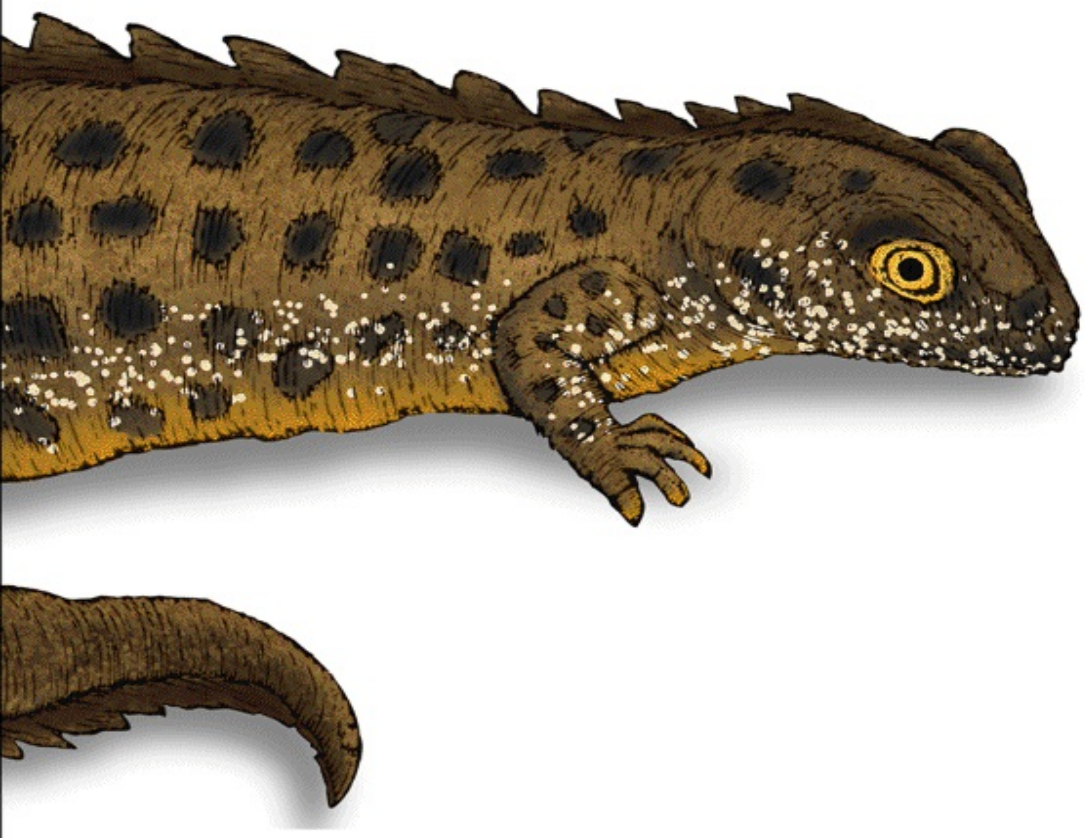
O'REILLY®

Machine Learning

Guia de Referência Rápida

Trabalhando com dados estruturados em Python



novatec

Matt Harrison

Machine Learning

Guia de Referência Rápida

Trabalhando com dados estruturados em Python

Matt Harrison

O'REILLY
Novatec
São Paulo | 2020

Tradução em português autorizada da edição em inglês da obra Machine Learning Pocket Reference, ISBN 9781492047544 © 2019 Matt Harrison. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Tássia Carvalho
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-818-0

Histórico de edições impressas:

Janeiro/2020 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

Machine learning (aprendizado de máquina) e ciência de dados (data science) são áreas muito populares atualmente, mas estão em rápida evolução. Já trabalhei com Python e dados em boa parte de minha carreira, e sempre quis ter um livro que contivesse uma referência aos métodos comuns que venho usando no mercado de trabalho e em sessões de treinamento em workshops para resolver problemas de machine learning com dados estruturados.

Este livro é o que acredito ser a melhor coleção de recursos e exemplos para encarar uma tarefa de modelagem preditiva caso você tenha dados estruturados. Há muitas bibliotecas que executam parte das tarefas necessárias, e procurei incluir aquelas que considerei úteis quando apliquei essas técnicas em consultorias ou trabalhando no mercado.

Muitas pessoas talvez lamentem a ausência de técnicas de deep learning (aprendizagem profunda). Essas técnicas mereceriam um livro próprio. Também prefiro técnicas mais simples, e outros profissionais no mercado parecem concordar comigo: deep learning para dados não estruturados (vídeo, áudio, imagens) e ferramentas eficazes, como o XGBoost, para dados estruturados.

Espero que este livro sirva como uma referência útil para você resolver seus problemas prementes.

O que esperar deste livro

O livro apresenta exemplos detalhados para solucionar problemas comuns envolvendo dados estruturados. Várias ferramentas e modelos serão apresentados, além de seu custo-benefício e como ajustar e interpretar esses modelos.

Os trechos de código foram dimensionados de modo que você possa usá-los e adaptá-los de acordo com seus projetos.

A quem este livro se destina

Não importa se você apenas começou a aprender machine learning ou se já vem trabalhando com ele há anos, este livro deve servir como uma boa referência. Algum conhecimento de Python é pressuposto, e sua sintaxe não será explicada. O livro mostra como usar diversas bibliotecas para resolver problemas do mundo real.

Esta obra não substituirá um curso detalhado, mas servirá como uma referência para o que um curso aplicado de machine learning deveria abordar. (Nota: o autor utiliza este livro como referência para os cursos de análise de dados e de machine learning ministrados por ele.)

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas no livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

DICA

Este elemento significa uma dica ou sugestão.

NOTA

Este elemento significa uma observação geral.

ALERTA

Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O’Reilly

Este livro está aqui para ajudá-lo a fazer seu trabalho. Se o código de exemplo for útil, você poderá usá-lo em seus programas e em sua

documentação. Não é necessário nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. No entanto, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer.

Agradecemos, mas não exigimos, atribuição, o que geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: "*Machine Learning Pocket Reference* by Matt Harrison (O'Reilly). Copyright 2019 Matt Harrison, 978-1-492-04754-4."

Se achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade em nos contatar em *permissions@oreilly.com*.

Como entrar em contato

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português
  https://novatec.com.br/livros/machine-learning-guia-referencia/
- Página da edição original em inglês
  http://www.oreilly.com/catalog/9781492047544
- Página com material suplementar (exemplos de códigos, exercícios etc.).
  https://github.com/mattharrison/ml_pocket_reference

Para obter mais informações sobre os livros da Novatec, acesse nosso site: *http://www.novatec.com.br*

Agradecimentos

Agradeço muito à minha esposa e minha família pelo apoio. Sou muito grato à comunidade Python por oferecer uma linguagem maravilhosa e um

conjunto de ferramentas para trabalhar com ela. Foi ótimo trabalhar com Nicole Tache, que me deu um excelente feedback. Meus revisores técnicos Mikio Braun, Natalino Busa e Justin Francis me permitiram ser verdadeiro. Obrigado!

CAPÍTULO 1

Introdução

O propósito deste livro não é ser um manual de instruções, mas uma obra que contém notas, tabelas e exemplos de machine learning (aprendizado de máquina). Foi escrito pelo autor para ser um recurso complementar a um curso, com o intuito de ser distribuído e servir como se fosse um bloco de anotações. Os participantes (que sejam a favor de materiais feitos com árvores mortas) poderão fazer suas próprias anotações e acrescentar suas ideias; desse modo, poderão ter uma referência importante, contendo exemplos bem organizados.

Descreveremos as classificações usando dados estruturados. Outras aplicações comuns de machine learning incluem previsão de um valor contínuo (regressão), criação de clusters (agrupamentos) ou tentativas de reduzir dimensões, entre outras. Este livro não discutirá as técnicas de deep learning (aprendizagem profunda). Embora estas funcionem bem para dados não estruturados, a maioria das pessoas recomenda as técnicas presentes neste livro para dados estruturados.

Partimos do pressuposto de que você tenha conhecimento e familiaridade com Python. Saber como manipular dados usando a biblioteca pandas (*https://pandas.pydata.org*) será conveniente. Temos muitos exemplos de uso do pandas, que é uma ótima ferramenta para lidar com dados estruturados. No entanto, algumas operações de indexação poderão ser confusas caso você não tenha familiaridade com o numpy. Uma descrição completa do pandas, por si só, mereceria um livro completo.

Bibliotecas usadas

Este livro utiliza várias bibliotecas. Isso pode ser bom ou ruim. Algumas dessas bibliotecas talvez sejam difíceis de instalar, ou poderão apresentar conflitos com versões de outras bibliotecas. Não considere que seja

necessário instalar todas as bibliotecas a seguir. Utilize uma “instalação JIT” e instale apenas as bibliotecas que você quiser usar à medida que precisar delas.

```
>>> import autosklearn, catboost, category_encoders, dtreeviz,
eli5, fancyimpute, fastai, featuretools, glmnet_py, graphviz,
hdbscan, imblearn, janitor, lime, matplotlib, missingno, mlxtend,
numpy, pandas, pdpbox, phate, pydotplus, rfpimp, scikitplot,
scipy, seaborn, shap, sklearn, statsmodels, tpot,
treeinterpreter, umap, xgbfir, xgboost, yellowbrick

>>> for lib in [
... autosklearn,
... catboost,
... category_encoders,
... dtreeviz,
... eli5,
... fancyimpute,
... fastai,
... featuretools,
... glmnet_py,
... graphviz,
... hdbscan,
... imblearn,
... lime,
... janitor,
... matplotlib,
... missingno,
... mlxtend,
... numpy,
... pandas,
... pandas_profiling,
... pdpbox,
... phate,
... pydotplus,
... rfpimp,
... scikitplot,
... scipy,
... seaborn,
... shap,
... sklearn,
... statsmodels,
... tpot,
... treeinterpreter,
... umap,
```

```
... xgbfir,
... xgboost,
... yellowbrick,
... ]:
... try:
... print(lib.__name__, lib.__version__)
... except:
... print("Missing", lib.__name__)
catboost 0.11.1
category_encoders 2.0.0
Missing dtreeviz
eli5 0.8.2
fancyimpute 0.4.2
fastai 1.0.28
featuretools 0.4.0
Missing glmnet_py
graphviz 0.10.1
hdbscan 0.8.22
imblearn 0.4.3
janitor 0.16.6
Missing lime
matplotlib 2.2.3
missingno 0.4.1
mlxtend 0.14.0
numpy 1.15.2
pandas 0.23.4
Missing pandas_profiling
pdpbox 0.2.0
phate 0.4.2
Missing pydotplus
rfpimp
scikitplot 0.3.7
scipy 1.1.0
seaborn 0.9.0
shap 0.25.2
sklearn 0.21.1
statsmodels 0.9.0
tpot 0.9.5
treeinterpreter 0.1.0
umap 0.3.8
xgboost 0.81
yellowbrick 0.9
```

NOTA

A maior parte dessas bibliotecas pode ser facilmente instalada usando pip ou conda. Com o fastai, precisei usar pip install --no-deps fastai. A biblioteca umap foi instalada com pip install umap-learn, a biblioteca janitor com pip install pyjanitor e a biblioteca autosklearn com pip install auto-sklearn.

Em geral, utilizo o Jupyter para fazer análises. Você pode usar outras ferramentas de notebook também. Note que algumas, como o Google Colab, têm várias bibliotecas pré-instaladas (embora suas versões possam estar desatualizadas).

Há duas opções principais para instalar bibliotecas em Python. Uma delas é usar o pip (um acrônimo para Pip Installs Python, isto é, Pip instala Python), uma ferramenta que acompanha Python. A outra opção é usar o Anaconda (*https://anaconda.org*). Ambas serão apresentadas a seguir.

Instalação com o pip

Antes de usar o pip, criaremos um ambiente sandbox no qual instalaremos nossas bibliotecas. É um ambiente virtual que chamaremos de env:

```
$ python -m venv env
```

NOTA

No Macintosh e no Linux, utilize python; no Windows, use python3. Se o Windows não reconhecer esse comando no prompt, talvez seja necessário fazer uma reinstalação ou uma correção em sua instalação a fim de garantir que a opção “Add Python to my PATH” (Adicionar Python ao meu PATH) esteja marcada.

Em seguida, ative o ambiente para que, quando instalar suas bibliotecas, elas sejam colocadas no ambiente sandbox, e não na instalação global de Python. Como muitas dessas bibliotecas mudam e são atualizadas, é melhor isolar as versões por projeto, para que você saiba que seu código executará.

Eis o modo como ativamos o ambiente virtual no Linux e no Macintosh:

```
$ source env/bin/activate
```

Você perceberá que o prompt será atualizado, mostrando que estamos usando o ambiente virtual:

```
(env) $ which python
```

```
env/bin/python
```

No Windows, ative o ambiente executando o seguinte comando:

```
C:> env\Scripts\activate.bat
```

Novamente, você perceberá que o prompt será atualizado, mostrando que estamos usando o ambiente virtual:

```
(env) C:> where python
env\Scripts\python.exe
```

Você poderá instalar pacotes usando o pip em qualquer plataforma. Para instalar o pandas, digite:

```
(env) $ pip install pandas
```

Alguns dos nomes de pacotes são diferentes dos nomes das bibliotecas. Você pode procurar os pacotes usando:

```
(env) $ pip search nomedabiblioteca
```

Depois que tiver seus pacotes instalados, será possível criar um arquivo com todas as versões dos pacotes usando pip:

```
(env) $ pip freeze > requirements.txt
```

Com esse arquivo requirements.txt, você poderá instalar facilmente os pacotes em um novo ambiente virtual:

```
(other_env) $ pip install -r requirements.txt
```

Instalação com o conda

A ferramenta conda acompanha o Anaconda, e ela nos permite criar ambientes e instalar pacotes.

Para criar um ambiente chamado env, execute o seguinte:

```
$ conda create --name env python=3.6
```

Para ativar esse ambiente, execute:

```
$ conda activate env
```

Esse comando atualizará o prompt tanto em sistemas Unix como Windows. Agora você pode procurar pacotes usando:

```
(env) $ conda search nomedabiblioteca
```

Para instalar um pacote, por exemplo o pandas, execute:

```
(env) $ conda install pandas
```

Para criar um arquivo contendo os requisitos de pacotes, execute:

```
(env) $ conda env export > environment.yml
```

Para instalar esses requisitos em um novo ambiente, execute:

```
(other_env) $ conda create -f environment.yml
```

ALERTA

Algumas das bibliotecas mencionadas neste livro não estão disponíveis no repositório do Anaconda para instalação. Não se preocupe. Você pode usar o `pip` dentro de um ambiente conda (não é necessário criar outro ambiente virtual) para instalar essas bibliotecas.

CAPÍTULO 2

Visão geral do processo de machine learning

O CRISP-DM (Cross-Industry Standard Process for Data Mining, ou Processo Padrão do Mercado para Mineração de Dados) é um processo para fazer mineração de dados (data mining), o qual contém vários passos que podem ser seguidos para uma melhoria contínua. São eles:

- Entendimento do negócio (Business understanding);
- Entendimento dos dados (Data understanding);
- Preparação dos dados (Data preparation);
- Modelagem (Modeling);
- Avaliação (Evaluation);
- Implantação (Deployment).

A Figura 2.1 mostra meu fluxo de trabalho para criar um modelo preditivo que expande a metodologia CRISP-DM. O próximo capítulo descreverá esses passos básicos.

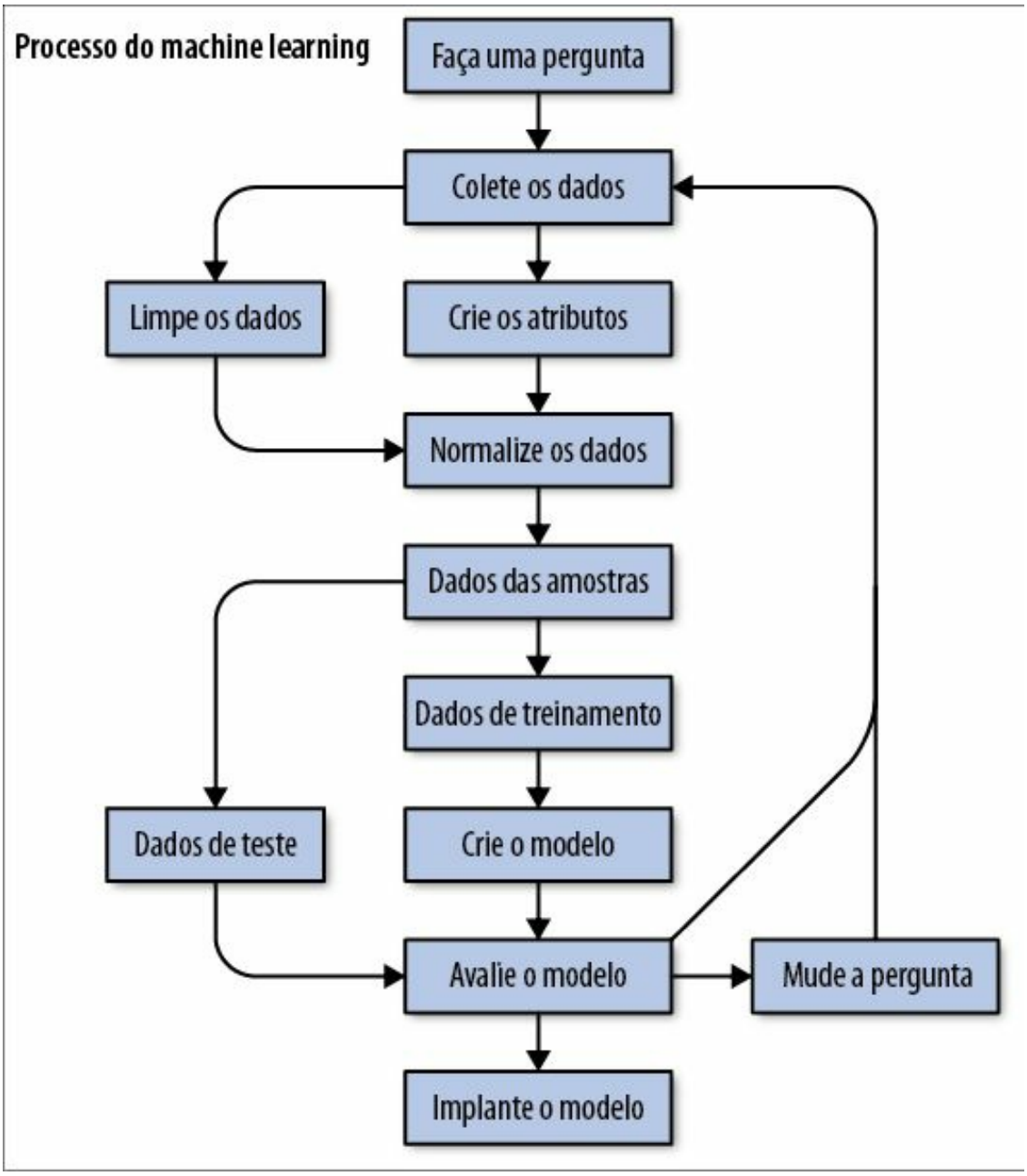


Figura 2.1 – Fluxo de trabalho comum no machine learning.

CAPÍTULO 3

Descrição da classificação: conjunto de dados do Titanic

Este capítulo descreverá um problema comum de classificação usando o conjunto de dados do Titanic (*https://oreil.ly/PjceO*). Outros capítulos mais adiante detalharão e expandirão os passos comuns executados em uma análise.

Sugestão para layout dos projetos

Uma ferramenta excelente para fazer uma análise de dados exploratória é o Jupyter (*https://jupyter.org*), um ambiente de notebook open source que aceita Python e outras linguagens. Ele permite que você crie *células* (cells) de código ou conteúdo Markdown.

Tenho a tendência de usar o Jupyter de duas maneiras. A primeira é para uma análise de dados exploratória, com o intuito de fazer verificações rápidas. A outra é um estilo mais voltado para uma apresentação, na qual formato um relatório usando células Markdown e insiro células de código para ilustrar pontos ou descobertas importantes. Se você não tomar cuidado, seus notebooks poderão precisar de refatoração e da aplicação de práticas de engenharia de software (remoção de variáveis globais, uso de funções e classes etc.).

O pacote cookiecutter para ciência de dados (*https://oreil.ly/86jL3*) sugere um layout para fazer uma análise que possibilite uma reprodução fácil e o compartilhamento de seu código.

Importações

O exemplo a seguir tem como base essencialmente o pandas (*http://pandas.pydata.org/*), o scikit-learn (*https://scikit-learn.org/*) e o

Yellowbrick (*http://www.scikit-yb.org/*). A biblioteca pandas nos fornece as ferramentas para facilitar a manipulação dos dados. A biblioteca scikit-learn faz uma ótima modelagem preditiva, enquanto o Yellowbrick é uma biblioteca de visualização para avaliação de modelos:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> from sklearn import (
... ensemble,
... preprocessing,
... tree,
... )
>>> from sklearn.metrics import (
... auc,
... confusion_matrix,
... roc_auc_score,
... roc_curve,
... )
>>> from sklearn.model_selection import (
... train_test_split,
... StratifiedKFold,
... )
>>> from yellowbrick.classifier import (
... ConfusionMatrix,
... ROCAUC,
... )
>>> from yellowbrick.model_selection import (
... LearningCurve,
... )
```

ALERTA

Você poderá ver documentações e exemplos online que incluam uma importação com asterisco, como esta:

```
from pandas import *
```

Evite usar importações com asterisco. Ser explícito deixa seu código mais fácil de entender.

Faça uma pergunta

Neste exemplo, queremos criar um modelo preditivo para responder a uma pergunta. Ele classificará se um indivíduo sobreviveu à catástrofe do navio Titanic com base nas características individuais e da viagem. É um exemplo

didático, mas serve como uma ferramenta pedagógica para demonstrar os vários passos da modelagem. Nosso modelo deverá ser capaz de prever se um passageiro sobreviveu ao Titanic a partir de informações sobre esse passageiro.

É uma pergunta de classificação, pois estamos fazendo a predição de um rótulo (label) acerca da sobrevivência: um passageiro sobreviveu ou morreu.

Termos para os dados

Em geral, fazemos o treinamento de um modelo com uma matriz de dados. (Prefiro usar os DataFrames do pandas porque é muito conveniente ter nomes para as colunas, mas os arrays numpy também servirão.)

Para uma aprendizagem supervisionada, como uma regressão ou classificação, nossa intenção é ter uma função que transforme atributos em um rótulo. Se fôssemos escrever isso como uma fórmula de álgebra, teríamos algo como:

```
y = f(X)
```

X é uma matriz. Cada linha representa uma *amostra* dos dados ou das informações acerca de um indivíduo. Cada coluna em X é um *atributo* (feature). A saída de nossa função, y, é um vetor que contém rótulos (em uma classificação) ou valores (em uma regressão) – veja a Figura 3.1.

Esse é um procedimento padrão de nomenclatura para nomear os dados e a saída. Se você ler artigos acadêmicos ou até mesmo consultar a documentação das bibliotecas, verá que elas seguem essa convenção. Em Python, usamos um nome de variável `X` para armazenar os dados das amostras, ainda que usar variáveis iniciadas com letra maiúscula seja uma violação das convenções de nomenclatura padrão (PEP 8). Não se preocupe, pois todos fazem isso; se você chamasse sua variável de `x`, todo mundo estranharia. A variável `y` armazena os rótulos (labels) ou objetivos (targets).

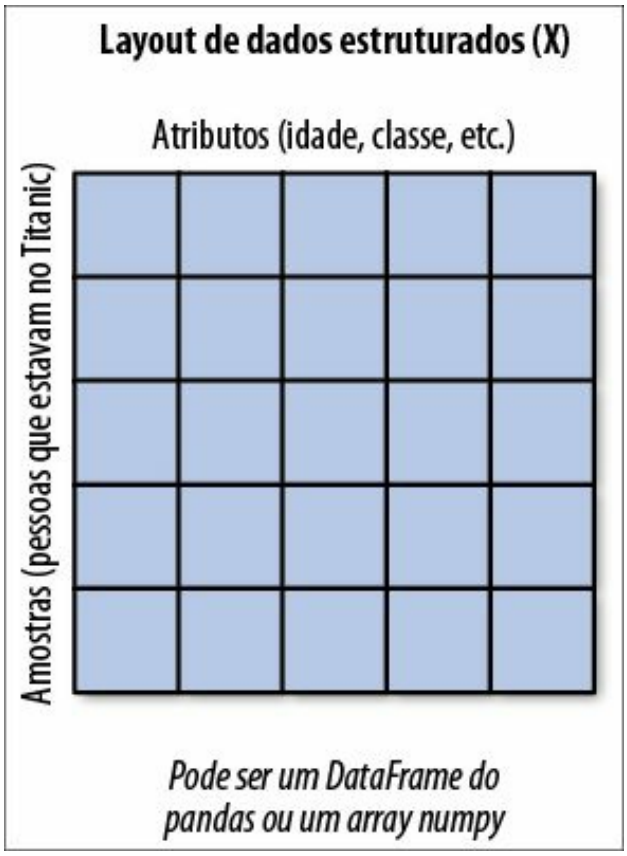


Figura 3.1 – Layout de dados estruturados.

A Tabela 3.1 mostra um conjunto de dados básico com duas amostras e três atributos para cada amostra.

Tabela 3.1 – Amostras (linhas) e atributos (colunas)

pclass	age	sibsp
1	29	0
1	2	1

Colete os dados

Carregaremos um arquivo Excel (certifique-se de ter o pandas e o xlrd[1] instalados) com os atributos das amostras do Titanic. Esse arquivo contém várias colunas, incluindo uma coluna relacionada à sobrevivência (survived), com um rótulo informando o que aconteceu com o indivíduo:

```
>>> url = (
... "http://biostat.mc.vanderbilt.edu/"
... "wiki/pub/Main/DataSets/titanic3.xls"
```

```
... )
>>> df = pd.read_excel(url)
>>> orig_df = df
```

As colunas a seguir estão incluídas no conjunto de dados:

- pclass: classe do passageiro (1 = primeira, 2 = segunda, 3 = terceira)
- survived: sobreviveu (0 = não, 1 = sim)
- name: nome
- sex: sexo
- age: idade
- sibsp: número de irmãos/esposa(o) a bordo
- parch: número de pais/filhos a bordo
- ticket: número da passagem
- fare: preço da passagem
- cabin: cabine
- embarked: (local em que o passageiro embarcou C = Cherbourg, Q = Queenstown, S = Southampton)
- boat: bote salva-vidas
- body: número de identificação do corpo
- home.dest: lar/destino

O pandas é capaz de ler essa planilha e convertê-la em um DataFrame para nós. Precisaremos verificar pontualmente os dados e garantir que não apresentem problemas para as análises.

Limpe os dados

De posse dos dados, devemos garantir que estejam em um formato que possamos usar para criar um modelo. A maioria dos modelos do scikit-learn exige que nossos atributos sejam numéricos (inteiros ou números de ponto flutuante). Além disso, muitos modelos falham caso recebam valores ausentes (NaN no pandas ou no numpy). Alguns modelos terão melhor desempenho se os dados estiverem *padronizados* (têm um valor de média igual a 0 e um desvio-padrão igual a 1). Cuidaremos dessas questões usando o pandas ou o scikit-learn. Além disso, o conjunto de dados do Titanic contém atributos que provocam *vazamento* de informações (leaky features).

Atributos que causam vazamento de informações são variáveis que contêm dados sobre o futuro ou o objetivo. Não há nada ruim em ter dados sobre o objetivo e, com frequência, temos dados desse tipo no momento da criação do modelo. No entanto, se essas variáveis não estiverem disponíveis quando fizermos uma predição em uma nova amostra, devemos removê-las do modelo, pois haverá vazamento de dados futuros.

Uma limpeza nos dados pode ser um pouco demorada. Ter acesso a um SME (Subject Matter Expert, ou Especialista no Assunto), que dê orientações sobre como lidar com dados discrepantes (outliers) ou ausentes poderá ajudar.

```
>>> df.dtypes
pclass int64
survived int64
name object
sex object
age float64
sibsp int64
parch int64
ticket object
fare float64
cabin object
embarked object
boat object
body float64
home.dest object
dtype: object
```

Em geral, vemos os tipos `int64`, `float64`, `datetime64[ns]` ou `object`. São os tipos usados pelo pandas para armazenar uma coluna de dados. Os tipos `int64` e `float64` são numéricos. O tipo `datetime64[ns]` armazena dados de data e hora. O tipo `object` geralmente implica que o dado armazenado é uma string, embora possa ser uma combinação de string e outros tipos.

Quando lemos arquivos CSV, o pandas tentará fazer uma coerção dos dados para o tipo apropriado, mas recorrerá ao tipo `object` caso não seja possível. Ler dados de planilhas, bancos de dados ou de outros sistemas poderá resultar em tipos mais apropriados no DataFrame. Qualquer que seja o caso, vale a pena verificar os dados e garantir que os tipos façam sentido.

Tipos inteiros em geral não apresentam problemas. Tipos float (ponto flutuante) podem ter alguns valores ausentes. Tipos data e string precisarão ser convertidos ou usados para gerar tipos numéricos. Tipos string com baixa

cardinalidade são chamados de colunas de categorias, e talvez valha a pena criar colunas dummy a partir delas (a função pd.get_dummies faz isso).

NOTA

Até o pandas 0.23, se o tipo for int64, temos a garantia de que não há valores ausentes. Se o tipo for float64, os valores poderão ser todos números de ponto flutuante, mas poderão ser também números do tipo inteiro com valores ausentes. A biblioteca pandas converte valores inteiros que tenham números ausentes em números de ponto flutuante, pois esse tipo aceita valores ausentes. O tipo object em geral significa tipos string (ou tanto strings como numéricos).

A partir do pandas 0.24, há um novo tipo Int64 (observe a letra maiúscula no início). Esse não é o tipo inteiro default, mas você pode fazer uma coerção para esse tipo e, desse modo, ter suporte para números ausentes.

A biblioteca pandas-profiling inclui um relatório de perfil. É possível gerar esse relatório em um notebook. Ele sintetizará os tipos das colunas e permitirá que você visualize os detalhes das estatísticas dos quantis, as estatísticas descritivas, um histograma, os valores comuns e os valores extremos (veja as figuras 3.2 e 3.3).

```
>>> import pandas_profiling
>>> pandas_profiling.ProfileReport(df)
```

Utilize o atributo .shape do DataFrame para inspecionar o número de linhas e de colunas:

```
>>> df.shape
(1309, 14)
```

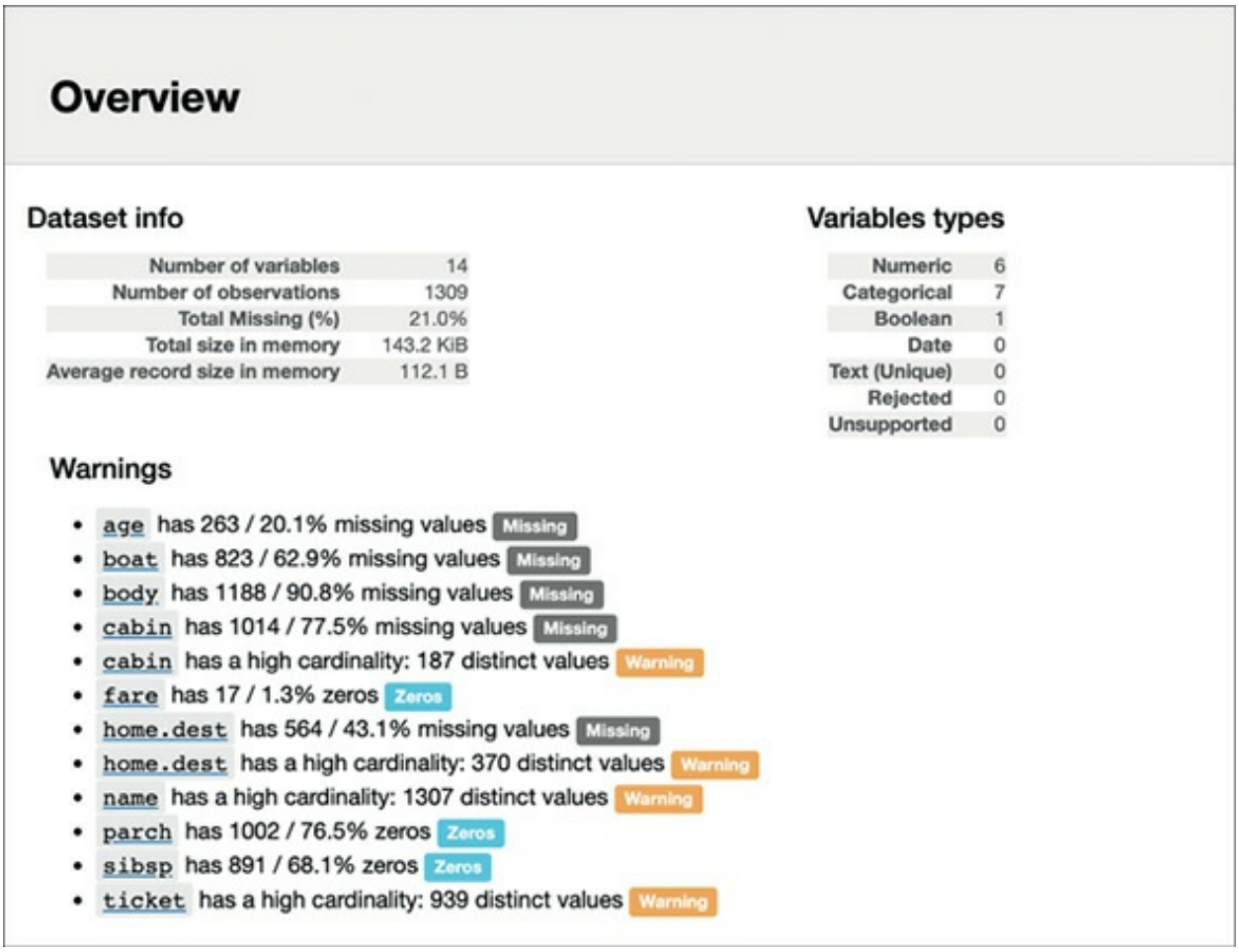


Figura 3.2 – Resumo gerado pelo pandas-profiling.

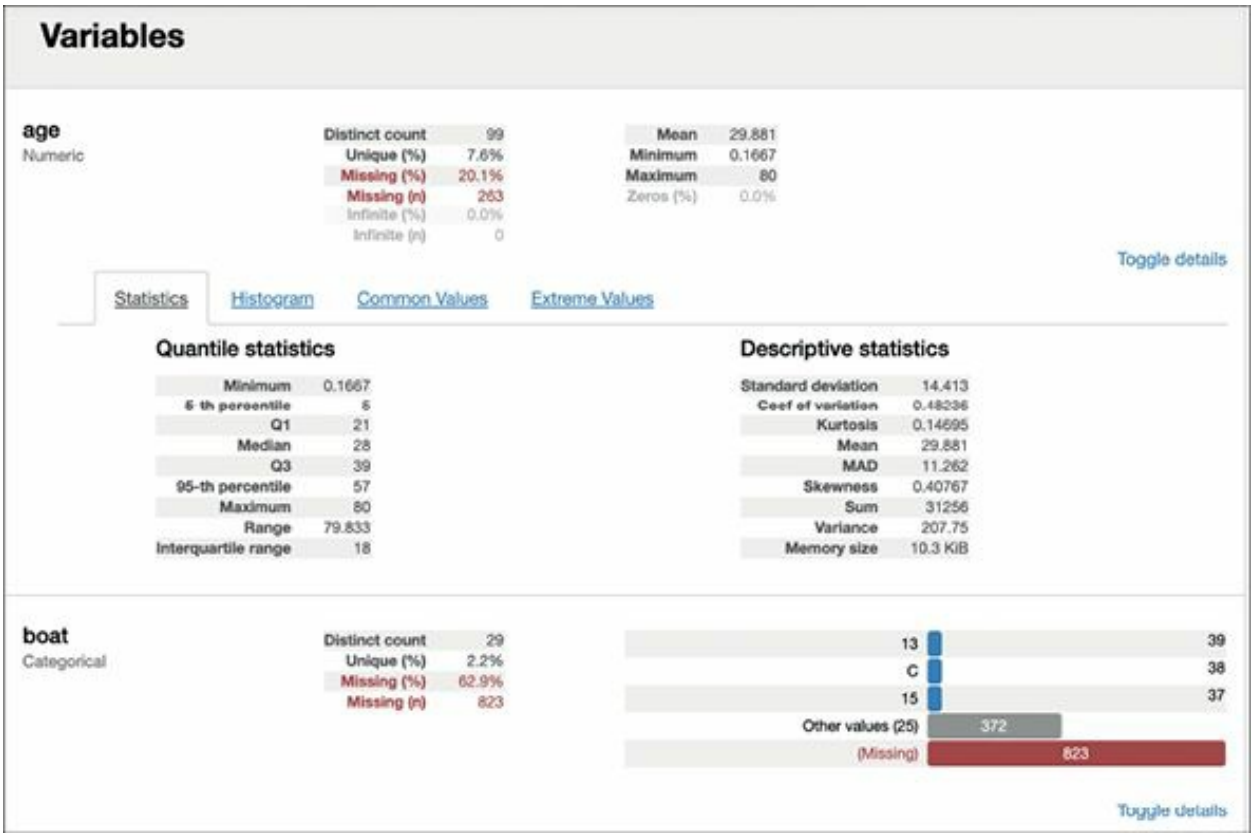


Figura 3.3 – Detalhes das variáveis exibidos pelo pandas-profiling.

Use o método .describe para obter as estatísticas resumidas, bem como para ver o contador de dados não nulos. O comportamento padrão desse método é informar dados apenas sobre as colunas numéricas. No exemplo a seguir, a saída foi truncada de modo a exibir somente as duas primeiras colunas:

```
>>> df.describe().iloc[:, :2]
            pclass survived
count 1309.000000 1309.000000
mean 2.294882 0.381971
std 0.837836 0.486055
min 1.000000 0.000000
25% 2.000000 0.000000
50% 3.000000 0.000000
75% 3.000000 1.000000
max 3.000000 1.000000
```

O dado estatístico count inclui apenas os valores que são diferentes de NaN, portanto é conveniente para verificar se uma coluna contém dados ausentes. Também é uma boa ideia conferir os valores mínimo e máximo para ver se há valores discrepantes. Verificar os dados estatísticos resumidos é uma forma de fazer isso. Gerar um histograma ou um gráfico de caixas (box plot) criará

representações visuais, conforme veremos posteriormente.

Lidar com dados ausentes é necessário. Utilize o método .isnull para identificar colunas ou linhas com valores ausentes. Chamar .isnull em um DataFrame devolve um novo DataFrame no qual cada célula conterá um valor True ou False. Em Python, esses valores são avaliados como 1 e 0, respectivamente. Isso nos permite somá-los ou até mesmo calcular a porcentagem de valores ausentes (calculando a média).

O resultado a seguir mostra o número de dados ausentes em cada coluna:

```
>>> df.isnull().sum()
pclass 0
survived 0
name 0
sex 0
age 263
sibsp 0
parch 0
ticket 0
fare 1
cabin 1014
embarked 2
boat 823
body 1188
home.dest 564
dtype: int64
```

DICA

Substitua .sum por .mean a fim de obter a porcentagem de valores nulos. Por padrão, chamar esses métodos fará com que a operação seja aplicada no eixo 0, que é o eixo do índice. Se quiser obter os contadores dos atributos ausentes em cada amostra, você poderá aplicar os métodos no eixo 1 (das colunas):

```
>>> df.isnull().sum(axis=1).loc[:10]
0 1
1 1
2 2
3 1
4 2
5 1
6 1
7 2
```

```
8 1
9 2
dtype: int64
```

Um especialista no assunto poderá ajudar você a determinar o que fazer com os dados ausentes. A coluna age (idade) poderia ser útil, portanto, mantê-la e interpolar valores poderia fornecer alguma informação para o modelo. As colunas em que a maioria dos valores está ausente (cabin, boat e body) tendem a não ser relevantes e poderão ser descartadas.

A coluna body (número de identificação do corpo) está ausente em muitas linhas. Devemos descartar essa coluna, de qualquer modo, pois ela causa vazamento de informação. Essa coluna informa que o passageiro não sobreviveu; por necessidade, nosso modelo poderia usar esse dado para trapacear. Então, vamos eliminá-la. (Se estamos criando um modelo para prever se um passageiro morreu, saber que ele teve um número de identificação do corpo nos levaria a saber, *a priori*, que ele morreu. Não queremos que nosso modelo saiba dessa informação, mas faça a predição com base nas demais colunas.) De modo semelhante, a coluna boat (bote salva-vidas) causa o vazamento de uma informação inversa (a informação de que o passageiro sobreviveu).

Vamos analisar algumas das linhas com dados ausentes. Podemos criar um array de booleanos (uma série com `True` ou `False` para informar se a linha contém dados ausentes) e usá-lo para inspecionar as linhas com dados ausentes:

```
>>> mask = df.isnull().any(axis=1)

>>> mask.head() # linhas
0 True
1 True
2 True
3 True
4 True
dtype: bool

>>> df[mask].body.head()
0 NaN
1 NaN
2 NaN
3 135.0
4 NaN
```

```
Name: body, dtype: float64
```

Imputaremos dados aos valores ausentes (ou derivaremos valores para eles) para a coluna de idade mais tarde.

As colunas com tipo `object` tendem a conter valores de categorias (mas podem também ser dados do tipo string com alta cardinalidade, ou uma mistura de tipos das colunas). Nas colunas do tipo `object` que acreditamos ser colunas de categorias, utilize o método `.value_counts` para analisar os contadores dos valores:

```
>>> df.sex.value_counts(dropna=False)
male 843
female 466
Name: sex, dtype: int64
```

Lembre-se de que o pandas em geral ignora valores nulos ou NaN. Se quiser incluí-los, utilize `dropna=False` para exibir também os contadores para NaN:

```
>>> df.embarked.value_counts(dropna=False)
S 914
C 270
Q 123
NaN 2
Name: embarked, dtype: int64
```

Há algumas opções para lidar com valores ausentes no campo embarked (local em que o passageiro embarcou). Usar S pode parecer lógico, pois este é o valor mais comum. Poderíamos explorar os dados e tentar determinar se outra opção seria melhor. Também poderíamos descartar esses dois valores. Ou, por ser uma categoria, poderíamos ignorá-los e usar o pandas para criar colunas dummy, e essas duas amostras simplesmente teriam entradas 0 para qualquer opção. Usaremos essa última opção no caso desse atributo.

Crie os atributos

Podemos descartar as colunas que não tenham nenhuma variação ou sinal. Não há atributos como esses no conjunto de dados do Titanic, mas, se houvesse uma coluna chamada “é um ser humano” que contivesse o valor 1 para todas as amostras, essa coluna não estaria fornecendo nenhuma informação.

Além disso, a menos que estivéssemos usando NLP (Natural Language Processing, ou Processamento de Língua Natural) ou extraindo dados de

colunas de texto, nas quais cada valor fosse distinto, um modelo não seria capaz de tirar proveito dessa coluna. A coluna de nomes (name) é um exemplo desse caso. Alguns extraíram o pronome de tratamento t (título) do nome e o trataram como uma coluna de categorias.

Também queremos descartar as colunas que causem vazamento de informações. Tanto as colunas boat como body causam vazamento da informação acerca de o passageiro ter sobrevivido ou não.

O método .drop do pandas pode descartar linhas ou colunas:

```
>>> name = df.name
>>> name.head(3)
0 Allen, Miss. Elisabeth Walton
1 Allison, Master. Hudson Trevor
2 Allison, Miss. Helen Loraine
Name: name, dtype: object

>>> df = df.drop(
... columns=[
... "name",
... "ticket",
... "home.dest",
... "boat",
... "body",
... "cabin",
... ]
... )
```

Temos de criar colunas dummy a partir das colunas de string. Com isso, novas colunas serão criadas para sex (sexo) e embarked (local em que o passageiro embarcou). O pandas tem uma função get_dummies conveniente para isso:

```
>>> df = pd.get_dummies(df)

>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
   'parch', 'fare', 'sex_female', 'sex_male',
   'embarked_C', 'embarked_Q', 'embarked_S'],
   dtype='object')
```

Nesse momento, as colunas sex_male e sex_female estão inversamente correlacionadas de forma perfeita. Em geral, removemos qualquer coluna com uma correlação perfeita ou com uma correção positiva ou negativa bem alta. A multicolinearidade pode causar impactos na interpretação da

importância dos atributos e dos coeficientes em alguns modelos. Eis o código para remover a coluna sex_male:

```
>>> df = df.drop(columns="sex_male")
```

Como alternativa, podemos adicionar um parâmetro drop_first=True à chamada de get_dummies:

```
>>> df = pd.get_dummies(df, drop_first=True)


>>> df.columns
Index(['pclass', 'survived', 'age', 'sibsp',
   'parch', 'fare', 'sex_male',
   'embarked_Q', 'embarked_S'],
   dtype='object')
```

Crie um DataFrame (X) com os atributos e uma série (y) com os rótulos. Poderíamos também usar arrays numpy, mas, nesse caso, não teríamos nomes para as colunas:

```
>>> y = df.survived
>>> X = df.drop(columns="survived")
```

DICA

Podemos usar a biblioteca pyjanitor (*https://oreil.ly/_IWbA*) para substituir as duas linhas anteriores:

```
>>> import janitor as jn
>>> X, y = jn.get_features_targets(
... df, target_columns="survived"
... )
```

Separe as amostras

Sempre devemos fazer treinamento e testes em dados distintos. Caso contrário, você não saberá realmente quão bem seu modelo poderá ser generalizado para dados que ainda não tenham sido vistos antes. Usaremos o scikit-learn para extrair 30% dos dados para testes (usamos random_state=42 para eliminar a aleatoriedade caso venhamos a comparar diferentes modelos):

```
>>> X_train, X_test, y_train, y_test = model_selection.train_test_split(
... X, y, test_size=0.3, random_state=42
... )
```

Faça a imputação de dados

A coluna de idade tem valores ausentes. Devemos imputar uma idade a partir dos valores numéricos. Queremos imputar dados apenas no conjunto de treinamento, e então usar esse imputer para preencher os dados no conjunto de testes. Caso contrário, causaremos vazamento de informações (trapaceando ao dar informações futuras para o modelo).

Agora que temos dados para testes e para treinamento, podemos imputar dados aos valores ausentes no conjunto de treinamento e usar os imputers treinados para preencher o conjunto de dados de teste. A biblioteca fancyimpute (*https://oreil.ly/Vlf9e*) tem vários algoritmos implementados. Infelizmente, a maioria desses algoritmos não está implementada de modo *indutivo*. Isso significa que você não poderá chamar `.fit` e, em seguida, `.transform`, ou seja, você não poderá imputar informação em novos dados com base no modo como o modelo foi treinado.

A classe `IterativeImputer` (que estava em fancyimpute, mas foi passada para o scikit-learn) não aceita o modo indutivo. Para usá-la, é necessário acrescentar uma importação experimental especial (na versão 0.21.2 do scikit-learn):

```
>>> from sklearn.experimental import (
... enable_iterative_imputer,
... )
>>> from sklearn import impute
>>> num_cols = [
... "pclass",
... "age",
... "sibsp",
... "parch",
... "fare",
... "sex_female",
... ]

>>> imputer = impute.IterativeImputer()
>>> imputed = imputer.fit_transform(
... X_train[num_cols]
... )
>>> X_train.loc[:, num_cols] = imputed
>>> imputed = imputer.transform(X_test[num_cols])
>>> X_test.loc[:, num_cols] = imputed
```

Se quisermos imputar valores usando a mediana, podemos utilizar o pandas para isso:

```
>>> meds = X_train.median()
```

```
>>> X_train = X_train.fillna(meds)
>>> X_test = X_test.fillna(meds)
```

Normalize os dados

Normalizar ou pré-processar os dados ajudará muitos modelos a terem um melhor desempenho depois, particularmente, para aqueles que dependam de uma métrica de distância para determinar a semelhança. (Observe que modelos em árvore, que tratam cada atributo por si só, não têm esse requisito.)

Vamos padronizar os dados para o pré-processamento. Padronizar significa traduzir os dados de modo que tenham um valor de média igual a zero e um desvio-padrão igual a um. Desse modo, os modelos não tratarão as variáveis com escalas maiores como mais importantes que as variáveis com menor escala. Colocarei o resultado (array numpy) de volta em um DataFrame do pandas para facilitar a manipulação (e manter os nomes das colunas).

Em geral, também não padronizo colunas dummy, portanto elas serão ignoradas:

```
>>> cols = "pclass,age,sibsp,fare".split(",")
>>> sca = preprocessing.StandardScaler()
>>> X_train = sca.fit_transform(X_train)
>>> X_train = pd.DataFrame(X_train, columns=cols)
>>> X_test = sca.transform(X_test)
>>> X_test = pd.DataFrame(X_test, columns=cols)
```

Refatore

A essa altura, gosto de refatorar o meu código. Geralmente, crio duas funções. Uma para uma limpeza geral e outra para fazer uma divisão entre um conjunto de treinamento e um conjunto de testes e fazer as alterações que devam ocorrer de forma diferente nesses conjuntos:

```
>>> def tweak_titanic(df):
... df = df.drop(
... columns=[
... "name",
... "ticket",
... "home.dest",
... "boat",
... "body",
... "cabin",
```

```
... ]
... ).pipe(pd.get_dummies, drop_first=True)
... return df

>>> def get_train_test_X_y(
... df, y_col, size=0.3, std_cols=None
... ):
... y = df[y_col]
... X = df.drop(columns=y_col)
... X_train, X_test, y_train, y_test = model_selection.train_test_split(
... X, y, test_size=size, random_state=42
... )
... cols = X.columns
... num_cols = [
... "pclass",
... "age",
... "sibsp",
... "parch",
... "fare",
... ]
... fi = impute.IterativeImputer()
... X_train.loc[
... :, num_cols
... ] = fi.fit_transform(X_train[num_cols])
... X_test.loc[:, num_cols] = fi.transform(
... X_test[num_cols]
... )
...
... if std_cols:
... std = preprocessing.StandardScaler()
... X_train.loc[
... :, std_cols
... ] = std.fit_transform(
... X_train[std_cols]
... )
... X_test.loc[
... :, std_cols
... ] = std.transform(X_test[std_cols])
...
... return X_train, X_test, y_train, y_test

>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test = get_train_test_X_y(
... ti_df, "survived", std_cols=std_cols
```

```
... )
```

Modelo de base

Criar um modelo que faça algo realmente simples pode nos dar uma base com a qual poderemos comparar o nosso modelo. Observe que usar o resultado `.score` default nos dará uma precisão que pode ser enganosa. Um problema em que um caso positivo seja de 1 em 10 mil poderia facilmente ter uma precisão acima de 99% ao fazer sempre uma predição negativa.

```
>>> from sklearn.dummy import DummyClassifier
>>> bm = DummyClassifier()
>>> bm.fit(X_train, y_train)
>>> bm.score(X_test, y_test) # precisão
0.5292620865139949

>>> from sklearn import metrics
>>> metrics.precision_score(
... y_test, bm.predict(X_test)
... )
0.4027777777777778
```

Várias famílias

O código a seguir testa diversas famílias de algoritmos. O teorema “No Free Lunch” afirma que nenhum algoritmo tem bom desempenho em todos os dados. No entanto, para um número finito de dados, pode haver um algoritmo que tenha um bom desempenho nesse conjunto. (Uma opção popular para machine learning com dados estruturados atualmente é um algoritmo baseado em árvore, como o XGBoost.)

Em nosso caso, usaremos algumas famílias diferentes e compararemos a pontuação AUC e o desvio-padrão usando uma validação cruzada k-fold. Um algoritmo que tenha uma pontuação média um pouco menor, porém um desvio-padrão menor, poderia ser uma melhor opção.

Como estamos usando uma validação cruzada k-fold, alimentaremos todos os modelos com `X` e `y`:

```
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
>>> from sklearn import model_selection
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.linear_model import (
```

```
... LogisticRegression,
... )
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import (
... KNeighborsClassifier,
... )
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import (
... RandomForestClassifier,
... )
>>> import xgboost

>>> for model in [
... DummyClassifier,
... LogisticRegression,
... DecisionTreeClassifier,
... KNeighborsClassifier,
... GaussianNB,
... SVC,
... RandomForestClassifier,
... xgboost.XGBClassifier,
... ]:
... cls = model()
... kfold = model_selection.KFold(
... n_splits=10, random_state=42
... )
... s = model_selection.cross_val_score(
... cls, X, y, scoring="roc_auc", cv=kfold
... )
... print(
... f"{model.__name__:22} AUC: "
... f"{s.mean():.3f} STD: {s.std():.2f}"
... )
DummyClassifier AUC: 0.511 STD: 0.04
LogisticRegression AUC: 0.843 STD: 0.03
DecisionTreeClassifier AUC: 0.761 STD: 0.03
KNeighborsClassifier AUC: 0.829 STD: 0.05
GaussianNB AUC: 0.818 STD: 0.04
SVC AUC: 0.838 STD: 0.05
RandomForestClassifier AUC: 0.829 STD: 0.04
XGBClassifier AUC: 0.864 STD: 0.04
```

Stacking

Se você vai seguir o caminho do Kaggle (ou quer o máximo de desempenho à custa da facilidade de interpretação), o *stacking* (empilhamento) é uma opção. Um classificador stacking utiliza a saída de outros modelos para fazer a predição de um objetivo ou de um rótulo. Usaremos as saídas dos modelos anteriores e as combinaremos para ver se um classificador stacking poderia ser melhor:

```
>>> from mlxtend.classifier import (
... StackingClassifier,
... )
>>> clfs = [
... x()
... for x in [
... LogisticRegression,
... DecisionTreeClassifier,
... KNeighborsClassifier,
... GaussianNB,
... SVC,
... RandomForestClassifier,
... ]
... ]
>>> stack = StackingClassifier(
... classifiers=clfs,
... meta_classifier=LogisticRegression(),
... )
>>> kfold = model_selection.KFold(
... n_splits=10, random_state=42
... )
>>> s = model_selection.cross_val_score(
... stack, X, y, scoring="roc_auc", cv=kfold
... )
>>> print(
... f"{stack.__class__.__name__} "
... f"AUC: {s.mean():.3f} STD: {s.std():.2f}"
... )
StackingClassifier AUC: 0.804 STD: 0.06
```

Nesse caso, parece que o desempenho piorou um pouco, assim como o desvio-padrão.

Crie o modelo

Usarei um classificador de floresta aleatória (random forest) para criar um modelo. É um modelo flexível, que tende a dar resultados razoáveis

prontamente. Lembre-se de treiná-lo (chamando .fit) com os dados de treinamento que separamos antes, quando criamos os conjuntos de treinamento e de testes:

```
>>> rf = ensemble.RandomForestClassifier(
... n_estimators=100, random_state=42
... )
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
  class_weight=None, criterion='gini',
  max_depth=None, max_features='auto',
  max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,
  min_samples_leaf=1, min_samples_split=2,
  min_weight_fraction_leaf=0.0, n_estimators=10,
  n_jobs=1, oob_score=False, random_state=42,
  verbose=0, warm_start=False)
```

Avalie o modelo

Agora que temos um modelo, podemos usar os dados de teste para verificar se o modelo pode ser generalizado para dados nunca vistos antes. O método .score de um classificador devolve a média da precisão da predição. Devemos garantir que chamaremos o método .score com os dados de teste (supostamente, o desempenho deverá ser melhor com os dados de treinamento):

```
>>> rf.score(X_test, y_test)
0.7964376590330788
```

Também podemos observar outras métricas, por exemplo, a precisão:

```
>>> metrics.precision_score(
... y_test, rf.predict(X_test)
... )
0.8013698630136986
```

Uma boa vantagem dos modelos baseados em árvore é que você pode inspecionar a importância dos atributos. A importância dos atributos nos diz quanto um atributo contribui para o modelo. Observe que remover um atributo não significa que a pontuação cairá na mesma medida, pois outros atributos poderão ser colineares (em nosso caso, poderíamos remover a coluna sex_male ou sex_female, pois elas têm uma correlação negativa

perfeita):

```
>>> for col, val in sorted(
... zip(
... X_train.columns,
... rf.feature_importances_,
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
age 0.277
fare 0.265
sex_female 0.240
pclass 0.092
sibsp 0.048
```

A importância dos atributos é calculada observando o aumento no erro. Se a remoção de um atributo causar um aumento no erro do modelo, é sinal de que o atributo é importante.

Gosto muito da biblioteca SHAP para explorar quais são os atributos que um modelo considera importantes, e para explicar as predições. Essa biblioteca funciona com modelos caixa-preta, e ela será apresentada mais adiante.

Otimize o modelo

Os modelos têm *hiperparâmetros* que controlam o seu comportamento. Ao variar os valores desses parâmetros, alteramos o desempenho dos modelos. O sklearn tem uma classe de busca em grade (grid search) para avaliar um modelo com diferentes combinações de parâmetros, devolvendo o melhor resultado. Podemos usar esses parâmetros para instanciar a classe do modelo:

```
>>> rf4 = ensemble.RandomForestClassifier()
>>> params = {
... "max_features": [0.4, "auto"],
... "n_estimators": [15, 200],
... "min_samples_leaf": [1, 0.1],
... "random_state": [42],
... }
>>> cv = model_selection.GridSearchCV(
... rf4, params, n_jobs=-1
... ).fit(X_train, y_train)
>>> print(cv.best_params_)
{'max_features': 'auto', 'min_samples_leaf': 0.1,
```

```
 'n_estimators': 200, 'random_state': 42}

>>> rf5 = ensemble.RandomForestClassifier(
... **{
... "max_features": "auto",
... "min_samples_leaf": 0.1,
... "n_estimators": 200,
... "random_state": 42,
... }
... )
>>> rf5.fit(X_train, y_train)
>>> rf5.score(X_test, y_test)
0.7888040712468194
```

Um parâmetro `scoring` pode ser passado para `GridSearchCV` a fim de fazer otimizações de acordo com diferentes métricas. Consulte o Capítulo 12, que apresenta uma lista de métricas e seus significados.

Matriz de confusão

Uma matriz de confusão (confusion matrix) nos permite ver as classificações corretas, bem como os falso-positivos e os falso-negativos. Podemos otimizar em relação aos falso-positivos ou aos falso-negativos, e diferentes modelos ou parâmetros são capazes de alterar isso. O sklearn pode ser usado para uma versão textual, ou o Yellowbrick para gerar um gráfico (veja a Figura 3.4):

```
>>> from sklearn.metrics import confusion_matrix
>>> y_pred = rf5.predict(X_test)
>>> confusion_matrix(y_test, y_pred)
array([[196, 28],
       [ 55, 114]])

>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
... rf5,
... classes=["died", "survived"],
... label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig(
... "images/mlpr_0304.png",
... dpi=300,
... bbox_inches="tight",
```

...)

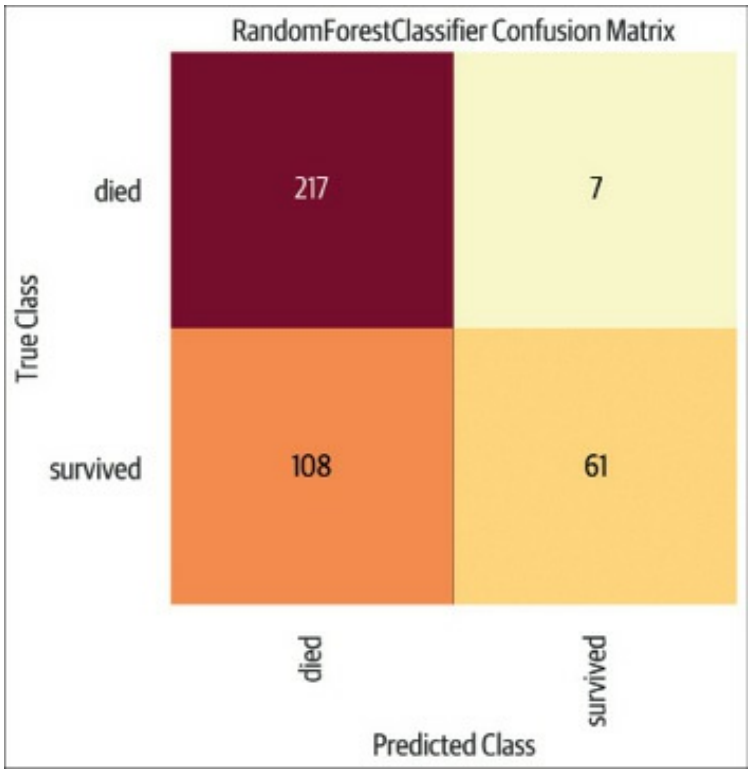


Figura 3.4 – Matriz de confusão do Yellowbrick. É uma ferramenta de avaliação útil, que apresenta a classe de predições na parte inferior e a classe real na lateral. Um bom classificador teria todos os valores na diagonal, e zeros nas demais células.

Curva ROC

Um gráfico ROC (Receiver Operating Characteristic) é uma ferramenta comum, usada para avaliar classificadores. Ao calcular a área sob a curva (AUC), podemos obter uma métrica para comparar diferentes classificadores (veja a Figura 3.5).

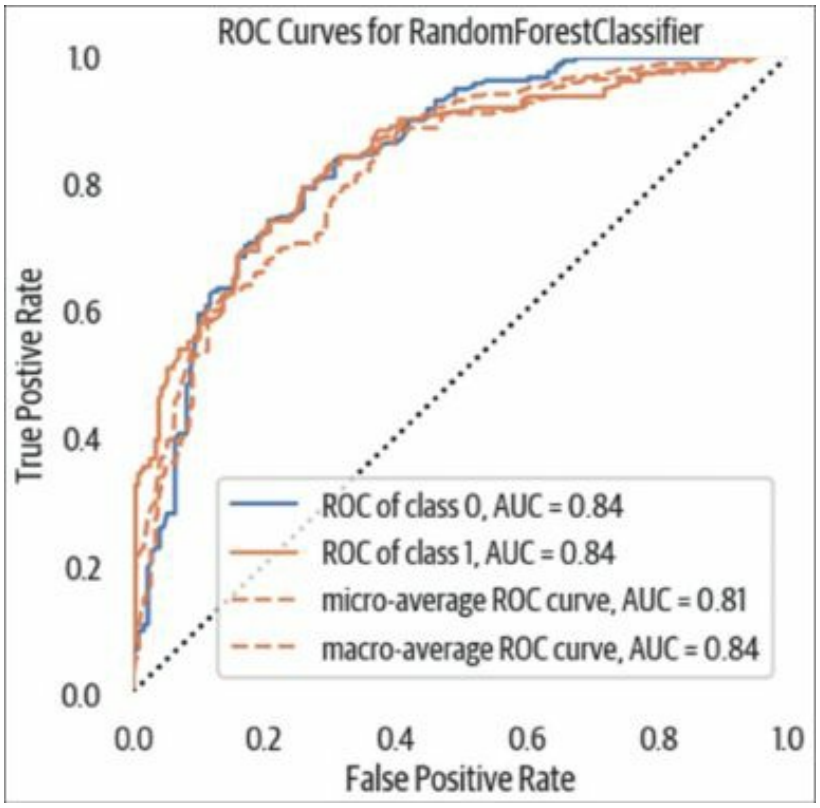


Figura 3.5 – Curva ROC. Mostra a taxa dos realmente positivos em relação à taxa dos falso-positivos. Em geral, quanto mais saliente, melhor. Calcular o AUC fornece um único número a ser avaliado. Um valor mais próximo de um é melhor. Abaixo de 0,5 será um modelo ruim.

O gráfico mostra a taxa dos realmente positivos em relação à taxa dos falso-positivos. O sklearn pode ser usado para calcular o AUC:

```
>>> y_pred = rf5.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

Ou podemos utilizar o Yellowbrick para visualizar o gráfico:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(rf5)
>>> roc_viz.score(X_test, y_test)
0.8279691030696217
>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_0305.png")
```

Curva de aprendizado

Uma curva de aprendizado (learning curve) é usada para nos dizer se temos dados de treinamento suficientes. O modelo é treinado com porções cada vez

maiores dos dados e a pontuação é calculada (veja a Figura 3.6).

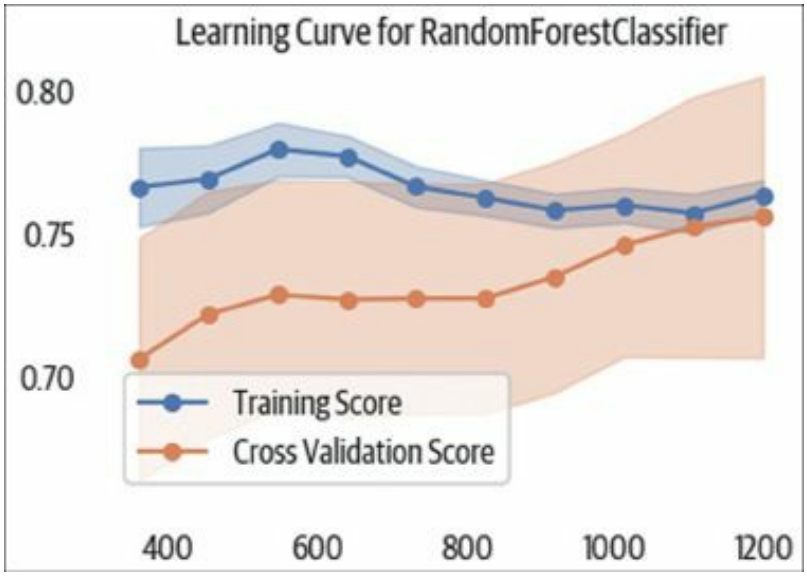


Figura 3.6 – Essa curva de aprendizado mostra que, à medida que adicionamos mais amostras para treinamento, nossas pontuações de validação cruzada (testes) parecem melhorar.

Se a pontuação da validação cruzada continuar subindo, talvez seja necessário investir em coletar mais dados. Eis um exemplo com o Yellowbrick:

```
>>> import numpy as np
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> cv = StratifiedKFold(12)
>>> sizes = np.linspace(0.3, 1.0, 10)
>>> lc_viz = LearningCurve(
... rf5,
... cv=cv,
... train_sizes=sizes,
... scoring="f1_weighted",
... n_jobs=4,
... ax=ax,
... )
>>> lc_viz.fit(X, y)
>>> lc_viz.poof()
>>> fig.savefig("images/mlpr_0306.png")
```

Implante o modelo

Ao usar o módulo pickle de Python, podemos fazer a persistência dos modelos

e carregá-los. Depois que tivermos um modelo, chamamos o método .predict para obter uma classificação ou um resultado de regressão:

```
>>> import pickle
>>> pic = pickle.dumps(rf5)
>>> rf6 = pickle.loads(pic)
>>> y_pred = rf6.predict(X_test)
>>> roc_auc_score(y_test, y_pred)
0.7747781065088757
```

Usar o Flask (*https://palletsprojects.com/p/flask*) para a implantação de um web service para predições é muito comum. Atualmente, outros produtos open source e comerciais estão surgindo, os quais oferecem suporte para a implantação. Entre eles, temos o Clipper (*http://clipper.ai/*), o Pipeline (*https://oreil.ly/UfHdP*) e o Cloud Machine Learning Engine do Google (*https://oreil.ly/1qYkH*).

---

1 Embora essa biblioteca não seja diretamente chamada, quando um arquivo Excel é carregado, o pandas a utiliza internamente.

CAPÍTULO 4

Dados ausentes

É necessário lidar com dados ausentes. O capítulo anterior mostrou um exemplo. Neste capítulo, exploraremos um pouco mais o assunto. A maioria dos algoritmos não funcionará se houver dados ausentes. Exceções dignas de nota são as recentes bibliotecas inovadoras XGBoost, CatBoost e LightGBM.

Assim como ocorre em relação a várias questões no machine learning, não há respostas únicas sobre como tratar dados ausentes. Além do mais, a ausência de dados poderia representar diferentes situações. Suponha que tenhamos recebido dados de censo e que um atributo idade tenha sido informado como ausente. Isso é porque a amostra não quis revelar a sua idade? Não sabia a idade? Quem fez as perguntas se esqueceu de pedi-la? Há algum padrão para as idades ausentes? Ela está correlacionada com outro atributo? Ou é totalmente aleatória?

Há também diversas maneiras de lidar com dados ausentes:

- remover qualquer linha com dados ausentes;
- remover qualquer coluna com dados ausentes;
- imputar dados aos valores ausentes;
- criar uma coluna para informar que os dados estavam ausentes.

Analisando dados ausentes

Vamos voltar aos dados do Titanic. Pelo fato de Python tratar `True` e `False` como `1` e `0` respectivamente, podemos usar esse truque no pandas para obter o percentual dos dados ausentes:

```
>>> df.isnull().mean() * 100
pclass 0.000000
survived 0.000000
name 0.000000
sex 0.000000
age 20.091673
```

```
sibsp 0.000000
parch 0.000000
ticket 0.000000
fare 0.076394
cabin 77.463713
embarked 0.152788
boat 62.872422
body 90.756303
home.dest 43.086325
dtype: float64
```

Para visualizar padrões nos dados ausentes, utilize a biblioteca missingno (*https://oreil.ly/rgYJG*), a qual é conveniente para visualizar áreas contíguas de dados ausentes, que sinalizariam que os dados ausentes não são aleatórios (veja a Figura 4.1).

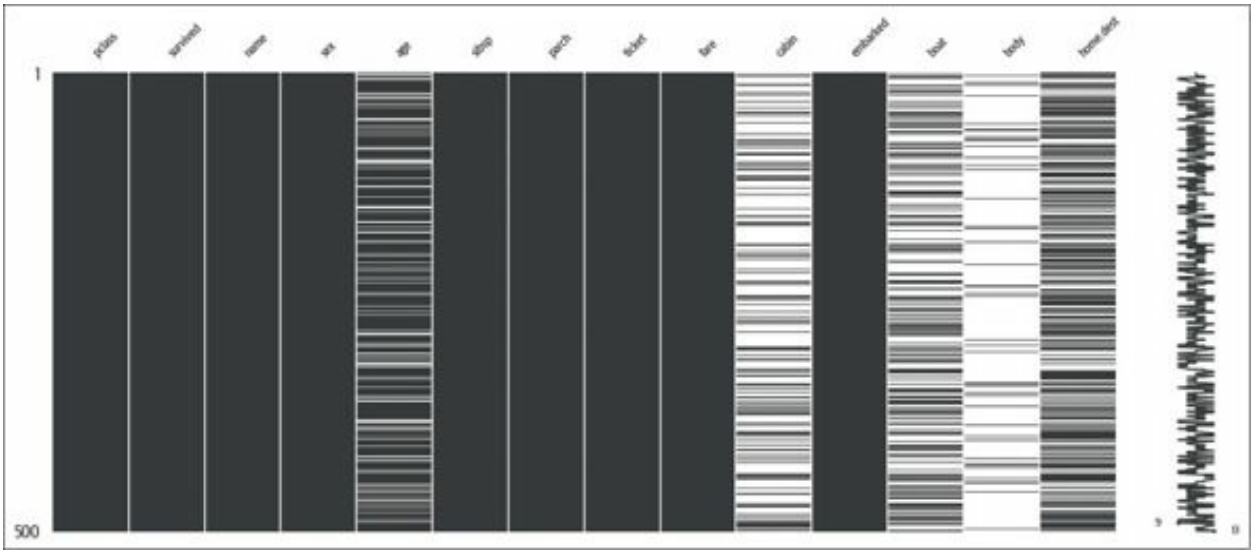


Figura 4.1 – Locais em que há dados ausentes. Nenhum padrão se destaca claramente para o autor.

A função matrix inclui uma área em destaque do lado direito. Padrões nesse local também indicariam que os dados ausentes não são aleatórios. Talvez seja necessário limitar o número de amostras para que seja possível ver os padrões:

```
>>> import missingno as msno
>>> ax = msno.matrix(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0401.png")
```

Podemos criar um gráfico de barras com os contadores de dados ausentes usando o pandas (veja a Figura 4.2).

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (1 - df.isnull().mean()).abs().plot.bar(ax=ax)
>>> fig.savefig("images/mlpr_0402.png", dpi=300)
```

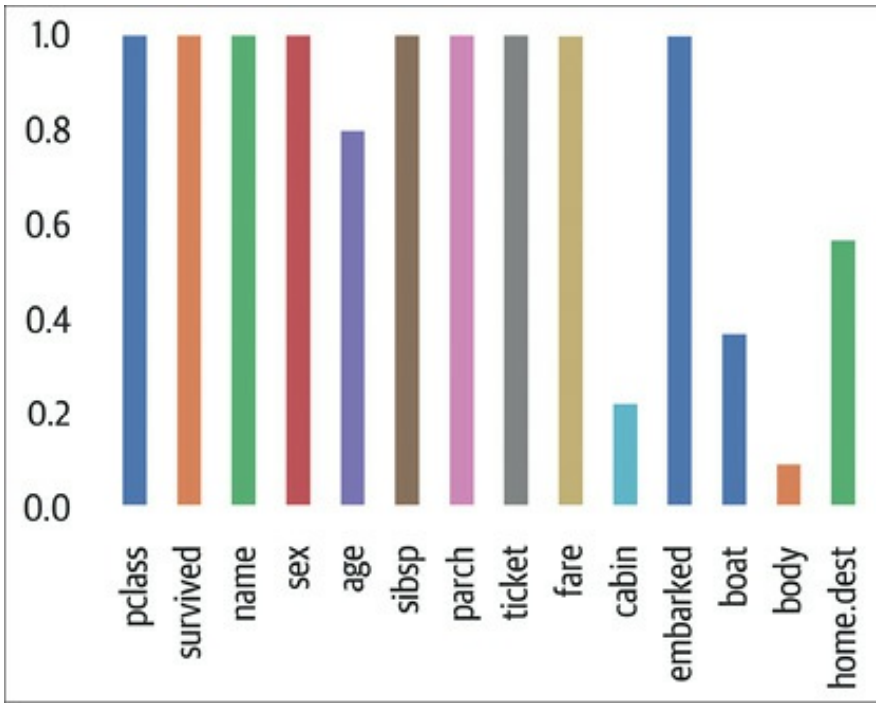


Figura 4.2 – Percentual de dados não ausentes com o pandas. Os dados de boat (barco salva-vidas) e body (número de identificação do corpo) causam vazamento de informações, portanto devemos ignorá-los. É interessante o fato de haver idades ausentes.

Podemos também usar a biblioteca missingno para gerar o mesmo gráfico (veja a Figura 4.3).

```
>>> ax = msno.bar(orig_df.sample(500))
>>> ax.get_figure().savefig("images/mlpr_0403.png")
```

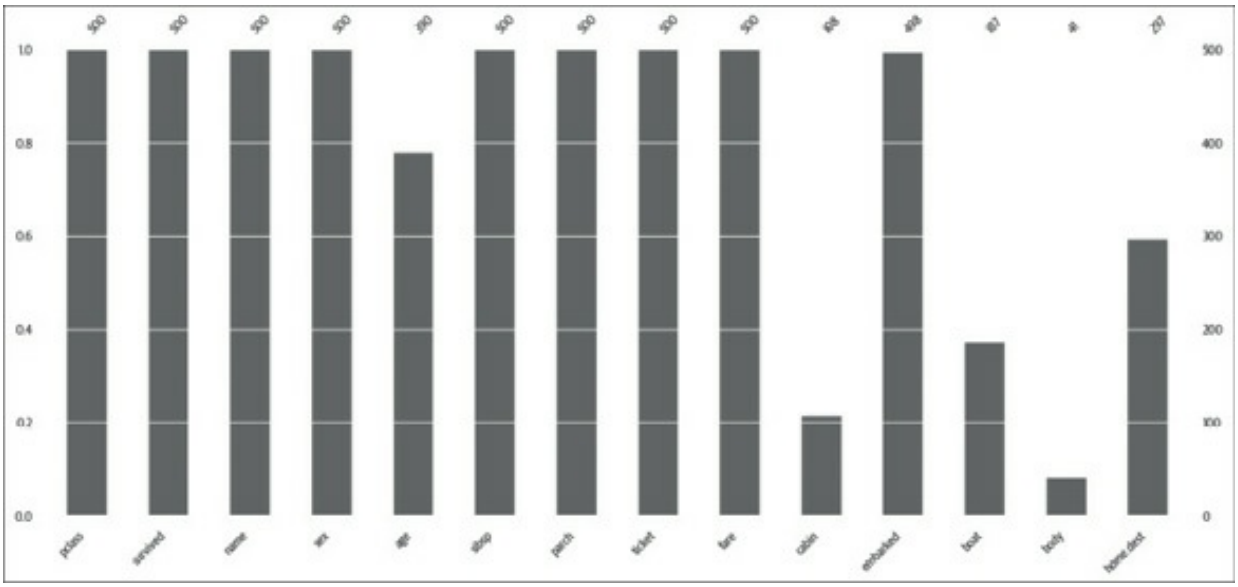


Figura 4.3 – Percentuais de dados não ausentes gerados com o missingno.

Um heat map (mapa de calor) pode ser criado, mostrando se há correlações no caso de haver dados ausentes (veja a Figura 4.4). Em nosso caso, não parece haver correlações entre os atributos que apresentam dados ausentes:

```
>>> ax = msno.heatmap(df, figsize=(6, 6))
>>> ax.get_figure().savefig("/tmp/mlpr_0404.png")
```

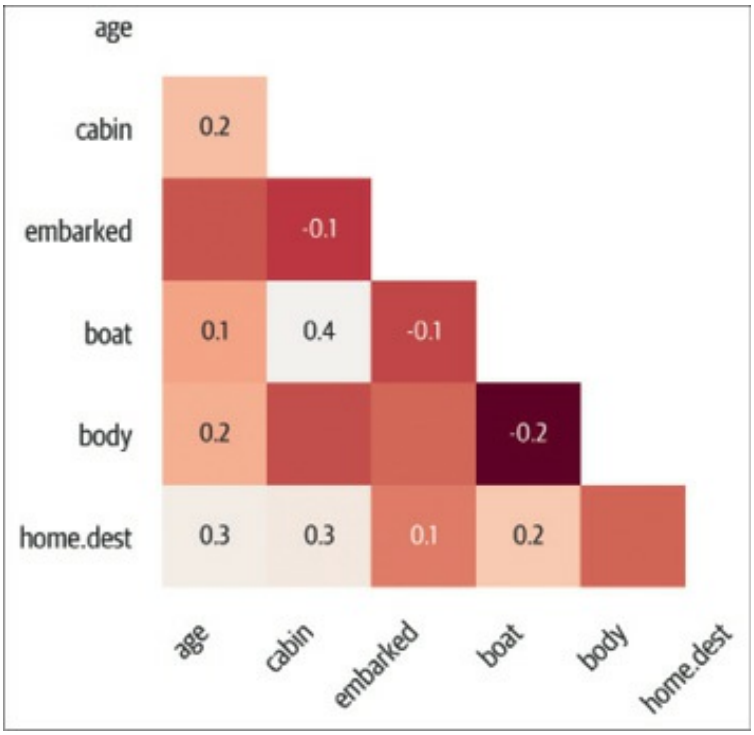


Figura 4.4 – Correlações entre dados ausentes geradas com o missingno.

Podemos criar um dendrograma que mostra os clusterings (agrupamentos) de dados ausentes (veja a Figura 4.5). As folhas que estão no mesmo nível fazem a predição da presença umas das outras (cheias ou preenchidas). Os braços verticais são usados para mostrar a diferença entre os clusters. Braços curtos indicam que os ramos são semelhantes:

```
>>> ax = msno.dendrogram(df)
>>> ax.get_figure().savefig("images/mlpr_0405.png")
```

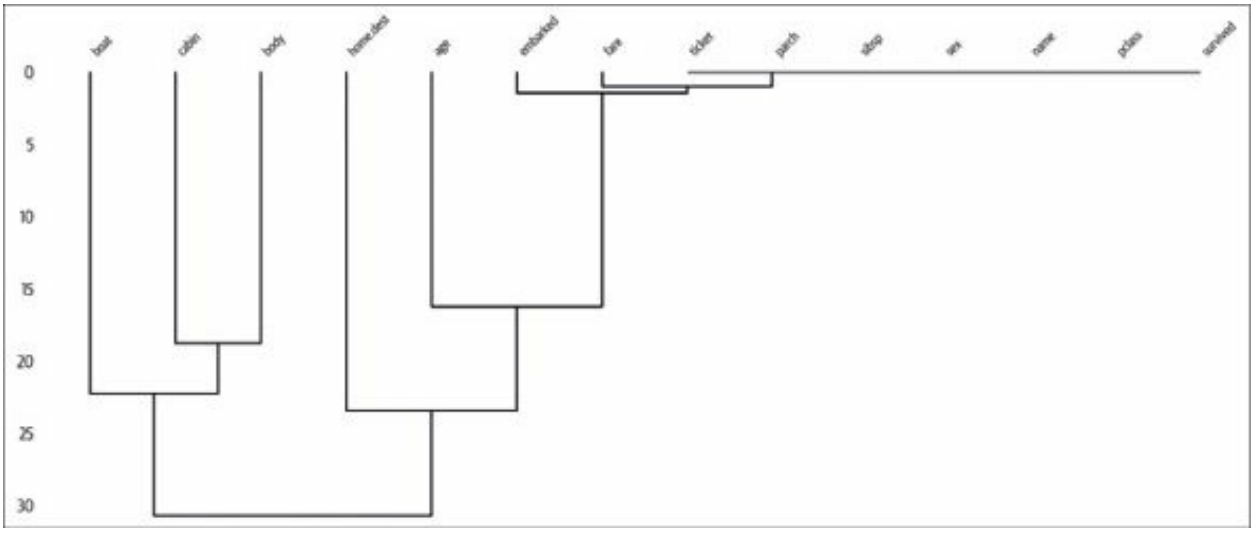


Figura 4.5 – Dendrograma de dados ausentes gerado com o missingno. Podemos ver as colunas que não têm dados ausentes na parte superior à direita.

Descartando dados ausentes

A biblioteca pandas é capaz de descartar todas as linhas contendo dados ausentes usando o método .dropna:

```
>>> df1 = df.dropna()
```

Para descartar colunas, podemos observar quais colunas contêm dados ausentes e utilizar o método .drop. É possível passar uma lista de nomes de colunas ou um único nome de coluna:

```
>>> df1 = df.drop(columns="cabin")
```

Como alternativa, podemos usar o método .dropna e definir axis=1 (descartar no eixo das colunas):

```
>>> df1 = df.dropna(axis=1)
```

Tome cuidado ao descartar dados. Em geral, encaro essa opção como um

último recurso.

Imputando dados

Depois que tivermos uma ferramenta para predição de dados, podemos usá-la para prever valores no caso de dados ausentes. A tarefa geral para definir valores a dados ausentes chama-se *imputação* (imputation).

Se você estiver imputando dados, será necessário construir um pipeline e usar a mesma lógica de imputação na criação do modelo e no momento da predição. A classe `SimpleImputer` do scikit-learn trabalhará com a média, a mediana e com os valores mais frequentes dos atributos.

O comportamento default é calcular a média:

```
>>> from sklearn.impute import SimpleImputer
>>> num_cols = df.select_dtypes(
... include="number"
... ).columns
>>> im = SimpleImputer() # média
>>> imputed = im.fit_transform(df[num_cols])
```

Especifique `strategy='median'` ou `strategy='most_frequent'` para substituir o valor usado na substituição pela mediana ou pelo valor mais comum, respectivamente. Se quiser preencher o dado com um valor constante, por exemplo, `-1`, use `strategy='constant'` junto com `fill_value=-1`.

DICA

Você pode usar o método `.fillna` do pandas para imputar dados aos valores ausentes também. Certifique-se, porém, de que não causará vazamento de dados. Se você estiver preenchendo valores usando a média, não se esqueça de usar o mesmo valor de média na criação do modelo e no momento da predição.

As estratégias para usar o valor mais frequente ou uma constante podem ser aplicadas com dados numéricos ou do tipo string. A média e a mediana exigem dados numéricos.

A biblioteca fancyimpute implementa vários algoritmos e sua interface está em consonância com o scikit-learn. Infelizmente, a maioria dos algoritmos são *transdutivos*, o que significa que você não poderá chamar o método `.transform` por si só após a adequação do algoritmo. `IterativeImputer` é *indutivo* (foi

passado do fancyimpute para o scikit-learn) e aceita uma transformação após a adequação ao modelo.

Acrescentando colunas informativas

A ausência de dados por si só pode fornecer alguns sinais a um modelo. A biblioteca pandas é capaz de acrescentar uma nova coluna para informar que um valor estava ausente:

```
>>> def add_indicator(col):
... def wrapper(df):
... return df[col].isna().astype(int)
...
... return wrapper

>>> df1 = df.assign(
... cabin_missing=add_indicator("cabin")
... )
```

CAPÍTULO 5

Fazendo uma limpeza nos dados

Ferramentas genéricas como o pandas e ferramentas especializadas como o pyjanitor podem ser usadas para ajudar na limpeza dos dados.

Nomes das colunas

Ao usar o pandas, ter nomes de colunas apropriadas para Python torna possível o acesso aos atributos. A função clean_names do pyjanitor devolverá um DataFrame com as colunas em letras minúsculas e os espaços substituídos por underscores:

```
>>> import janitor as jn
>>> Xbad = pd.DataFrame(
... {
... "A": [1, None, 3],
... " sales numbers ": [20.0, 30.0, None],
... }
... )
>>> jn.clean_names(Xbad)
    a _sales_numbers_
0 1.0 20.0
1 NaN 30.0
2 3.0 NaN
```

DICA

Recomendo atualizar as colunas usando atribuição com índices, o método .assign ou atribuições com .loc ou .iloc. Também não recomendo usar atribuição para atualizar colunas no pandas. Por causa do risco de sobrescrever métodos existentes com o mesmo nome de uma coluna, não há garantias de que uma atribuição vá funcionar.

A biblioteca pyjanitor é conveniente, mas não nos permite remover espaços em branco em torno das colunas. Podemos usar o pandas para ter um controle

mais minucioso ao renomear as colunas:

```
>>> def clean_col(name):
... return (
... name.strip().lower().replace(" ", "_")
... )

>>> Xbad.rename(columns=clean_col)
    a sales_numbers
0 1.0 20.0
1 NaN 30.0
2 3.0 NaN
```

Substituindo valores ausentes

A função coalesce do pyjanitor recebe um DataFrame e uma lista de colunas para serem consideradas. Essa é uma funcionalidade semelhante àquela que vemos no Excel e em bancos de dados SQL. Ela devolve o primeiro valor não nulo para cada linha:

```
>>> jn.coalesce(
... Xbad,
... columns=["A", " sales numbers "],
... new_column_name="val",
... )
   val
0 1.0
1 30.0
2 3.0
```

Se quisermos substituir os valores ausentes por um valor específico, podemos usar o método .fillna do DataFrame:

```
>>> Xbad.fillna(10)
     A sales numbers
0 1.0 20.0
1 10.0 30.0
2 3.0 10.0
```

ou a função fill_empty do pyjanitor:

```
>>> jn.fill_empty(
... Xbad,
... columns=["A", " sales numbers "],
... value=10,
... )
     A sales numbers
0 1.0 20.0
```

```
1 10.0 30.0
2 3.0 10.0
```

Com frequência, usaremos imputações de dados mais específicas do pandas, do scikit-learn ou do fancyimpute para fazer substituição de valores nulos por coluna.

Para uma verificação de sanidade antes de criar os modelos, você pode usar o pandas para garantir que todos os valores ausentes foram tratados. O código a seguir devolve um único booleano para informar se há alguma célula com valor ausente em um DataFrame:

```
>>> df.isna().any().any()
True
```

CAPÍTULO 6

Explorando os dados

Dizem que é mais fácil treinar um SME (Subject Matter Expert, ou Especialista no Assunto) em ciência de dados do que fazer o inverso. Não tenho certeza se concordo 100% com essa afirmação, mas é verdade que os dados têm suas nuances e que um SME pode ajudar a compreendê-las. Ao entender o negócio e os dados, os especialistas são capazes de criar modelos melhores e causar melhor impacto em seus negócios.

Antes de criar um modelo, farei algumas análises de dados exploratórias. Isso me dará uma noção dos dados, mas também é uma ótima desculpa para conhecer as unidades de negócio que controlam esses dados e discutir os problemas com elas.

Tamanho dos dados

Novamente, usaremos o conjunto de dados do Titanic neste caso. A propriedade .shape do pandas devolve uma tupla com o número de linhas e de colunas:

```
>>> X.shape
(1309, 13)
```

Podemos ver que esse conjunto de dados contém 1.309 linhas e 13 colunas.

Estatísticas resumidas

Podemos usar o pandas para obter estatísticas resumidas de nossos dados. O método .describe nos dará também o número de valores diferentes de NaN. Vamos ver o resultado para a primeira e a última colunas:

```
>>> X.describe().iloc[:, [0, -1]]
        pclass embarked_S
count 1309.000000 1309.000000
mean -0.012831 0.698243
std 0.995822 0.459196
```

```
min -1.551881 0.000000
25% -0.363317 0.000000
50% 0.825248 1.000000
75% 0.825248 1.000000
max 0.825248 1.000000
```

A linha de contadores (count) nos informa que as duas colunas estão preenchidas. Não há valores ausentes. Também temos a média, o desvio-padrão, os valores mínimo e máximo e dos quartis.

NOTA

Um DataFrame pandas tem um atributo `iloc` que podemos usar para fazer operações com índices. Ele nos permite selecionar linhas e colunas com base na posição do índice. Passamos as posições das linhas na forma de um escalar, uma lista ou uma fatia (slice) e, em seguida, podemos adicionar uma vírgula e passar as posições das colunas na forma de um escalar, uma lista ou uma fatia.

No exemplo a seguir, extraímos a segunda e a quinta linhas, e as últimas três colunas:

```
>>> X.iloc[[1, 4], -3:]
    sex_male embarked_Q embarked_S
677 1.0 0 1
864 0.0 0 1
```

Há também um atributo `.loc`, e podemos extrair as linhas e colunas com base no nome (em vez da posição). Eis a mesma porção do DataFrame:

```
>>> X.loc[[677, 864], "sex_male":]
    sex_male embarked_Q embarked_S
677 1.0 0 1
864 0.0 0 1
```

Histograma

Um histograma é uma ótima ferramenta para visualizar dados numéricos. Podemos ver quantos modos há, bem como observar a distribuição (veja a Figura 6.1). A biblioteca pandas tem um método `.plot` para exibir histogramas:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.fare.plot(kind="hist", ax=ax)
>>> fig.savefig("images/mlpr_0601.png", dpi=300)
```

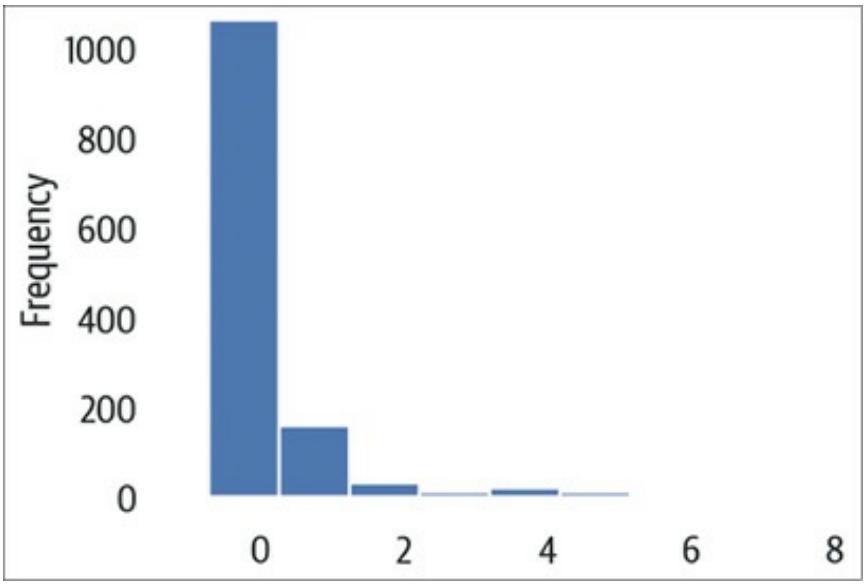


Figura 6.1 – Histograma gerado com o pandas.

Ao usar a biblioteca seaborn, podemos gerar um histograma de valores contínuos em relação ao alvo (veja a Figura 6.2).

```
fig, ax = plt.subplots(figsize=(12, 8))
mask = y_train == 1
ax = sns.distplot(X_train[mask].fare, label='survived')
ax = sns.distplot(X_train[~mask].fare, label='died')
ax.set_xlim(-1.5, 1.5)
ax.legend()
fig.savefig('images/mlpr_0602.png', dpi=300, bbox_inches='tight')
```

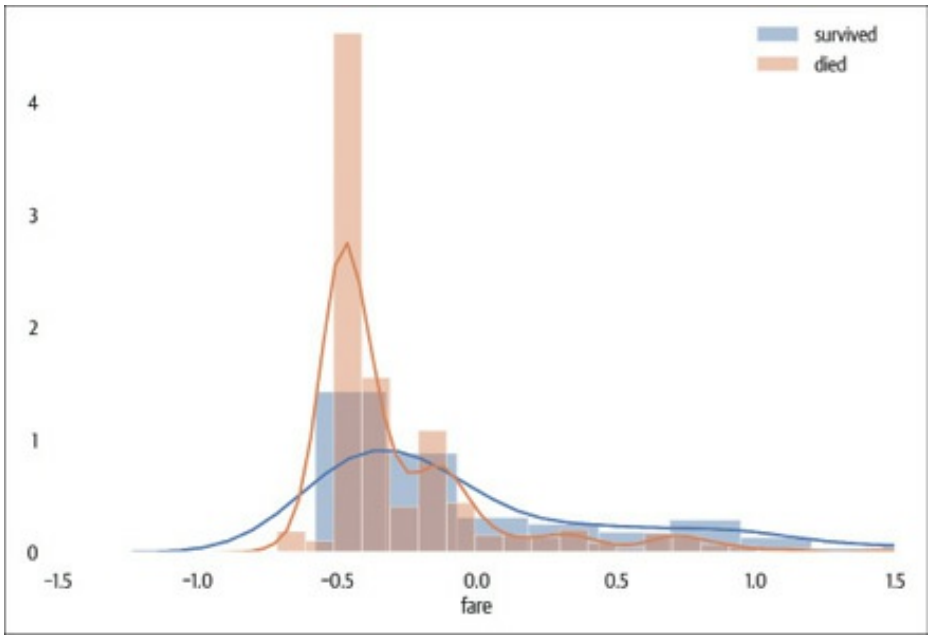


Figura 6.2 – Histograma gerado com o seaborn.

Gráfico de dispersão

Um gráfico de dispersão (scatter plot) mostra o relacionamento entre duas colunas numéricas (veja a Figura 6.3). Novamente, isso é fácil de fazer com o pandas. Ajuste o parâmetro alpha caso tenha dados que se sobreponham.

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> X.plot.scatter(
... x="age", y="fare", ax=ax, alpha=0.3
... )
>>> fig.savefig("images/mlpr_0603.png", dpi=300)
```

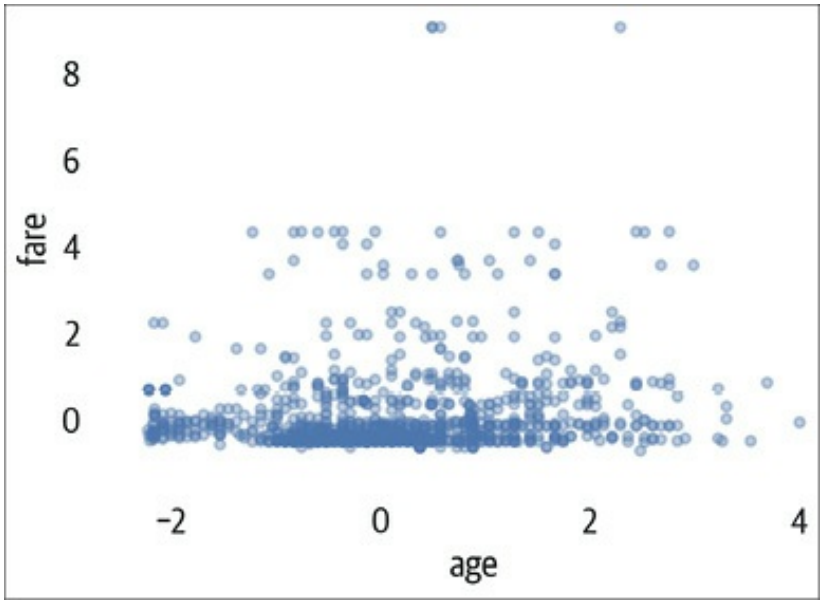


Figura 6.3 – Gráfico de dispersão gerado com o pandas.

Não parece haver muita correlação entre esses dois atributos. Podemos usar a correlação de Pearson entre duas colunas (pandas) aplicando o método .corr para quantificá-la:

```
>>> X.age.corr(X.fare)
0.17818151568062093
```

Gráfico conjunto

O Yellowbrick tem um gráfico de dispersão mais sofisticado que inclui histogramas nas bordas e uma linha de regressão, chamada gráfico conjunto (joint plot) – veja a Figura 6.4.

```
>>> from yellowbrick.features import (
```

```
... JointPlotVisualizer,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> jpv = JointPlotVisualizer(
... feature="age", target="fare"
... )
>>> jpv.fit(X["age"], X["fare"])
>>> jpv.poof()
>>> fig.savefig("images/mlpr_0604.png", dpi=300)
```

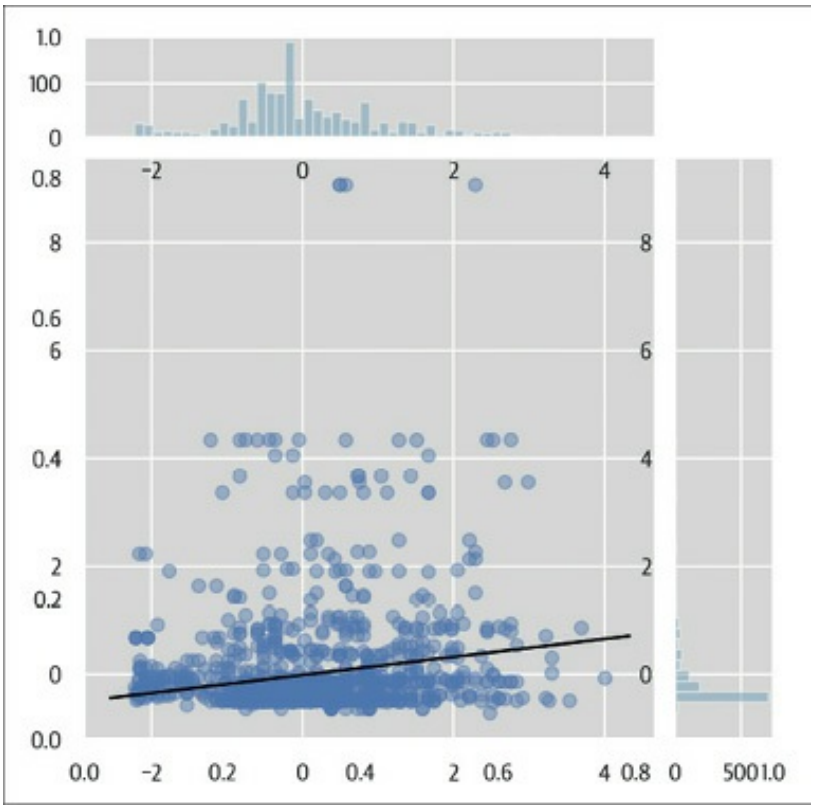


Figura 6.4 – Gráfico conjunto gerado com o Yellowbrick.

ALERTA

Nesse método .fit, X e y se referem, cada um, a uma coluna. Em geral, X é um DataFrame, e não uma série.

Também podemos usar a biblioteca seaborn (*https://seaborn.pydata.org*) para criar um gráfico conjunto (veja a Figura 6.5):

```
>>> from seaborn import jointplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
```

```
>>> p = jointplot(
... "age", "fare", data=new_df, kind="reg"
... )
>>> p.savefig("images/mlpr_0605.png", dpi=300)
```

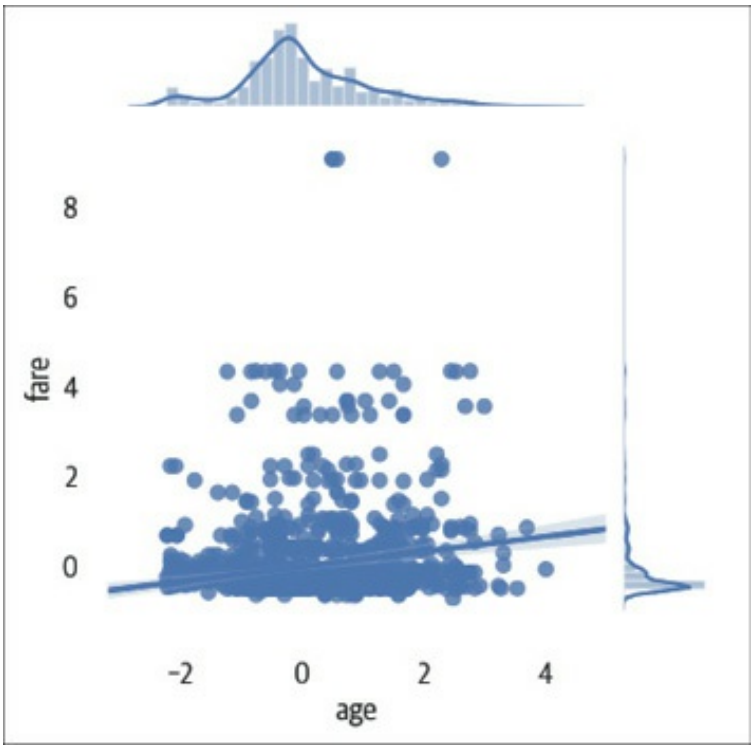


Figura 6.5 – Gráfico conjunto gerado com o seaborn.

Matriz de pares

A biblioteca seaborn é capaz de criar uma matriz de pares (pair grid) – veja a Figura 6.6. Esse gráfico contém uma matriz de colunas e estimativas de densidade kernel. Para colorir de acordo com uma coluna de um DataFrame, utilize o parâmetro `hue`. Ao colorir de acordo com o alvo, podemos ver se os atributos lhe causam efeitos diferentes:

```
>>> from seaborn import pairplot
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> vars = ["pclass", "age", "fare"]
>>> p = pairplot(
... new_df, vars=vars, hue="target", kind="reg"
... )
>>> p.savefig("images/mlpr_0606.png", dpi=300)
```

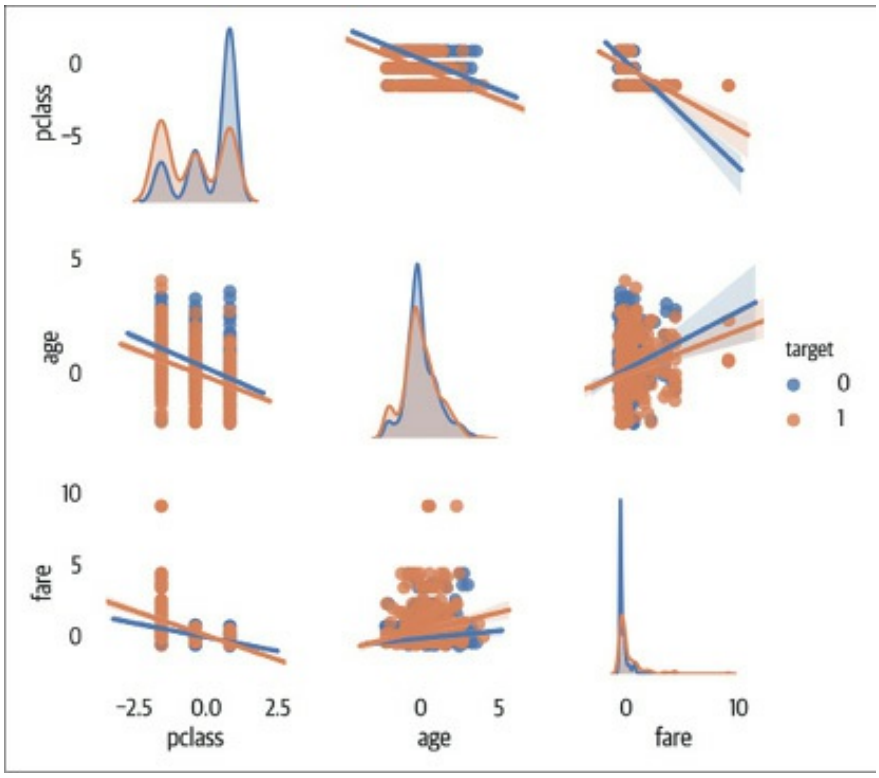


Figura 6.6 – Matriz de pares gerada com o seaborn.

Gráfico de caixas e gráfico violino

O seaborn tem diversos gráficos para visualizar as distribuições. Mostraremos exemplos de um gráfico de caixas (box plot) e de um gráfico violino (violin plot) – veja as figuras 6.7 e 6.8. Esses gráficos podem exibir um atributo em relação a um alvo:

```
>>> from seaborn import box plot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> boxplot(x="target", y="age", data=new_df)
>>> fig.savefig("images/mlpr_0607.png", dpi=300)
```

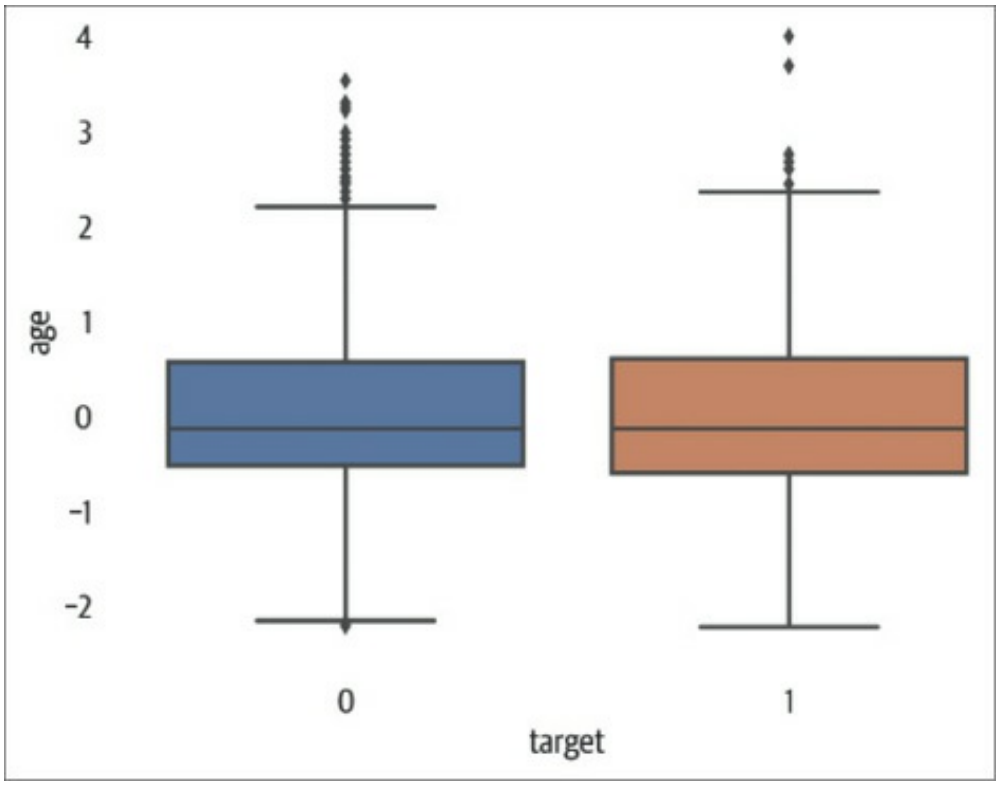


Figura 6.7– – Gráfico de caixas gerado com o seaborn.

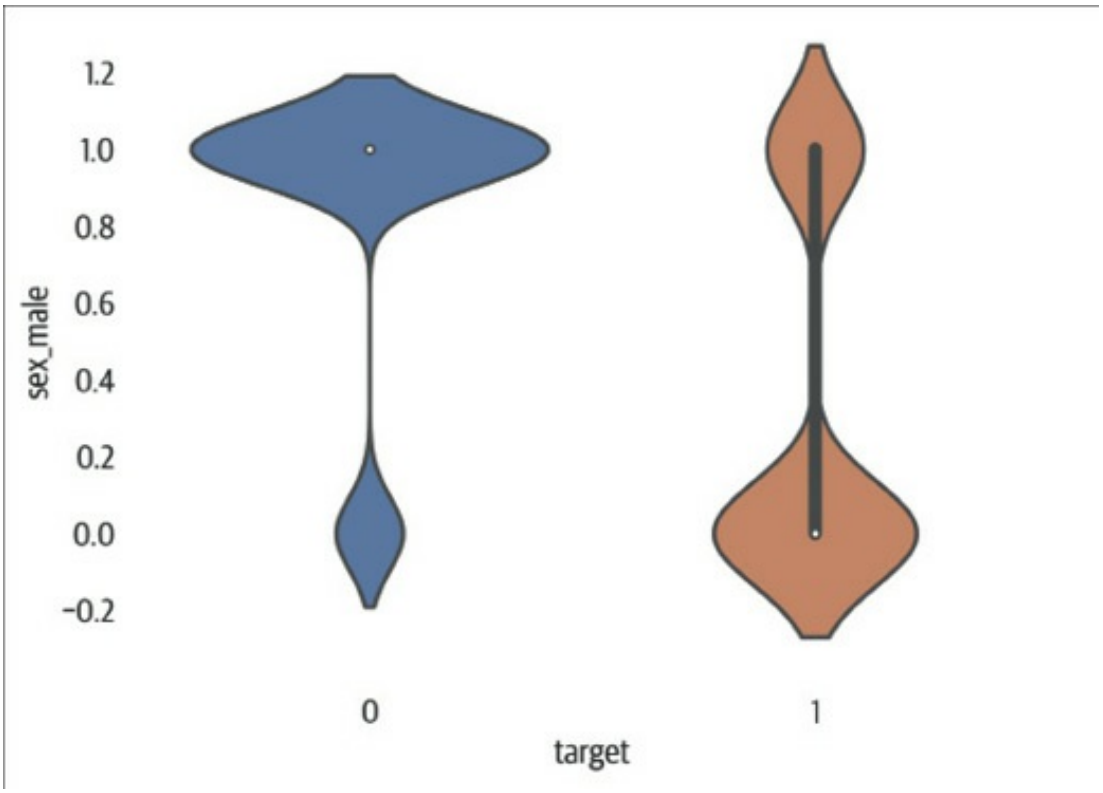


Figura 6.8 – Gráfico violino gerado com o seaborn.

Gráficos violino podem ajudar na visualização de distribuições:

```
>>> from seaborn import violinplot
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> violinplot(
... x="target", y="sex_male", data=new_df
... )
>>> fig.savefig("images/mlpr_0608.png", dpi=300)
```

Comparando dois valores ordinais

Neste exemplo, temos um código pandas que compara duas categorias ordinais. Estou simulando isso separando a idade em dez quantis e pclass em três grupos. O gráfico foi normalizado, portanto ele preenche toda a área vertical. Isso facilita ver que, no quantil 40%, a maioria das passagens era de terceira classe (veja a Figura 6.9):

```
>>> fig, ax = plt.subplots(figsize=(8, 6))
>>> (
... X.assign(
... age_bin=pd.qcut(
... X.age, q=10, labels=False
... ),
... class_bin=pd.cut(
... X.pclass, bins=3, labels=False
... ),
... )
... .groupby(["age_bin", "class_bin"])
... .size()
... .unstack()
... .pipe(lambda df: df.div(df.sum(1), axis=0))
... .plot.bar(
... stacked=True,
... width=1,
... ax=ax,
... cmap="viridis",
... )
... .legend(bbox_to_anchor=(1, 1))
... )
>>> fig.savefig(
... "image/mlpr_0609.png",
... dpi=300,
... bbox_inches="tight",
```

...)

NOTA

As linhas:

```
.groupby(["age_bin", "class_bin"])
.size()
.unstack()
```

podem ser substituídas por:

```
.pipe(lambda df: pd.crosstab(
    df.age_bin, df.class_bin)
)
```

No pandas, em geral há mais de uma maneira de fazer uma tarefa, e algumas funções auxiliares que fazem parte de outras funcionalidades estão disponíveis, por exemplo, pd.crosstab.

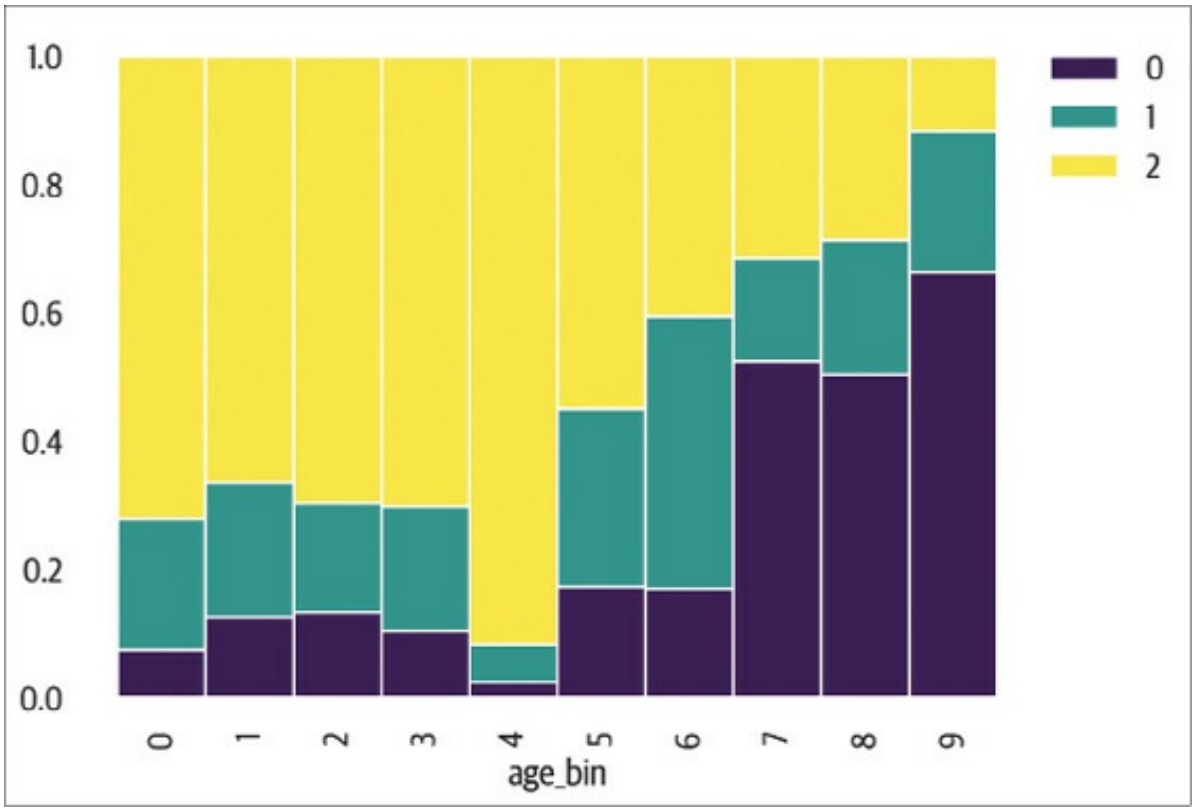


Figura 6.9 – Comparando valores ordinais.

Correlação

O Yellowbrick é capaz de fazer comparações aos pares entre os atributos (veja a Figura 6.10). O gráfico a seguir mostra uma correlação de Pearson (o parâmetro algorithm também aceita 'spearman' e 'covariance'):

```
>>> from yellowbrick.features import Rank2D
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pcv = Rank2D(
... features=X.columns, algorithm="pearson"
... )
>>> pcv.fit(X, y)
>>> pcv.transform(X)
>>> pcv.poof()
>>> fig.savefig(
... "images/mlpr_0610.png",
... dpi=300,
... bbox_inches="tight",
... )
```

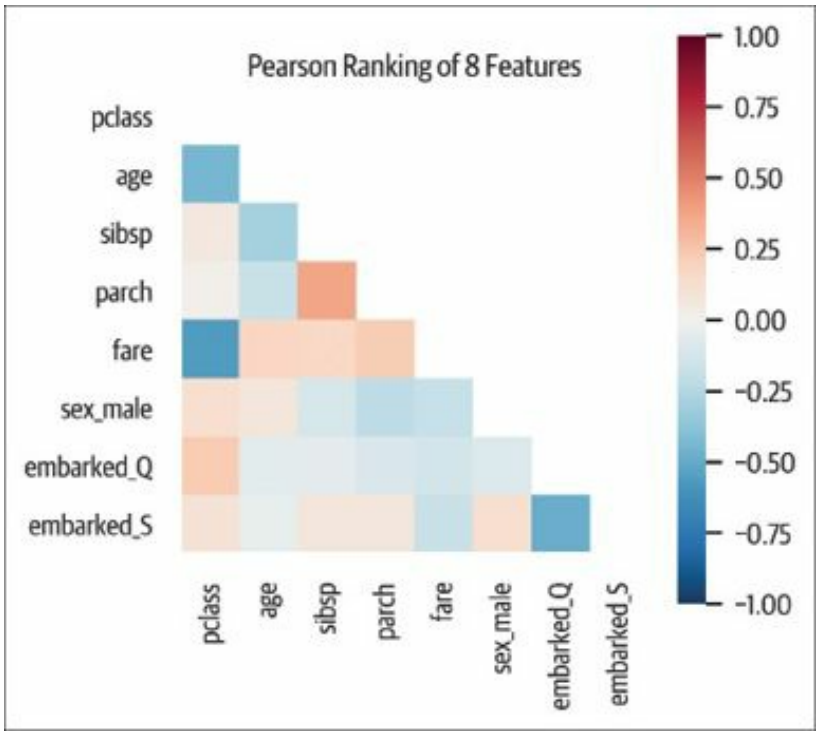


Figura 6.10 – Correlação de covariância gerada com o Yellowbrick.

Um gráfico semelhante – um heat map (mapa de calor) – está disponível na biblioteca seaborn (veja a Figura 6.11). Temos de passar um DataFrame de correlações como dado. Infelizmente, a barra de cores não se estende entre -1 e 1, a menos que os valores na matriz o façam, ou que acrescentemos os parâmetros vmin e vmax:

```
>>> from seaborn import heatmap
>>> fig, ax = plt.subplots(figsize=(8, 8))
>>> ax = heatmap(
... X.corr(),
... fmt=".2f",
... annot=True,
... ax=ax,
... cmap="RdBu_r",
... vmin=-1,
... vmax=1,
... )
>>> fig.savefig(
... "images/mlpr_0611.png",
... dpi=300,
... bbox_inches="tight",
... )
```

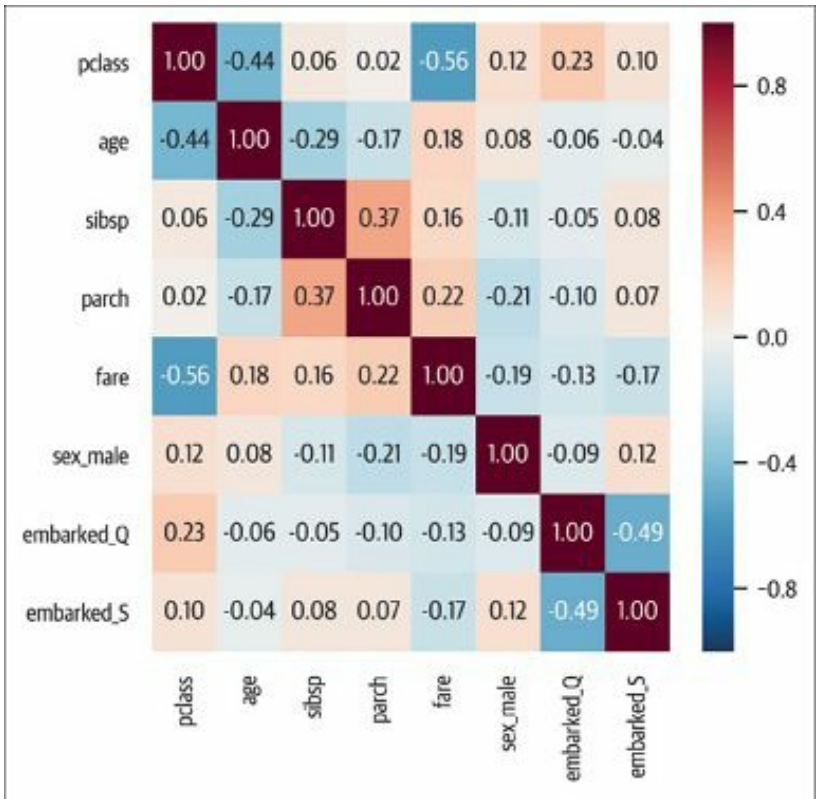


Figura 6.11 – Heat map gerado com o seaborn.

A biblioteca pandas também pode apresentar uma correlação entre colunas do DataFrame. Exibiremos apenas as duas primeiras colunas do resultado. O método default é 'pearson', mas é possível definir também o parâmetro method com 'kendall', 'spearman' ou com um callable personalizado que devolva um

número de ponto flutuante, dadas as duas colunas:

```
>>> X.corr().iloc[:, :2]
              pclass age
pclass 1.000000 -0.440769
age -0.440769 1.000000
sibsp 0.060832 -0.292051
parch 0.018322 -0.174992
fare -0.558831 0.177205
sex_male 0.124617 0.077636
embarked_Q 0.230491 -0.061146
embarked_S 0.096335 -0.041315
```

Colunas com alto grau de correlação não agregam valor e podem prejudicar a interpretação da importância dos atributos e dos coeficientes de regressão. A seguir, apresentamos um código para identificar as colunas correlacionadas. Em nossos dados, nenhuma das colunas tem alto grau de correlação (lembre-se de que removemos a coluna sex_male).

Se tivéssemos colunas correlacionadas, poderíamos optar por remover as colunas de level_0 ou level_1 dos dados:

```
>>> def correlated_columns(df, threshold=0.95):
... return (
... df.corr()
... .pipe(
... lambda df1: pd.DataFrame(
... np.tril(df1, k=-1),
... columns=df.columns,
... index=df.columns,
... )
... )
... .stack()
... .rename("pearson")
... .pipe(
... lambda s: s[
... s.abs() > threshold
... ].reset_index()
... )
... .query("level_0 not in level_1")
... )

>>> correlated_columns(X)
Empty DataFrame
Columns: [level_0, level_1, pearson]
Index: []
```

Ao usar o conjunto de dados com mais colunas, podemos ver que muitas delas estão correlacionadas:

```
>>> c_df = correlated_columns(agg_df)
>>> c_df.style.format({"pearson": "{:.2f}"})
      level_0 level_1 pearson
3 pclass_mean pclass 1.00
4 pclass_mean pclass_min 1.00
5 pclass_mean pclass_max 1.00
6 sibsp_mean sibsp_max 0.97
7 parch_mean parch_min 0.95
8 parch_mean parch_max 0.96
9 fare_mean fare 0.95
10 fare_mean fare_max 0.98
12 body_mean body_min 1.00
13 body_mean body_max 1.00
14 sex_male sex_female -1.00
15 embarked_S embarked_C -0.95
```

RadViz

Um gráfico RadViz exibe todas as amostras em um círculo, com os atributos na circunferência (veja a Figura 6.12). Os valores são normalizados, e você pode imaginar que cada figura tenha uma mola que puxe as amostras para ela com base no valor.

Essa é uma técnica para visualizar o grau de separação entre os alvos.

O Yellowbrick é capaz de fazer isso:

```
>>> from yellowbrick.features import RadViz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> rv = RadViz(
... classes=["died", "survived"],
... features=X.columns,
... )
>>> rv.fit(X, y)
>>> _ = rv.transform(X)
>>> rv.poof()
>>> fig.savefig("images/mlpr_0612.png", dpi=300)
```

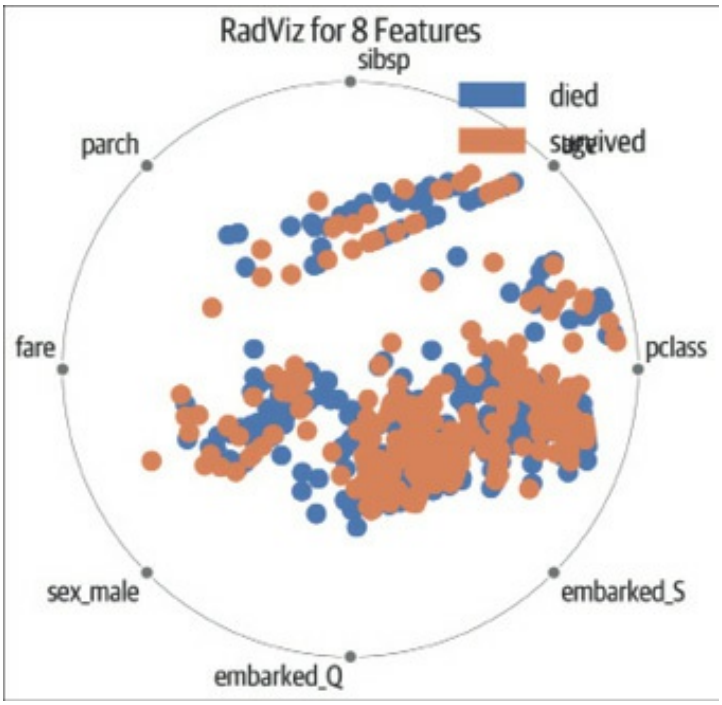


Figura 6.12 – Gráfico RadViz gerado com o Yellowbrick.

A biblioteca pandas também gera gráficos RadViz (veja a Figura 6.13).

```
>>> from pandas.plotting import radviz
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> radviz(
... new_df, "target", ax=ax, colormap="PiYG"
... )
>>> fig.savefig("images/mlpr_0613.png", dpi=300)
```

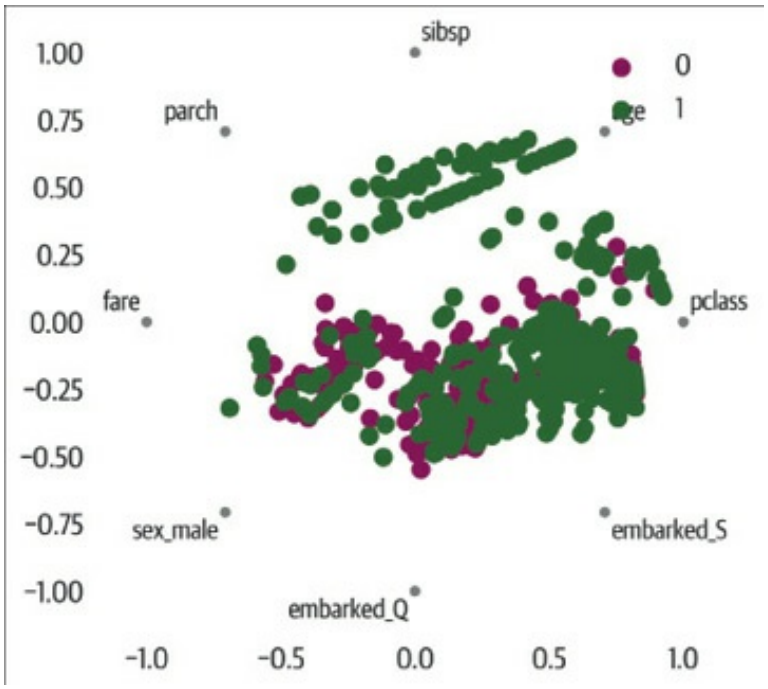


Figura 6.13 – Gráfico RadViz gerado com o pandas.

Coordenadas paralelas

Para dados multivariados, podemos usar um gráfico de coordenadas paralelas para observar visualmente os agrupamentos (veja as figuras 6.14 e 6.15).

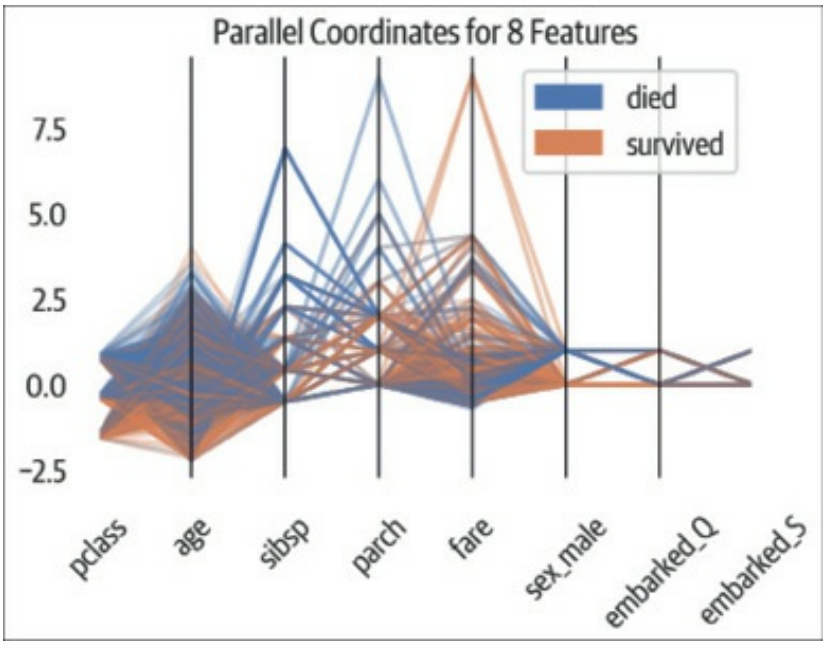


Figura 6.14 – Gráfico de coordenadas paralelas gerado com o Yellowbrick.

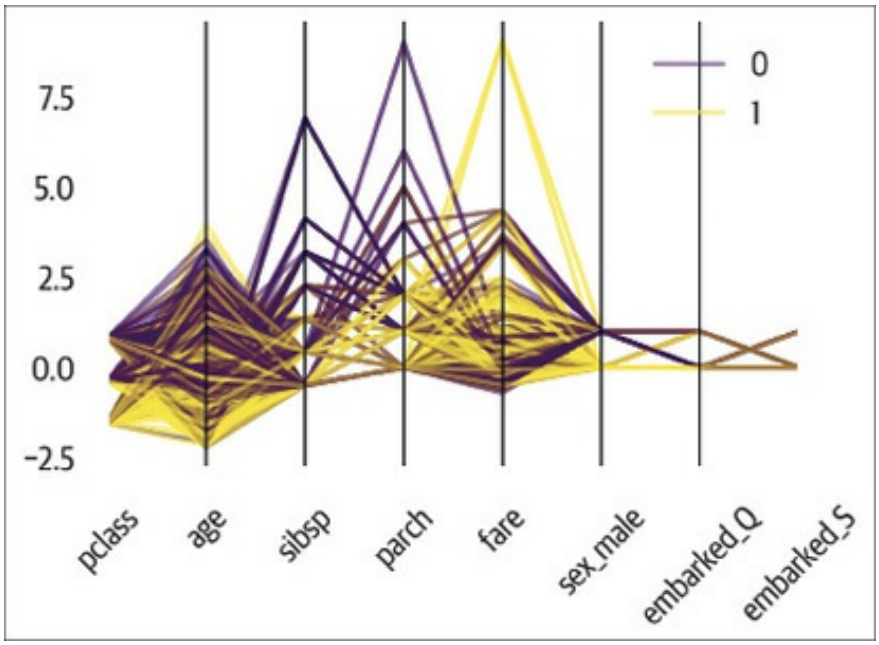


Figura 6.15 – Gráfico de coordenadas paralelas gerado com o pandas.

Mais uma vez, apresentamos uma versão com o Yellowbrick:

```
>>> from yellowbrick.features import (
... ParallelCoordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pc = ParallelCoordinates(
... classes=["died", "survived"],
... features=X.columns,
... )
>>> pc.fit(X, y)
>>> pc.transform(X)
>>> ax.set_xticklabels(
... ax.get_xticklabels(), rotation=45
... )
>>> pc.poof()
>>> fig.savefig("images/mlpr_0614.png", dpi=300)
```

E uma versão com o pandas:

```
>>> from pandas.plotting import (
... parallel_coordinates,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> new_df = X.copy()
>>> new_df["target"] = y
>>> parallel_coordinates(
... new_df,
```

```
... "target",
... ax=ax,
... colormap="viridis",
... alpha=0.5,
... )
>>> ax.set_xticklabels(
... ax.get_xticklabels(), rotation=45
... )
>>> fig.savefig(
... "images/mlpr_0615.png",
... dpi=300,
... bbox_inches="tight",
... )
```

CAPÍTULO 7

Pré-processamento dos dados

Neste capítulo, exploraremos os passos comuns de pré-processamento de dados com os dados a seguir:

```
>>> X2 = pd.DataFrame(
... {
... "a": range(5),
... "b": [-100, -50, 0, 200, 1000],
... }
... )
>>> X2
   a b
0 0 -100
1 1 -50
2 2 0
3 3 200
4 4 1000
```

Padronize os dados

Alguns algoritmos, como o SVM, apresentam melhor desempenho quando os dados estão *padronizados.* Cada coluna deve ter um valor de média igual a zero e um desvio-padrão igual a 1. O sklearn disponibiliza um método .fit_transform que combina .fit e .transform:

```
>>> from sklearn import preprocessing
>>> std = preprocessing.StandardScaler()
>>> std.fit_transform(X2)
array([[-1.41421356, -0.75995002],
       [-0.70710678, -0.63737744],
       [ 0. , -0.51480485],
       [ 0.70710678, -0.02451452],
       [ 1.41421356, 1.93664683]])
```

Após a adequação dos dados, há diversos atributos que podemos inspecionar:

```
>>> std.scale_
array([ 1.41421356, 407.92156109])
```

```
>>> std.mean_
array([ 2., 210.])
>>> std.var_
array([2.000e+00, 1.664e+05])
```

A seguir, apresentamos uma versão com o pandas. Lembre-se de que você deverá manter o controle da média e do desvio-padrão originais caso use esse código no pré-processamento. Qualquer amostra que você vá usar para predições mais tarde deverá ser padronizada com esses mesmos valores:

```
>>> X_std = (X2 - X2.mean()) / X2.std()
>>> X_std
          a b
0 -1.264911 -0.679720
1 -0.632456 -0.570088
2 0.000000 -0.460455
3 0.632456 -0.021926
4 1.264911 1.732190

>>> X_std.mean()
a 4.440892e-17
b 0.000000e+00
dtype: float64

>>> X_std.std()
a 1.0
b 1.0
dtype: float64
```

A biblioteca fastai também implementa esse procedimento:

```
>>> X3 = X2.copy()
>>> from fastai.structured import scale_vars
>>> scale_vars(X3, mapper=None)
>>> X3.std()
a 1.118034
b 1.118034
dtype: float64
>>> X3.mean()
a 0.000000e+00
b 4.440892e-17
dtype: float64
```

Escale para um intervalo

Escalar para um intervalo consiste em traduzir os dados de modo que estejam

entre 0 e 1, inclusive. Ter os dados limitados pode ser conveniente. No entanto, se houver valores discrepantes, tome cuidado ao usar este recurso:

```
>>> from sklearn import preprocessing
>>> mms = preprocessing.MinMaxScaler()
>>> mms.fit(X2)
>>> mms.transform(X2)
array([[0. , 0. ],
       [0.25 , 0.04545],
       [0.5 , 0.09091],
       [0.75 , 0.27273],
       [1. , 1. ]])
```

Eis uma versão com o pandas:

```
>>> (X2 - X2.min()) / (X2.max() - X2.min())
      a b
0 0.00 0.000000
1 0.25 0.045455
2 0.50 0.090909
3 0.75 0.272727
4 1.00 1.000000
```

Variáveis dummy

Podemos usar o pandas para criar variáveis dummy a partir de dados de categoria. Esse procedimento é também conhecido como codificação one-hot (one-hot encoding) ou codificação de indicadores (indicator encoding). Variáveis dummy são particularmente úteis caso os dados sejam nominais (não ordenados). A função get_dummies do pandas cria várias colunas para uma coluna de categorias, cada uma contendo 1 ou 0 de acordo com o fato de a coluna original ter a respectiva categoria:

```
>>> X_cat = pd.DataFrame(
... {
... "name": ["George", "Paul"],
... "inst": ["Bass", "Guitar"],
... }
... )
>>> X_cat
    name inst
0 George Bass
1 Paul Guitar
```

A seguir, apresentamos uma versão com o pandas. Observe que a opção drop_first pode ser usada para eliminar uma coluna (uma das colunas dummy é

uma combinação linear das demais colunas):

```
>>> pd.get_dummies(X_cat, drop_first=True)
  name_Paul inst_Guitar
0 0 0
1 1 1
```

A biblioteca pyjanitor também pode separar colunas com a função expand_column:

```
>>> X_cat2 = pd.DataFrame(
... {
... "A": [1, None, 3],
... "names": [
... "Fred,George",
... "George",
... "John,Paul",
... ],
... }
... )
>>> jn.expand_column(X_cat2, "names", sep=",")
    A names Fred George John Paul
0 1.0 Fred,George 1 1 0 0
1 NaN George 0 1 0 0
2 3.0 John,Paul 0 0 1 1
```

Se tivermos dados nominais com alta cardinalidade, podemos usar uma *codificação de rótulos* (label encoding). Esse recurso será apresentado na próxima seção.

Codificador de rótulos

Uma alternativa à codificação de variáveis dummy é a codificação de rótulos. Nesse caso, cada dado de categoria será atribuído a um número. É um método conveniente para dados com alta cardinalidade. Esse codificador impõe uma ordem, que poderá ser ou não desejável. Menos espaço poderá ser ocupado em comparação com uma codificação one-hot, e alguns algoritmos (de árvore) são capazes de lidar com essa codificação.

O codificador de rótulos consegue lidar somente com uma coluna de cada vez:

```
>>> from sklearn import preprocessing
>>> lab = preprocessing.LabelEncoder()
>>> lab.fit_transform(X_cat)
array([0,1])
```

Se você tiver valores codificados, aplique o método .inverse_transform para decodificá-los:

```
>>> lab.inverse_transform([1, 1, 0])
array(['Guitar', 'Guitar', 'Bass'], dtype=object)
```

O pandas também pode ser usado para uma codificação de rótulos. Inicialmente, você deve converter a coluna em um tipo para categorias e, em seguida, extrair daí o código numérico.

O código a seguir criará uma nova série de dados numéricos a partir de uma série do pandas. Usaremos o método .as_ordered para garantir que a categoria esteja ordenada:

```
>>> X_cat.name.astype(
... "category"
... ).cat.as_ordered().cat.codes + 1
0 1
1 2
dtype: int8
```

Codificação de frequência

Outra opção para lidar com dados de categoria com alta cardinalidade é a *codificação de frequência* (frequency encoding). Isso significa substituir o nome da categoria pelo contador que ela tinha nos dados de treinamento. Usaremos o pandas para essa tarefa. Inicialmente, utilizaremos o método .value_counts do pandas para fazer um mapeamento (uma série do pandas que mapeia strings a contadores). Com o mapeamento, podemos usar o método .map para fazer a codificação:

```
>>> mapping = X_cat.name.value_counts()
>>> X_cat.name.map(mapping)
0 1
1 1
Name: name, dtype: int64
```

Não se esqueça de armazenar o mapeamento dos dados de treinamento para que seja possível codificar dados futuros com os mesmos dados.

Extraindo categorias a partir de strings

Uma forma de aumentar a precisão do modelo de dados do Titanic é extrair os títulos dos nomes. Um truque rápido para encontrar as trincas mais comuns é usar a classe Counter:

```
>>> from collections import Counter
>>> c = Counter()
>>> def triples(val):
... for i in range(len(val)):
... c[val[i : i + 3]] += 1
>>> df.name.apply(triples)
>>> c.most_common(10)
[(', M', 1282),
 (' Mr', 954),
 ('r. ', 830),
 ('Mr.', 757),
 ('s. ', 460),
 ('n, ', 320),
 (' Mi', 283),
 ('iss', 261),
 ('ss.', 261),
 ('Mis', 260)]
```

Podemos ver que “Mr.” e “Miss.” são muito comuns.

Outra opção é usar uma expressão regular para extrair a letra maiúscula seguida de letras minúsculas e um ponto:

```
>>> df.name.str.extract(
... "([A-Za-z]+)\.", expand=False
... ).head()
0 Miss
1 Master
2 Miss
3 Mr
4 Mrs
Name: name, dtype: object
```

Podemos usar .value_counts para ver a frequência desses títulos:

```
>>> df.name.str.extract(
... "([A-Za-z]+)\.", expand=False
... ).value_counts()
Mr 757
Miss 260
Mrs 197
Master 61
Dr 8
Rev 8
Col 4
Mlle 2
Ms 2
Major 2
```

```
Dona 1
Don 1
Lady 1
Countess 1
Capt 1
Sir 1
Mme 1
Jonkheer 1
Name: name, dtype: int64
```

NOTA

Uma discussão completa sobre expressões regulares está além do escopo deste livro. A expressão anterior captura um grupo com um ou mais caracteres alfabéticos, seguido de um ponto.

Ao usar essas manipulações e o pandas, você pode criar variáveis dummy ou combinar colunas com contadores baixos em outras categorias (ou descartá-las).

Outras codificações de categoria

A biblioteca categorical_encoding (*https://oreil.ly/JbxWG*) é um conjunto de transformadores do scikit-learn, usados para converter dados de categorias em dados numéricos. Um bom recurso dessa biblioteca é que ela gera DataFrames do pandas (de modo diferente do scikit-learn, que transforma os dados em arrays numpy).

Um algoritmo implementado nessa biblioteca é um codificador de hashes. É útil caso você não saiba com antecedência quantas categorias há, ou se estiver usando um conjunto de palavras para representar texto. O algoritmo gerará o hash de colunas de categorias em `n_components`. Se estiver usando online learning (aprendizado online), isto é, modelos que podem ser atualizados, esse recurso poderá ser muito conveniente:

```
>>> import category_encoders as ce
>>> he = ce.HashingEncoder(verbose=1)
>>> he.fit_transform(X_cat)
   col_0 col_1 col_2 col_3 col_4 col_5 col_6 col_7
0 0 0 0 0 1 0 1 0 0
1 0 2 0 0 0 0 0 0 0
```

O codificador de ordinais (ordinal encoder) pode converter colunas de

categorias que tenham uma ordem em uma única coluna de números. Em nosso exemplo, converteremos a coluna de tamanho (size) em números ordinais. Se houver um valor ausente no dicionário de mapeamento, um valor default igual a -1 será usado:

```
>>> size_df = pd.DataFrame(
... {
... "name": ["Fred", "John", "Matt"],
... "size": ["small", "med", "xxl"],
... }
... )
>>> ore = ce.OrdinalEncoder(
... mapping=[
... {
... "col": "size",
... "mapping": {
... "small": 1,
... "med": 2,
... "lg": 3,
... },
... }
... ]
... )
>>> ore.fit_transform(size_df)
   name size
0 Fred 1.0
1 John 2.0
2 Matt -1.0
```

A referência em *https://oreil.ly/JUtYh* explica vários algoritmos que estão na biblioteca categorical_encoding.

Se você tiver dados com alta cardinalidade (um número grande de valores únicos), considere usar um dos codificadores bayesianos (Bayesian encoders) que geram uma única coluna por coluna de categoria. São eles: TargetEncoder, LeaveOneOutEncoder, WOEEncoder, JamesSteinEncoder e MEstimateEncoder.

Por exemplo, para converter a coluna survived (sobrevivência) dos dados do Titanic em uma combinação da probabilidade posterior do alvo com a probabilidade anterior dada a informação de título (dado de categoria), utilize o código a seguir:

```
>>> def get_title(df):
... return df.name.str.extract(
... "([A-Za-z]+)\.", expand=False
```

```
... )
>>> te = ce.TargetEncoder(cols="Title")
>>> te.fit_transform(
... df.assign(Title=get_title), df.survived
... )["Title"].head()
0 0.676923
1 0.508197
2 0.676923
3 0.162483
4 0.786802
Name: Title, dtype: float64
```

Engenharia de dados para datas

A biblioteca fastai tem uma função add_datepart que gerará colunas com atributos de data com base em uma coluna de data e hora. Isso é conveniente, pois a maioria dos algoritmos de machine learning não é capaz de inferir esse tipo de sinal a partir da representação numérica de uma data:

```
>>> from fastai.tabular.transform import (
... add_datepart,
... )
>>> dates = pd.DataFrame(
... {
... "A": pd.to_datetime(
... ["9/17/2001", "Jan 1, 2002"]
... )
... }
... )

>>> add_datepart(dates, "A")
>>> dates.T
                      0 1
AYear 2001 2002
AMonth 9 1
AWeek 38 1
ADay 17 1
ADayofweek 0 1
ADayofyear 260 1
AIs_month_end False False
AIs_month_start False True
AIs_quarter_end False False
AIs_quarter_start False True
AIs_year_end False False
AIs_year_start False True
```

AElapsed 1000684800 1009843200

ALERTA

add_datepart altera o DataFrame, algo que o pandas pode fazer, mas, em geral, não faz!

Adição do atributo col_na

A biblioteca fastai costumava ter uma função para criar uma coluna para preenchimento de um valor ausente (com a mediana) e indicar que um valor estava ausente. Saber que um valor estava ausente poderia ser usado como uma informação. Eis uma cópia da função e um exemplo de sua utilização:

```
>>> from pandas.api.types import is_numeric_dtype
>>> def fix_missing(df, col, name, na_dict):
... if is_numeric_dtype(col):
... if pd.isnull(col).sum() or (
... name in na_dict
... ):
... df[name + "_na"] = pd.isnull(col)
... filler = (
... na_dict[name]
... if name in na_dict
... else col.median()
... )
... df[name] = col.fillna(filler)
... na_dict[name] = filler
... return na_dict
>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> fix_missing(data, data.A, "A", {})
{'A': 5.0}
>>> data
     A A_na
0 0.0 False
1 5.0 True
2 5.0 False
3 100.0 False
```

A seguir, apresentamos uma versão com o pandas:

```
>>> data = pd.DataFrame({"A": [0, None, 5, 100]})
>>> data["A_na"] = data.A.isnull()
>>> data["A"] = data.A.fillna(data.A.median())
```

Engenharia de dados manual

O pandas pode ser usado para gerar novos atributos. Para o conjunto de dados do Titanic, podemos agregar dados de cabine (idade máxima por cabine, idade média por cabine etc.). Para obter dados agregados por cabine, utilize o método .groupby do pandas para criá-los. Em seguida, alinhe-os com os dados originais usando o método .merge:

```
>>> agg = (
... df.groupby("cabin")
... .agg("min,max,mean,sum".split(","))
... .reset_index()
... )
>>> agg.columns = [
... "_".join(c).strip("_")
... for c in agg.columns.values
... ]
>>> agg_df = df.merge(agg, on="cabin")
```

Se você quisesse calcular as colunas “boas” ou “ruins”, poderia criar outra coluna que fosse a soma das colunas agregadas (ou usar outra operação matemática). Essa é uma forma de arte, e exige uma boa compreensão dos dados.

CAPÍTULO 8

Seleção de atributos

Usamos seleção de atributos (feature selection) para selecionar aqueles que sejam úteis ao modelo. Atributos irrelevantes podem causar um efeito negativo em um modelo. Atributos correlacionados podem deixar os coeficientes de uma regressão (ou a importância dos atributos em modelos de árvore) instáveis ou difíceis de interpretar.

A *maldição da dimensionalidade* (curse of dimensionality) é outra questão a ser considerada. À medida que você aumentar o número de dimensões de seus dados, eles se tornarão mais esparsos. Isso pode dificultar a obtenção de um sinal, a menos que você tenha mais dados. Cálculos de vizinhança tendem a perder a utilidade à medida que mais dimensões são adicionadas.

Além disso, o tempo para treinamento dos dados em geral é uma função do número de colunas (às vezes, poderá ser pior que uma função linear). Se você conseguir ser conciso e preciso com suas colunas, será possível obter um modelo melhor em menos tempo. Descreveremos alguns exemplos usando o conjunto de dados `agg_df` do capítulo anterior. Lembre-se de que esse é o conjunto de dados do Titanic com algumas colunas extras com informações sobre as cabines. Como esse conjunto de dados agrega valores numéricos para cada cabine, ele exibirá muitas correlações. Outras opções incluem uso de PCA e a observação de `.feature_importances_` em um classificador de árvore.

Colunas colineares

Podemos usar a função `correlated_columns` definida antes, ou executar o código a seguir para encontrar colunas que tenham um coeficiente de correlação maior ou igual a 0,95:

```
>>> limit = 0.95
>>> corr = agg_df.corr()
>>> mask = np.triu(
... np.ones(corr.shape), k=1
```

```
... ).astype(bool)
>>> corr_no_diag = corr.where(mask)
>>> coll = [
... c
... for c in corr_no_diag.columns
... if any(abs(corr_no_diag[c]) > threshold)
... ]
>>> coll
['pclass_min', 'pclass_max', 'pclass_mean',
 'sibsp_mean', 'parch_mean', 'fare_mean',
 'body_max', 'body_mean', 'sex_male', 'embarked_S']
```

O visualizador Rank2 do Yellowbrick, mostrado antes, gerará um heat map (mapa de calor) com as correlações.

O pacote rfpimp (*https://oreil.ly/MsnXc*) tem um recurso de visualização de *multicolinearidade*. A função plot_dependence_heatmap faz o treinamento de uma floresta aleatória (random forest) para cada coluna numérica a partir das outras colunas em um conjunto de dados de treinamento. O valor da dependência é a pontuação R2 das estimativas OOB (Out-Of-Bag) para a predição dessa coluna (veja a Figura 8.1).

A forma sugerida de usar esse gráfico é encontrar valores próximos de 1. O rótulo no eixo X é o atributo que faz a predição do rótulo do eixo Y. Se um atributo faz a predição de outro, podemos remover o atributo previsto (o atributo no eixo Y). Em nosso exemplo, fare faz a predição de pclass, sibsp, parch e embarked_Q. Poderemos manter fare e remover os demais atributos, obtendo um desempenho semelhante:

```
>>> rfpimp.plot_dependence_heatmap(
... rfpimp.feature_dependence_matrix(X_train),
... value_fontsize=12,
... label_fontsize=14,
... figsize=(8, 8),sn
... )
>>> fig = plt.gcf()
>>> fig.savefig(
... "images/mlpr_0801.png",
... dpi=300,
... bbox_inches="tight",
... )
```

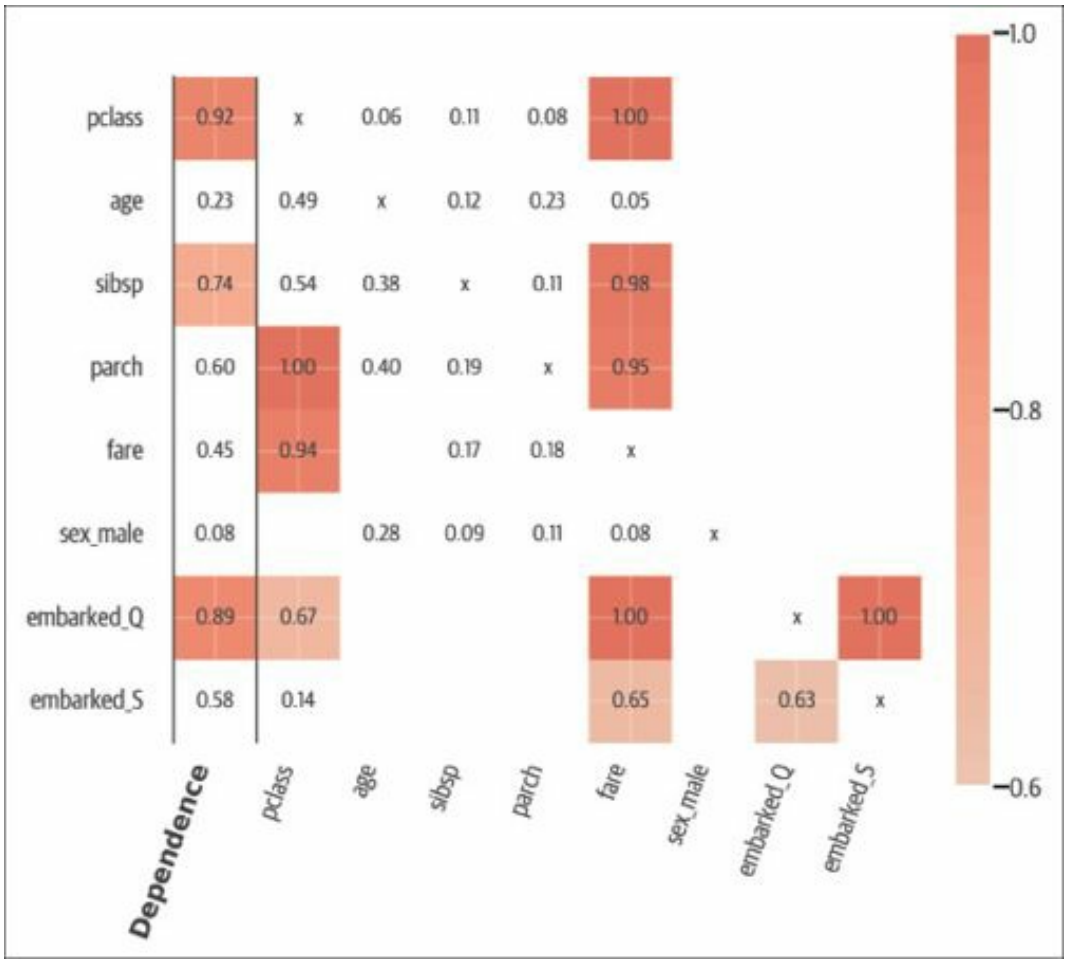


Figura 8.1 – Heat map das dependências. pclass, sibsp, parch e embarked_Q podem ser previstos a partir de fare, portanto podemos removê-los.

Eis o código que mostra que teremos uma pontuação parecida se removermos essas colunas:

```
>>> cols_to_remove = [
... "pclass",
... "sibsp",
... "parch",
... "embarked_Q",
... ]
>>> rf3 = RandomForestClassifier(random_state=42)
>>> rf3.fit(
... X_train[
... [
... c
... for c in X_train.columns
... if c not in cols_to_remove
... ]
```

```
... ],
... y_train,
... )
>>> rf3.score(
... X_test[
... [
... c
... for c in X_train.columns
... if c not in cols_to_remove
... ]
... ],
... y_test,
... )
0.7684478371501272

>>> rf4 = RandomForestClassifier(random_state=42)
>>> rf4.fit(X_train, y_train)
>>> rf4.score(X_test, y_test)
0.7659033078880407
```

Regressão lasso

Se você usar a regressão lasso, poderá definir um parâmetro alpha que atuará como um parâmetro de regularização. À medida que seu valor aumentar, menor peso será dado aos atributos que são menos importantes. No código a seguir, usamos o modelo LassoLarsCV para iterar por diversos valores de alpha e monitorar os coeficientes dos atributos (veja a Figura 8.2).

```
>>> from sklearn import linear_model
>>> model = linear_model.LassoLarsCV(
... cv=10, max_n_alphas=10
... ).fit(X_train, y_train)
>>> fig, ax = plt.subplots(figsize=(12, 8))
>>> cm = iter(
... plt.get_cmap("tab20")(
... np.linspace(0, 1, X.shape[1])
... )
... )
>>> for i in range(X.shape[1]):
... c = next(cm)
... ax.plot(
... model.alphas_,
... model.coef_path_.T[:, i],
... c=c,
... alpha=0.8,
```

```
... label=X.columns[i],
... )
>>> ax.axvline(
... model.alpha_,
... linestyle="-",
... c="k",
... label="alphaCV",
... )
>>> plt.ylabel("Regression Coefficients")
>>> ax.legend(X.columns, bbox_to_anchor=(1, 1))
>>> plt.xlabel("alpha")
>>> plt.title(
... "Regression Coefficients Progression for Lasso Paths"
... )
>>> fig.savefig(
... "images/mlpr_0802.png",
... dpi=300,
... bbox_inches="tight",
... )
```

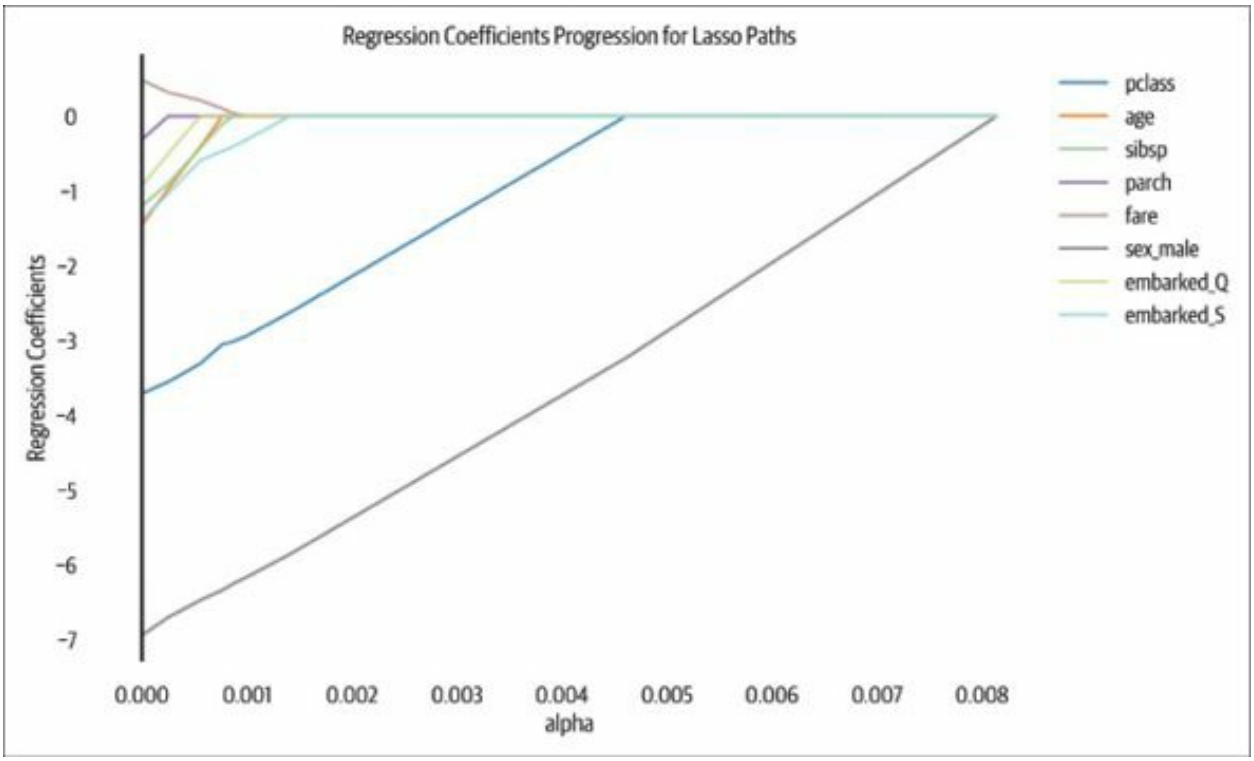


Figura 8.2 – Coeficientes de atributos à medida que alpha varia durante uma regressão lasso.

Eliminação recursiva de atributos

A eliminação recursiva de atributos removerá os atributos mais fracos, e então fará a adequação de um modelo (veja a Figura 8.3). Fazemos isso passando um modelo do scikit-learn com um atributo .coef_ ou .feature_importances_:

```
>>> from yellowbrick.features import RFECV
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rfe = RFECV(
... ensemble.RandomForestClassifier(
... n_estimators=100
... ),
... cv=5,
... )
>>> rfe.fit(X, y)

>>> rfe.rfe_estimator_.ranking_
array([1, 1, 2, 3, 1, 1, 5, 4])

>>> rfe.rfe_estimator_.n_features_
4
>>> rfe.rfe_estimator_.support_
array([ True, True, False, False, True,
        True, False, False])

>>> rfe.poof()
>>> fig.savefig("images/mlpr_0803.png", dpi=300)
```

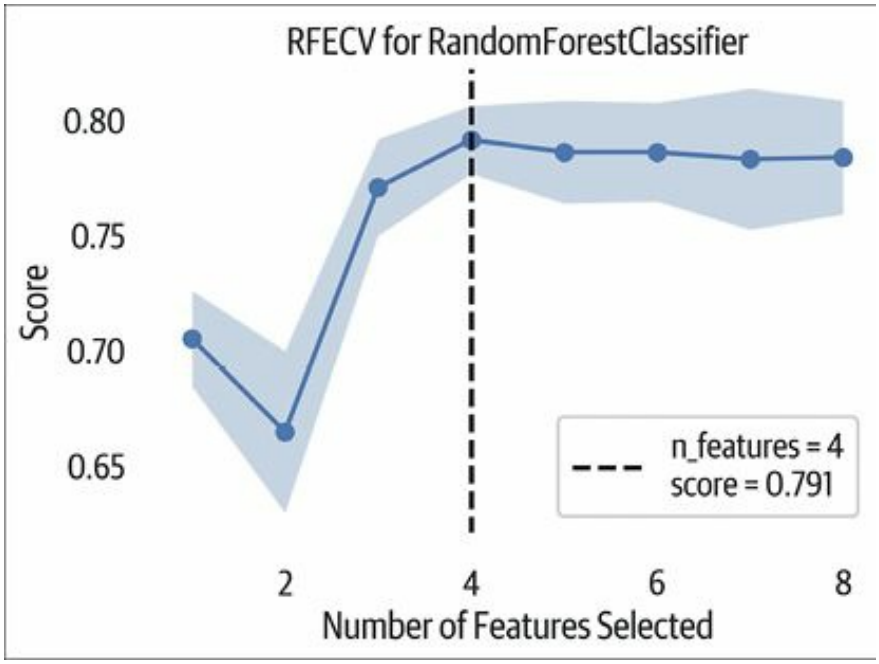


Figura 8.3 – Eliminação recursiva de atributos.

Usaremos a eliminação recursiva de atributos para encontrar os dez atributos mais importantes. (Nesse conjunto de dados agregados, vimos que há vazamento da coluna de sobrevivência!)

```
>>> from sklearn.feature_selection import RFE
>>> model = ensemble.RandomForestClassifier(
... n_estimators=100
... )
>>> rfe = RFE(model, 4)
>>> rfe.fit(X, y)
>>> agg_X.columns[rfe.support_]
Index(['pclass', 'age', 'fare', 'sex_male'], dtype='object')
```

Informações mútuas

O sklearn disponibiliza testes não paramétricos que usarão os k vizinhos mais próximos (k-nearest neighbor) para determinar as *informações mútuas* entre os atributos e o objetivo. As informações mútuas quantificam o volume de informações obtidas observando outra variável. O valor pode ser maior ou igual a zero. Se o valor for igual a zero, é sinal de que não há nenhuma relação entre os atributos e o alvo (veja a Figura 8.4). Esse número não é limitado e representa o número de *porções* compartilhadas entre eles:

```
>>> from sklearn import feature_selection

>>> mic = feature_selection.mutual_info_classif(
... X, y
... )
>>> fig, ax = plt.subplots(figsize=(10, 8))
>>> (
... pd.DataFrame(
... {"feature": X.columns, "vimp": mic}
... )
... .set_index("feature")
... .plot.barh(ax=ax)
... )
>>> fig.savefig("images/mlpr_0804.png")
```

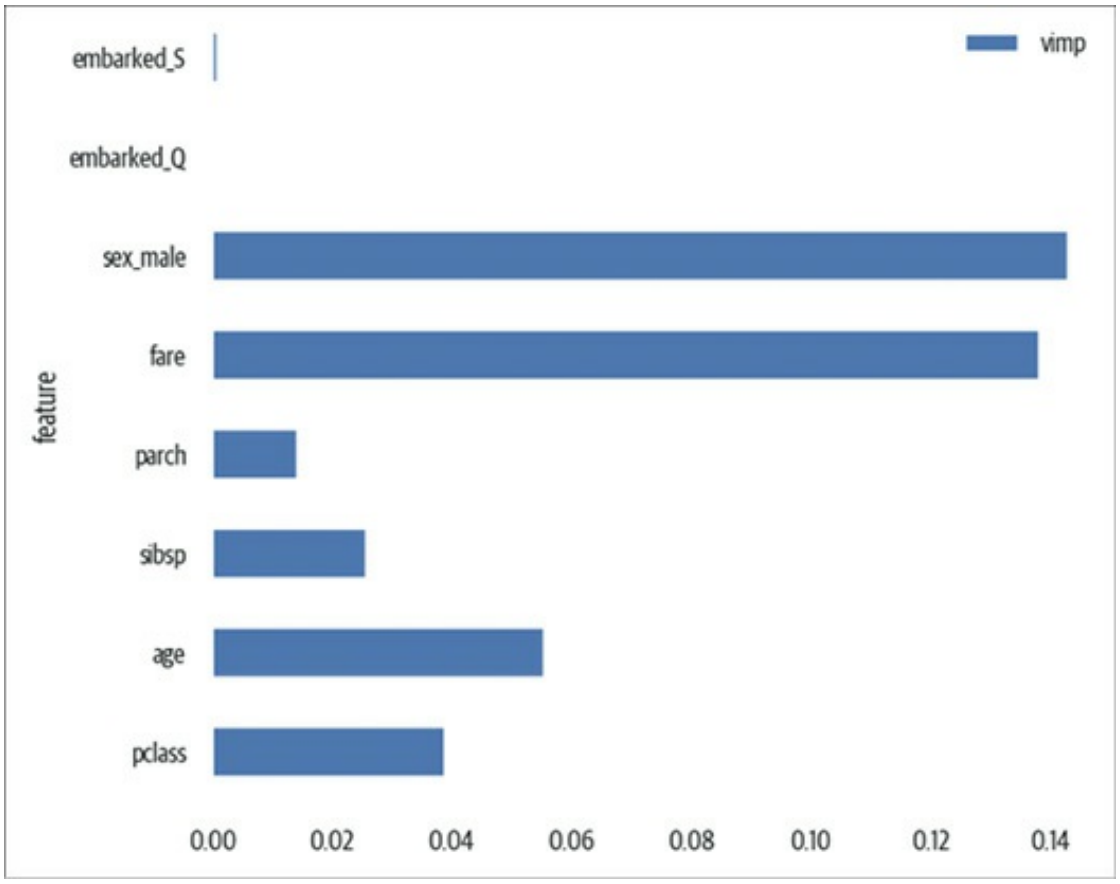


Figura 8.4 – Diagrama de informações mútuas.

Principal Component Analysis

Outra opção para seleção de atributos é executar uma PCA (Principal Component Analysis, ou Análise de Componentes Principais). Uma vez que tiver os componentes principais, analise os atributos que contribuem mais com eles. Esses são os atributos que têm maior variação. Note que esse é um algoritmo não supervisionado, e não leva `y` em consideração.

Veja a seção “PCA”, a qual apresenta mais detalhes.

Importância dos atributos

A maioria dos modelos baseados em árvore oferece acesso a um atributo `.feature_importances_` após o treinamento. Uma importância maior em geral implica que haverá um erro maior se o atributo for removido do modelo. Consulte os capítulos que explicam os diversos modelos em árvore para ver mais detalhes.

CAPÍTULO 9

Classes desbalanceadas

Se você estiver classificando dados e as classes não estiverem relativamente balanceadas quanto ao tamanho, a distorção em direção às classes mais populares poderão transparecer em seu modelo. Por exemplo, se você tiver 1 caso positivo e 99 casos negativos, poderá obter 99% de exatidão simplesmente classificando tudo como negativo. Há várias opções para lidar com *classes desbalanceadas* (imbalanced classes).

Use uma métrica diferente

Uma dica é usar uma medida que não seja a exatidão (accuracy) para calibrar os modelos (o AUC é uma boa opção). Precisão (precision) e recall também são ótimas opções quando os tamanhos dos alvos (targets) forem diferentes. No entanto, há outras opções a serem consideradas também.

Algoritmos baseados em árvores e ensembles

Modelos baseados em árvore poderão ter um melhor desempenho conforme a distribuição da classe menor. Se os dados tiverem a tendência de estar agrupados, poderão ser mais facilmente classificados.

Os métodos de ensemble podem ainda ajudar a extrair as classes minoritárias. Bagging e boosting são opções encontradas em modelos de árvore como florestas aleatórias (random forests) e o XGBoost (Gradient Boosting).

Modelos de penalização

Muitos modelos de classificação do scikit-learn aceitam o parâmetro `class_weight`. Defini-lo com `'balanced'` tentará regularizar as classes minoritárias e incentivará o modelo a classificá-las corretamente. Como alternativa, você pode fazer uma busca em grade (grid search) e especificar as opções de peso, passando um dicionário que mapeie classes a pesos (dando pesos maiores às

classes menores).

A biblioteca XGBoost (*https://xgboost.readthedocs.io*) tem um parâmetro max_delta_step, que pode ser definido com um valor entre 1 e 10 para deixar o passo de atualização mais conservador. Há também um parâmetro scale_pos_weight que define a razão entre amostras negativas e positivas (para classes binárias). Além do mais, eval_metric deve ser definido com 'auc', em vez de usar o valor default igual a 'error' para a classificação.

O modelo KNN tem um parâmetro weights que pode gerar distorção em vizinhos mais próximos. Se as amostras da classe minoritária estiverem próximas, definir esse parâmetro com 'distance' poderá melhorar o desempenho.

Upsampling da minoria

Você pode fazer um upsampling da classe minoritária de algumas maneiras. Eis uma implementação com o sklearn:

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_upsample = resample(
... surv_df,
... replace=True,
... n_samples=len(death_df),
... random_state=42,
... )
>>> df2 = pd.concat([death_df, df_upsample])

>>> df2.survived.value_counts()
1 809
0 809
Name: survived, dtype: int64
```

A biblioteca imbalanced-learn também pode ser usada para amostrar aleatoriamente com substituição:

```
>>> from imblearn.over_sampling import (
... RandomOverSampler,
... )
>>> ros = RandomOverSampler(random_state=42)
>>> X_ros, y_ros = ros.fit_sample(X, y)
>>> pd.Series(y_ros).value_counts()
1 809
```

```
0 809
dtype: int64
```

Gerando dados de minorias

A biblioteca imbalanced-learn também pode gerar novas amostras das classes minoritárias com os algoritmos para amostragens como SMOTE (Synthetic Minority Oversampling Technique) e ADASYN (Adaptive Synthetic). O SMOTE funciona selecionando um de seus k vizinhos mais próximos, conectando uma linha a um deles e selecionando um ponto nessa linha. O ADASYN é semelhante ao SMOTE, mas gera mais amostras a partir daquelas cujo aprendizado é mais difícil. As classes em imbanced-learn se chamam over_sampling.SMOTE e over_sampling.ADASYN.

Downsampling da maioria

Outro método para balancear classes é fazer o downsampling das classes majoritárias. Eis um exemplo com o sklearn:

```
>>> from sklearn.utils import resample
>>> mask = df.survived == 1
>>> surv_df = df[mask]
>>> death_df = df[~mask]
>>> df_downsample = resample(
... death_df,
... replace=False,
... n_samples=len(surv_df),
... random_state=42,
... )
>>> df3 = pd.concat([surv_df, df_downsample])

>>> df3.survived.value_counts()
1 500
0 500
Name: survived, dtype: int64
```

DICA

Não use substituição quando fizer um downsampling.

A biblioteca imbalanced-learn também implementa diversos algoritmos de downsampling:

ClusterCentroids

Essa classe utiliza k-means (k-médias) para sintetizar dados com os centroides.

RandomUnderSampler

Essa classe seleciona amostras aleatoriamente.

NearMiss

Essa classe faz um downsampling removendo amostras que estão próximas umas das outras.

TomekLink

Essa classe reduz as amostras removendo aquelas que estão mais próximas entre si.

EditedNearestNeighbours

Essa classe remove amostras que tenham vizinhos que não estão na classe majoritária ou que estejam todos na mesma classe.

RepeatedNearestNeighbours

Essa classe chama EditedNearestNeighbours repetidamente.

AllKNN

Essa classe é semelhante, mas aumenta o número de vizinhos mais próximos durante as iterações do downsampling.

CondensedNearestNeighbour

Essa classe escolhe uma amostra da classe para downsampling e, em seguida, itera pelas outras amostras da classe; se KNN não fizer uma classificação incorreta, essa amostra será adicionada.

OneSidedSelection

Essa classe remove amostras com ruído.

NeighbourhoodCleaningRule

Essa classe usa os resultados de EditedNearestNeighbours, aplicando aí o KNN.

InstanceHardnessThreshold

Essa classe faz o treinamento de um modelo e então remove as amostras com baixas probabilidades.

Todas essas classes aceitam o método .fit_sample.

Upsampling e depois downsampling

A biblioteca imbalanced-learn implementa SMOTEENN e SMOTETomek, que fazem um upsampling e depois aplicam um downsampling para limpar os dados.

CAPÍTULO 10

Classificação

A classificação é um método de *aprendizagem supervisionada* (supervised learning) para atribuir um rótulo a uma amostra com base nos atributos. A aprendizagem supervisionada implica que temos rótulos para classificação ou números para regressão, os quais o algoritmo deve aprender.

Veremos vários modelos de classificação neste capítulo. O sklearn implementa diversos modelos úteis e comuns. Veremos também alguns modelos que não estão no sklearn, porém implementam a sua interface. Por terem a mesma interface, é fácil testar diferentes famílias de modelos e verificar como é o seu desempenho.

No sklearn, criamos uma instância de modelo e chamamos aí o método .fit com os dados e rótulos de treinamento. Podemos então chamar o método .predict (ou os métodos .predict_proba ou .predict_log_proba) com o modelo após a adequação. Para avaliar o modelo, usamos .score com os dados e rótulos de teste.

Em geral, o maior desafio é organizar os dados em um formato apropriado para o sklearn. Os dados (X) devem estar em um array (m por n) numpy (ou em um DataFrame pandas) com m linhas de dados de amostras, cada uma com n atributos (colunas). O rótulo (y) é um vetor (ou uma séria Pandas) de tamanho m com um valor (classe) para cada amostra.

O método .score devolve a precisão (accuracy) média que, por si só, poderia não ser suficiente para avaliar um classificador. Apresentaremos também outras métricas para avaliação.

Veremos vários modelos e discutiremos sua eficiência, as técnicas de pré-processamento que eles exigem, como evitar a superadequação (overfitting) e se o modelo aceita uma interpretação intuitiva dos resultados.

Os métodos genéricos que os modelos do sklearn implementam são:

fit(X, y[, sample_weight])

Faz a adequação de um modelo.

predict(X)

Faz a predição de classes.

predict_log_proba(X)

Faz a predição do logaritmo das probabilidades.

predict_proba(X)

Faz a predição de probabilidade.

score(X, y[, sample_weight])

Obtém a precisão (accuracy).

Regressão logística

A regressão logística estima probabilidades usando uma função logística. (Tome cuidado: apesar de ter regressão no nome, ela é usada para classificação.) Esse tem sido o modelo de classificação padrão para a maioria das ciências.

A seguir, apresentamos algumas das características que incluiremos em cada modelo:

Eficiência na execução

Pode usar `n_jobs` se não estiver usando o solucionador `'liblinear'`.

Pré-processamento dos dados

Se `solver` estiver definido com `'sag'` ou `'saga'`, padronize para que a convergência funcione. É capaz de lidar com entradas esparsas.

Para evitar uma superadequação

O parâmetro `C` controla a regularização. (Valores menores de `C` significam mais regularização, enquanto valores maiores significam menos.) É possível especificar `penalty` com `'l1'` ou `'l2'` (o default).

Interpretação dos resultados

O atributo `.coef` do modelo após a adequação mostra os coeficientes da função de decisão. Uma mudança de x em uma unidade modifica o log odds ratio (logaritmo da razão de chances) de acordo com o coeficiente. O atributo `.intercept_` é o inverso dos log odds da condição de base.

Eis um exemplo de uso desse modelo:

```
>>> from sklearn.linear_model import (
... LogisticRegression,
... )
>>> lr = LogisticRegression(random_state=42)
>>> lr.fit(X_train, y_train)
LogisticRegression(C=1.0, class_weight=None,
    dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100,
    multi_class='ovr', n_jobs=1, penalty='l2',
    random_state=42, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
>>> lr.score(X_test, y_test)
0.8040712468193384

>>> lr.predict(X.iloc[[0]])
array([1])
>>> lr.predict_proba(X.iloc[[0]])
array([[0.08698937, 0.91301063]])
>>> lr.predict_log_proba(X.iloc[[0]])
array([[-2.4419694 , -0.09100775]])
>>> lr.decision_function(X.iloc[[0]])
array([2.35096164])
```

Parâmetros da instância:

penalty='l2'

Norma de penalização, 'l1' ou 'l2'.

dual=False

Usa formulação dual (somente com 'l2' e 'liblinear').

C=1.0

Ponto flutuante positivo. Inverso da força de regularização. Um valor menor significa uma regularização mais forte.

fit_intercept=True

Adiciona bias (viés) à função de decisão.

intercept_scaling=1

Se fit_intercept e 'liblinear', escala o intercepto.

max_iter=100

Número máximo de iterações.

multi_class='ovr'

Usa one-versus-rest (um contra todos) para cada classe ou, para 'multinomial', treina uma classe.

class_weight=None

Dicionário ou 'balanced'.

solver='liblinear'

'liblinear' é apropriado para poucos dados. 'newton-cg', 'sag', 'saga' e 'lbfgs' devem ser usados para dados multiclasse. 'liblinear' e 'saga' só funcionam com penalidade 'l1'. Os demais funcionam com 'l2'.

tol=0.0001

Tolerância para parada.

verbose=0

Verboso (se int diferente de zero).

warm_start=False

Se for True, lembra a adequação anterior.

njobs=1

Número de CPUS a ser usado. -1 são todas as CPUs. Funciona somente com multi_class='ovr' e solver diferente de 'liblinear'.

Atributos após a adequação:

coef_

Coeficientes da função de decisão.

intercept_

Intercepto da função de decisão.

n_iter_

Número de iterações.

O intercepto é o log odds (logaritmo das chances) da condição de base. Podemos convertê-lo de volta em uma precisão percentual (proporção):

```
>>> lr.intercept_
array([-0.62386001])
```

Usando a função logit inversa, vemos que o valor de base para a sobrevivência é de 34%:

```
>>> def inv_logit(p):
... return np.exp(p) / (1 + np.exp(p))

>>> inv_logit(lr.intercept_)
array([0.34890406])
```

É possível inspecionar os coeficientes. O logit inverso dos coeficientes nos dá a proporção dos casos positivos. Nesse caso, se fare (preço da passagem) for maior, há mais chances de sobrevivência. Se sex (sexo) for male (masculino), haverá menos chances de sobrevivência:

```
>>> cols = X.columns
>>> for col, val in sorted(
... zip(cols, lr.coef_[0]),
... key=lambda x: x[1],
... reverse=True,
... ):
... print(
... f"{col:10}{val:10.3f} {inv_logit(val):10.3f}"
... )
fare 0.104 0.526
parch -0.062 0.485
sibsp -0.274 0.432
age -0.296 0.427
embarked_Q -0.504 0.377
embarked_S -0.507 0.376
pclass -0.740 0.323
sex_male -2.400 0.083
```

O Yellowbrick também permite visualizar os coeficientes. Esse visualizador tem um parâmetro relative=True que faz com que o maior valor seja 100 (ou -100), e os demais sejam porcentagens desse valor (veja a Figura 10.1):

```
>>> from yellowbrick.features.importances import (
... FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(lr)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1001.png", dpi=300)
```

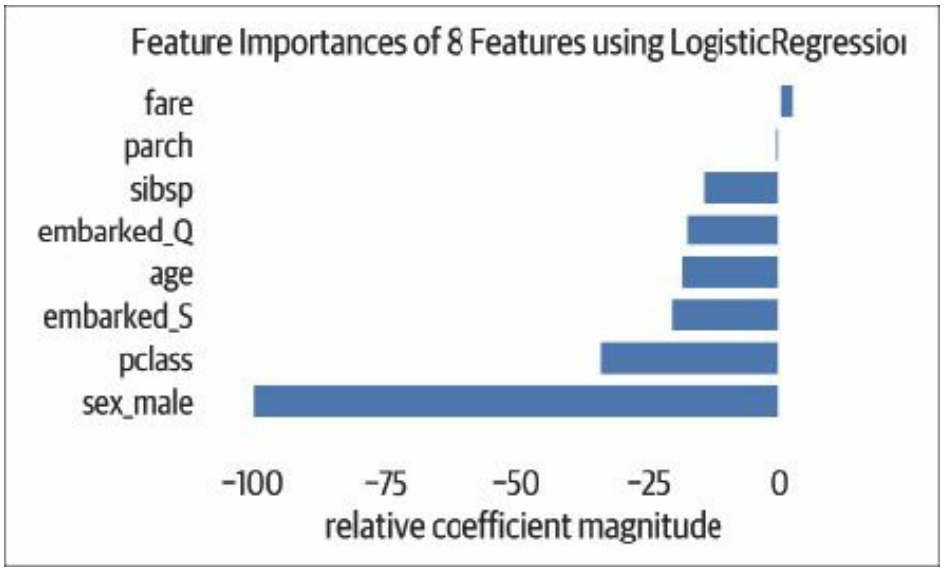


Figura 10.1 – Importância dos atributos (relativa ao maior coeficiente de regressão absoluto).

Naive Bayes

O Naive Bayes é um classificador probabilístico que pressupõe uma independência entre os atributos dos dados. É popular para aplicações de classificação de textos, por exemplo, para identificação de spams. Uma vantagem desse modelo é que, por supor uma independência entre os atributos, ele é capaz de fazer o treinamento de um modelo com um número pequeno de amostras. (Uma desvantagem é que o modelo não conseguirá capturar as interações entre os atributos.) Esse modelo simples também pode trabalhar com dados que tenham vários atributos. Desse modo, serve como um bom modelo de base.

Há três classes no sklearn: GaussianNB, MultinomialNB e BernoulliNB. A primeira supõe uma distribuição gaussiana (atributos contínuos com uma distribuição normal), a segunda é para contadores de ocorrência discretos, e a terceira, para atributos booleanos discretos.

Esse modelo tem as seguintes propriedades:

Eficiência na execução

Treinamento O(Nd), em que N é o número de exemplos para treinamento, e d é a dimensionalidade. Testes O(cd), em que c é o número de classes.

Pré-processamento dos dados

É pressuposto que os dados são independentes. O desempenho deverá ser melhor após a remoção de colunas colineares. Para dados numéricos contínuos, talvez seja uma boa ideia separar os dados em bins. A classe gaussiana implica uma distribuição normal, e pode ser necessário ter de transformar os dados a fim de convertê-los em uma distribuição desse tipo.

Para evitar uma superadequação

Exibe alto bias e baixa variância (os emsembles não reduzirão a variância).

Interpretação dos resultados

A porcentagem é a probabilidade de uma amostra pertencer a uma classe com base em priors (conhecimento prévio).

Eis um exemplo que utiliza esse modelo:

```
>>> from sklearn.naive_bayes import GaussianNB
>>> nb = GaussianNB()
>>> nb.fit(X_train, y_train)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> nb.score(X_test, y_test)
0.7837150127226463

>>> nb.predict(X.iloc[[0]])
array([1])
>>> nb.predict_proba(X.iloc[[0]])
array([[2.17472227e-08, 9.99999978e-01]])
>>> nb.predict_log_proba(X.iloc[[0]])
array([[-1.76437798e+01, -2.17472227e-08]])
```

Parâmetros da instância:

priors=None

Probabilidades prévias (prior) das classes.

var_smoothing=1e-9

Adicionado à variância para cálculos estáveis.

Atributos após a adequação:

class_prior_

Probabilidades das classes.

class_count_

Contadores de classes.

theta_
: Média de cada coluna por classe.

sigma_
: Variância de cada coluna por classe.

epsilon_
: Valor a ser somado para cada variância.

DICA

Esses modelos são suscetíveis ao *problema da probabilidade zero*. Se você tentar classificar uma nova amostra que não tenha dados de treinamento, ela terá uma probabilidade zero. Uma solução é usar a *suavização de Laplace* (Laplace smoothing). O sklearn controla isso com o parâmetro alpha, cujo default é 1, e permite uma suavização nos modelos MultinomialNB e BernoulliNB.

Máquina de vetores suporte

Uma SVM (Support Vector Machine, ou Máquina de Vetores Suporte) é um algoritmo que tenta fazer a adequação de uma linha (ou plano ou hiperplano) entre as diferentes classes de modo a maximizar a distância da linha até os pontos das classes. Dessa maneira, ela tenta encontrar uma separação robusta entre as classes. Os *vetores suporte* (support vectors) são os pontos da fronteira do hiperplano divisor.

NOTA

Há algumas implementações distintas de SVM no sklearn. SVC encapsula a biblioteca libsvm, enquanto LinearSVC encapsula liblinear.

Há também o linear_model.SGDClassifier, que implementa a SVM quando o parâmetro loss default é usado. Este capítulo descreverá a primeira implementação.

Em geral, a SVM tem um bom desempenho e oferece suporte para espaços lineares e não lineares usando um *truque de kernel* (kernel trick). O truque de kernel é a ideia de que podemos criar uma fronteira de decisão em uma nova dimensão minimizando uma fórmula que seja mais fácil de calcular, em comparação a realmente mapear os pontos para a nova dimensão. O kernel

default é o Radial Basis Function, ou Função de Base Radial ('rbf'), controlado pelo padrão gamma, o qual é capaz de mapear um espaço de entrada em um espaço com mais dimensões.

As SVMs têm as seguintes propriedades:

Eficiência na execução

A implementação do scikit-learn é $O(n^4)$, portanto pode ser difícil escalar para tamanhos maiores. Usar um kernel linear ou o modelo LinearSVC pode melhorar o desempenho da execução, talvez à custa da precisão. Aumentar o valor do parâmetro cache_size pode reduzir a ordem para $O(n^3)$.

Pré-processamento dos dados

O algoritmo não é invariante à escala. Padronizar os dados é extremamente recomendável.

Para evitar uma superadequação

O parâmetro C (parâmetro de penalidade) controla a regularização. Um valor menor permite ter uma margem menor no hiperplano. Um valor maior para gamma tenderá a uma superadequação nos dados de treinamento. O modelo LinearSVC aceita parâmetros loss e penalty para regularização.

Interpretação dos resultados

Inspecione .support_vectors_, embora possam ser difíceis de explicar. Com kernels lineares, você poderá inspecionar .coef_.

Eis um exemplo que usa a implementação de SVM do scikit-learn:

```
>>> from sklearn.svm import SVC
>>> svc = SVC(random_state=42, probability=True)
>>> svc.fit(X_train, y_train)
SVC(C=1.0, cache_size=200, class_weight=None,
  coef0=0.0, decision_function_shape='ovr',
  degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=True, random_state=42,
  shrinking=True, tol=0.001, verbose=False)
>>> svc.score(X_test, y_test)
0.8015267175572519

>>> svc.predict(X.iloc[[0]])
array([1])
>>> svc.predict_proba(X.iloc[[0]])
array([[0.15344656, 0.84655344]])
```

```
>>> svc.predict_log_proba(X.iloc[[0]])
array([[-1.87440289, -0.16658195]])
```

Para obter probabilidades, utilize probability=True, que deixará a adequação do modelo mais lenta.

Isso é parecido com um perceptron, mas encontrará a margem máxima. Se os dados não forem linearmente separáveis, ele minimizará o erro. Como alternativa, um kernel diferente pode ser usado.

Parâmetros da instância:

C=1.0

Parâmetro de penalidade. Quanto menor o valor, mais estreita será a fronteira de decisão (mais superadequação).

cache_size=200

Tamanho do cache (MB). Aumentar esse valor pode melhorar o tempo de treinamento em conjuntos grandes de dados.

class_weight=None

Dicionário ou 'balanced'. Use um dicionário para definir C para cada classe.

coef0=0.0

Termo independente para kernels polinomial e sigmoide.

decision_function_shape='ovr'

Use one-versus-rest (um contra todos) ('ovr') ou one-versus-one (um contra um).

degree=3

Grau para kernel polinomial.

gamma='auto'

Coeficiente do kernel. Pode ser um número, 'scale' (default em 0.22, 1 / (num atributos * X.std())) ou 'auto' (default anterior, 1 / num atributos). Um valor menor resulta na superadequação dos dados de treinamento.

kernel='rbf'

Tipo de kernel: 'linear', 'poly', 'rbf' (default), 'sigmoid', 'precomputed' ou uma função.

max_iter=-1

Número máximo de iterações para o solucionador (solver). -1 indica que não há limites.

probability=False
  Ativa a estimação de probabilidades. Deixa o treinamento mais lento.

random_state=None
  Semente (seed) aleatória.

shrinking=True
  Usa heurística de shrinking (encolhimento).

tol=0.001
  Tolerância para parada.

verbose=False
  Verbosidade.

Atributos após a adequação:

support_
  Índices dos vetores suporte.

support_vectors_
  Vetores suporte.

n_support_vectors_
  Número de vetores suporte por classe.

coef_
  Coeficientes para kernel (linear).

K vizinhos mais próximos

O algoritmo KNN (K-Nearest Neighbor, ou K Vizinhos Mais Próximos) faz a classificação com base na distância até algumas amostras (k) de treinamento. A família de algoritmos é chamada de aprendizado *baseado em instâncias* (instance-based learning), pois não há parâmetros para aprender. O modelo pressupõe que a distância é suficiente para fazer uma inferência; afora isso, nenhuma pressuposição é feita sobre os dados subjacentes ou suas distribuições.

A parte complicada é selecionar o valor apropriado de k. Além disso, a maldição da dimensionalidade pode atrapalhar as métricas de distância, pois haverá pouca diferença entre os vizinhos mais próximos e mais distantes no caso de mais dimensões.

Os modelos que usam vizinhos mais próximos têm as seguintes propriedades:

Eficiência na execução

Treinamento O(1), mas precisa armazenar dados. Testes O(Nd), em que N é o número de exemplos de treinamento e d é a dimensionalidade.

Pré-processamento dos dados

Sim, cálculos baseados em distância têm melhor desempenho se houver padronização.

Para evitar uma superadequação

Eleve n_neighbors. Mude p para métrica L1 ou L2.

Interpretação dos resultados

Interpreta os k vizinhos mais próximos para a amostra (usando o método .kneighbors). Esses vizinhos (se você puder explicá-los) explicarão o seu resultado.

Eis um exemplo de uso do modelo:

```
>>> from sklearn.neighbors import (
... KNeighborsClassifier,
... )
>>> knc = KNeighborsClassifier()
>>> knc.fit(X_train, y_train)
KNeighborsClassifier(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')
>>> knc.score(X_test, y_test)
0.7837150127226463

>>> knc.predict(X.iloc[[0]])
array([1])

>>> knc.predict_proba(X.iloc[[0]])
array([[0., 1.]])
```

Atributos:

algorithm='auto'

Pode ser 'brute', 'ball_tree' ou 'kd_tree'.

leaf_size=30

Usado para algoritmos baseados em árvore.

metric='minkowski'

Métrica de distância.

metric_params=None

Dicionário adicional de parâmetros para função de métrica personalizada.

n_jobs=1

Número de CPUs.

n_neighbors=5

Número de vizinhos.

p=2

Parâmetro de potência de Minkowski: 1 = manhattan (L1); 2 = euclidiana (L2).

weights='uniform'

Pode ser 'distance', caso em que pontos mais próximos terão mais influência.

As métricas de distância incluem: 'euclidean', 'manhattan', 'chebyshev', 'minkowski', 'wminkowski', 'seuclidean', 'mahalanobis', 'haversine', 'hamming', 'canberra', 'braycurtis', 'jaccard', 'matching', 'dice', 'rogerstanimoto', 'russellrao', 'sokalmichener', 'sokalsneath' ou um callable (definido pelo usuário).

NOTA

Se k for um número par e os vizinhos forem separados, o resultado dependerá da ordem dos dados de treinamento.

Árvore de decisão

Uma árvore de decisão é como ir a um médico que faz uma série de perguntas a fim de determinar a causa de seus sintomas. Podemos usar um processo para criar uma árvore de decisão e ter uma série de perguntas para prever uma classe alvo. As vantagens desse modelo incluem suporte para dados não numéricos (em algumas implementações), pouca preparação dos dados (não há necessidade de escalar), suporte para lidar com relacionamentos não lineares, a importância dos atributos é revelada e é fácil de explicar.

O algoritmo padrão usado para a criação se chama CART (Classification And

Regression Tree, Árvore de Classificação e Regressão). Ele usa a impureza de Gini ou medida de índices para tomada de decisões. Isso é feito percorrendo os atributos em um laço e encontrando o valor que forneça a menor probabilidade de erro de classificação.

DICA

Os valores default resultarão em uma árvore muito grande (entenda-se, com superadequação). Utilize um método como max_depth e validação cruzada para controlar isso.

As árvores de decisão têm as seguintes propriedades:

Eficiência na execução

Para a criação, percorre cada um dos m atributos e ordena todas as n amostras, O(mn log n). Para predição, você percorrerá a árvore, O(altura).

Pré-processamento dos dados

Não é necessário escalar. É preciso se livrar dos valores ausentes e convertê-los em dados numéricos.

Para evitar uma superadequação

Defina max_depth com um número menor e aumente min_impurity_decrease.

Interpretação dos resultados

É possível percorrer a árvore de opções. Por haver passos, uma árvore é ruim para lidar com relacionamentos lineares (uma pequena mudança em um número pode levar a um caminho diferente). A árvore também é extremamente dependente dos dados de treinamento. Uma pequena mudança pode modificar a árvore toda.

Eis um exemplo que utiliza a biblioteca scikit-learn:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier(
... random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
DecisionTreeClassifier(class_weight=None,
  criterion='gini', max_depth=None,
  max_features=None, max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,
```

```
  min_samples_leaf=1, min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dt.score(X_test, y_test)
0.8142493638676844

>>> dt.predict(X.iloc[[0]])
array([1])
>>> dt.predict_proba(X.iloc[[0]])
array([[0.02040816, 0.97959184]])
>>> dt.predict_log_proba(X.iloc[[0]])
array([[-3.8918203 , -0.02061929]])
```

Parâmetros da instância:

class_weight=None

Pesos das classes em um dicionário. 'balanced' definirá valores na proporção inversa das frequências das classes. O default é 1 para cada classe. Para várias classes, é necessário ter uma lista de dicionários, OVR (one-versus-rest, ou um contra todos) para cada classe.

criterion='gini'

Função de separação, 'gini' ou 'entropy'.

max_depth=None

Profundidade da árvore. O default será construir a árvore até que as folhas contenham menos que min_samples_split.

max_features=None

Número de atributos a serem analisados para separação. O default são todos.

max_leaf_nodes=None

Limita o número de folhas. O default é sem limites.

min_impurity_decrease=0.0

Separa um nó se a separação diminuir a impureza, por um valor maior ou igual a este.

min_impurity_split=None

Obsoleto.

min_samples_leaf=1

Número mínimo de amostras em cada folha.

min_samples_split=2

Número mínimo de amostras exigido para separar um nó.

min_weight_fraction_leaf=0.0

Soma mínima de pesos exigida para nós do tipo folha.

presort=False

Pode agilizar o treinamento com um conjunto de dados menor ou com uma profundidade restrita se definido com `True`.

random_state=None

Semente (seed) aleatória.

splitter='best'

Use `'random'` ou `'best'`.

Atributos após a adequação:

classes_

Rótulos das classes.

feature_importances_

Array de importância de Gini.

n_classes_

Número de classes.

n_features_

Número de atributos.

tree_

Objeto árvore subjacente.

Visualize a árvore com o código a seguir (veja a Figura 10.2):

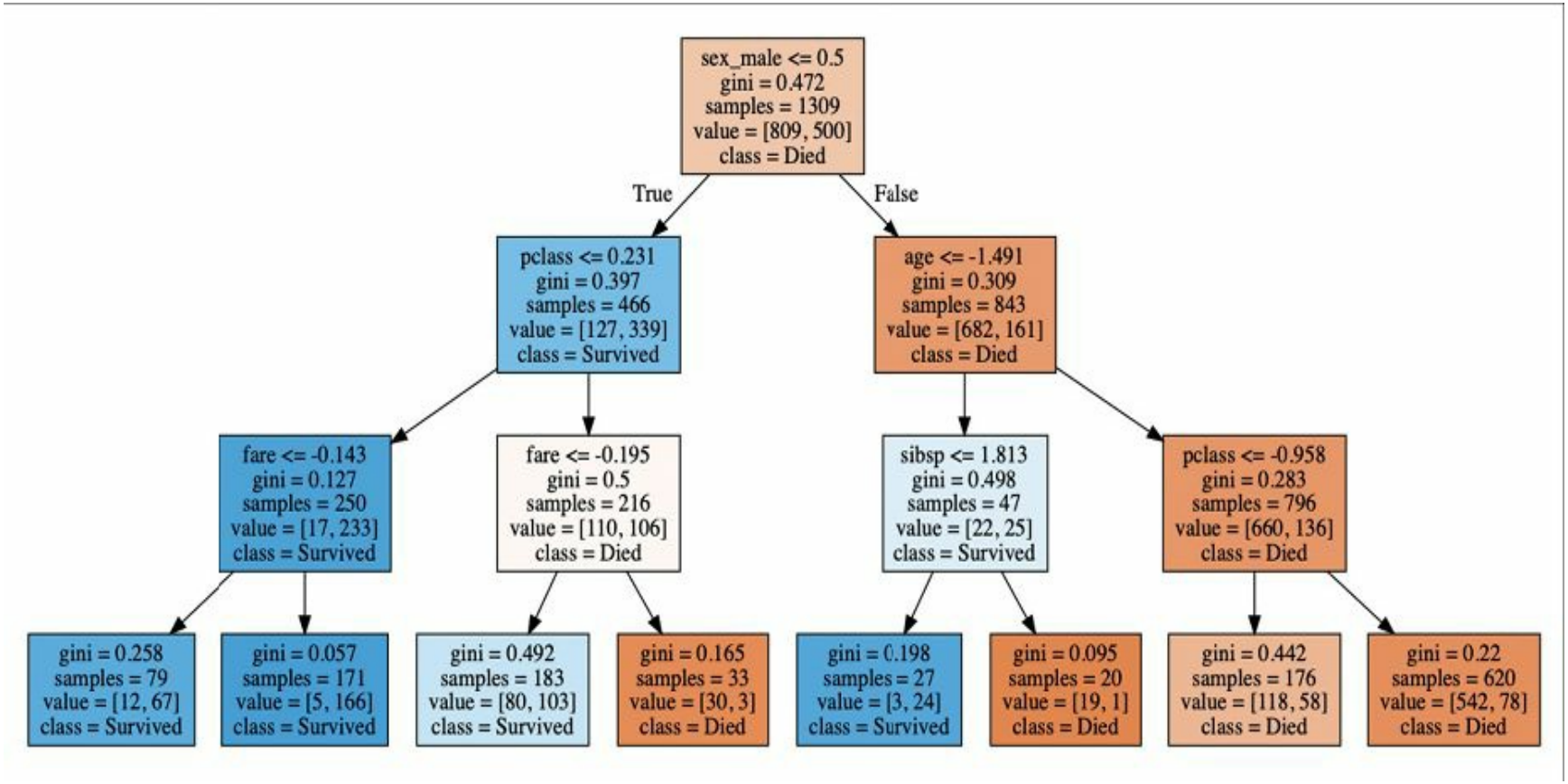

sex_male <= 0.5
gini = 0.472
samples = 1309
value = [809, 500]
class = Died
True
False
pclass <= 0.231
gini = 0.397
samples = 466
value = [127, 339]
class = Survived
age <= -1.491
gini = 0.309
samples = 843
value = [682, 161]
class = Died
fare <= -0.143
gini = 0.127
samples = 250
value = [17, 233]
class = Survived
fare <= -0.195
gini = 0.5
samples = 216
value = [110, 106]
class = Died
sibsp <= 1.813
gini = 0.498
samples = 47
value = [22, 25]
class = Survived
pclass <= -0.958
gini = 0.283
samples = 796
value = [660, 136]
class = Died
gini = 0.258
samples = 79
value = [12, 67]
class = Survived
gini = 0.057
samples = 171
value = [5, 166]
class = Survived
gini = 0.492
samples = 183
value = [80, 103]
class = Survived
gini = 0.165
samples = 33
value = [30, 3]
class = Died
gini = 0.198
samples = 27
value = [3, 24]
class = Survived
gini = 0.095
samples = 20
value = [19, 1]
class = Died
gini = 0.442
samples = 176
value = [118, 58]
class = Died
gini = 0.22
samples = 620
value = [542, 78]
class = Died

Figura 10.2 – Árvore de decisão.

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
... dt,
... out_file=dot_data,
... feature_names=X.columns,
... class_names=["Died", "Survived"],
... filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
... dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1002.png")
```

No Jupyter, utilize:

```
from IPython.display import Image
Image(g.create_png())
```

O pacote dtreeviz (*https://github.com/parrt/dtreeviz*) pode ajudar a compreender como a árvore de decisão funciona. Ele cria uma árvore com histogramas contendo rótulos, a qual possibilita insights valiosos (veja a Figura 10.3).

A seguir, mostraremos um exemplo. No Jupyter, podemos exibir o objeto `viz` diretamente. Se estivermos trabalhando com um script, podemos chamar o método `.save` para criar um PDF, um SVG ou um PNG:

```
>>> viz = dtreeviz.trees.dtreeviz(
... dt,
... X,
... y,
... target_name="survived",
... feature_names=X.columns,
... class_names=["died", "survived"],
... )
>>> viz
```

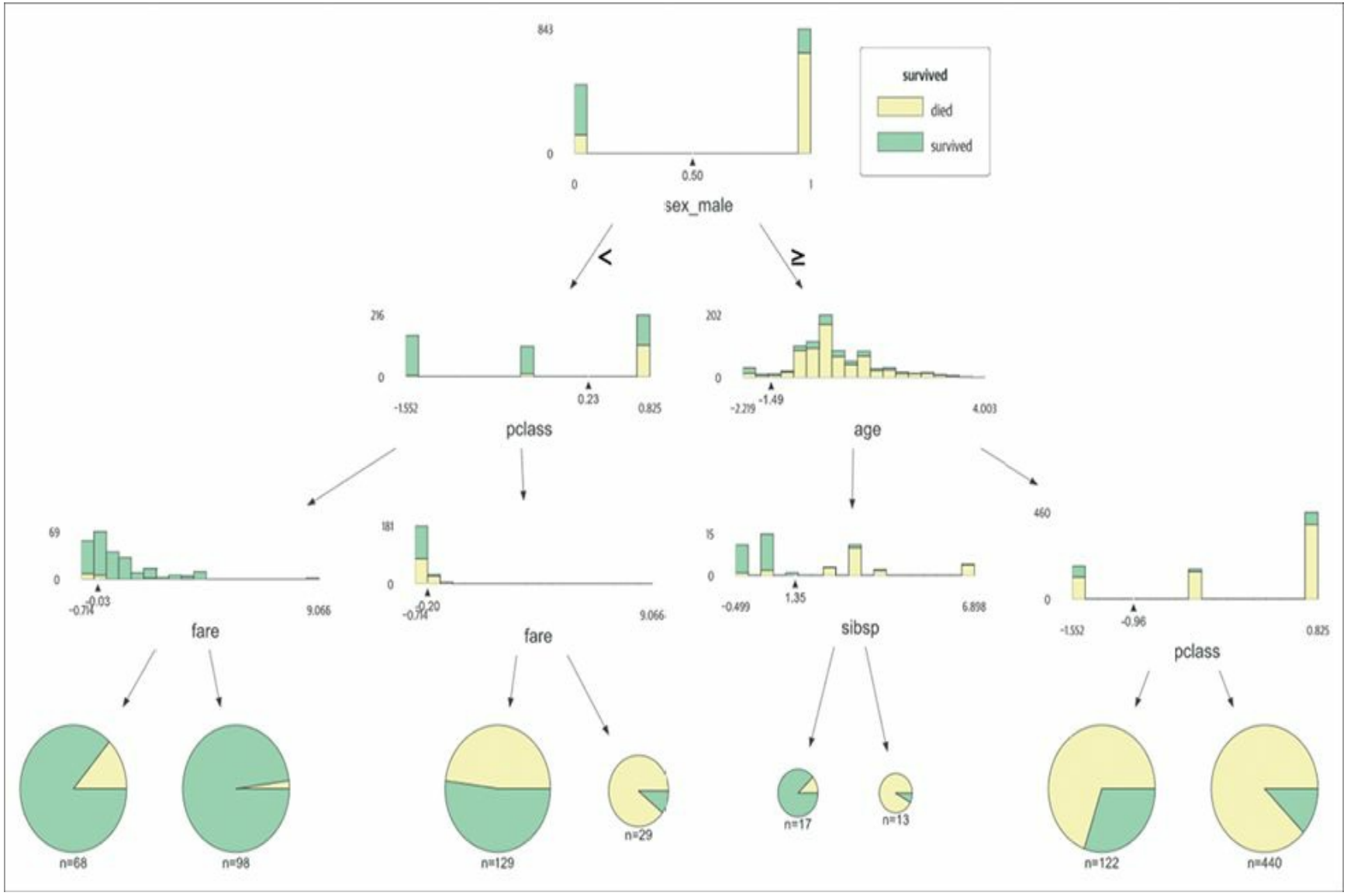

survived
died
survived
sex_male
pclass
age
fare
fare
sibsp
pclass
n=68
n=98
n=129
n=29
n=17
n=13
n=122
n=440

Figura 10.3 – Saída do dtreeviz.

Importância dos atributos mostrando a importância de Gini (redução de erro usando esse atributo):

```
>>> for col, val in sorted(
... zip(X.columns, dt.feature_importances_),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
sex_male 0.607
pclass 0.248
sibsp 0.052
fare 0.050
age 0.043
```

Podemos usar também o Yellowbrick para visualizar a importância dos atributos (veja a Figura 10.4):

```
>>> from yellowbrick.features.importances import (
... FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(dt)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1004.png", dpi=300)
```

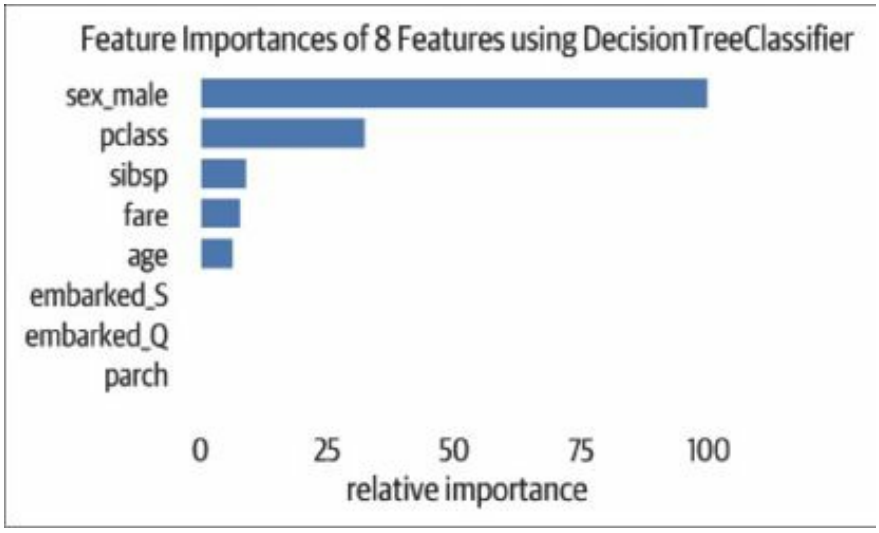


Figura 10.4 – Importância dos atributos (coeficiente de Gini) para árvore de decisão (normalizada para a importância do sexo masculino [male]).

Floresta aleatória

Uma floresta aleatória (random forest) é um conjunto de árvores de decisão. Ela usa *bagging* para corrigir a tendência das árvores de decisão à superadequação. Ao criar várias árvores treinadas com subamostras e atributos aleatórios dos dados, a variância é reduzida.

Como o treinamento é feito em subamostras dos dados, as florestas aleatórias são capazes de avaliar o erro OOB e o desempenho. Podem também exibir a importância dos atributos, calculando a média da importância em todas as árvores.

A intuição para compreender o bagging vem de um artigo de 1785 do Marquês de Condorcet. Essencialmente, ele diz que, se você estiver criando um júri, deve adicionar qualquer pessoa que tenha uma chance maior que 50% de dar um veredicto correto e então tirar a média das decisões. Sempre que adicionar outro membro (e seu processo de seleção for independente dos demais), um resultado melhor será obtido.

A ideia das florestas aleatórias é criar uma "floresta" de árvores de decisão treinadas em diferentes colunas dos dados de treinamento. Se cada árvore tiver uma chance melhor que 50% de fazer uma classificação correta, você deverá incorporar a sua predição. A floresta aleatória tem sido uma excelente ferramenta tanto para classificação como para regressão, embora, recentemente, tenha cedido espaço para as árvores com gradient boosting.

Ela tem as seguintes propriedades:

Eficiência na execução

Deve criar j árvores aleatórias. Isso pode ser feito em paralelo usando `n_jobs`. A complexidade de cada árvore é de O(mn log n), em que n é o número de amostras e m é o número de atributos. Para a criação, percorre cada um dos m atributos em um laço e ordena todas as n amostras, O(mn log n). Para predição, percorre a árvore, O(altura).

Pré-processamento dos dados

Não é necessário.

Para evitar uma superadequação

Adicione mais árvores (`n_estimators`). Use um valor menor para `max_depth`.

Interpretação dos resultados

Tem suporte para importância de atributos, porém não há uma única árvore de decisão para percorrer. É possível inspecionar árvores únicas do conjunto.

Eis um exemplo:

```
>>> from sklearn.ensemble import (
... RandomForestClassifier,
... )
>>> rf = RandomForestClassifier(random_state=42)
>>> rf.fit(X_train, y_train)
RandomForestClassifier(bootstrap=True,
  class_weight=None, criterion='gini',
  max_depth=None, max_features='auto',
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0,
  n_estimators=10, n_jobs=1, oob_score=False,
  random_state=42, verbose=0, warm_start=False)
>>> rf.score(X_test, y_test)
0.7862595419847328

>>> rf.predict(X.iloc[[0]])
array([1])
>>> rf.predict_proba(X.iloc[[0]])
array([[0., 1.]])
>>> rf.predict_log_proba(X.iloc[[0]])
array([[-inf, 0.]])
```

Parâmetros da instância (essas opções espelham a árvore de decisão):

bootstrap=True

Utiliza bootstrap ao construir as árvores.

class_weight=None

Pesos das classes no dicionário. 'balanced' definirá valores na proporção inversa das frequências das classes. O valor default é 1 para cada classe. Para várias classes, é necessário ter uma lista de dicionários (OVR) para cada classe.

criterion='gini'

Função de separação, 'gini' ou 'entropy'.

max_depth=None

Profundidade da árvore. O default será construir a árvore até que as folhas contenham menos que min_samples_split.

max_features='auto'

Número de atributos para analisar na separação. O default são todos.

max_leaf_nodes=None

Limita o número de folhas. O default é sem limites.

min_impurity_decrease=0.0

Separa um nó se uma separação diminuir a impureza, por um valor maior ou igual a esse.

min_impurity_split=None

Obsoleto.

min_samples_leaf=1

Número mínimo de amostras em cada folha.

min_samples_split=2

Número mínimo de amostras exigido para separar um nó.

min_weight_fraction_leaf=0.0

Soma mínima de pesos exigida para nós do tipo folha.

n_estimators=10

Número de árvores na floresta.

n_jobs=1

Número de jobs para adequação e predição.

oob_score=False

Informa se deve estimar oob_score.

random_state=None

Semente (seed) aleatória.

verbose=0

Verbosidade.

warm_start=False

Faz a adequação de uma nova floresta ou usa uma floresta existente.

Atributos após a adequação:

classes_

Rótulos das classes.

feature_importances_

Array de importância de Gini.

n_classes_

Número de classes.

n_features_

Número de atributos.

oob_score_

Pontuação OOB. Precisão média para cada observação não usada em árvores.

Importância dos atributos mostrando a importância de Gini (redução do erro usando esse atributo):

```
>>> for col, val in sorted(
... zip(X.columns, rf.feature_importances_),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
age 0.285
fare 0.268
sex_male 0.232
pclass 0.077
sibsp 0.059
```

DICA

O classificador de floresta aleatória calcula a importância dos atributos determinando a *diminuição média da impureza* para cada atributo (também conhecida como importância de Gini). Os atributos que reduzem a incerteza na classificação recebem pontuações maiores.

Esses números poderão se tornar imprecisos se os atributos variarem em escala ou na cardinalidade das colunas de categorias. Uma pontuação mais confiável é a *importância da permutação* (em que cada coluna tem seu valor permutado e a queda na precisão é calculada). Um método mais confiável é a *importância da coluna descartada* (em que cada coluna é descartada

e o modelo é reavaliado); infelizmente, porém, isso exige a criação de um novo modelo para cada coluna que for descartada. Veja a função importances do pacote rfpimp:

```
>>> import rfpimp
>>> rf = RandomForestClassifier(random_state=42)
>>> rf.fit(X_train, y_train)
>>> rfpimp.importances(
... rf, X_test, y_test
... ).Importance
Feature
sex_male 0.155216
fare 0.043257
age 0.033079
pclass 0.027990
parch 0.020356
embarked_Q 0.005089
sibsp 0.002545
embarked_S 0.000000
Name: Importance, dtype: float64
```

XGBoost

Embora o sklearn tenha um GradientBoostedClassifier, é melhor usar uma implementação de terceiros que utilize o extreme boosting. Elas tendem a fornecer melhores resultados.

O XGBoost (*https://oreil.ly/WBo0g*) é uma biblioteca popular, além do scikit-learn. Ele cria uma árvore fraca e, então, "melhora" as árvores subsequentes (faz um boosting) a fim de reduzir os erros residuais. O algoritmo tenta capturar e tratar qualquer padrão nos erros, até que pareçam ser aleatórios.

O XGBoost tem as seguintes propriedades:

Eficiência na execução

O XGBoost pode executar em paralelo. Utilize a opção n_jobs para informar o número de CPUs. Use a GPU para ter um desempenho melhor ainda.

Pré-processamento dos dados

Não é necessário escalar com modelos baseados em árvore. É preciso codificar os dados de categoria.

Para evitar uma superadequação

O parâmetro early_stopping_rounds=N pode ser definido para interromper o treinamento caso não haja melhoras após N rodadas. As regularizações L1 e L2 são controladas por reg_alpha e reg_lambda, respectivamente. Números maiores são mais conservadores.

Interpretação dos resultados

Inclui importância de atributos.

O XGBoost tem um parâmetro extra no método .fit. O parâmetro early_stopping_rounds pode ser combinado com o parâmetro eval_set para dizer ao XGBoost que pare de criar árvores caso a métrica de avaliação não tenha melhorado após esse número de rodadas de boosting (melhorias). 'eval_metric também pode ser definido com um dos seguintes valores: 'rmse', 'mae', 'logloss', 'error' (default), 'auc', 'aucpr', bem como com uma função personalizada.

Eis um exemplo de uso da biblioteca:

```
>>> import xgboost as xgb
>>> xgb_class = xgb.XGBClassifier(random_state=42)
>>> xgb_class.fit(
... X_train,
... y_train,
... early_stopping_rounds=10,
... eval_set=[(X_test, y_test)],
... )
XGBClassifier(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1, gamma=0,
  learning_rate=0.1, max_delta_step=0, max_depth=3,
  min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='binary:logistic', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)

>>> xgb_class.score(X_test, y_test)
0.7862595419847328

>>> xgb_class.predict(X.iloc[[0]])
array([1])
>>> xgb_class.predict_proba(X.iloc[[0]])
array([[0.06732017, 0.93267983]], dtype=float32)
```

Parâmetros da instância:

max_depth=3

Profundidade máxima.

learning_rate=0.1

Taxa de aprendizagem (também chamada de eta) para boosting (entre 0 e 1). Após cada passo de boosting, (melhoria) os pesos recém-adicionados são escalados de acordo com esse fator. Quanto menor o valor, mais conservador será, mas também serão necessárias mais árvores para convergir. Na chamada a .train, você pode passar um parâmetro learning_rates, que é uma lista de taxas em cada rodada (isto é, [.1]*100 + [.05]*100).

n_estimators=100

Número de rodadas ou árvores melhoradas.

silent=True

Inverso de verboso. Informa se deve exibir mensagens enquanto executa o boosting.

objective='binary:logistic'

Tarefa de aprendizagem ou callable para classificação.

booster='gbtree'

Pode ser 'gbtree', 'gblinear' ou 'dart'.

nthread=None

Obsoleto.

n_jobs=1

Número de threads a serem usadas.

gamma=0

Controla a poda (pruning). Varia de 0 a infinito. Redução de perda mínima necessária para separar mais uma folha. Quanto maior o valor de gama, mais conservador será. Se as pontuações de treinamento e de teste divergirem, insira um número maior (em torno de 10). Se as pontuações de treinamento e teste estiverem próximas, utilize um número menor.

min_child_weight=1

Valor mínimo para a soma hessiana de um filho.

max_delta_step=0

Deixa as atualizações mais conservadoras. Defina com valores de 1 a 10 para classes desbalanceadas.

subsample=1

Fração das amostras a serem usadas na próxima rodada.

colsample_bytree=1

Fração das colunas a serem usadas por rodada.

colsample_bylevel=1

Fração das colunas a serem usadas por nível.

colsample_bynode=1

Fração das colunas a serem usadas por nó.

reg_alpha=0

A regularização L1 (médias dos pesos) incentiva a dispersão. Aumente o valor para ser mais conservador.

reg_lambda=1

A regularização L2 (raiz dos quadrados dos pesos) incentiva pesos menores. Aumente o valor para ser mais conservador.

scale_pos_weight=1

Razão entre peso negativo/positivo.

base_score=.5

Previsão inicial.

seed=None

Obsoleto.

random_state=0

Semente (seed) aleatória.

missing=None

Valor de interpretação para missing. None quer dizer np.nan.

importance_type='gain'

Tipo da importância do atributo: 'gain', 'weight', 'cover', 'total_gain' ou 'total_cover'.

Atributos:

coef_

Coeficientes para learners gblinear.

feature_importances_

Importância de atributos para learners gbtree.

A importância dos atributos é o ganho médio em todos os nós em que o atributo é usado:

```
>>> for col, val in sorted(
... zip(
... X.columns,
... xgb_class.feature_importances_,
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
fare 0.420
age 0.309
pclass 0.071
sex_male 0.066
sibsp 0.050
```

O XGBoost é capaz de gerar um gráfico da importância dos atributos (veja a Figura 10.5).

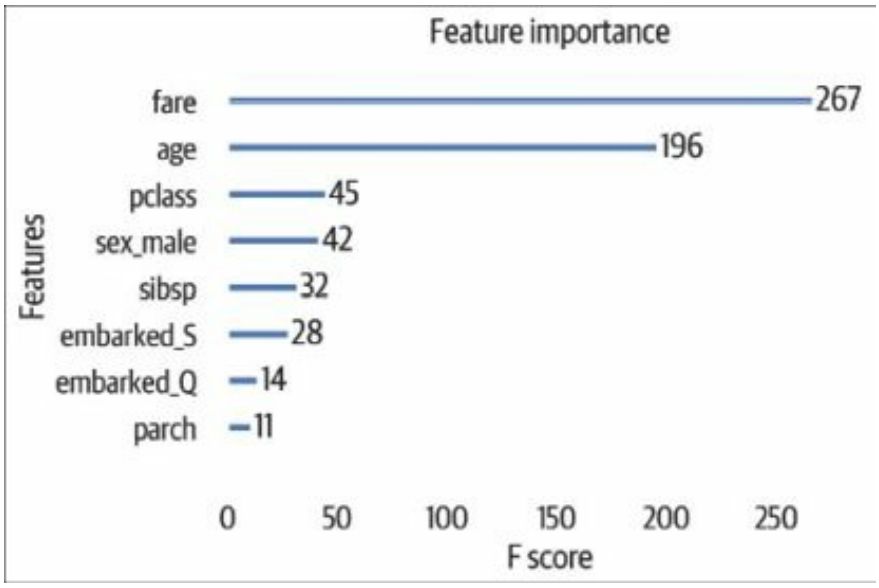


Figura 10.5 – Importância dos atributos mostrando o peso (quantas vezes um atributo aparece nas árvores).

Ele tem um parâmetro importance_type. O valor default é "weight", que é o número de vezes que um atributo aparece em uma árvore. Também pode ser "gain", que mostra o ganho médio quando o atributo é usado, ou "cover", que é o número de amostras afetado por uma separação:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgb_class, ax=ax)
>>> fig.savefig("images/mlpr_1005.png", dpi=300)
```

Podemos gerar esse gráfico com o Yellowbrick, que normaliza os valores (veja a Figura 10.6):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgb_class)
>>> fi_viz.fit(X, y)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1006.png", dpi=300)
```

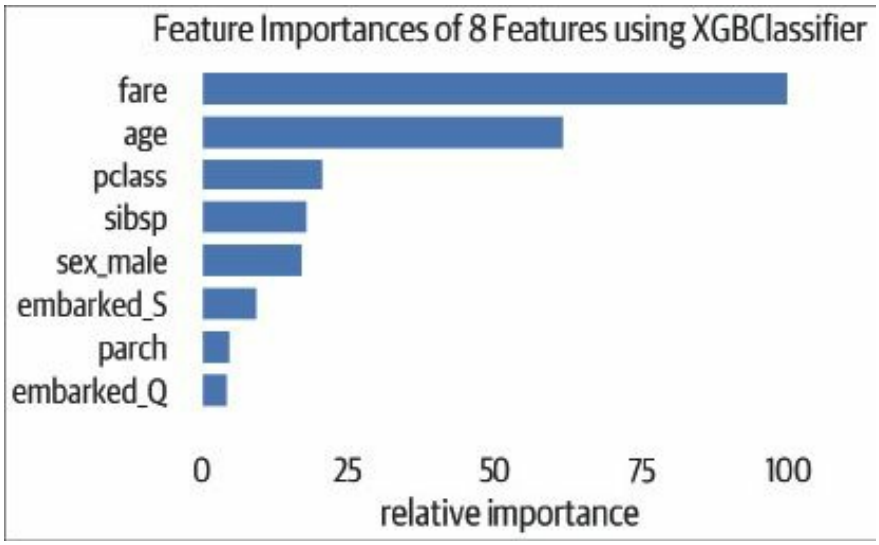


Figura 10.6 – Importância dos atributos gerada com o Yellowbrick para o XGBoost (normalizado para 100).

O XGBoost fornece uma representação tanto textual como gráfica das árvores. Eis a representação textual:

```
>>> booster = xgb_class.get_booster()
>>> print(booster.get_dump()[0])
0:[sex_male<0.5] yes=1,no=2,missing=1
  1:[pclass<0.23096] yes=3,no=4,missing=3
    3:[fare<-0.142866] yes=7,no=8,missing=7
      7:leaf=0.132530
      8:leaf=0.184
    4:[fare<-0.19542] yes=9,no=10,missing=9
      9:leaf=0.024598
      10:leaf=-0.1459
  2:[age<-1.4911] yes=5,no=6,missing=5
    5:[sibsp<1.81278] yes=11,no=12,missing=11
      11:leaf=0.13548
```

```
  12:leaf=-0.15000
 6:[pclass<-0.95759] yes=13,no=14,missing=13
  13:leaf=-0.06666
  14:leaf=-0.1487
```

O valor na folha é a pontuação para a classe 1. Pode ser convertido em uma probabilidade usando a função logística. Se as decisões chegarem à folha 7, a probabilidade da classe 1 será de 53%. Essa é a pontuação de uma única árvore. Se nosso modelo tivesse 100 árvores, você somaria o valor de cada folha e obteria a probabilidade com a função logística:

```
>>> # pontuação da folha 7 da primeira árvore
>>> 1 / (1 + np.exp(-1 * 0.1238))
0.5309105310475829
```

A seguir, apresentamos a versão gráfica da primeira árvore do modelo (veja a Figura 10.7):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_tree(xgb_class, ax=ax, num_trees=0)
>>> fig.savefig("images/mlpr_1007.png", dpi=300)
```

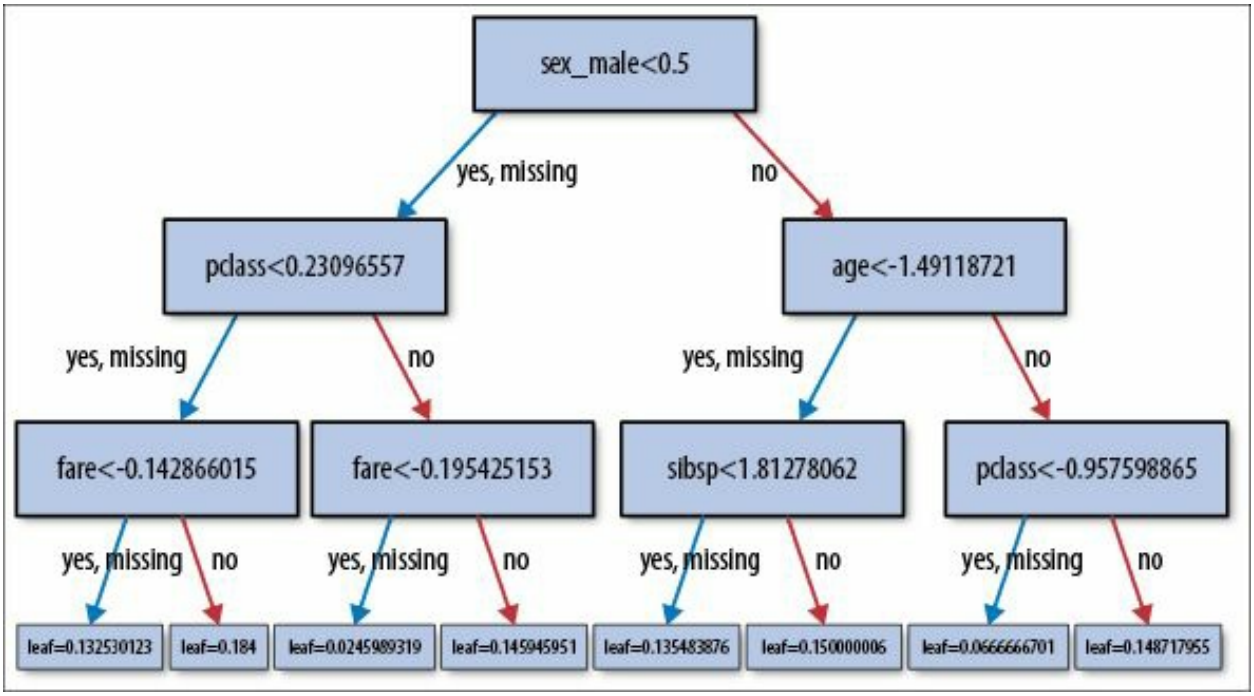


Figura 10.7 – Árvore do XGBoost.

O pacote xgbfir (*https://oreil.ly/kPnRv*) é uma biblioteca desenvolvida com base no XGBoost. Essa biblioteca fornece várias medidas relacionadas à importância dos atributos. Sua característica única é que ela fornece essas medidas sobre as colunas, e sobre pares de colunas também, de modo que

será possível ver as interações. Além disso, você pode obter informações sobre interações entre trincas (três colunas).

As medidas fornecidas são:

Gain

Ganho total de cada atributo ou interação entre atributos.

FScore

Quantidade de separações possíveis em um atributo ou interação entre atributos.

wFScore

Quantidade de possíveis separações em um atributo ou interação entre atributos, com pesos atribuídos de acordo com a probabilidade de as separações ocorrerem.

Average wFScore

wFScore dividido por FScore.

Average Gain

Gain dividido por FScore.

Expected Gain

Ganho total de cada atributo ou interação entre atributos, com pesos de acordo com a probabilidade de obter o ganho.

A interface consiste apenas de dados exportados para uma planilha, portanto, usaremos o pandas para ler os dados de volta. Eis a importância das colunas:

```
>>> import xgbfir
>>> xgbfir.saveXgbFI(
... xgb_class,
... feature_names=X.columns,
... OutputXlsxFile="fir.xlsx",
... )
>>> pd.read_excel("/tmp/surv-fir.xlsx").head(3).T
                          0 1 2
Interaction sex_male pclass fare
Gain 1311.44 585.794 544.884
FScore 42 45 267
wFScore 39.2892 21.5038 128.33
Average wFScore 0.935458 0.477861 0.480636
Average Gain 31.2247 13.0177 2.04076
Expected Gain 1307.43 229.565 236.738
```

```
Gain Rank 1 2 3
FScore Rank 4 3 1
wFScore Rank 3 4 1
Avg wFScore Rank 1 5 4
Avg Gain Rank 1 2 4
Expected Gain Rank 1 3 2
Average Rank 1.83333 3.16667 2.5
Average Tree Index 32.2381 20.9778 51.9101
Average Tree Depth 0.142857 1.13333 1.50562
```

A partir dessa tabela, vemos que sex_male tem uma posição elevada quanto ao ganho (gain), wFScore médio (average wFScore), ganho médio (average gain) e ganho esperado (expected gain), enquanto fare (preço da passagem) se destaca quanto a FScore e wFScore.

Vamos analisar os pares para ver as interações entre colunas:

```
>>> pd.read_excel(
... "fir.xlsx",
... sheet_name="Interaction Depth 1",
... ).head(2).T
Interaction pclass|sex_male age|sex_male
Gain 2090.27 964.046
FScore 35 18
wFScore 14.3608 9.65915
Average wFScore 0.410308 0.536619
Average Gain 59.722 53.5581
Expected Gain 827.49 616.17
Gain Rank 1 2
FScore Rank 5 10
wFScore Rank 4 8
Avg wFScore Rank 8 5
Avg Gain Rank 1 2
Expected Gain Rank 1 2
Average Rank 3.33333 4.83333
Average Tree Index 18.9714 38.1111
Average Tree Depth 1 1.11111
```

Nesse caso, vemos que as duas principais interações envolvem a coluna sex_male em combinação com pclasse e age. Se você pudesse criar apenas um modelo com dois atributos, provavelmente iria querer escolher pclass e sex_male.

Por fim, vamos observar as trincas:

```
>>> pd.read_excel(
... "fir.xlsx",
```

```
... sheet_name="Interaction Depth 2",
... ).head(1).T
                                0
Interaction fare|pclass|sex_male
Gain 2973.16
FScore 44
wFScore 8.92572
Average wFScore 0.202857
Average Gain 67.5719
Expected Gain 549.145
Gain Rank 1
FScore Rank 1
wFScore Rank 4
Avg wFScore Rank 21
Avg Gain Rank 3
Expected Gain Rank 2
Average Rank 5.33333
Average Tree Index 16.6591
Average Tree Depth 2
```

A saída mostra apenas a primeira trinca por causa da limitação no espaço, mas a planilha inclui várias outras trincas:

```
>>> pd.read_excel(
... "/tmp/surv-fir.xlsx",
... sheet_name="Interaction Depth 2",
... )[["Interaction", "Gain"]].head()
          Interaction Gain
0 fare|pclass|sex_male 2973.162529
1 age|pclass|sex_male 1621.945151
2 age|sex_male|sibsp 1042.320428
3 age|fare|sex_male 366.860828
4 fare|fare|sex_male 196.224791
```

Gradient Boosted com LightGBM

O LightGBM é uma implementação da Microsoft. Ele utiliza um método de amostragem para lidar com valores contínuos. Isso permite uma criação mais rápida das árvores (em comparação com, por exemplo, o XGBoost), além de reduzir o uso de memória.

O LightGBM também cria árvores em profundidade antes (por *folhas*, em vez de *níveis*). Por causa disso, em vez de usar `max_depth` para controlar a superadequação, utilize `num_leaves` (esse valor é < 2^(`max_depth`)).

NOTA

A instalação dessa biblioteca atualmente exige ter um compilador, e é um pouco mais complicada do que apenas usar `pip install`.

Esse modelo tem as seguintes propriedades:

Eficiência na execução

Consegue tirar proveito de várias CPUs. Se usar binning, pode ser 15 vezes mais rápido que o XGBoost.

Pré-processamento dos dados

Tem algum suporte para codificação de colunas de categorias como inteiros (ou o tipo `Categorical` do pandas), mas o AUC parece ser pior em comparação com a codificação one-hot.

Para evitar uma superadequação

Reduza `num_leaves`, aumente `min_data_in_leaf` e utilize `min_gain_to_split` com `lambda_l1` ou `lambda_l2`.

Interpretação dos resultados

A importância dos atributos está disponível. Árvores individuais são fracas e tendem a ser difíceis de interpretar.

Eis um exemplo de uso da biblioteca:

```
>>> import lightgbm as lgb
>>> lgbm_class = lgb.LGBMClassifier(
... random_state=42
... )
>>> lgbm_class.fit(X_train, y_train)
LGBMClassifier(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
  min_split_gain=0.0, n_estimators=100,
  n_jobs=-1, num_leaves=31, objective=None,
  random_state=42, reg_alpha=0.0, reg_lambda=0.0,
  silent=True, subsample=1.0,
  subsample_for_bin=200000, subsample_freq=0)

>>> lgbm_class.score(X_test, y_test)
0.7964376590330788
```

```
>>> lgbm_class.predict(X.iloc[[0]])
array([1])
>>> lgbm_class.predict_proba(X.iloc[[0]])
array([[0.01637168, 0.98362832]])
```

Parâmetros da instância

boosting_type='gbdt'

Pode ser: 'gbdt' (gradient boosting), 'rf' (random forest), 'dart' (dropouts meet multiple additive regression trees) ou 'goss' (gradient-based, one-sided sampling).

class_weight=None

Dicionário ou 'balanced'. Use um dicionário para definir pesos para cada rótulo de classe quando estiver resolvendo problemas de várias classes. Para problemas binários, utilize is_unbalance ou scale_pos_weight.

colsample_bytree=1.0

Intervalo de (0, 1.0]. Seleciona uma porcentagem dos atributos para cada rodada de boosting.

importance_type='split'

Como calcular a importância dos atributos. 'split' quer dizer o número de vezes que um atributo é usado. 'gain' são os ganhos totais das separações para um atributo.

learning_rate=0.1

Intervalo de (0, 1.0]. Taxa de aprendizagem para boosting. Um valor menor atrasa a superadequação, pois as rodadas de boosting terão menos impacto. Um número menor deve resultar em um desempenho melhor, mas exigirá um valor maior para num_iterations.

max_depth=-1

Profundidade máxima da árvore. -1 significa sem limites. Profundidades maiores tendem a causar mais superadequação.

min_child_samples=20

Número de amostras exigido para uma folha. Números menores implicam mais superadequação.

min_child_weight=0.001

Soma do peso hessiano exigido para uma folha.

min_split_gain=0.0

Redução de perda exigida para particionar uma folha.

n_estimators=100

Número de árvores ou rodadas de boosting.

n_jobs=-1

Número de threads.

num_leaves=31

Número máximo de folhas na árvore.

objective=None

None é 'binary' ou 'multiclass' para classificador. Pode ser uma função ou uma string.

random_state=42

Semente (seed) aleatória.

reg_alpha=0.0

Regularização L1 (média dos pesos). Aumente esse valor para ser mais conservador.

reg_lambda=0.0

Regularização L2 (raiz dos quadrados dos pesos). Aumente esse valor para ser mais conservador.

silent=True

Modo verboso.

subsample=1.0

Fração das amostras a serem usadas na próxima rodada.

subsample_for_bin=200000

Amostras necessárias para criar bins.

subsample_freq=0

Frequência das subamostras. Mude para 1 para ativar.

Importância dos atributos com base em 'splits' (número de vezes que um item é usado):

```
>>> for col, val in sorted(
... zip(cols, lgbm_class.feature_importances_),
```

```
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
fare 1272.000
age 1182.000
sibsp 118.000
pclass 115.000
sex_male 110.000
```

A biblioteca LightGBM é capaz de gerar um gráfico da importância dos atributos (veja a Figura 10.8). O default é baseado em 'splits', isto é, no número de vezes que um atributo é usado. Você pode definir 'importance_type' se quiser mudá-lo para 'gain':

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgbm_class, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1008.png", dpi=300)
```

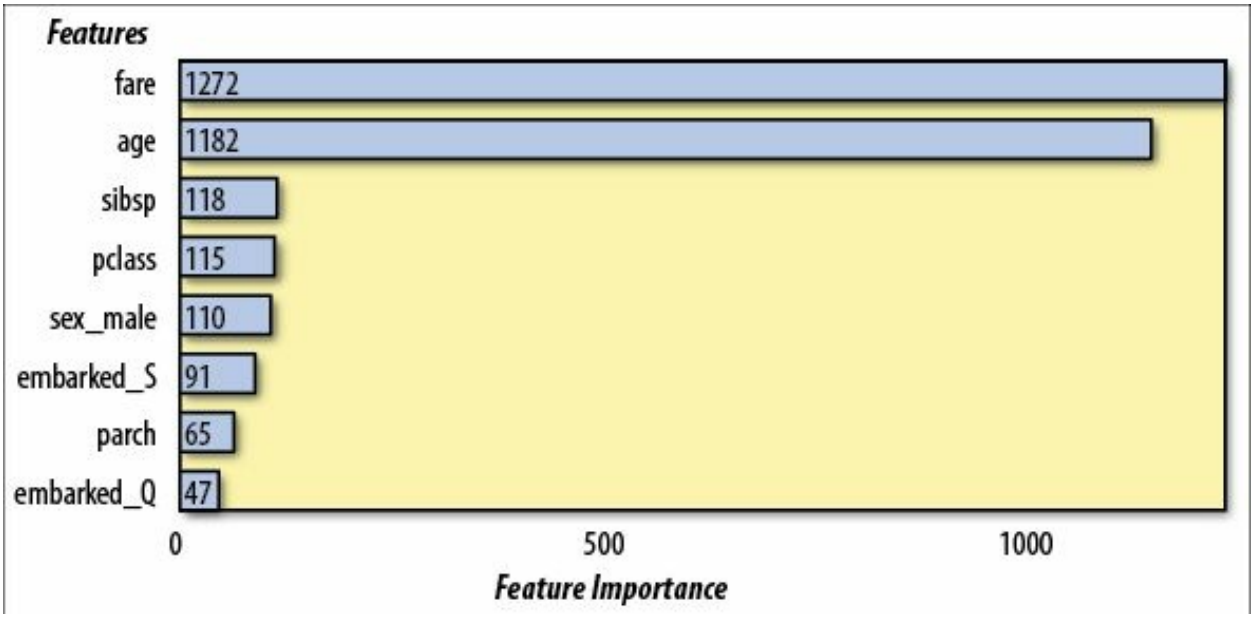


Figura 10.8 – Separação da importância dos atributos para o LightGBM.

ALERTA

Na versão 0.9, o Yellowbrick não funciona com o LightGBM para criar gráficos de importância de atributos.

Podemos também criar uma árvore de decisões (veja a Figura 10.9):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
```

```
>>> lgb.plot_tree(lgbm_class, tree_index=0, ax=ax)
>>> fig.savefig("images/mlpr_1009.png", dpi=300)
```

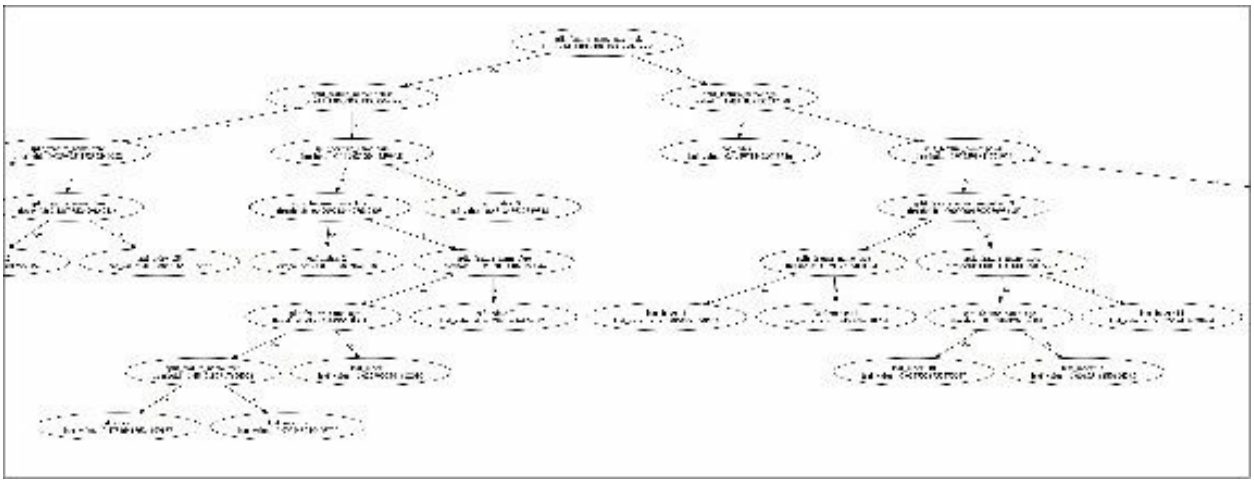


Figura 10.9 – Árvore gerada com o LightGBM.

DICA

No Jupyter, utilize o comando a seguir para visualizar uma árvore:

```
lgb.create_tree_digraph(lgbm_class)
```

TPOT

O TPOT (*https://oreil.ly/NFJvl*) usa um algoritmo genético para testar diferentes modelos e ensembles (conjuntos). Pode demorar horas ou dias para executar, pois o algoritmo considera diversos modelos e passos de pré-processamento, assim como os hiperparâmetros desses modelos e as opções de ensembling (agrupamentos). Em uma máquina típica, uma geração dessas pode demorar cinco minutos ou mais para executar.

Esse modelo tem as seguintes propriedades:

Eficiência na execução

Pode demorar horas ou dias. Use _jobs=-1 para utilizar todas as CPUs.

Pré-processamento dos dados

Você deve remover NaN e dados de categoria.

Para evitar uma superadequação

O ideal é que os resultados devam usar validação cruzada para minimizar a superadequação.

Interpretação dos resultados

Depende dos resultados.

Eis um exemplo de uso da biblioteca:

```
>>> from tpot import TPOTClassifier
>>> tc = TPOTClassifier(generations=2)
>>> tc.fit(X_train, y_train)
>>> tc.score(X_test, y_test)
0.7888040712468194

>>> tc.predict(X.iloc[[0]])
array([1])
>>> tc.predict_proba(X.iloc[[0]])
array([[0.07449919, 0.92550081]])
```

Parâmetros da instância:

generations=100

Iterações a serem executadas.

population_size=100

Tamanho da população para a programação genética. Tamanhos maiores em geral têm melhor desempenho, porém exigem mais tempo e dinheiro.

offspring_size=None

Descendentes para cada geração. O default é population_size.

mutation_rate=.9

Taxa de mutação para o algoritmo, de [0, 1]. O default é 0,9.

crossover_rate=.1

Taxa de cross-over (quantos pipelines devem ser criados em uma geração). Intervalo de [0, 1]. O default é 0,1.

scoring='accuracy'

Sistema de pontuação. Usa strings do sklearn.

cv=5

Folds de validação cruzada.

subsample=1

Subamostra das instâncias de treinamento. Intervalo de [0, 1]. O default é 1.

n_jobs=1

Número de CPUs a serem usadas, -1 para todos os cores (núcleos).

max_time_mins=None

Quantidade máxima de minutos para executar.

max_eval_time_mins=5

Quantidade máxima de minutos para avaliar um único pipeline.

random_state=None

Semente (seed) aleatória.

config_dict

Opções de configuração para otimização.

warm_start=False

Reutiliza chamadas anteriores de .fit.

memory=None

Pode fazer cache de pipelines. 'auto' ou um path fará a persistência em um diretório.

use_dask=False

Usa dask.

periodic_checkpoint_folder=None

Path para uma pasta na qual será feita, periodicamente, a persistência do melhor pipeline.

early_stop=None

Para após executar essa quantidade de gerações sem que haja melhorias.

verbosity=0

0 = nenhuma, 1 = mínima, 2 = alta ou 3 = total. Para um valor maior ou igual a 2, uma barra de progresso será exibida.

disable_update_check=False

Desativa a verificação de versão.

Atributos:

evaluated_individuals_

Dicionário com todos os pipelines que foram avaliados.

fitted_pipeline_

Melhor pipeline.

Depois de executar, você poderá exportar o pipeline:

```
>>> tc.export("tpot_exported_pipeline.py")
```

O resultado terá o seguinte aspecto:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import \
    train_test_split
from sklearn.pipeline import make_pipeline, \
    make_union
from sklearn.preprocessing import Normalizer
from tpot.builtins import StackingEstimator

# NOTA: Certifique-se de que o rótulo da classe seja
# 'target' no arquivo de dados
tpot_data = pd.read_csv('PATH/TO/DATA/FILE',
    sep='COLUMN_SEPARATOR', dtype=np.float64)
features = tpot_data.drop('target', axis=1).values
training_features, testing_features, \
    training_target, testing_target = \
    train_test_split(features,
      tpot_data['target'].values, random_state=42)

# Pontuação no conjunto de treinamento foi de: 0.8122535043953432
exported_pipeline = make_pipeline(
  Normalizer(norm="max"),
  StackingEstimator(
    estimator=ExtraTreesClassifier(bootstrap=True,
      criterion="gini", max_features=0.85,
      min_samples_leaf=2, min_samples_split=19,
      n_estimators=100)),
  ExtraTreesClassifier(bootstrap=False,
    criterion="entropy", max_features=0.3,
    min_samples_leaf=13, min_samples_split=9,
    n_estimators=100)
)

exported_pipeline.fit(training_features, training_target)
results = exported_pipeline.predict(testing_features)
```

CAPÍTULO 11

Seleção do modelo

Este capítulo discutirá a otimização de hiperparâmetros. Também explorará a questão acerca de o modelo exigir mais dados para ter um melhor desempenho.

Curva de validação

Criar uma curva de validação é uma forma de determinar um valor apropriado para um hiperparâmetro. Uma curva de validação é um gráfico que mostra como o desempenho do modelo responde a mudanças no valor do hiperparâmetro (veja a Figura 11.1).

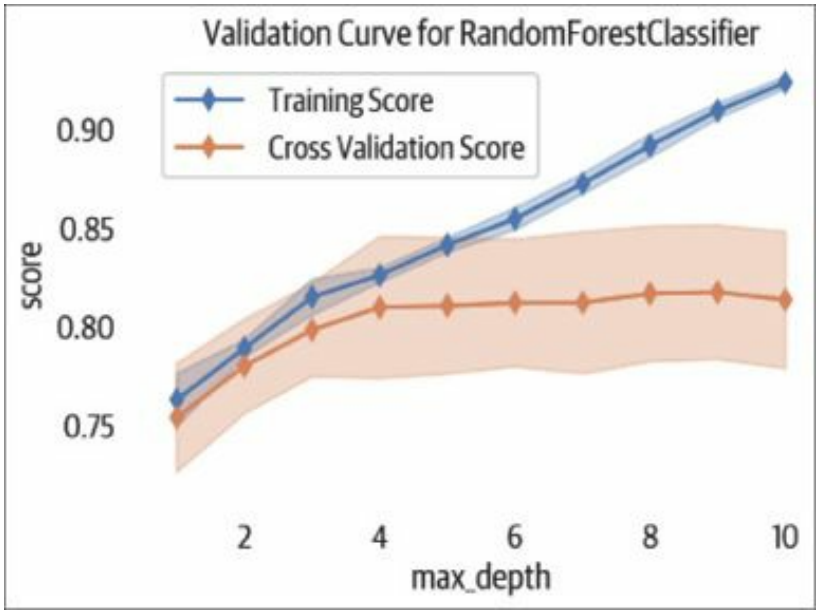


Figura 11.1 – Gráfico da curva de validação.

O gráfico mostra tanto os dados de treinamento como de validação. As pontuações dos dados de validação nos permitem inferir como o modelo responderia a dados não vistos anteriormente. Em geral, escolhemos um hiperparâmetro que maximize a pontuação dos dados de validação.

No exemplo a seguir, usaremos o Yellowbrick para ver se uma modificação no valor do hiperparâmetro max_depth altera o desempenho de um modelo de floresta aleatória (random forest). Podemos fornecer um parâmetro scoring definido com uma métrica de um modelo do scikit-learn (o default para uma classificação é 'accuracy'):

DICA

Utilize o parâmetro n_jobs para tirar proveito das CPUs e executar esse código mais rápido. Se esse parâmetro for definido com -1, todas as CPUs serão usadas.

```
>>> from yellowbrick.model_selection import (
... ValidationCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> vc_viz = ValidationCurve(
... RandomForestClassifier(n_estimators=100),
... param_name="max_depth",
... param_range=np.arange(1, 11),
... cv=10,
... n_jobs=-1,
... )
>>> vc_viz.fit(X, y)
>>> vc_viz.poof()
>>> fig.savefig("images/mlpr_1101.png", dpi=300)
```

A classe ValidationCurve aceita um parâmetro scoring. O parâmetro pode ser uma função personalizada ou uma das opções a seguir, conforme a tarefa.

As opções de scoring para classificação incluem: 'accuracy', 'average_precision', 'f1', 'f1_micro', 'f1_macro', 'f1_weighted', 'f1_samples', 'neg_log_loss', 'precision', 'recall' e 'roc_auc'.

As opções de scoring para clustering são: 'adjusted_mutual_info_score', 'adjusted_rand_score', 'completeness_score', 'fowlkesmallows_score', 'homogeneity_score', 'mutual_info_score', 'normalized_mutual_info_score' e 'v_measure_score'.

As opções de scoring para regressão são: 'explained_variance', 'neg_mean_absolute_error', 'neg_mean_squared_error', 'neg_mean_squared_log_error', 'neg_median_absolute_error' e 'r2'.

Curva de aprendizagem

Para selecionar o melhor modelo para o seu projeto, quantos dados serão necessários? Uma curva de aprendizado pode nos ajudar a responder a essa

pergunta. O gráfico mostra as instâncias de treinamento e a pontuação para validação cruzada à medida que criamos modelos com mais amostras. Se a pontuação da validação cruzada continuar a subir, por exemplo, poderá ser um sinal de que mais dados ajudariam o modelo a ter um melhor desempenho.

A figura a seguir mostra uma curva de validação, além de nos ajudar a explorar o bias (viés) e a variância em nosso modelo (veja a Figura 11.2).

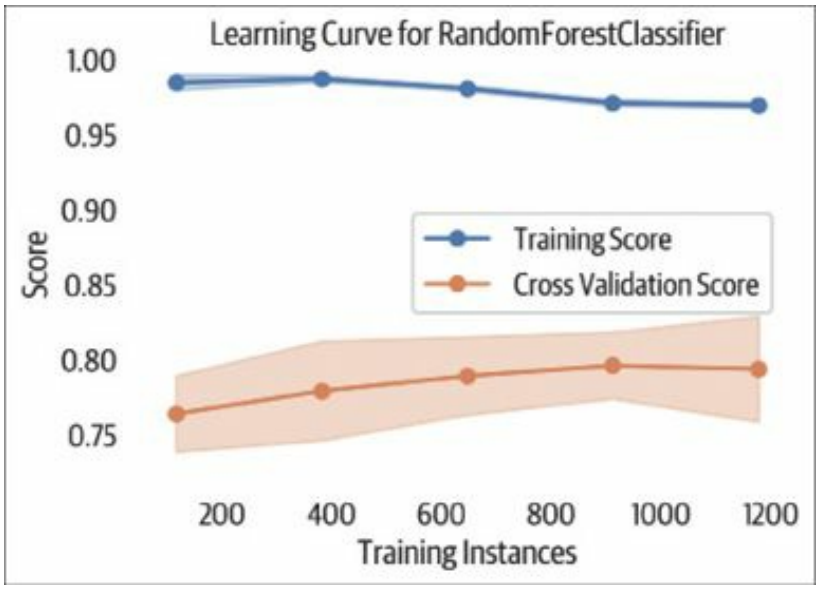


Figura 11.2 – Gráfico da curva de aprendizagem. O ponto mais alto na pontuação dos dados de validação mostra que adicionar mais dados não aperfeiçoaria esse modelo.

Se houver variabilidade (uma área sombreada grande) na pontuação dos dados de treinamento, é sinal de que o modelo sofre de erros de bias e é simples demais (há subadequação). Se houver variabilidade na pontuação para validação cruzada, é porque o modelo está sujeito a erros de variância e é complicado demais (há superadequação). Outro sinal de que o modelo apresenta superadequação é o fato de que o desempenho para o conjunto de dados de validação é muito pior do que o desempenho para o conjunto de dados de treinamento.

A seguir, mostramos um exemplo de como criar uma curva de aprendizagem usando o Yellowbrick:

```
>>> from yellowbrick.model_selection import (
... LearningCurve,
... )
```

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lc3_viz = LearningCurve(
... RandomForestClassifier(n_estimators=100),
... cv=10,
... )
>>> lc3_viz.fit(X, y)
>>> lc3_viz.poof()
>>> fig.savefig("images/mlpr_1102.png", dpi=300)
```

Essa visualização também pode ser usada para regressão ou clustering, se modificarmos as opções de pontuação.

CAPÍTULO 12

Métricas e avaliação de classificação

Discutiremos as seguintes métricas e ferramentas de avaliação neste capítulo: matrizes de confusão, métricas variadas, um relatório de classificação e algumas visualizações.

Tudo isso será avaliado utilizando um modelo de árvore de decisão que faz a predição de sobrevivência ao Titanic.

Matriz de confusão

Uma matriz de confusão pode ajudar a compreender o desempenho de um classificador.

Um classificador binário pode ter quatro resultados de classificação: TP (True Positives, ou Verdadeiros Positivos), TN (True Negatives, ou Verdadeiros Negativos), FP (False Positives, ou Falso-Positivos) e FN (False Negatives, ou Falso-Negativos). As duas primeiras classificações são corretas.

A seguir, veremos um exemplo comum para que nos lembremos dos outros resultados. Supondo que positivo significa “estar grávida” e negativo seja “não estar grávida”, um falso-positivo seria como afirmar que um homem está grávido. Um falso-negativo seria afirmar que uma mulher grávida não está (quando a gravidez está claramente visível) (veja a Figura 12.1). Esses dois últimos tipos de erro são conhecidos como erros do *tipo 1* e do *tipo 2*, respectivamente (veja a Tabela 12.1).

Outra maneira de se lembrar dessas classificações é por meio do fato de que P (para falso-positivo) contém uma única linha vertical (erro do tipo 1), enquanto N (para falso-negativo) contém duas.

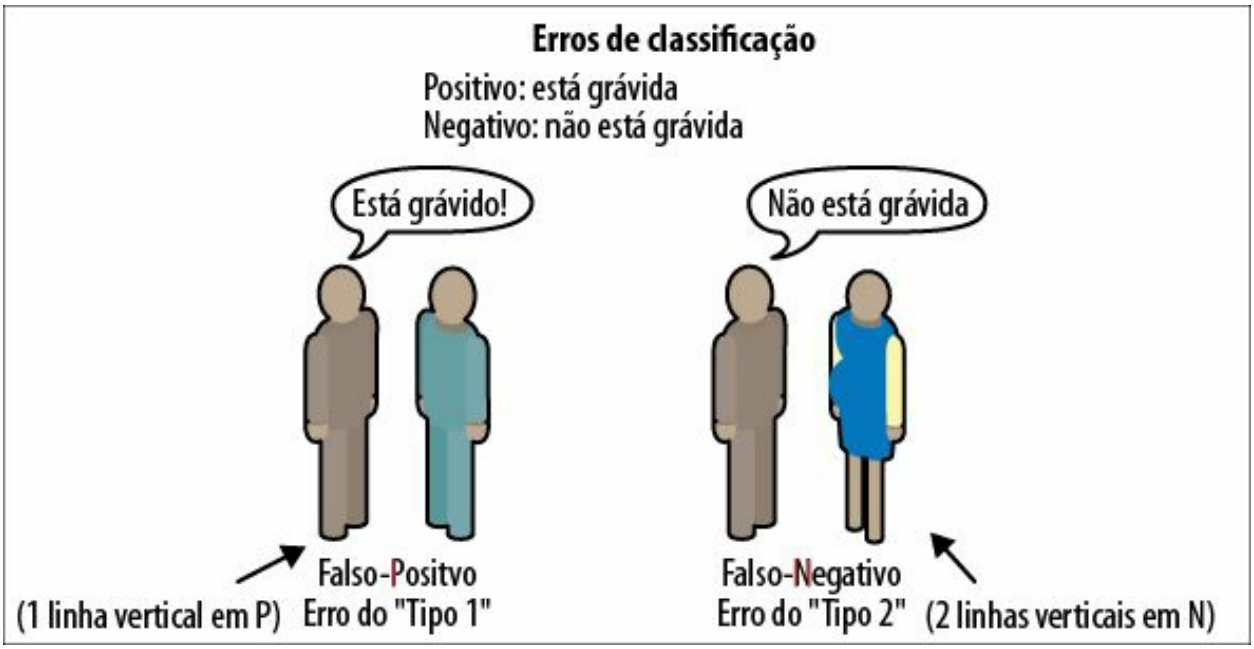


Figura 12.1 – Erros de classificação.

Tabela 12.1 – Resultados de uma classificação binária em uma matriz de confusão

Real	Previsto como negativo	Previsto como positivo
Verdadeiramente negativo	Verdadeiro negativo	Falso-positivo (tipo 1)
Verdadeiramente positivo	Falso-negativo (tipo 2)	Verdadeiro positivo

A seguir, vemos o código pandas para calcular os resultados da classificação. Os comentários mostram os resultados. Usaremos essas variáveis para calcular outras métricas:

```
>>> y_predict = dt.predict(X_test)
>>> tp = (
... (y_test == 1) & (y_test == y_predict)
... ).sum() # 123
>>> tn = (
... (y_test == 0) & (y_test == y_predict)
... ).sum() # 199
>>> fp = (
... (y_test == 0) & (y_test != y_predict)
... ).sum() # 25
>>> fn = (
... (y_test == 1) & (y_test != y_predict)
... ).sum() # 46
```

O ideal é que classificadores bem-comportados tenham contadores elevados

na linha diagonal dos verdadeiros. Podemos criar um DataFrame usando a função confusion_matrix do sklearn:

```
>>> from sklearn.metrics import confusion_matrix
>>> y_predict = dt.predict(X_test)
>>> pd.DataFrame(
... confusion_matrix(y_test, y_predict),
... columns=[
... "Predict died",
... "Predict Survive",
... ],
... index=["True Death", "True Survive"],
... )
              Predict died Predict Survive
True Death 199 25
True Survive 46 123
```

O Yellowbrick tem um gráfico para a matriz de confusão (veja a Figura 12.2):

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
... ConfusionMatrix,
... )
>>> mapping = {0: "died", 1: "survived"}
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cm_viz = ConfusionMatrix(
... dt,
... classes=["died", "survived"],
... label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1202.png", dpi=300)
```

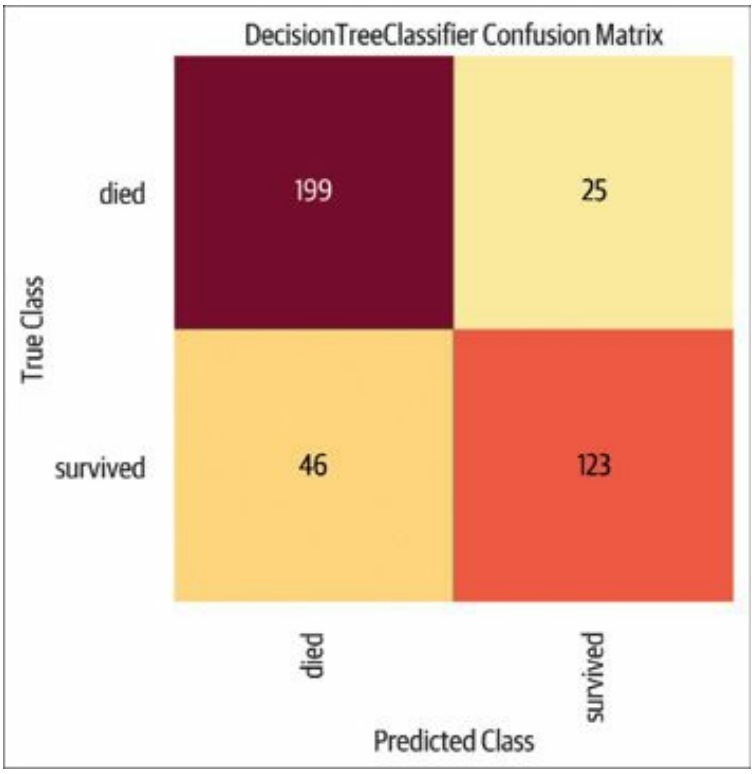


Figura 12.2 – Matriz de confusão. A parte superior à esquerda e a parte inferior à direita são as classificações corretas. A parte inferior à esquerda são os falso-negativos. A parte superior à direita são os falso-positivos.

Métricas

O módulo sklearn.metrics implementa várias métricas muito comuns para classificação, incluindo:

'accuracy'
: Percentual de predições corretas.

'average_precision'
: Resumo da curva de precisão e recall (revocação).

'f1'
: Média harmônica de precisão e recall (revocação).

'neg_log_loss'
: Perda logística ou de entropia cruzada (cross-entropy) (o modelo deve aceitar predict_proba).

'precision'

Capacidade de encontrar somente as amostras relevantes (não atribuir um rótulo positivo a um negativo).

'recall'

Capacidade de encontrar todas as amostras positivas.

'roc_auc'

Área sob a curva ROC (Receiver Operator Characteristic, ou Característica de Operação do Receptor).

Essas strings podem ser usadas no parâmetro scoring em uma busca em grade (grid search), ou você pode usar funções do módulo sklearn.metrics com os mesmos nomes das strings, mas que terminem com _score. Veja as notas a seguir como exemplos.

NOTA

'f1', 'precision' e 'recall' aceitam os sufixos a seguir para classificadores com várias classes (multiclass):

'_micro'

Média ponderada global da métrica.

'_macro'

Média não ponderada da métrica.

'_weighted'

Média ponderada pelas várias classes da métrica.

'_samples'

Métrica por amostra.

Acurácia

A acurácia (accuracy) é a porcentagem de classificações corretas:

```
>>> (tp + tn) / (tp + tn + fp + fn)
0.8142493638676844
```

O que é uma boa acurácia? Isso depende. Se estou fazendo a predição de uma fraude (que, em geral, é um evento raro, por exemplo, 1 em 10 mil), eu

poderia ter uma acurácia muito elevada sempre prevendo que não haveria uma fraude. Porém, esse modelo não seria muito útil. Observar outras métricas e o custo de se prever um falso-positivo e um falso-negativo podem nos ajudar a determinar se um modelo é razoável.

Podemos usar o sklearn para calcular esse valor para nós:

```
>>> from sklearn.metrics import accuracy_score
>>> y_predict = dt.predict(X_test)
>>> accuracy_score(y_test, y_predict)
0.8142493638676844
```

Recall

O recall (revocação) – também conhecido como *sensibilidade* (sensitivity) – é a porcentagem de valores positivos classificados corretamente. (Quantos resultados relevantes são devolvidos?)

```
>>> tp / (tp + fn)
0.7159763313609467

>>> from sklearn.metrics import recall_score
>>> y_predict = dt.predict(X_test)
>>> recall_score(y_test, y_predict)
0.7159763313609467
```

Precisão

A precisão (precision) é a porcentagem de predições positivas que estavam corretas (TP dividido por (TP + FP)). (Quão relevantes são os resultados?)

```
>>> tp / (tp + fp)
0.8287671232876712

>>> from sklearn.metrics import precision_score
>>> y_predict = dt.predict(X_test)
>>> precision_score(y_test, y_predict)
0.8287671232876712
```

F1

F1 é a média harmônica de recall (revocação) e precisão:

```
>>> pre = tp / (tp + fp)
>>> rec = tp / (tp + fn)
>>> (2 * pre * rec) / (pre + rec)
```

```
0.7682539682539683

>>> from sklearn.metrics import f1_score
>>> y_predict = dt.predict(X_test)
>>> f1_score(y_test, y_predict)
0.7682539682539683
```

Relatório de classificação

O Yellowbrick tem um relatório de classificação que mostra as pontuações de precisão, recall e f1, para valores tanto positivos como negativos (veja a Figura 12.3). É um relatório colorido, e, quanto mais vermelha a célula (valor mais próximo de um), melhor será a pontuação:

```
>>> import matplotlib.pyplot as plt
>>> from yellowbrick.classifier import (
... ClassificationReport,
... )
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cm_viz = ClassificationReport(
... dt,
... classes=["died", "survived"],
... label_encoder=mapping,
... )
>>> cm_viz.score(X_test, y_test)
>>> cm_viz.poof()
>>> fig.savefig("images/mlpr_1203.png", dpi=300)
```

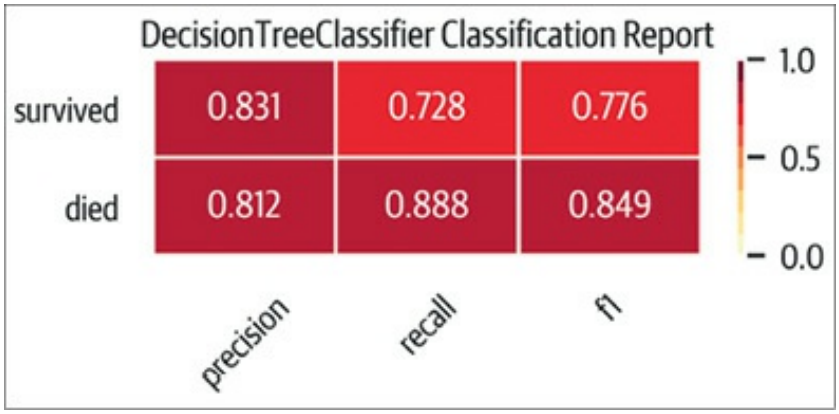


Figura 12.3 – Relatório de classificação.

ROC

A curva ROC mostra o desempenho do classificador, exibindo a taxa de

verdadeiros positivos (recall/sensibilidade) à medida que a taxa de falso-positivos (especificidade invertida) muda (veja a Figura 12.4).

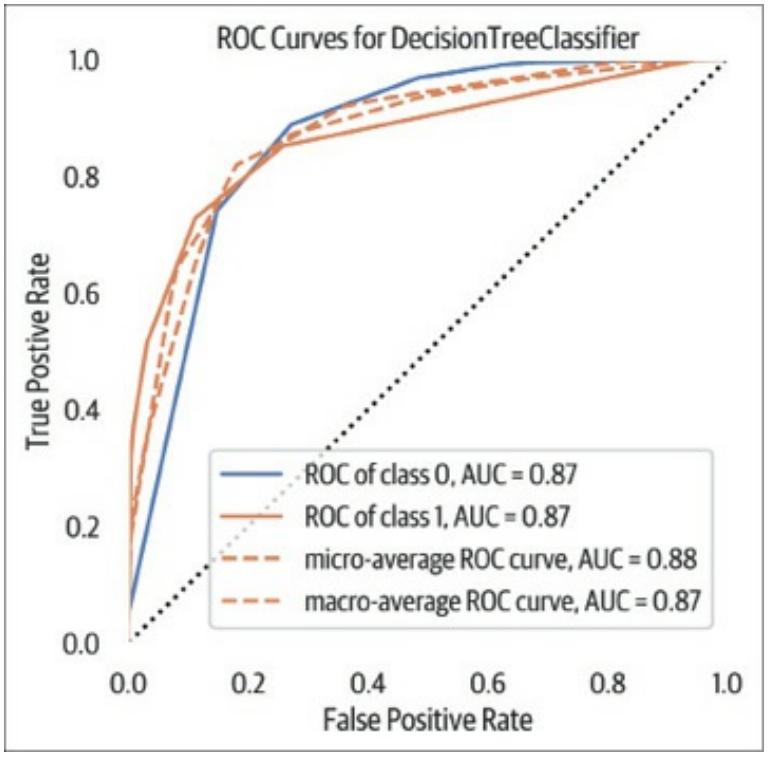


Figura 12.4 – Curva ROC.

Uma regra geral é que o gráfico deve ter uma protuberância em direção ao canto superior esquerdo. Um traçado que esteja à esquerda e acima de outro sinaliza um desempenho melhor. A diagonal nesse gráfico mostra o comportamento de um classificador com palpites aleatórios. Ao considerar o AUC, você terá uma métrica para avaliar o desempenho:

```
>>> from sklearn.metrics import roc_auc_score
>>> y_predict = dt.predict(X_test)
>>> roc_auc_score(y_test, y_predict)
0.8706304346418559
```

O Yellowbrick é capaz de gerar esse gráfico para nós:

```
>>> from yellowbrick.classifier import ROCAUC
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> roc_viz = ROCAUC(dt)
>>> roc_viz.score(X_test, y_test)
0.8706304346418559
>>> roc_viz.poof()
>>> fig.savefig("images/mlpr_1204.png", dpi=300)
```

Curva de precisão-recall

A curva ROC pode ser otimista demais para classes desbalanceadas. Outra opção para avaliar classificadores é a curva de precisão-recall (veja a Figura 12.5).

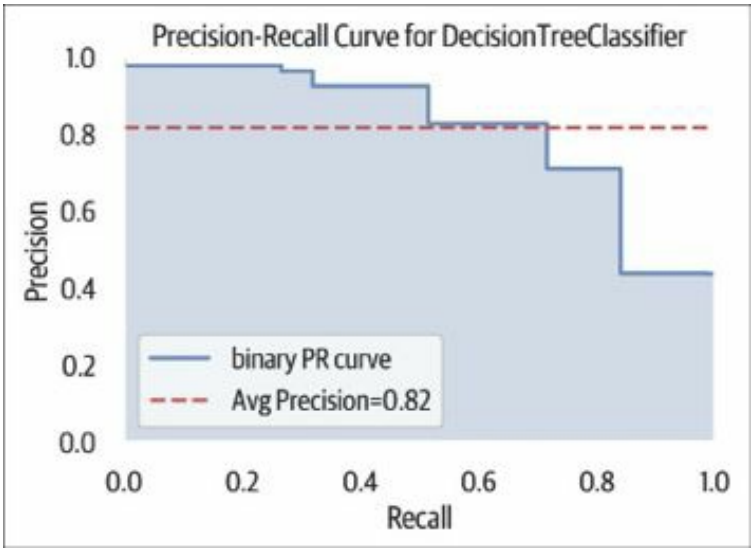


Figura 12.5 – Curva de precisão-recall.

Uma classificação é uma tarefa de balanceamento de modo a encontrar tudo de que você precisa (recall), ao mesmo tempo que limita os resultados ruins (precisão). Em geral, há um compromisso: à medida que o recall aumenta, a precisão diminui – e vice-versa.

```
>>> from sklearn.metrics import (
... average_precision_score,
... )
>>> y_predict = dt.predict(X_test)
>>> average_precision_score(y_test, y_predict)
0.7155150490642249
```

Eis o gráfico de uma curva de precisão-recall gerada com o Yellowbrick:

```
>>> from yellowbrick.classifier import (
... PrecisionRecallCurve,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> viz = PrecisionRecallCurve(
... DecisionTreeClassifier(max_depth=3)
... )
>>> viz.fit(X_train, y_train)
>>> print(viz.score(X_test, y_test))
>>> viz.poof()
```

```
>>> fig.savefig("images/mlpr_1205.png", dpi=300)
```

Gráfico de ganhos cumulativos

Um gráfico de ganhos cumulativos (cumulative gains) pode ser usado para avaliar um classificador binário. Esse gráfico modela a taxa de verdadeiros positivos (sensibilidade) em relação à taxa de suporte (fração das predições positivas). A intuição por trás desse gráfico está em ordenar todas as classificações de acordo com a probabilidade prevista. O ideal é que houvesse uma separação clara dividindo as amostras positivas das negativas. Se os primeiros 10% das predições tiverem 30% das amostras positivas, teríamos um ponto de (0,0) a (0,1, 0,3). Você continuaria esse processo por todas as amostras (veja a Figura 12.6).

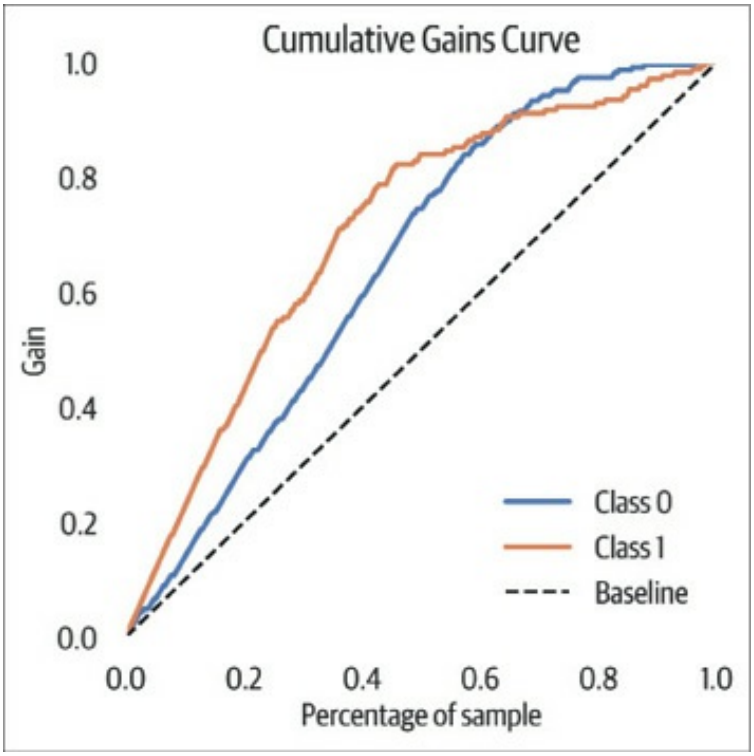


Figura 12.6 – Gráfico de ganhos cumulativos. Se ordenássemos as pessoas do Titanic com base em nosso modelo e observássemos 20% delas, teríamos 40% de sobreviventes.

Um uso comum desse gráfico é determinar a resposta dos consumidores. Os gráficos de ganhos cumulativos exibem o suporte, isto é, a taxa de predições positivas, ao longo do eixo x. Nosso gráfico nomeia essa informação como “Percentage of sample” (Porcentagem da amostra). Ele mostra a

sensibilidade, ou a taxa de verdadeiros positivos, ao longo do eixo y. Em nosso gráfico, essa informação é chamada de “Gain” (Ganho).

Se quisesse entrar em contato com 90% dos consumidores que iriam responder (sensibilidade), você poderia traçar de 0,9 no eixo y para a direita, até atingir essa curva. O eixo x nesse ponto mostrará com quantos consumidores no total seria necessário entrar em contato (suporte) a fim de obter 90%.

Em nosso caso, não estamos entrando em contato com consumidores que responderiam a uma pesquisa, mas prevendo a sobrevivência ao Titanic. Se ordenássemos todos os passageiros do Titanic com base em nosso modelo sobre a probabilidade de sobreviverem e considerássemos os primeiros 65%, teríamos 90% de sobreviventes. Se tivermos um custo associado por contato e uma receita por resposta, seria possível calcular qual é o melhor número.

Em geral, um modelo que estiver à esquerda e acima de outro modelo será melhor. Os melhores modelos são linhas que vão para cima (se 10% das amostras forem positivas, ele atingiria (0,1, 1)) e, então, diretamente para a direita. Se o traçado estiver abaixo da linha de base, será melhor atribuir rótulos aleatoriamente a usar o nosso modelo.

A biblioteca scikit-plot (*https://oreil.ly/dg0iQ*) é capaz de gerar um gráfico de ganhos cumulativos:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
>>> scikitplot.metrics.plot_cumulative_gain(
... y_test, y_probas, ax=ax
... )
>>> fig.savefig(
... "images/mlpr_1206.png",
... dpi=300,
... bbox_inches="tight",
... )
```

Gráfico de elevação

Um gráfico de elevação (lift curve) é outra forma de olhar a informação que está em um gráfico de ganhos cumulativos. A *elevação* (lift) mostra quão melhor é o nosso desempenho em relação ao modelo de base. Em nosso gráfico a seguir, podemos ver que, se ordenássemos os passageiros do Titanic de acordo com a probabilidade de sobrevivência e considerássemos os

primeiros 20%, nossa elevação (lift) seria de aproximadamente 2,2 vezes (o ganho dividido pela porcentagem da amostra) de melhoria em comparação com escolher os sobreviventes de modo aleatório (veja a Figura 12.7). (Teríamos 2,2 vezes mais sobreviventes.)

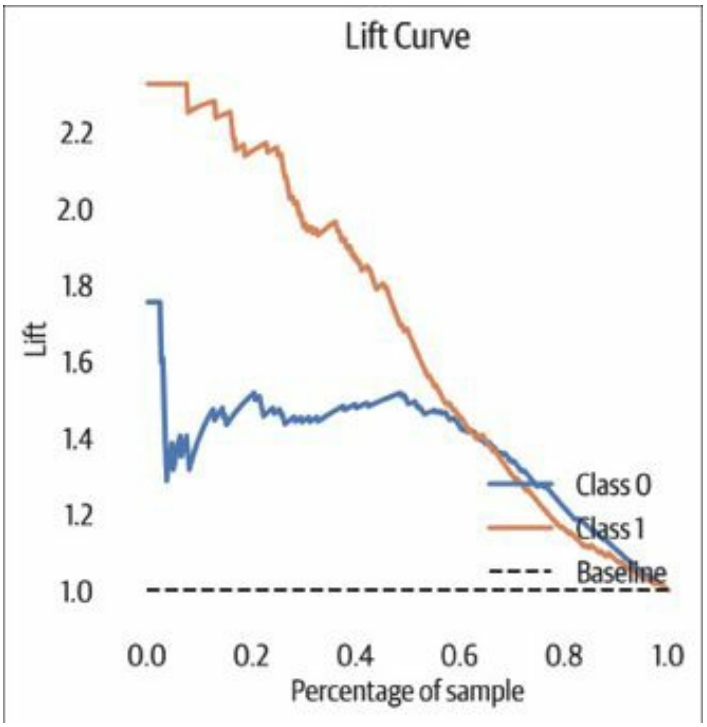


Figura 12.7 – Gráfico de elevação.

A biblioteca scikit-plot é capaz de gerar um gráfico de elevação:

```
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> y_probas = dt.predict_proba(X_test)
>>> scikitplot.metrics.plot_lift_curve(
... y_test, y_probas, ax=ax
... )
>>> fig.savefig(
... "images/mlpr_1207.png",
... dpi=300,
... bbox_inches="tight",
... )
```

Balanceamento das classes

O Yellowbrick tem um gráfico de barras simples para visualizar os tamanhos das classes. Quando os tamanhos relativos das classes são diferentes, a acurácia não é uma boa métrica de avaliação (veja a Figura 12.8).

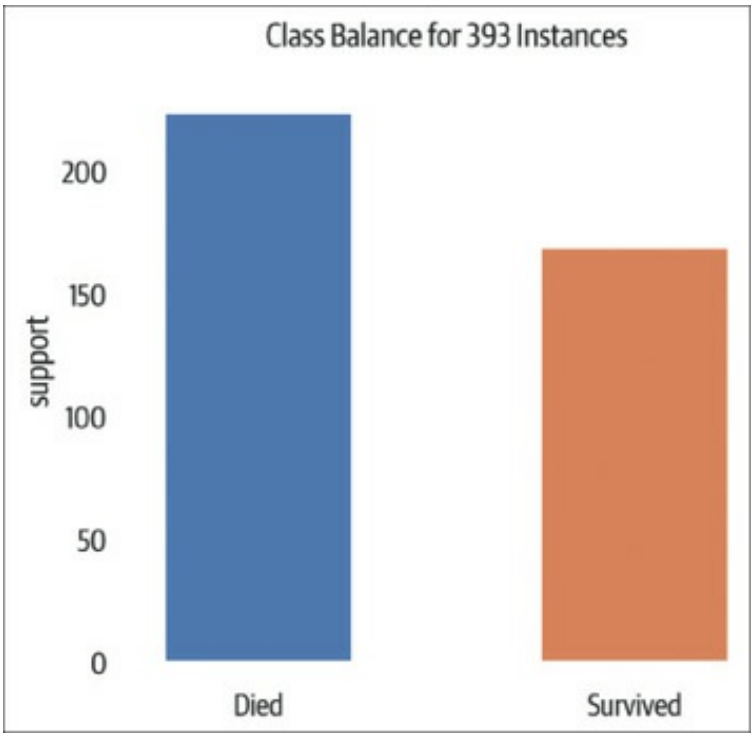


Figura 12.8 – Um pequeno desbalanceamento de classes.

Ao separar os dados em conjuntos de treinamento e de teste, utilize uma *amostragem estratificada*, de modo que os conjuntos mantenham uma proporção relativa das classes. (A função test_train_split faz isso se você definir o parâmetro stratify com os rótulos.)

```
>>> from yellowbrick.classifier import ClassBalance
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> cb_viz = ClassBalance(
... labels=["Died", "Survived"]
... )
>>> cb_viz.fit(y_test)
>>> cb_viz.poof()
>>> fig.savefig("images/mlpr_1208.png", dpi=300)
```

Erro de predição de classe

O gráfico de erro de predição de classe do Yellowbrick é um gráfico de barras que exibe uma matriz de confusão (veja a Figura 12.9):

```
>>> from yellowbrick.classifier import (
... ClassPredictionError,
... )
>>> fig, ax = plt.subplots(figsize=(6, 3))
>>> cpe_viz = ClassPredictionError(
```

```
... dt, classes=["died", "survived"]
... )
>>> cpe_viz.score(X_test, y_test)
>>> cpe_viz.poof()
>>> fig.savefig("images/mlpr_1209.png", dpi=300)
```

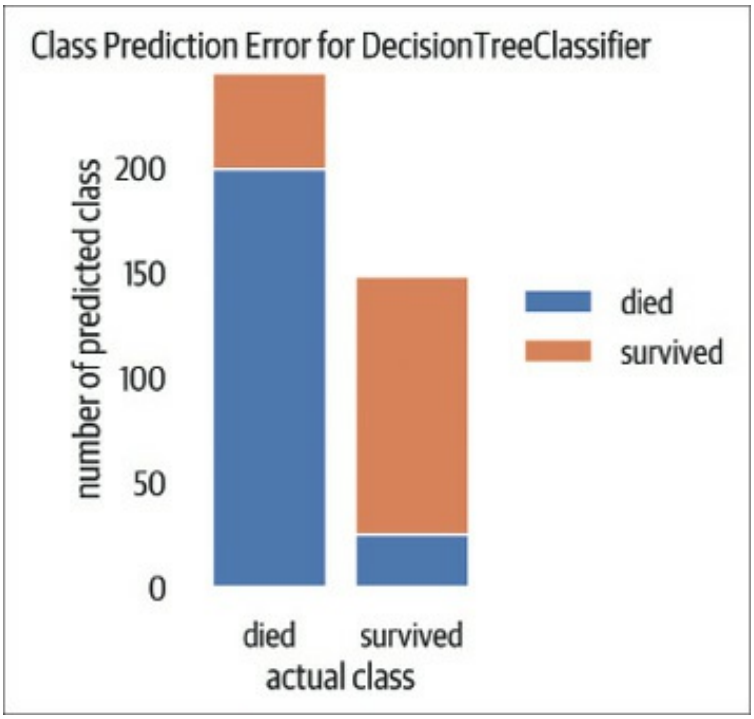


Figura 12.9 – Erro de predição de classe. Na parte superior da barra à esquerda estão as pessoas que faleceram, mas para as quais foi previsto que sobreviveram (falso-positivos). Na parte inferior da barra à direita estão as pessoas que sobreviveram, mas para as quais o modelo fez a predição de que faleceram (falso-negativos).

Limiar de discriminação

A maioria dos classificadores binários que fazem a predição de probabilidades tem um *limiar de discriminação* (discrimination threshold) de 50%. Se a probabilidade prevista estiver acima de 50%, o classificador atribuirá um rótulo positivo. A Figura 12.10 varia esse valor de limiar entre 0 e 100 e mostra o impacto na precisão, no recall (revocação), em f1 e na taxa de fila (queue rate).

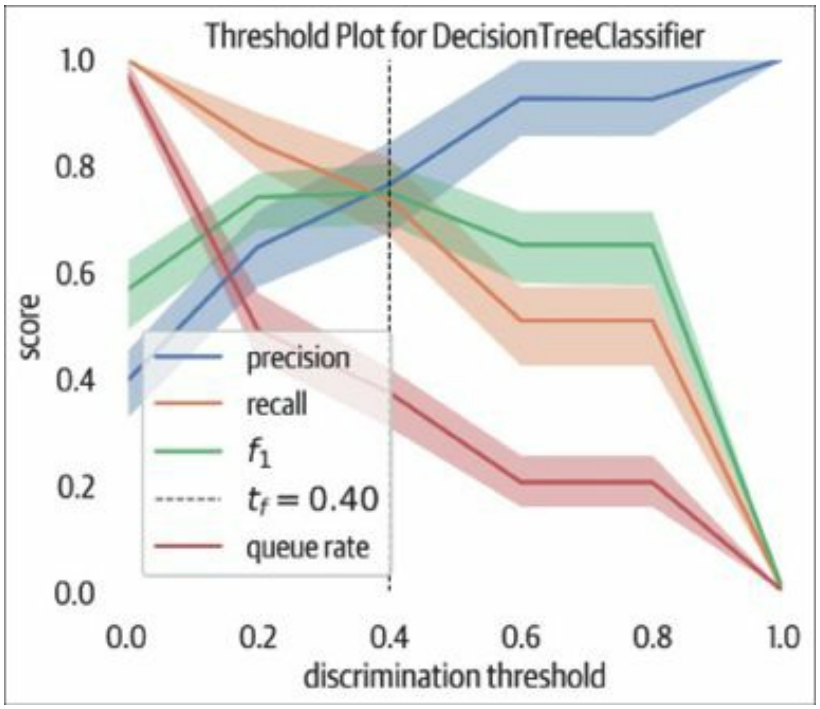


Figura 12.10 – Limiar de discriminação.

Esse gráfico pode ser conveniente para visualizar a relação de compromisso entre precisão e recall. Suponha que estamos procurando uma fraude (e considerando a fraude como a classificação positiva). Para ter um valor alto de recall (identificar todas as fraudes), podemos simplesmente classificar tudo como fraude. Porém, em uma situação envolvendo um banco, isso não compensaria, e exigiria um exército de funcionários. Para ter uma precisão alta (identificar uma fraude somente se for uma fraude), poderíamos ter um modelo que identificasse somente casos de fraude extrema. No entanto, isso faria com que deixássemos de identificar várias fraudes que poderiam não estar tão óbvias. Há um compromisso nesse caso.

A *taxa de fila* (queue rate) é a porcentagem de previsões acima do limiar. Podemos considerá-la como a porcentagem de casos a serem analisados se você estivesse lidando com fraudes.

Se você tiver o custo para cálculos positivos, negativos e errôneos, é possível determinar o limiar com o qual você se sentiria confortável.

O gráfico a seguir é conveniente para verificar qual é o limiar de discriminação que maximizará a pontuação f1 ou para ajustar a precisão ou o recall com um número aceitável quando combinados com a taxa de fila.

O Yellowbrick disponibiliza esse visualizador. Ele embaralha os dados e

executa 50 tentativas por padrão, separando 10% para validação:

```
>>> from yellowbrick.classifier import (
... DiscriminationThreshold,
... )
>>> fig, ax = plt.subplots(figsize=(6, 5))
>>> dt_viz = DiscriminationThreshold(dt)
>>> dt_viz.fit(X, y)
>>> dt_viz.poof()
>>> fig.savefig("images/mlpr_1210.png", dpi=300)
```

CAPÍTULO 13

Explicando os modelos

Modelos preditivos têm diferentes propriedades. Alguns foram concebidos para lidar com dados lineares; outros são capazes de modelar dados de entrada mais complexos. Alguns modelos podem ser muito facilmente interpretados, enquanto outros são como caixas-pretas e não oferecem muitos insights acerca de como a predição é feita.

Neste capítulo, veremos como interpretar diferentes modelos. Analisaremos alguns exemplos usando os dados do Titanic.

```
>>> dt = DecisionTreeClassifier(
... random_state=42, max_depth=3
... )
>>> dt.fit(X_train, y_train)
```

Coeficientes de regressão

Os interceptos e os coeficientes de regressão explicam o valor esperado e como os atributos causam impacto na predição. Um coeficiente positivo indica que, à medida que o valor de um atributo aumenta, o mesmo ocorrerá com a predição.

Importância dos atributos

Modelos baseados em árvore da biblioteca scikit-learn incluem um atributo .feature_importances_ para inspecionar como os atributos de um conjunto de dados afetam o modelo. Podemos inspecionar esses dados ou colocá-los em um gráfico.

LIME

O LIME (*https://oreil.ly/shCR_*) ajuda a explicar modelos caixa-preta. Ele faz uma interpretação *local*, em vez de uma interpretação geral, e ajuda a explicar

uma única amostra.

Para um determinado ponto de dado ou amostra, o LIME mostra quais atributos foram importantes para determinar o resultado. Ele faz isso causando uma perturbação na amostra em questão e fazendo a adequação de um modelo linear para ela. O modelo linear aproxima o modelo à amostra (veja a Figura 13.1).

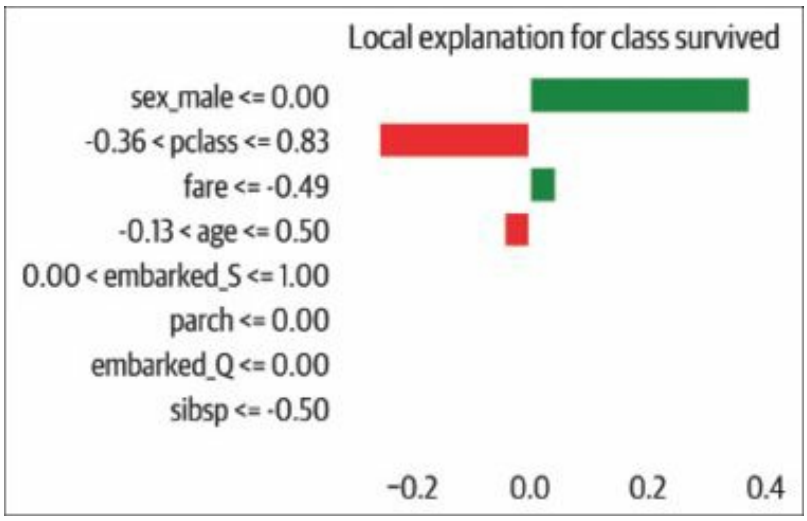


Figura 13.1 – Explicação do LIME para o conjunto de dados do Titanic. Os atributos da amostra empurram a previsão para a direita (sobrevivência) ou para a esquerda (falecimento).

Eis um exemplo que explica a última amostra (para a qual nossa árvore de decisão fez a previsão de que sobreviveria) dos dados de treinamento:

```
>>> from lime import lime_tabular
>>> explainer = lime_tabular.LimeTabularExplainer(
... X_train.values,
... feature_names=X.columns,
... class_names=["died", "survived"],
... )
>>> exp = explainer.explain_instance(
... X_train.iloc[-1].values, dt.predict_proba
... )
```

O LIME não gosta de usar DataFrames como entrada. Observe que convertemos os dados em arrays numpy usando .values.

DICA

Se você estiver fazendo isso no Jupyter, utilize o código a seguir:

```
exp.show_in_notebook()
```

Uma versão HTML contendo a explicação será renderizada.

Podemos criar uma figura com a matplotlib se quisermos exportar a explicação (ou se não estivermos usando o Jupyter):

```
>>> fig = exp.as_pyplot_figure()
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1301.png")
```

Brinque com esses dados e observe que, se você trocar o sexo, o resultado será afetado. A seguir, consideramos a antepenúltima linha dos dados de treinamento. A predição para essa linha é de 48% para falecimento e 52% para sobrevivência. Se trocarmos o sexo, veremos que a predição muda para 88% de falecimento:

```
>>> data = X_train.iloc[-2].values.copy()
>>> dt.predict_proba(
... [data]
... ) # previsão de que uma mulher sobreviva
[[0.48062016 0.51937984]]
>>> data[5] = 1 # muda para o sexo masculino
>>> dt.predict_proba([data])
array([[0.87954545, 0.12045455]])
```

NOTA

O método .predict_proba devolve uma probabilidade para cada rótulo.

Interpretação de árvores

Para modelos baseados em árvore do sklearn (árvore de decisão, floresta aleatória e outros modelos de árvore), podemos usar o pacote treeinterpreter (*https://oreil.ly/vN1Bl*). Ele calculará o bias (viés) e a contribuição de cada atributo. O bias é a média do conjunto de treinamento.

Cada contribuição lista como o atributo contribui com cada um dos rótulos. (A soma do bias mais as contribuições deve ser igual à predição.) Como essa é uma classificação binária, há apenas dois rótulos. Vemos que sex_male é o atributo mais importante, seguido de age (idade) e fare (preço da passagem):

```
>>> from treeinterpreter import (
... treeinterpreter as ti,
... )
>>> instances = X.iloc[:2]
```

```
>>> prediction, bias, contribs = ti.predict(
... rf5, instances
... )
>>> i = 0
>>> print("Instance", i)
>>> print("Prediction", prediction[i])
>>> print("Bias (trainset mean)", bias[i])
>>> print("Feature contributions:")
>>> for c, feature in zip(
... contribs[i], instances.columns
... ):
... print(" {} {}".format(feature, c))
Instance 0
Prediction [0.98571429 0.01428571]
Bias (trainset mean) [0.63984716 0.36015284]
Feature contributions:
  pclass [ 0.03588478 -0.03588478]
  age [ 0.08569306 -0.08569306]
  sibsp [ 0.01024538 -0.01024538]
  parch [ 0.0100742 -0.0100742]
  fare [ 0.06850243 -0.06850243]
  sex_male [ 0.12000073 -0.12000073]
  embarked_Q [ 0.0026364 -0.0026364]
  embarked_S [ 0.01283015 -0.01283015]
```

NOTA

Esse exemplo serve para classificação, mas há suporte também para regressão.

Gráficos de dependência parcial

Com a importância dos atributos em árvores, sabemos que um atributo causa impactos no resultado, mas não sabemos como o impacto varia à medida que o valor do atributo muda. Os gráficos de dependência parcial nos permitem visualizar a relação entre mudanças em apenas um atributo e o resultado. Usaremos o pdpbox (*https://oreil.ly/O9zY2*) para visualizar como a idade (age) afeta a sobrevivência (veja a Figura 13.2).

Este exemplo utiliza um modelo de floresta aleatória:

```
>>> rf5 = ensemble.RandomForestClassifier(
... **{
... "max_features": "auto",
... "min_samples_leaf": 0.1,
```

```
... "n_estimators": 200,
... "random_state": 42,
... }
... )
>>> rf5.fit(X_train, y_train)
>>> from pdpbox import pdp
>>> feat_name = "age"
>>> p = pdp.pdp_isolate(
... rf5, X, X.columns, feat_name
... )
>>> fig, _ = pdp.pdp_plot(
... p, feat_name, plot_lines=True
... )
>>> fig.savefig("images/mlpr_1302.png", dpi=300)
```

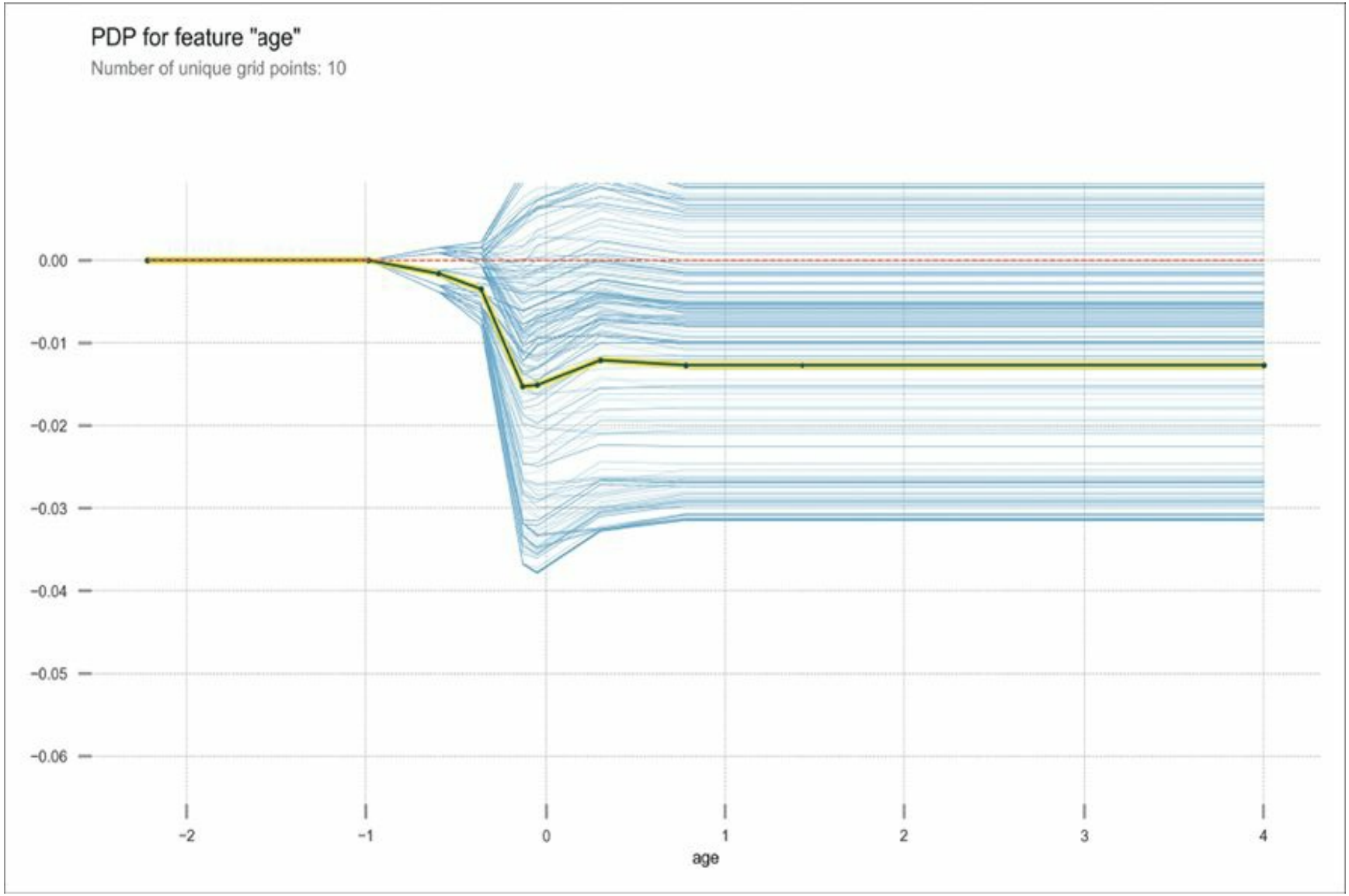

PDP for feature "age"
Number of unique grid points: 10
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
−0.06
−2
−1
0
1
2
3
4
age

Figura 13.2 – Gráfico de dependência parcial mostrando o que acontece com o alvo à medida que a idade (age) varia.

Podemos visualizar também as interações entre dois atributos (veja a Figura 13.3):

```
>>> features = ["fare", "sex_male"]
>>> p = pdp.pdp_interact(
... rf5, X, X.columns, features
... )
>>> fig, _ = pdp.pdp_interact_plot(p, features)
>>> fig.savefig("images/mlpr_1303.png", dpi=300)
```

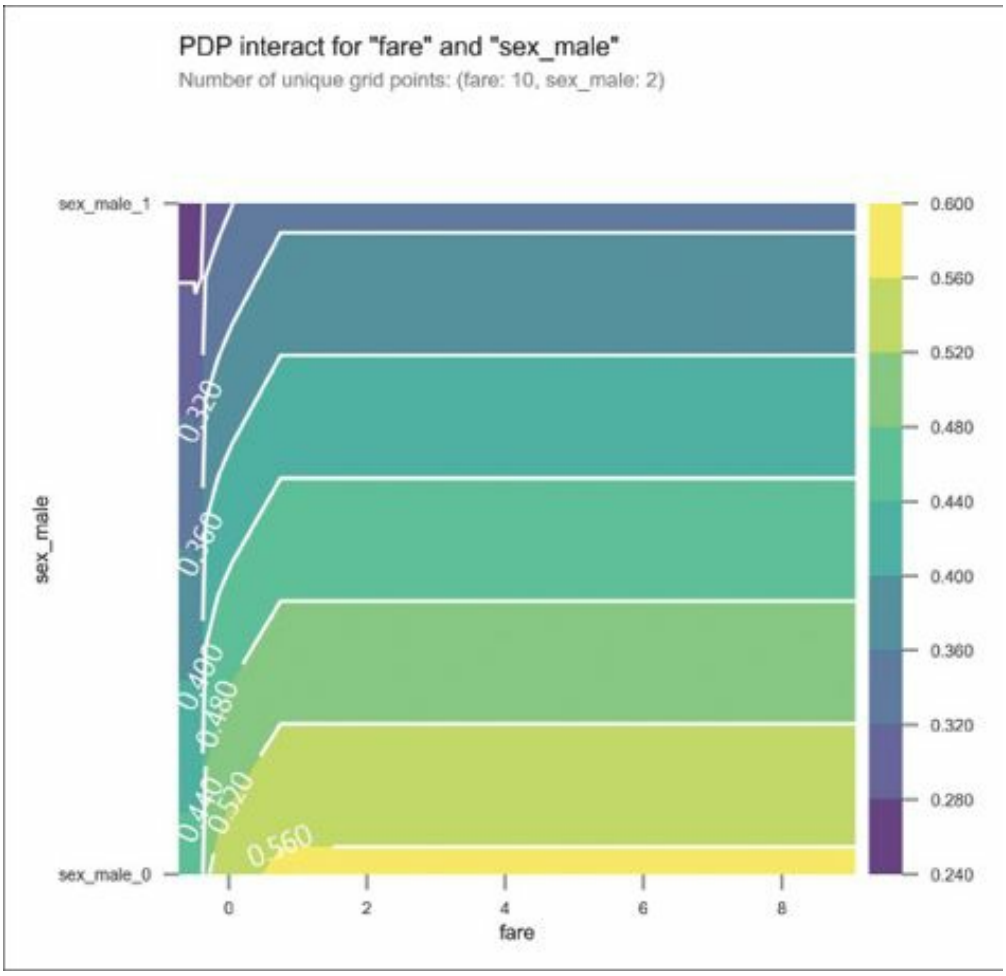


Figura 13.3 – Gráfico de dependência parcial com dois atributos. À medida que fare (preço da passagem) aumenta e o sexo muda de masculino (male) para feminino (female), a sobrevivência aumenta.

NOTA

O gráfico de dependência parcial fixa o valor de um atributo nas amostras e então calcula a média do resultado. (Tome cuidado com valores discrepantes (outliers) e médias.) Além disso, esse gráfico pressupõe que os atributos são independentes. (Nem sempre será o caso; por exemplo, considerar a largura de uma sépala como fixa provavelmente teria um efeito na altura.) A biblioteca pdpbox também exibe as expectativas condicionais individuais para uma melhor visualização desses relacionamentos.

Modelos substitutos

Se você tiver um modelo que não seja interpretável (SVM ou rede neural), poderá fazer a adequação de um modelo interpretável (árvore de decisão) para esse modelo. Ao usar o modelo substituto (surrogate model), poderá analisar a importância dos atributos.

Em nosso exemplo, criaremos um SVC (Support Vector Classifier, ou Classificador de Vetor Suporte), mas faremos o treinamento de uma árvore de decisão (sem um limite de profundidade para superadequação e para capturar o que acontece nesse modelo) a fim de explicá-lo:

```
>>> from sklearn import svm
>>> sv = svm.SVC()
>>> sv.fit(X_train, y_train)
>>> sur_dt = tree.DecisionTreeClassifier()
>>> sur_dt.fit(X_test, sv.predict(X_test))
>>> for col, val in sorted(
... zip(
... X_test.columns,
... sur_dt.feature_importances_,
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:7]:
... print(f"{col:10}{val:10.3f}")
sex_male 0.723
pclass 0.076
sibsp 0.061
age 0.056
embarked_S 0.050
fare 0.028
parch 0.005
```

Shapley

O pacote SHAP (SHapley Additive exPlanations, *https://oreil.ly/QYj-q*) é capaz de exibir as contribuições dos atributos para qualquer modelo. É um pacote realmente muito bom, pois não só funciona para a maioria dos modelos como também é capaz de explicar predições individuais e as contribuições globais dos atributos.

O SHAP funciona tanto para classificação como para regressão e gera valores "SHAP". Para modelos de classificação, o valor SHAP é a soma dos log odds (logaritmo das chances) em uma classificação binária. Para regressão, os valores SHAP são a soma da predição do alvo.

Essa biblioteca exige o Jupyter (JavaScript) para interatividade em alguns de seus gráficos. (Alguns podem renderizar imagens estáticas com a matplotlib.) Eis um exemplo para a amostra 20, cuja previsão é de falecimento:

```
>>> rf5.predict_proba(X_test.iloc[[20]])
array([[0.59223553, 0.40776447]])
```

No gráfico de forças (force plot) para a amostra 20, podemos ver o "valor de base" (base value). É uma mulher, e a previsão é de que faleceria (veja a Figura 13.4).

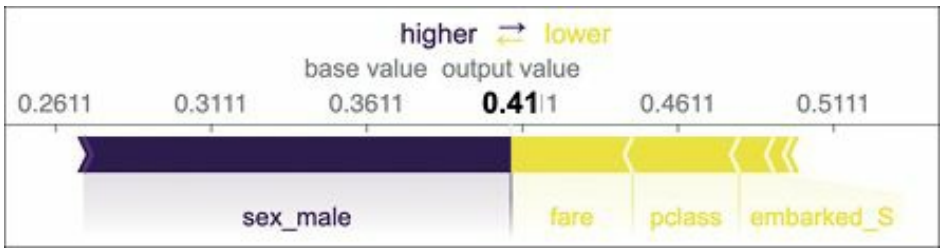


Figura 13.4 – Contribuições dos atributos no Shapley para a amostra 20. Esse gráfico mostra o valor de base e os atributos que pressionam em direção ao falecimento.

Usaremos o índice de sobrevivência (1) porque queremos que o lado direito do gráfico seja a sobrevivência. Os atributos forçam isso para a direita ou para a esquerda. Quanto maior o atributo, mais impacto ele causará. Neste caso, um valor baixo de fare (preço da passagem) e a terceira classe (third class) pressionam em direção ao falecimento (o valor do resultado está abaixo de 0,5):

```
>>> import shap
>>> s = shap.TreeExplainer(rf5)
```

```
>>> shap_vals = s.shap_values(X_test)
>>> target_idx = 1
>>> shap.force_plot(
... s.expected_value[target_idx],
... shap_vals[target_idx][20, :],
... feature_names=X_test.columns,
... )
```

Você também pode visualizar as explicações para o conjunto de dados completo (fazendo uma rotação de 90 graus e a plotagem no eixo x) (veja a Figura 13.5):

```
>>> shap.force_plot(
... s.expected_value[1],
... shap_vals[1],
... feature_names=X_test.columns,
... )
```

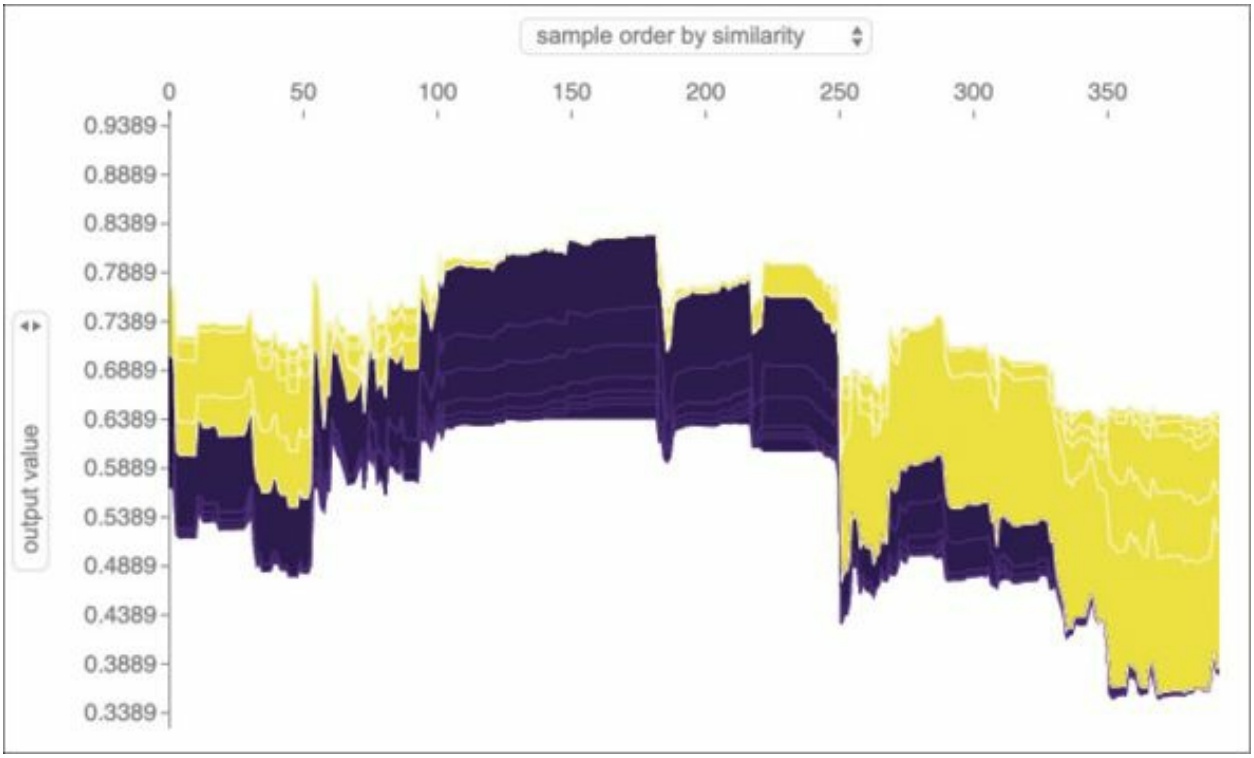


Figura 13.5 – Contribuições dos atributos no Shapley para o conjunto de dados completo.

A biblioteca SHAP também pode gerar gráficos de dependência. O gráfico a seguir (veja a Figura 13.6) mostra o relacionamento entre a idade (age) e o valor SHAP (está colorido de acordo com pclass, que o SHAP seleciona automaticamente; especifique um nome de coluna no parâmetro interaction_index

para escolher o seu próprio atributo):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> res = shap.dependence_plot(
... "age",
... shap_vals[target_idx],
... X_test,
... feature_names=X_test.columns,
... alpha=0.7,
... )
>>> fig.savefig(
... "images/mlpr_1306.png",
... bbox_inches="tight",
... dpi=300,
... )
```

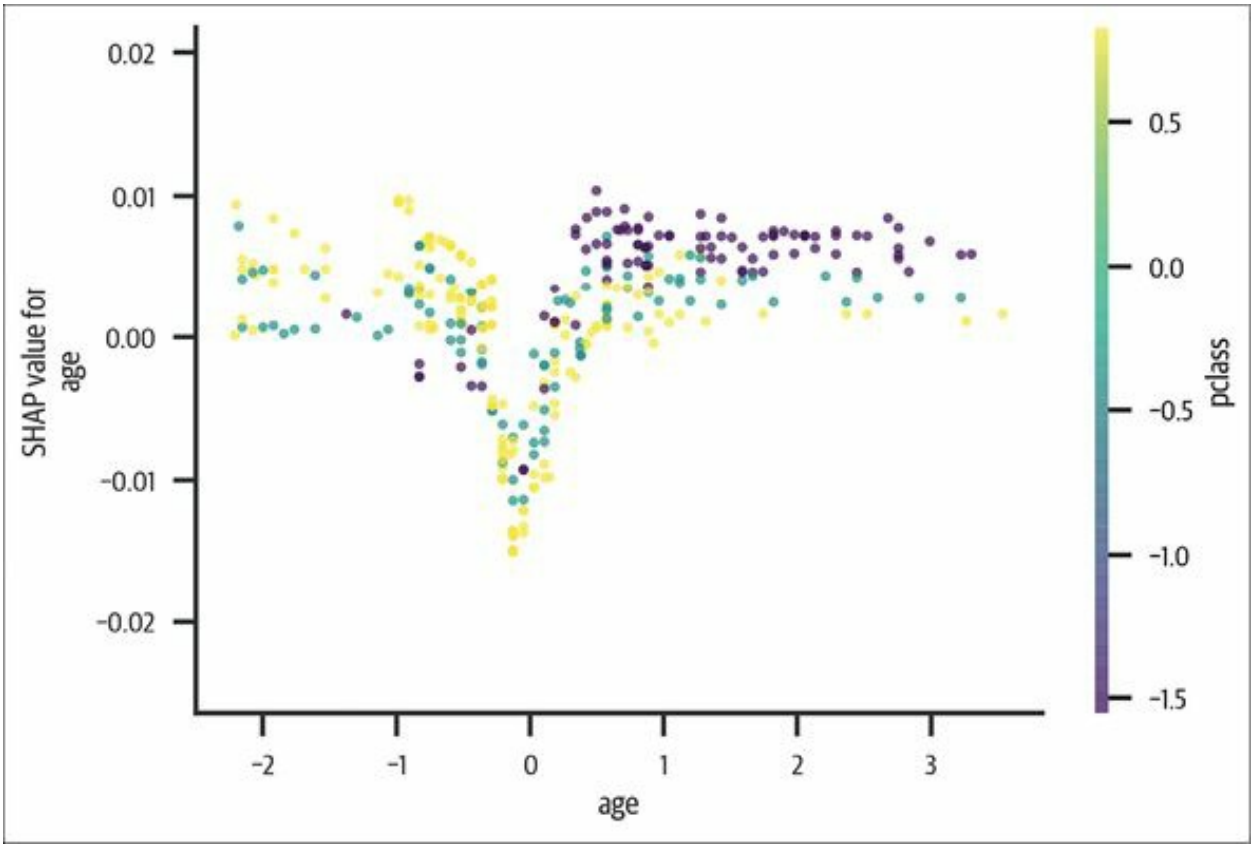


Figura 13.6 – Gráfico de dependências do Shapley para age (idade). Os mais jovens e os mais velhos têm uma taxa maior de sobrevivência. À medida que a idade aumenta, um pclasse menor tem mais chances de sobrevivência.

DICA

Você poderia obter um gráfico de dependência com linhas verticais.

Definir o parâmetro x_jitter com 1 será conveniente se você estiver visualizando atributos de categoria ordinais.

Além disso, podemos fazer uma síntese de todos os atributos. Esse é um gráfico muito eficaz para a compreensão dos dados. Ele mostra o impacto global, mas também os impactos individuais. Os atributos são classificados segundo a importância, e os mais importantes estão na parte superior.

Além disso, os atributos recebem cores de acordo com seus valores. Podemos ver que uma pontuação baixa para sex_male (female, ou seja, sexo feminino) pressiona mais em direção à sobrevivência, enquanto uma pontuação alta exerce menos pressão. O atributo age (idade) é um pouco mais difícil de ser interpretado. Isso porque valores para mais jovens e mais velhos pressionam em direção à sobrevivência, enquanto idades medianas pressionam em direção ao falecimento.

Ao combinar o gráfico de síntese com o gráfico de dependências, você poderá ter bons insights sobre o comportamento do modelo (veja a Figura 13.7):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(shap_vals[0], X_test)
>>> fig.savefig("images/mlpr_1307.png", dpi=300)
```

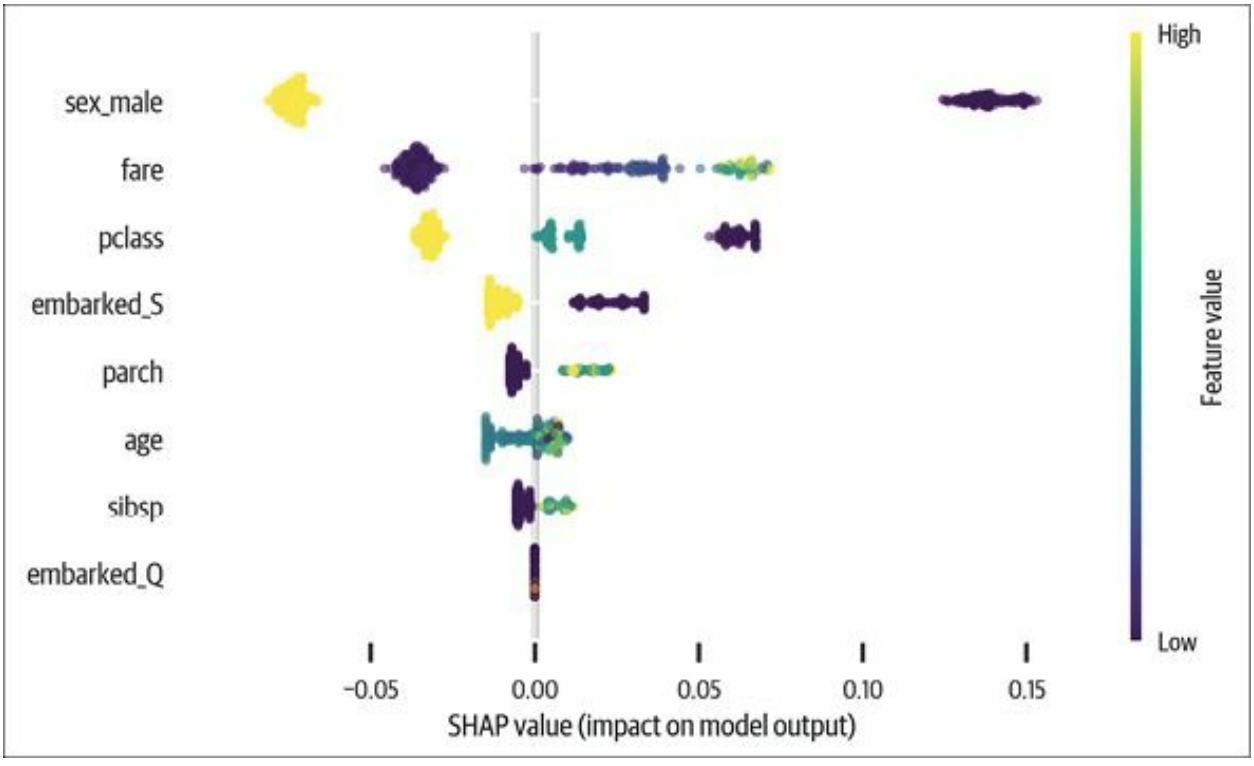


Figura 13.7 – Gráfico de síntese do Shapley mostrando os atributos mais importantes na parte superior. As cores mostram como os valores dos atributos afetam o alvo.

CAPÍTULO 14

Regressão

A regressão é um processo de machine learning supervisionado. É semelhante à classificação, mas, em vez de prever um rótulo, tentamos prever um valor contínuo. Se você estiver tentando prever um número, utilize a regressão.

O fato é que o sklearn é capaz de aplicar muitos dos mesmos modelos de classificação em problemas de regressão. Com efeito, a API é a mesma e chama .fit, .score e .predict. Isso também vale para as bibliotecas de boosting de próxima geração, como XGBoost e LightGBM.

Embora haja semelhanças quanto aos modelos de classificação e os hiperparâmetros, as métricas de avaliação são diferentes no caso de uma regressão. Neste capítulo, analisaremos vários tipos de modelos de regressão. Usaremos o conjunto de dados habitacionais de Boston (*https://oreil.ly/b2bKQ*) para explorá-los.

A seguir, carregaremos os dados, criaremos uma versão com os dados de treinamento e de teste separados e outra versão diferente com os dados padronizados:

```
>>> import pandas as pd
>>> from sklearn.datasets import load_boston
>>> from sklearn import (
... model_selection,
... preprocessing,
... )
>>> b = load_boston()
>>> bos_X = pd.DataFrame(
... b.data, columns=b.feature_names
... )
>>> bos_y = b.target

>>> bos_X_train, bos_X_test, bos_y_train, bos_y_test = model_selection.train_test_split(
... bos_X,
```

```
... bos_y,
... test_size=0.3,
... random_state=42,
... )


>>> bos_sX = preprocessing.StandardScaler().fit_transform(
... bos_X
... )
>>> bos_sX_train, bos_sX_test, bos_sy_train, bos_sy_test = model_selection.train_test_split(
... bos_sX,
... bos_y,
... test_size=0.3,
... random_state=42,
... )
```

Eis a descrição dos atributos do conjunto de dados habitacionais:

CRIM

Taxa de crimes per capita por cidade.

ZN

Proporção de áreas residenciais zoneadas para lotes acima de 25 mil pés quadrados (aproximadamente 2.320 metros quadrados).

INDUS

Proporção de acres para negócios não ligados ao varejo por cidade.

CHAS

Variável dummy sobre o Rio Charles (1 se a região faz fronteira com o rio; 0 caso contrário).

NOX

Concentração de óxidos nítricos (partes por 10 milhões).

RM

Número médio de cômodos por habitação.

AGE

Proporção de unidades ocupadas por proprietários construídas antes de 1940.

DIS

Distâncias ponderadas até cinco centros de empregos em Boston.

RAD

Índice de acessibilidade às rodovias radiais.

TAX

Taxa de impostos sobre o valor total da propriedade por 10 mil dólares.

PTRATIO

Razão entre aluno-professor por cidade.

B

1000(Bk - 0.63)^2, em que Bk é a proporção de negros (Bk = Black) por cidade (esse conjunto de dados é de 1978).

LSTAT

Porcentagem da população com status mais baixo.

MEDV

Valor médio das casas ocupadas por proprietários em incrementos de 1000 dólares.

Modelo de base

Um modelo de regressão básico nos dará algo com o qual poderemos comparar outros modelos. No sklearn, o resultado default do método .score é o *coeficiente de determinação* (r^2 ou R^2). Esse número explica o percentual de variação dos dados de entrada capturado pela predição. Em geral, o valor estará entre 0 e 1, mas pode ser negativo no caso de modelos particularmente ruins.

A estratégia default de DummyRegressor é prever o valor médio do conjunto de treinamento. Podemos ver que esse modelo não tem um bom desempenho:

```
>>> from sklearn.dummy import DummyRegressor
>>> dr = DummyRegressor()
>>> dr.fit(bos_X_train, bos_y_train)
>>> dr.score(bos_X_test, bos_y_test)
-0.03469753992352409
```

Regressão linear

Uma regressão linear simples é ensinada em cursos de matemática e em cursos introdutórios de estatística. Ela tenta fazer a adequação da fórmula y =

mx + b, ao mesmo tempo que minimiza o quadrado dos erros. Quando resolvida, temos um intercepto e um coeficiente. O intercepto nos dá um valor de base para uma predição, modificado pela soma do produto entre o coeficiente e o dado de entrada.

Esse formato pode ser generalizado para dimensões maiores. Nesse caso, cada atributo terá um coeficiente. Quanto maior o valor absoluto do coeficiente, mais impacto terá o atributo no alvo.

Esse modelo pressupõe que a predição é uma combinação linear dos dados de entrada. Para alguns conjuntos de dados, isso não será suficientemente flexível. Mais complexidade poderá ser acrescentada por meio da transformação dos atributos (o transformador `preprocessing.PolynomialFeatures` do sklearn é capaz de criar combinações polinomiais dos atributos). Se isso resultar em superadequação, regressões ridge e lasso poderão ser usadas para regularizar o estimador.

Esse modelo também é suscetível a *heterocedasticidade*. A heterocedasticidade é a ideia de que, à medida que os valores de entrada mudam, o erro da predição (ou resíduos) geralmente muda também. Se você colocar os dados de entrada e os resíduos em um gráfico, verá um formato de leque ou de cone. Veremos exemplos disso mais adiante.

Outra questão que deve ser considerada é a *multicolinearidade*. Se as colunas tiverem um alto nível de correlação, será mais difícil interpretar os coeficientes. Em geral, isso não causará impactos no modelo, mas apenas no significado dos coeficientes.

Um modelo de regressão linear tem as seguintes propriedades:

Eficiência na execução

Use `n_jobs` para melhorar o desempenho.

Pré-processamento dos dados

Padronize os dados antes de fazer o treinamento do modelo.

Para evitar uma superadequação

Você pode simplificar o modelo não usando ou adicionando atributos polinomiais.

Interpretação dos resultados

É possível interpretar os resultados como pesos para contribuição dos

atributos, mas supõe-se que os atributos tenham uma distribuição normal e sejam independentes. Você pode remover atributos colineares para facilitar a interpretação. R^2 informará até que ponto a variância total do resultado é explicada pelo modelo.

Eis um exemplo de execução com os dados default:

```
>>> from sklearn.linear_model import (
... LinearRegression,
... )
>>> lr = LinearRegression()
>>> lr.fit(bos_X_train, bos_y_train)
LinearRegression(copy_X=True, fit_intercept=True,
  n_jobs=1, normalize=False)
>>> lr.score(bos_X_test, bos_y_test)
0.7109203586326287
>>> lr.coef_
array([-1.32774155e-01, 3.57812335e-02,
  4.99454423e-02, 3.12127706e+00,
  -1.54698463e+01, 4.04872721e+00,
  -1.07515901e-02, -1.38699758e+00,
  2.42353741e-01, -8.69095363e-03,
  -9.11917342e-01, 1.19435253e-02,
  -5.48080157e-01])
```

Parâmetros da instância:

n_jobs=None

Número de CPUs a ser usadas. -1 se todas.

Atributos após a adequação:

coef_

Coeficientes da regressão linear.

intercept_

Intercepto do modelo linear.

O valor de .intercept_ é o valor médio esperado. Podemos ver como o fato de escalar os dados afeta os coeficientes. O sinal dos coeficientes explica a direção da relação entre o atributo e o alvo (target). Um sinal positivo informa que, à medida que o atributo aumenta, o rótulo (label) aumenta. Um sinal negativo mostra que, à medida que o atributo aumenta, o rótulo diminui. Quanto maior o valor absoluto do coeficiente, mais impacto ele causará:

```
>>> lr2 = LinearRegression()
>>> lr2.fit(bos_sX_train, bos_sy_train)
LinearRegression(copy_X=True, fit_intercept=True,
  n_jobs=1, normalize=False)
>>> lr2.score(bos_sX_test, bos_sy_test)
0.7109203586326278
>>> lr2.intercept_
22.50945471291039
>>> lr2.coef_
array([-1.14030209, 0.83368112, 0.34230461,
  0.792002, -1.7908376, 2.84189278, -0.30234582,
  -2.91772744, 2.10815064, -1.46330017,
  -1.97229956, 1.08930453, -3.91000474])
```

Podemos usar o Yellowbrick para visualizar os coeficientes (veja a Figura 14.1). Como os dados de Boston escalados não são um DataFrame pandas, mas um array numpy, é necessário passar o parâmetro labels se quisermos usar os nomes das colunas:

```
>>> from yellowbrick.features import (
... FeatureImportances,
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(
... lr2, labels=bos_X.columns
... )
>>> fi_viz.fit(bos_sX, bos_sy)
>>> fi_viz.poof()
>>> fig.savefig(
... "images/mlpr_1401.png",
... bbox_inches="tight",
... dpi=300,
... )
```

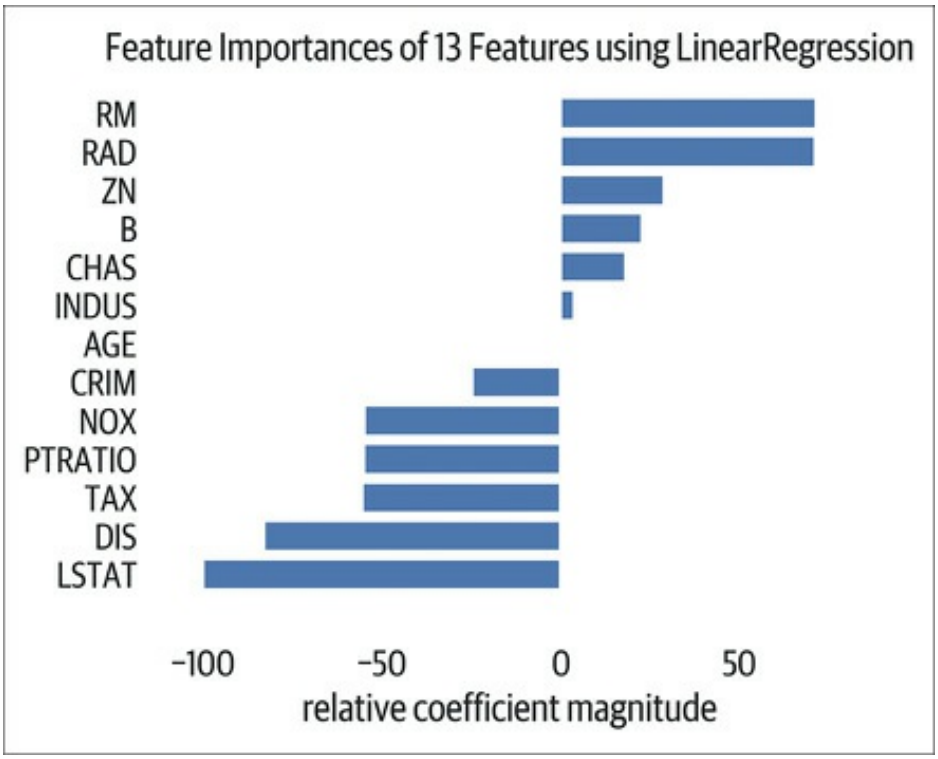


Figura 14.1 – Importância dos atributos. Este gráfico mostra que RM (número de cômodos) aumenta o preço, a idade (age) não importa e LSTAT (porcentagem da população com status mais baixo) reduz o preço.

SVMs

As SVMs (Support Vector Machines, ou Máquinas de Vetores Suporte) também podem fazer regressão.

As SVMs têm as seguintes propriedades:

Eficiência na execução

A implementação do scikit-learn é $O(n^4)$, portanto pode ser difícil escalar para tamanhos maiores. Usar um kernel linear ou o modelo `LinearSVR` pode melhorar o desempenho da execução, talvez à custa da acurácia. Aumentar o valor do parâmetro `cache_size` pode reduzir a ordem para $O(n^3)$.

Pré-processamento dos dados

O algoritmo não é invariante à escala (scale invariant), portanto padronizar os dados é extremamente recomendável.

Para evitar uma superadequação

O parâmetro `C` (parâmetro de penalidade) controla a regularização. Um

valor menor permite uma margem menor no hiperplano. Um valor maior para gamma tenderá a uma superadequação nos dados de treinamento. O modelo LinearSVR aceita parâmetros loss e penalty para regularização. O parâmetro epsilon pode ser aumentado (com 0, você deve esperar uma superadequação).

Interpretação dos resultados

Inspecione .support_vectors_, embora esses sejam difíceis de interpretar. Com kernels lineares, você poderá inspecionar .coef_.

Eis um exemplo de uso da biblioteca:

```
>>> from sklearn.svm import SVR
>>> svr = SVR()
>>> svr.fit(bos_sX_train, bos_sy_train)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
  epsilon=0.1, gamma='auto', kernel='rbf',
  max_iter=-1, shrinking=True, tol=0.001,
  verbose=False)

>>> svr.score(bos_sX_test, bos_sy_test)
0.6555356362002485
```

Parâmetros da instância:

C=1.0

É o parâmetro de penalidade. Quanto menor o valor, mais estreita será a fronteira de decisão (mais superadequação).

cache_size=200

Tamanho do cache (MB). Aumentar esse valor pode melhorar o tempo de treinamento em conjuntos de dados grandes.

coef0=0.0

Termo independente para kernels polinomiais e sigmoides.

epsilon=0.1

Define uma margem de tolerância na qual nenhuma penalidade será aplicada aos erros. Deve ser menor para conjuntos de dados maiores.

degree=3

Grau para kernel polinomial.

gamma='auto'

Coeficiente do kernel. Pode ser um número, 'scale' (default em 0.22, 1 / (num atributos * X.std())) ou 'auto' (default anterior, 1 / num atributos). Um valor menor resulta em superadequação nos dados de treinamento.

kernel='rbf'

Tipo de kernel: 'linear', 'poly', 'rbf' (default), 'sigmoid', 'precomputed' ou uma função.

max_iter=-1

Número máximo de iterações para o solucionador (solver). -1 significa que não há limites.

probability=False

Ativa a estimação de probabilidades. Deixa o treinamento mais lento.

random_state=None

Semente (seed) aleatória.

shrinking=True

Usa heurística de shrinking (encolhimento).

tol=0.001

Tolerância para parada.

verbose=False

Verbosidade.

Atributos após a adequação:

support_

Índices dos vetores suporte.

support_vectors_

Vetores suporte.

coef_

Coeficientes para kernel (linear).

intercept_

Constante para função de decisão.

K vizinhos mais próximos

O modelo KNN (K-Nearest Neighbor, ou K vizinhos mais próximos) também aceita regressão, e encontra k vizinhos alvos para a amostra sobre a qual

queremos fazer uma predição. Na regressão, esse modelo calcula a média dos alvos para determinar uma predição.

Modelos de vizinhos mais próximos têm as seguintes propriedades:

Eficiência na execução

Tempo de treinamento é O(1), mas há um compromisso, pois os dados das amostras precisam ser armazenados. O tempo de teste é O(Nd), em que N é o número de exemplos de treinamento e d é a dimensionalidade.

Pré-processamento dos dados

Sim, cálculos baseados em distância têm melhor desempenho quando há padronização.

Para evitar uma superadequação

Eleve `n_neighbors`. Mude `p` para métrica L1 ou L2.

Interpretação dos resultados

Interprete os k vizinhos mais próximos da amostra (usando o método `.kneighbors`). Esses vizinhos (se você puder explicá-los) explicarão o seu resultado.

Eis um exemplo de uso do modelo:

```
>>> from sklearn.neighbors import (
... KNeighborsRegressor,
... )
>>> knr = KNeighborsRegressor()
>>> knr.fit(bos_sX_train, bos_sy_train)
KNeighborsRegressor(algorithm='auto',
  leaf_size=30, metric='minkowski',
  metric_params=None, n_jobs=1, n_neighbors=5,
  p=2, weights='uniform')

>>> knr.score(bos_sX_test, bos_sy_test)
0.747112767457727
```

Atributos:

algorithm='auto'

Pode ser `'brute'`, `'ball_tree'` ou `'kd_tree'`.

leaf_size=30

Usado em algoritmos baseados em árvore.

metric='minkowski'
  Métrica de distância.

metric_params=None
  Dicionário adicional de parâmetros para função de métrica personalizada.

n_jobs=1
  Número de CPUs.

n_neighbors=5
  Número de vizinhos.

p=2
  Parâmetro de potência de Minkowski: 1 = manhattan (L1); 2 = euclidiana (L2).

weights='uniform'
  Pode ser 'distance', caso em que pontos mais próximos terão mais influência.

Árvore de decisão

As árvores de decisão (decision trees) aceitam classificação e regressão. Em cada nível da árvore, várias separações nos atributos são avaliadas. A separação que gerar o menor erro (impureza) será escolhida. O parâmetro criterion pode ser ajustado para determinar a métrica para a impureza.

As árvores de decisão têm as seguintes propriedades:

Eficiência na execução
  Para a criação, percorre cada um dos m atributos e ordena todas as n amostras: O(mn log n). Para predição, você percorrerá a árvore: O(altura).

Pré-processamento dos dados
  Não é necessário escalar. É preciso se livrar dos valores ausentes e convertê-los em dados numéricos.

Para evitar uma superadequação
  Defina max_depth com um número menor e aumente min_impurity_decrease.

Interpretação dos resultados
  É possível percorrer a árvore de opções. Por haver passos, uma árvore é ruim para lidar com relacionamentos lineares (uma pequena mudança nos

valores de um atributo pode resultar na formação de uma árvore totalmente diferente). A árvore também é extremamente dependente dos dados de treinamento. Uma pequena mudança pode modificar a árvore toda.

Eis um exemplo que utiliza a biblioteca scikit-learn:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> dtr = DecisionTreeRegressor(random_state=42)
>>> dtr.fit(bos_X_train, bos_y_train)
DecisionTreeRegressor(criterion='mse',
  max_depth=None, max_features=None,
  max_leaf_nodes=None, min_impurity_decrease=0.0,
  min_impurity_split=None, min_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0, presort=False,
  random_state=42, splitter='best')

>>> dtr.score(bos_X_test, bos_y_test)
0.8426751288675483
```

Parâmetros da instância:

criterion='mse'

Função de separação. O padrão é o erro quadrático médio (perda L2). Pode ser 'friedman_mse' ou 'mae' (perda L1).

max_depth=None

Profundidade da árvore. O default será construir a árvore até que as folhas contenham menos que min_samples_split.

max_features=None

Número de atributos a serem analisados para separação. O default são todos.

max_leaf_nodes=None

Limita o número de folhas. O default é sem limites.

min_impurity_decrease=0.0

Divide um nó se a separação diminuir a impureza por um valor maior ou igual a esse.

min_impurity_split=None

Obsoleto.

min_samples_leaf=1

Número mínimo de amostras em cada folha.

min_samples_split=2

Número mínimo de amostras necessárias para dividir um nó.

min_weight_fraction_leaf=0.0

Soma mínima de pesos exigida para nós do tipo folha.

presort=False

Pode agilizar o treinamento com um conjunto de dados menor ou com uma profundidade restrita se definido com True.

random_state=None

Semente (seed) aleatória.

splitter='best'

Use 'random' ou 'best'.

Atributos após a adequação:

feature_importances_

Array de importância de Gini.

max_features_

Valor calculado de max_features.

n_outputs_

Número de saídas.

n_features_

Número de atributos.

tree_

Objeto árvore subjacente.

Visualize a árvore (veja a Figura 14.2):

```
>>> import pydotplus
>>> from io import StringIO
>>> from sklearn.tree import export_graphviz
>>> dot_data = StringIO()
>>> tree.export_graphviz(
... dtr,
... out_file=dot_data,
... feature_names=bos_X.columns,
... filled=True,
```

```
... )
>>> g = pydotplus.graph_from_dot_data(
... dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1402.png")
```

Com o Jupyter, utilize:

```
from IPython.display import Image
Image(g.create_png())
```

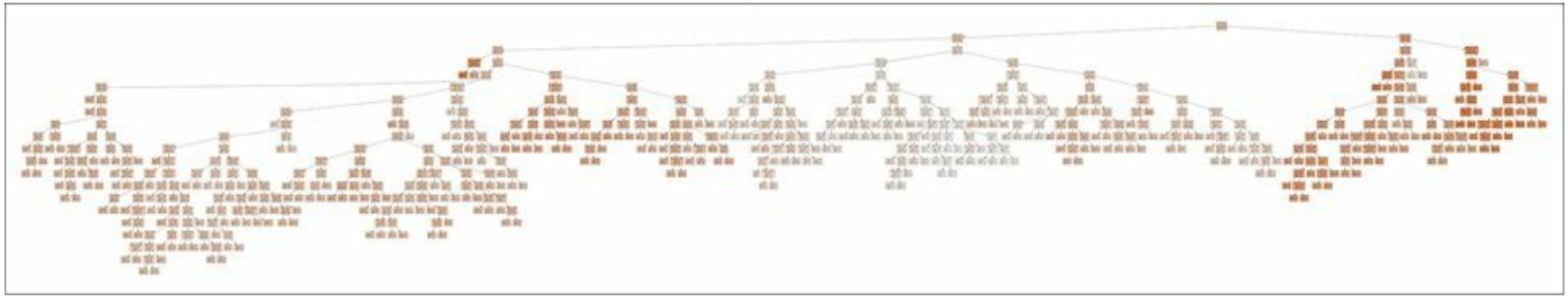


Figura 14.2 – Árvore de decisão.

Esse gráfico ficou um pouco grande. Em um computador, você pode fazer zoom em partes dele. Também é possível limitar a profundidade do gráfico (veja a Figura 14.3). (O fato é que os atributos mais importantes em geral estarão próximos ao topo da árvore.) Usaremos o parâmetro max_depth para isso:

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
... dtr,
... max_depth=2,
... out_file=dot_data,
... feature_names=bos_X.columns,
... filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
... dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1403.png")
```

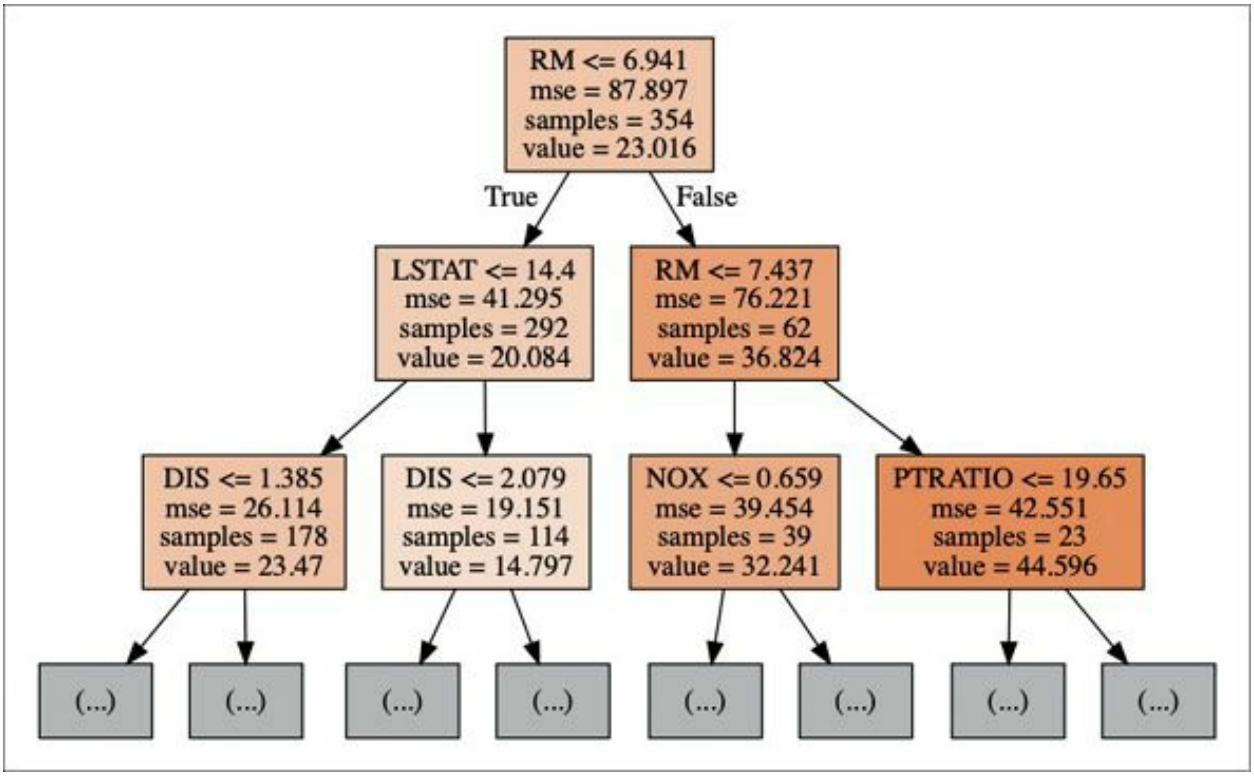


Figura 14.3 – Os dois primeiros níveis de uma árvore de decisão.

Também podemos usar o pacote dtreeviz para visualizar um gráfico de

dispersão em cada um dos nós da árvore (veja a Figura 14.4). Usaremos uma árvore limitada com uma profundidade de dois para que vejamos os detalhes:

```
>>> dtr3 = DecisionTreeRegressor(max_depth=2)
>>> dtr3.fit(bos_X_train, bos_y_train)
>>> viz = dtreeviz.trees.dtreeviz(
... dtr3,
... bos_X,
... bos_y,
... target_name="price",
... feature_names=bos_X.columns,
... )
>>> viz
```

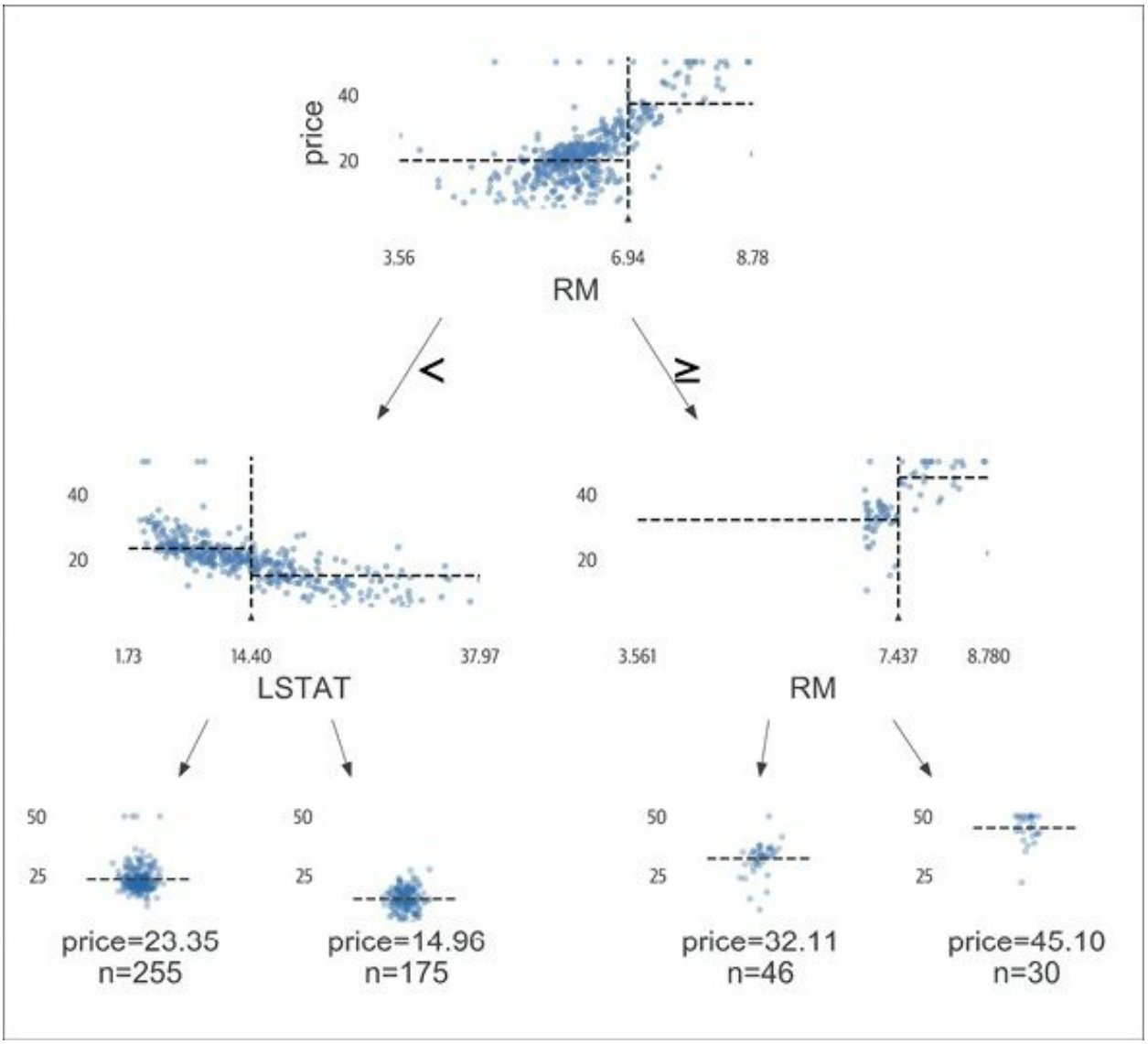


Figura 14.4 – Regressão com o dtviz.

Importância dos atributos:

```
>>> for col, val in sorted(
... zip(
... bos_X.columns, dtr.feature_importances_
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
RM 0.574
LSTAT 0.191
DIS 0.110
CRIM 0.061
RAD 0.018
```

Floresta aleatória

As árvores de decisão são boas porque são explicáveis, mas têm uma tendência à superadequação. Uma floresta aleatória (random forest) troca parte da facilidade de explicação por um modelo com uma tendência melhor à generalização. Esse modelo também pode ser usado para regressão.

As florestas aleatórias têm as seguintes propriedades:

Eficiência na execução

Deve criar j árvores aleatórias. Isso pode ser feito em paralelo usando `n_jobs`. A complexidade de cada árvore é de O(mn log n), em que n é o número de amostras e m é o número de atributos. Na criação, percorre cada um dos m atributos em um laço e ordena todas as n amostras: O(mn log n). Para predição, você percorrerá a árvore: O(altura).

Pré-processamento dos dados

Não é necessário, desde que a entrada seja numérica e não haja valores ausentes.

Para evitar uma superadequação

Adicione mais árvores (`n_estimators`). Utilize um valor menor de `max_depth`.

Interpretação dos resultados

Tem suporte para importância de atributos, porém não há uma única árvore de decisão para percorrer. É possível inspecionar árvores individuais do conjunto.

Eis um exemplo de uso do modelo:

```
>>> from sklearn.ensemble import (
... RandomForestRegressor,
... )
>>> rfr = RandomForestRegressor(
... random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
RandomForestRegressor(bootstrap=True,
  criterion='mse', max_depth=None,
  max_features='auto', max_leaf_nodes=None,
  min_impurity_decrease=0.0,
  min_impurity_split=None,_samples_leaf=1,
  min_samples_split=2,
  min_weight_fraction_leaf=0.0,
  n_estimators=100, n_jobs=1,
  oob_score=False, random_state=42,
  verbose=0, warm_start=False)

>>> rfr.score(bos_X_test, bos_y_test)
0.8641887615545837
```

Parâmetros da instância (estas opções espelham a árvore de decisão):

bootstrap=True

Utiliza bootstrap ao construir as árvores.

criterion='mse'

Função de separação, 'mae'.

max_depth=None

Profundidade da árvore. O default será construir a árvore até que as folhas contenham menos que min_samples_split.

max_features='auto'

Número de atributos para analisar na separação. O default são todos.

max_leaf_nodes=None

Limita o número de folhas. O default é sem limites.

min_impurity_decrease=0.0

Divide um nó se a separação diminuir a impureza por um valor maior ou igual a esse.

min_impurity_split=None

Obsoleto.

min_samples_leaf=1

Número mínimo de amostras em cada folha.

min_samples_split=2

Número mínimo de amostras exigido para dividir um nó.

min_weight_fraction_leaf=0.0

Soma mínima de pesos exigida para nós do tipo folha.

n_estimators=10

Número de árvores na floresta.

n_jobs=None

Número de jobs para adequação e predição. (None que dizer 1.)

oob_score=False

Informa se deve usar amostras OOB para estimar a pontuação em dados não vistos antes.

random_state=None

Semente (seed) aleatória.

verbose=0

Verbosidade.

warm_start=False

Faz a adequação de uma nova floresta ou usa uma floresta existente.

Atributos após a adequação:

estimators_

Coleção de árvores.

feature_importances_

Array de importância de Gini.

n_classes_

Número de classes.

n_features_

Número de atributos.

oob_score_

Pontuação do conjunto de dados de treinamento usando estimativas OOB.

Importância dos atributos:

```
>>> for col, val in sorted(
... zip(
... bos_X.columns, rfr.feature_importances_
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
RM 0.505
LSTAT 0.283
DIS 0.115
CRIM 0.029
PTRATIO 0.016
```

Regressão XGBoost

A biblioteca XGBoost também aceita regressão. Ela constrói uma árvore de decisão simples e, em seguida, faz uma "melhoria" (boosting) adicionando árvores subsequentes. Cada árvore tenta corrigir os resíduos da saída anterior. Na prática, funciona muito bem em dados estruturados.

O XGBoost tem as seguintes propriedades:

Eficiência na execução

O XGBoost pode executar em paralelo. Utilize a opção `n_jobs` para informar o número de CPUs. Use a GPU para ter um desempenho melhor ainda.

Pré-processamento dos dados

Não é necessário escalar com modelos baseados em árvore. É preciso codificar os dados de categoria. Aceita dados ausentes!

Para evitar uma superadequação

O parâmetro `early_stopping_rounds=N` pode ser definido para interromper o treinamento caso não haja melhoras após N rodadas. As regularizações L1 e L2 são controladas por `reg_alpha` e `reg_lambda`, respectivamente. Números maiores são mais conservadores.

Interpretação dos resultados

Inclui importância de atributos.

Eis um exemplo de uso da biblioteca:

```
>>> xgr = xgb.XGBRegressor(random_state=42)
>>> xgr.fit(bos_X_train, bos_y_train)
XGBRegressor(base_score=0.5, booster='gbtree',
  colsample_bylevel=1, colsample_bytree=1,
  gamma=0, learning_rate=0.1, max_delta_step=0,
  max_depth=3, min_child_weight=1, missing=None,
  n_estimators=100, n_jobs=1, nthread=None,
  objective='reg:linear', random_state=42,
  reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
  seed=None, silent=True, subsample=1)

>>> xgr.score(bos_X_test, bos_y_test)
0.871679473122472

>>> xgr.predict(bos_X.iloc[[0]])
array([27.013563], dtype=float32)
```

Parâmetros da instância:

max_depth=3

Profundidade máxima.

learning_rate=0.1

Taxa de aprendizagem (eta) para boosting (entre 0 e 1). Após cada passo de boosting (melhoria), os pesos recém-adicionados são escalados de acordo com esse fator. Quanto menor o valor, mais conservador será, mas também serão necessárias mais árvores para convergir. Na chamada a .train, você pode passar um parâmetro learning_rates, que é uma lista de taxas em cada rodada (isto é, [.1]*100 + [.05]*100).

n_estimators=100

Número de rodadas ou árvores melhoradas.

silent=True

Informa se deve exibir mensagens durante a execução do boosting.

objective="reg:linear"

Tarefa de aprendizagem ou callable para classificação.

booster="gbtree"

Pode ser 'gbtree', 'gblinear' ou 'dart'. A opção 'dart' adiciona descartes (descarta árvores aleatórias para evitar superadequação). A opção 'gblinear' cria um

modelo linear regularizado (leia-se não uma árvore, mas algo semelhante a uma regressão lasso).

nthread=None

Obsoleto.

n_jobs=1

Número de threads a serem usadas.

gamma=0

Redução de perda mínima necessária para dividir mais uma folha.

min_child_weight=1

Valor mínimo para a soma hessiana de um filho.

max_delta_step=0

Deixa as atualizações mais conservadoras. Defina com valores de 1 a 10 para classes desbalanceadas.

subsample=1

Fração das amostras a serem usadas na próxima rodada de boosting.

colsample_bytree=1

Fração das colunas a serem usadas por rodada de boosting.

colsample_bylevel=1

Fração das colunas a serem usadas por nível da árvore.

colsample_bynode=1

Fração das colunas a serem usadas por divisão (nó da árvore).

reg_alpha=0

Regularização L1 (média dos pesos). Aumente o valor para ser mais conservador.

reg_lambda=1

Regularização L2 (raiz dos quadrados dos pesos). Aumente o valor para ser mais conservador.

base_score=.5

Previsão inicial.

seed=None

Obsoleto.

random_state=0

Semente (seed) aleatória.

missing=None

Valor de interpretação para dado ausente. None quer dizer np.nan.

importance_type='gain'

Tipo da importância dos atributos: 'gain', 'weight', 'cover', 'total_gain' ou 'total_cover'.

Atributos:

coef_

Coeficientes para learners gblinear (booster = 'gblinear').

intercept_

Intercepto para learners gblinear.

feature_importances_

Importância de atributos para learners gbtree.

A importância dos atributos é o ganho médio em todos os nós em que o atributo é usado:

```
>>> for col, val in sorted(
... zip(
... bos_X.columns, xgr.feature_importances_
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
DIS 0.187
CRIM 0.137
RM 0.137
LSTAT 0.134
AGE 0.110
```

O XGBoost inclui recursos de geração de gráficos para importância de atributos. Observe que o parâmetro importance_type modifica os valores no gráfico a seguir (veja a Figura 14.5). O default é usar pesos para determinar a importância dos atributos:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> xgb.plot_importance(xgr, ax=ax)
>>> fig.savefig("images/mlpr_1405.png", dpi=300)
```

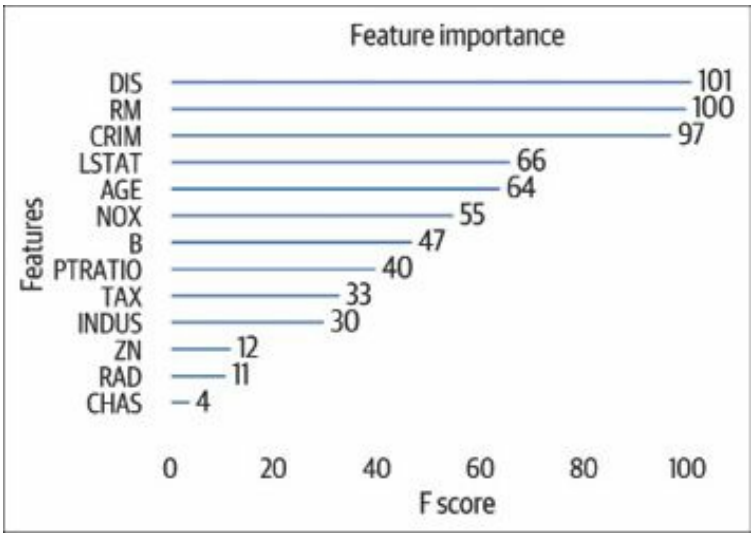


Figura 14.5 – Importância dos atributos usando pesos (quantas vezes um atributo é dividido nas árvores).

Usando o Yellowbrick para gerar um gráfico da importância dos atributos (o atributo feature_importances_ será normalizado) (veja a Figura 14.6):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> fi_viz = FeatureImportances(xgr)
>>> fi_viz.fit(bos_X_train, bos_y_train)
>>> fi_viz.poof()
>>> fig.savefig("images/mlpr_1406.png", dpi=300)
```

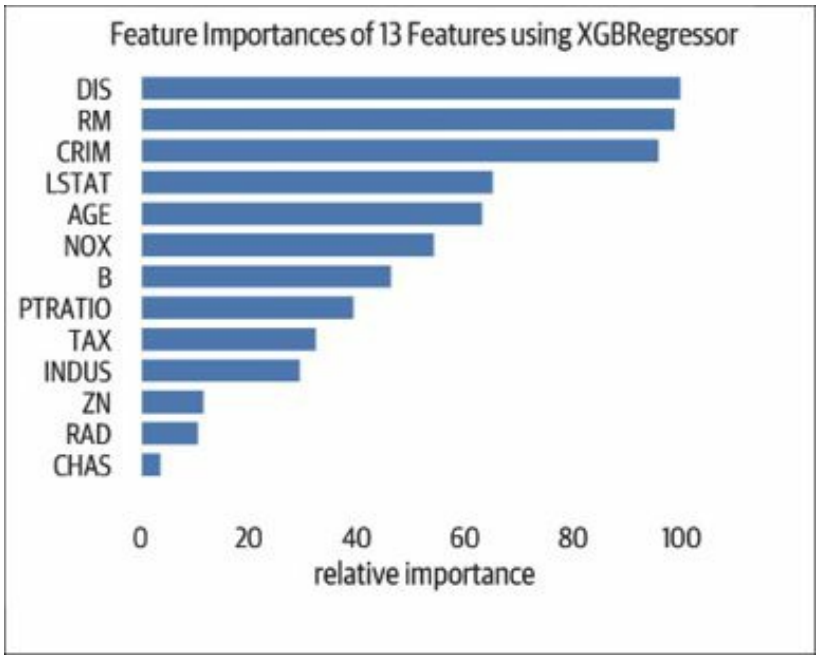


Figura 14.6 – Importância dos atributos usando a importância relativa do ganho (importância percentual do atributo mais importante).

O XGBoost possibilita uma representação tanto textual como gráfica das árvores. Eis a representação textual:

```
>>> booster = xgr.get_booster()
>>> print(booster.get_dump()[0])
0:[LSTAT<9.72500038] yes=1,no=2,missing=1
 1:[RM<6.94099998] yes=3,no=4,missing=3
  3:[DIS<1.48494995] yes=7,no=8,missing=7
   7:leaf=3.9599998
   8:leaf=2.40158272
  4:[RM<7.43700027] yes=9,no=10,missing=9
   9:leaf=3.22561002
   10:leaf=4.31580687
 2:[LSTAT<16.0849991] yes=5,no=6,missing=5
  5:[B<116.024994] yes=11,no=12,missing=11
   11:leaf=1.1825
   12:leaf=1.99701393
  6:[NOX<0.603000045] yes=13,no=14,missing=13
   13:leaf=1.6868
   14:leaf=1.18572915
```

Os valores das folhas podem ser interpretados como a soma de `base_score` e da folha. (Para validar isso, chame `.predict` com o parâmetro `ntree_limit=1` a fim de limitar o modelo para que use o resultado da primeira árvore.)

Eis uma versão gráfica da árvore (veja a Figura 14.7):

```
fig, ax = plt.subplots(figsize=(6, 4))
xgb.plot_tree(xgr, ax=ax, num_trees=0)
fig.savefig('images/mlpr_1407.png', dpi=300)
```

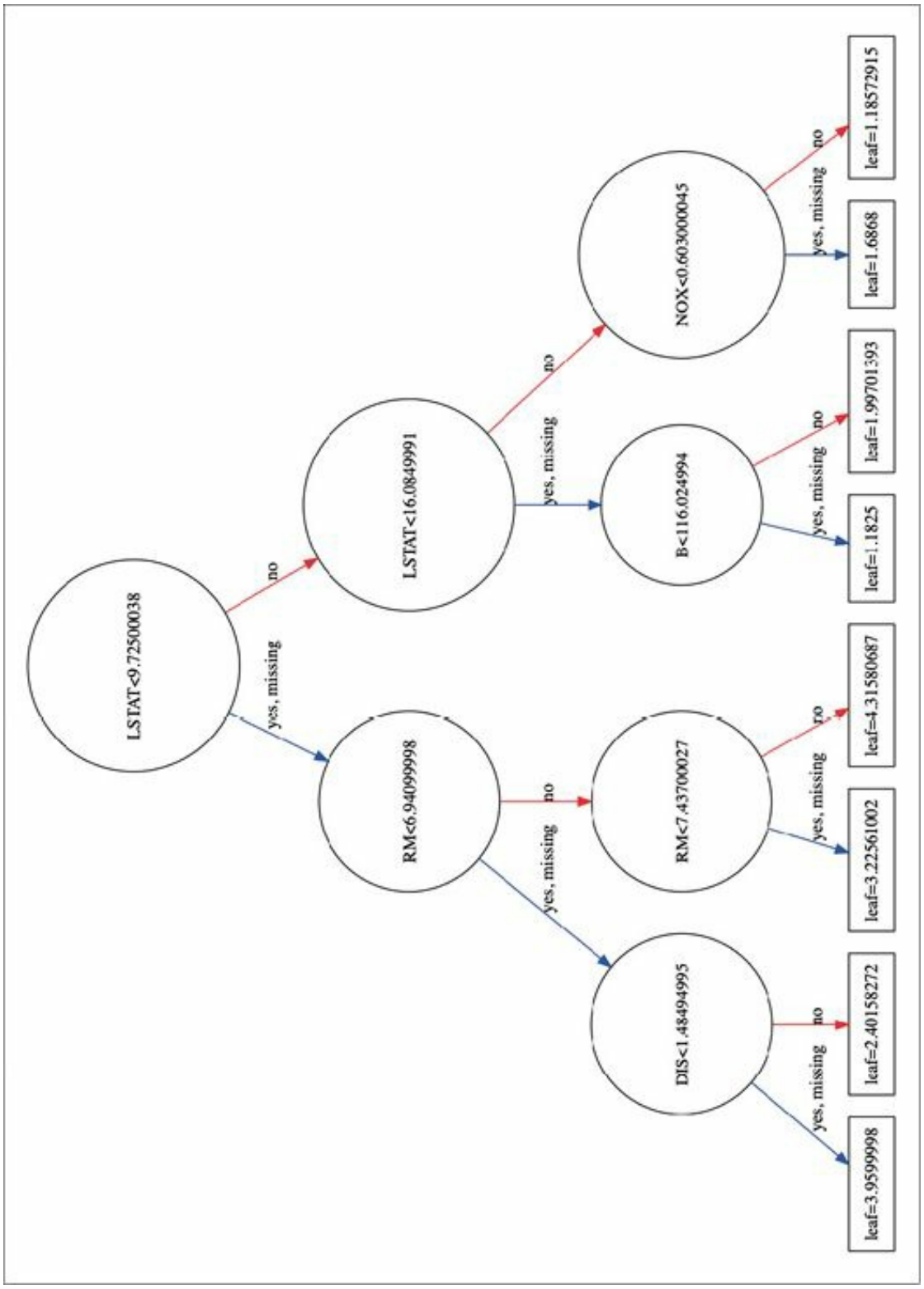


Figura 14.7 – Árvore do XGBoost.

Regressão LightGBM

A biblioteca de Gradient Boosting Tree, LightGBM, também aceita regressão. Conforme mencionamos no capítulo sobre classificação, ela pode ser mais rápida que o XGBoost para criar árvores em virtude do método de amostragem usado para determinar as separações dos nós.

Além disso, lembre-se de que ela cria as árvores inicialmente de acordo com a profundidade, portanto limitá-la pode prejudicar o modelo. O LightGBM tem as seguintes propriedades:

Eficiência na execução

Consegue tirar proveito de várias CPUs. Se usar binning, pode ser 15 vezes mais rápido que o XGBoost.

Pré-processamento dos dados

Tem algum suporte para codificação de colunas de categorias como inteiros (ou o tipo `Categorical` do pandas), mas o AUC parece ser pior em comparação com a codificação one-hot.

Para evitar uma superadequação

Reduza `num_leaves`, aumente `min_data_in_leaf` e utilize `min_gain_to_split` com `lambda_l1` ou `lambda_l2`.

Interpretação dos resultados

A importância dos atributos está disponível. Árvores individuais são fracas e tendem a ser difíceis de interpretar.

Eis um exemplo de uso do modelo:

```
>>> import lightgbm as lgb
>>> lgr = lgb.LGBMRegressor(random_state=42)
>>> lgr.fit(bos_X_train, bos_y_train)
LGBMRegressor(boosting_type='gbdt',
  class_weight=None, colsample_bytree=1.0,
  learning_rate=0.1, max_depth=-1,
  min_child_samples=20, min_child_weight=0.001,
  min_split_gain=0.0, n_estimators=100,
  n_jobs=-1, num_leaves=31, objective=None,
  random_state=42, reg_alpha=0.0,
  reg_lambda=0.0, silent=True, subsample=1.0,
  subsample_for_bin=200000, subsample_freq=0)

>>> lgr.score(bos_X_test, bos_y_test)
0.847729219534575
```

```
>>> lgr.predict(bos_X.iloc[[0]])
array([30.31689569])
```

Parâmetros da instância:

boosting_type='gbdt'

Pode ser: 'gbdt' (gradient boosting), 'rf' (random forest), 'dart' (dropouts meet multiple additive regression trees) ou 'goss' (gradient-based, one-sided sampling).

num_leaves=31

Número máximo de folhas das árvores.

max_depth=-1

Profundidade máxima da árvore. -1 significa sem limites. Profundidades maiores tendem a causar mais superadequação.

learning_rate=0.1

Intervalo de (0, 1.0]. Taxa de aprendizagem para boosting. Um valor menor atrasa a superadequação, pois as rodadas de boosting terão menos impacto. Um número menor deve resultar em um desempenho melhor, mas exigirá um valor maior para num_iterations.

n_estimators=100

Número de árvores ou rodadas de boosting.

subsample_for_bin=200000

Amostras necessárias para criar bins.

objective=None

None faz uma regressão por default. Pode ser uma função ou uma string.

min_split_gain=0.0

Redução de perda exigida para particionar uma folha.

min_child_weight=0.001

Soma do peso hessiano exigido para uma folha. Valores maiores serão mais conservadores.

min_child_samples=20

Número de amostras exigidas para uma folha. Números menores implicam mais superadequação.

subsample=1.0

Fração das amostras a serem usadas na próxima rodada.

subsample_freq=0

Frequência das subamostras. Mude para 1 para ativar.

colsample_bytree=1.0

Intervalo de (0, 1.0]. Seleciona uma porcentagem dos atributos para cada rodada de boosting.

reg_alpha=0.0

Regularização L1 (média dos pesos). Aumente o valor para ser mais conservador.

reg_lambda=0.0

Regularização L2 (raiz dos quadrados dos pesos). Aumente o valor para ser mais conservador.

random_state=42

Semente (seed) aleatória.

n_jobs=-1

Número de threads.

silent=True

Modo verboso.

importance_type='split'

Determina como a importância é calculada: split (número de vezes que um atributo foi usado) ou gain (ganhos totais das separações quando um atributo foi usado).

O LightGBM aceita importância de atributos. O parâmetro importance_type determina como isso é calculado (o default é baseado no número de vezes que um atributo foi usado):

```
>>> for col, val in sorted(
... zip(
... bos_X.columns, lgr.feature_importances_
... ),
... key=lambda x: x[1],
... reverse=True,
... )[:5]:
... print(f"{col:10}{val:10.3f}")
```

LSTAT 226.000
RM 199.000
DIS 172.000
AGE 130.000
B 121.000

Gráfico da importância dos atributos mostrando quantas vezes um atributo é usado (veja a Figura 14.8):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> lgb.plot_importance(lgr, ax=ax)
>>> fig.tight_layout()
>>> fig.savefig("images/mlpr_1408.png", dpi=300)
```

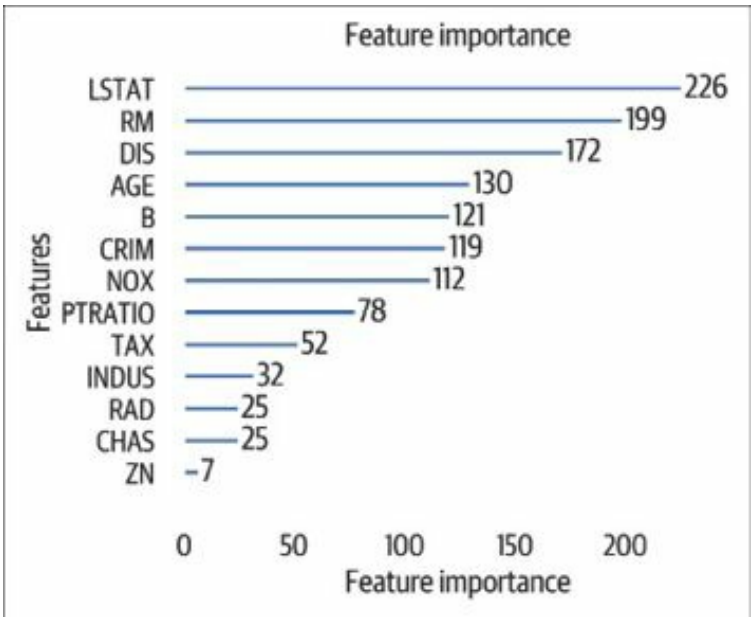


Figura 14.8 – Gráfico da importância de atributos mostrando quantas vezes um atributo é usado.

DICA

No Jupyter, utilize o comando a seguir para visualizar uma árvore:

```
lgb.create_tree_digraph(lgbr)
```

CAPÍTULO 15

Métricas e avaliação de regressão

Neste capítulo, avaliaremos os resultados de uma regressão com floresta aleatória, cujo treinamento foi feito com os dados habitacionais de Boston:

```
>>> rfr = RandomForestRegressor(
... random_state=42, n_estimators=100
... )
>>> rfr.fit(bos_X_train, bos_y_train)
```

Métricas

O módulo sklearn.metrics inclui métricas para avaliar modelos de regressão. As funções de métrica terminadas com loss ou error devem ser minimizadas. Funções terminadas com score devem ser maximizadas.

O *coeficiente de determinação* (r^2) é uma métrica de regressão comum. Em geral, esse valor está entre 0 e 1 e representa o percentual da variância do alvo (target) com o qual os atributos contribuem. Valores maiores são melhores, mas, de modo geral, é difícil avaliar o modelo exclusivamente a partir dessa métrica. Um valor igual a 0,7 representa uma boa pontuação? Depende. Para um dado conjunto de dados, 0,5 pode ser uma boa pontuação, enquanto, para outro, um valor igual a 0,9 poderia ser ruim. Geralmente usamos esse número junto com outras métricas ou visualizações para avaliar um modelo.

Por exemplo, é fácil criar um modelo que faça a predição dos preços de ações para o dia seguinte com um r^2 igual a 0,99. No entanto, eu não comprometeria o meu dinheiro com base nesse modelo. Ele poderia fazer predições um pouco abaixo ou acima, o que significaria um grande problema nas negociações.

A métrica r^2 é a métrica default, usada durante buscas em grade (grid search). Outras métricas podem ser especificadas com o parâmetro scoring.

O método .score calcula esse valor para modelos de regressão:

```
>>> from sklearn import metrics
>>> rfr.score(bos_X_test, bos_y_test)
0.8721182042634867

>>> metrics.r2_score(bos_y_test, bos_y_test_pred)
0.8721182042634867
```

NOTA

Há também uma métrica de *variância explicada* ('explained_variance' na busca em grade). Se a média dos *resíduos* (erros nas predições) for 0 (em modelos OLS (Ordinary Least Squares, ou Mínimos Quadrados Ordinários)), a variância explicada será igual ao coeficiente de determinação:

```
>>> metrics.explained_variance_score(
... bos_y_test, bos_y_test_pred
... )
0.8724890451227875
```

O *erro médio absoluto* ('neg_mean_absolute_error' quando usado na busca em grade) expressa o erro de previsão médio absoluto do modelo. Um modelo perfeito teria uma pontuação igual a 0, mas essa métrica não tem limites superiores, de modo diferente do coeficiente de determinação. No entanto, como está em unidades do alvo, ele é mais interpretável. Se quiser ignorar os valores discrepantes (outliers), essa é uma boa métrica a ser usada.

Essa medida não é capaz de informar quão ruim é um modelo, mas pode ser usada para comparar dois modelos. Se você tiver dois, o modelo com a pontuação menor será melhor.

Esse número nos informa que o erro médio está aproximadamente dois valores acima ou abaixo do valor real:

```
>>> metrics.mean_absolute_error(
... bos_y_test, bos_y_test_pred
... )
2.0839802631578945
```

A *raiz do erro quadrático médio* ('neg_mean_squared_error' na busca em grade) também avalia o erro do modelo em termos do alvo. No entanto, como ele calcula a média do quadrado dos erros antes de calcular a raiz quadrada, erros maiores são penalizados. Se quiser penalizar erros maiores, essa é uma boa métrica a ser usada: por exemplo, se estar errado em oito for mais do que duas vezes pior do que estar errado por quatro.

Como no caso do erro médio absoluto, essa medida não é capaz de informar quão ruim é um modelo, mas pode ser usada para comparar dois modelos. Caso você suponha que os erros estão distribuídos de modo normal, essa será uma boa opção.

O resultado nos informa que, se elevarmos os erros ao quadrado e tiramos a sua média, o valor resultante estará em torno de 9,5:

```
>>> metrics.mean_squared_error(
... bos_y_test, bos_y_test_pred
... )
9.52886846710526
```

O *erro logarítmico quadrático médio* (na busca em grade, `'neg_mean_squared_log_error'`) penaliza a subprevisão, mais do que a superprevisão. Se você tiver alvos que apresentem crescimento exponencial (população, ações etc.), essa é uma boa métrica.

Caso você considere o log do erro e então o eleve ao quadrado, a média desses resultados será igual a 0,021:

```
>>> metrics.mean_squared_log_error(
... bos_y_test, bos_y_test_pred
... )
0.02128263061776433
```

Gráfico de resíduos

Bons modelos (com pontuações R2 apropriadas) exibirão *homocedasticidade*. Isso significa que a variância é a mesma para todos os valores dos alvos, independentemente da entrada. Ao serem colocados em um gráfico de resíduos, os valores parecerão distribuídos aleatoriamente. Se houver padrões, é sinal de que o modelo ou os dados são problemáticos.

Os gráficos de resíduos também mostram valores discrepantes, que podem ter um grande impacto na adequação do modelo (veja a Figura 15.1).

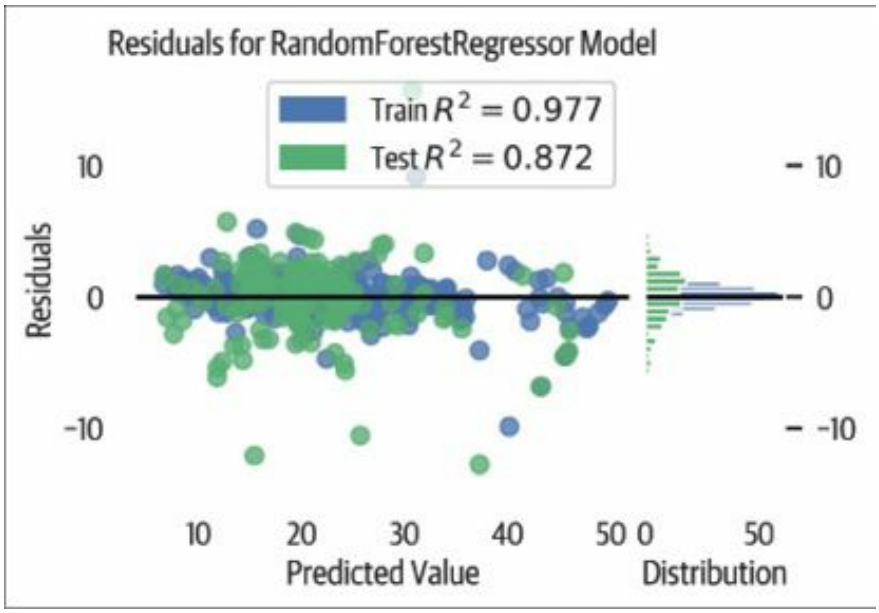


Figura 15.1 – Gráfico de resíduos. Outros testes mostrarão que esses dados são heterocedásticos.

O Yellowbrick é capaz de criar gráficos de resíduos para visualizar esses dados:

```
>>> from yellowbrick.regressor import ResidualsPlot
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> rpv = ResidualsPlot(rfr)
>>> rpv.fit(bos_X_train, bos_y_train)
>>> rpv.score(bos_X_test, bos_y_test)
>>> rpv.poof()
>>> fig.savefig("images/mlpr_1501.png", dpi=300)
```

Heterocedasticidade

A biblioteca statsmodel (*https://oreil.ly/HtIi5*) inclui o *teste de Breusch-Pagan* para heterocedasticidade. Isso significa que a variância dos resíduos varia nos valores previstos. No teste de Breusch-Pagan, se os valores p (p-values) forem significativos (p-value menor que 0,05), a hipótese nula da homocedasticidade será rejeitada. Isso mostra que os resíduos são heterocedásticos e que as previsões apresentam distorções.

O teste a seguir confirma a heterocedasticidade:

```
>>> import statsmodels.stats.api as sms
>>> hb = sms.het_breuschpagan(resids, bos_X_test)
>>> labels = [
... "Lagrange multiplier statistic",
```

```
... "p-value",
... "f-value",
... "f p-value",
... ]
>>> for name, num in zip(name, hb):
... print(f"{name}: {num:.2}")
Lagrange multiplier statistic: 3.6e+01
p-value: 0.00036
f-value: 3.3
f p-value: 0.00022
```

Resíduos com distribuição normal

A biblioteca scipy inclui um *gráfico de probabilidades* e o teste de *Kolmogorov-Smirnov*; ambos verificam se os resíduos têm distribuição normal.

Podemos gerar um histograma (veja a Figura 15.2) para visualizar os resíduos e verificar se têm distribuição normal:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> resids = bos_y_test - rfr.predict(bos_X_test)
>>> pd.Series(resids, name="residuals").plot.hist(
... bins=20, ax=ax, title="Residual Histogram"
... )
>>> fig.savefig("images/mlpr_1502.png", dpi=300)
```

A Figura 15.3 mostra um gráfico de probabilidades. Se as amostras representadas em relação aos quantis estiverem alinhadas, é sinal de que os resíduos têm distribuição normal. Podemos ver que isso não acontece neste caso:

```
>>> from scipy import stats
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> _ = stats.probplot(resids, plot=ax)
>>> fig.savefig("images/mlpr_1503.png", dpi=300)
```

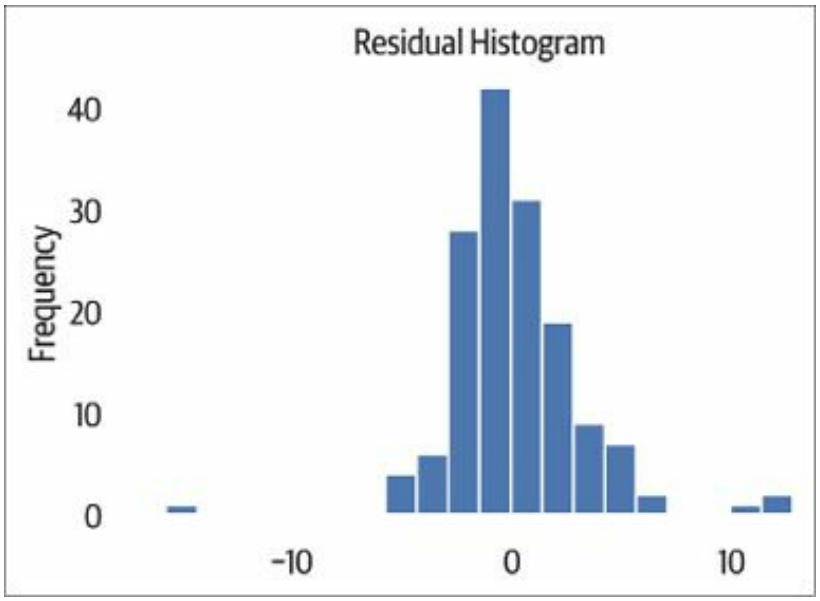


Figura 15.2 – Histograma dos resíduos.

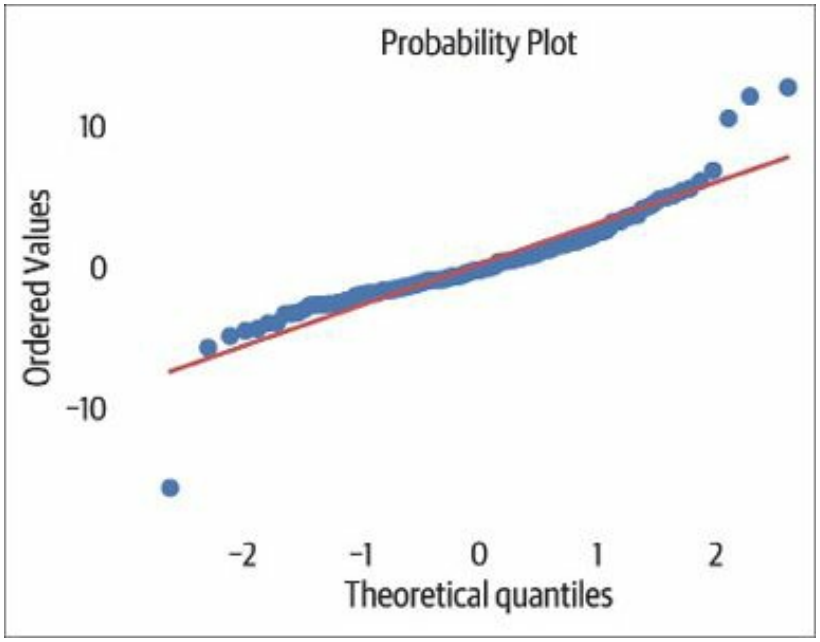


Figura 15.3 – Gráfico de probabilidade dos resíduos.

O teste de Kolmogorov-Smirnov é capaz de avaliar se uma distribuição é normal. Se o valor p for significativo (< 0,05), é sinal de que os valores não apresentam uma distribuição normal.

Esse teste também falha, informando que os resíduos não têm distribuição normal:

```
>>> stats.kstest(resids, cdf="norm")
KstestResult(statistic=0.1962230021010155, pvalue=1.3283596864921421e-05)
```

Gráfico de erros de predição

Um gráfico de erros de predição mostra os alvos reais em relação aos valores previstos. Em um modelo perfeito, esses pontos estariam alinhados em 45 graus.

Como nosso modelo parece prever valores menores na extremidade mais alta de y, o modelo tem alguns problemas de desempenho. Isso também é evidente no gráfico de resíduos (veja a Figura 15.4).

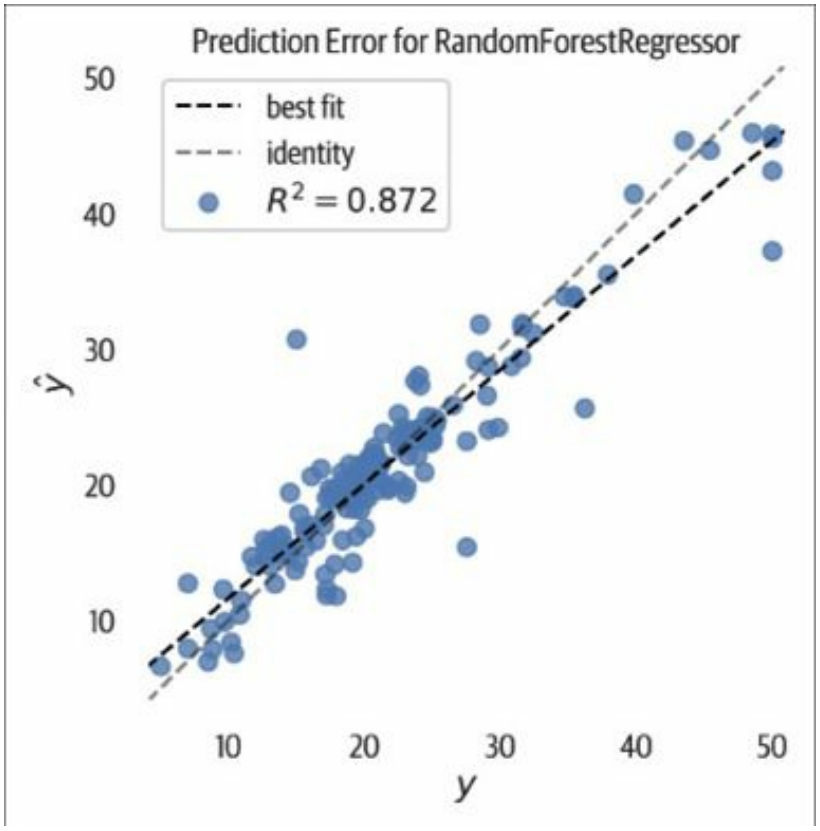


Figura 15.4 – Erro de predição. Mostra o y previsto (y circunflexo) versus o y real.

Eis a versão gerada com o Yellowbrick:

```
>>> from yellowbrick.regressor import (
... PredictionError,
... )
>>> fig, ax = plt.subplots(figsize=(6, 6))
>>> pev = PredictionError(rfr)
>>> pev.fit(bos_X_train, bos_y_train)
>>> pev.score(bos_X_test, bos_y_test)
>>> pev.poof()
```

```
>>> fig.savefig("images/mlpr_1504.png", dpi=300)
```

CAPÍTULO 16

Explicando os modelos de regressão

A maioria das técnicas usadas para explicar os modelos de classificação se aplicam também aos modelos de regressão. Neste capítulo, mostrarei como usar a biblioteca SHAP para interpretar os modelos de regressão.

Interpretaremos um modelo XGBoost para o conjunto de dados habitacionais de Boston:

```
>>> import xgboost as xgb
>>> xgr = xgb.XGBRegressor(
... random_state=42, base_score=0.5
... )
>>> xgr.fit(bos_X_train, bos_y_train)
```

Shapley

Sou grande fã do Shapley porque ele não depende do modelo. Essa biblioteca também nos dá insights globais sobre o nosso modelo e ajuda a explicar predições individuais. Caso você tenha um modelo caixa-preta, acho essa biblioteca bastante útil.

Inicialmente, veremos a predição para o índice 5. Nosso modelo prevê que o valor será 27,26:

```
>>> sample_idx = 5
>>> xgr.predict(bos_X.iloc[[sample_idx]])
array([27.269186], dtype=float32)
```

Para usar o modelo, temos de criar um TreeExplainer a partir de nosso modelo e estimar os valores SHAP para nossas amostras. Se quisermos usar o Jupyter e ter uma interface interativa, também será necessário chamar a função initjs:

```
>>> import shap
>>> shap.initjs()

>>> exp = shap.TreeExplainer(xgr)
>>> vals = exp.shap_values(bos_X)
```

Com o explainer e os valores SHAP, podemos criar um gráfico de forças (force plot) para explicar a predição (veja a Figura 16.1). Esse gráfico informa que a previsão de base é 23, e que o status da população (LSTAT) e a taxa de impostos da propriedade (TAX) empurram o preço para cima, enquanto o número de cômodos (RM) empurra o preço para baixo:

```
>>> shap.force_plot(
... exp.expected_value,
... vals[sample_idx],
... bos_X.iloc[sample_idx],
... )
```

Figura 16.1 – Gráfico de forças para regressão. O valor esperado é empurrado para cima, de 23 para 27, em virtude do status da população e da taxa de impostos.

Podemos visualizar o gráfico de forças para todas as amostras, bem como ter uma noção geral do comportamento. Se estivermos usando o modo JavaScript interativo no Jupyter, é possível passar o mouse sobre as amostras e ver quais atributos estão causando impacto no resultado (veja a Figura 16.2):

```
>>> shap.force_plot(
... exp.expected_value, vals, bos_X
... )
```

A partir do gráfico de forças da amostra, vimos que o atributo LSTAT causa um grande impacto. Para visualizar como LSTAT afeta o resultado, podemos criar um gráfico de dependências. A biblioteca escolherá automaticamente um atributo para colorir de acordo com ele (você pode fornecer o parâmetro `interaction_index` para definir o seu próprio atributo).

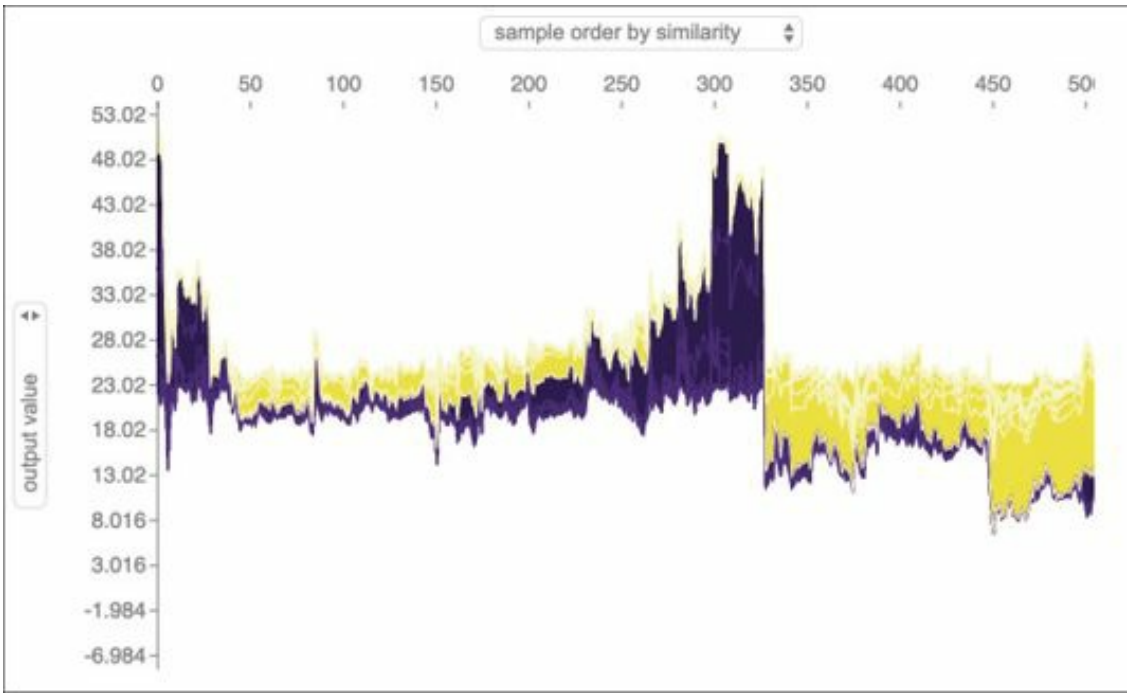


Figura 16.2 – Gráfico de forças para regressão, com todas as amostras.

A partir do gráfico de dependência para LSTAT (veja a Figura 16.3), podemos notar que, à medida que LSTAT aumenta (porcentagem da população de mais baixo status), o valor SHAP diminui (forçando o alvo para baixo). Um valor muito baixo de LSTAT força SHAP para cima. Ao observar as cores de TAX (taxa de impostos da propriedade), percebemos que, à medida que a taxa diminui (mais azul), o valor SHAP aumenta:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot("LSTAT", vals, bos_X)
>>> fig.savefig(
... "images/mlpr_1603.png",
... bbox_inches="tight",
... dpi=300,
... )
```

A seguir, apresentamos outro gráfico de dependência, exibido na Figura 16.4, para explorar o atributo DIS (distância até os centros de empregos). Parece que esse atributo tem pouco efeito, a menos que tenha valores bem baixos:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.dependence_plot(
... "DIS", vals, bos_X, interaction_index="RM"
... )
>>> fig.savefig(
... "images/mlpr_1604.png",
```

```
... bbox_inches="tight",
... dpi=300,
... )
```

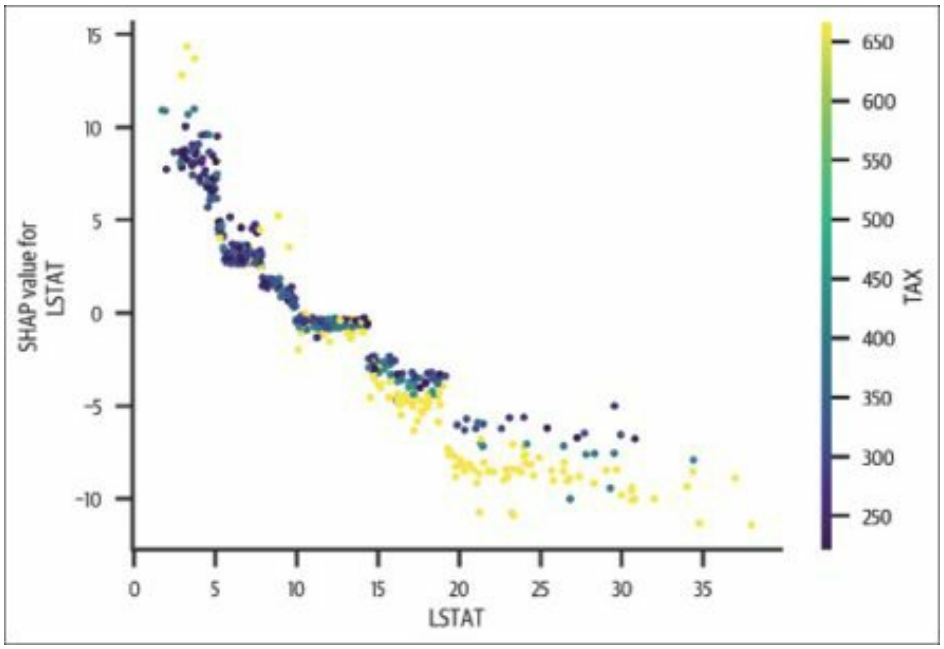


Figura 16.3 – Gráfico de dependência para LSTAT. À medida que LSTAT aumenta, o valor previsto diminui.

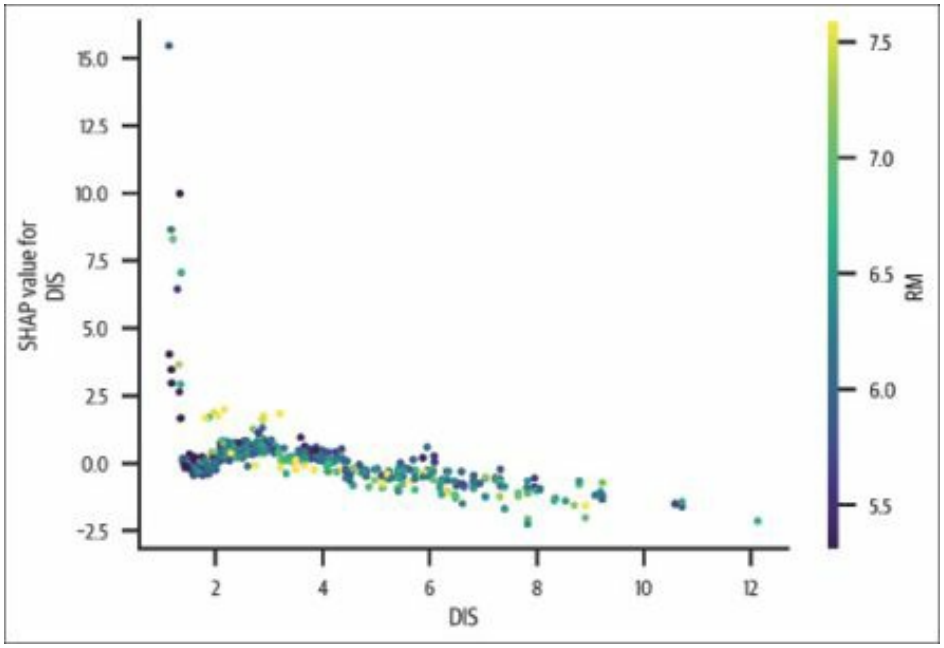


Figura 16.4 – Gráfico de dependência para DIS. A menos que DIS tenha valores muito baixos, o valor SHAP permanecerá relativamente constante.

Por fim, veremos o efeito global dos atributos usando um gráfico de resumo

(veja a Figura 16.5).

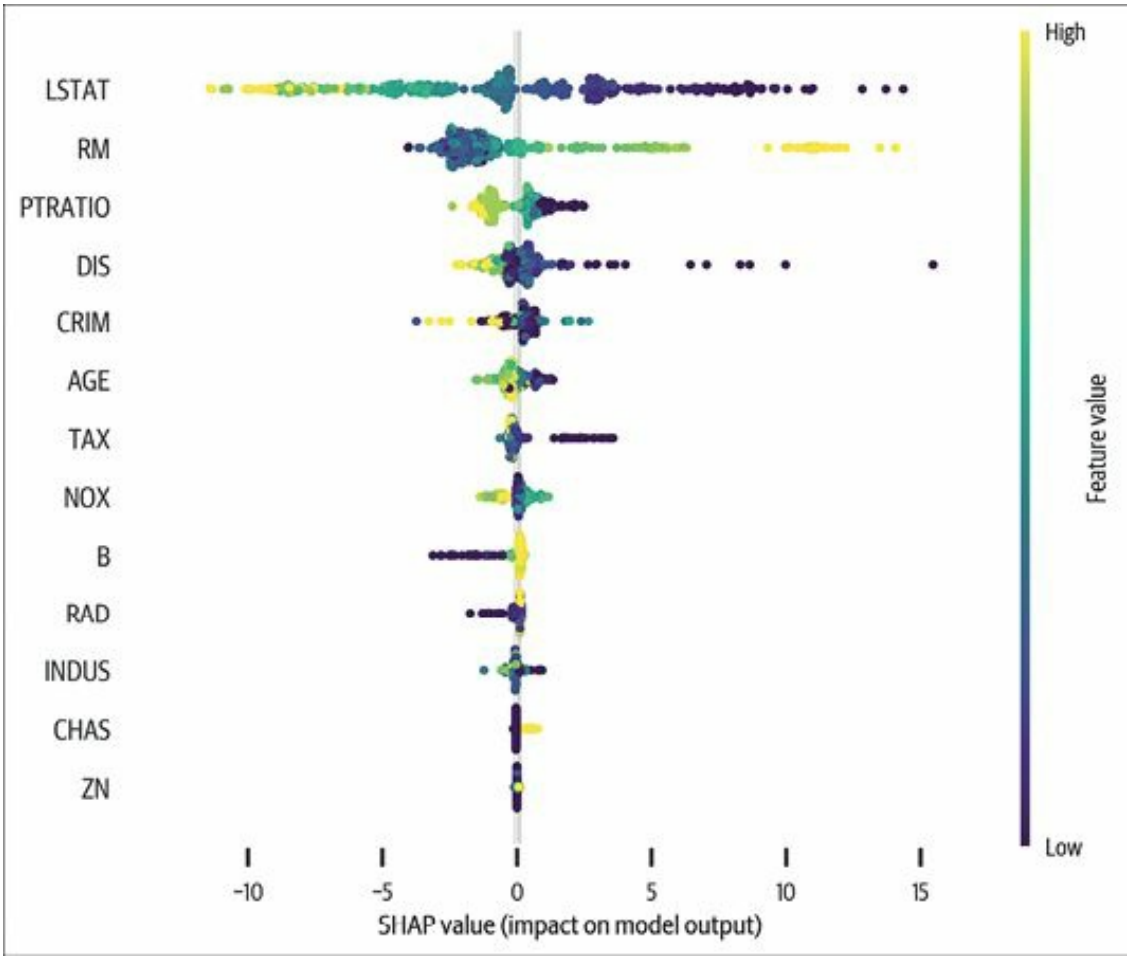


Figura 16.5 – Gráfico de resumo. Os atributos mais importantes estão na parte superior.

Os atributos na parte superior causam mais impacto no modelo. A partir dessa visualização, podemos notar que valores maiores de RM (número de cômodos) empurram bastante o valor do alvo para cima, enquanto valores medianos e menores empurram o valor um pouco para baixo:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> shap.summary_plot(vals, bos_X)
>>> fig.savefig(
... "images/mlpr_1605.png",
... bbox_inches="tight",
... dpi=300,
... )
```

A biblioteca SHAP é uma ótima ferramenta para você ter em seu cinto de utilidades. Ela ajuda a compreender o impacto global dos atributos, além de

ajudar a explicar predições individuais.

CAPÍTULO 17

Redução da dimensionalidade

Há várias técnicas para decompor atributos em um subconjunto menor. Isso pode ser conveniente para uma análise de dados exploratória, visualização, criação de modelos preditivos ou clustering (agrupamento).

Neste capítulo, exploraremos o conjunto de dados do Titanic usando várias técnicas. Veremos as técnicas de PCA, UMAP, t-SNE e PHATE.

Eis os dados que usaremos:

```
>>> ti_df = tweak_titanic(orig_df)
>>> std_cols = "pclass,age,sibsp,fare".split(",")
>>> X_train, X_test, y_train, y_test = get_train_test_X_y(
... ti_df, "survived", std_cols=std_cols
... )
>>> X = pd.concat([X_train, X_test])
>>> y = pd.concat([y_train, y_test])
```

PCA

A PCA (Principal Component Analysis, ou Análise de Componentes Principais) usa uma matriz (X) de linhas (amostras) e colunas (atributos). Ela devolve uma nova matriz cujas colunas são combinações lineares das colunas originais. Essas combinações lineares maximizam a variância.

Cada coluna é ortogonal às demais colunas (tem um ângulo reto). As colunas são ordenadas de acordo com uma variância decrescente.

O scikit-learn tem uma implementação para esse modelo. É melhor padronizar os dados antes de executar o algoritmo. Depois de chamar o método `.fit`, teremos acesso a um atributo `.explained_variance_ratio_` que lista o percentual de variância em cada coluna.

A PCA é conveniente para visualizar dados em duas (ou três) dimensões. É usada também como um passo de pré-processamento para filtrar ruídos aleatórios nos dados. É adequada para identificar estruturas globais mas não

locais, e funciona bem com dados lineares.

Neste exemplo, executaremos a PCA nos atributos do Titanic. A classe PCA é um *transformador* (transformer) no scikit-learn; você deve chamar o método .fit para ensiná-lo a obter os componentes principais e, em seguida, chamar .transform para converter uma matriz em uma matriz de componentes principais:

```
>>> from sklearn.decomposition import PCA
>>> from sklearn.preprocessing import (
... StandardScaler,
... )
>>> pca = PCA(random_state=42)
>>> X_pca = pca.fit_transform(
... StandardScaler().fit_transform(X)
... )
>>> pca.explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
  0.10460882, 0.08170342, 0.07229959,
  0.05133752, 0.04199068])

>>> pca.components_[0]
arrayarray([-0.63368693, 0.39682566,
  0.00614498, 0.11488415, 0.58075352,
  -0.19046812, -0.21190808, -0.09631388])
```

Parâmetros da instância:

n_components=None

Número de componentes a serem gerados. Se for igual a None, devolve o mesmo número correspondente ao número de colunas. Pode ser um número de ponto flutuante de (0,1); nesse caso, serão criados tantos componentes quantos forem necessários para obter essa razão de variância.

copy=True

Modificará os dados em .fit se for True.

whiten=False

Limpará os dados após a transformação para garantir componentes não correlacionados.

svd_solver='auto'

'auto' executa SVD 'randomized' se n_components for menor que 80% da menor dimensão (mais rápido, porém é uma aproximação). Caso contrário, executa

'full'.

tol=0.0

Tolerância para valores únicos.

iterated_power='auto'

Número de iterações para svd_solver 'randomized'.

random_state=None

Estado aleatório para svd_solver 'randomized'.

Atributos:

components_

Componentes principais (colunas de pesos de combinações lineares para atributos originais.)

explained_variance_

Valor de variância para cada componente.

explained_variance_ratio_

Valor de variância para cada componente normalizado (a soma é 1).

singular_values_

Valores únicos para cada componente.

mean_

Média de cada atributo.

n_components_

Se n_components for um número de ponto flutuante, esse será o número de componentes.

noise_variance_

Covariância estimada de ruído.

O gráfico da soma cumulativa da razão de variância explicada é chamado de *gráfico de declive* (scree plot) (veja a Figura 17.1).

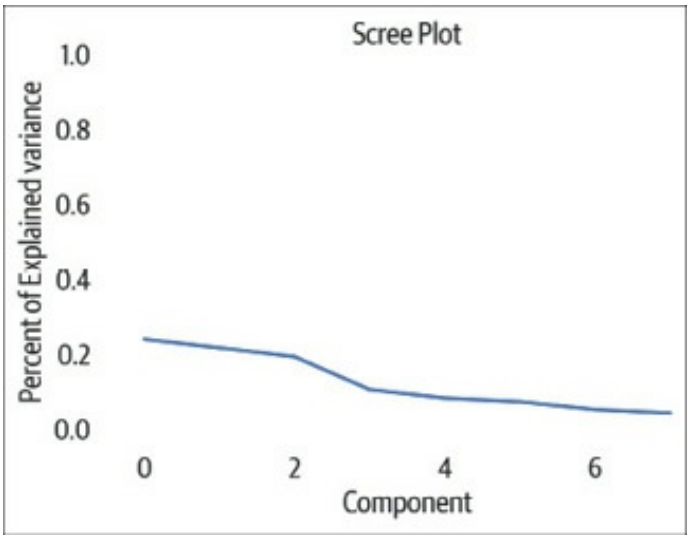


Figura 17.1 – Gráfico de declive para PCA.

Esse gráfico mostra quanta informação está armazenada nos componentes. O *método do cotovelo* (elbow method) pode ser utilizado para ver se ele se curva, a fim de determinar quantos componentes devem ser usados:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(pca.explained_variance_ratio_)
>>> ax.set(
... xlabel="Component",
... ylabel="Percent of Explained variance",
... title="Scree Plot",
... ylim=(0, 1),
... )
>>> fig.savefig(
... "images/mlpr_1701.png",
... dpi=300,
... bbox_inches="tight",
... )
```

Outra maneira de visualizar esses dados é usar um gráfico cumulativo (veja a Figura 17.2). Nossos dados originais tinham 8 colunas, mas, observando o gráfico, parece que manteremos aproximadamente 90% da variância se usarmos apenas 4 dos componentes PCA:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> ax.plot(
... np.cumsum(pca.explained_variance_ratio_)
... )
>>> ax.set(
... xlabel="Component",
... ylabel="Percent of Explained variance",
```

```
... title="Cumulative Variance",
... ylim=(0, 1),
... )
>>> fig.savefig("images/mlpr_1702.png", dpi=300)
```

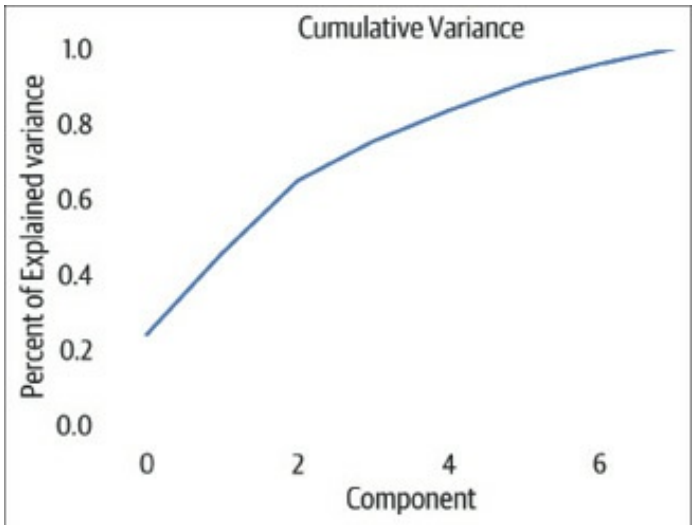


Figura 17.2 – Variância explicada cumulativa da PCA.

Em que medida os atributos causam impacto nos componentes? Utilize a função imshow da matplotlib para gerar um gráfico com os componentes no eixo x e os atributos originais no eixo y (veja a Figura 17.3). Quanto mais escura for a cor, maior será a contribuição da coluna original ao componente.

Parece que o primeiro componente é bastante influenciado pelas colunas pclass (classe do passageiro), age (idade) e fare (preço da passagem). (Usar o mapa de cores espectral (cmap) enfatiza valores diferentes de zero, e especificar vmin e vmax acrescenta limites à legenda de cores.)

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> plt.imshow(
... pca.components_.T,
... cmap="Spectral",
... vmin=-1,
... vmax=1,
... )
>>> plt.yticks(range(len(X.columns)), X.columns)
>>> plt.xticks(range(8), range(1, 9))
>>> plt.xlabel("Principal Component")
>>> plt.ylabel("Contribution")
>>> plt.title(
... "Contribution of Features to Components"
... )
```

```
>>> plt.colorbar()
>>> fig.savefig("images/mlpr_1703.png", dpi=300)
```

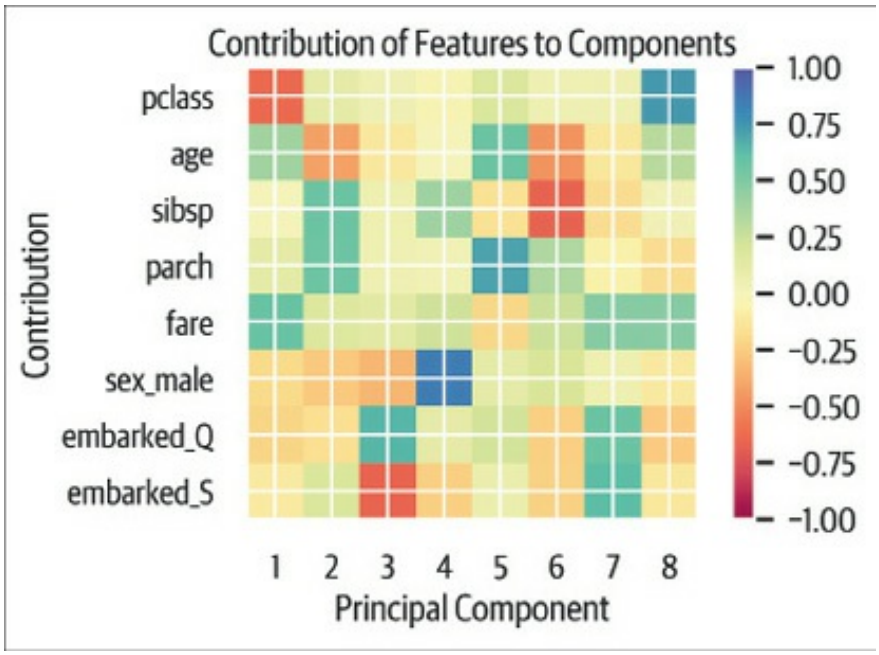


Figura 17.3 – Atributos da PCA nos componentes.

Uma alternativa de visualização é observar um gráfico de barras (veja a Figura 17.4). Cada componente é exibido com as contribuições dos dados originais:

```
>>> fig, ax = plt.subplots(figsize=(8, 4))
>>> pd.DataFrame(
... pca.components_, columns=X.columns
... ).plot(kind="bar", ax=ax).legend(
... bbox_to_anchor=(1, 1)
... )
>>> fig.savefig("images/mlpr_1704.png", dpi=300)
```

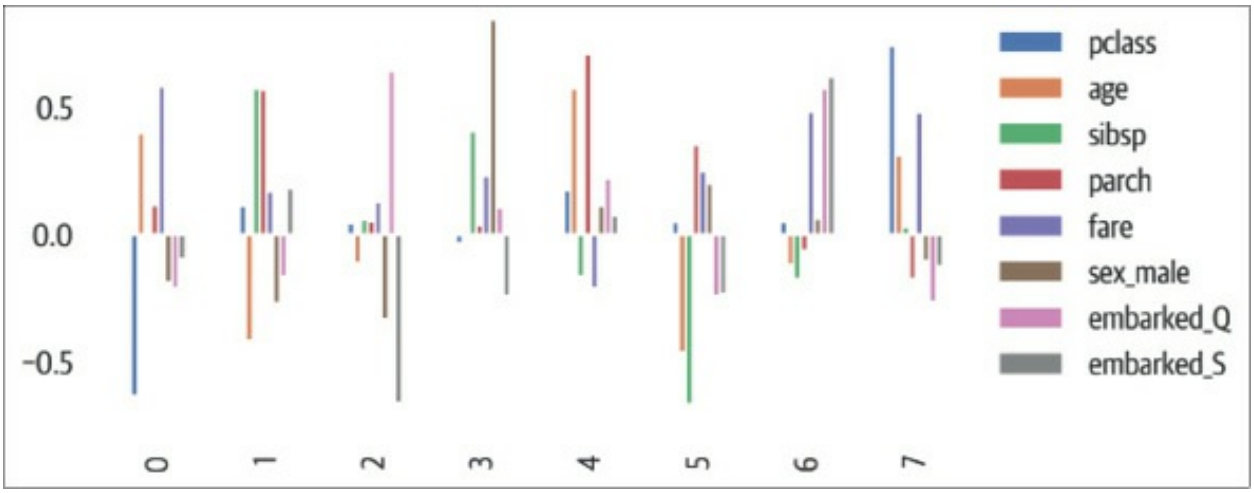


Figura 17.4 – Atributos da PCA nos componentes.

Se tivermos vários atributos, podemos limitar os gráficos anteriores, mostrando apenas os atributos que atendam a um peso mínimo. Eis o código para encontrar todos os atributos nos dois primeiros componentes, cujos valores absolutos sejam no mínimo 0,5:

```
>>> comps = pd.DataFrame(
... pca.components_, columns=X.columns
... )
>>> min_val = 0.5
>>> num_components = 2
>>> pca_cols = set()
>>> for i in range(num_components):
... parts = comps.iloc[i][
... comps.iloc[i].abs() > min_val
... ]
... pca_cols.update(set(parts.index))
>>> pca_cols
{'fare', 'parch', 'pclass', 'sibsp'}
```

A PCA é geralmente usada para visualizar conjuntos de dados com muitas dimensões em dois componentes. Nesse caso, visualizamos os atributos do Titanic em 2D. Eles foram coloridos de acordo com o status de sobrevivência. Às vezes, podem aparecer clusters (agrupamentos) na visualização. Em nosso caso, não parece haver clusterings de sobreviventes (veja a Figura 17.5).

Vamos gerar essa visualização usando o Yellowbrick:

```
>>> from yellowbrick.features.pca import (
... PCADecomposition,
... )
```

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> pca_viz = PCADecomposition(color=colors)
>>> pca_viz.fit_transform(X, y)
>>> pca_viz.poof()
>>> fig.savefig("images/mlpr_1705.png", dpi=300)
```

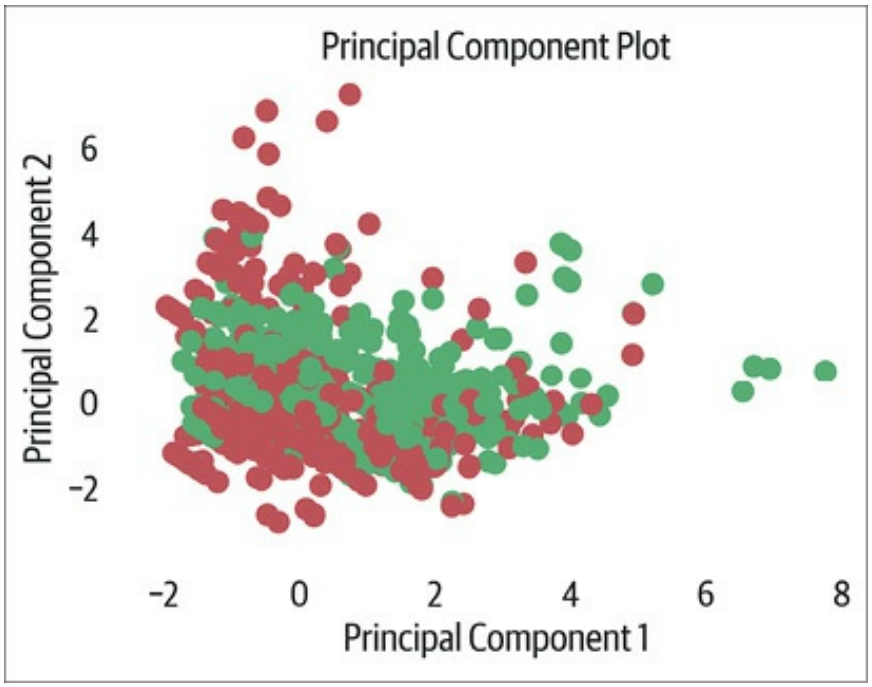


Figura 17.5 – Gráfico de PCA gerado com o Yellowbrick.

Se quiser colorir o gráfico de dispersão de acordo com uma coluna e acrescentar uma legenda (não uma barra de cores), é necessário percorrer cada cor em um laço e gerar o gráfico desse grupo individualmente usando o pandas ou a matplotlib (ou o seaborn). A seguir, também definimos a taxa de aspecto com a razão das variâncias explicadas para os componentes que estamos observando (veja a Figura 17.6). Como o segundo componente tem apenas 90% do primeiro, ele é um pouco menor.

Eis uma versão com o seaborn:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
... X_pca,
... columns=[
... f"PC{i+1}"
... for i in range(X_pca.shape[1])
... ],
... )
```

```
>>> pca_df["status"] = [
... ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[1] / evr[0])
>>> sns.scatterplot(
... x="PC1",
... y="PC2",
... hue="status",
... data=pca_df,
... alpha=0.5,
... ax=ax,
... )
>>> fig.savefig(
... "images/mlpr_1706.png",
... dpi=300,
... bbox_inches="tight",
... )
```

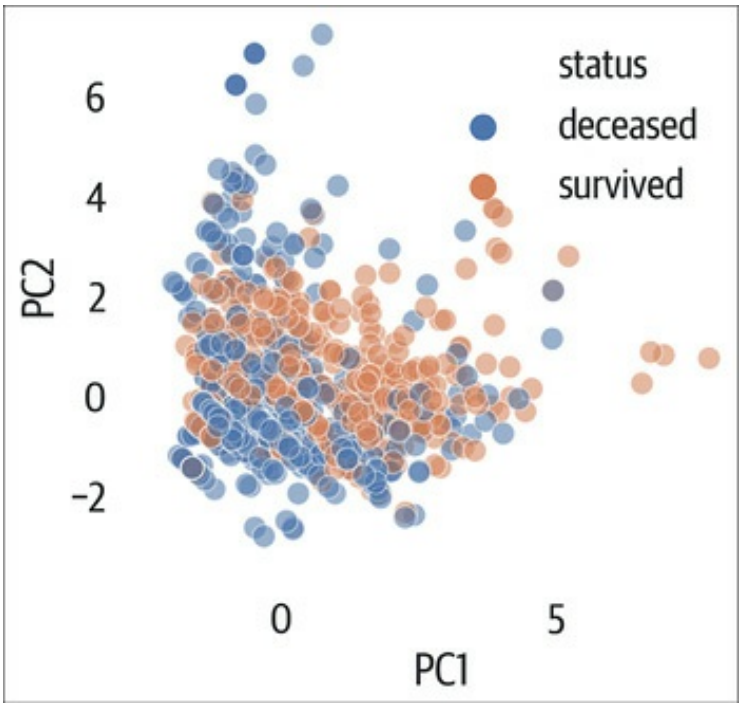


Figura 17.6 – PCA no seaborn com legenda e aspecto relativo.

A seguir, expandiremos o gráfico de dispersão mostrando um *gráfico de carga* (loading plot) sobre ele. Esse gráfico é chamado de gráfico duplo (biplot) porque contém o gráfico de dispersão e de cargas (veja a Figura 17.7).

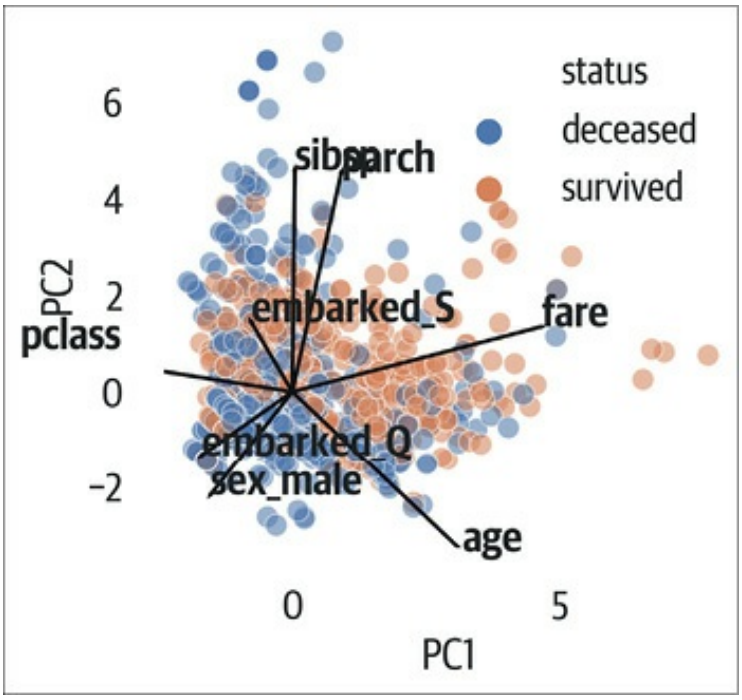


Figura 17.7 – Gráfico duplo do seaborn, com um gráfico de dispersão e um gráfico de cargas.

As cargas mostram a força dos atributos e como estão correlacionados. Se formarem ângulos agudos, é sinal de que provavelmente estão correlacionados. Se os ângulos forem de 90 graus, é provável que não estejam correlacionados. Por fim, se o ângulo entre eles estiver próximo de 180 graus, haverá uma correlação negativa:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
... X_pca,
... columns=[
... f"PC{i+1}"
... for i in range(X_pca.shape[1])
... ],
... )
>>> pca_df["status"] = [
... ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> x_idx = 0 # x_pc
>>> y_idx = 1 # y_pc
>>> ax.set_aspect(evr[y_idx] / evr[x_idx])
>>> x_col = pca_df.columns[x_idx]
>>> y_col = pca_df.columns[y_idx]
```

```
>>> sns.scatterplot(
... x=x_col,
... y=y_col,
... hue="status",
... data=pca_df,
... alpha=0.5,
... ax=ax,
... )
>>> scale = 8
>>> comps = pd.DataFrame(
... pca.components_, columns=X.columns
... )
>>> for idx, s in comps.T.iterrows():
... plt.arrow(
... 0,
... 0,
... s[x_idx] * scale,
... s[y_idx] * scale,
... color="k",
... )
... plt.text(
... s[x_idx] * scale,
... s[y_idx] * scale,
... idx,
... weight="bold",
... )
>>> fig.savefig(
... "images/mlpr_1707.png",
... dpi=300,
... bbox_inches="tight",
... )
```

Com base nos modelos de árvore anteriores, sabemos que age (idade), fare (preço da passagem) e sex (sexo) são importantes para determinar se um passageiro sobreviveu. O primeiro componente principal é influenciado por pclass (classe do passageiro), age (idade) e fare (preço da passagem), enquanto o quarto é influenciado por sex (sexo). Vamos colocar esses componentes em um gráfico, uns em relação aos outros.

Novamente, esse gráfico escala a taxa de aspecto do gráfico com base nas razões da variância dos componentes (veja a Figura 17.8).

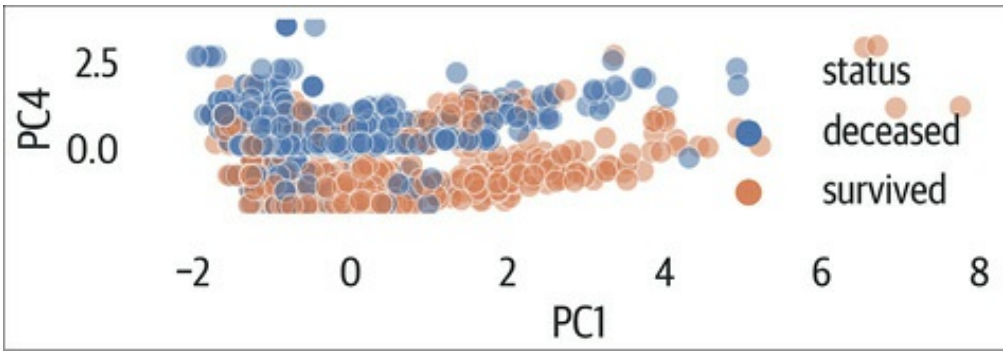


Figura 17.8 – Gráfico de PCA mostrando o componente 1 em relação ao 4.

O gráfico parece separar mais claramente os sobreviventes:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pca_df = pd.DataFrame(
... X_pca,
... columns=[
... f"PC{i+1}"
... for i in range(X_pca.shape[1])
... ],
... )
>>> pca_df["status"] = [
... ("deceased", "survived")[i] for i in y
... ]
>>> evr = pca.explained_variance_ratio_
>>> ax.set_aspect(evr[3] / evr[0])
>>> sns.scatterplot(
... x="PC1",
... y="PC4",
... hue="status",
... data=pca_df,
... alpha=0.5,
... ax=ax,
... )
>>> fig.savefig(
... "images/mlpr_1708.png",
... dpi=300,
... bbox_inches="tight",
... )
```

A matplotlib é capaz de criar gráficos elegantes, mas é menos útil para gráficos interativos. Ao usar a PCA, em geral será conveniente visualizar os dados em gráficos de dispersão. Incluí uma função que utiliza a biblioteca Bokeh (*https://bokeh.pydata.org*) para interagir com gráficos de dispersão (veja a Figura 17.9). Ela funciona bem com o Jupyter:

```
>>> from bokeh.io import output_notebook
>>> from bokeh import models, palettes, transform
>>> from bokeh.plotting import figure, show
>>>
>>> def bokeh_scatter(
... x,
... y,
... data,
... hue=None,
... label_cols=None,
... size=None,
... legend=None,
... alpha=0.5,
... ):
... """
... x - x column name to plot
... y - y column name to plot
... data - pandas DataFrame
... hue - column name to color by (numeric)
... legend - column name to label by
... label_cols - columns to use in tooltip
... (None all in DataFrame)
... size - size of points in screen space unigs
... alpha - transparency
... """
... output_notebook()
... circle_kwargs = {}
... if legend:
... circle_kwargs["legend"] = legend
... if size:
... circle_kwargs["size"] = size
... if hue:
... color_seq = data[hue]
... mapper = models.LinearColorMapper(
... palette=palettes.viridis(256),
... low=min(color_seq),
... high=max(color_seq),
... )
... circle_kwargs[
... "fill_color"
... ] = transform.transform(hue, mapper)
... ds = models.ColumnDataSource(data)
... if label_cols is None:
... label_cols = data.columns
... tool_tips = sorted(
```

```
... [
... (x, "@{}".format(x))
... for x in label_cols
... ],
... key=lambda tup: tup[0],
... )
... hover = models.HoverTool(
... tooltips=tool_tips
... )
... fig = figure(
... tools=[
... hover,
... "pan",
... "zoom_in",
... "zoom_out",
... "reset",
... ],
... toolbar_location="below",
... )
...
... fig.circle(
... x,
... y,
... source=ds,
... alpha=alpha,
... **circle_kwargs
... )
... show(fig)
... return fig
>>> res = bokeh_scatter(
... "PC1",
... "PC2",
... data=pca_df.assign(
... surv=y.reset_index(drop=True)
... ),
... hue="surv",
... size=10,
... legend="surv",
... )
```

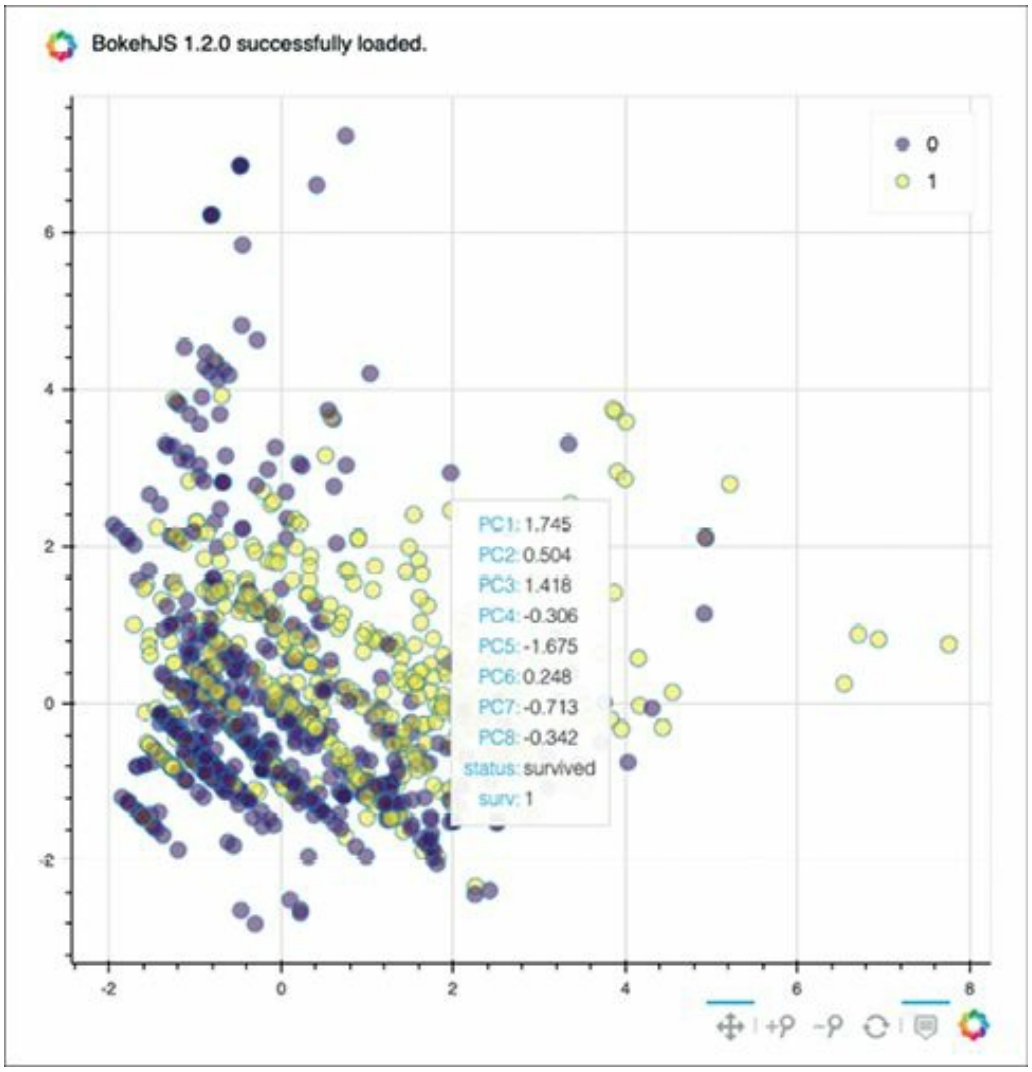


Figura 17.9 – Gráfico de dispersão do Bokeh com dicas de contexto (tooltips).

O Yellowbrick também é capaz de gerar gráficos em três dimensões (veja a Figura 17.10):

```
>>> from yellowbrick.features.pca import (
... PCADecomposition,
... )
>>> colors = ["rg"[j] for j in y]
>>> pca3_viz = PCADecomposition(
... proj_dim=3, color=colors
... )
>>> pca3_viz.fit_transform(X, y)
>>> pca3_viz.finalize()
>>> fig = plt.gcf()
>>> plt.tight_layout()
```

```
>>> fig.savefig(
... "images/mlpr_1710.png",
... dpi=300,
... bbox_inches="tight",
... )
```

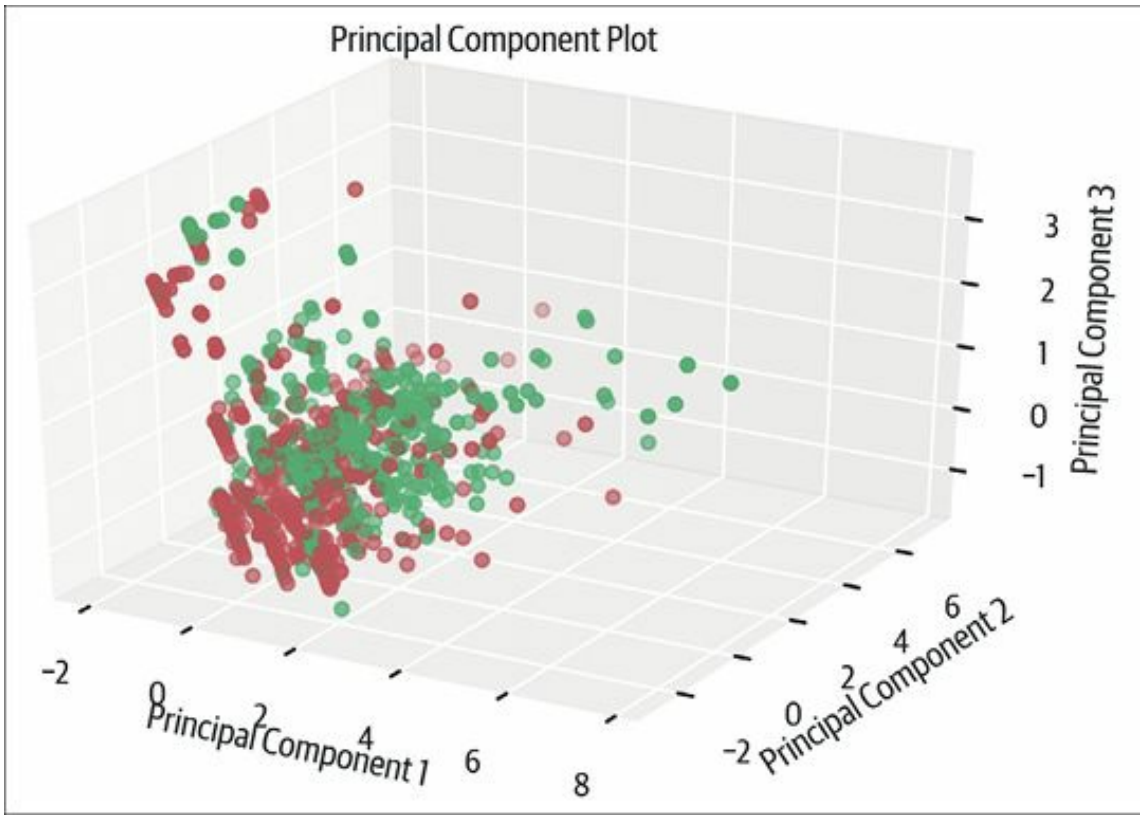


Figura 17.10 – PCA 3D gerada com o Yellowbrick.

A biblioteca scprep (*https://oreil.ly/Jdq1s*) (uma dependência da biblioteca PHATE, que será discutida em breve) tem uma função conveniente para gerar gráfico. A função rotate_scatter3d é capaz de gerar um gráfico animado no Jupyter (veja a Figura 17.11). Isso facilita compreender gráficos 3D.

Essa biblioteca pode ser usada para visualizar quaisquer dados 3D, e não apenas o PHATE:

```
>>> import scprep
>>> scprep.plot.rotate_scatter3d(
... X_pca[:, :3],
... c=y,
... cmap="Spectral",
... figsize=(8, 6),
... label_prefix="Principal Component",
... )
```

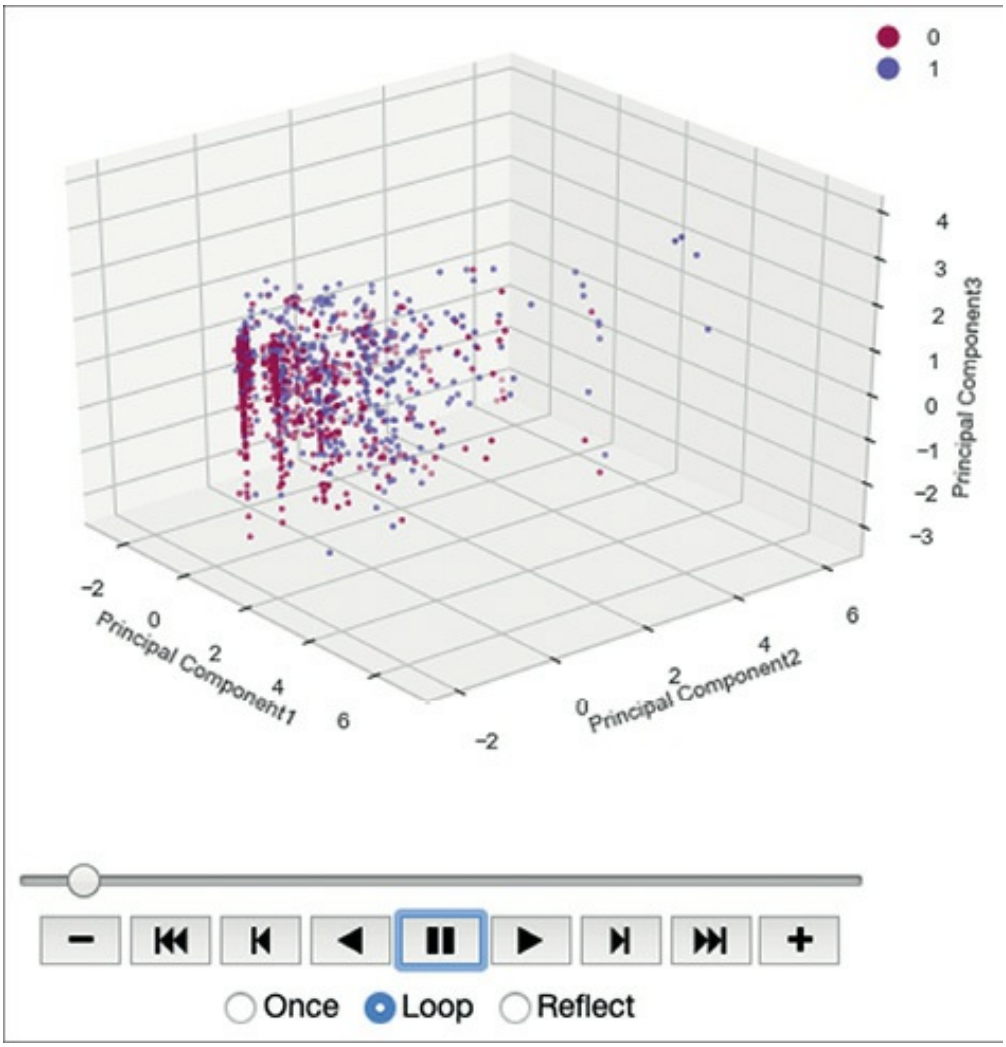


Figura 17.11 – Animação de PCA 3D com o scprep.

Se você modificar o modo mágico de célula da matplotlib no Jupyter para notebook, poderá obter um gráfico 3D interativo da matplotlib (veja a Figura 17.12).

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure(figsize=(6, 4))
>>> ax = fig.add_subplot(111, projection="3d")
>>> ax.scatter(
... xs=X_pca[:, 0],
... ys=X_pca[:, 1],
... zs=X_pca[:, 2],
... c=y,
... cmap="viridis",
... )
>>> ax.set_xlabel("PC 1")
>>> ax.set_ylabel("PC 2")
```

```
>>> ax.set_zlabel("PC 3")
```

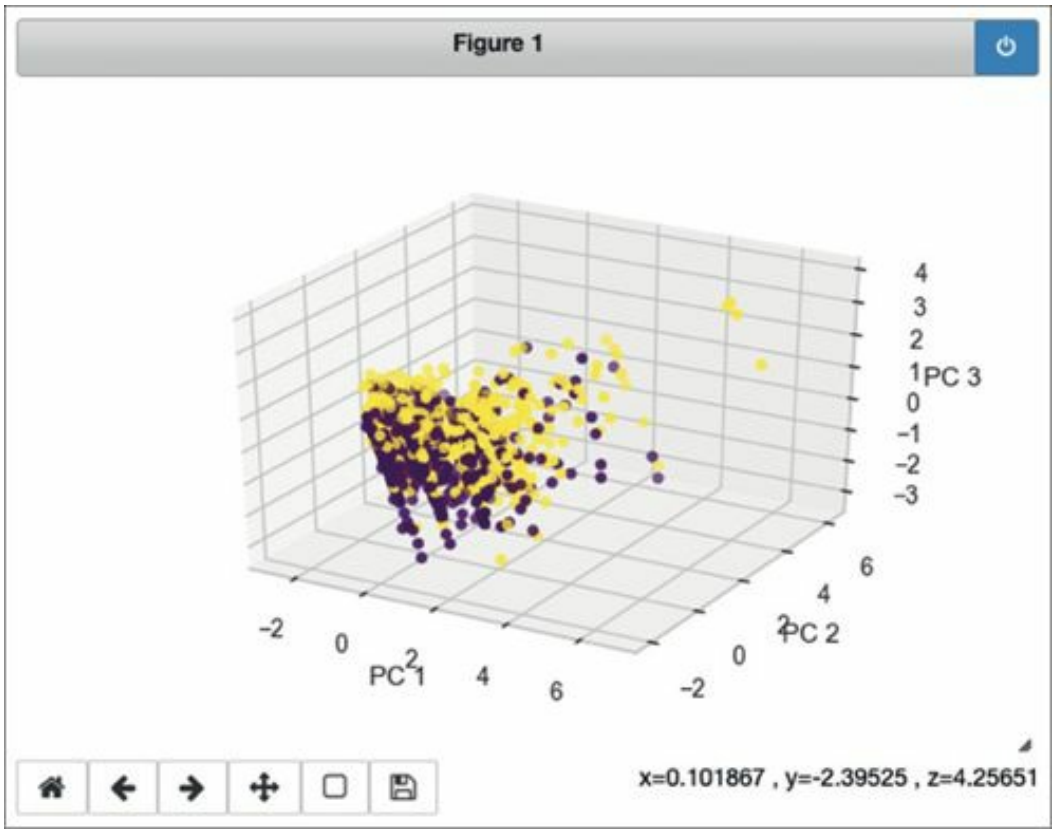


Figura 17.12 – PCA 3D interativa da matplotlib com o modo notebook.

ALERTA

Observe que mudar o modo mágico de célula da matplotlib no Jupyter de:

```
% matplotlib inline
```

para:

```
% matplotlib notebook
```

às vezes pode fazer o Jupyter parar de responder. Utilize com cuidado.

UMAP

O UMAP (Uniform Manifold Approximation and Projection, ou Aproximação e Projeção Uniforme de Variedades) (*https://oreil.ly/qF8RJ*) é uma técnica de redução de dimensionalidade que utiliza manifold learning (aprendizado de variedades). Assim, ele tende a manter itens similares topologicamente juntos e procura preservar tanto a estrutura global como

local, em oposição ao t-SNE (será explicado na seção “t-SNE”), que favorece a estrutura local.

A implementação Python não oferece suporte para várias cores (núcleos) de CPU.

Fazer a normalização dos atributos é uma boa ideia para ter os valores na mesma escala.

O UMAP é muito sensível a hiperparâmetros (_neighbors, min_dist, n_components ou metric). Eis alguns exemplos:

```
>>> import umap
>>> u = umap.UMAP(random_state=42)
>>> X_umap = u.fit_transform(
... StandardScaler().fit_transform(X)
... )
>>> X_umap.shape
(1309, 2)
```

Parâmetros da instância:

n_neighbors=15

Tamanho da vizinhança local. Valores maiores implicam uma visão global, enquanto valores menores significam uma visão mais local.

n_components=2

Número de dimensões para embedding.

metric='euclidean'

Métrica a ser usada para distância. Pode ser uma função que aceite dois arrays 1D e devolva um número de ponto flutuante.

n_epochs=None

Número de epochs (épocas) de treinamento. O default será 200 ou 500 (dependendo do tamanho dos dados).

learning_rate=1.0

Taxa de aprendizagem para otimização de embedding.

init='spectral'

Tipo de inicialização. Embedding espectral é o default. Pode ser 'random' ou um array numpy de posições.

min_dist=0.1

Entre 0 e 1. Distância mínima entre pontos embutidos. Valores menores significam mais aglomerações, valores maiores implicam mais dispersão.

spread=1.0

Determina a distância dos pontos após o embedding.

set_op_mix_ratio=1.0

Entre 0 e 1: união fuzzy (1) ou intersecção fuzzy (0).

local_connectivity=1.0

Número de vizinhos para conectividade local. À medida que esse valor aumentar, mais conexões locais serão criadas.

repulsion_strength=1.0

Força de repulsão. Valores maiores dão mais peso às amostras negativas.

negative_sample_rate=5

Amostras negativas por amostra positiva. Valores maiores apresentam mais repulsão, mais custos de otimização e mais precisão.

transform_queue_size=4.0

Agressividade para busca de vizinhos mais próximos. Um valor maior tem pior desempenho, mas terá mais precisão.

a=None

Parâmetro para controle de embedding. Se for igual a None, o UMAP determinará o valor a partir de min_dist e spread.

b=None

Parâmetro para controle de embedding. Se for igual a None, o UMAP determinará o valor a partir de min_dist e spread.

random_state=None

Semente (seed) aleatória.

metric_kwds=None

Dicionário de métricas para parâmetros adicionais se uma função for usada para metric. Além disso, minkowsi (e outras métricas) pode ser parametrizado com essa informação.

angular_rp_forest=False

Usa projeção angular aleatória.

target_n_neighbors=-1

Número de vizinhos para conjunto simpliciais.

target_metric='categorical'

Para usar redução supervisionada. Também pode ser 'L1' ou 'L2'. Além disso, aceita uma função que recebe dois arrays de X como entrada e devolve o valor da distância entre eles.

target_metric_kwds=None

Dicionário de métricas a ser utilizado caso uma função seja usada em target_metric.

target_weight=0.5

Fator de ponderação. Entre 0,0 e 1,0, em que 0 significa basear-se somente nos dados e 1 significa basear-se somente no alvo.

transform_seed=42

Semente aleatória para operações de transformação.

verbose=False

Verbosidade.

Atributos:

embedding_

Resultados do embedding.

Vamos visualizar o resultado default do UMAP no conjunto de dados do Titanic (veja a Figura 17.13):

```
>>> fig, ax = plt.subplots(figsize=(8, 4))
>>> pd.DataFrame(X_umap).plot(
... kind="scatter",
... x=0,
... y=1,
... ax=ax,
... c=y,
... alpha=0.2,
... cmap="Spectral",
... )
>>> fig.savefig("images/mlpr_1713.png", dpi=300)
```

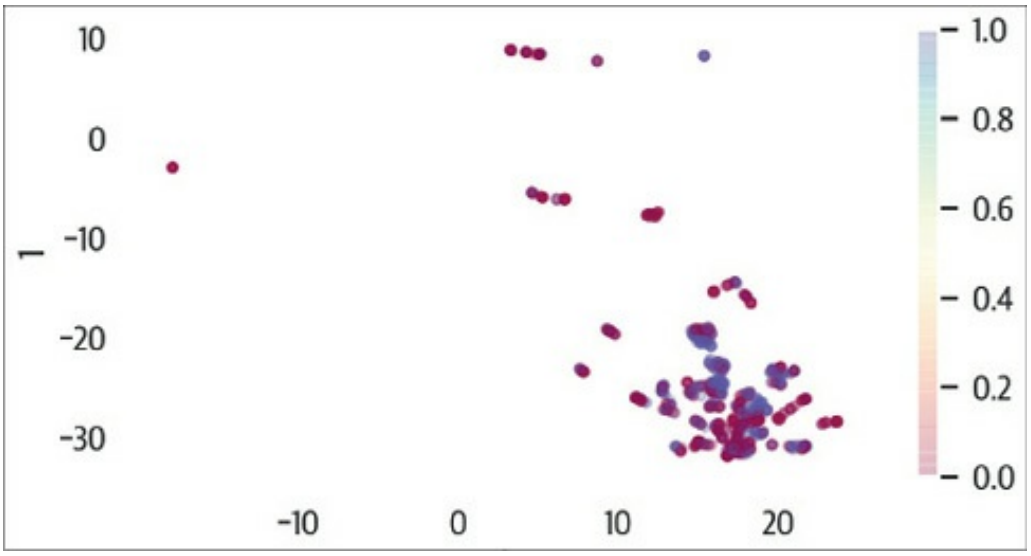


Figura 17.13 – Resultados do UMAP.

Para ajustar o resultado do UMAP, concentre-se nos hiperparâmetros n_neighbors e min_dist antes. Seguem exemplos de mudanças nesses valores (veja as figuras 17.14 e 17.15).

```
>>> X_std = StandardScaler().fit_transform(X)
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([2, 5, 10, 50]):
... ax = axes[i]
... u = umap.UMAP(
... random_state=42, n_neighbors=n
... )
... X_umap = u.fit_transform(X_std)
...
... pd.DataFrame(X_umap).plot(
... kind="scatter",
... x=0,
... y=1,
... ax=ax,
... c=y,
... cmap="Spectral",
... alpha=0.5,
... )
... ax.set_title(f"nn={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1714.png", dpi=300)
```

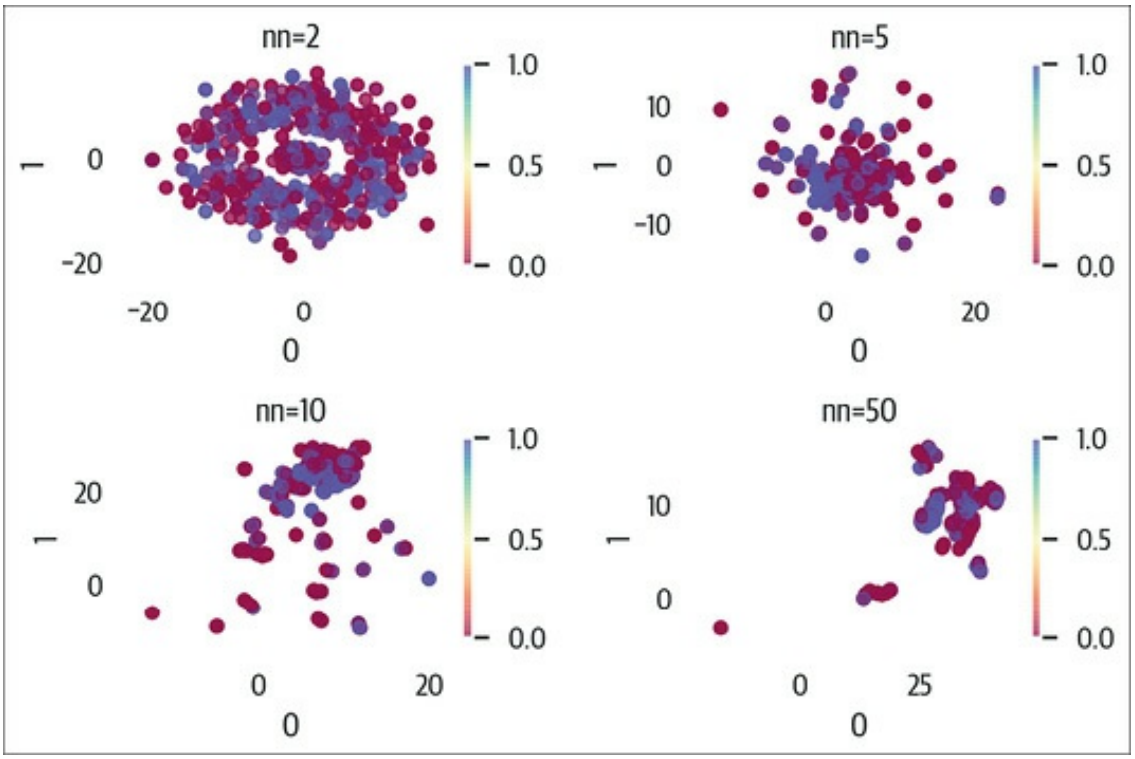


Figura 17.14 – Resultados do UMAP ajustando n_neighbors**.**

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate([0, 0.33, 0.66, 0.99]):
... ax = axes[i]
... u = umap.UMAP(random_state=42, min_dist=n)
... X_umap = u.fit_transform(X_std)
... pd.DataFrame(X_umap).plot(
... kind="scatter",
... x=0,
... y=1,
... ax=ax,
... c=y,
... cmap="Spectral",
... alpha=0.5,
... )
... ax.set_title(f"min_dist={n}")
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1715.png", dpi=300)
```

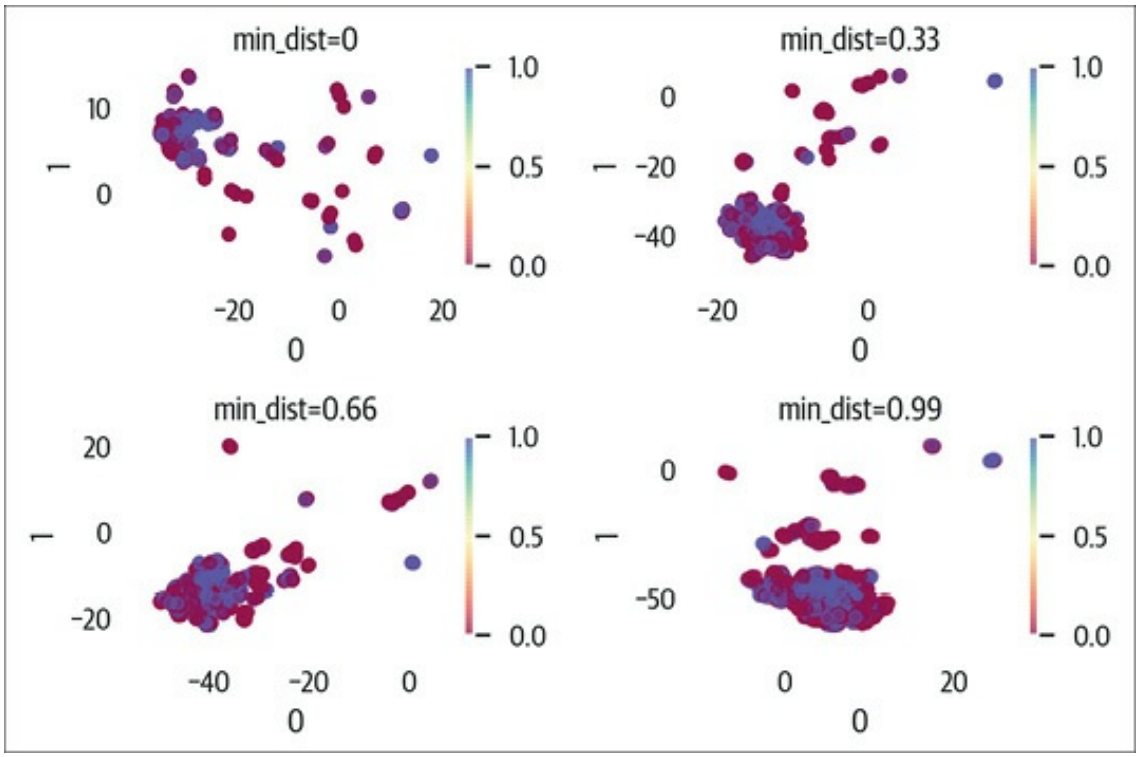


Figura 17.15 – Resultados do UMAP ajustando min_dist.

Às vezes, a PCA é executada antes do UMAP para reduzir as dimensões e agilizar o processamento.

t-SNE

A técnica de t-SNE (t-Distributed Stochastic Neighboring Embedding) é uma técnica de visualização e de redução de dimensionalidade. Utiliza distribuições dos dados de entrada e embedding para ter menos dimensões, e minimiza as probabilidades de junções entre elas. Por exigir bastante do ponto de vista de processamento, talvez não seja possível usar essa técnica com um conjunto de dados grande.

Uma característica do t-SNE é que ele é muito sensível a hiperparâmetros. Além disso, embora preserve clusters locais muito bem, as informações globais não são preservadas. Desse modo, a distância entre os clusters não é significativa. Por fim, esse não é um algoritmo determinístico, e talvez não haja convergência.

Padronizar os dados antes de usar essa técnica é uma boa ideia:

```
>>> from sklearn.manifold import TSNE
>>> X_std = StandardScaler().fit_transform(X)
```

```
>>> ts = TSNE()
>>> X_tsne = ts.fit_transform(X_std)
```

Parâmetros da instância:

n_components=2

Número de dimensões para embedding.

perplexity=30.0

Valores sugeridos estão entre 5 e 50. Números menores tendem a criar aglomerações mais rígidas.

early_exaggeration=12.0

Controla a rigidez dos clusters e o espaçamento entre eles. Valores maiores significam espaçamentos maiores.

learning_rate=200.0

Em geral, entre 10 e 1000. Se os dados se assemelharem a uma bola, reduza esse valor. Se parecerem compactados, eleve-o.

n_iter=1000

Número de iterações.

n_iter_without_progress=300

Aborta se não houver progressos após esse número de iterações.

min_grad_norm=1e-07

A otimização será interrompida se a norma de gradiente estiver abaixo desse valor.

metric='euclidean'

Métrica de distância de scipy.spatial.distance.pdist, pairwise.PAIRWISE_DISTANCE_METRIC ou uma função.

init='random'

Inicialização de embedding.

verbose=0

Verbosidade.

random_state=None

Semente (seed) aleatória.

method='barnes_hut'

Algoritmo de cálculo de gradiente.

angle=0.5

Para cálculo de gradiente. Um valor menor que 0,2 aumenta o tempo de execução. Se for maior que 0,8, aumentará o erro.

Atributos:

embedding_

Vetores de embedding.

kl_divergence_

Divergência de Kullback-Leibler.

n_iter_

Número de iterações.

Eis uma visualização dos resultados do t-SNE usando a matplotlib (veja a Figura 17.16):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> colors = ["rg"[j] for j in y]
>>> scat = ax.scatter(
... X_tsne[:, 0],
... X_tsne[:, 1],
... c=colors,
... alpha=0.5,
... )
>>> ax.set_xlabel("Embedding 1")
>>> ax.set_ylabel("Embedding 2")
>>> fig.savefig("images/mlpr_1716.png", dpi=300)
```

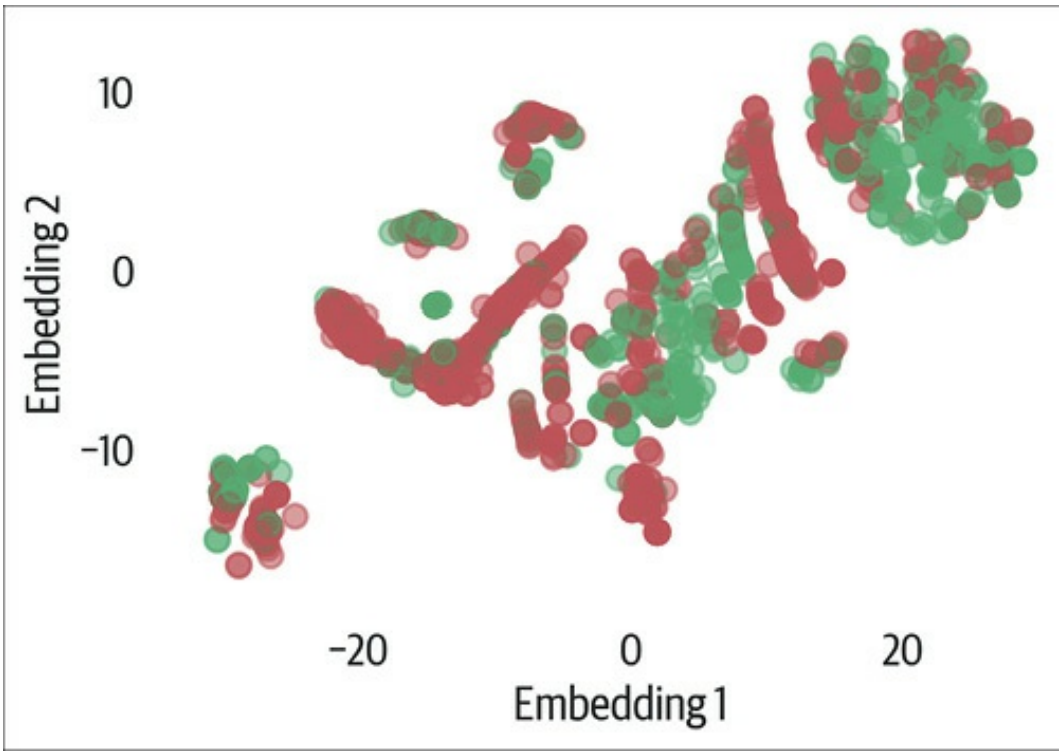


Figura 17.16 – Resultado do t-SNE gerado com a matplotlib.

Alterar o valor de perplexity pode provocar grandes efeitos no gráfico (veja a Figura 17.17). A seguir, usamos alguns valores distintos:

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> for i, n in enumerate((2, 30, 50, 100)):
... ax = axes[i]
... t = TSNE(random_state=42, perplexity=n)
... X_tsne = t.fit_transform(X)
... pd.DataFrame(X_tsne).plot(
... kind="scatter",
... x=0,
... y=1,
... ax=ax,
... c=y,
... cmap="Spectral",
... alpha=0.5,
... )
... ax.set_title(f"perplexity={n}")
... plt.tight_layout()
... fig.savefig("images/mlpr_1717.png", dpi=300)
```

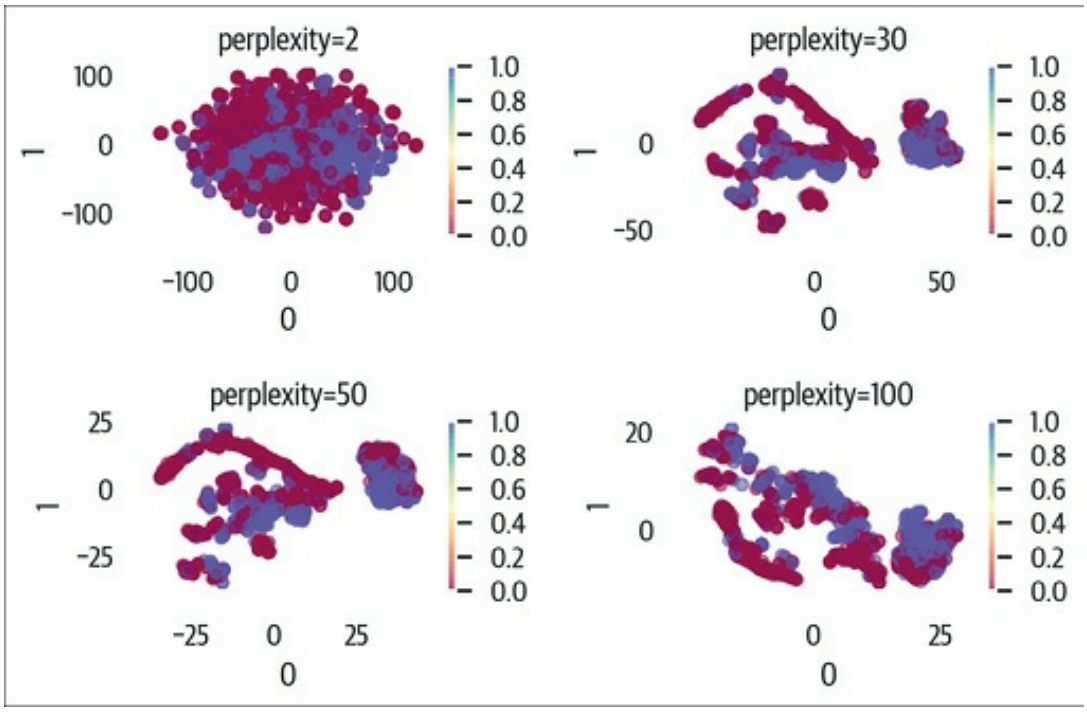


Figura 17.17 – Modificando perplexity *para o t-SNE.*

PHATE

O PHATE (Potential of Heat-diffusion for Affinity-based Trajectory Embedding) (*https://phate.readthedocs.io*) é uma ferramenta para visualização de dados com muitas dimensões. Ele tende a manter tanto a estrutura global (como a PCA) quanto a estrutura local (como o t-SNE).

O PHATE inicialmente codifica as informações locais (pontos próximos uns aos outros devem permanecer próximos). Ele utiliza “difusão” para descobrir dados globais e então reduz a dimensionalidade:

```
>>> import phate
>>> p = phate.PHATE(random_state=42)
>>> X_phate = p.fit_transform(X)
>>> X_phate.shape
```

Parâmetros da instância:

n_components=2

Número de dimensões.

knn=5

Número de vizinhos para o kernel. Aumente se o embedding estiver desconectado ou o conjunto de dados tiver mais de 100 mil amostras.

decay=40

Taxa de queda do kernel. Reduzir esse valor aumenta a conectividade do grafo.

n_landmark=2000

Landmarks (marcas) a serem usadas.

t='auto'

Potência de difusão. Uma suavização é feita nos dados. Aumente esse valor se faltar estrutura no embedding. Diminua se a estrutura for rígida e compacta.

gamma=1

Potencial logarítmico (entre -1 e 1). Se os embeddings estiverem concentrados em torno de um único ponto, experimente definir esse valor com 0.

n_pca=100

Número de componentes principais para cálculo de vizinhança.

knn_dist='euclidean'

Métrica de KNN.

mds_dist='euclidean'

Métrica de MDS (Multidimensional Scaling, ou Escala Multidimensional).

mds='metric'

Algoritmo de MDS para redução de dimensões.

n_jobs=1

Número de CPUs a serem usadas.

random_state=None

Semente (seed) aleatória.

verbose=1

Verbosidade.

Atributos (observe que estes não são seguidos de _):

X

Dados de entrada.

embedding

Espaço de embedding.

diff_op

Operador de difusão.

graph

Grafo de KNN construído a partir dos dados de entrada.

Eis um exemplo de uso do PHATE (veja a Figura 17.18):

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> phate.plot.scatter2d(p, c=y, ax=ax, alpha=0.5)
>>> fig.savefig("images/mlpr_1718.png", dpi=300)
```

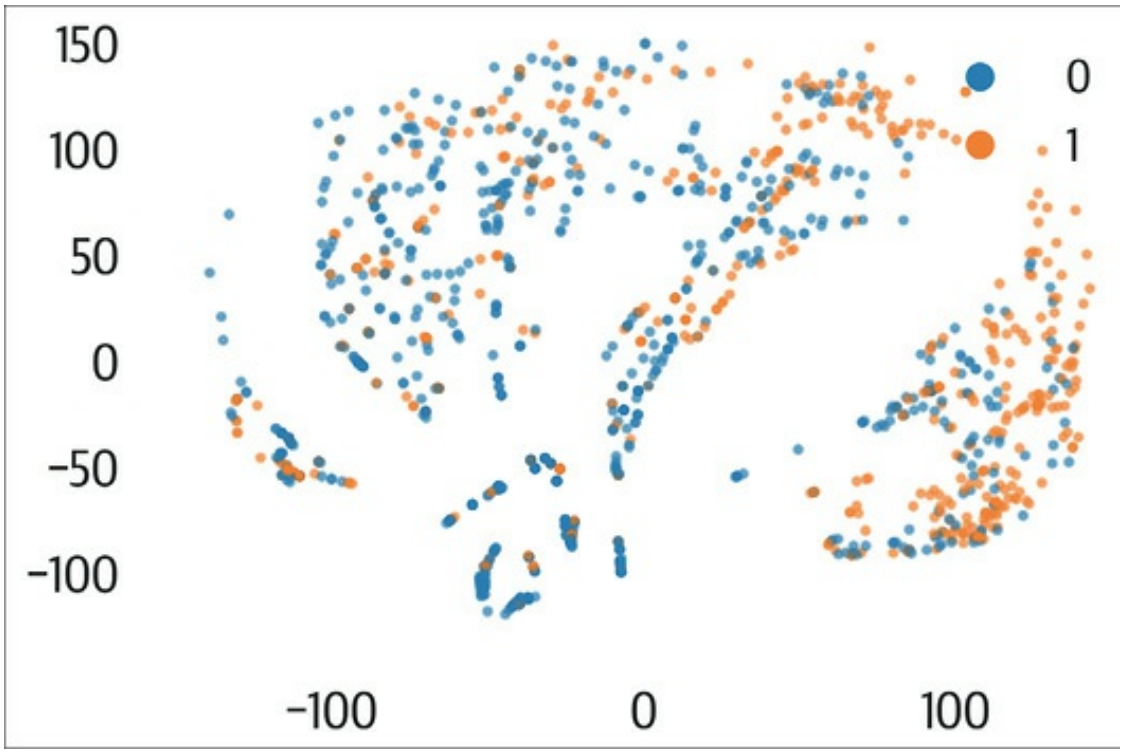


Figura 17.18 – Resultado do PHATE.

Conforme observamos antes nos parâmetros da instância, há alguns parâmetros que podem ser ajustados para alterar o comportamento do modelo. A seguir, apresentamos um exemplo de como ajustar o parâmetro knn (veja a Figura 17.19). Observe que, se usarmos o método .set_params, o cálculo será agilizado, pois o grafo previamente processado e o operador de difusão serão utilizados:

```
>>> fig, axes = plt.subplots(2, 2, figsize=(6, 4))
>>> axes = axes.reshape(4)
>>> p = phate.PHATE(random_state=42, n_jobs=-1)
```

```
>>> for i, n in enumerate((2, 5, 20, 100)):
... ax = axes[i]
... p.set_params(knn=n)
... X_phate = p.fit_transform(X)
... pd.DataFrame(X_phate).plot(
... kind="scatter",
... x=0,
... y=1,
... ax=ax,
... c=y,
... cmap="Spectral",
... alpha=0.5,
... )
... ax.set_title(f"knn={n}")
... plt.tight_layout()
... fig.savefig("images/mlpr_1719.png", dpi=300)
```

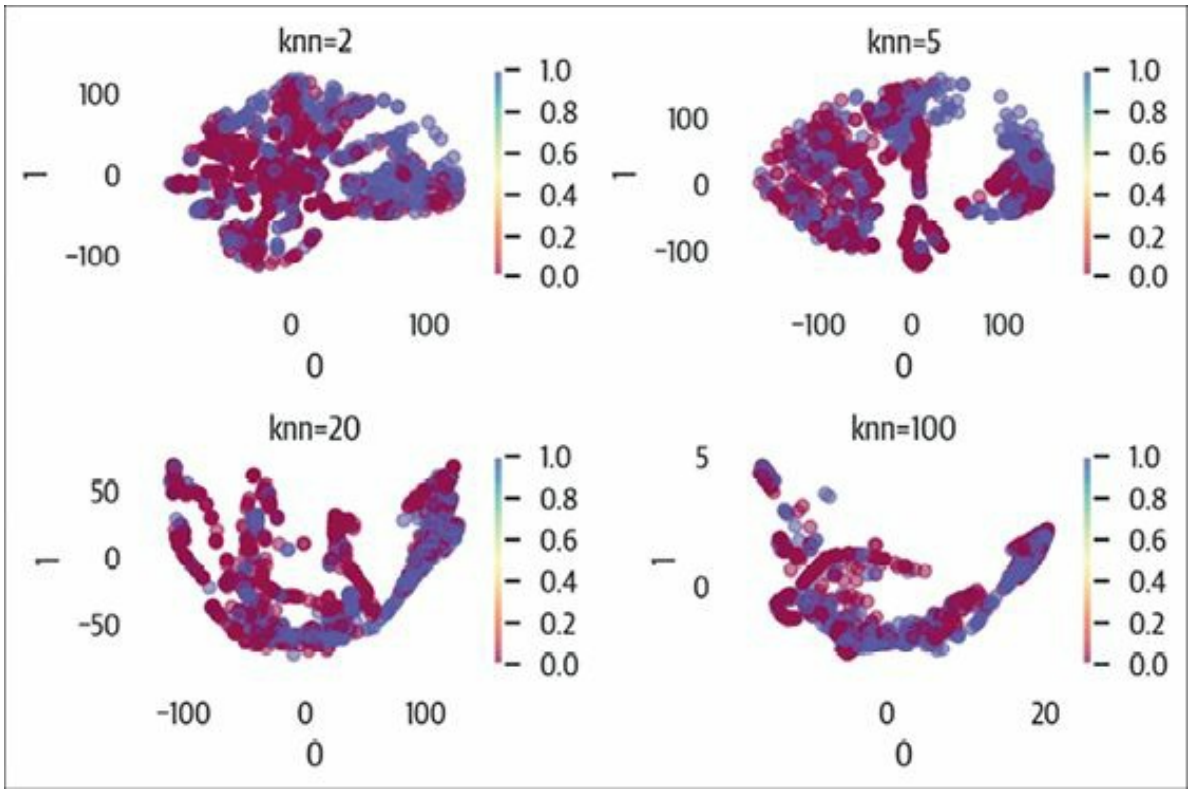


Figura 17.19 – Modificando o parâmetro knn *no PHATE.*

CAPÍTULO 18

Clustering

O clustering (agrupamento) é uma técnica de machine learning não supervisionada, usada para dividir um grupo em conjuntos. É não supervisionada porque não fornecemos nenhum rótulo (label) ao modelo; ele simplesmente inspeciona os atributos e determina quais amostras são semelhantes e pertencem a um cluster. Neste capítulo, veremos os métodos K-means (K-médias) e o clustering hierárquico. Também exploraremos o conjunto de dados do Titanic novamente usando várias técnicas.

K-Means

O algoritmo k-means (k-médias) exige que o usuário selecione o número de clusters, isto é, “k”. Então, ele escolhe aleatoriamente k centroides e atribui cada amostra a um cluster com base na métrica de distância a partir do centroide. Após a atribuição, os centroides são recalculados com base no centro de todas as amostras atribuídas a um rótulo. Em seguida, a atribuição das amostras aos clusters se repete com base nos novos centroides. Após algumas iterações, deve haver uma convergência.

Como o clustering utiliza métricas de distância para determinar quais amostras são semelhantes, o comportamento poderá mudar dependendo da escala dos dados. Podemos padronizar os dados e colocar todos os atributos na mesma escala. Algumas pessoas sugeriram que um especialista no assunto poderia aconselhar contra uma padronização caso a escala sinalize que alguns atributos tenham mais importância. Em nosso exemplo, padronizaremos os dados.

Criaremos clusters para os passageiros do Titanic em nosso caso. Começaremos com dois clusters para ver se o clustering é capaz de separar os sobreviventes (não faremos o vazamento dos dados de sobrevivência no clustering, e usaremos apenas `X`, e não `y`).

Algoritmos não supervisionados têm um método .fit e um método .predict. Passamos apenas X para .fit:

```
>>> from sklearn.cluster import KMeans
>>> X_std = preprocessing.StandardScaler().fit_transform(
... X
... )
>>> km = KMeans(2, random_state=42)
>>> km.fit(X_std)
KMeans(algorithm='auto', copy_x=True,
  init='k-means', max_iter=300,
  n_clusters=2, n_init=10, n_jobs=1,
  precompute_distances='auto',
  random_state=42, tol=0.0001, verbose=0)
```

Depois que o modelo é treinado, podemos chamar o método .predict para atribuir novas amostras a um cluster:

```
>>> X_km = km.predict(X)
>>> X_km
array([1, 1, 1, ..., 1, 1, 1], dtype=int32)
```

Parâmetros da instância:

n_clusters=8

Número de clusters a serem criados.

init='kmeans++'

Método de inicialização.

n_init=10

Número de vezes que o algoritmo deve executar com diferentes centroides. A melhor pontuação vencerá.

max_iter=300

Número de iterações para uma execução.

tol=0.0001

Tolerância até a convergência.

precompute_distances='auto'

Pré-calcular distâncias (exige mais memória, porém é mais rápido). auto fará o cálculo prévio se n_samples * n_clusters for menor ou igual a 12 milhões.

verbose=0

Verbosidade.

random_state=None
  Semente (seed) aleatória.

copy_x=True
  Copia dados antes de processar.

n_jobs=1
  Número de CPUs a serem usadas.

algorithm='auto'
  Algoritmo K-means. 'full' funciona com dados esparsos, mas 'elkan' é mais eficaz. 'auto' usa 'elkan' com dados densos.

Atributos:

cluster_centers_
  Coordenadas dos centros.

labels_
  Rótulos para as amostras.

inertia_
  Soma dos quadrados das distâncias até o centroide do cluster.

n_iter_
  Número de iterações.

Se você não souber com antecedência quantos clusters serão necessários, poderá executar o algoritmo com diversos tamanhos e avaliar várias métricas. Pode ser complicado.

Você pode implementar o próprio gráfico elbow (gráfico de cotovelo) usando o cálculo de .inertia_. Procure o ponto em que a curva dobra, pois essa será, possivelmente, uma boa opção para o número de clusters. Em nosso caso, a curva é suave, mas, depois de oito, não parece haver muita melhoria (veja a Figura 18.1).

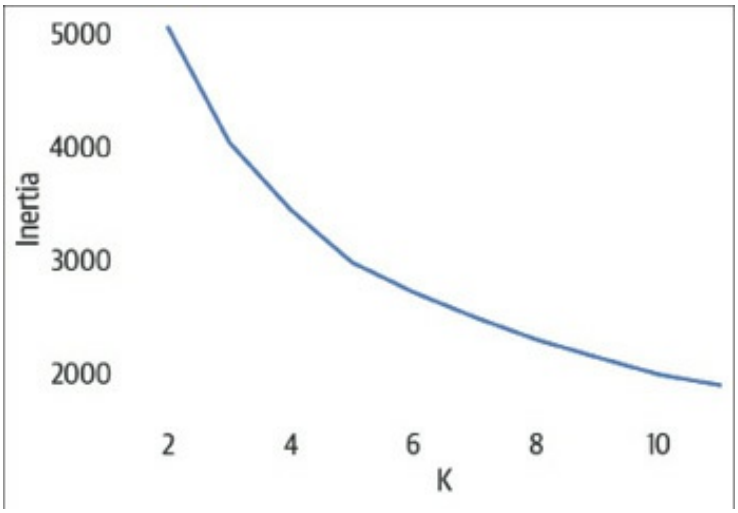


Figura 18.1 – Gráfico de cotovelo com uma aparência bem suave.

Para gráficos sem um cotovelo, temos algumas opções. Podemos usar outras métricas, algumas das quais serão apresentadas a seguir. Também podemos inspecionar visualmente o clustering e ver se os clusters são visíveis. Podemos adicionar atributos aos dados e ver se isso ajuda no clustering.

Eis o código para um gráfico de cotovelo:

```
>>> inertias = []
>>> sizes = range(2, 12)
>>> for k in sizes:
... k2 = KMeans(random_state=42, n_clusters=k)
... k2.fit(X)
... inertias.append(k2.inertia_)
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> pd.Series(inertias, index=sizes).plot(ax=ax)
>>> ax.set_xlabel("K")
>>> ax.set_ylabel("Inertia")
>>> fig.savefig("images/mlpr_1801.png", dpi=300)
```

O scikit-learn tem outas métricas de clustering quando os verdadeiros rótulos não são conhecidos. Podemos calcular esses valores e colocá-los em um gráfico também. O *coeficiente de silhueta* (silhouette coefficient) é um valor entre -1 e 1. Quanto maior o valor, melhor será. O valor 1 indica clusters mais claros, enquanto 0 implica clusters que se sobrepõem. A partir dessa medida, dois clusters nos dão a melhor pontuação.

O *Índice de Calinski-Harabasz* é a razão entre a dispersão interclusters e a dispersão intracluster. Uma pontuação maior será melhor. Dois clusters dão a melhor pontuação para essa métrica.

O *Índice de Davis-Bouldin* é a semelhança média entre cada cluster e o cluster mais próximo. As pontuações variam de 0 ou mais. O valor 0 indica um clustering melhor.

A seguir, colocaremos em um gráfico as informações de inércia, o coeficiente de silhueta, o Índice de Calinski-Harabasz e o Índice de Davies-Bouldin para diversos tamanhos de clusters, a fim de ver se há um tamanho evidente de clusters para os dados (veja a Figura 18.2). Parece que a maior parte dessas métricas concorda que deve haver dois clusters:

```
>>> from sklearn import metrics
>>> inertias = []
>>> sils = []
>>> chs = []
>>> dbs = []
>>> sizes = range(2, 12)
>>> for k in sizes:
... k2 = KMeans(random_state=42, n_clusters=k)
... k2.fit(X_std)
... inertias.append(k2.inertia_)
... sils.append(
... metrics.silhouette_score(X, k2.labels_)
... )
... chs.append(
... metrics.calinski_harabasz_score(
... X, k2.labels_
... )
... )
... dbs.append(
... metrics.davies_bouldin_score(
... X, k2.labels_
... )
... )
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> (
... pd.DataFrame(
... {
... "inertia": inertias,
... "silhouette": sils,
... "calinski": chs,
... "davis": dbs,
... "k": sizes,
... }
... )
```

```
... .set_index("k")
... .plot(ax=ax, subplots=True, layout=(2, 2))
... )
>>> fig.savefig("images/mlpr_1802.png", dpi=300)
```

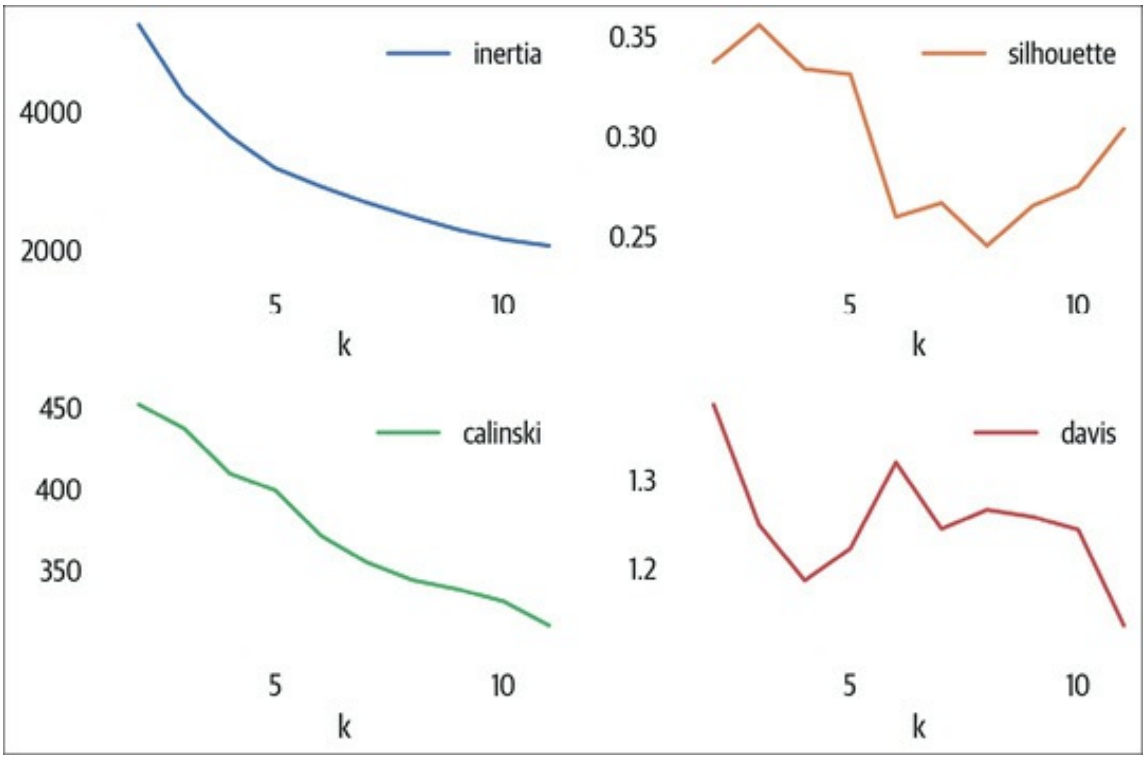


Figura 18.2 – Métricas de clusters. Essas métricas, em sua maioria, concordam que deve haver dois clusters.

Outra técnica para determinar clusters é visualizar as pontuações de silhueta para cada cluster. O Yellowbrick tem um visualizador para isso (veja a Figura 18.3).

A linha vermelha pontilhada vertical nesse gráfico é a pontuação média. Um modo de interpretar isso é garantir que cada cluster se destaque acima da média, e as pontuações do cluster pareçam razoáveis. Não se esqueça de usar os mesmos limites x (ax.set_xlim). A partir desses gráficos, eu optaria por dois clusters:

```
>>> from yellowbrick.cluster.silhouette import (
... SilhouetteVisualizer,
... )
>>> fig, axes = plt.subplots(2, 2, figsize=(12, 8))
>>> axes = axes.reshape(4)
>>> for i, k in enumerate(range(2, 6)):
... ax = axes[i]
```

```
... sil = SilhouetteVisualizer(
... KMeans(n_clusters=k, random_state=42),
... ax=ax,
... )
... sil.fit(X_std)
... sil.finalize()
... ax.set_xlim(-0.2, 0.8)
>>> plt.tight_layout()
>>> fig.savefig("images/mlpr_1803.png", dpi=300)
```

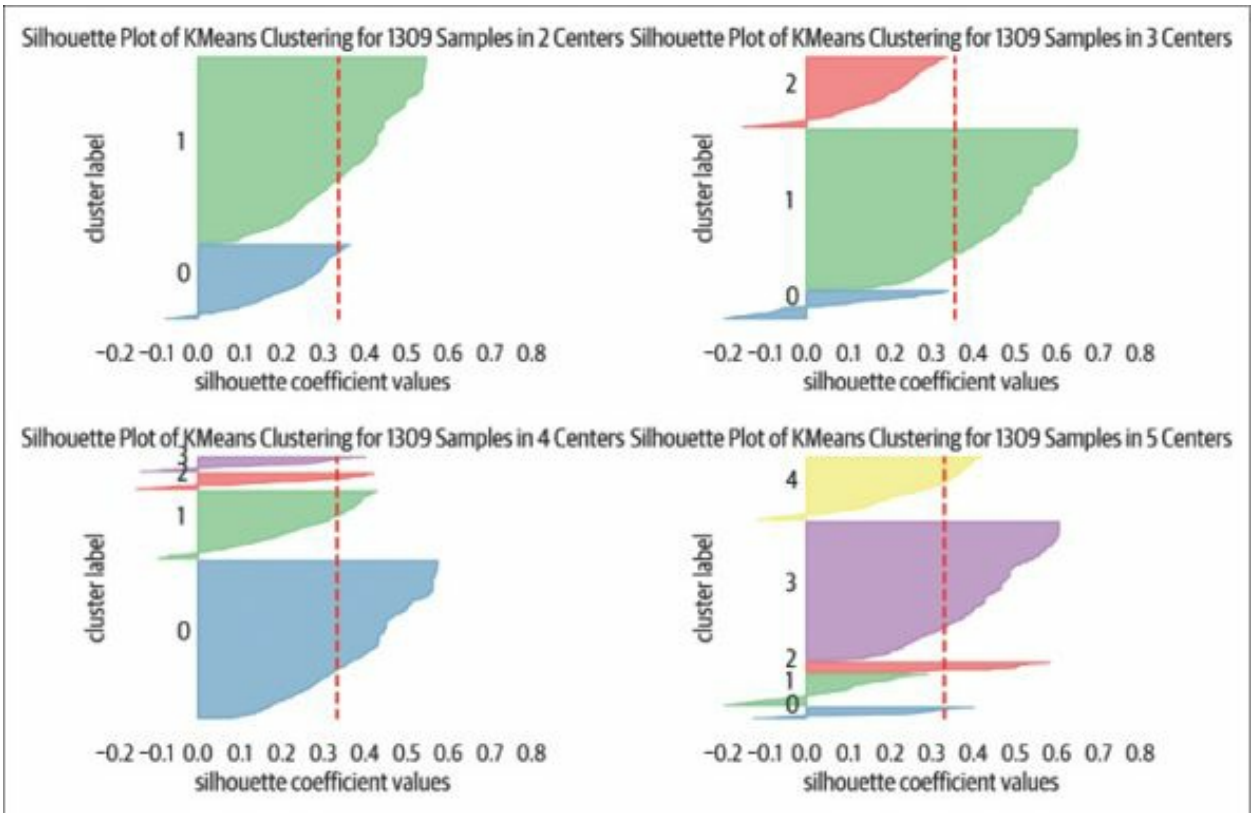


Figura 18.3 – Visualizador de silhuetas do Yellowbrick.

Clustering (hierárquico) aglomerativo

Outra metodologia é o clustering aglomerativo (agglomerative clustering). Comece com cada amostra em seu próprio cluster. Em seguida, combine os clusters “mais próximos”. Repita até terminar, enquanto mantém o controle dos tamanhos mais próximos.

Quando terminar, você terá um *dendrograma*, isto é, uma árvore que controla quando os clusters foram criados e qual é a métrica das distâncias. Podemos usar a biblioteca scipy para visualizar o dendrograma.

A scipy pode ser usada para criar um dendrograma (veja a Figura 18.4). Como podemos ver, se você tiver muitas amostras, será difícil ler os nós do tipo folha:

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
... hierarchy.linkage(X_std, method="ward")
... )
>>> fig.savefig("images/mlpr_1804.png", dpi=300)
```

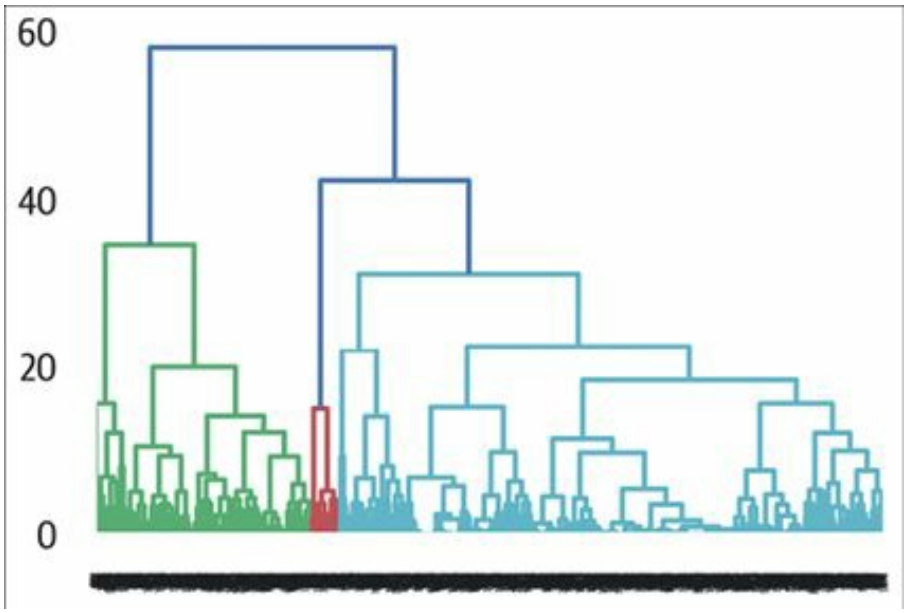


Figura 18.4 – Dendrograma de clustering hierárquico gerado com o scipy.

Depois que tiver o dendrograma, você terá todos os clusters (de um até a quantidade de amostras). As alturas representam o nível de semelhança dos clusters quando são unidos. Para descobrir quantos clusters há nos dados, você poderia “passar” uma linha horizontal no ponto em que ela cruzaria as linhas mais altas.

Nesse caso, ao traçar essa linha, você teria três clusters.

O gráfico anterior apresenta um pouco de ruído, pois contém todas as amostras. Você também pode usar o parâmetro truncate_mode para combinar as folhas em um único nó (veja a Figura 18.5):

```
>>> from scipy.cluster import hierarchy
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> dend = hierarchy.dendrogram(
... hierarchy.linkage(X_std, method="ward"),
... truncate_mode="lastp",
```

```
... p=20,
... show_contracted=True,
... )
>>> fig.savefig("images/mlpr_1805.png", dpi=300)
```

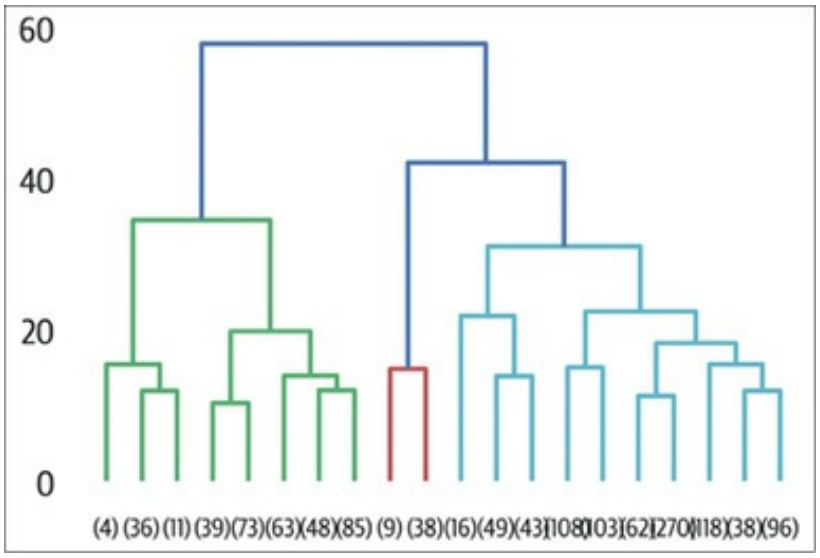


Figura 18.5 – Dendrograma de clustering hierárquico truncado. Se cruzarmos as maiores linhas verticais, teremos três clusters.

Depois que soubermos quantos clusters serão necessários, poderemos usar o scikit-learn para criar um modelo:

```
>>> from sklearn.cluster import (
... AgglomerativeClustering,
... )
>>> ag = AgglomerativeClustering(
... n_clusters=4,
... affinity="euclidean",
... linkage="ward",
... )
>>> ag.fit(X)
```

NOTA

O pacote fastcluster (*https://oreil.ly/OuNuo*) disponibiliza um pacote de clustering aglomerativo otimizado caso a implementação do scikit-learn seja muito lenta.

Entendendo os clusters

Ao usar o K-means no conjunto de dados do Titanic, criaremos dois clusters. Podemos usar a funcionalidade de agrupamento do pandas para analisar as

diferenças entre os clusters. O código a seguir analisa a média e a variância para cada atributo. Parece que o valor médio de pclass varia bastante.

Estou inserindo de volta os dados de sobrevivência para ver se o clustering estava relacionado com eles:

```
>>> km = KMeans(n_clusters=2)
>>> km.fit(X_std)
>>> labels = km.predict(X_std)
>>> (
... X.assign(cluster=labels, survived=y)
... .groupby("cluster")
... .agg(["mean", "var"])
... .T
... )
cluster 0 1
pclass mean 0.526538 -1.423831
        var 0.266089 0.136175
age mean -0.280471 0.921668
        var 0.653027 1.145303
sibsp mean -0.010464 -0.107849
        var 1.163848 0.303881
parch mean 0.387540 0.378453
        var 0.829570 0.540587
fare mean -0.349335 0.886400
        var 0.056321 2.225399
sex_male mean 0.678986 0.552486
        var 0.218194 0.247930
embarked_Q mean 0.123548 0.016575
        var 0.108398 0.016345
embarked_S mean 0.741288 0.585635
        var 0.191983 0.243339
survived mean 0.596685 0.299894
        var 0.241319 0.210180
```

NOTA

No Jupyter, você pode associar o código a seguir a um DataFrame, e o valor maior e o menor de cada linha serão marcados. Isso é conveniente para ver quais valores se destacam nos dados de cluster anteriores:

```
.style.background_gradient(cmap='RdBu', axis=1)
```

Na Figura 18.6, apresentamos um gráfico de barras das médias de cada cluster:

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
```

```
... (
... X.assign(cluster=labels, survived=y)
... .groupby("cluster")
... .mean()
... .T.plot.bar(ax=ax)
... )
>>> fig.savefig(
... "images/mlpr_1806.png",
... dpi=300,
... bbox_inches="tight",
... )
```

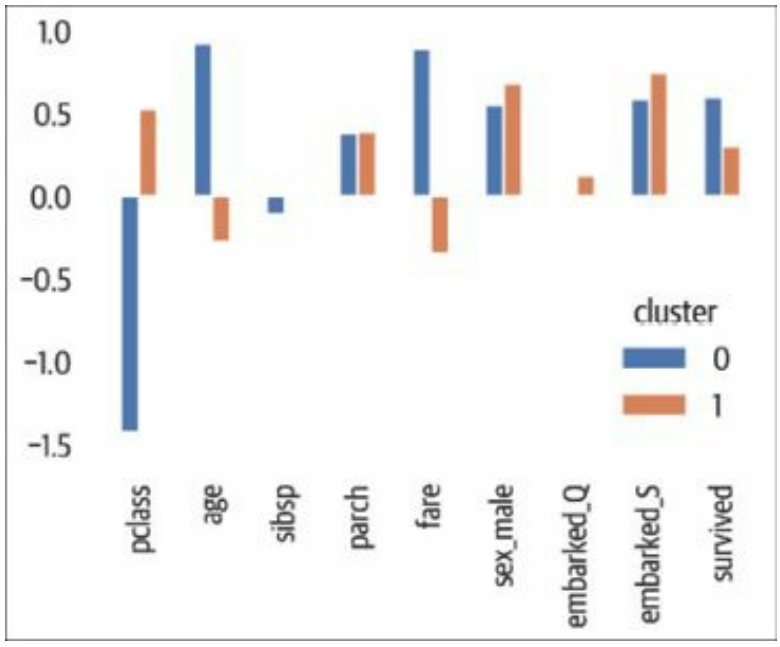


Figura 18.6 – Valores médios de cada cluster.

Também gosto de colocar os componentes da PCA em um gráfico, porém coloridos de acordo com o rótulo do cluster (veja a Figura 18.7). Nesse caso, usamos o Seaborn para isso. Também é interessante modificar os valores de `hue` para explorar os atributos que são distintos para os clusters.

```
>>> fig, ax = plt.subplots(figsize=(6, 4))
>>> sns.scatterplot(
... "PC1",
... "PC2",
... data=X.assign(
... PC1=X_pca[:, 0],
... PC2=X_pca[:, 1],
... cluster=labels,
... ),
... hue="cluster",
```

```
... alpha=0.5,
... ax=ax,
... )
>>> fig.savefig(
... "images/mlpr_1807.png",
... dpi=300,
... bbox_inches="tight",
... )
```

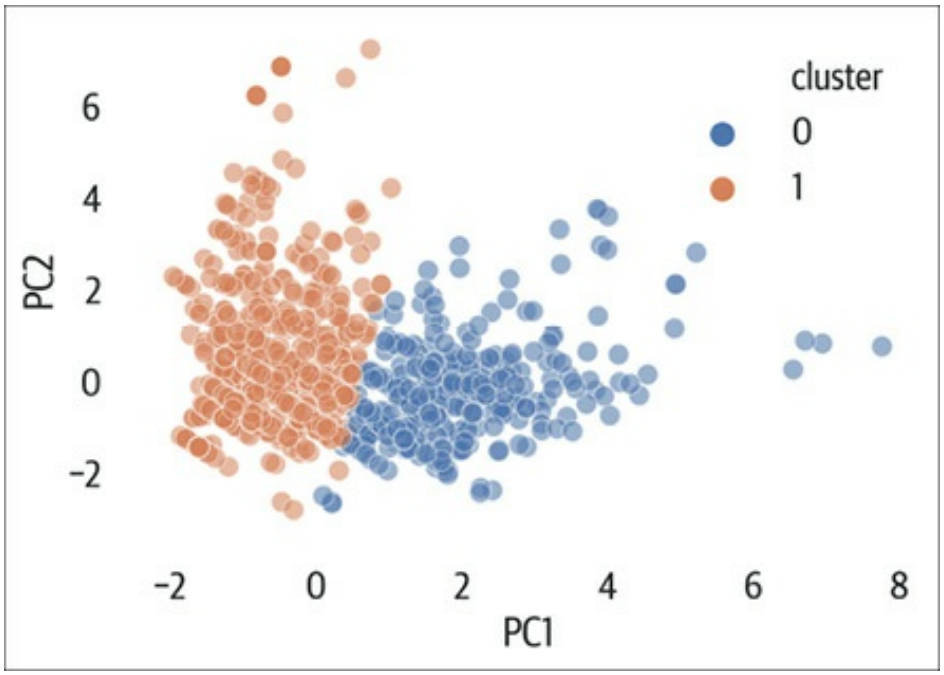


Figura 18.7 – Gráfico de PCA dos clusters.

Se quisermos analisar um único atributo, podemos usar o método .describe do pandas:

```
>>> (
... X.assign(cluster=label)
... .groupby("cluster")
... .age.describe()
... .T
... )
cluster 0 1
count 362.000000 947.000000
mean 0.921668 -0.280471
std 1.070188 0.808101
min -2.160126 -2.218578
25% 0.184415 -0.672870
50% 0.867467 -0.283195
75% 1.665179 0.106480
```

```
max 4.003228 3.535618
```

Também podemos criar um modelo substituto (surrogate model) para explicar os clusters. Nesse caso, usaremos uma árvore de decisão para explicá-los. Ele também mostra que pclass (que apresentava uma grande diferença na média) é muito importante:

```
>>> dt = tree.DecisionTreeClassifier()
>>> dt.fit(X, labels)
>>> for col, val in sorted(
... zip(X.columns, dt.feature_importances_),
... key=lambda col_val: col_val[1],
... reverse=True,
... ):
... print(f"{col:10}{val:10.3f}")
pclass 0.902
age 0.074
sex_male 0.016
embarked_S 0.003
fare 0.003
parch 0.003
sibsp 0.000
embarked_Q 0.000
```

Podemos também visualizar as decisões na Figura 18.8. Ela mostra que pclass é o primeiro atributo que o modelo substituto observa para tomar uma decisão:

```
>>> dot_data = StringIO()
>>> tree.export_graphviz(
... dt,
... out_file=dot_data,
... feature_names=X.columns,
... class_names=["0", "1"],
... max_depth=2,
... filled=True,
... )
>>> g = pydotplus.graph_from_dot_data(
... dot_data.getvalue()
... )
>>> g.write_png("images/mlpr_1808.png")
```

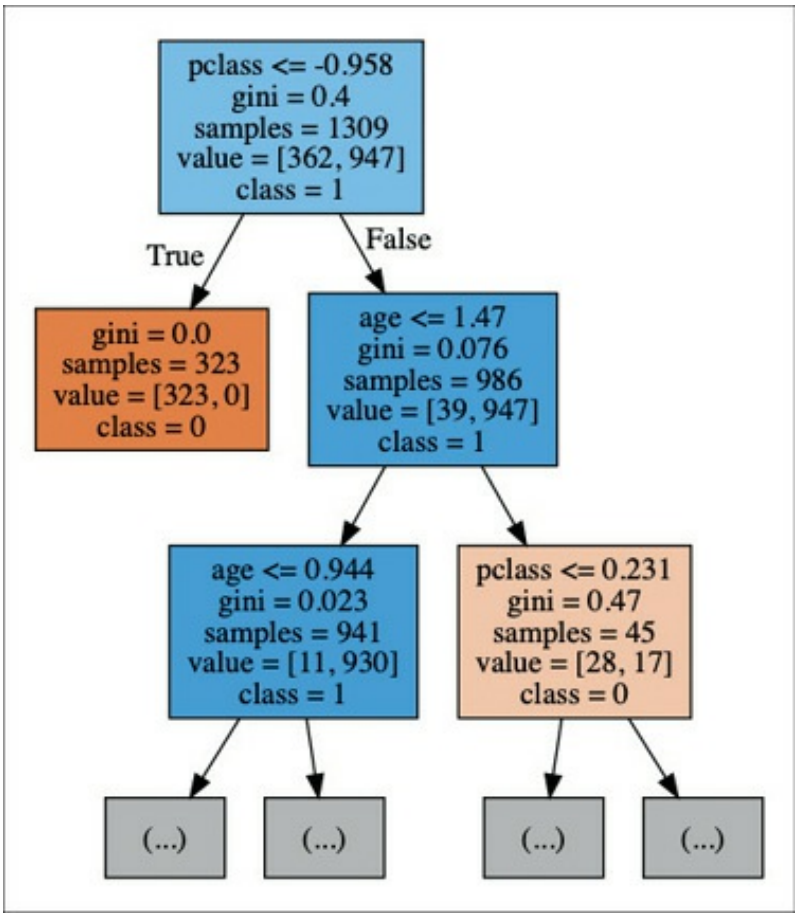


Figura 18.8 – Árvore de decisão que explica o clustering.

CAPÍTULO 19

Pipelines

O scikit-learn utiliza a noção de pipeline. Ao usar a classe Pipeline, você poderá não apenas encadear transformadores e modelos, mas também tratar o processo todo como um único modelo do scikit-learn. É possível até mesmo usar uma lógica personalizada.

Pipeline de classificação

Eis um exemplo que usa a função tweak_titanic em um pipeline:

```
>>> from sklearn.base import (
... BaseEstimator,
... TransformerMixin,
... )
>>> from sklearn.pipeline import Pipeline

>>> def tweak_titanic(df):
... df = df.drop(
... columns=[
... "name",
... "ticket",
... "home.dest",
... "boat",
... "body",
... "cabin",
... ]
... ).pipe(pd.get_dummies, drop_first=True)
... return df

>>> class TitanicTransformer(
... BaseEstimator, TransformerMixin
... ):
... def transform(self, X):
... # supõe que X é o resultado
... # da leitura de um arquivo Excel
... X = tweak_titanic(X)
```

```
... X = X.drop(column="survived")
... return X
...
... def fit(self, X, y):
... return self

>>> pipe = Pipeline(
... [
... ("titan", TitanicTransformer()),
... ("impute", impute.IterativeImputer()),
... (
... "std",
... preprocessing.StandardScaler(),
... ),
... ("rf", RandomForestClassifier()),
... ]
... )
```

Com um pipeline em mãos, podemos chamar .fit e .score:

```
>>> from sklearn.model_selection import (
... train_test_split,
... )
>>> X_train2, X_test2, y_train2, y_test2 = train_test_split(
... orig_df,
... orig_df.survived,
... test_size=0.3,
... random_state=42,
... )

>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Os pipelines podem ser usados na busca em grade (grid search). Nosso param_grid deve ter os parâmetros prefixados pelo nome da etapa do pipeline, seguidos de dois underscores. No exemplo a seguir, adicionamos alguns parâmetros para a etapa de floresta aleatória (random forest):

```
>>> params = {
... "rf__max_features": [0.4, "auto"],
... "rf__n_estimators": [15, 200],
... }

>>> grid = model_selection.GridSearchCV(
... pipe, cv=3, param_grid=params
... )
```

```
>>> grid.fit(orig_df, orig_df.survived)
```

Agora podemos extrair os melhores parâmetros e treinar o modelo final. (Nesse caso, a floresta aleatória não melhora depois da busca em grade.)

```
>>> grid.best_params_
{'rf__max_features': 0.4, 'rf__n_estimators': 15}
>>> pipe.set_params(**grid.best_params_)
>>> pipe.fit(X_train2, y_train2)
>>> pipe.score(X_test2, y_test2)
0.7913486005089059
```

Podemos usar o pipeline no lugar em que usamos modelos do scikit-learn:

```
>>> metrics.roc_auc_score(
... y_test2, pipe.predict(X_test2)
... )
0.7813688715131023
```

Pipeline de regressão

Eis um exemplo de um pipeline que faz uma regressão linear no conjunto de dados de Boston:

```
>>> from sklearn.pipeline import Pipeline

>>> reg_pipe = Pipeline(
... [
... (
... "std",
... preprocessing.StandardScaler(),
... ),
... ("lr", LinearRegression()),
... ]
... )
>>> reg_pipe.fit(bos_X_train, bos_y_train)
>>> reg_pipe.score(bos_X_test, bos_y_test)
0.7112260057484934
```

Se quisermos extrair partes do pipeline a fim de analisar suas propriedades, podemos fazer isso com o atributo .named_steps:

```
>>> reg_pipe.named_steps["lr"].intercept_
23.01581920903956
>>> reg_pipe.named_steps["lr"].coef_
array([-1.10834602, 0.80843998, 0.34313466,
       0.81386426, -1.79804295, 2.913858 ,
       -0.29893918, -2.94251148, 2.09419303,
       -1.44706731, -2.05232232, 1.02375187,
```

```
       -3.88579002])_
```

O pipeline pode ser usado nos cálculos de métricas também:

```
>>> from sklearn import metrics
>>> metrics.mean_squared_error(
... bos_y_test, reg_pipe.predict(bos_X_test)
... )
21.517444231177205
```

Pipeline de PCA

Os pipelines do scikit-learn também podem ser usados na PCA.

Nesse caso, padronizamos o conjunto de dados do Titanic e executamos aí a PCA:

```
>>> pca_pipe = Pipeline(
... [
... (
... "std",
... preprocessing.StandardScaler(),
... ),
... ("pca", PCA()),
... ]
... )
>>> X_pca = pca_pipe.fit_transform(X)
```

Ao usar o atributo .named_steps, podemos obter as propriedades da parte do pipeline referente à PCA:

```
>>> pca_pipe.named_steps[
... "pca"
... ].explained_variance_ratio_
array([0.23917891, 0.21623078, 0.19265028,
       0.10460882, 0.08170342, 0.07229959,
       0.05133752, 0.04199068])
>>> pca_pipe.named_steps["pca"].components_[0]
array([-0.63368693, 0.39682566, 0.00614498,
        0.11488415, 0.58075352, -0.19046812,
        -0.21190808, -0.09631388])
```

Sobre o autor

Matt Harrison administra a MetaSnake, uma empresa de treinamento e consultoria para Python e ciência de dados. Desde 2000, o autor usa Python em diversos domínios: ciência de dados, inteligência de negócios, armazenagem, testes e automação, gerenciamento de pilhas open source, na área financeira e em pesquisa de informações

Colofão

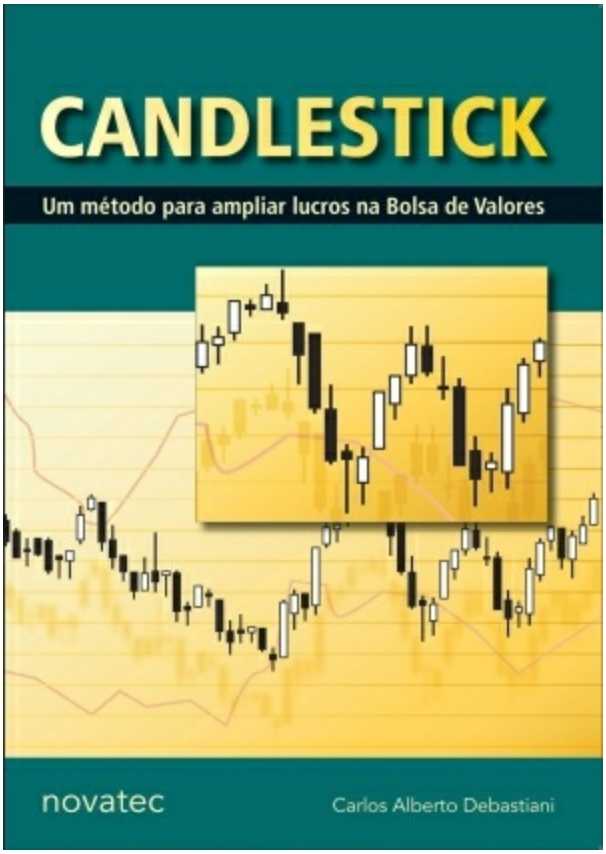
O animal na capa de *Machine learning – Guia de Referência Rápida* é um tritão-de-crista do norte (*Triturus cristatus*), um anfíbio encontrado próximo a regiões de águas paradas no leste britânico, estendendo-se até o continente europeu e a Rússia ocidental.

Esse tritão tem costas castanho-acinzentadas com pontos escuros e a parte inferior amarelo-alaranjada com manchas brancas. Os machos desenvolvem cristas grandes e serrilhadas durante a época do acasalamento, enquanto as fêmeas sempre têm uma listra alaranjada na cauda.

Apesar de não hibernarem na lama nem sob rochas durante os meses de inverno, o tritão-de-crista do norte caça outros tritões, girinos, rãs jovens, minhocas, larvas de insetos e caramujos em ambientes aquáticos, e insetos, larvas e outros invertebrados em ambientes terrestres. Poder viver até 27 anos e alcançar aproximadamente 0,2 metro de comprimento.

Embora o status atual de preservação do tritão-de-crista seja considerado como de pouca preocupação, muitos dos animais das capas dos livros da O’reilly estão ameaçados e todos são importantes para o planeta.

A ilustração da capa é de Karen Montgomery, com base em uma gravura em preto e branco do livro *Meyers Kleines Lexicon*.


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

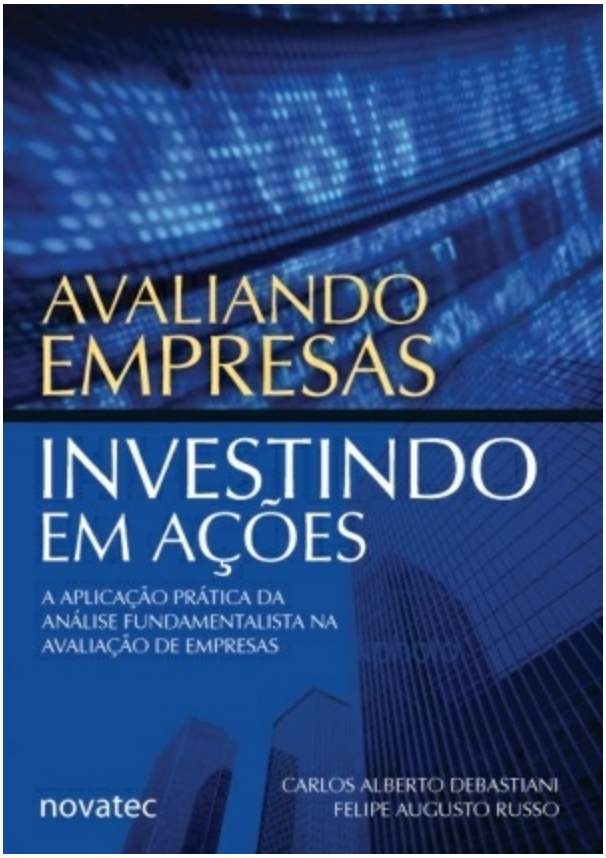
Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa

de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.


Fundos de
Investimento
Imobiliário
ASPECTOS GERAIS E
PRINCÍPIOS DE ANÁLISE
novatec
Roni Antônio Mendes

Fundos de Investimento Imobiliário

Mendes, Roni Antônio

9788575226766

256 páginas

Você sabia que o investimento em imóveis é um dos preferidos dos brasileiros? Você também gostaria de investir em imóveis, mas tem pouco dinheiro? Saiba que é possível, mesmo com poucos recursos, investir no mercado de imóveis por meio dos Fundos de Investimento Imobiliário (FIIs). Investir em FIIs representa uma excelente alternativa para aumentar o patrimônio no longo prazo. Além disso, eles são ótimos ativos geradores de renda que pode ser usada para complementar a aposentadoria. Infelizmente, no Brasil, os FIIs são pouco conhecidos. Pouco mais de 100 mil pessoas investem nesses ativos. Lendo este livro, você aprenderá os aspectos gerais dos FIIs: o que são; as vantagens que oferecem; os riscos que possuem; os diversos tipos de FIIs que existem no mercado e como proceder para investir bem e com segurança. Você também aprenderá os princípios básicos para avaliá-los, inclusive empregando um método poderoso, utilizado por investidores do mundo inteiro: o método do Fluxo de Caixa Descontado (FCD).

Alguns exemplos reais de FIIs foram estudados neste livro e os resultados são apresentados de maneira clara e didática, para que você aprenda a conduzir os próprios estudos e tirar as próprias conclusões. Também são apresentados conceitos gerais de como montar e gerenciar uma carteira de investimentos. Aprenda a investir em FIIs. Leia este livro.


O'REILLY
Microsserviços prontos para a produção
CONSTRUINDO SISTEMAS PADRONIZADOS EM UMA ORGANIZAÇÃO DE ENGENHARIA DE SOFTWARE
novatec
Susan J. Fowler

Microsserviços prontos para a produção

Fowler, Susan J.

9788575227473

224 páginas

Um dos maiores desafios para as empresas que adotaram a arquitetura de microsserviços é a falta de padronização de arquitetura – operacional e organizacional. Depois de dividir uma aplicação monolítica ou construir um ecossistema de microsserviços a partir do zero, muitos engenheiros se perguntam o que vem a seguir. Neste livro prático, a autora Susan Fowler apresenta com profundidade um conjunto de padrões de microsserviço, aproveitando sua experiência de padronização de mais de mil microsserviços do Uber. Você aprenderá a projetar microsserviços que são estáveis, confiáveis, escaláveis, tolerantes a falhas, de alto desempenho, monitorados, documentados e preparados para qualquer catástrofe. Explore os padrões de disponibilidade de produção, incluindo: Estabilidade e confiabilidade – desenvolva, implante, introduza e descontinue microsserviços; proteja-se contra falhas de dependência.

Escalabilidade e desempenho – conheça os componentes essenciais para alcançar mais eficiência do microsserviço. Tolerância a falhas e prontidão para catástrofes – garanta a disponibilidade forçando ativamente os microsserviços a falhar em tempo real. Monitoramento – aprenda como monitorar, gravar logs e exibir as principais métricas; estabeleça procedimentos de alerta e de prontidão. Documentação e compreensão – atenue os efeitos negativos das contrapartidas que acompanham a adoção dos microsserviços, incluindo a dispersão organizacional e a defasagem técnica.