

Capítulo 2

Projeto de Aprendizado de Máquina de Ponta a Ponta

Neste capítulo você verá um exemplo de projeto de ponta a ponta na pele de um cientista de dados recentemente contratado por uma empresa do mercado imobiliário.¹ Você seguirá estes principais passos:

1. Olhar para o quadro geral;
2. Obter os dados;
3. Descobrir e visualizar os dados para obter informações;
4. Preparar os dados para os algoritmos do Aprendizado de Máquina;
5. Selecionar e treinar um modelo;
6. Ajustar o seu modelo;
7. Apresentar sua solução;
8. Lançar, monitorar e manter seu sistema.

Trabalhando com Dados Reais

É melhor experimentar com dados do mundo real e não apenas conjuntos de dados artificiais ao estudar Aprendizado de Máquina. Felizmente, existem milhares de conjuntos de dados disponíveis a sua escolha, variando em todos os tipos de domínios. Você pode procurar em muitos lugares, tais quais:

- Reppositórios Populares de *open data*:
 - UC Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>)

¹ Este exemplo de projeto é completamente fictício. O objetivo é apenas ilustrar os níveis principais de um projeto de Aprendizado de Máquina, não aprender sobre o mercado imobiliário.

- Conjunto de dados no Kaggle (<https://www.kaggle.com/datasets>)
- Conjunto de Dados no AWS da Amazon (<http://aws.amazon.com/fr/datasets/>)
- Meta portais (eles listam repositórios *open data*):
 - <http://dataportals.org/>
 - <http://opendatamonitor.eu/>
 - <http://quandl.com/>
- Outras páginas que listam muitos repositórios populares de *open data*:
 - Lista de conjuntos de dados de Aprendizado de Máquina do Wikipedia (<https://goo.gl/SJHN2k>)
 - Pergunta no Quora.com (<http://goo.gl/zDR78y>)
 - Conjuntos de dados no Reddit (<https://www.reddit.com/r/datasets>)

Neste capítulo, escolhemos o conjunto de dados do repositório StatLib referente a preços do setor imobiliário na Califórnia² (veja a Figura 2-1). Este conjunto de dados foi baseado no censo de 1990 na Califórnia. Não são dados recentes (naquela época, ainda era possível comprar uma casa agradável perto da Baía de São Francisco), mas possuem muitas qualidades para o aprendizado, então vamos fingir que são dados recentes. Também adicionamos um atributo categórico e removemos algumas características para fins de ensino.

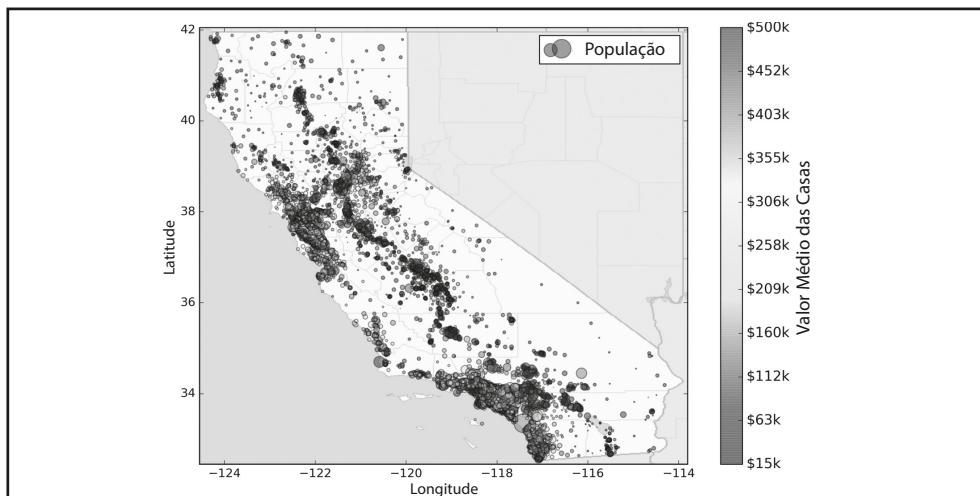


Figura 2-1. Preços de casas na Califórnia

² O conjunto de dados original figurou em R. Kelley Pace Ronald Barry, “Sparse Spatial Autoregressions”, Statistics & Probability Letters 33, nº 3 (1997): 291–297.

Um Olhar no Quadro Geral

Bem-vindo à Machine Learning Housing Corporation! A primeira tarefa que você executará será construir um modelo de preços do setor imobiliário utilizando os dados do censo da Califórnia. Esses dados têm métricas como população, renda média, preço médio do setor imobiliário e assim por diante para cada grupo de bairros. Os grupos de bairros são a menor unidade geográfica para a qual o *US Census Bureau* publica dados de amostra (um grupo de bairros geralmente tem uma população de 600 a 3 mil pessoas). Para abreviar, os chamaremos de “bairros”.

Seu modelo deve aprender com esses dados e ser capaz de prever o preço médio em qualquer bairro, considerando todas as outras métricas.



Como um cientista de dados bem organizado, a primeira coisa a fazer é obter sua lista de verificação do projeto. Você pode começar com a lista do Apêndice B, que deve funcionar razoavelmente bem para a maioria dos projetos de Aprendizado de Máquina, mas assegure-se de adaptá-la às suas necessidades. Neste capítulo, passaremos por muitos itens da lista de verificação, mas também ignoraremos alguns porque eles são autoexplicativos ou porque serão discutidos em capítulos posteriores.

Enquadre o Problema

A primeira pergunta a ser feita ao seu chefe é: qual é exatamente o objetivo comercial? Provavelmente construir um modelo não será o objetivo final. Como a empresa espera usar e se beneficiar deste modelo? Isso é importante porque determinará como você enquadra o problema, quais algoritmos selecionará, que medida de desempenho utilizará para avaliar seu modelo e quanto esforço você deve colocar nos ajustes.

Seu chefe responde que o resultado do seu modelo (uma previsão do preço médio do setor imobiliário no bairro) será enviado para outro sistema de Aprendizado de Máquina (veja a Figura 2-2), juntamente com muitos outros *sinais*.³ Esse sistema de downstream determinará se vale a pena investir em uma determinada área ou não. É fundamental acertar nessa etapa, uma vez que afetará diretamente a receita.

³ Dados fornecidos para um sistema de Aprendizado de Máquina são chamados de sinais em referência à teoria da informação de Shannon: você quer que a proporção sinal/ruído seja alta.

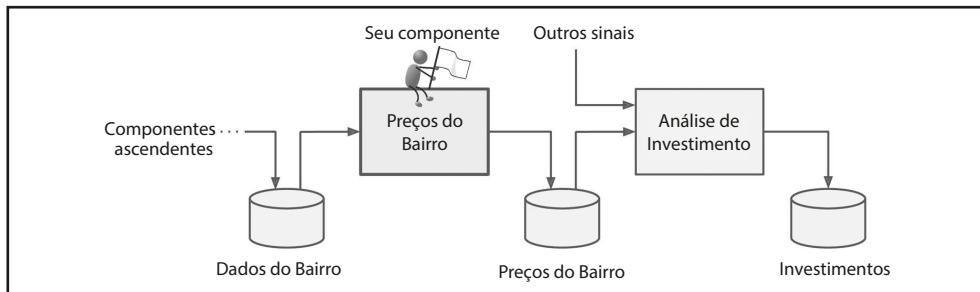


Figura 2-2. Um pipeline de Aprendizado de Máquina para investimentos imobiliários

Pipelines

Uma sequência de *componentes* de processamento de dados é chamada de *pipeline* de dados. Os pipelines são muito comuns em sistemas do Aprendizado de Máquina, uma vez que existem muitos dados para manipular e muitas transformações para aplicar neles.

Os componentes normalmente rodam de forma assíncrona. Cada componente puxa uma grande quantidade de dados, os processa e envia o resultado para outro armazenador dados, e então, algum tempo depois, o próximo componente no pipeline puxa esses dados e envia sua própria saída, e assim por diante. Cada componente é bastante autônomo: a interface entre os componentes é simplesmente o armazenamento de dados. Isso faz com que entender o sistema seja bastante simples (com a ajuda de um grafo do fluxo de dados), e diferentes equipes podem se concentrar em diferentes componentes. Além disso, se um componente se romper, os componentes de downstream geralmente continuam a funcionar normalmente (pelo menos por um tempo), utilizando apenas a última saída do componente rompido. Isso torna a arquitetura bastante robusta.

Por outro lado, se um monitoramento adequado não for implementado, um componente rompido pode passar despercebido por algum tempo. Os dados ficarão obsoletos e o desempenho geral do sistema decairá.

A próxima pergunta a ser feita é: qual é a solução no momento (se houver)? Muitas vezes a resposta lhe dará um desempenho de referência, bem como informações sobre como resolver o problema. Seu chefe responde que os preços das casas nos bairros são estimados manualmente por especialistas: uma equipe reúne informações atualizadas sobre um bairro e, quando não conseguem obter o preço médio, o estimam utilizando regras complexas.

Este processo é caro e demorado, e suas estimativas não são boas; nos casos em que conseguem descobrir o preço médio real do setor imobiliário, geralmente percebem que suas estimativas ficaram mais de 10% abaixo do valor. É por isso que a empresa acha útil treinar um modelo a partir de outros dados sobre esse bairro para prever o preço médio do setor imobiliário. Os dados do recenseamento

parecem um ótimo conjunto de dados para serem explorados com este propósito, pois incluem os preços médios de milhares de bairros, bem como outros dados.

Bem, com toda essa informação você já está pronto para começar a projetar seu sistema. Primeiro, você precisa enquadrar o problema: será supervisionado, sem supervisão ou um Aprendizado por Reforço? Será uma tarefa de classificação, de regressão ou outra coisa? Você deve utilizar técnicas de aprendizado em lote ou online? Antes de ler, pause e tente responder para si estas perguntas.

Você encontrou as respostas? Vamos ver: claramente temos uma tarefa típica de aprendizado supervisionado, uma vez que você recebe exemplos *rotulados* de treinamento (cada instância vem com o resultado esperado, ou seja, o preço médio do setor imobiliário do bairro). Além disso, também é uma tarefa típica de regressão, já que você é solicitado a prever um valor. Mais especificamente, trata-se de um problema de *regressão multivariada*, uma vez que o sistema utilizará múltiplas características para fazer uma previsão (ele usará a população do bairro, a renda média, etc.). No primeiro capítulo, você previu níveis de satisfação de vida com base em apenas uma característica, o PIB per capita, de modo que era um problema de *regressão univariante*. Finalmente, não há um fluxo contínuo de dados entrando no sistema, não há uma necessidade em especial de se ajustar rapidamente à mudança de dados, e os dados são pequenos o suficiente para caber na memória, de modo que o aprendizado simples em lote deve funcionar bem.



Se os dados fossem enormes, você poderia dividir seu trabalho de aprendizado em lotes em vários servidores (utilizando a técnica *MapReduce*, como veremos mais adiante), ou poderia utilizar uma técnica de *aprendizado online*.

Seleciona uma Medida de Desempenho

Seu próximo passo é selecionar uma medida de desempenho. Uma medida típica de desempenho para problemas de regressão é a *Raiz do Erro Quadrático Médio* (sigla RMSE, em inglês). Ela dá uma ideia da quantidade de erros gerados pelo sistema em suas previsões, com um peso maior para grandes erros. A Equação 2-1 mostra a fórmula matemática para calcular a RMSE.

Equação 2-1. Raiz do Erro Quadrático Médio (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

Notações

Esta equação apresenta vários apontamentos muito comuns do Aprendizado de Máquina que usaremos ao longo deste livro:

- m é o número de instâncias no conjunto de dados no qual você está medindo o RMSE.
 - Por exemplo, se estiver avaliando o RMSE em um conjunto de validação de 2 mil bairros, então $m = 2$ mil.
- $\mathbf{x}^{(i)}$ é um vetor de todos os valores da característica (excluindo o *rótulo*) da i -ésima instância no conjunto de dados, e $y^{(i)}$ é seu *rótulo* (o valor desejado da saída para aquela instância).
 - Por exemplo, se o primeiro bairro no conjunto de dados estiver localizado na longitude $-118,29^\circ$, latitude $33,91^\circ$, e ele tem 1.416 habitantes com uma renda média de US\$ 38.372, e o valor médio da casa é de US\$ 156.400 (ignorando as outras características por ora), então,

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118,29 \\ 33,91 \\ 1416 \\ 38372 \end{pmatrix}$$

e

$$y^{(1)} = 156.400$$

- \mathbf{X} é uma matriz que contém todos os valores da característica (excluindo rótulos) de todas as instâncias no conjunto de dados. Existe uma linha por instância e a i -ésima linha é igual a transposição de $\mathbf{x}^{(i)}$, notado por $(\mathbf{x}^{(i)})^T$.
 - Por exemplo, se o primeiro bairro for conforme descrito, então a matriz \mathbf{X} fica assim:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118,29 & 33,91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h é a função de previsão do seu sistema, também chamada de *hipótese*. Quando seu sistema recebe o vetor $\mathbf{x}^{(i)}$ da característica de uma instância, ele exibe um valor previsto $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ para aquela instância (\hat{y} se pronuncia “y-chapéu”).

⁴ Lembre-se de que o operador de transposição vira um vetor de coluna em um vetor de linha (e vice-versa).

- Por exemplo, se o seu sistema prevê que o preço médio no primeiro bairro é de US\$ 158.400, então $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158.400$. A previsão de erro para este bairro é $\hat{y}^{(1)} - y^{(1)} = 2.000$.
- RMSE(\mathbf{X}, h) é a função de custo medida no conjunto de exemplos utilizando sua hipótese h .

Usamos a fonte em itálico e caixa baixa para valores escalares (como m ou $y^{(i)}$) e nomes de funções (como h), fonte em negrito e caixa baixa para vetores (como $\mathbf{x}^{(i)}$) e em negrito e caixa alta para matrizes (como \mathbf{X}).

Embora a RMSE seja geralmente a medida de desempenho preferencial para tarefas de regressão, em alguns contextos você pode preferir utilizar outra função. Por exemplo, suponha que existam vários bairros outliers. Nesse caso, você pode considerar usar o *Erro Médio Absoluto* (sigla MAE, em inglês) (também chamado de Desvio Médio Absoluto; ver Equação 2-2):

Equação 2-2. Erro Médio Absoluto

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

Tanto a RMSE quanto o MAE são formas de medir a distância entre dois vetores: o vetor das previsões e o vetor dos valores alvo. Várias medidas de distância, ou *normas*, são possíveis:

- Calcular a raiz de uma soma de quadrados (RMSE) corresponde à *norma euclidiana*: é a noção de distância que você conhece. Também chamado de *norma ℓ_2* , notado por $\|\cdot\|_2$ (ou apenas $\|\cdot\|$).
- Calcular a soma de absolutos (MAE) corresponde à *norma ℓ_1* , notado por $\|\cdot\|_1$. Às vezes é chamado de *norma Manhattan* porque mede a distância entre dois pontos em uma cidade, se você só puder viajar ao longo de bairros ortogonais da cidade.
- Mais genericamente, a norma ℓ_k de um vetor \mathbf{v} que contém n elementos é definida como $\|\mathbf{v}\|_k = (\|v_0\|^k + \|v_1\|^k + \dots + \|v_n\|^k)^{\frac{1}{k}}$. ℓ_0 apenas dá o número de elementos diferentes de zero no vetor, e ℓ_∞ dá o valor máximo absoluto no vetor.
- Quanto maior o índice da norma, mais ela se concentra em grandes valores e negligencia os pequenos. É por isso que a RMSE é mais sensível a outliers do que o MAE. Mas, quando os outliers são exponencialmente raros (como em uma curva em forma de sino), a RMSE funciona muito bem, e geralmente é a preferida.

Verifique as Hipóteses

Por último, uma boa prática é listar e verificar as hipóteses que foram feitas até agora (por você ou outros); fica mais fácil pegar problemas sérios logo no início. Por exemplo, os preços dos bairros mostrados pelo seu sistema serão alimentados em um sistema downstream do Aprendizado de Máquina, e assumiremos que esses preços serão usados como tal. Mas, e se o sistema downstream converter os preços em categorias (por exemplo, “barato”, “médio” ou “caro”) e utilizar essas categorias em vez dos próprios preços? Neste caso, não é importante saber o preço exato se o seu sistema só precisa obter a categoria certa. Se for assim, então o problema deveria ter sido enquadrado como uma tarefa de classificação, não uma tarefa de regressão. Você não vai querer descobrir isso após trabalhar por meses em um sistema de regressão.

Felizmente, depois de conversar com a equipe responsável pelo sistema downstream, você está confiante de que realmente precisa dos preços reais e não apenas das categorias. Ótimo! Você está pronto, as luzes estão verdes e você pode começar a programar agora!

Obtenha os Dados

É hora de colocar as mãos na massa. Não hesite em pegar seu laptop e acompanhar os seguintes exemplos de código em um notebook do Jupyter, que está disponível na íntegra em <https://github.com/ageron/handson-ml>.

Crie o Espaço de Trabalho

Primeiro, você precisará ter o Python instalado. Provavelmente ele já está instalado em seu sistema. Caso contrário, você pode baixá-lo em <https://www.python.org/>.⁵

Em seguida, crie um diretório de trabalho para seu código e conjuntos de dados do Aprendizado de Máquina. Abra um terminal e digite os seguintes comandos (após o prompts \$):

```
$ export ML_PATH="$HOME/ml"      # Mude o caminho se preferir  
$ mkdir -p $ML_PATH
```

Você precisará de vários módulos Python: Jupyter, NumPy, Pandas, Matplotlib e Scikit-Learn. Se você já tem o Jupyter rodando com todos esses módulos instalados, pode pular com segurança para “Baixe os dados” na página 45. Se você ainda não tem, há várias

⁵ A versão mais recente do Python 3 é recomendada. O Python 2.7+ também funcionará bem, mas está obsoleto. Se usar o Python 2, você deve adicionar `from future import division, print_function, unicode_literals` no início do seu código.

maneiras de instalá-los (e suas dependências). Você pode utilizar o sistema de empacotamento do seu sistema (por exemplo, apt-get no Ubuntu, ou MacPorts ou HomeBrew no macOS), instale uma distribuição científica do Python, como o Anaconda, e utilize seu sistema de empacotamento, ou utilize o sistema de empacotamento do Python, pip, que está incluído por padrão com os instaladores binários Python (desde a versão Python 2.7.9).⁶ Você pode verificar se o pip está instalado digitando o seguinte comando:

```
$ pip3 --version  
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

Você deve se certificar de que tenha uma versão recente do pip instalada, pelo menos > 1.4 para suportar a instalação do módulo binário (também conhecida como wheels). Para atualizar o módulo do pip, digite:⁷

```
$ pip3 install --upgrade pip  
Collecting pip  
[...]  
Successfully installed pip-9.0.1
```

Criando um Ambiente Isolado

Se você quiser trabalhar em um ambiente isolado (o que é fortemente recomendado para que se possa trabalhar em projetos diferentes sem ter versões conflitantes de biblioteca), instale o virtualenv executando o seguinte comando pip:

```
$ pip3 install --user --upgrade virtualenv  
Collecting virtualenv  
[...]  
Successfully installed virtualenv
```

Agora você pode criar um ambiente Python isolado digitando:

```
$ cd $ML_PATH  
$ virtualenv env  
Using base prefix '[...]'  
New python executable in [...]/ml/env/bin/python3.5  
Also creating executable in [...]/ml/env/bin/python  
Installing setuptools, pip, wheel...done.
```

Agora, sempre que quiser ativar esse ambiente, basta abrir um terminal e digitar:

```
$ cd $ML_PATH  
$ source env/bin/activate
```

⁶ Mostraremos a seguir os passos para a instalação usando pip em uma *bash shell* no Linux ou no macOS. Talvez seja preciso adaptar estes comandos ao seu próprio sistema. Para o Windows, recomendamos a instalação do Anaconda.

⁷ Talvez seja necessário ter direitos de administrador para usar este comando; se for o caso, tente prefixá-lo com *sudo*.

Qualquer pacote que você instalar utilizando pip será instalado neste ambiente isolado enquanto o ambiente estiver ativo, e o Python somente terá acesso a esses pacotes (se você também quiser acesso aos pacotes do sistema no site, deverá criar o ambiente utilizando a opção `--system-site-packages` do virtualenv). Confira a documentação do virtualenv para obter mais informações.

Agora você pode instalar todos os módulos necessários e suas dependências utilizando este simples comando pip (se você não estiver utilizando um virtualenv, precisará de direitos de administrador ou adicionar a opção `--user`):

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

Para verificar sua instalação, tente importar todos os módulos assim:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

Não deve haver nenhuma saída e nenhum erro. Agora você pode iniciar o Jupyter digitando:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

Um servidor Jupyter está sendo executado agora pela porta 8888 em seu terminal. Você pode visitar este servidor abrindo seu navegador em <http://localhost:8888/> (isso geralmente acontece automaticamente quando o servidor é iniciado). Você deve ver seu diretório de trabalho vazio (contendo apenas o diretório `env` se você seguiu as instruções anteriores do virtualenv).

Agora, crie um novo notebook Python clicando no botão New e selecionando a versão adequada do Python⁸ (veja a Figura 2-3).

Isto faz três coisas: primeiro, cria um novo arquivo notebook chamado `Untitled.ipynb` em seu espaço de trabalho; segundo, inicia um *Jupyter Python kernel* para rodar este notebook; e, terceiro, abre este notebook em uma nova guia. Você deve começar clicando em Untitled e digitando o novo nome, renomeando este notebook para “Housing” (isso mudará o arquivo automaticamente para `Housing.ipynb`).

⁸ Note que o Jupyter pode lidar com várias versões do Python, e até muitas outras linguagens, como R ou Octave.

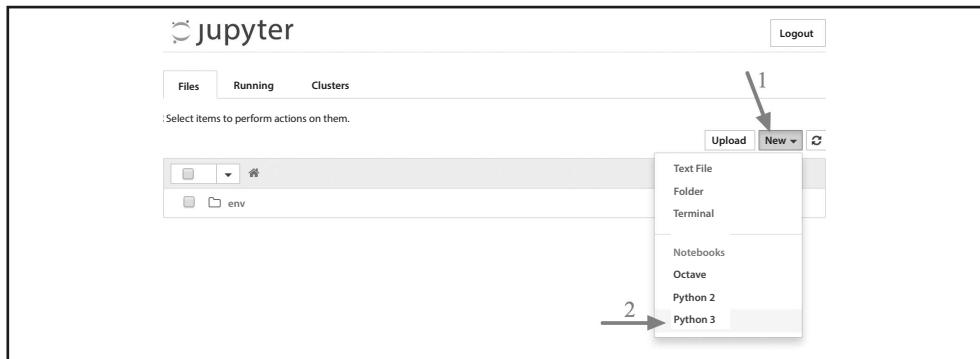


Figura 2-3. Seu espaço de trabalho no Jupyter

Um notebook contém uma lista de células. Cada célula pode conter código executável ou texto formatado. No momento, o notebook contém apenas uma célula de código vazio, rotulada “In [1]”. Tente digitar `print("Hello world!")` na célula, e clique no botão Reproduzir (veja a Figura 2-4) ou pressione Shift-Enter. Isso envia a célula atual para o *kernel* Python deste notebook, que o executa e retorna a saída. O resultado é exibido abaixo da célula, e, já que chegamos ao final do notebook, uma nova célula é automaticamente criada. Confira o User Interface Tour no menu de Ajuda do Jupyter para aprender o básico.

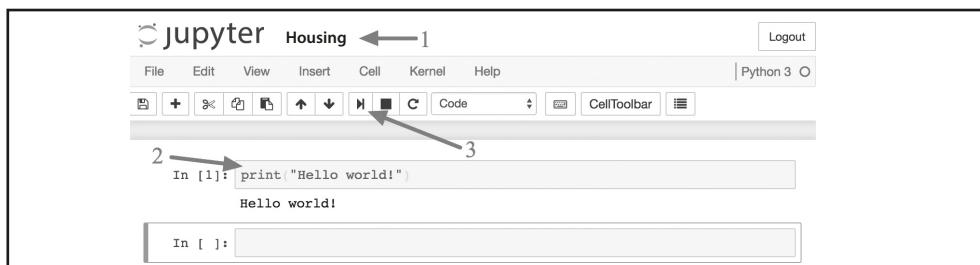


Figura 2-4. Notebook Python Hello world

Baixe os Dados

Em ambientes típicos, seus dados estariam disponíveis em um banco de dados relacional (ou algum outro armazenamento comum de dados) e espalhados por várias tabelas/documentos/arquivos. Para acessá-lo, primeiro você precisaria obter suas credenciais e autorizações⁹ de acesso e familiarizar-se com o esquema de dados. Neste projeto, no entanto, as coisas são muito mais simples: você só vai baixar um único arquivo compactado,

⁹ Você também pode precisar verificar restrições legais, como campos privados que nunca devem ser copiados para armazenamentos inseguros de dados.

housing.tgz, que contém um arquivo com valores separados por vírgulas (CSV) chamado *housing.csv* com todos os dados.

Você poderia usar o navegador da Web para baixá-lo e executar o `tar xzf housing.tgz` para descompactá-lo e extrair o arquivo CSV, mas é preferível criar uma pequena função para fazer isso. Ela será útil, principalmente, se os dados mudarem regularmente, pois permite que você escreva um pequeno script que poderá ser executado sempre que precisar para buscar os dados mais recentes (ou poderá configurar um trabalho agendado para fazer isso automaticamente em intervalos regulares). Também será útil automatizar o processo de busca caso você precise instalar o conjunto de dados em várias máquinas.

Esta é a função para buscar os dados:¹⁰

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")

    urlib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Agora, quando chamamos `fetch_housing_data()`, ele cria um diretório *datasets/housing* no seu espaço de trabalho, baixa o arquivo *housing.tgz* e extrai o *housing.csv* neste diretório.

Agora, vamos carregar os dados com o Pandas. Mais uma vez você deve escrever uma pequena função para carregá-los:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

Esta função retorna um objeto *DataFrame Pandas* contendo todos os dados.

¹⁰ Em um projeto real, você salvaria este código em um arquivo Python, mas, por enquanto, você pode apenas escrevê-lo no seu notebook do Jupyter.

Uma Rápida Olhada na Estrutura dos Dados

Utilizando o método `head()` do DataFrame, vejamos as cinco linhas superiores (veja Figura 2-5).

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Figura 2-5. As cinco linhas superiores no conjunto de dados

Cada linha representa um bairro. Existem 10 atributos (você pode ver os primeiros seis na captura de tela): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` e `ocean_proximity`.

O método `info()` é útil para a obtenção de uma rápida descrição dos dados, em especial o número total de linhas, o tipo de cada atributo e o número de valores não nulos (veja a Figura 2-6).

In [6]:	housing.info()
	<class 'pandas.core.frame.DataFrame'> RangeIndex: 20640 entries, 0 to 20639 Data columns (total 10 columns): longitude 20640 non-null float64 latitude 20640 non-null float64 housing_median_age 20640 non-null float64 total_rooms 20640 non-null float64 total_bedrooms 20433 non-null float64 population 20640 non-null float64 households 20640 non-null float64 median_income 20640 non-null float64 median_house_value 20640 non-null float64 ocean_proximity 20640 non-null object dtypes: float64(9), object(1) memory usage: 1.6+ MB

Figura 2-6. Informações das Casas

Existem 20.640 instâncias no conjunto de dados, o que significa que é muito pequeno para os padrões de Aprendizado de Máquina, mas é perfeito para começar. Repare que o atributo `total_bedrooms` tem apenas 20.433 valores não nulos, significando que 207 bairros não possuem esta característica. Cuidaremos disso mais tarde.

Todos os atributos são numéricos, exceto o campo `ocean_proximity`. Seu tipo é `object`, então ele poderia conter qualquer tipo de objeto Python, mas, como você carregou esses dados de um arquivo CSV, você sabe que ele deve ser um atributo de texto. Quando você olhou para as cinco primeiras linhas, provavelmente percebeu que os valores na coluna `ocean_proximity` eram repetitivos, o que significa que é provavelmente um atributo categórico. Você pode descobrir quais categorias existem e quantos bairros pertencem a cada categoria utilizando o método `value_counts()`:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: ocean_proximity, dtype: int64
```

Vamos olhar para outros campos. O método `describe()` mostra um resumo dos atributos numéricos (Figura 2-7).

In [8]:	housing.describe()					
Out[8]:		longitude	latitude	housing_median_age	total_rooms	total_bedrooms
	count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
	mean	-119.569704	35.631861	28.639486	2635.763081	537.870556
	std	2.003532	2.135952	12.585558	2181.615252	421.385076
	min	-124.350000	32.540000	1.000000	2.000000	1.000000
	25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
	50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
	75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
	max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Figura 2-7. Resumo de cada atributo numérico

As linhas `count`, `mean`, `min` e `max` são autoexplicativas. Observe que os valores nulos são ignorados (então, por exemplo, `count` do `total_bedrooms` é 20.433, não 20.640). A linha `std` mostra o *desvio padrão*, que mede a dispersão dos valores.¹¹ As linhas `25%`, `50%` e `75%` mostram os *percentis* correspondentes: um percentil indica o valor abaixo no qual uma dada porcentagem cai em um grupo de observações. Por exemplo, 25% dos bairros tem uma `housing_median_age` menor que 18, enquanto 50% é menor que 29, e 75% é menor que 37. Estes são frequentemente chamados de 25º percentil (ou 1º quartil), a média e o 75º percentil (ou 3º quartil).

¹¹ O desvio padrão é geralmente denotado σ (da letra grega sigma) e é a raiz quadrada da *variância*, que é a média do desvio quadrado da média. Quando uma característica tem uma *distribuição normal* em forma de sino (também chamada de *distribuição Gaussiana*), que é muito comum, aplica-se a regra “68-95-99,7”: cerca de 68% dos valores se enquadram em 1σ da média, 95% dentro de 2σ e 99,7% dentro de 3σ .

Outro método rápido de perceber o tipo de dados com o qual você está lidando é traçar um histograma para cada atributo numérico. Um histograma mostra o número de instâncias (no eixo vertical) que possuem um determinado intervalo de valores (no eixo horizontal). Você pode traçar esse único atributo por vez ou pode chamar o método `hist()` em todo o conjunto de dados e traçar um histograma para cada atributo numérico (veja a Figura 2-8). Por exemplo, você pode ver que pouco mais de 800 bairros possuem um `median_house_value` equivalente a mais ou menos US\$ 100 mil.

```
%matplotlib inline # somente no notebook Jupyter
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



O método `hist()` depende do Matplotlib, que por sua vez depende de um backend gráfico especificado pelo usuário para figurar em sua tela. Então, antes que você consiga plotar qualquer coisa, é preciso especificar qual backend o Matplotlib deve usar. A opção mais simples é usar o comando mágico do Jupyter `%matplotlib inline`. Isso leva o Jupyter a configurar o Matplotlib para que ele utilize seu próprio backend. As plotagens são então processadas dentro do próprio notebook. Observe que é opcional em um notebook do Jupyter chamar o `show()`, pois ele exibirá gráficos automaticamente sempre que uma célula for executada.

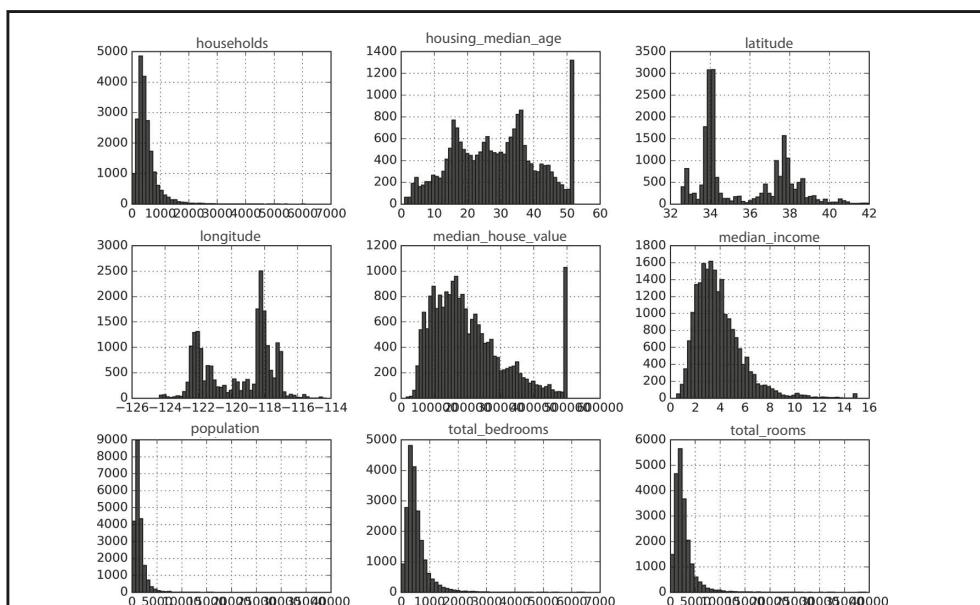
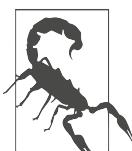


Figura 2-8. Um histograma para cada atributo numérico

Preste atenção em alguns pontos destes histogramas:

1. Primeiro, o atributo da renda média não parece estar expresso em dólares americanos (USD). Depois de verificar com a equipe de coleta de dados, você é informado que os dados foram dimensionados e limitados em 15 (na verdade 15,0001) para a média dos maiores rendimentos, e em 0,5 (na verdade 0,4999) para a média dos rendimentos mais baixos. É comum trabalhar com atributos pré-processados no Aprendizado de Máquina e isto não é necessariamente um problema, mas você deve tentar entender como os dados foram calculados.
2. A idade média e o valor médio da casa também foram limitados. Este último pode ser um problema sério, pois é seu atributo alvo (seus rótulos). Os algoritmos de Aprendizado de Máquina podem aprender que os preços nunca ultrapassam esse limite. Você precisa verificar com a equipe do seu cliente (a equipe que usará a saída do seu sistema) para ver se isso é ou não um problema. Se eles disserem que precisam de previsões precisas mesmo acima de US\$ 500 mil, então você terá duas opções:
 - a. Coletar rótulos adequados para os bairros cujos rótulos foram limitados.
 - b. Remover esses bairros do conjunto de treinamento (e também do conjunto de testes, já que seu sistema não deve ser responsabilizado por prever valores além de US\$ 500 mil).
3. Esses atributos têm escalas muito diferentes. Discutiremos isso mais adiante neste capítulo, quando explorarmos o escalonamento das características.
4. Finalmente, muitos histogramas têm um *rastro alongado*: eles se estendem muito mais à direita da média do que à esquerda. Isso pode dificultar a detecção de padrões em alguns algoritmos do Aprendizado de Máquina. Vamos tentar transformar esses atributos mais tarde para conseguir mais distribuições na forma de sino.

Espero que você tenha agora uma melhor compreensão do tipo de dados com os quais está lidando.



Espere! Antes de analisar ainda mais os dados, você precisa criar um conjunto de teste, colocá-lo de lado e nunca checá-lo.

Crie um Conjunto de Testes

Pode parecer estranho colocar de lado, voluntariamente, uma parte dos dados nesta fase. Afinal, você só deu uma rápida olhada nos dados e certamente precisa aprender muito mais a respeito antes de decidir quais algoritmos usar, certo? Isso é verdade, mas o seu cérebro é um incrível sistema de detecção de padrões, o que significa que é altamente propenso ao *sobreajuste*: se você olhar para o conjunto de teste, pode tropeçar em algum padrão aparentemente interessante nos dados de teste que o levará a selecionar um tipo particular de modelo do Aprendizado de Máquina. Quando você estima o erro de generalização utilizando o conjunto de teste, sua estimativa será muito otimista e você lançará um sistema que não funcionará tão bem quanto o esperado. Isso é chamado de *data snooping bias*.

Criar um conjunto de testes é, teoricamente, bastante simples: basta escolher aleatoriamente algumas instâncias, geralmente 20% do conjunto de dados, e colocá-las de lado:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Você pode usar essa função assim:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

Bem, isso funciona, mas não é perfeito: ao executar o programa novamente, será gerado um conjunto diferente de testes! Ao longo do tempo, você (ou seus algoritmos do Aprendizado de Máquina) verá todo o conjunto de dados, o que deve ser evitado.

Uma solução seria salvar o conjunto de testes na primeira execução e depois carregá-lo em execuções subsequentes. Outra opção é definir a semente do gerador de números aleatórios (por exemplo, `np.random.seed(42)`)¹² antes de chamar a `np.random.permutation()`, de modo que ele sempre gere os mesmos índices embaralhados.

Mas ambas as soluções serão interrompidas na próxima vez que você buscar um conjunto de dados atualizado. Uma solução comum é utilizar o identificador de cada instância para decidir se ela deve ou não ir no conjunto de teste (supondo que as instâncias tenham um identificador único e imutável). Por exemplo, você pode calcular um hash do identi-

¹² Você verá com frequência pessoas colocarem a *random seed* em 42. Esse número não possui propriedade especial alguma além de ser A Resposta para a Vida, para o Universo e Tudo Mais.

fificador de cada instância, manter apenas o último byte do hash e colocar a instância no conjunto de teste se esse valor for menor ou igual a 51 (~20% de 256). Isso garante que o conjunto de teste permanecerá consistente em várias execuções, mesmo ao atualizar o conjunto de dados. O novo conjunto de testes conterá 20% das novas instâncias, mas não conterá nenhuma instância que já estivesse no conjunto de treinamento. Veja uma possível implementação:

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Infelizmente, o conjunto de dados do setor imobiliário não possui uma coluna de identificação. A solução mais simples é utilizar o índice da linha como ID:

```
housing_with_id = housing.reset_index() # adiciona uma coluna 'index'
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

Se você utilizar o índice de linha como um identificador exclusivo, precisa se certificar de que novos dados sejam anexados ao final do conjunto de dados e que nenhuma linha seja excluída. Se isso não for possível, tente utilizar características mais estáveis para criar um identificador exclusivo. Por exemplo, a latitude e a longitude de um bairro serão certamente estáveis por alguns milhões de anos, então você pode combiná-las em um ID dessa forma:¹³

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

O Scikit-Learn fornece algumas funções para dividir conjuntos de dados em vários subconjuntos de diversas maneiras. A função mais simples é `train_test_split`, que faz praticamente o mesmo que a função `split_train_test` definida anteriormente, mas com alguns recursos adicionais. Primeiro, temos um parâmetro `random_state` que permite que você defina a semente do gerador de números aleatórios como explicado anteriormente e, em segundo lugar, você pode passar múltiplos conjuntos de dados com um número idêntico de linhas, e ele os dividirá nos mesmos índices (isso é muito útil se, por exemplo, você tiver um `DataFrame` separado para os rótulos):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

¹³ A informação de localização é realmente bastante grosseira e, como resultado, muitos bairros terão exatamente a mesma identificação, acabando no mesmo conjunto (*test* ou *train*). Infelizmente, isso introduz algum viés de amostragem.

Até agora consideramos somente métodos de amostragem puramente aleatórios. Isso geralmente é bom se o seu conjunto de dados for suficientemente grande (especialmente em relação ao número de atributos), mas, se não for, corre o risco de apresentar um viés significativo de amostragem. Quando uma empresa de pesquisa decide ligar para mil pessoas e lhes fazer algumas perguntas, eles não escolhem aleatoriamente mil pessoas em uma lista telefônica. Eles tentam garantir que essas mil pessoas representem toda a população. Por exemplo, a população dos EUA é composta por 51,3% de pessoas do sexo feminino e 48,7% do sexo masculino, de modo que uma pesquisa bem conduzida tentaria manter essa proporção na amostragem: 513 mulheres e 487 homens. Isso é chamado de *amostragem estratificada*: a população é dividida em subgrupos homogêneos, chamados de *estratos*, e o número certo de instâncias de cada estrato é amostrado para garantir que o conjunto de testes seja representativo da população em geral. Se eles utilizassem amostragem puramente aleatória, haveria cerca de 12% de chance de amostrar um conjunto de teste distorcido, tanto com menos de 49% feminino quanto com mais de 54%. De qualquer forma, os resultados da pesquisa seriam significativamente tendenciosos.

Suponha que você tenha conversado com especialistas que lhe disseram que a renda média é um atributo muito importante para estimar os preços médios. Você quer garantir que o conjunto de testes seja representativo das várias categorias de rendimentos em todo o conjunto de dados. Uma vez que a renda média é um atributo numérico contínuo, primeiro você precisa criar um atributo na categoria da renda. Vejamos mais de perto o histograma da renda média (de volta à Figura 2-8): a maioria dos valores médios da renda está agrupada em torno de US\$ 20 mil–US\$ 50 mil mas alguns rendimentos médios ultrapassam os US\$ 60 mil. É importante ter um número suficiente de instâncias para cada estrato em seu conjunto de dados, ou então a estimativa da importância do estrato poderá ser tendenciosa. Isso significa que você não deve ter muitos estratos, e cada estrato deve ser grande o suficiente. O código a seguir cria um atributo da categoria da renda dividindo a renda média por 1,5 (para limitar o número de categorias da renda) e arredondando com a utilização do `ceil` (para ter categorias discretas) e, em seguida, mesclando todas as categorias maiores que 5, na categoria 5:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

Essas categorias de renda estão representadas na (Figura 2-9):

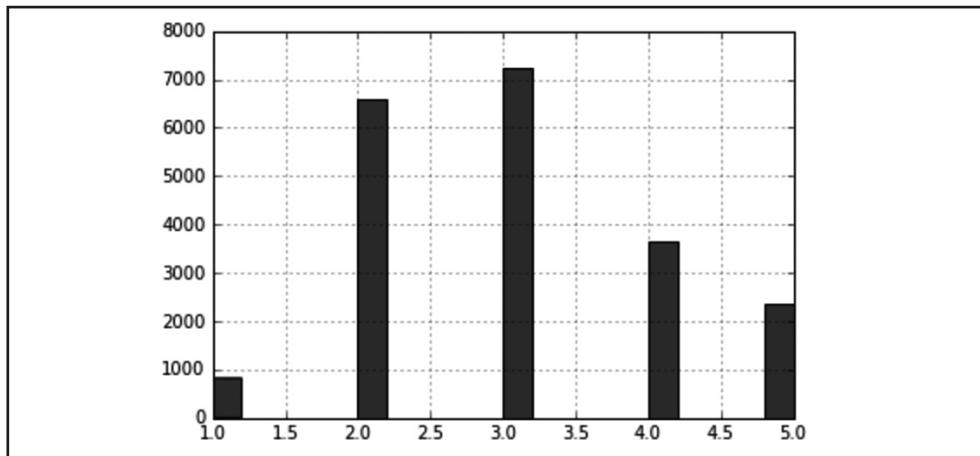


Figura 2-9. Histograma das categorias de renda

Agora você está pronto para fazer uma amostragem estratificada com base na categoria da renda. Para isso você pode utilizar a classe `StratifiedShuffleSplit` do Scikit-Learn:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Vamos ver se isso funcionou como esperado. Você pode começar pela análise das proporções da categoria de renda no conjunto de testes:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3.0    0.350533
2.0    0.318798
4.0    0.176357
5.0    0.114583
1.0    0.039729
Name: income_cat, dtype: float64
```

Com um código similar, você pode medir as proporções da categoria de renda no conjunto completo de dados. A Figura 2-10 compara as proporções da categoria de renda no conjunto geral de dados, no conjunto de teste gerado com a amostragem estratificada e em um conjunto de testes gerado a partir da amostragem puramente aleatória. Como você pode ver, o conjunto de testes gerado com a utilização da amostragem estratificada tem proporções da categoria de renda quase idênticas às do conjunto completo de dados, enquanto o conjunto de testes gerado com amostragem puramente aleatória é bastante distorcido.

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

Figura 2-10. Comparação de viés de amostragem estratificada versus amostragem aleatória

Agora, você deve remover o atributo `income_cat` para que os dados voltem ao seu estado original:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Ficamos um tempo na geração de conjuntos de testes por uma boa razão: esta parte crítica é muitas vezes negligenciada em um projeto de Aprendizado de Máquina. Além disso, muitas dessas ideias serão úteis mais tarde quando discutirmos a validação cruzada. Agora, é hora de avançar para o próximo estágio: explorar os dados.

Descubra e Visualize os Dados para Obter Informações

Até aqui você deu apenas uma rápida olhada para ter uma compreensão geral do tipo de dados que está manipulando. Agora, o objetivo é aprofundar-se um pouco mais.

Primeiro, certifique-se de colocar o teste de lado e apenas explorar o conjunto de treinamento. Além disso, se o conjunto de treinamento for muito grande, talvez seja melhor experimentar um conjunto de exploração para fazer manipulações fácil e rapidamente. No nosso caso, o conjunto é muito pequeno, então você pode trabalhar diretamente no conjunto completo. Criaremos uma cópia para que você possa treinar com ela sem prejudicar o conjunto de treinamento:

```
housing = strat_train_set.copy()
```

Visualizando Dados Geográficos

Como existem informações geográficas (latitude e longitude), é uma boa ideia criar um diagrama de dispersão para visualizar os dados de todos os bairros (Figura 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

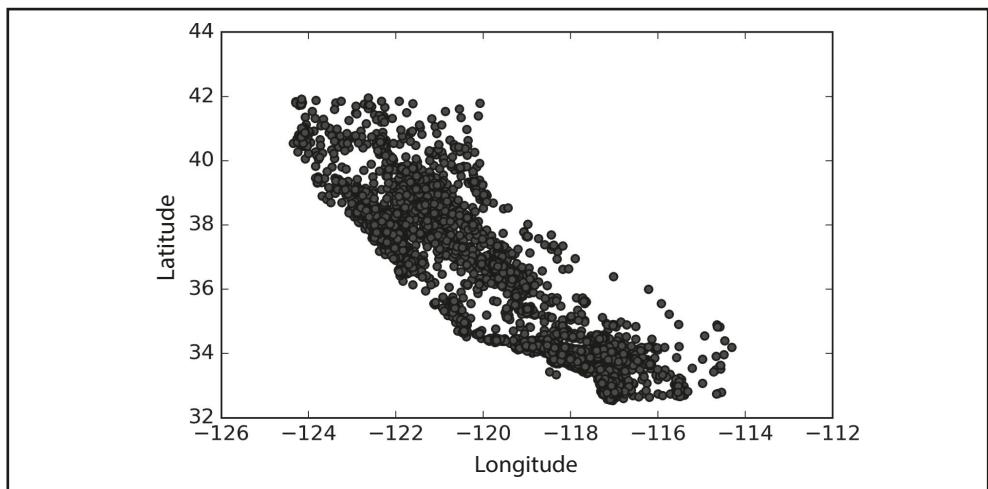


Figura 2-11. Um diagrama de dispersão geográfica dos dados

Isso se parece com a Califórnia, mas, além disso, é difícil ver qualquer padrão específico. Definir a opção `alpha` em `0,1` facilita a visualização dos locais onde existe uma alta densidade de pontos de dados (Figura 2-12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

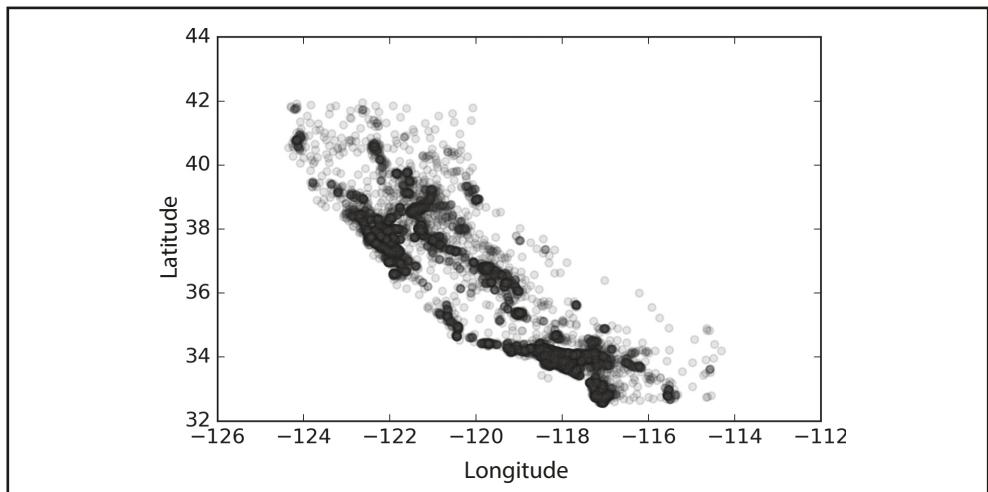


Figura 2-12. Uma melhor visualização destacando áreas de alta densidade

Agora já está melhor: é possível ver claramente as áreas de alta densidade, especificamente a Área da Baía de Los Angeles e San Diego, além de uma longa linha de alta densidade no Vale Central, principalmente ao redor de Sacramento e Fresno.

No geral, nosso cérebro é muito bom em detectar padrões em imagens, mas talvez seja necessário brincar com parâmetros de visualização para que os padrões se destaquem.

Agora, vejamos os preços do setor imobiliário (Figura 2-13). O raio de cada círculo representa a população do bairro (opção `s`) e a cor representa o preço (opção `c`). Usaremos um mapa de cores pré-definido (opção `cmap`) chamado `jet`, que varia do azul (valores baixos) para o vermelho (preços altos).¹⁴

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
             s=housing["population"]/100, label="population", figsize=(10,7),
             c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

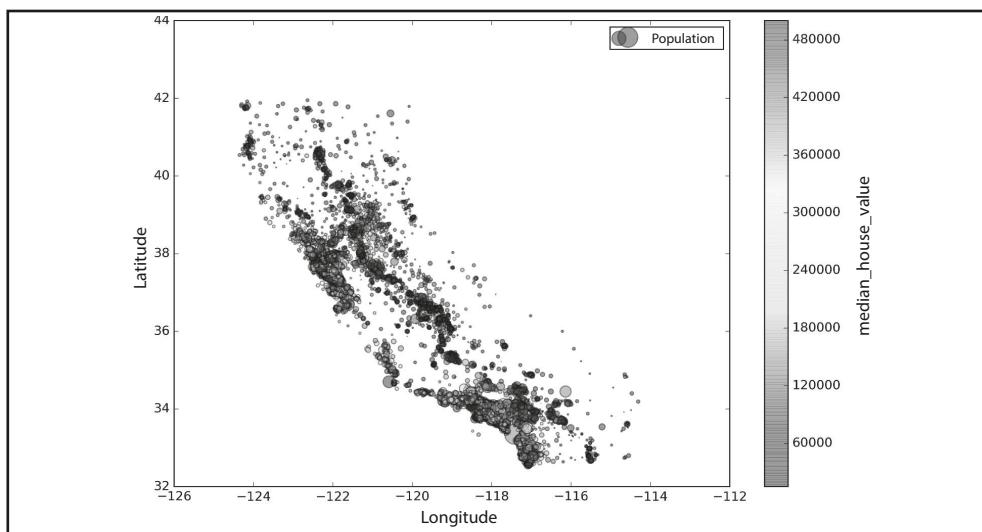


Figura 2-13. Preços das casas na Califórnia

Esta imagem informa que os preços do setor imobiliário estão muito relacionados à localização (por exemplo, perto do oceano) e à densidade populacional, como provavelmente você já sabia. Será útil utilizar um algoritmo de agrupamento para detectar os grupos principais e adicionar novas características que medem a proximidade com os centros de agrupamento. O atributo de proximidade do oceano também pode ser útil, embora na costa norte da Califórnia os preços não sejam muito altos, então essa não é uma regra simples.

14 Se você estiver lendo em escala de cinza, pegue uma caneta vermelha e rabisque na maior parte do litoral da Área da Baía até San Diego (como esperado). Você também pode acrescentar um trecho amarelo ao redor de Sacramento.

Buscando Correlações

Uma vez que o conjunto de dados não é muito grande, você pode calcular facilmente o *coeficiente de correlação padrão* (também chamado *r de Pearson*) entre cada par de atributos utilizando o método `corr()`:

```
corr_matrix = housing.corr()
```

Agora, vejamos o quanto cada atributo se correlaciona com o valor médio da habitação:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms          0.135231
housing_median_age   0.114220
households           0.064702
total_bedrooms        0.047865
population            -0.026699
longitude             -0.047279
latitude              -0.142826
Name: median_house_value, dtype: float64
```

O coeficiente de correlação varia de -1 a 1. Quando está próximo de 1, significa que existe uma forte correlação positiva; por exemplo, o valor médio da habitação tende a aumentar quando a renda média aumenta. Quando o coeficiente está próximo de -1, significa que existe uma forte correlação negativa; é possível ver uma pequena correlação negativa entre a latitude e o valor médio da habitação (ou seja, os preços tendem a diminuir quando você vai para o norte). Finalmente, coeficientes próximos de zero significam que não há correlação linear. A Figura 2-14 mostra várias plotagens juntamente com o coeficiente de correlação entre seus eixos horizontal e vertical.

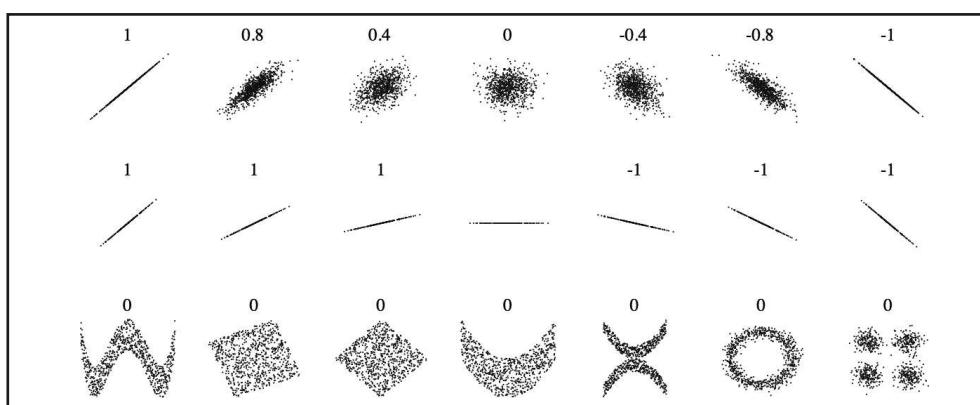
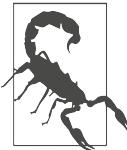


Figura 2-14. Coeficiente de correlação padrão de vários conjuntos de dados (fonte: Wikipedia; imagem de domínio público)



O coeficiente de correlação apenas mede correlações lineares (“se x sobe, então y geralmente sobe/desce”). Ele pode perder completamente as relações não lineares (por exemplo, “se x é próximo a zero, então, y , geralmente, sobe”). Observe como todas as plotagens da linha inferior têm um coeficiente de correlação igual a zero, apesar do fato de seus eixos claramente não serem independentes: são exemplos de relações não lineares. Além disso, a segunda linha mostra exemplos em que o coeficiente de correlação é igual a 1 ou -1; observe que isso não tem nada a ver com a inclinação. Por exemplo, sua altura em polegadas tem um coeficiente de correlação de 1 com sua altura em pés ou em nanômetros.

Outra maneira de verificar a correlação entre atributos é utilizar a função `scatter_matrix`, do Pandas, que plota cada atributo numérico em relação a qualquer outro atributo numérico. Uma vez que existem 11 atributos numéricos, você obteria $11^2 = 121$ plotagens, o que não caberia em uma página, então focaremos apenas em alguns atributos promissores que parecem mais correlacionados com o valor médio do setor imobiliário (Figura 2-15):

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
               "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

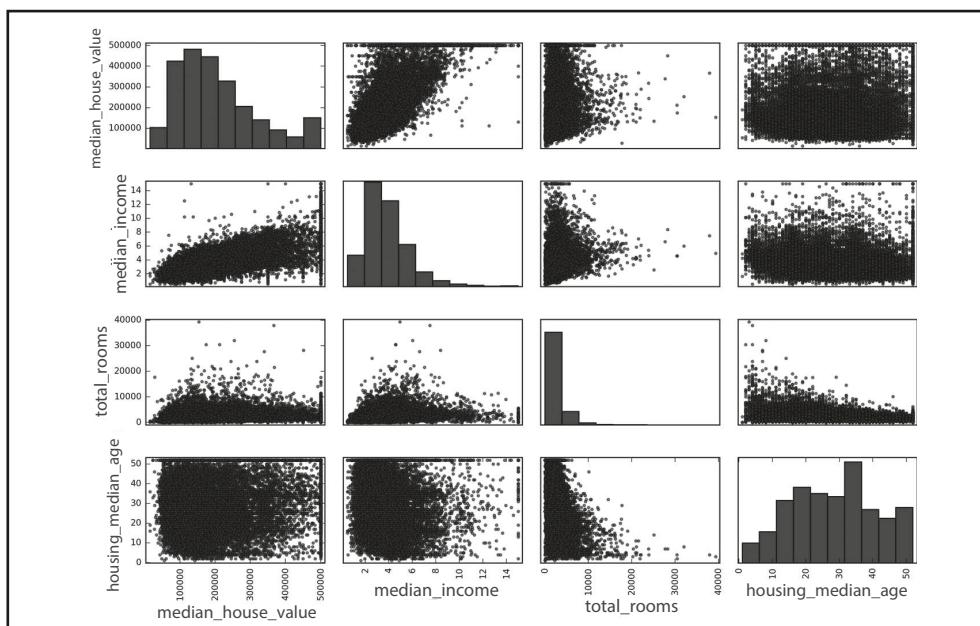


Figura 2-15. Matriz de dispersão

A diagonal principal (superior esquerda até a parte inferior direita) seria cheia de linhas retas se o Pandas plotasse cada variável em relação a si mesma, o que não seria muito útil. Então, em vez disso, o Pandas exibe um histograma para cada atributo (outras opções estão disponíveis, veja a documentação do Pandas para mais detalhes).

O atributo mais promissor para prever o valor médio da habitação é a renda média, então vamos observar o gráfico de dispersão de correlação (Figura 2-16):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
alpha=0.1)
```

Esta plotagem revela algumas coisas. Primeiro, a correlação é realmente muito forte; é possível ver claramente a tendência ascendente, e os pontos não estão muito dispersos. Em segundo lugar, o limite de preços que percebemos anteriormente é claramente visível como uma linha horizontal em US\$ 500 mil. Mas esta plotagem revela outras linhas retas menos óbvias: uma horizontal em torno de US\$ 450 mil, outra em torno de US\$ 350 mil, talvez uma em torno de US\$ 280 mil, e mais algumas abaixo disso. Você pode tentar remover os bairros correspondentes para evitar que seus algoritmos aprendam a reproduzir essas peculiaridades dos dados.

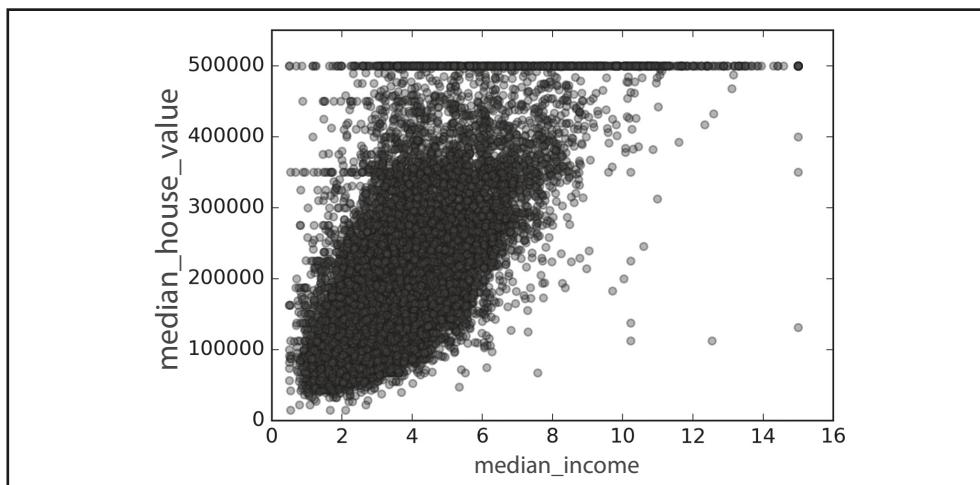


Figura 2-16. Renda média versus valor médio da habitação

Experimentando com Combinações de Atributo

Esperamos que as seções anteriores tenham dado a você uma ideia de algumas maneiras de explorar os dados e obter informações. Você identificou algumas peculiaridades dos dados que talvez queira limpar antes de fornecê-los a um algoritmo de Aprendizado de Máquina, e também encontrou correlações interessantes entre atributos, especialmente

com o atributo-alvo. Também foi possível perceber que alguns atributos têm um rastro de distribuição prolongado, então pode ser que você queira transformá-los (por exemplo, ao calcular seu logaritmo). Claro, o seu raio de ação variará consideravelmente a cada projeto, mas as ideias gerais são semelhantes.

Uma última coisa que você pode querer fazer é tentar várias combinações de atributos antes de preparar os dados para os algoritmos de Aprendizado de Máquina. Por exemplo, o número total de cômodos em um bairro não terá muita utilidade se você não souber quantos domicílios existem. O que você realmente quer é o número de cômodos por domicílio. Da mesma forma, o número total de quartos por si só não é muito útil: você provavelmente vai querer compará-lo com o número de cômodos. E a população por domicílio também parece uma combinação de atributos interessante. Criaremos esses novos atributos:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

E, agora, vejamos a matriz de correlação novamente:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income          0.687160
rooms_per_household   0.146285
total_rooms            0.135097
housing_median_age     0.114110
households             0.064506
total_bedrooms         0.047689
population_per_household -0.021985
population              -0.026920
longitude                -0.047432
latitude                  -0.142724
bedrooms_per_room       -0.259984
Name: median_house_value, dtype: float64
```

Ei, nada mal! O novo atributo `bedrooms_per_room` está muito mais correlacionado com o valor médio da habitação do que com o número total de cômodos ou quartos. Aparentemente, habitações com uma baixa relação quarto/cômodo tendem a ser mais caras. O número de cômodos por família também é mais informativo do que o número total de cômodos em um bairro — obviamente, quanto maiores as habitações, mais caras elas serão.

Esta rodada de exploração não precisa ser absolutamente minuciosa; o objetivo é começar com o pé direito e rapidamente obter informações que o ajudarão a produzir um primeiro protótipo razoavelmente bom. Mas este é um processo iterativo: uma vez que você obtiver um protótipo funcional, poderá analisar sua saída para adquirir mais informações e voltar a este passo da exploração.

Prepare os Dados para Algoritmos do Aprendizado de Máquina

É hora de preparar os dados para seus algoritmos de Aprendizado de Máquina. Em vez de fazer isso manualmente, escreva funções, por vários bons motivos:

- Isso permitirá que você reproduza essas transformações facilmente em qualquer conjunto de dados (por exemplo, na próxima vez que você receber um novo conjunto de dados);
- Você gradualmente construirá uma biblioteca de funções de transformação que poderão ser reutilizadas em projetos futuros;
- Você pode usar essas funções em seu sistema ao vivo para transformar os novos dados antes de fornecê-lo aos seus algoritmos;
- Isso possibilitará que você tente várias transformações facilmente e veja qual combinação funciona melhor.

Mas primeiro vamos reverter para um conjunto de treinamento limpo (copiando `strat_train_set` mais uma vez), e vamos separar os previsores e os rótulos, uma vez que não queremos necessariamente aplicar as mesmas transformações às previsões e aos valores-alvo (observe que `drop()` cria uma cópia dos dados e não afeta `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Limpando os Dados

A maioria dos algoritmos de Aprendizado de Máquina não pode funcionar com características faltantes, então criaremos algumas funções para cuidar delas. Você notou anteriormente que o atributo `total_bedrooms` tem alguns valores faltantes, então vamos consertar isso. Há três opções:

- Livrar-se dos bairros correspondentes;
- Livrar-se de todo o atributo;
- Definir valores para algum valor (zero, a média, intermediária, etc.).

Você pode fazer isso facilmente utilizando os métodos `dropna()`, `drop()` e `fillna()` do `DataFrame`:

```
housing.dropna(subset=["total_bedrooms"])      # opção 1
housing.drop("total_bedrooms", axis=1)          # opção 2
median = housing["total_bedrooms"].median()    # opção 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Se você escolher a opção 3, deve calcular o valor médio no conjunto de treinamento e usá-lo para preencher os valores faltantes neste, mas não se esqueça de também salvar o valor médio que você calculou. Você precisará dele mais tarde para substituir os valores faltantes no conjunto de testes quando quiser avaliar seu sistema e também quando o sistema entrar em operação para substituir os valores faltantes nos novos dados.

O Scikit-Learn fornece uma classe acessível para cuidar dos valores faltantes: `Imputer`. Veja como utilizá-la. Primeiro você cria uma instância do `Imputer`, especificando que deseja substituir os valores faltantes de cada atributo pela média desse atributo:

```
from sklearn.preprocessing import Imputer  
  
imputer = Imputer(strategy="median")
```

Uma vez que a média só pode ser calculada em atributos numéricos, precisamos criar uma cópia dos dados sem o atributo de texto `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Agora, você pode ajustar a instância `imputer` aos dados de treinamento utilizando o método `fit()`:

```
imputer.fit(housing_num)
```

O `imputer` simplesmente calculou a média de cada atributo e armazenou o resultado em sua variável da instância `statistics_`. Somente o atributo `total_bedrooms` tinha valores faltantes, mas não podemos ter certeza de que não haverá valores faltantes nos novos dados após o sistema entrar em operação por isso é mais seguro aplicar o `imputer` a todos os atributos numéricos:

```
>>> imputer.statistics_  
array([-118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])  
>>> housing_num.median().values  
array([-118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
```

Agora, você pode utilizar esse `imputer` “treinado” substituindo os valores perdidos pelas médias aprendidas a fim de transformar o conjunto de treinamento:

```
X = imputer.transform(housing_num)
```

O resultado será um array Numpy simples que contém as características transformadas. Se quiser colocá-lo de volta em um `DataFrame` Pandas, é simples:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

Scikit-Learn Design

A API do Scikit-Learn é muito bem projetada. Os principais princípios de design (<http://goo.gl/wL10sI>) são:¹⁵

- **Consistência.** Todos os objetos compartilham uma interface simples e consistente:
 - *Estimadores.* Qualquer objeto que possa estimar alguns parâmetros com base em um conjunto de dados é chamado de *estimador* (por exemplo, um `Imputer` é um estimador). A estimativa em si é realizada pelo método `fit()`, e é necessário apenas um conjunto de dados como parâmetro (ou dois para algoritmos de aprendizado supervisionado; o segundo conjunto de dados contém os rótulos). Qualquer outro parâmetro necessário para orientar o processo de estimativa é considerado um hiperparâmetro (como `imputer's strategy`), e deve ser configurado como uma variável da instância (geralmente por meio de um parâmetro do construtor).
 - *Transformadores.* Alguns estimadores (como um `Imputer`) também podem transformar um conjunto de dados; esses são chamados *transformadores*. Mais uma vez, a API é bem simples: a transformação é realizada pelo método `transform()` para transformar o conjunto de dados como um parâmetro. Ele retorna o conjunto de dados transformado. Esta transformação geralmente depende dos parâmetros aprendidos, como é o caso de um `Imputer`. Todos os transformadores também possuem um método de conveniência chamado `fit_transform()` que é o equivalente a chamar a `fit()` e então `transform()` (mas algumas vezes `fit_transform()` é otimizado e roda muito mais rápido).
 - *Previsores.* Finalmente, a partir de um conjunto de dados, alguns estimadores são capazes de fazer previsões; eles são chamados *previsores*. Por exemplo, o modelo `LinearRegression` do capítulo anterior foi um previsor: previu a satisfação de vida, dado o PIB per capita de um país. Um previsor tem um método `predict()` que pega um conjunto de dados de novas instâncias e retorna um conjunto de dados de previsões correspondentes. Ele também tem um método `score()`, que mede a qualidade das previsões, dado um conjunto de teste (e os rótulos correspondentes no caso de algoritmos de aprendizado supervisionado).¹⁶
- **Inspeção.** Todos os hiperparâmetros do estimador são diretamente acessíveis por meio de variáveis das instâncias públicas (por exemplo, `imputer.strategy`), e todos os parâmetros aprendidos do estimador também são acessíveis por variáveis das instâncias públicas com um sufixo de sublinhado (por exemplo, `imputer.statistics_`).

15 Para mais detalhes em princípios de design, consulte “API design for machine learning software: experiences from the scikit-learn project”, L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Müller, et al. (2013).

16 Alguns previsores também fornecem métodos para medir a confiança de suas previsões.

- **Não proliferação de classes.** Os conjuntos de dados são representados como arrays NumPy ou matrizes esparsas SciPy em vez de classes caseiras. Os hiperparâmetros são apenas strings ou números Python.
- **Composição.** Os blocos de construção existentes são reutilizados tanto quanto possível. Por exemplo, é fácil criar um estimador Pipeline seguido por um estimador final a partir de uma sequência arbitrária de transformadores, como veremos mais à frente.
- **Padrões sensíveis.** O Scikit-Learn fornece valores padrão razoáveis para a maioria dos parâmetros, facilitando a criação rápida de um sistema padronizado de trabalho.

Manipulando Texto e Atributos Categóricos

Anteriormente, excluímos o atributo categórico `ocean_proximity` por ser um atributo de texto, portanto não podemos calcular sua média:

```
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat.head(10)
17606      <1H OCEAN
18632      <1H OCEAN
14650      NEAR OCEAN
3230        INLAND
3555      <1H OCEAN
19480      INLAND
8879      <1H OCEAN
13685      INLAND
4937      <1H OCEAN
4861      <1H OCEAN
Name: ocean_proximity, dtype: object
```

De qualquer modo, a maioria dos algoritmos de Aprendizado de Máquina prefere trabalhar com números, então vamos converter essas categorias de texto para números. Para tanto, podemos utilizar o método `factorize()` do Pandas, que mapeia cada categoria para um número inteiro diferente:

```
>>> housing_cat_encoded, housing_categories = housing_cat.factorize()
>>> housing_cat_encoded[:10]
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])
```

Este é melhor: `housing_cat_encoded` agora é puramente numérico. O método `factorize()` também retorna a lista de categorias (“<1H OCEAN” foi mapeado para 0, “NEAR OCEAN” foi mapeado para 1, etc.):

```
>>> housing_categories
Index(['<1H OCEAN', 'NEAR OCEAN', 'INLAND', 'NEAR BAY', 'ISLAND'], dtype='object')
```

Um problema nesta representação é que os algoritmos de Aprendizado de Máquina assumirão que dois valores próximos são mais parecidos do que dois valores distantes. Obviamente, este não é o caso (por exemplo, as categorias 0 e 4 são mais semelhantes do que as categorias 0 e 2). Para corrigir este problema, uma solução comum seria a criação de um atributo binário por categoria: um atributo igual a 1 quando a categoria for “<1H OCEAN” (e 0 caso contrário), outro atributo igual a 1 quando a categoria for “NEAR OCEAN” (e 0 caso contrário), e assim por diante. Isso é chamado de *one-hot encoding*, porque apenas um atributo será igual a 1 (*hot*), enquanto os outros serão 0 (*cold*).

O Scikit-Learn fornece um codificador `OneHotEncoder` para converter valores categóricos inteiros em vetores *one-hot*. Vamos programar as categorias como vetores *one-hot*:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>
      with 16512 stored elements in Compressed Sparse Row format>
```

Note que `fit_transform()` espera um *array* 2D, mas `housing_cat_encoded` é um *array* 1D, então precisamos remodelá-lo.¹⁷ Além disso, observe que a saída é uma *matriz esparsa* SciPy, não um *array* NumPy. Isso é muito útil quando você possui atributos categóricos com milhares de categorias. Após um *one-hot encoding*, obtemos uma matriz com milhares de colunas e cheia de zeros, exceto por um único 1 por linha. Seria muito desperdício utilizar toneladas de memória para armazenar principalmente zeros, então, em vez disso, uma matriz esparsa armazena apenas a localização dos elementos diferentes de zero. Você pode utilizá-la principalmente como um *array* 2D normal,¹⁸ mas, se realmente quiser convertê-la em um *array* (denso) NumPy, basta chamar o método `toarray()`:

```
>>> housing_cat_1hot.toarray()
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Você pode aplicar ambas as transformações de uma única vez (de categorias de texto a categorias de inteiros, depois de categorias de inteiros para vetores *one-hot*) utilizando a classe `CategoricalEncoder`. Ela não faz parte do Scikit-Learn 0.19.0 e mais antigos, mas

¹⁷ A função NumPy `reshape()` permite que uma dimensão seja -1, o que significa “não especificado”: o valor é inferido do comprimento do array e das dimensões restantes.

¹⁸ Veja a documentação do SciPy para mais detalhes.

será adicionada em breve, então talvez já esteja disponível no momento em que você lê este livro. Se não estiver, pode obtê-la no notebook Jupyter para este capítulo (o código foi copiado do Pull Request #9151). Segue abaixo como usá-la:

```
>>> from sklearn.preprocessing import CategoricalEncoder # ou pegue do notebook
>>> cat_encoder = CategoricalEncoder()
>>> housing_cat_reshaped = housing_cat.values.reshape(-1, 1)
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>
    with 16512 stored elements in Compressed Sparse Row format>
```

Por padrão, o `CategoricalEncoder` mostra uma matriz esparsa, mas você pode configurar a codificação para “onehot-dense” se preferir uma matriz densa:

```
>>> cat_encoder = CategoricalEncoder(encoding="onehot-dense")
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
>>> housing_cat_1hot
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Você pode obter a lista de categorias utilizando a variável de instância `categories_`. É uma lista que contém um array de 1D de categorias para cada atributo categórico (neste caso, uma lista contendo um único array, uma vez que existe apenas um atributo categórico):

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)]
```



Se um atributo categórico tiver um grande número de categorias possíveis (por exemplo, código do país, profissão, espécies, etc.), então uma codificação *one-hot* resultará em um grande número de características de entrada. Isso pode diminuir o treinamento e degradar o desempenho. Se isso acontecer, será necessário produzir representações mais densas chamadas *embeddings*, mas isso requer uma boa compreensão das redes neurais (veja o Capítulo 14 para mais detalhes).

Customize Transformadores

Embora o Scikit-Learn forneça muitos transformadores úteis, você precisará escrever seus próprios para tarefas como operações de limpeza personalizadas ou combinar atributos específicos. É preciso que o seu transformador funcione perfeitamente

com as funcionalidades do Scikit-Learn (como os *pipelines*) e, como o Scikit-Learn depende da tipagem *duck typing* (não herança), você só precisa criar uma classe e implementar os três métodos: `fit()` (retornando `self`), `transform()` e `fit_transform()`. Você pode obter o último de graça ao simplesmente acrescentar `TransformerMixin` em uma classe base. Além disso, se você adicionar `BaseEstimator` como uma classe base (e evitar `*args` e `**kargs` em seu construtor) você receberá dois métodos extras (`get_params()` e `set_params()`) que serão úteis para o ajuste automático dos hiperparâmetros. Por exemplo, esta é uma pequena classe `transformer` que adiciona os atributos combinados discutidos anteriormente:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # sem *args ou **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                       bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Neste exemplo, o transformador tem um hiperparâmetro, `add_bedrooms_per_room`, definido por padrão como `True` (geralmente é útil fornecer padrões sensíveis). Este hiperparâmetro permitirá que você descubra facilmente se a adição deste atributo ajuda ou não os algoritmos de Aprendizado de Máquina. De forma mais geral, você pode adicionar um hiperparâmetro para controlar qualquer etapa da preparação de dados sobre a qual você não tem 100% de certeza. Quanto mais você automatizar essas etapas de preparação de dados, mais combinações poderá experimentar automaticamente, tornando muito mais provável encontrar uma ótima combinação (e economizar muito tempo).

Escalonamento das Características

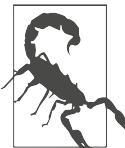
Uma das transformações mais importantes que você precisa aplicar aos seus dados é o *escalonamento das características*. Com poucas exceções, os algoritmos de Aprendizado de Máquina não funcionam bem quando atributos numéricos de entrada têm escalas muito diferentes. Este é o caso dos dados do setor imobiliário: o número total de cômodos

varia de 6 a 39.320, enquanto os rendimentos médios variam apenas de 0 a 15. Observe que geralmente não é necessário escalar os valores-alvo.

Existem duas maneiras comuns de todos os atributos obterem a mesma escala: *escala min-max e padronização*.

O escalonamento min-max (muitas pessoas chamam de *normalização*) é bastante simples: os valores são deslocados e redimensionados para que acabem variando de 0 a 1. Fazemos isso subtraindo o valor mínimo e dividindo pelo máximo menos o mínimo. O Scikit-Learn fornece um transformador chamado `MinMaxScaler` para isso. Ele possui um hiperparâmetro `feature_range` que permite alterar o intervalo se você não quiser 0-1 por algum motivo.

A padronização é bem diferente: em primeiro lugar ela subtrai o valor médio (assim os valores padronizados sempre têm média zero) e, em seguida, divide pela variância, de modo que a distribuição resultante tenha variância unitária. Ao contrário do escalonamento min-max, a padronização não vincula valores a um intervalo específico, o que pode ser um problema para alguns algoritmos (por exemplo, as redes neurais geralmente esperam um valor de entrada variando de 0 a 1). No entanto, a padronização é muito menos afetada por outliers. Por exemplo, suponha que um bairro tenha uma renda média igual a 100 (por engano). O escalonamento min-max, em seguida, comprimiria todos os outros valores de 0-15 para 0-0,15, enquanto a padronização não seria muito afetada. O Scikit-Learn fornece um transformador para padronização chamado `StandardScaler`.



Tal como acontece com todas as transformações, é importante encaixar os *escalonadores* apenas nos dados de treinamento e não no conjunto completo de dados (incluindo o conjunto de testes). Só então você pode utilizá-los para transformar o conjunto de treinamento e o conjunto de teste (e novos dados).

Pipelines de Transformação

Como você pode ver, existem muitas etapas de transformação de dados que precisam ser executadas na ordem correta. Felizmente, o Scikit-Learn fornece a classe `Pipeline` para ajudar com tais sequências de transformações. Eis um pequeno pipeline para os atributos numéricos:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

O construtor `Pipeline` se vale de uma lista de pares de nome/estimador que definem uma sequência de etapas. Todos, exceto o último estimador, devem ser transformadores (ou seja, eles devem ter um método `fit_transform()`). Os nomes podem ser o que você quiser (desde que não contenham sublinhados duplos “`__`”).

Quando você chama o método `fit()` do pipeline, ele chama `fit_transform()` sequencialmente em todos os transformadores, passando a saída de cada chamada como parâmetro para a próxima chamada, até chegar ao estimador final, o qual chama apenas o método `fit()`.

O pipeline expõe os mesmos métodos que o estimador final. Neste exemplo, o último estimador é um `StandardScaler`, que é um transformador, então o pipeline possui um método `transform()` que aplica todas as transformações aos dados em sequência (também possui um método `fit_transform` que poderíamos ter usado em vez de chamar a `fit()` e depois `transform()`).

Agora, seria bom se pudéssemos fornecer diretamente em nosso pipeline um `DataFrame` Pandas que contivesse colunas não numéricas em vez de termos que primeiro extrair manualmente as colunas numéricas em um array NumPy. Não há nada no Scikit-Learn que lide com os `DataFrames` Pandas,¹⁹ mas podemos escrever um transformador personalizado para esta tarefa:

```
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Nosso `DataFrameSelector` transformará os dados selecionando os atributos desejados, descartando o resto e convertendo o `DataFrame` resultante em um array NumPy. Com isso, você pode facilmente escrever um pipeline que terá um `DataFrame` Pandas e lidar apenas com os valores numéricos: o pipeline iniciaria apenas com um `DataFrameSelector` para escolher os atributos numéricos, seguido dos outros passos de pré-processamento que discutimos anteriormente. E você também pode escrever outro pipeline com facilidade para os atributos categóricos simplesmente ao selecioná-los utilizando um `DataFrameSelector` e depois aplicando um `CategoricalEncoder`.

¹⁹ Confira também Pull Request #3886, que deve introduzir uma classe `ColumnTransformer`, que facilitará transformações específicas de atributos. Você também pode testar `pip3 install sklearn-pandas` para obter uma classe `DataFrameMapper` com um objetivo semelhante.

```

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

```

Mas como é possível juntar esses dois pipelines em um único? A resposta é utilizar a classe `FeatureUnion` do Scikit-Learn. Você lhe dá uma lista de transformadores (que podem ser pipelines transformadores inteiros); quando o método `transform()` é chamado, ele executa cada método `transform()` em paralelo, aguarda sua saída e, em seguida, os concatena e retorna o resultado (e, claro, chamar o método `fit()` chama cada método `fit()` do transformador). Um pipeline completo que manipula ambos atributos numéricos e categóricos pode ser mais ou menos assim:

```

from sklearn.pipeline import FeatureUnion

full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])

```

E você pode executar todo o pipeline simplesmente:

```

>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ -1.15604281,   0.77194962,   0.74333089, ...,  0.        ,
         0.        ,   0.        ],
       [-1.17602483,   0.6596948 ,  -1.1653172 , ...,  0.        ,
         0.        ,   0.        ],
       [...]
      ])
>>> housing_prepared.shape
(16512, 16)

```

Selecionar e Treinar um Modelo

Finalmente! Você enquadrou o problema, obteve os dados e os explorou, selecionou um conjunto de treinamento e um conjunto de testes e escreveu canais de transformação para limpar e preparar automaticamente seus dados para os algoritmos de Aprendizado de Máquina. Agora você está pronto para selecionar e treinar um modelo de Aprendizado de Máquina.

Treinando e Avaliando o Conjunto de Treinamento

A boa notícia é que, graças a todas essas etapas anteriores, as coisas agora serão muito mais simples do que você pensa. Treinaremos primeiro um modelo de Regressão Linear, como fizemos no capítulo anterior:

```
from sklearn.linear_model import LinearRegression  
  
lin_reg = LinearRegression()  
lin_reg.fit(housing_prepared, housing_labels)
```

Feito! Agora, você possui um modelo de Regressão Linear funcional. Vamos tentar isso em algumas instâncias do conjunto de treinamento:

```
>>> some_data = housing.iloc[:5]  
>>> some_labels = housing_labels.iloc[:5]  
>>> some_data_prepared = full_pipeline.transform(some_data)  
>>> print("Predictions:", lin_reg.predict(some_data_prepared))  
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]  
>>> print("Labels:", list(some_labels))  
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Funciona, embora as previsões não sejam exatamente precisas (por exemplo, a primeira previsão tem cerca de 40% de erro!). Vamos medir a RMSE desse modelo de regressão em todo o conjunto de treinamento com o uso da função `mean_squared_error` do Scikit-Learn:

```
>>> from sklearn.metrics import mean_squared_error  
>>> housing_predictions = lin_reg.predict(housing_prepared)  
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)  
>>> lin_rmse = np.sqrt(lin_mse)  
>>> lin_rmse  
68628.198198489219
```

Ok, isso é melhor do que nada, mas certamente não é uma grande pontuação: a maioria dos `median_housing_values` dos bairros varia entre US\$ 120 mil e US\$ 265 mil, então um erro de margem típico de US\$ 68.628 não é muito aceitável. Este é um exemplo de um modelo de subajuste dos dados de treinamento. Quando isso acontece, pode significar que as características não fornecem informações suficientes para fazer boas previsões ou que o modelo não é suficientemente poderoso. Como vimos no capítulo anterior, as principais formas de corrigir o *subajuste* são: selecionar um modelo mais poderoso, alimentar o algoritmo de treinamento com melhores características ou reduzir as restrições no modelo. Este modelo não é regularizado, o que exclui a última opção. Você poderia tentar adicionar mais características (por exemplo, o registro da população), mas primeiro vamos tentar um modelo mais complexo para ver como ele se sai.

Treinaremos um `DecisionTreeRegressor`. Este é um modelo poderoso, capaz de encontrar relações não lineares complexas nos dados (as Árvores de Decisão serão apresentadas com mais detalhes no Capítulo 6). O código deve ser familiar agora:

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor()  
tree_reg.fit(housing_prepared, housing_labels)
```

Agora que o modelo está treinado, vamos avaliá-lo no conjunto de treinamento:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)  
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)  
>>> tree_rmse = np.sqrt(tree_mse)  
>>> tree_rmse  
0.0
```

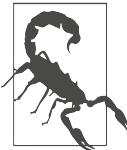
Espere, o quê?! Nenhum erro? Será que esse modelo é realmente absolutamente perfeito? Claro, é muito mais provável que o modelo tenha se sobreajustado mal aos dados. Como você pode ter certeza? Como vimos anteriormente, você não pode tocar no conjunto de testes até que esteja pronto para lançar um modelo confiável, então você precisa utilizar parte do conjunto de treinamento para treinar, e parte para validar o modelo.

Avaliando Melhor com a Utilização da Validação Cruzada

Uma maneira de avaliar o modelo da Árvore de Decisão seria utilizar a função `train_test_split` para dividir o conjunto de treinamento em um conjunto menor de treinamento e um conjunto de validação, em seguida treinar seus modelos com o conjunto menor e avaliá-los com o conjunto de validação. É um pouco trabalhoso, mas nada muito difícil e funciona muito bem.

Uma ótima alternativa é utilizar o recurso da *validação cruzada* do Scikit-Learn. O código a seguir executa a *validação cruzada K-fold*: ele divide aleatoriamente o conjunto de treinamento em 10 subconjuntos distintos chamados de partes (*folds*), então treina e avalia o modelo da Árvore de Decisão 10 vezes escolhendo uma parte (*fold*) diferente a cada uma delas para avaliação e treinando nas outras 9 partes. O resultado é um array contendo as 10 pontuações de avaliação:

```
from sklearn.model_selection import cross_val_score  
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,  
                         scoring="neg_mean_squared_error", cv=10)  
tree_rmse_scores = np.sqrt(-scores)
```



Os recursos da validação cruzada do Scikit-Learn esperam uma função de utilidade (mais alta é melhor) ao invés de uma função de custo (mais baixa é melhor), de modo que a função de pontuação é exatamente o oposto do MSE (ou seja, um valor negativo), e é por isso que o código anterior calcula `-scores` antes de calcular a raiz quadrada.

Vamos olhar os resultados:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 70232.0136482  66828.46839892  72444.08721003  70761.50186201
         71125.52697653  75581.29319857  70169.59286164  70055.37863456
         75370.49116773  71222.39081244]
Mean: 71379.0744771
Standard deviation: 2458.31882043
```

Agora a Árvore de Decisão não tem uma aparência tão boa quanto antes. Na verdade, parece ser pior do que o modelo de Regressão Linear! Observe que a validação cruzada permite que você obtenha não apenas uma estimativa do desempenho do seu modelo, mas também uma medida da precisão dessa estimativa (ou seja, seu desvio padrão). A Árvore de Decisão possui uma pontuação de aproximadamente 71.379, geralmente \pm 2.458. Você não teria essa informação se utilizasse apenas um conjunto de validação. Mas a validação cruzada treina o modelo várias vezes, então nem sempre é possível.

Calcularemos as mesmas pontuações para o modelo de Regressão Linear apenas para ter certeza:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                                 scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 66782.73843989  66960.118071    70347.95244419  74739.57052552
         68031.13388938  71193.84183426  64969.63056405  68281.61137997
         71552.91566558  67665.10082067]
Mean: 69052.4613635
Standard deviation: 2731.6740018
```

É isso mesmo: o modelo da Árvore de Decisão está se sobreajustando tanto que acaba sendo pior do que o modelo de Regressão Linear.

Vamos tentar um último modelo agora: `RandomForestRegressor`. Como veremos no Capítulo 7, Florestas Aleatórias funcionam com o treinamento de muitas Árvores de Decisão em subconjuntos aleatórios das características, e em seguida calculam a média de suas

previsões. Construir um modelo em cima de muitos outros modelos é chamado *Ensemble Learning*, e muitas vezes é uma ótima maneira de aumentar ainda mais os algoritmos de Aprendizado de Máquina. Ignoraremos a maior parte do código, pois é essencialmente o mesmo para os outros modelos:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
21941.911027380233
>>> display_scores(forest_rmse_scores)
Scores: [ 51650.94405471  48920.80645498  52979.16096752  54412.74042021
         50861.29381163  56488.55699727  51866.90120786  49752.24599537
         55399.50713191  53309.74548294]
Mean: 52564.1902524
Standard deviation: 2301.87380392
```

Isso é muito melhor: Florestas Aleatórias parecem muito promissoras. No entanto, note que a pontuação no conjunto de treinamento ainda é muito menor do que nos conjuntos de validação, o que significa que o modelo ainda está se sobreajustando ao conjunto de treinamento. Possíveis soluções para sobreajustes são: simplificar o modelo, restringi-lo (ou seja, regularizá-lo), ou obter muito mais dados de treinamento. No entanto, antes de mergulhar mais profundamente em Florestas Aleatórias, você deve experimentar muitos outros modelos de várias categorias de algoritmos de Aprendizado de Máquina (várias Máquinas de Vetores de Suporte com diferentes *kernels*, possivelmente uma rede neural, etc.), sem gastar muito tempo ajustando os hiperparâmetros. O objetivo é selecionar alguns modelos promissores (de dois a cinco).



Você deve salvar todos os modelos que experimenta para que possa voltar facilmente para qualquer um deles. Certifique-se de salvar os hiperparâmetros e os parâmetros treinados, bem como as pontuações de validação cruzada e talvez as previsões também. Isso permitirá que você compare facilmente as pontuações entre tipos de modelo e compare os tipos de erros que eles cometem. Você pode facilmente salvar modelos do Scikit-Learn utilizando o módulo `pickle` do Python ou usando o `sklearn.externals.joblib`, que é mais eficiente na serialização de grandes arrays NumPy:

```
from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

Ajuste Seu Modelo

Assumiremos que você tem uma lista restrita de modelos promissores. Agora, você precisa ajustá-los. Vejamos algumas maneiras de fazer isso.

Grid Search

Uma maneira de fazer isso seria alterar manualmente os hiperparâmetros até encontrar uma ótima combinação de valores. Este seria um trabalho muito tedioso, e talvez você não tenha tempo para explorar muitas combinações.

Em vez disso, acione o `GridSearchCV` do Scikit-Learn para efetuar a busca por você. Você só precisa passar quais hiperparâmetros deseja que ele experimente e quais valores tentar, e ele avaliará todas as combinações de valores possíveis por meio de validação cruzada. Por exemplo, o código a seguir busca a melhor combinação de valores dos hiperparâmetros para `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```



Uma abordagem simples seria testar potências de 10 consecutivas quando você não tem ideia do valor que um hiperparâmetro deve ter (ou um número menor se você quiser uma busca mais refinada, como mostrado neste exemplo com o hiperparâmetro `n_estimators`).

Este `param_grid` pede ao Scikit-Learn que primeiro avalie todas as $3 \times 4 = 12$ combinações de valores dos hiperparâmetros `n_estimators` e `max_features` especificados na primeira `dict` (não se preocupe com o que esses hiperparâmetros significam por enquanto, eles serão explicados no Capítulo 7), então tente todas as $2 \times 3 = 6$ combinações de valores do hiperparâmetro na segunda `dict`, mas desta vez com o hiperparâmetro de inicialização definido como `False` em vez de `True` (que é o valor padrão para este hiperparâmetro).

De modo geral, a grid search explorará $12 + 6 = 18$ combinações de valores do hiperparâmetro `RandomForestRegressor` e treinará cada modelo cinco vezes (já que estamos

utilizando validação cruzada de cinco partes). Em outras palavras, no geral, haverá $18 \times 5 = 90$ rodadas de treinamento! Pode demorar muito, mas quando terminar você obterá a melhor combinação de parâmetros:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```



Uma vez que 8 e 30 são os valores máximos avaliados, você provavelmente deve tentar pesquisar outra vez com valores mais elevados, já que a pontuação pode continuar a melhorar.

Você também pode obter diretamente o melhor estimador:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features=8, 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=30, n_jobs=1, oob_score=False, random_state=42,
                      verbose=0, warm_start=False)
```



Se `GridSearchCV` for inicializado com `refit=True` (que é o padrão), assim que ele encontrar o melhor estimador utilizando a validação cruzada, ele o treinará novamente em todo o conjunto. Esta geralmente é uma boa ideia, pois fornecer mais dados melhora potencialmente seu desempenho.

E, claro, as pontuações de avaliação também estão disponíveis:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63647.854446 {'n_estimators': 3, 'max_features': 2}
55611.5015988 {'n_estimators': 10, 'max_features': 2}
53370.0640736 {'n_estimators': 30, 'max_features': 2}
60959.1388585 {'n_estimators': 3, 'max_features': 4}
52740.5841667 {'n_estimators': 10, 'max_features': 4}
50374.1421461 {'n_estimators': 30, 'max_features': 4}
58661.2866462 {'n_estimators': 3, 'max_features': 6}
52009.9739798 {'n_estimators': 10, 'max_features': 6}
50154.1177737 {'n_estimators': 30, 'max_features': 6}
57865.3616801 {'n_estimators': 3, 'max_features': 8}
51730.0755087 {'n_estimators': 10, 'max_features': 8}
49694.8514333 {'n_estimators': 30, 'max_features': 8}
62874.4073931 {'n_estimators': 3, 'bootstrap': False, 'max_features': 2}
54643.4998083 {'n_estimators': 10, 'bootstrap': False, 'max_features': 2}
59437.8922859 {'n_estimators': 3, 'bootstrap': False, 'max_features': 3}
52735.3582936 {'n_estimators': 10, 'bootstrap': False, 'max_features': 3}
57490.0168279 {'n_estimators': 3, 'bootstrap': False, 'max_features': 4}
51008.2615672 {'n_estimators': 10, 'bootstrap': False, 'max_features': 4}
```

Neste exemplo, obtemos a melhor solução definindo o hiperparâmetro `max_features` em 8 e o hiperparâmetro `n_estimators` em 30. A pontuação da RMSE para esta combinação é 49.694, o que é um pouco melhor do que o resultado obtido anteriormente utilizando os valores padrão do hiperparâmetro (que era 52.564). Parabéns, você ajustou com sucesso o seu melhor modelo!



Não se esqueça de que alguns dos passos da preparação de dados podem ser tratados como hiperparâmetros. Por exemplo, a grid search descobrirá automaticamente se deve ou não adicionar uma característica que você não tinha certeza (por exemplo, utilizando o hiperparâmetro `add_bedrooms_per_room` do seu transformador `CombinedAttributesAdder`). Similarmente, ele também pode ser usado para encontrar automaticamente a melhor maneira de lidar com os outliers, características perdidas, seleção das características e muito mais.

Randomized Search

Se estiver explorando relativamente poucas combinações, a abordagem da grid search é boa, como no exemplo anterior, mas quando o *espaço de busca* do hiperparâmetro for grande, é preferível utilizar `RandomizedSearchCV` em seu lugar. Esta classe pode ser utilizada da mesma maneira que a classe `GridSearchCV`, mas, em vez de tentar todas as combinações possíveis, ela seleciona um valor aleatório para cada hiperparâmetro em cada iteração e avalia um determinado número de combinações aleatórias. Esta abordagem tem dois benefícios principais:

- Se você deixar que a pesquisa randomizada execute, por exemplo, mil iterações, essa abordagem explorará mil valores diferentes para cada hiperparâmetro (em vez de apenas alguns valores por hiperparâmetro na abordagem da grid search);
- Estabelecendo o número de iterações, você terá mais controle na pesquisa dos hiperparâmetros sobre o orçamento que deseja alocar.

Métodos de Ensemble

Outra maneira de ajustar seu sistema é tentar combinar os modelos de melhor desempenho. O grupo (ou “ensemble”) geralmente será superior ao melhor modelo individual (assim como as Florestas Aleatórias funcionam melhor do que as Árvores de Decisão individuais em que se baseiam), especialmente se os modelos individuais tiverem tipos muito diferentes de erros. Abordaremos este tópico com mais detalhes no Capítulo 7.

Analise os Melhores Modelos e Seus Erros

Muitas vezes você obterá boas ideias sobre o problema ao inspecionar os melhores modelos. Por exemplo, o `RandomForestRegressor` pode indicar a importância relativa de cada atributo para fazer previsões precisas:

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.33442355e-02,   6.29090705e-02,   4.11437985e-02,
       1.46726854e-02,   1.41064835e-02,   1.48742809e-02,
       1.42575993e-02,   3.66158981e-01,   5.64191792e-02,
      1.08792957e-01,   5.33510773e-02,   1.03114883e-02,
      1.64780994e-01,   6.02803867e-05,   1.96041560e-03,
      2.85647464e-03])
```

Mostraremos essas pontuações de importância ao lado de seus nomes de atributos correspondentes:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = cat_pipeline.named_steps["cat_encoder"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.0028564746373201579, 'NEAR OCEAN'),
 (0.0019604155994780701, 'NEAR BAY'),
 (6.0280386727365991e-05, 'ISLAND')]
```

Com esta informação, você pode tentar descartar algumas das características menos úteis (por exemplo, aparentemente apenas uma categoria `ocean_proximity` é realmente útil, então você pode tentar descartar as outras).

Você também deve analisar os erros específicos cometidos pelo seu sistema, depois tentar entender por que ele os faz e o que poderia solucionar o problema (adicionar características extras ou, ao contrário, se livrar das não informativas, limpar outliers, etc.).

Avalie Seu Sistema no Conjunto de Testes

Depois de ajustar seus modelos por um tempo você terá um sistema que funciona suficientemente bem. Agora é a hora de avaliar o modelo final no conjunto de teste. Não há nada de especial neste processo; apenas obtenha as previsões e os rótulos do seu conjunto de teste, execute `full_pipeline` para transformar os dados (chame `transform()`, *não* `fit_transform()!`) e avalie o modelo final no conjunto de teste:

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
```

O desempenho geralmente será um pouco pior do que o medido usando validação cruzada se você fez muitos ajustes de hiperparâmetro (porque o sistema acabou sendo ajustado para executar bem com dados de validação e provavelmente não funcionará tão bem em conjuntos desconhecidos de dados). Não é o caso neste exemplo, mas, quando isso acontece, você deve resistir à tentação de ajustar os hiperparâmetros para que os números fiquem mais atraentes no conjunto de teste; as melhorias não seriam generalizadas para novos dados.

Agora, vem a fase de pré-lançamento do projeto: você precisa apresentar sua solução (destacando o que aprendeu, o que funcionou e o que não, quais pressupostos foram feitos e quais as limitações do seu sistema); documente tudo e crie apresentações detalhadas com visualizações claras e declarações que sejam fáceis de lembrar (por exemplo, “a renda média é o principal previsor dos preços do setor imobiliário”).

Lance, Monitore e Mantenha seu Sistema

Perfeito, você obteve aprovação para o lançamento! Você precisa preparar sua solução para a produção, principalmente conectando as fontes de dados de entrada da produção ao seu sistema e escrevendo testes.

Você também precisa escrever o código de monitoramento para verificar o desempenho ao vivo em intervalos regulares do seu sistema e acionar alertas quando ele ficar offline. Isso é importante, não apenas para capturar uma quebra súbita, mas também a degradação do desempenho. Isso é bastante comum porque os modelos tendem a “deteriorar”

à medida que os dados evoluem ao longo do tempo, a menos que sejam regularmente treinados em novos dados.

A avaliação do desempenho do seu sistema exigirá uma amostragem das previsões do sistema e sua avaliação. Isso geralmente requer uma análise humana. Esses analistas podem ser especialistas ou trabalhadores de uma plataforma de *crowdsourcing* (como a Amazon Mechanical Turk ou CrowdFlower). De qualquer forma, você precisará conectar o canal de avaliação humana ao seu sistema.

Você também deve certificar-se de avaliar a qualidade de dados de entrada do sistema. Às vezes, o desempenho se deteriorará levemente por causa de um sinal de má qualidade (por exemplo, uma falha do sensor que envia valores aleatórios ou o resultado de outra equipe que se torna obsoleto), mas pode demorar até que o desempenho se degrade o suficiente para disparar um alerta. Você pode capturar isso antecipadamente se monitorar suas entradas. O monitoramento das entradas é particularmente importante para os sistemas de *aprendizado online*.

Finalmente, você deve treinar regularmente seus modelos com a utilização de novos dados. Esse processo deve ser automatizado tanto quanto possível. Se não for, a probabilidade é que você apenas o atualize a cada seis meses (na melhor das hipóteses), e o seu desempenho pode variar bastante ao longo do tempo. Se for um sistema de *aprendizado online* certifique-se de salvar *capturas de tela* do seu estado atual em intervalos regulares para que seja possível reverter facilmente para um estágio anterior do trabalho.

Experimente!

Esperamos que este capítulo tenha dado uma boa ideia do que é um projeto de Aprendizado de Máquina e tenha mostrado algumas das ferramentas que você utilizará para treinar um grande sistema. Como você pode ver, grande parte do trabalho está na etapa de preparação dos dados, na construção de ferramentas de monitoramento, na criação de canais de avaliação humana e na automação do treinamento regular de um modelo. Os algoritmos de Aprendizado de Máquina também são importantes, é claro, mas é preferível ficar confortável com o processo geral e ter um bom conhecimento de três ou quatro algoritmos do que gastar todo o seu tempo explorando algoritmos avançados e não ter tempo suficiente para o processo como um todo.

Então, se você ainda não o fez, agora é um bom momento para pegar um notebook, selecionar um conjunto de dados do seu interesse e tentar passar por todo o processo de A a Z. Um bom lugar para começar seria em um website de competição como o <http://kaggle.com/>: você terá um conjunto de dados para utilização, um objetivo claro e pessoas com quem compartilhar a experiência.

Exercícios

Utilizando o conjunto de dados do setor imobiliário deste capítulo:

1. Experimente um regressor da Máquina de Vetores de Suporte (`sklearn.svm.SVR`), com vários hiperparâmetros, como `kernel="linear"` (com vários valores para o hiperparâmetro `C`) ou `kernel="rbf"` (com vários valores para os hiperparâmetros `C` e `gamma`). Não se preocupe com o significado desses hiperparâmetros por enquanto. Qual será o desempenho do melhor previsor da SVR?
2. Tente substituir `GridSearchCV` por `RandomizedSearchCV`.
3. Tente acrescentar um transformador no pipeline de preparação para selecionar apenas os atributos mais importantes.
4. Tente criar um pipeline único que faça a preparação completa de dados mais a previsão final.
5. Explore automaticamente algumas opções de preparação utilizando o `GridSearchCV`.

As soluções para estes exercícios estão disponíveis online nos notebooks Jupyter em <https://github.com/ageron/handson-ml>.

Capítulo 3

Classificação

No Capítulo 1, mencionamos que regressão (previsão de valores) e classificação (previsão de classes) são as tarefas de aprendizado supervisionado mais comuns. No Capítulo 2, utilizando diversos algoritmos como a Regressão Linear, Árvores de Decisão e Florestas Aleatórias (que será explicado em detalhes em capítulos posteriores), exploramos uma tarefa de regressão para prever preços do mercado imobiliário. Agora, voltaremos nossa atenção para os sistemas de classificação.

MNIST

Neste capítulo, utilizaremos o conjunto de dados MNIST, composto de 70 mil pequenas imagens de dígitos escritos à mão por estudantes do ensino médio e funcionários do *US Census Bureau*. Cada imagem é rotulada com o dígito que a representa. Este conjunto tem sido tão estudado que, muitas vezes, é chamado de o "Hello World" do Aprendizado de Máquina: sempre que as pessoas apresentam um novo algoritmo de classificação, elas têm curiosidade em ver como será seu desempenho no MNIST. Sempre que alguém estuda o Aprendizado de Máquina, mais cedo ou mais tarde lidará com o MNIST.

O Scikit-Learn fornece muitas funções auxiliares para baixar conjuntos de dados populares. O MNIST é um deles. O código a seguir se vale do conjunto de dados MNIST:¹

```
>>> from sklearn.datasets import fetch_mldata  
>>> mnist = fetch_mldata('MNIST original')  
>>> mnist  
{'COL_NAMES': ['label', 'data'],  
 'DESCR': 'mldata.org dataset: mnist-original',  
 'data': array([[0, 0, 0, ..., 0, 0, 0],  
 [0, 0, 0, ..., 0, 0, 0],
```

¹ Por padrão, o Scikit-Learn armazena conjuntos de dados baixados em um diretório chamado \$HOME/scikit_learn_data.