 315 linhas têm zeros para cada coluna exceto a primeira. Esse número é maior do que o número de IDs duplicados (313), logo, se excluirmos todas as “linhas de zeros”, podemos nos livrar do problema dos IDs duplicados.

10. Limpe o DataFrame eliminando as linhas só com zeros, exceto pelo ID, usando o código a seguir:

```
df_clean_1 = df.loc[~feature_zero_mask,:].copy()
```

Ao executar a operação de limpeza da etapa anterior, retornamos um novo DataFrame chamado `df_clean_1`. Observe que usamos o método `.copy()` após a operação de indexação `.loc` para criar uma cópia dessa saída, em vez de visualizarmos o DataFrame original. Podemos considerar essa ação como a criação de um novo DataFrame em vez de referenciarmos o original. Dentro do método `.loc`, usamos

o operador lógico not, ~, para selecionar todas as linhas que não têm zeros para todas as características e a resposta, e: para selecionar todas as colunas. Esses são os dados válidos que queremos manter. Após fazer isso, queremos saber se o número de linhas restantes é igual ao número de IDs exclusivos.

11. Verifique o número de linhas e colunas de `df_clean_1` executando o código a seguir:

```
df_clean_1.shape
```

Você verá esta saída:

```
df_clean_1 = df.loc[~feature_zero_mask,:].copy()

df_clean_1.shape

(29685, 25)
```

Figura 1.34 – Dimensões do DataFrame limpo.

12. Obtenha o número de IDs exclusivos executando este código:

```
df_clean_1['ID'].nunique()
```

```
df_clean_1['ID'].nunique()

29685
```

Figura 1.35 – Número de IDs exclusivos no DataFrame limpo.

Na saída anterior, podemos ver que eliminamos com sucesso as duplicatas, já que o número de IDs exclusivos é igual ao número de linhas. Relaxe e anime-se. Essa foi uma introdução complexa a algumas técnicas do pandas para indexação e caracterização de dados. Agora que extraímos os IDs duplicados, estamos prontos para começar a examinar os dados propriamente ditos: as características e a resposta. Vamos conduzi-lo por esse processo.

Exercício 5: Explorando e limpando os dados

Até agora, identificamos um problema de qualidade de dados relacionado aos metadados: fomos informados de que cada amostra de nosso dataset corresponderia a um ID de conta exclusivo, o que não ocorreu. Conseguimos usar a indexação lógica e o pandas para resolver o problema. Esse foi um problema básico de qualidade de dados, referente apenas a que amostras estavam presentes, com base nos metadados. Fora isso, na verdade não estamos interessados na coluna de metadados dos IDs de contas: por enquanto eles não nos ajudarão a desenvolver um modelo preditivo de inadimplência.

Estamos prontos para começar a examinar os valores das características e da resposta, os dados que usaremos para desenvolver nosso modelo preditivo. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Obtenha o tipo de dado das colunas do dataset usando o método `.info()` como mostrado:

```
df_clean_1.info()
```

Você deve ver a seguinte saída:

```
df_clean_1.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 29685 entries, 0 to 29999
Data columns (total 25 columns):
ID                                29685 non-null object
LIMIT_BAL                        29685 non-null int64
SEX                              29685 non-null int64
EDUCATION                        29685 non-null int64
MARRIAGE                         29685 non-null int64
AGE                              29685 non-null int64
PAY_1                            29685 non-null object
PAY_2                            29685 non-null int64
PAY_3                            29685 non-null int64
PAY_4                            29685 non-null int64
PAY_5                            29685 non-null int64
PAY_6                            29685 non-null int64
BILL_AMT1                        29685 non-null int64
BILL_AMT2                        29685 non-null int64
BILL_AMT3                        29685 non-null int64
BILL_AMT4                        29685 non-null int64
BILL_AMT5                        29685 non-null int64
BILL_AMT6                        29685 non-null int64
PAY_AMT1                         29685 non-null int64
PAY_AMT2                         29685 non-null int64
PAY_AMT3                         29685 non-null int64
PAY_AMT4                         29685 non-null int64
PAY_AMT5                         29685 non-null int64
PAY_AMT6                         29685 non-null int64
default payment next month       29685 non-null int64
dtypes: int64(23), object(2)
memory usage: 5.9+ MB
```

Figura 1.36 – Obtendo metadados das colunas.

Podemos ver na *Figura 1.34* que há 25 colunas. Cada linha tem 29685 valores **não nulos**, de acordo com esse resumo, que é o número de linhas do Data-Frame. Isso indicaria que não há dados ausentes, já que cada célula contém algum valor. No entanto, se houver um valor de preenchimento para representar dados ausentes, ele não ficaria evidente aqui.

Também vemos que a maioria das colunas exibe `int64` próximo a elas, indicando que têm o tipo de dado **integer**, isto é, números como ..., -2, -1, 0, 1, 2,... . As exceções são `ID` e `PAY_1`. Já estamos familiarizados com `ID`; ela contém strings, que são IDs de contas. E quanto a `PAY_1`? De acordo com os valores do dicionário de dados, é esperado que contenha inteiros, como todas as outras características. Vejamos essa coluna com mais detalhes.

2. Use o método `.head(n)` do pandas para visualizar as `n` linhas superiores da série `PAY_1`:

```
df_clean_1['PAY_1'].head(5)
```

Você deve ver esta saída:

```
df_clean_1['PAY_1'].head(5)
0      2
1     -1
2      0
3      0
4     -1
Name: PAY_1, dtype: object
```

Figura 1.37 – Examine o conteúdo de algumas colunas.

Os inteiros da esquerda da saída são o índice, que são simplesmente inteiros consecutivos começando em 0. Os dados da coluna `PAY_1` são mostrados à direita. Eles são o status de pagamento da fatura mensal mais recente, usando os valores -1, 1, 2, 3 e assim por diante. No entanto, podemos ver que há valores 0 aqui, que não estão documentados no dicionário de dados. De acordo com o dicionário de dados, “A escala de medida do status de reembolso é: -1 = pagamento pontual; 1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento; . . . ; 8 = atraso de oito meses no pagamento; 9 = atraso de nove meses ou mais no pagamento” (<https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>). Faremos um exame mais detalhado, usando as contagens de valores dessa coluna.

3. Obtenha as contagens de valores da coluna `PAY_1` usando o método `.value_counts()`:

```
df_clean1['PAY_1'].value_counts()
```

Você deve ver a saída a seguir:


```
df_clean_1['PAY_1'].value_counts()

0          13087
-1          5047
1           3261
Not available 3021
-2          2476
2           2378
3            292
4             63
5             23
8             17
6             11
7              9
Name: PAY_1, dtype: int64
```

Figura 1.38 – Contagens de valores da coluna PAY_1.

A saída anterior revela a presença de dois valores não documentados, 0 e -2, e também a razão de essa coluna ter sido importada pelo pandas como um tipo de dado `object`, em vez de `int64` como era de se esperar para dados inteiros. Há uma string `'Not available'` presente na coluna, simbolizando dados ausentes. Posteriormente no livro, retornaremos a ela quando considerarmos como lidar com dados ausentes. Por enquanto, removeremos as linhas do dataset nas quais a característica tem um valor ausente.

4. Use uma máscara lógica com o operador `!=` (que significa “diferente de” em Python) para encontrar todas as linhas que não têm dados ausentes para a característica de PAY_1:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'
valid_pay_1_mask[0:5]
```

Ao executar o código anterior, você obterá a saída a seguir:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'

valid_pay_1_mask[0:5]

0    True
1    True
2    True
3    True
4    True
Name: PAY_1, dtype: bool
```

Figura 1.39 – Criando uma máscara booleana.

5. Verifique quantas linhas não têm dados ausentes calculando a soma da máscara:

```
sum(valid_pay_1_mask)
```

Você obterá esta saída:

```
sum(valid_pay_1_mask)
26664
```

Figura 140 – Soma da máscara booleana para dados não ausentes.

Vemos que 26.664 linhas não têm o valor 'Not available' na coluna PAY_1. Na contagem de valores, 3.021 linhas tinham esse valor, e, já que $29.685 - 3.021 = 26.664$, isso está correto.

6. Limpe os dados eliminando as linhas de PAY_1 com valores ausentes como mostrado aqui:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask,:].copy()
```

7. Obtenha a dimensão dos dados limpos usando o comando a seguir:

```
df_clean_2.shape
```

Você verá esta saída:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask,:].copy()

df_clean_2.shape
(26664, 25)
```

Figura 141 – Dimensão dos dados limpos.

Após remover essas linhas, vemos que o DataFrame resultante tem a dimensão esperada. Você também pode verificar por sua própria conta se as contagens de valores indicam que os valores desejados foram removidos dessa forma: `df_clean_2['PAY_1'].value_counts()`.

Por fim, para que o tipo de dado dessa coluna seja consistente com os outros, vamos convertê-lo do tipo genérico `object` para `int64` como em todas as outras características, usando o método `.astype`. Em seguida, selecionaremos duas colunas, inclusive PAY_1, para examinar os tipos de dados e nos certificarmos se funcionou.

8. Execute o comando a seguir para converter o tipo de dado de PAY_1 de `object` para `int64` e exiba os metadados das colunas PAY_1 e PAY_2:

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')
df_clean_2[['PAY_1', 'PAY_2']].info()
```

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')

df_clean_2[['PAY_1', 'PAY_2']].info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 26664 entries, 0 to 29999
Data columns (total 2 columns):
PAY_1      26664 non-null int64
PAY_2      26664 non-null int64
dtypes: int64(2)
memory usage: 624.9 KB
```

Figura 1.42 – Verifique o tipo de dado da coluna limpa.

Parabéns, você concluiu sua segunda operação de limpeza de dados! No entanto, deve se lembrar de que durante o processo também notamos os valores não documentados -2 e 0 em `PAY_1`. Suponhamos que entrássemos em contato novamente com o sócio da empresa e nos fosse dada a seguinte informação:

- -2 significa que a conta começou o mês sem valor a ser pago e o crédito não foi usado
- -1 significa que a conta usou um valor que foi totalmente pago
- 0 significa que o pagamento mínimo foi feito, mas o saldo total devedor não foi pago (isto é, uma parcela do saldo devedor foi transportada para o próximo mês)

Agradecemos ao sócio da empresa, já que por enquanto essas informações esclarecem nossas dúvidas. É importante manter um canal de comunicação e uma relação de trabalho satisfatórios com o sócio da empresa, como você pode ver aqui, porque isso é capaz de determinar o sucesso ou a falha de um projeto.

Exploração e garantia da qualidade dos dados

Até aqui, resolvemos dois problemas de qualidade dos dados apenas fazendo perguntas básicas ou examinando o resumo de `.info()`. Examinaremos agora as primeiras colunas. Antes de chegarmos no histórico de pagamento de faturas, temos os limites de crédito das contas em `LIMIT_BAL` e as características demográficas `SEX`, `EDUCATION`, `MARRIAGE` e `AGE`. O sócio da empresa nos procurou para informar que o gênero não deve ser usado na previsão de solvibilidade, já que por seus padrões seria antiético. Logo, memorizaremos isso para referência futura. Exploraremos então o resto dessas colunas, fazendo as correções necessárias.

Para explorar melhor os dados, usaremos **histogramas**. Os histogramas são uma boa maneira de visualizar dados que estejam em uma escala contínua, como valores monetários e faixas etárias. Um histograma agrupa valores semelhantes em bins e exibe o número de pontos de dados existentes nesses bins como um gráfico de barras.

Para plotar os histogramas, nos familiarizaremos com os recursos gráficos do pandas, que conta com uma biblioteca chamada **Matplotlib** para criar gráficos, logo, também definiremos algumas opções usando o `matplotlib`. Empregando essas ferramentas, aprenderemos como obter rapidamente sínteses estatísticas dos dados no pandas.

Exercício 6: Explorando o limite de crédito e as características demográficas

Nesse exercício, começaremos nossa exploração dos dados com o limite de crédito e as características etárias. Vamos visualizá-los e obter sínteses estatísticas para verificar se os dados contidos nessas características são aceitáveis. Em seguida, examinaremos as características categóricas de instrução e estado civil para ver se os valores fazem sentido e os corrigiremos se necessário. `LIMIT_BAL` e `AGE` são características numéricas, o que significa que são medidas em uma escala contínua. Consequentemente, usaremos histogramas para visualizá-las. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Importe o `matplotlib` e defina algumas opções de plotagem com este fragmento de código:

```
import matplotlib.pyplot as plt #importa o pacote de plotagem

#renderiza a plotagem automaticamente
%matplotlib inline

import matplotlib as mpl #recurso adicional de plotagem

mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
```

Esse código importa o `matplotlib` e usa `.rcParams` para definir a resolução (dpi = pontos por polegada) para obtenção de uma imagem com boa nitidez; não é preciso se preocupar com essa parte a não ser que você esteja preparando material para apresentação, já que isso pode tornar as imagens grandes demais em seu notebook.

2. Execute `df_clean_2[['LIMIT_BAL', 'AGE']].hist()` e deve ver os histogramas a seguir:

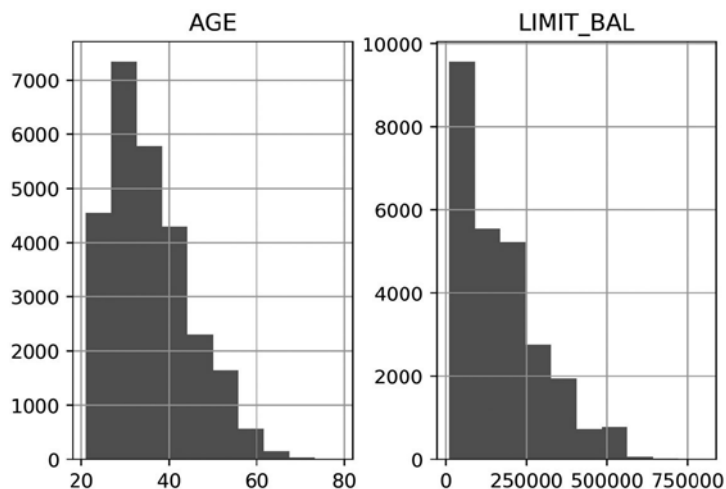


Figura 143 – Histogramas dos dados de limite de crédito e idade.

Esse é um bom snapshot visual dessas características. Podemos obter uma rápida visualização aproximada de todos os dados dessa forma. Para vermos sínteses estatísticas, como a média e a mediana (isto é, o quinquagésimo percentil), há outra função útil do pandas.

3. Gere um relatório tabular de síntese estatística usando o comando a seguir:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

Você deve ver esta saída:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

	LIMIT_BAL	AGE
count	26664.000000	26664.000000
mean	167919.054905	35.505213
std	129839.453081	9.227442
min	10000.000000	21.000000
25%	50000.000000	28.000000
50%	140000.000000	34.000000
75%	240000.000000	41.000000
max	800000.000000	79.000000

Figura 144 – Sínteses estatísticas de dados de limite de crédito e idade.

Com base nos histogramas e nas convenientes estatísticas calculadas por `.describe()`, que inclui uma contagem de não nulos, a média e o desvio-padrão,

o valor mínimo, o valor máximo e os quartis, podemos chegar a algumas conclusões.

LIMIT_BAL, o limite de crédito, parece fazer sentido. Os limites de crédito têm um valor mínimo igual a 10.000. Esse dataset é de Taiwan; a unidade monetária (novos dólares taiwaneses) não é familiar, mas intuitivamente sabemos que o limite de crédito deve ser um valor acima de zero. Recomendamos que você tente converter para a moeda local e considere esses limites. Por exemplo, 1 dólar americano é igual a cerca de 30 novos dólares taiwaneses.

A característica AGE também parece bem distribuída, com ninguém com idade abaixo de 21 anos possuindo uma conta de crédito.

Para as características categóricas, é útil verificar as contagens de valores, já que há relativamente poucos valores exclusivos.

4. Obtenha as contagens de valores da característica EDUCATION usando o código a seguir:

```
df_clean_2['EDUCATION'].value_counts()
```

Você deve ver esta saída:

```
df_clean_2['EDUCATION'].value_counts()
2    12458
1     9412
3     4380
5      245
4       115
6        43
0         11
Name: EDUCATION, dtype: int64
```

Figura 1.45 – Contagens de valores da característica EDUCATION.

Aqui, vemos os graus de instrução não documentados 0, 5 e 6, já que o dicionário de dados descreve apenas “Instrução (1 = pós-graduação; 2 = universidade; 3 = ensino médio; 4 = outros)”. O sócio da empresa nos disse que não conhece os outros graus. Já que eles não são predominantes, vamos agrupá-los na categoria “outros”, que parece apropriada, claro que com o consentimento de nosso cliente.

5. Execute este código para combinar os graus não documentados da característica EDUCATION com o grau “outros” e examine os resultados:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)
df_clean_2['EDUCATION'].value_counts()
```

O método `.replace` do pandas ajuda a fazermos rapidamente as substituições descritas na etapa anterior. Quando você executar o código, deve ver esta saída:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)

df_clean_2['EDUCATION'].value_counts()

2      12458
1       9412
3       4380
4        414
Name: EDUCATION, dtype: int64
```

Figura 146 – Limpando a característica EDUCATION.

Observe que fizemos essa alteração **in loco** (`inplace=True`). Isso significa que, em vez de retornar um novo DataFrame, a operação fará a alteração no DataFrame existente.

- Obtenha as contagens de valores da característica MARRIAGE usando o código a seguir:

```
df_clean_2['MARRIAGE'].value_counts()
```

Você deve ver a seguinte saída:

```
df_clean_2['MARRIAGE'].value_counts()

2      14158
1      12172
3        286
0         48
Name: MARRIAGE, dtype: int64
```

Figura 147 – Contagens de valores da característica MARRIAGE bruta.

O problema aqui é semelhante ao encontrado na característica EDUCATION; há um valor, 0, que não está documentado no dicionário de dados: “1 = casado; 2 = solteiro; 3 = outros”. Logo, vamos agrupá-lo com “outros”.

- Altere os valores 0 da característica MARRIAGE para 3 e examine o resultado com este código:

```
df_clean_2['MARRIAGE'].replace(to_replace=0, value=3, inplace=True)
df_clean_2['MARRIAGE'].value_counts()
```

A saída deve ser:

```
2    14158
1    12172
3     334
Name: MARRIAGE, dtype: int64
```

Figura 1.48 – Contagens de valores da característica MARRIAGE limpa.

Fizemos uma extensa exploração e limpeza dos dados. Posteriormente, executaremos operações mais avançadas de visualização e exploração das características de histórico financeiro, que vêm depois disso no DataFrame.

Aprofundamento nas características categóricas

Os algoritmos de machine learning só funcionam com números. Se seus dados contiverem características textuais, por exemplo, você terá de transformá-las em números. Na verdade, já sabemos que os dados de nosso estudo de caso são totalmente numéricos. No entanto, é interessante ponderar por que precisa ser assim. Especificamente, considere a característica EDUCATION.

Esse é um exemplo do que é conhecido como **característica categórica**: sabemos que, como dados brutos, essa coluna seria composta dos rótulos de texto 'pós-graduação', 'universidade', 'ensino médio' e 'outros'. Esses são os chamados **níveis** da característica categórica; aqui, há quatro níveis. Somente por intermédio de um mapeamento, que já foi definido para nós, é que esses dados existem como os números 1, 2, 3 e 4 em nosso dataset. Essa atribuição específica de categorias a números cria o que é conhecido como **característica ordinal**, já que os níveis são mapeados para números em ordem. Como cientista de dados, você precisa conhecer esses mapeamentos, se não foi quem os escolheu.

Quais são as implicações desse mapeamento?

Faz sentido que os graus de instrução sejam categorizados, com 1 correspondendo ao mais alto grau de nosso dataset, 2 ao grau superior seguinte, 3 ao próximo e 4 presumivelmente incluindo os graus mais baixos. No entanto, quando usarmos essa codificação como característica numérica em um modelo de machine learning, ela será tratada como qualquer outra característica numérica. Para alguns modelos, esse efeito pode não ser o desejado.

E se a função de um modelo for encontrar um relacionamento linear entre as características e a resposta?

Se isso vai funcionar ou não dependerá do relacionamento real entre os diferentes graus de instrução e o resultado que estivermos tentando prever.

Examinaremos dois casos hipotéticos de variáveis categóricas ordinais, com 10 níveis em cada um. Os níveis medem os graus de satisfação relatados por clientes que visitam um site. O período médio em minutos gasto no site pelos clientes que relataram cada nível está plotado no eixo y. Também plotamos a linha de melhor ajuste em cada caso para ilustrar como um modelo linear lidaria com esses dados, como mostrado na Figura 1.49:

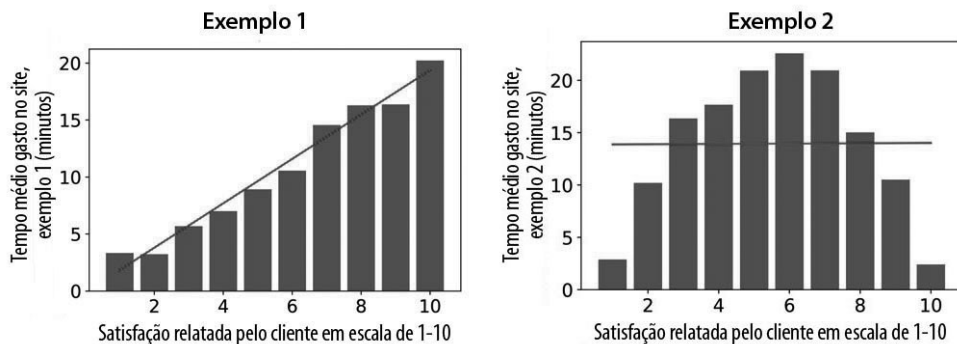


Figura 1.49 – Características ordinais podem ou não funcionar bem em um modelo linear.

Podemos ver que, se um algoritmo assumir um relacionamento linear (linha reta) entre as características e a resposta, isso pode ou não funcionar bem dependendo do relacionamento real entre essa característica e a variável de resposta. Observe que, no exemplo anterior, estamos modelando um problema de regressão: a variável de resposta adota um intervalo de números contínuo. No entanto, alguns algoritmos de classificação como a **regressão logística** também assumem um efeito linear das características. Discutiremos isso com mais detalhes posteriormente quando modelarmos os dados de nosso estudo de caso.

De um modo geral, para um modelo de classificação binária, você pode examinar os diferentes níveis de uma característica categórica em relação aos valores médios da variável de resposta, os quais representam as “taxas” da classe positiva (isto é, as amostras em que a variável de resposta = 1) para cada nível. Isso pode lhe dar uma ideia de se uma codificação ordinal funcionará bem com um modelo linear. Supondo que você tenha importado para o seu Jupyter notebook os mesmos pacotes das seções anteriores, poderá verificar isso rapidamente usando `groupby/agg` e a plotagem de barras no `pandas`:

```
f_clean_2.groupby('EDUCATION').agg({'default payment next month':'mean'}).plot.  
    bar(legend=False)  
plt.ylabel('Default rate')  
plt.xlabel('Education level: ordinal encoding')
```

Quando você executar o código, deve ver a saída a seguir:

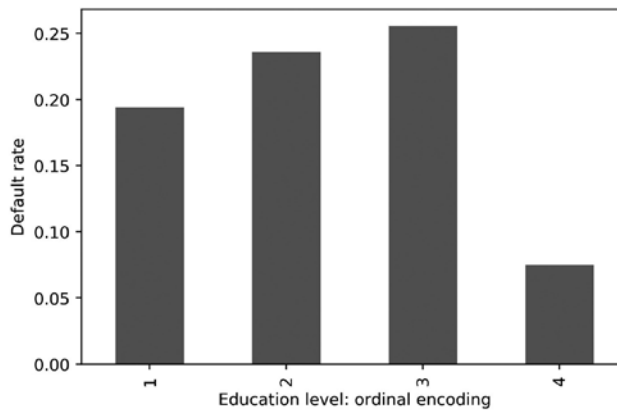


Figura 1.50 – Taxa de inadimplência dentro dos graus de instrução.

Como no exemplo 2 da *Figura 1.49*, aqui parece que um ajuste de linha reta não seria a melhor descrição dos dados. Caso uma característica tenha um efeito não linear como esse, pode ser melhor usar um algoritmo mais complexo como uma **árvore de decisão** ou **floresta aleatória**. Ou, se um modelo linear mais simples e interpretável como a regressão logística for desejado, poderíamos evitar uma codificação ordinal e usar uma maneira diferente de codificar variáveis categóricas. Uma maneira popular de fazer isso se chama **codificação one-hot (OHE, one-hot encoding)**.

A OHE é uma maneira de transformarmos uma característica categórica, que pode ser composta de rótulos de texto nos dados brutos, em uma característica numérica que possa ser usada em modelos matemáticos.

Aprenderemos isso em um exercício. E, se você estiver se perguntando por que uma regressão logística é mais interpretável e uma floresta aleatória é mais complexa, aprenderemos esses conceitos com detalhes no decorrer do livro.

Nota: Variáveis categóricas em diferentes pacotes de machine learning

Alguns pacotes de machine learning, por exemplo, certos pacotes do R ou versões mais recentes da plataforma Spark para big data, manipulam variáveis categóricas sem assumir que elas sejam ordinais. Certifique-se sempre de ler cuidadosamente a documentação para saber o que o modelo assumirá sobre as características, e como especificar se uma variável é categórica, se essa opção estiver disponível.

Exercício 7: Implementando a OHE para uma característica categórica

Nesse exercício, executaremos uma “engenharia reversa” na característica EDUCATION do dataset para obter os rótulos de texto que representam os diferentes graus de instrução e mostraremos como usar o pandas para criar uma OHE.

Primeiro, consideraremos nossa característica EDUCATION, antes de ela ser codificada como um ordinal. Pelo dicionário de dados, sabemos que 1 = pós-graduação (graduate school), 2 = universidade (university), 3 = ensino médio (high school), 4 = outros (others). Queremos criar uma coluna que tenha essas strings, em vez de números. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Crie uma coluna vazia para os rótulos categóricos chamada EDUCATION_CAT usando o comando a seguir:

```
df_clean_2['EDUCATION_CAT'] = 'none'
```

2. Examine as primeiras linhas do DataFrame referentes às colunas EDUCATION e EDUCATION_CAT:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

A saída deve ser esta:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	none
1	2	none
2	2	none
3	2	none
4	2	none
5	1	none
6	1	none
7	2	none
8	3	none
9	3	none

Figura 1.51 – Selecionando colunas e visualizando as 10 primeiras linhas.

Precisamos preencher essa nova coluna com as strings apropriadas. O pandas fornece uma funcionalidade conveniente para o mapeamento dos valores de

uma série (*Series*) para novos valores. Na verdade, essa função chama-se `.map` e usa um dicionário para estabelecer a correspondência entre os valores antigos e os novos. Nosso objetivo aqui é mapear os números de `EDUCATION` para as strings que eles representam. Por exemplo, onde a coluna `EDUCATION` for igual a 1, atribuiremos a string 'graduate school' à coluna `EDUCATION_CAT`, e assim por diante para os outros graus de instrução.

3. Crie um dicionário que descreva o mapeamento das categorias de instrução usando o código a seguir:

```
cat_mapping = {
    1: "graduate school",
    2: "university",
    3: "high school",
    4: "others"
}
```

4. Aplique o mapeamento à coluna `EDUCATION` original usando `.map` e atribua o resultado à nova coluna `EDUCATION_CAT`:

```
df_clean_2['EDUCATION_CAT'] = df_clean_2['EDUCATION'].map(cat_mapping)
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

Após executar essas linhas, você deve ver esta saída:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	university
1	2	university
2	2	university
3	2	university
4	2	university
5	1	graduate school
6	1	graduate school
7	2	university
8	3	high school
9	3	high school

Figura 1.52 – Examinando os valores de string correspondentes à codificação ordinal de `EDUCATION`.

Excelente! Observe que poderíamos ter pulado a *Etapa 1*, na qual atribuímos a nova coluna com 'none', e ter ido direto para as *Etapas 3 e 4* para criar a nova coluna. No entanto, às vezes é útil criar uma nova coluna inicializada com um único valor, logo, é bom saber como fazê-lo.

Estamos prontos para a codificação one-hot. Podemos executá-la passando uma série (Series) de um DataFrame para a função `get_dummies()` do pandas. A função recebeu esse nome porque as colunas de codificação one-hot também são chamadas de **variáveis dummy**. O resultado será um novo DataFrame, com um número igual de colunas e níveis da variável categórica.

5. Execute esse código para criar um DataFrame de codificação one-hot da coluna `EDUCATION_CAT`. Examine as 10 primeiras linhas:

```
edu_oh = pd.get_dummies(df_clean_2['EDUCATION_CAT'])  
edu_oh.head(10)
```

O código deve produzir a saída a seguir:

```
edu_oh = pd.get_dummies(df_clean_2['EDUCATION_CAT'])  
edu_oh.head(10)
```

	graduate school	high school	none	others	university
0	0	0	0	0	1
1	0	0	0	0	1
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0
7	0	0	0	0	1
8	0	1	0	0	0
9	0	1	0	0	0

Figura 1.53 – Data frame de codificação one-hot.

Agora podemos ver por que essa abordagem chama-se “codificação one-hot”: em todas as colunas, qualquer linha específica terá um número 1 em exatamente 1 coluna e zeros nas outras. Em uma linha específica, a coluna com o 1 deve corresponder ao nível da variável categórica original. Para verificar isso, precisamos concatenar esse novo DataFrame com o original e examinar os resultados lado a lado. Usaremos a função `concat` do pandas, para a qual passaremos a lista de DataFrames que queremos concatenar e a palavra-chave

`axis=1` solicitando que eles sejam concatenados horizontalmente, isto é, ao longo do eixo da coluna. Basicamente, isso significa que estamos combinando esses dois DataFrames “lado a lado”, o que sabemos que podemos fazer porque acabamos de criar esse novo DataFrame a partir do original: sabemos que ele terá o mesmo número de linhas, que estarão na mesma ordem do DataFrame original.

6. Concatene o DataFrame de codificação one-hot com o original da seguinte forma:

```
df_with_oh = pd.concat([df_clean_2, edu_oh], axis=1)
df_with_oh[['EDUCATION_CAT', 'graduate school',
            'high school', 'university', 'others']].head(10)
```

Você deve ver esta saída:

```
df_with_oh = pd.concat([df_clean_2, edu_oh], axis=1)
df_with_oh[['EDUCATION_CAT', 'graduate school',
            'high school', 'university', 'others']].head(10)
```

	EDUCATION_CAT	graduate school	high school	university	others
0	university	0	0	1	0
1	university	0	0	1	0
2	university	0	0	1	0
3	university	0	0	1	0
4	university	0	0	1	0
5	graduate school	1	0	0	0
6	graduate school	1	0	0	0
7	university	0	0	1	0
8	high school	0	1	0	0
9	high school	0	1	0	0

Figura 1.54 – Verificando as colunas de codificação one-hot.

Certo, parece que funcionou como esperado. A OHE é outra maneira de codificar características categóricas que evita a estrutura numérica inerente a uma codificação ordinal. No entanto, observe o que ocorreu aqui: pegamos uma única coluna, `EDUCATION`, e a explodimos em um número de colunas igual aos dos níveis da característica. Nesse caso, já que há apenas quatro níveis, não foi tão difícil. Porém, se sua variável categórica tivesse um número muito grande de níveis, seria melhor considerar uma estratégia alternativa, como agrupar alguns níveis em categorias individuais.

É um bom momento para salvarmos o DataFrame criado aqui, que resume nossos esforços de limpar os dados e adicionar uma coluna OHE.

Selecione um nome de arquivo e grave o último DataFrame em um arquivo CSV (de valor separado por vírgulas) desta forma: `df_with_ohe.to_csv('../Data/Chapter_1_cleaned_data.csv', index=False)`; não incluímos o índice, já que isso não é necessário e pode gerar colunas adicionais quando fizermos o carregamento posteriormente.

Explorando as características de histórico financeiro do dataset

Estamos prontos para explorar o resto das características do dataset do estudo de caso. Primeiro, tentaremos carregar um DataFrame a partir do arquivo **CSV** que salvamos no fim da última seção. Isso pode ser feito com o uso do fragmento a seguir:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Observe que, se você continuar escrevendo código no mesmo notebook, isso sobreporá o valor contido anteriormente na variável `df`, que era o DataFrame de dados brutos. Recomendamos que verifique as funções `.head()`, `.columns` e `.shape` do DataFrame. São itens importantes a verificar sempre que carregamos um DataFrame. Não o faremos aqui por falta de espaço, mas fizemos no notebook que acompanha o livro.

Nota: O caminho de seu arquivo CSV pode ser diferente dependendo de onde você o salvou.

As características que faltam ser examinadas são as de histórico financeiro. Elas se encaixam naturalmente em três grupos: o status dos pagamentos mensais nos últimos seis meses e as quantias cobradas e pagas no mesmo período. Primeiro examinaremos o status dos pagamentos. É conveniente relacioná-los em uma lista para podermos estudá-los em conjunto.

```
pay_feats = ['PAY_1', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6']
```

Podemos usar o método `.describe` nessas seis séries para examinar sínteses estatísticas:

```
df[pay_feats].describe()
```

Isso deve produzir a saída a seguir:

```
df[pay_feats].describe()
```

	PAY_1	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6
count	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000
mean	-0.017777	-0.133363	-0.167679	-0.225023	-0.269764	-0.293579
std	1.126769	1.198640	1.199165	1.167897	1.131735	1.150229
min	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000
25%	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	8.000000	8.000000	8.000000	8.000000	8.000000	8.000000

Figura 1.55 – Síntese estatística das características de status de pagamento.

Aqui, podemos ver que o intervalo de valores é o mesmo para todas as características: -2, -1, 0, ... 8. Parece que o valor 9, descrito no dicionário de dados como “atraso de nove meses ou mais no pagamento”, nunca ocorre.

Já explicamos o significado de todos esses valores, alguns dos quais não se encontram no dicionário de dados original. Agora examinaremos novamente as contagens de valores de PAY_1, dessa vez classificada pelos valores que estamos contando, que são o índice desta série:

```
df[pay_feats[0]].value_counts().sort_index()
```

Esse código deve produzir a seguinte saída:

```
df[pay_feats[0]].value_counts().sort_index()
-2      2476
-1      5047
0      13087
1       3261
2       2378
3        292
4         63
5         23
6         11
7          9
8         17
Name: PAY_1, dtype: int64
```

Figura 1.56 – Contagens de valores do status dos pagamentos do mês anterior.

Em comparação com os valores inteiros positivos, a maioria dos valores são -2, -1 ou 0, que correspondem a uma conta que estava em boa situação no mês passado: não usada, totalmente paga ou com pelo menos o pagamento mínimo feito.

Observe que, devido à definição dos outros valores dessa variável (1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento e assim por diante), essa característica é como um híbrido de características categóricas e numéricas. Por que a não utilização do crédito deve corresponder a um valor igual a -2, enquanto o valor 2 significa atraso de 2 meses no pagamento etc.? Precisamos entender que a codificação numérica de status de pagamento -2, -1 e 0 constitui uma decisão tomada pelo criador do dataset sobre como codificar certas características categóricas, que seriam agrupadas em uma característica que na verdade é numérica: o número de meses de atraso no pagamento (valores igual a 1 e maiores). Posteriormente, consideraremos os possíveis efeitos dessa forma de fazer as coisas sobre a capacidade preditiva dessa característica.

Por enquanto, continuaremos simplesmente explorando os dados. Esse dataset é suficientemente pequeno, com 18 características financeiras e algumas outras, para podermos examinar individualmente cada característica. Se ele tivesse milhares de características, provavelmente abandonaríamos essa abordagem e exploraríamos técnicas de **redução de dimensionalidade**, que são maneiras de condensar as informações de um grande número de características em um número menor de características derivadas, ou, alternativamente, métodos de **seleção de características**, que podem ser usados para isolar as características importantes a partir de um grupo com várias candidatas. Demonstraremos e explicaremos algumas técnicas de seleção de características posteriormente. Nesse dataset, é viável visualizar cada característica. Como vimos no último capítulo, um histograma é uma boa maneira de obtermos uma interpretação visual rápida do mesmo tipo de informações que obteríamos com tabelas de contagens de valores. Você pode usá-lo nas características de status de pagamento do último mês com `df[pay_feats[0]].hist()`, para produzir a saída mostrada na Figura 1.57.

Agora examinaremos com detalhes como esse gráfico foi produzido e consideraremos se ele é tão informativo quanto deveria. Um ponto-chave sobre a funcionalidade gráfica do pandas é que **na verdade a sua plotagem chama o matplotlib em segundo plano**. Observe que o último argumento disponível para o método `.hist()` do pandas é `**kwargs`, que a documentação indica serem argumentos de palavra-chave do `matplotlib`.

Nota: Para obter mais informações, consulte a página a seguir: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html>.

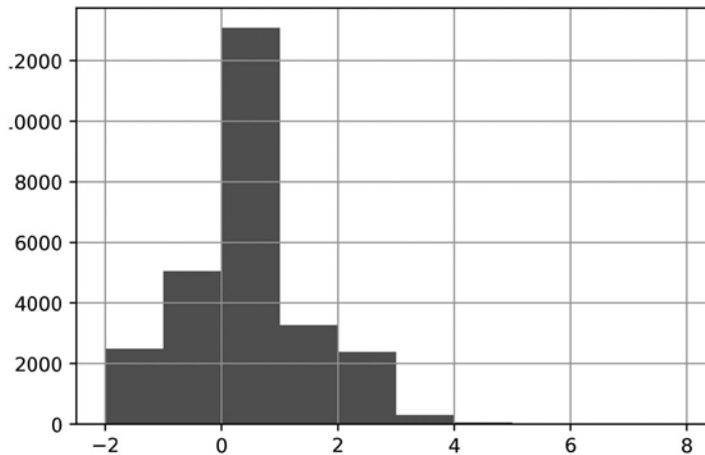


Figura 1.57 – Histograma de PAY_1 usando argumentos padrão.

Se examinarmos a documentação do `matplotlib` no que diz respeito a `matplotlib.pyplot.hist`, ela mostra argumentos adicionais que podemos usar com o método `.hist()` do pandas, como o tipo de histograma a ser plotado (consulte https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html para ver mais detalhes). Em geral, para a obtenção de mais detalhes sobre a funcionalidade de plotagem, é importante incluir o `matplotlib`, e, em alguns cenários, é melhor usá-lo diretamente, em vez do pandas, se você quiser ter mais controle sobre a aparência das plotagens.

É bom ressaltar que o pandas usa o `matplotlib`, que por sua vez usa o NumPy. Na plotagem de histogramas com o `matplotlib`, o cálculo numérico dos valores que compõem o histograma na verdade é executado pela função `.histogram` do NumPy. Esse é um exemplo típico de reutilização de código, ou de “não reinventar a roda”. Se uma funcionalidade padrão, como a plotagem de um histograma, já tem uma boa implementação em Python, não há razão para criar outra. E, se a operação matemática que cria os dados do histograma já foi implementada, ela também deve ser utilizada. Isso mostra a interconectividade do ecossistema Python.

Agora abordaremos algumas questões importantes que surgem no cálculo e na plotagem de histogramas.

Número de bins

Os histogramas funcionam agrupando valores no que chamamos de **bins**. O número de bins é o número de barras verticais que compõem a plotagem discreta do histograma que vemos. Se houver uma grande quantidade de valores exclusivos em uma escala contínua, como no histograma de faixas etárias que visualizamos

anteriormente, a plotagem do histograma funcionará relativamente bem “de forma imediata”, com argumentos padrão. No entanto, quando o número de valores exclusivos é quase igual ao número de bins, os resultados podem ser duvidosos. O número padrão de bins é 10, enquanto, na característica `PAY_1`, há 11 valores exclusivos. Em casos assim, é melhor definir manualmente o número de bins do histograma para que ele seja igual ao número de valores exclusivos.

Em nosso exemplo atual, já que há poucos valores nos bins mais altos de `PAY_1`, a plotagem pode não ter uma aparência tão diferente. Porém, em geral, é importante lembrar-se disso ao plotar histogramas.

Bordas dos bins

Os locais das bordas dos bins determinam como os valores foram agrupados no histograma. Em vez de indicar o número de bins para a função de plotagem, alternativamente você poderia fornecer uma lista ou um array de números para o argumento de palavra-chave `bins`. Essa entrada seria interpretada como os locais das bordas dos bins no eixo x. É importante entendermos a maneira como os valores são agrupados nos bins no `matplotlib`, com o uso dos locais das bordas. Todos os bins, exceto o último, agrupam valores tão baixos quanto os da borda esquerda e que vão até, **mas sem incluir**, valores tão altos quanto os da borda direita. Em outras palavras, a borda esquerda é fechada, mas a direita é aberta. No entanto, o último bin inclui as duas bordas; ele tem as bordas esquerda e direita fechadas. Isso tem muita importância na prática quando agrupamos uma quantidade relativamente pequena de valores exclusivos que podem coincidir com as bordas dos bins.

Para termos controle sobre a aparência da plotagem, em geral é melhor especificar os locais das bordas dos bins. Criaremos um array de 12 números, que resultará em 11 bins, dedicados a cada um dos valores exclusivos de `PAY_1`:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins
```

A saída mostra os locais das bordas dos bins:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins  
array([-2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  
        8.5])
```

Figura 1.58 – Especificando as bordas dos bins do histograma.

Como última observação sobre o estilo, é importante sempre *rotular suas plotagens* para que elas sejam interpretáveis. Não fizemos isso manualmente, porque, em alguns casos, o pandas o faz automaticamente, e, em outros, apenas deixamos as plotagens não rotuladas. De agora em diante, seguiremos a melhor prática e rotularemos todas as plotagens. Usaremos as funções `xlabel` e `ylabel` do `matplotlib` para adicionar rótulos de eixos a essa plotagem. O código é o seguinte:

```
df[pay_feats[0]].hist(bins=pay_1_bins)
plt.xlabel('PAY_1')
plt.ylabel('Number of accounts')
```

A saída deve ser esta:

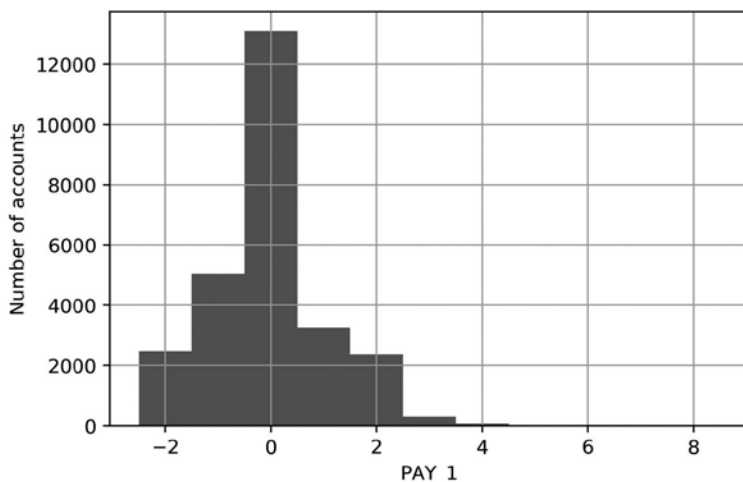


Figura 1.59 – Um histograma melhor para PAY_1.

Embora seja tentador, e com frequência suficiente, chamar as funções de plotagem com os argumentos padrão, uma de suas funções como cientista de dados é criar *visualizações de dados precisas e representativas*. Para isso, às vezes é necessário manipular os detalhes do código de plotagem, como fizemos aqui.

O que aprendemos com essa visualização de dados?

Como vimos nas contagens de valores, essa visualização confirma que a maioria das contas está em boa situação (valores -2, -1 e 0). Para as que não estão, é mais comum que o “atraso em meses” tenha um número menor. Isso faz sentido; provavelmente, a maioria das pessoas está pagando sua dívida sem demorar muito. Caso contrário, sua conta pode ser fechada ou passada para uma agência de cobrança.

Examinar a distribuição das características e verificar se ela parece razoável é algo bom para se confirmar com o cliente, já que a qualidade desses dados formará a base da modelagem preditiva que você pretende executar.

Agora que estabelecemos um bom estilo de plotagem para os histogramas, usaremos o pandas para plotar vários histogramas em conjunto e visualizar as características de status de pagamento para cada um dos últimos seis meses. Podemos passar nossa lista de nomes de colunas `pay_feats` para acessar várias colunas a serem plotadas com o método `.hist()`, especificando as bordas de bins que já determinamos e indicando que gostaríamos de uma grade de plotagens 2 por 3. Primeiro, definiremos um tamanho de fonte suficientemente pequeno para caber entre essas **subplotagens**. Aqui está o código que fará isso:

```
mpl.rcParams['font.size'] = 4  
df[pay_feats].hist(bins=pay_1_bins, layout=(2,3))
```

Os títulos das plotagens foram criados automaticamente para nós com base nos nomes das colunas. Os eixos y são considerados como contagens. As visualizações resultantes são as seguintes:

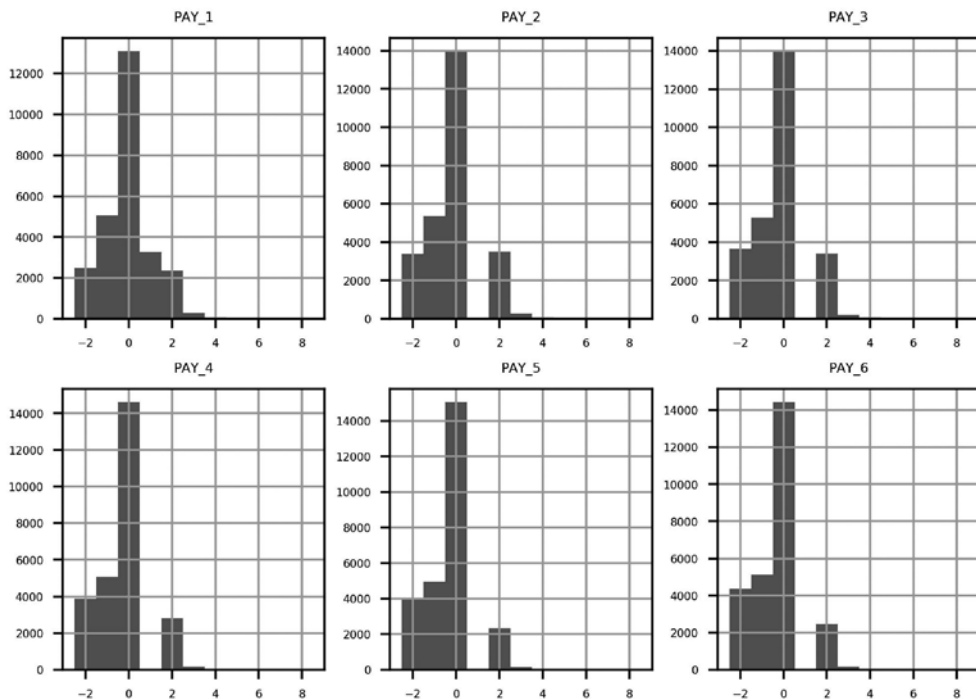


Figura 1.60 – Grade de subplotagens do histograma.

Já vimos a primeira dessas visualizações, e ela faz sentido. E o resto? Lembre-se das definições dos valores inteiros positivos dessas características e do que cada característica significa. Por exemplo, `PAY_2` é o status de reembolso em agosto, `PAY_3` é o status de reembolso em julho, e as outras vão voltando no tempo. Um valor igual a 1 significa atraso de 1 mês no pagamento, enquanto o valor 2 significa atraso de dois meses e assim por diante.

Notou que algo não parece certo? Considere os valores entre julho (`PAY_3`) e agosto (`PAY_2`). Em julho, poucas contas tiveram atraso de 1 mês no pagamento; essa barra não pode ser vista no histograma. No entanto, em agosto, repentinamente há milhares de contas com atraso de 2 meses no pagamento. Isso não faz sentido: o número de contas com atraso de 2 meses em um mês específico deveria ser menor ou igual ao número de contas com atraso de 1 mês no mês anterior. Examinaremos com mais detalhes as contas com atraso de 2 meses em agosto e veremos qual foi o status de pagamento em julho. Podemos fazer isso com o código a seguir, usando uma máscara booleana e `.loc`, como mostrado neste fragmento:

```
df.loc[df['PAY_2']==2, ['PAY_2', 'PAY_3']].head()
```

A saída desse código é:

```
df.loc[df['PAY_2']==2, ['PAY_2', 'PAY_3']].head()
```

	PAY_2	PAY_3
0	2	-1
1	2	0
13	2	2
15	2	0
47	2	2

Figura 1.61 – Status de pagamento em julho (`PAY_3`) de contas com atraso de 2 meses no pagamento em agosto (`PAY_2`).

Na *Figura 1.61*, fica claro que as contas com atraso de 2 meses em agosto têm valores absurdos para o status de pagamento de julho. A única maneira de chegarmos a um atraso de 2 meses seria haver um atraso de um mês no mês anterior, mas nenhuma dessas contas indica isso.

Quando você vir algo assim nos dados, terá de verificar a lógica da consulta usada para criar o dataset ou entrar em contato com a pessoa que o forneceu. Após confirmar os resultados, por exemplo usando `.value_counts()` para visualizar os números diretamente, entramos em contato com nosso cliente para perguntar sobre esse problema.

O cliente informou que está tendo problemas para obter dados do último mês, o que tem gerado relatórios incorretos para contas que têm atraso de 1 mês no pagamento. Em setembro, ele resolveu grande parte dos problemas (mas não totalmente; é por isso que há valores faltando na característica `PAY_1`, como descobrimos). Logo, em nosso dataset, o valor 1 foi subnotificado em todos os meses exceto setembro (a característica `PAY_1`). Teoricamente, o cliente poderia criar uma consulta para fazer uma nova pesquisa em seu banco de dados e determinar os valores corretos para `PAY_2`, `PAY_3` e assim por diante até `PAY_6`. No entanto, por razões práticas, ele não poderá concluir essa análise retrospectiva a tempo de a recebermos e incluímos em nossa análise.

Portanto, só o mês mais recente de nossos dados de status de pagamento está correto. Isso significa que, de todas as características de status de pagamento, só `PAY_1` é representativa de dados futuros, aqueles que serão usados para fazermos previsões com o modelo que desenvolvemos. Esse é um ponto-chave: *um modelo preditivo depende da obtenção do mesmo tipo de dado para fazer as previsões para as quais foi treinado*. Ou seja, podemos usar `PAY_1` como característica em nosso modelo, mas não `PAY_2` ou as outras características de status de pagamento de meses anteriores.

Esse episódio mostra a importância de uma verificação abrangente da qualidade dos dados. Só descobrimos o problema vasculhando cuidadosamente os dados. Seria bom se o cliente tivesse informado antecipadamente que estava tendo problemas de relatório nos últimos meses, quando nosso dataset foi coletado, e que o procedimento de geração de relatórios não era **consistente** durante esse período. No entanto, no fim das contas é nossa responsabilidade construir um modelo confiável, logo, temos de nos certificar se os dados estão corretos, fazendo esse tipo de exame cuidadoso. Explicaremos ao cliente que não podemos usar as características mais antigas, já que elas não são representativas dos dados futuros nos quais o modelo **se baseará** (isto é, fará previsões sobre meses futuros) e solicitaremos educadamente que nos informem sobre qualquer problema adicional nos dados do qual tenham conhecimento. No momento não há nenhum.

Atividade 1: Explorando as características financeiras restantes do dataset

Nessa atividade, você examinará as características financeiras restantes de maneira semelhante a como examinamos `PAY_1`, `PAY_2`, `PAY_3` e assim por diante. Para visualizar melhor alguns desses dados, usaremos uma função matemática que deve ser familiar: o logaritmo. Você usará o pandas com `apply`, que serve para aplicar qualquer função a uma coluna ou DataFrame inteiro. Ao concluir a atividade, deverá ter o conjunto a seguir de histogramas de transformações logarítmicas de pagamentos diferentes de zero:

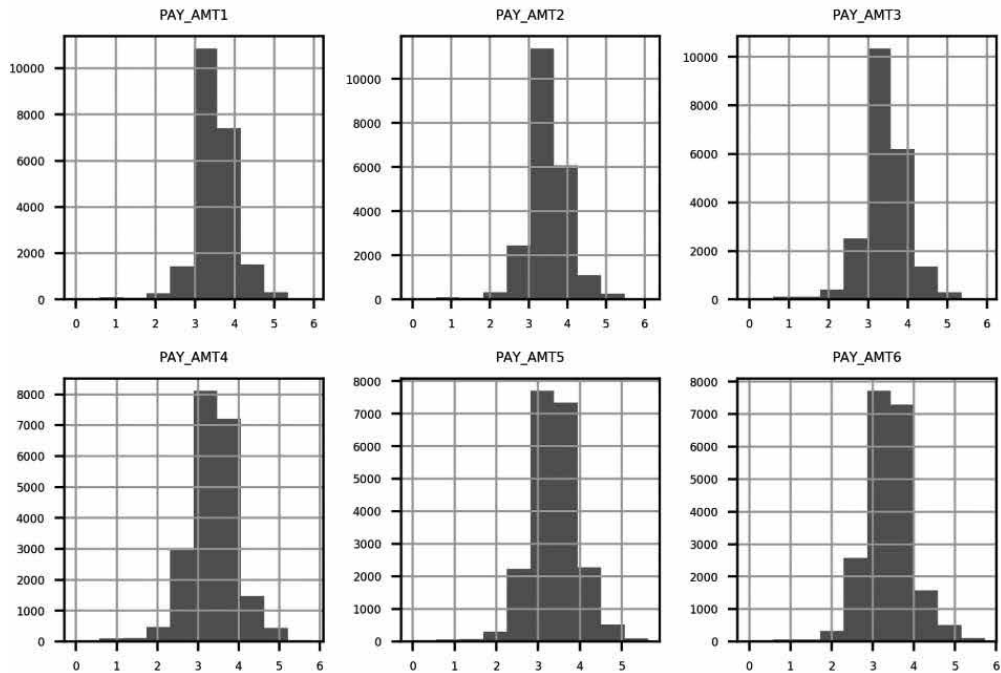


Figura 1.62 – Conjunto esperado de histogramas.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e o gráfico de saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2TXZmrA>.

1. Crie listas com nomes para as características financeiras restantes.
2. Use `.describe()` para examinar as sínteses estatísticas das características de valor da fatura. Reflita sobre o que viu. Faz sentido?
3. Visualize as características de valor da fatura usando uma grade 2 por 3 de plotagens de histograma.
Dica: Você pode usar 20 bins para essa visualização.
4. Obtenha o resumo de `.describe()` para as características de valor do pagamento. Faz sentido?
5. Plote um histograma das características de pagamento da fatura semelhante ao das características de valor da fatura, mas aplique também alguma rotação aos rótulos do eixo x com o argumento de palavra-chave `xrot` para que eles não

se sobreponham. Podemos incluir o argumento de palavra-chave `xrot=<ângulo>` em qualquer função de plotagem para girar os rótulos do eixo x de acordo com um ângulo específico em graus. Considere os resultados.

6. Use uma máscara booleana para ver quantos dos dados de valor do pagamento são exatamente iguais a 0. O resultado faz sentido dado o histograma da etapa anterior?
7. Ignorando os pagamentos iguais a 0 usando a máscara que criou na etapa anterior, utilize o método `.apply()` do pandas e o método `np.log10()` do NumPy para plotar histogramas de transformações logarítmicas dos pagamentos diferentes de zero. Considere os resultados.

Dica: Você pode usar `.apply()` para aplicar qualquer função, inclusive `log10`, a todos os elementos de um DataFrame ou de uma coluna usando a sintaxe a seguir: `.apply(<nome_função>)`.

Nota: A solução dessa atividade pode ser encontrada na página 290.

Resumo

Este foi o primeiro capítulo de nosso livro, *Projetos de ciência de dados com Python*. Aqui, usamos bastante o pandas para carregar e explorar os dados do estudo de caso. Aprendemos como verificar a consistência básica e a precisão usando uma combinação de sínteses estatísticas e visualizações. Respondemos a perguntas como “Os IDs de conta exclusivos são realmente exclusivos?”, “Há dados ausentes que receberam um valor de preenchimento?” e “Os valores das características fazem sentido dada sua definição?”.

Observe que passamos quase o capítulo inteiro identificando e corrigindo problemas em nosso dataset. Com frequência, esse é o estágio mais demorado de um projeto de ciência de dados. Embora nem sempre seja a parte mais empolgante do trabalho, ela fornece os materiais brutos necessários para a construção de modelos e insights interessantes. Esses serão os assuntos de grande parte do resto do livro.

O domínio de ferramentas de software e conceitos matemáticos é o que nos permite executar projetos de ciência de dados, em um nível técnico. No entanto, gerenciar o relacionamento com os clientes, que contam com nossos serviços para gerar insights a partir dos dados, é igualmente importante para um projeto bem-sucedido. Você deve usar o máximo possível o conhecimento que o sócio da empresa

tem dos dados. Provavelmente ele estará mais familiarizado, a não ser que você seja um especialista nos dados do projeto que está executando. Contudo, mesmo nesse caso, sua primeira etapa deve ser uma revisão abrangente e crítica dos dados que está usando.

Em nossa exploração dos dados, descobrimos um problema que poderia ter prejudicado o projeto: os dados que recebemos não eram consistentes internamente. A maioria dos meses das características de status de pagamento estava contaminada por um problema de relatório de dados, incluía valores absurdos e não era representativa dos dados mensais mais recentes ou dos dados que estariam disponíveis para o modelo avançar. Só descobrimos esse problema examinando cuidadosamente todas as características. Embora nem sempre isso seja possível em vários projetos, principalmente quando há um número muito grande de características, você deve tentar inspecionar quantas características puder. Se não puder examinar todas, será útil verificar algumas de cada categoria (se as características se encaixarem em categorias, como financeiras ou demográficas).

Ao discutir problemas de dados como esse com seu cliente, certifique-se de ser respeitoso e profissional. O cliente pode simplesmente ter esquecido o problema ao apresentar os dados. Ou talvez soubesse sobre ele, mas presumiu por alguma razão que não afetaria a análise. Seja como for, você está prestando um serviço essencial mostrando o problema e explicando por que seria incorreto usar dados inadequados para construir um modelo. Você deve apoiar suas alegações em resultados se possível, mostrando que usar dados incorretos leva a um desempenho fraco, ou inalterado, do modelo. Ou, alternativamente, poderia explicar que se um tipo de dado diferente só estiver disponível no futuro, em comparação com os que estão disponíveis no momento para o treinamento, o modelo construído agora não será útil. Seja tão específico quanto puder, apresentando os tipos de gráficos e tabelas que usamos aqui para descobrir o problema nos dados.

No próximo capítulo, examinaremos a variável de resposta de nosso problema de estudo de caso, o que conclui a exploração inicial dos dados. Em seguida, começaremos a ganhar experiência prática com modelos de machine learning e aprenderemos como decidir se um modelo é ou não útil.