Data Science do Zero

Joel Grus

ALTA BOOKS
E D I T O R A
Rio de Janeiro, 2016

Sumário

Prefácio ..... **xi**

1. Introdução **1**
A Ascensão dos Dados 1
O Que É Data Science? 1
Motivação Hipotética: DataSciencester 2
Encontrando Conectores-Chave 3
Cientistas de Dados Que Você Talvez Conheça 6
Salários e Experiência 8
Contas Pagas 11
Tópicos de Interesse 11
Em Diante 13

2. Curso Relâmpago de Python **15**
O Básico 15
Iniciando em Python 15
Python Zen 16
Formatação de Espaço em Branco 16
Módulos 17
Aritmética 18
Funções 18
Strings (cadeias de caracteres) 19
Exceções 19
Listas 20
Tuplas 21
Dicionários 21
Conjuntos 24
Controle de Fluxo 25

Veracidade 25
Não Tão Básico 26
Ordenação 27
Compreensões de Lista 27
Geradores e Iteradores 28
Aleatoriedade 29
Expressões Regulares 30
Programação Orientada a Objeto 30
Ferramentas Funcionais 31
Enumeração (enumerate) 32
Descompactação de Zip e Argumentos 33
args e kwargs 34
Bem-vindo à DataSciencester! 35
Para Mais Esclarecimentos 35

3. Visualizando Dados 37
matplotlib 37
Gráficos de Barra 39
Gráficos de Linhas 43
Gráficos de Dispersão 44
Para Mais Esclarecimentos 47

4. Álgebra Linear 49
Vetores 49
Matrizes 53
Para Mais Esclarecimentos 55

5. Estatística 57
Descrevendo um Conjunto Único de Dados 57
Tendências Centrais 59
Dispersão 61
Correlação 62
Paradoxo de Simpson 65
Alguns Outros Pontos de Atenção sobre Correlação 66
Correlação e Causalidade 67
Para Mais Esclarecimentos 68

6. Probabilidade 69
Dependência e Independência 69
Probabilidade Condicional 70
Teorema de Bayes 72
Variáveis Aleatórias 73

Distribuições Contínuas 74
A Distribuição Normal 75
O Teorema do Limite Central 78
Para Mais Esclarecimentos 80

7. Hipótese e Inferência 81
Teste Estatístico de Hipótese 81
Exemplo: Lançar Uma Moeda 81
p-values 84
Intervalos de Confiança 85
P-Hacking 86
Exemplo: Executando um Teste A/B 87
Inferência Bayesiana 88
Para Mais Esclarecimentos 92

8. Gradiente Descendente 93
A Ideia Por Trás do Gradiente Descendente 93
Estimando o Gradiente 94
Usando o Gradiente 97
Escolhendo o Tamanho do Próximo Passo 97
Juntando Tudo 98
Gradiente Descendente Estocástico 99
Para Mais Esclarecimentos 100

9. Obtendo Dados 103
stdin e stdout 103
Lendo Arquivos 105
O Básico de Arquivos Texto 105
Arquivos delimitados 106
Extraindo Dados da Internet 108
HTML e Sua Subsequente Pesquisa 108
Exemplo: Livros O'Reilly Sobre Dados 110
Usando APIs 114
JSON (e XML) 114
Usando Uma API Não Autenticada 115
Encontrando APIs 116
Exemplo: Usando as APIs do Twitter 117
Obtendo Credenciais 117
Para Mais Esclarecimentos 120

10. Trabalhando com Dados 121
Explorando Seus Dados 121
Explorando Dados Unidimensionais 121

Duas Dimensões 123
Muitas Dimensões 125
Limpando e Transformando 127
Manipulando Dados 129
Redimensionando 132
Redução da Dimensionalidade 134
Para Mais Esclarecimentos 139

11. Aprendizado de Máquina 141
Modelagem 141
O Que É Aprendizado de Máquina? 142
Sobreajuste e Sub-Ajuste 142
Precisão 145
Compromisso entre Polarização e Variância 147
Recursos Extração e Seleção de Característica 148
Para Mais Esclarecimentos 150

12. K–Vizinhos Mais Próximos 151
O Modelo 151
Exemplo: Linguagens Favoritas 153
A Maldição da Dimensionalidade 156
Para Mais Esclarecimentos 163

13. Naive Bayes 165
Um Filtro de Spam Muito Estúpido 165
Um Filtro de Spam Mais Sofisticado 166
Implementação 168
Testando Nosso Modelo 169
Para Mais Esclarecimentos 172

14. Regressão Linear Simples 173
O Modelo 173
Usando o Gradiente Descendente 176
Estimativa Máxima da Probabilidade 177
Para Mais Esclarecimentos 177

15. Regressão Múltipla 179
O Modelo 179
Mais Suposições do Modelo dos Mínimos Quadrados 180
Ajustando o Modelo 181
Interpretando o Modelo 182
O Benefício do Ajuste 183

Digressão: A Inicialização 183
Erros Padrões de Coeficientes de Regressão 184
Regularização 186
Para Mais Esclarecimentos 188

16. Regressão Logística 189
O Problema 189
A Função Logística 192
Aplicando o Modelo 194
O Benefício do Ajuste 195
Máquina de Vetor de Suporte 196
Para Mais Esclarecimentos 200

17. Árvores de Decisão 201
O Que É uma Árvore de Decisão? 201
Entropia 203
A Entropia de uma Partição 205
Criando uma Árvore de Decisão 206
Juntando Tudo 208
Florestas Aleatórias 211
Para Maiores Esclarecimentos 212

18. Redes Neurais 213
Perceptrons 213
Redes Neurais Feed-Forward 215
Backpropagation 218
Exemplo: Derrotando um CAPTCHA 219
Para Mais Esclarecimentos 224

19. Agrupamento 225
A Ideia 225
O Modelo 226
Exemplo: Encontros 227
Escolhendo k 230
Exemplo: Agrupando Cores 231
Agrupamento Hierárquico Bottom-up 233
Para Mais Esclarecimentos 238

20. Processamento de Linguagem Natural 239
Nuvens de Palavras 239
Modelos n-gramas 241
Gramáticas 244

Um Adendo: Amostragem de Gibbs 246
Modelagem de Tópicos 247
Para Mais Esclarecimentos 253

21. Análise de Rede 255
Centralidade de Intermediação 255
Centralidade de Vetor Próprio 260
Multiplicação de Matrizes 260
Centralidade 262
Gráficos Direcionados e PageRank 264
Para Mais Esclarecimentos 266

22. Sistemas Recomendadores 267
Curadoria Manual 268
Recomendando O Que é Popular 268
Filtragem Colaborativa Baseada no Usuário 269
Filtragem Colaborativa Baseada em Itens 272
Para Mais Esclarecimentos 274

23. Bases de Dados e SQL 275
CREATE TABLE e INSERT 275
UPDATE 277
DELETE 278
SELECT 278
GROUP BY 280
ORDER BY 282
JOIN 283
Subconsultas 285
Índices 285
Otimização de Consulta 286
NoSQL 287
Para Mais Esclarecimentos 287

24. MapReduce 289
Exemplo: Contagem de Palavras 289
Por que MapReduce? 291
MapReduce Mais Generalizado 292
Exemplo: Analisando Atualizações de Status 293
Exemplo: Multiplicação de Matriz 294
Um Adendo: Combinadores 296
Para Mais Esclarecimentos 296

25. Vá em Frente e Pratique Data Science 299
IPython 299
Matemática 300
Não Do Zero 300
NumPy 301
pandas 301
scikit-learn 301
Visualização 301
R 302
Encontre Dados 302
Pratique Data Science 303
Hacker News 303
Carros de Bombeiros 303
Camisetas 304
E Você? 304
26. Índice 305

Prefácio

Data Science

Data science tem sido chamada de "o emprego mais sexy do Século 21" *(http://bit.ly/1Bqe-1WY)*, provavelmente por alguém que nunca tenha visitado um quartel do corpo de bombeiros. De qualquer forma, data science é um campo em evidência e está em alta; não requer muita investigação para encontrar prognósticos de analistas de que, nos próximos dez anos, precisaremos de bilhões e bilhões de cientistas de dados a mais do que possuímos atualmente.

Mas o que é data science? Afinal de contas, não conseguimos produzir cientistas de dados se não soubermos o que realmente é. De acordo com o diagrama de Venn *(http://bit.ly/1EQN-Z4A)*, um tanto famoso nesta área, data science se encontra na interseção de:

- Habilidades de hacker
- Conhecimento de estatística e matemática
- Competência significativa

Originalmente, planejei escrever um livro abordando os três, mas eu rapidamente percebi que uma abordagem completa de "competência significativa" exigiria dezenas de milhares de páginas. Assim, eu decidi focar nos dois primeiros. Meu objetivo é ajudá-lo a desenvolver habilidades de hacker, as quais você precisará para iniciar a prática em data science. Meu outro objetivo é fazer você se sentir confortável com matemática e estatística, que são a base de data science.

De alguma forma, este livro é uma grande ambição. A melhor maneira de aprender a hackear é hackeando coisas. Ao ler este livro, você terá um bom entendimento de como eu hackeio as coisas, que talvez não seja a melhor forma para você. Você entenderá quais ferramentas eu uso que talvez não sejam as melhores para você. Você verá como eu abordo os problemas com dados, que talvez não seja a melhor abordagem para você. A intenção (e a

esperança) é que meus exemplos inspirarão você a experimentar as coisas do seu jeito. Todo o código e dados deste livro estão disponíveis no GitHub *(https://github.com/joelgrus/data-science-from-scratch)* para ajudar.

Do mesmo modo, a melhor maneira de aprender matemática é praticando. Na verdade, este não é um livro de matemática e, na maior parte, nós não "praticaremos matemática". No entanto, você não pode praticar data science sem ter *algum* entendimento de probabilidade, estatística e álgebra linear. Isso significa que, quando necessário, vamos a fundo nas equações matemáticas, intuições matemáticas, axiomas matemáticos e versões cartunescas de grandes ideias matemáticas. Espero que você não tenha medo de ir fundo comigo.

Durante todo o livro, também espero que você veja que brincar com dados é divertido, porque, bem, brincar com dados é divertido! 'Especialmente se comparado a algumas alternativas, como declaração de impostos ou exploração de carvão.'

Do Zero

Existem várias e várias bibliotecas, estruturas, módulos e kits de ferramentas de data science que implementam de modo eficiente os mais comuns (e também os menos comuns) algoritmos e técnicas. Se você se tornar um cientista de dados, será íntimo de NumPy, de scikit-learn, de pandas e de diversas outras bibliotecas. Elas são ótimas para praticar data science e também ótimas para começar a praticar sem entender de fato o que é data science.

Neste livro, abordaremos data science do zero. Isso significa que construiremos ferramentas e implementaremos algoritmos à mão, a fim de entendê-los melhor. Eu me empenhei bastante em criar implementações e exemplos que são claros, bem comentados e legíveis. Na maioria dos casos, as ferramentas que construiremos serão esclarecedoras, mas pouco práticas. Elas funcionarão bem em pequenos conjuntos de dados, mas fracassarão nas escalas encontradas na web.

No decorrer do livro, eu indicarei bibliotecas que você talvez use para aplicar tais técnicas para aumentar os conjuntos de dados. Porém, não as usaremos aqui.

Há um sólido debate sobre qual a melhor linguagem para aprender data science. Muitos acreditam que é a linguagem de programação estatística R. (Achamos que essas pessoas estão *erradas*.) Poucos sugerem Java ou Scala. Contudo, Python é a escolha evidente.

Python possui diversos recursos que o tornam mais adequado para o aprendizado (e prática) de data science:

- É gratuito.
- É relativamente simples de codificar (e, o principal, de entender).
- Possui muitas bibliotecas úteis relacionadas ao data science.

Fico receoso ao dizer que Python é minha linguagem de programação favorita. Há outras linguagens que considero mais agradáveis, mais bem projetadas, ou apenas mais divertidas de trabalhar. E, ainda assim, toda vez que eu começo um projeto novo de data science, eu acabo usando Python. Toda vez que preciso fazer um protótipo rápido que funcione, eu acabo usando Python. E toda vez que quero demonstrar conceitos precisos de data science, de maneira fácil de entender, acabo usando Python. Desta forma, o livro usa Python.

O objetivo deste livro não é ensinar Python. (Apesar de ser bem óbvio que, ao ler este livro, você aprenderá um pouco de Python.) Irei levá-lo em um curso intensivo pelo capítulo que destaca os recursos mais importantes para os nossos propósitos, mas se você não sabe nada sobre programar em Python (ou sobre programação no geral), talvez você queira turbinar este livro com algo como um tutorial "Python para Iniciantes".

O restante desta introdução ao data science terá a mesma abordagem — entrando em detalhes quando parecer essencial ou esclarecedor, outras vezes deixando os detalhes para você descobrir por si só (ou procurar na Wikipédia).

Ao longo dos anos, treinei um grande número de cientistas de dados. Apesar de que nem todos eles seguiram o caminho de se tornarem cientistas de dados ninjas rockstars, os deixei melhores do que quando os encontrei. Vim a acreditar que qualquer pessoa que tenha alguma aptidão para a matemática e alguma habilidade para programação tem o que é necessário para praticar data science. Tudo o que precisa é de uma mente curiosa, vontade trabalhar bastante e este livro. Portanto, este livro.

Convenções Usadas Neste Livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

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

Constant width

Usada para listagens de programas, e, também, dentro do texto se referindo aos elementos dos programas como variáveis ou nomes de funções, bancos de dados, tipos de dados, variáveis de ambiente, declarações e palavras-chave.

Constant width bold

Mostra comandos ou outro texto que deve ser literalmente digitado pelo usuário.

Constant width italic

Mostra texto que deve ser substituído com valores fornecidos pelo usuário ou por valores determinados pelo contexto.

Este ícone significa uma dica ou sugestão.

Este ícone significa uma observação geral.

Este ícone significa um aviso ou precaução.

Usando exemplos de código

O material complementar (exemplos de código, exercícios, etc.) está disponível para download em *https://github.com/joelgrus/data-science-from-scratch*. Todos os sites mencionados nesta obra estão em inglês e a editora não se responsabiliza pela manutenção ou conteúdo de sites de terceiros.

Este livro está aqui para ajudar a realizar o trabalho. De modo geral, se o exemplo de código é oferecido com ele, você pode usá-lo em seus programas e documentações. Você não precisa nos contatar para permissão a menos que você esteja reproduzindo uma porção significativa do código. Por exemplo, escrever um programa que usa vários pedações do código deste livro não precisa de permissão. Vender ou distribuir um CD-ROM com os exemplos dos livros da O'Reilly precisa de permissão. Responder a uma pergunta citando este livro ou um exemplo não precisa de permissão. Incorporar uma quantidade significativa de exemplos de código deste livro na documentação do seu produto precisa de permissão.

Agradecimentos

Primeiramente, eu gostaria de agradecer a Mike Loukides por aceitar minha proposta para este livro (e por insistir que eu o diminuísse para um tamanho razoável). Seria muito mais fácil para ele ter dito "Quem é esta pessoa que vive me enviando e-mails com amostras de capítulos e como faço para ela ir embora?" Fico agradecido por ele não ter feito isso. Também gostaria de agradecer a minha editora, Marie Beaugureau, por me guiar pelo processo de publicação e ter um livro em um estado muito melhor do que eu teria se estivesse sozinho.

Eu não poderia ter escrito este livro se eu nunca tivesse aprendido data science e, provavelmente, não teria aprendido se não fosse pela influência de Dave Hsu, Igor Tatarinov, John Rauser e o restante da turma Farecast. (Há tanto tempo que nem se usava o nome data science!) A galera legal da Coursera também merece bastante crédito.

Também sou muito agradecido pelos meus leitores e revisores. Jay Fundling encontrou toneladas de erros e apontou muitas explicações que não estavam claras, e o livro está muito melhor (e muito mais correto) graças a ele. Debashis Ghosh é um herói por checar todas as minhas estatísticas. Andrew Musselman sugeriu diminuir os aspectos do tipo "pessoas que preferem R ao Python são moralmente responsáveis" do livro, que acabei achando um ótimo conselho. Trey Causey, Ryan Matthew Balfanz, Loris Mularoni, Núria Pujol, Rob Jefferson, Mary Pat Campbell, Zach Geary e Wendy Grus também forneceram valiosas opiniões. Quaisquer erros remanescentes são de minha exclusiva responsabilidade.

Devo muito à comunidade do Twitter #datascience, por me expor a toneladas de novos conceitos, me apresentar para muitas pessoas e parar de me sentir um fracassado, tanto que eu me superei e escrevi um livro para compensar. Um agradecimento especial para Trey Causey (novamente) por (inadvertidamente) me lembrar de incluir um capítulo sobre álgebra linear, e para Scan J. Taylor por (inadvertidamente) apontar algumas lacunas no capítulo "Trabalhando com Dados".

Acima de tudo, eu devo um imenso agradecimento para Ganga e Madeline. A única coisa mais difícil do que escrever um livro é morar com alguém que esteja escrevendo um. Eu não teria conseguido sem o apoio deles.

CAPÍTULO 1

Introdução

"Dados! Dados! Dados!" ele gritou impacientemente. "Não posso fabricar tijolos sem barro."
—Arthur Conan Doyle

A Ascensão dos Dados

Vivemos em um mundo que está soterrado por dados. Os websites rastreiam todos os cliques de todos os usuários. Seu smartphone está fazendo um registro da sua localização e sua velocidade a cada segundo diariamente. Atletas avaliados usam pedômetros com esteroides que estão sempre registrando suas batidas do coração, hábitos de movimentos, dieta e padrões do sono. Carros inteligentes coletam hábitos de direção, casas inteligentes coletam hábitos de moradia e marqueteiros inteligentes coletam hábitos de compra. A própria internet representa um diagrama grande de conhecimento que contém (entre outras coisas) uma enorme enciclopédia de referências cruzadas: bases de dados específicos de domínio sobre filmes, música, resultados de esportes, máquinas de pinball, memes e coquetéis; e muitas estatísticas do governo (algumas delas são verdades!) sobre tantos governos que causariam um nó na sua cabeça.

Soterrados sob esses dados estão as respostas para as inúmeras questões que ninguém nunca pensou em perguntar. Neste livro, aprenderemos como encontrá-las.

O Que É Data Science?

Há uma piada que diz que um cientista de dados é alguém que sabe mais sobre estatística do que um cientista da computação e mais sobre ciência da computação do que um estatístico (eu não disse que a piada era boa). Na verdade, alguns cientistas de dados são — para todos os propósitos práticos — estatísticos, enquanto outros são quase indistinguíveis dos engenheiros de software. Alguns são experts em aprendizado de máquina, enquanto outros não conseguiram aprender muita coisa sobre o assunto. Alguns são PhDs com um impressio-

nante registro de publicações, enquanto outros nunca leram um trabalho acadêmico (apesar de ser uma vergonha). Resumindo, basicamente não importa como você define data science, pois você encontrará praticantes para quem a definição está total e absolutamente errada.

De qualquer forma, não permitiremos que isso nos impeça de tentar. Digamos que um cientista de dados seja alguém que extrai conhecimento de dados desorganizados. O mundo de hoje está cheio de pessoas tentando transformar dados em conhecimento.

Por exemplo, o site de namoro OkCupid pede que seus membros respondam milhares de perguntas a fim de encontrar as combinações mais adequadas para eles. Mas também analisa tais resultados para descobrir perguntas aparentemente inócuas as quais você poderia perguntar para alguém e descobrir qual a possibilidade de essa pessoa dormir com você no primeiro encontro (*http://bit.ly/1EQU0hI*).

O Facebook pede que você adicione sua cidade natal e sua localização atual, supostamente para facilitar que seus amigos o encontrem e se conectem com você. Porém, ele também analisa essas localizações para identificar padrões de migração global (*http://on.fb.me/1EQTq3A*) e onde vivem os fã-clubes dos times de futebol (*http://on.fb.me/1EQTvnO*).

Como uma grande empresa, a Target rastreia suas encomendas e interações, tanto online como na loja física. Ela usa os dados em um modelo preditivo (*http://nyti.ms/1EQTznL*) para saber quais clientes estão grávidas a fim de melhorar sua oferta de artigos relacionados a bebês.

Em 2012, a campanha do Obama empregou muitos cientistas de dados que mineraram os dados e experimentaram uma forma de identificar os eleitores que precisavam de uma atenção extra, otimizar programas e recursos para a captação de fundos de doadores específicos e focando esforços para votos onde provavelmente eles teriam sido úteis. Normalmente, é de comum acordo pensar que esses esforços tiveram um papel importante na reeleição do presidente, o que significa que é seguro apostar que as campanhas políticas do futuro se tornarão cada vez mais dependentes de dados, resultando em uma corrida armamentista sem fim de data science e coleta de dados.

Agora, antes que você se sinta muito exausto: alguns cientistas de dados também usam suas habilidades para o bem, ocasionalmente — usar os dados para tornar o governo mais eficiente (*http://bit.ly/1EQTGiW*), ajudar os desabrigados (*http://bit.ly/1EQTIYl*), e melhorar a saúde pública (http://bit.ly/1EQTPTv). Mas, certamente, não afetará sua carreira se você gosta de encontrar a melhor maneira de fazer o público clicar em seus anúncios.

Motivação Hipotética: DataSciencester

Parabéns! Você acabou de ser contratado para liderar os esforços de data science na DataSciencester, *a* rede social para cientistas de dados.

Apesar de ser *para* os cientistas de dados, a DataSciencester nunca investiu em construir sua própria atividade de data science (na verdade, a DataSciencester nunca investiu em construir seu próprio produto). Esse será seu trabalho! No decorrer do livro, aprenderemos sobre os conceitos de data science ao resolver problemas com os quais você se depara no traba-

lho. Algumas vezes, olharemos para os dados explicitamente fornecidos pelo usuário, outras vezes olharemos para os gerados por suas interações com um site e, às vezes, olharemos para os dados dos experimentos que projetaremos.

E, devido à DataSciencester possuir uma forte mentalidade de "não-foi-inventado-aqui", nós construiremos nossas próprias ferramentas do zero. No final, você terá um sólido entendimento dos fundamentos de data science. Você estará pronto para aplicar suas habilidades em sua empresa com uma premissa menos duvidosa, ou em qualquer outro problema que vier a despertar seu interesse.

Bem-vindo a bordo e boa sorte! 'Você pode usar jeans às sextas e o toalete é no final do corredor à direita.'

Encontrando Conectores-Chave

É seu primeiro dia de trabalho na DataSciencester e o vice-presidente de Rede (networking) está cheio de perguntas sobre seus usuários. Até agora, ele não teve ninguém para perguntar, então ele está muito empolgado em ter você aqui.

Particularmente, ele quer que você identifique quem são os "conectores-chave" entre os cientistas de dados. Para isso, ele lhe dá uma parte de toda a rede da DataSciencester. Na vida real, você geralmente não recebe os dados de que precisa. O Capítulo 9 é voltado para a obtenção de dados.

Com o que se parece essa parte dos dados? Ela consiste em uma lista de usuários, cada um representado por um `dict` que contém um `id` (um número) para cada usuário ou usuária e um `name` (que por uma das grandes coincidências cósmicas que rima com o `id` do usuário):

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

Ele também fornece dados "amigáveis", representados por uma lista de pares de IDs:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Por exemplo, a tupla `(0,1)` indica que o cientista de dados com a `id` 0 (Hero) e o cientista de dados com a `id` 1 (Dunn) são amigos. A rede é ilustrada na Figura 1-1.

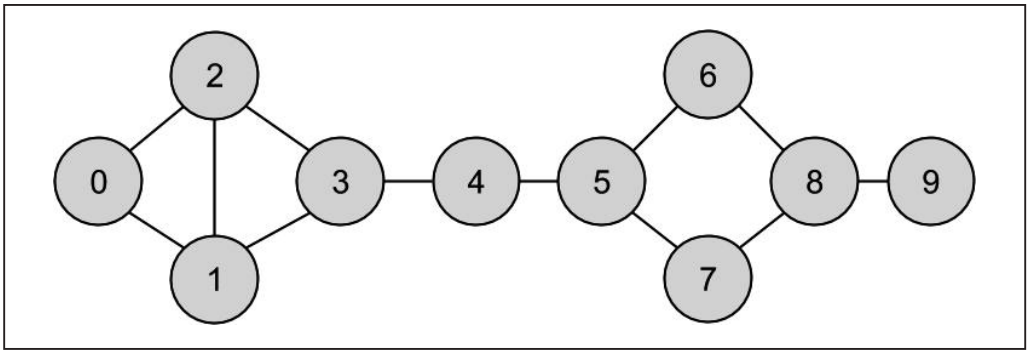

Figura 1-1. A rede da DataSciencester

Já que representamos nossos usuários como `dict`s, é fácil de aumentá-los com dados extras.

Não fique preso aos detalhes do código agora. No Capítulo 2, vamos levá-lo a um curso intensivo de Python. Por enquanto, tente pegar o sentido geral do que estamos fazendo.

Por exemplo, talvez nós queiramos adicionar uma lista de amigos para cada usuário. Primeiro nós configuramos a propriedade `friends` de cada usuário em uma lista vazia:

```
for user in users:
    user["friends"] = []
```

Então, nós povoamos a lista com os dados de `friendships`:

```
for i, j in friendships:
    # isso funciona porque users[i] é o usuário cuja id é i
    users[i]["friends"].append(users[j]) # adiciona i como um amigo de j
    users[j]["friends"].append(users[i]) # adiciona j como um amigo de i
```

Uma vez que o `dict` de cada usuário contenha uma lista de amigos, podemos facilmente perguntar sobre nosso gráfico, como "qual é o número médio de conexões?"

Primeiro, encontramos um número *total* de conexões, resumindo os tamanhos de todas as listas de `friends`:

```
def number_of_friends(user):
    """quantos amigos o usuário tem?"""
    return len(user["friends"])                  # tamanho da lista friend_ids

total_connections = sum(number_of_friends(user)
                        for user in users)       # 24
```

Então, apenas dividimos pelo número de usuários:

```
from __future__ import division                    # divisão inteira está incompleta
num_users = len(users)                             # tamanho da lista de usuários
avg_connections = total_connections / num_users    # 2.4
```

Também é fácil de encontrar as pessoas mais conectadas — são as que possuem o maior número de amigos.

Como não há muitos usuários, podemos ordená-los de "muito amigos" para "menos amigos":

```
# cria uma lista (user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

sorted(num_friends_by_id,                                  # é ordenado
       key=lambda (user_id, num_friends): num_friends,     # por num_friends
       reverse=True)                                       # do maior para o menor

# cada par é (user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

Uma maneira de pensar sobre o que nós fizemos é uma maneira de identificar as pessoas que são, de alguma forma, centrais para a rede. Na verdade, o que acabamos de computar é uma rede métrica de *grau de centralidade* (Figura 1-2).

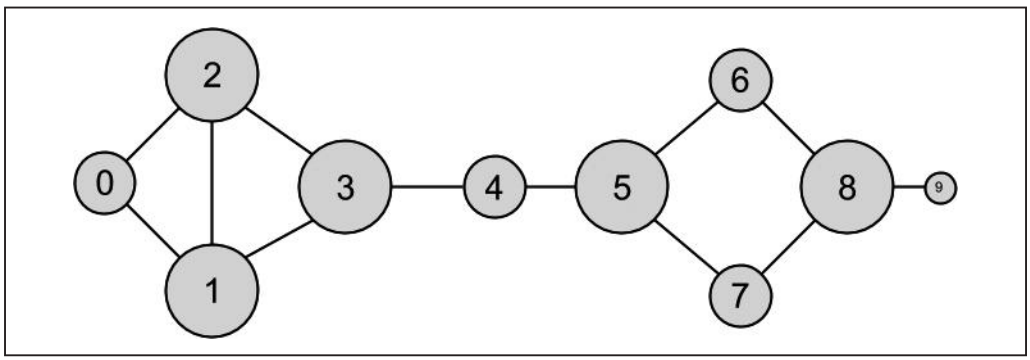

Figura 1-2. A rede DataSciencester ordenada pelo grau

Essa figura tem a vantagem de ser fácil de calcular, mas nem sempre lhe dá os resultados que você queria ou esperaria. Por exemplo, a rede Thor da DataSciencester (`id` 4) possui somente duas conexões enquanto que Dunn (`id` 1) possui três. Ainda olhando para a rede, parece que Thor deveria ser mais centralizado. No Capítulo 21, investigaremos as redes com mais detalhe, e veremos noções de centralidade mais complexas que podem ou não corresponder melhor à nossa intuição.

Cientistas de Dados Que Você Talvez Conheça

Enquanto você está preenchendo os papéis de admissão, a vice-presidente da Fraternidade chega a sua mesa. Ela quer estimular mais conexões entre os seus membros, e pede que você desenvolva sugestões de "Cientistas de Dados Que Você Talvez Conheça".

Seu primeiro instinto é sugerir um usuário que possa conhecer amigos de amigos. São fáceis de computar: para cada amigo de um usuário, itera sobre os amigos daquela pessoa, e coleta todos os resultados:

```
def friends_of_friend_ids_bad(user):
    # "foaf" é abreviação de "friend of a friend"
    return [foaf["id"]
            for friend in user["friends"]     # para cada amigo de usuário
            for foaf in friend["friends"]]    # pega cada _their_friends
```

Quando chamamos `users[0]` (Hero), ele produz:

```
[0, 2, 3, 0, 1, 3]
```

Isso inclui o usuário 0 (duas vezes), uma vez que Hero é, de fato, amigo de ambos os seus amigos. Inclui os usuários 1 e 2, apesar de eles já serem amigos do Hero. E inclui o usuário 3 duas vezes, já que Chi é alcançável por meio de dois amigos diferentes:

```
print [friend["id"] for friend in users[0]["friends"]]  # [1, 2]
print [friend["id"] for friend in users[1]["friends"]]  # [0, 2, 3]
print [friend["id"] for friend in users[2]["friends"]]  # [0, 1, 3]
```

Saber que as pessoas são amigas-de-amigas de diversas maneiras parece uma informação interessante, então talvez nós devêssemos produzir uma *contagem* de amigos em comum. Definitivamente, devemos usar uma função de ajuda para excluir as pessoas que já são conhecidas do usuário:

```
from collections import Counter                     # não carregado por padrão

def not_the_same(user, other_user):
    """dois usuários não são os mesmos se possuem ids diferentes"""
    return user["id"] != other_user["id"]

def not_friends(user, other_user):
    """other_user não é um amigo se não está em user["friends"];
    isso é, se é not_the_same com todas as pessoas em user["friends"]"""
    return all(not_the_same(friend, other_user)
               for friend in user["friends"])

def friends_of_friend_ids(user):
    return Counter(foaf["id"]
                   for friend in user["friends"]    # para cada um dos meus amigos
                   for foaf in friend["friends"]    # que contam *their* amigos
                   if not_the_same(user, foaf)      # que não sejam eu
                   and not_friends(user, foaf))     # e que não são meus amigos

print friends_of_friend_ids(users[3])               # Counter({0: 2, 5: 1})
```

Isso diz sobre Chi (`id` 3) que ela possui dois amigos em comum com Hero (`id` 0) mas somente um amigo em comum com Clive (`id` 5).

Como um cientista de dados, você sabe que você pode gostar de encontrar usuários com interesses similares (esse é um bom exemplo do aspecto "competência significativa" de data science). Depois de perguntar por aí, você consegue pôr as mãos nesse dado, como uma lista de pares `(user_id, interest)`:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

Por exemplo, Thor (`id` 4) não possui amigos em comum com Devin (`id` 7), mas compartilham do interesse em aprendizado de máquina.

É fácil construir uma função que encontre usuários com o mesmo interesse:

```
def data_scientists_who_like(target_interest):
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

Funciona, mas a lista inteira de interesses deve ser examinada para cada busca. Se tivermos muitos usuários e interesses (ou se quisermos fazer muitas buscas), seria melhor construir um índice de interesses para usuários:

```
from collections import defaultdict

# as chaves são interesses, os valores são listas de user_ids com interests
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

E outro de usuários para interesses:

```
# as chaves são user_ids, os valores são as listas de interests para aquele user_id
interests_by_user_id = defaultdict(list)
```

```
    for user_id, interest in interests:
        interests_by_user_id[user_id].append(interest)
```

Agora fica fácil descobrir quem possui os maiores interesses em comum com um certo usuário:

- Itera sobre os interesses do usuário.
- Para cada interesse, itera sobre os outros usuários com aquele interesse.
- Mantém a contagem de quantas vezes vemos cada outro usuário.

```
def most_common_interests_with(user):
    return Counter(interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"])
```

Poderíamos usar esse exemplo para construir um recurso mais rico de "Cientistas de Dados Que Você Deveria Conhecer" baseado em uma combinação de amigos e interesses em comum. Exploraremos esses tipos de aplicações no Capítulo 22.

Salários e Experiência

Na hora em que você está saindo para o almoço, o vice-presidente de Relações Públicas pergunta se você pode fornecer alguns fatos curiosos sobre quanto os cientistas de dados recebem. Dados de salário é, de fato, um tópico sensível, mas ele consegue fornecer um conjunto de dados anônimos contendo o `salary` (salário) de cada usuário (em dólares) e `tenure` (experiência) como um cientista de dados (em anos):

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

Naturalmente, o primeiro passo é traçar os dados (veremos como fazê-lo no Capítulo 3). Os resultados se encontram na Figura 1-3.

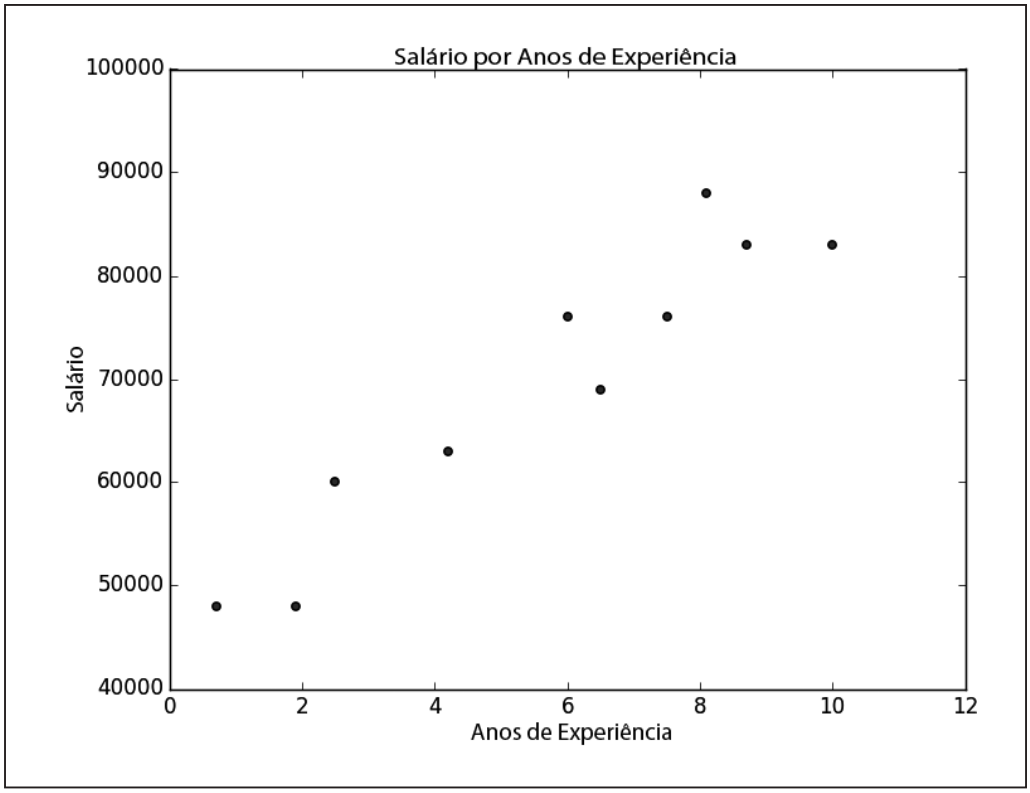

Figura 1-3. Salário por anos de experiência

Fica bem claro que os que possuem mais experiência tendem a receber mais. Como você pode transformar isso em um fato curioso? A primeira ideia é analisar a média salarial para cada ano:

```
# as chaves são os anos, os valores são as listas dos salários para cada ano
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# as chaves são os anos, cada valor é a média salarial para aquele ano
average_salary_by_tenure = {
    tenure : sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

Não é muito útil, já que nenhum dos usuários possui o mesmo caso, o que significa que estamos reportando apenas os salários individuais dos usuários:

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
```

```
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

Talvez fosse mais proveitoso agrupar os casos:

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

Então, o grupo junta os salários correspondentes para cada agrupamento:

```
# as chaves são agrupamentos dos casos, os valores são as listas
# dos salários para aquele agrupamento
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

E, finalmente, computar a média salarial para cada grupo:

```
# as chaves são agrupamentos dos casos, os valores são
# a média salarial para aquele agrupamento
average_salary_by_bucket = {
  tenure_bucket : sum(salaries) / len(salaries)
  for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
```

que é mais interessante:

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

E você tem um clichê: “os cientistas de dados com mais de cinco anos de experiência recebem 65% a mais do que os que possuem pouca ou nenhuma experiência!”

No entanto, nós escolhemos os casos de forma aleatória. O que realmente queríamos fazer era organizar um tipo de afirmação sobre o efeito do salário — em média — de ter um ano adicional de experiência. Além de tornar o fato mais intrigante, ainda permite que *façamos previsões* sobre salários que não conhecemos. Exploraremos mais essa ideia no Capítulo 14.

Contas Pagas

Ao voltar para a sua mesa, a vice-presidente da Receita está esperando por você. Ela quer entender melhor quais são os usuários que pagam por contas e quais que não pagam (ela sabe seus nomes, mas essa informação não é essencial).

Você percebe que parece haver uma correspondência entre os anos de experiência e as contas pagas:

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6   unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10  paid
```

Os usuários com poucos e muitos anos de experiência tendem a pagar; os usuários com uma quantidade mediana de experiência não.

Logo, se você quisesse criar um modelo — apesar de não haver dados o suficiente para servir de base para um — você talvez tentasse prever "paid" para os usuários com poucos e muitos anos de experiência, e "unpaid" para os usuários com quantidade mediana de experiência:

```
def predict_paid_or_unpaid(years_experience):
  if years_experience < 3.0:
    return "paid"
  elif years_experience < 8.5:
    return "unpaid"
  else:
    return "paid"
```

Certamente, nós definimos visualmente os cortes.

Com mais dados (e mais matemática), nós poderíamos construir um modelo prevendo a probabilidade de que um usuário pagaria, baseado em seus anos de experiência. Investigaremos esse tipo de problema no Capítulo 16.

Tópicos de Interesse

Quando seu dia está terminando, a vice-presidente da Estratégia de Conteúdo pede dados sobre em quais tópicos os usuários estão mais interessados, para que ela possa planejar o calendário do seu blog de acordo. Você já possui os dados brutos para o projeto sugerido:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
```

```
    (1, “NoSQL”), (1, “MongoDB”), (1, “Cassandra”), (1, “HBase”),
    (1, “Postgres”), (2, “Python”), (2, “scikit-learn”), (2, “scipy”),
    (2, “numpy”), (2, “statsmodels”), (2, “pandas”), (3, “R”), (3, “Python”),
    (3, “statistics”), (3, “regression”), (3, “probability”),
    (4, “machine learning”), (4, “regression”), (4, “decision trees”),
    (4, “libsvm”), (5, “Python”), (5, “R”), (5, “Java”), (5, “C++”),
    (5, “Haskell”), (5, “programming languages”), (6, “statistics”),
    (6, “probability”), (6, “mathematics”), (6, “theory”),
    (7, “machine learning”), (7, “scikit-learn”), (7, “Mahout”),
    (7, “neural networks”), (8, “neural networks”), (8, “deep learning”),
    (8, “Big Data”), (8, “artificial intelligence”), (9, “Hadoop”),
    (9, “Java”), (9, “MapReduce”), (9, “Big Data”)
]
```

Uma simples forma (e também fascinante) de encontrar os interesses mais populares é fazer uma simples contagem de palavras:

1. Coloque cada um em letras minúsculas (já que usuários diferentes podem ou não escrever seus interesses em letras maiúsculas).
2. Divida em palavras.
3. Conte os resultados.

No código:

```
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

Isso facilita listar as palavras que ocorrem mais de uma vez:

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print word, count
```

o que fornece o resultado esperado (a menos que você espere que “scikit-learn” possa ser dividido em duas palavras, o que não fornecerá o resultado esperado):

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
neural 2
```

```
scikit-learn 2
r 2
```

Veremos formas mais aprimoradas de extrair tópicos dos dados no Capítulo 20.

Em Diante

Foi um primeiro dia bem proveitoso! Exausto, você sai do prédio antes que alguém peça algo mais. Tenha uma boa noite de sono, porque amanhã será dia de treinamento para novos funcionários. Sim, você trabalhou um dia inteiro *antes* do treinamento. Culpa do RH.

CAPÍTULO 2

Curso Relâmpago de Python

As pessoas ainda são loucas pelo Python mesmo depois de vinte e cinco anos, o que é difícil de acreditar.
—Michael Palin

Todos os funcionários novos na DataSciencester são obrigados a passar pelo treinamento dos funcionários novos, a parte mais interessante é que contém um curso intensivo de Python.

Mas não é um tutorial compreensível sobre Python, é direcionado a destacar as partes da linguagem que serão mais importantes para nós (algumas delas, em geral, não são o foco dos tutoriais Python).

O Básico

Iniciando em Python

Você pode baixar o Python em python.org (*https://www.python.org/*). Mas se você ainda não o tiver, recomendo a instalação da distribuição Anaconda (*https://store.continuum.io/cshop/anaconda/*), que já contém a maioria das bibliotecas que você precisa para praticar data science.

No momento da escrita deste livro, a versão mais recente do Python é a 3.4. Porém, na DataSciencester, usamos o antigo e confiável Python 2.7. O Python 3 não é compatível com a versão anterior do Python 2 e muitas bibliotecas importantes somente funcionam bem com 2.7. A comunidade do data science ainda está presa ao 2.7, o que significa que nós estaremos também. Certifique-se de ter essa versão.

Se você não conseguir Anaconda, instale pip (*https://pypi.python.org/pypi/pip*), que é um gerenciador de pacote Python que permite que você facilmente instale pacotes de terceiros (precisaremos de alguns). Também vale a pena obter IPython (*http://ipython.org/*), o qual é um shell Python muito melhor de se trabalhar.

Se você instalou Anaconda, já deve vir com pip e IPython.

Apenas execute:

```
pip install ipython
```

e então procure na internet por soluções para quaisquer mensagens de erros enigmáticas que houver.

Python Zen

Python possui uma descrição Zen de seus princípios de design (*http://legacy.python.org/dev/peps/pep-0020/*), que você encontrar dentro do próprio interpretador Python ao digitar `import this`.

O mais discutido deles é:

> Deveria haver um — de preferência apenas um — modo óbvio de fazê-lo.

O código escrito de acordo com esse modo "óbvio"(que talvez não seja tão óbvio para um novato) frequentemente é descrito como "Pythonic". Apesar de este não ser um livro sobre Python, de vez em quando contrastaremos modos Pythonic e não-Pythonic de realizar os mesmos processos, e favoreceremos soluções Pythonic para os nossos problemas.

Formatação de Espaço em Branco

Muitas linguagens usam chaves para delimitar blocos de código. Python usa indentação:

```
for i in [1, 2, 3, 4, 5]:
    print i                     # primeira linha para o bloco "for i"
    for j in [1, 2, 3, 4, 5]:
        print j                 # primeira linha para o bloco "for j"
        print i + j             # última linha para o bloco "for j"
    print i                     # última linha para o bloco "for i"
print "done looping"
```

Isso faz com que o código Python seja bem legível, mas também significa que você tem que ser muito cuidadoso com a sua formatação. O espaço em branco é ignorado dentro dos parênteses e colchetes, o que poder ser muito útil em computações intermináveis:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

e para facilitar a leitura:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

Você também pode usar uma barra invertida para indicar que uma declaração continua na próxima linha, apesar de raramente fazermos isso:

```
two_plus_three = 2 + \
                 3
```

Uma consequência da formatação do espaço em branco é que pode ser difícil copiar e colar o código no Python shell. Por exemplo, se você tentasse colar este código:

```
for i in [1, 2, 3, 4, 5]:

    # note a linha em branco
    print i
```

na Python shell comum, você teria:

```
IndentationError: expected an indented block
```

porque o interpretador pensa que a linha em branco determina o final do bloco do loop for.

IPython tem a função mágica `%paste`, que copia corretamente o que quer que esteja na área de transferência, espaço em branco e tudo o mais. Apenas isso já é uma boa razão para usar IPython.

Módulos

Alguns recursos de Python não são carregados por padrão. Isto inclui tanto recursos como parte da linguagem assim como recursos de terceiros, que você baixa por conta própria. Para usar esses recursos, você precisará `import` (importar) os módulos que os contêm.

Uma abordagem é simplesmente importar o próprio módulo:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Aqui, `re` é o módulo que contém as funções e constantes para trabalhar com expressões regulares. Após esse tipo de `import`, você somente pode acessar tais funções usando o prefixo `re`...

Se você já tiver um `re` diferente no seu código você poderia usar um alias:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

Você talvez queira fazer isso se o seu módulo tem um nome complicado ou se você vai digitar bastante. Por exemplo, ao visualizar dados com `matplotlib`, uma convenção padrão é:

```
import matplotlib.pyplot as plt
```

Se você precisar de alguns valores específicos de um módulo, pode importá-los explicitamente e usá-los sem qualificação:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Se você fosse uma pessoa má, você poderia importar o conteúdo inteiro de um módulo dentro do seu conjunto de nomes, o que talvez pudesse sobrescrever variáveis que você já tinha definido:

```
match = 10
from re import *      # ih não, re tem uma função que combinação
print match           # "<function re.match>"
```

No entanto, já que você não é uma pessoa má, você nunca fará isso.

Aritmética

Python 2.7 usa a divisão de inteiros por padrão, portanto `5/2` é igual a `2`. Quase sempre isso não é o que queremos, então sempre começaremos nossos arquivos com:

```
from __future__ import division
```

depois disso, `5/2` é igual a `2.5`. Todo exemplo de código neste livro usa esse novo estilo de divisão. Na grande maioria dos casos em que precisaremos de divisão de inteiros, poderemos obtê-la com uma barra dupla `5 // 2`.

Funções

Uma função é uma regra para pegar zero e mais entradas e retornar uma saída correspondente. Em Python, definimos as funções usando `def`:

```
def double(x):
    """aqui é onde você coloca um docstring (cadeia de caracteres de documentação) opcional
    que explica o que a função faz.
    por exemplo, esta função multiplica sua entrada por 2"""
    return x * 2
```

As funções de Python são de *primeira classe*, que significa que podemos atribuí-las a variáveis e passá-las para as funções como quaisquer outros argumentos:

```
def apply_to_one(f):
    """chama a função f com 1 como seu argumento"""
    return f(1)

my_double = double              # refere-se à função definida anteriormente
x = apply_to_one(my_double)     # é igual a 2
```

Também é fácil criar pequenas funções anônimas, ou lambdas:

```
y = apply_to_one(lambda x: x + 4)      # é igual a 5
```

Você pode atribuir lambdas a variáveis, apesar de que maioria das pessoas lhe dirão para usar `def`:

```
another_double = lambda x: 2 * x       # não faça isso
def another_double(x): return 2 * x    # faça isso
```

Os parâmetros de função também podem receber argumentos padrões, que só precisam ser especificados quando você quiser um valor além do padrão:

```
def my_print(message="my default message"):
    print message

my_print("hello")   # exibe 'hello'
my_print()          # exibe 'my default message'
```

Às vezes é útil especificar argumentos pelo nome:

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # retorna 5
subtract(0, 5)  # retorna -5
subtract(b=5)   # mesmo que o anterior
```

Criaremos muitas, muitas funções.

Strings (cadeias de caracteres)

As strings podem ser delimitadas por aspas simples ou duplas (mas elas devem combinar):

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

O Python usa a barra invertida para codificar caracteres especiais. Por exemplo:

```
tab_string = "\t"       # representa o caractere tab
len(tab_string)         # é 1
```

Se você quiser barras invertidas como barras invertidas (que você vê nos nomes dos diretórios ou expressões regulares no Windows), você pode criar uma string *bruta* usando `r""`:

```
not_tab_string = r"\t"  # representa os caracteres '|' e 't'
len(not_tab_string)     # é 2
```

Você pode criar strings múltiplas usando aspas triplas ou duplas:

```
multi_line_string = """esta é a primeira linha.
e esta é a segunda
e esta é a terceira"""
```

Exceções

Quando algo dá errado, o Python exibe uma *exceção*. Se não for manipulada, o programa travará. Você pode manipulá-las usando `try` e `except`:

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

Apesar de serem consideradas ruins em muitas linguagens, as exceções são usadas livremente no Python para dar uma limpeza no código e, ocasionalmente, faremos o mesmo.

Listas

Provavelmente, a estrutura de dados mais básica em Python é a `list`. Uma lista é apenas uma coleção ordenada. (É parecida com o array das outras linguagens, mas com algumas funcionalidades a mais.)

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)      # é igual a 3
list_sum    = sum(integer_list)      # é igual a 6
```

Você pode ter ou configurar o elemento *n*-ésimo de uma lista com colchetes:

```
x = range(10)   # é a lista [0, 1, ..., 9]
zero = x[0]     # é igual a 0, as listas são indexadas a partir de 0
one = x[1]      # é igual a 1
nine = x[-1]    # é igual a 9, 'Pythonic' para o último elemento
eight = x[-2]   # é igual a 8, 'Pythonic' para o anterior ao último elemento
x[0] = -1       # agora x é [-1, 1, 2, 3, ..., 9]
```

Você também pode usar os colchetes para repartir as listas:

```
first_three    = x[:3]                  # [-1, 1, 2]
three_to_end = x[3:]                    # [3, 4, ..., 9]
one_to_four = x[1:5]                    # [1, 2, 3, 4]
last_three = x[-3:]                     # [7, 8, 9]
without_first_and_last = x[1:-1]        # [1, 2, ..., 8]
copy_of_x = x[:]                        # [-1, 1, 2, ..., 9]
```

O Python possui o operador `in` para verificar a associação à lista:

```
1 in [1, 2, 3]    # Verdadeiro
0 in [1, 2, 3]    # Falso
```

Essa verificação envolve examinar os elementos de uma lista um de cada vez, o que significa que você provavelmente não deveria usá-la a menos que você saiba que sua lista é pequena (ou a menos que você não se importe em quanto tempo a verificação durará).

É fácil concatenar as listas juntas:

```
x = [1, 2, 3]
x.extend([4, 5, 6])      # x agora é [1,2,3,4,5,6]
```

Se você não quiser modificar `x` você pode usar uma adição na lista:

```
x = [1, 2, 3]
y = x + [4, 5, 6]        # y é[1, 2, 3, 4, 5, 6]; x não mudou
```

Com mais frequência, anexaremos um item de cada vez nas listas:

```
x = [1, 2, 3]
x.append(0)        # x agora é [1, 2, 3, 0]
```

```
y = x[-1]        # é igual a 0
z = len(x)       # é igual a 4
```

Às vezes é conveniente *desfazer* as listas se você sabe quantos elementos elas possuem:

```
x, y = [1, 2]    # agora x é 1, y é 2
```

apesar de que você receberá um `ValueError` se não tiver os mesmos números de elementos dos dois lados.

É comum usar um sublinhado para um valor que você descartará:

```
_, y = [1, 2]    # agora y == 2, não se preocupou com o primeiro elemento
```

Tuplas

São as primas imutáveis das listas. Quase tudo que você pode fazer com uma lista, que não envolva modificá-la, é possível ser feito em uma tupla. Você especifica uma tupla ao usar parênteses (ou nada) em vez de colchetes:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3       # my_list agora é [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

As tuplas são uma maneira eficaz de retornar múltiplos valores a partir das funções:

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)     # é igual (5, 6)
s, p = sum_and_product(5, 10) # s é 15, p é 50
```

As tuplas (e listas) também podem ser usadas para *atribuições múltiplas*:

```
x, y = 1, 2      # agora x é 1, y é 2
x, y = y, x      # modo Pythonic de trocar as variáveis; agora x é 2, y é 1
```

Dicionários

Outra estrutura fundamental é um dicionário, que associa *valores* com *chaves* e permite que você recupere o valor correspondente de uma dada chave rapidamente:

```
empty_dict = {}                          # Pythonic
empty_dict2 = dict()                     # menos Pythonic
grades = { "Joel" : 80, "Tim" : 95 }     # dicionário literal
```

Você pode procurar o valor para uma chave usando colchetes:

```
joels_grade = grades["Joel"]                    # é igual a 80
```

Mas você receberá um `keyError` se perguntar por uma chave que não esteja no dicionário:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

Você pode verificar a existência de uma chave usando `in`:

```
joel_has_grade = "Joel" in grades       # Verdadeiro
kate_has_grade = "Kate" in grades       # Falso
```

Os dicionários possuem o método `get` que retorna um valor padrão (em vez de levantar uma exceção) quando você procura por uma chave que não esteja no dicionário:

```
joels_grade = grades.get("Joel", 0)   # é igual a 80
kates_grade = grades.get("Kate", 0)   # é igual a 0
no_ones_grade = grades.get("No One")  # padrão para padrão é None
```

Você atribui pares de valores-chave usando os mesmos colchetes:

```
grades["Tim"] = 99                    # substitui o valor antigo
grades["Kate"] = 100                  # adiciona uma terceira entrada
num_students = len(grades)            # é igual a 3
```

Frequentemente usaremos dicionários como uma simples maneira de representar dados estruturados:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Além de procurar por chaves específicas, podemos olhar para todas elas:

```
tweet_keys   = tweet.keys()     # lista de chaves
tweet_values = tweet.values()   # lista de valores-chave
tweet_items  = tweet.items()    # lista de (chave, valor) tuplas

"user" in tweet_keys            # Verdadeiro, mas usa list in, mais lento
"user" in tweet                 # mais Pythonic, usa dict in, mais rápido
"joelgrus" in tweet_values      # Verdadeiro
```

As chaves dos dicionários devem ser imutáveis; particularmente, você não pode usar `lists` como chaves. Se você precisar de uma chave multipart, você deveria usar uma `tuple` ou descobrir uma forma de transformar uma chave em uma string.

defaultdict

Imagine que você esteja tentando contar as palavras em um documento. Um método claro é criar um dicionário no qual as chaves são palavras e os valores são contagens. Conforme

você vai verificando cada palavra, você pode incrementar sua contagem se ela já estiver no dicionário e adicioná-la no dicionário se não estiver:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

Você também poderia usar o método "perdão é melhor do que permissão" e apenas manipular a exceção a partir da tentativa de procurar pela chave perdida:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

Uma terceira abordagem é usar `get`, que se comporta muito bem com as chaves perdidas:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Tudo isso é levemente complicado, por isso o `defaultdict` é útil. Um `defaultdict` é como um dicionário comum, exceto que, quando você tenta procurar por uma chave que ele não possui, ele primeiro adiciona um valor para ela usando a função de argumento zero que você forneceu ao criá-lo. Para usar `defaultdicts`, você tem que importá-los das `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produz 0
for word in document:
    word_counts[word] += 1
```

Eles também podem ser úteis com `list` ou `dict` ou até mesmo com suas próprias funções:

```
dd_list = defaultdict(list)             # list() produz uma lista vazia
dd_list[2].append(1)                    # agora dd_list contém {2: [1]}

dd_dict = defaultdict(dict)             # dict() produz um dict vazio
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # agora dd_pair contêm {2: [0,1]}
```

Isso será útil quando você usar dicionários para "coletar" resultados por alguma chave e não quiser verificar toda vez para ver se ela ainda existe.

Contador

Um `Counter` (contador) transforma uma sequência de valores em algo parecido com o objeto `defaultdict(int)` mapeando as chaves para as contagens. Primeiramente, o usaremos para criar histogramas:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c é (basicamente) { 0 : 2, 1 : 1, 2 : 1 }
```

Isso nos mostra uma forma simples de resolver nosso problema de `word_counts`:

```
word_counts = Counter(document)
```

Uma instância `Counter` possui um método `most_common` que é frequentemente útil:

```
# imprime as dez palavas mais comuns e suas contas
for word, count in word_counts.most_common(10):
    print word, count
```

Conjuntos

Outra estrutura de dados é o `set` (conjunto), que representa uma coleção de elementos *distintos*:

```
s = set()
s.add(1)       # s agora é { 1 }
s.add(2)       # s agora é { 1, 2 }
s.add(2)       # s ainda é { 1, 2 }
x = len(s)     # é igual a 2
y = 2 in s     # é igual a True
z = 3 in s     # é igual a False
```

Usaremos os conjuntos por duas razões principais. A primeira é que `in` é uma operação muito rápida em conjuntos. Se tivermos uma grande coleção de itens que queiramos usar para um teste de sociedade, um conjunto é mais adequado do que uma lista:

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list     # Falso, mas tem que verificar todos os elementos

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # muito rápido para verificar
```

A segunda razão é encontrar os itens *distintos* em uma coleção:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                # 6
item_set = set(item_list)                 # {1, 2, 3}
num_distinct_items = len(item_set)        # 3
distinct_item_list = list(item_set)       # [1, 2, 3]
```

Usaremos `sets` com menos frequência do que `dicts` e `lists`.

Controle de Fluxo

Como na maioria das linguagens de programação, você pode desempenhar uma ação condicionalmente usando `if`:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

Você também pode escrever um *ternário* if-then-else em uma linha, o que faremos ocasionalmente:

```
parity = "even" if x % 2 == 0 else "odd"
```

Python possui um loop `while`:

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

embora usaremos mais `for` e `in`:

```
for x in range(10):
    print x, "is less than 10"
```

Se você precisar de uma lógica mais complexa, pode usar `continue` e `break`:

```
for x in range(10):
    if x == 3:
        continue  # vai para a próxima iteração imediatamente
    if x == 5:
        break     # sai do loop completamente
    print x
```

Essa saída será 0, 1, 2 e 4.

Veracidade

Os Booleanos em Python funcionam como na maioria das linguagens, exceto que eles são iniciados por letras maiúsculas:

```
one_is_less_than_two = 1 < 2          # é igual a True
true_equals_false = True == False     # é igual a False
```

Python usa o valor `None` para indicar um valor não-existente. É parecido com o `null` das outras linguagens:

```
x = None
print x == None    # imprime True mas não é Pythonic
print x is None    # imprime True e é Pythonic
```

O Python permite que você use qualquer valor que espera por um Booleano. Todos os exemplos a seguir são "`Falsos`":

- `False`
- `None`
- `[]` (uma list vazia)
- `{}` (um dict vazio)
- `""`
- `set()`
- `0`
- `0.0`

Quase todo o restante pode ser tratado como `True`. Isso permite que você use declarações `if` para testar listas, strings ou dicionários vazios e assim por diante. Às vezes isso causa alguns pequenos bugs se você estiver esperando por este comportamento:

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

Uma forma mais simples de fazer o mesmo é:

```
first_char = s and s[0]
```

já que `and` retorna seu segundo valor quando o primeiro é "`verdadeiro`", ou o primeiro valor quando não é. Da mesma forma, se `x` é um número ou, possivelmente, `None`:

```
safe_x = x or 0
```

definitivamente é um número.

Python possui uma função `all`, que pega uma lista e retorna `True` precisamente quando todos os elementos forem verdadeiros, e uma função `any`, que retorna `True` quando pelo menos um elemento é verdadeiro:

```
all([True, 1, { 3 }])   # True
all([True, 1, {}])      # False, {} é falso
any([True, 1, {}])      # True, True é verdadeiro
all([])                 # True, sem elementos falsos na lista
any([])                 # False, sem elementos verdadeiros na lista
```

Não Tão Básico

Aqui, veremos alguns dos mais avançados recursos do Python que serão úteis para trabalhar com dados.

Ordenação

Toda lista de Python possui um método `sort` que ordena seu espaço. Se você não quer bagunçar sua lista, você pode usar a função `sorted`, que retornam uma lista nova:

```
x = [4,1,2,3]
y = sorted(x)      # é [1,2,3,4], x não mudou
x.sort()           # agora x é [1,2,3,4]
```

Por padrão, `sort` (e `sorted`) organizam uma lista da menor para a maior baseada em uma comparação ingênua de elementos uns com os outros.

Se você quer que os elementos sejam organizados do maior para o menor, você pode especificar o parâmetro `reverse=True`. E, em vez de comparar os elementos com eles mesmos, compare os resultados da função que você especificar com `key`:

```
# organiza a lista pelo valor absoluto do maior para o menor
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]

# organiza as palavras e contagens da mais alta para a mais baixa
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

Compreensões de Lista

Com frequência, você vai querer transformar uma lista em outra, escolhendo apenas alguns elementos, transformando tais elementos ou ambos. O modo Pythonic de fazer isso são as *compreensões de lista*:

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

você pode transformar dicionários em conjuntos da mesma forma:

```
square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set  = { x * x for x in [1, -1] }       # { 1 }
```

Se você não precisar do valor da lista, é comum usar um sublinhado como variável:

```
zeroes = [0 for _ in even_numbers]      # possui o mesmo tamanho de even_numbers
```

Uma compreensão de lista pode incluir múltiplos `for`:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

e os for que vêm depois podem usar os resultados dos primeiros:

```
increasing_pairs = [(x, y)                          # somente pares com x < y,
                    for x in range(10)              # range(lo, hi) é igual a
                    for y in range(x + 1, 10)]      # [lo, lo + 1, ..., hi - 1]
```

Usaremos bastantes compreensões de lista.

Geradores e Iteradores

Um problema com as listas é que elas podem crescer sem parar facilmente. `range(1000000)` cria uma lista com um milhão de elementos. Se você apenas precisa lidar com eles um de cada vez, isso pode ser uma fonte infinita de ineficiência (ou esgotamento de memória). Se você precisar de poucos valores, calcular todos seria uma perda de tempo.

Um *gerador* é algo sobre o qual você pode iterar (para nós, geralmente usando `for`) mas cujos valores são produzidos apenas quando necessários (*preguiçosamente*).

Uma forma de criar geradores é com funções e o operador `yield`:

```
def lazy_range(n):
    """uma versão preguiçosa de range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

O loop a seguir consumirá os valores `yield` um de cada vez até não sobrar mais nenhum:

```
for i in lazy_range(10):
    do_something_with(i)
```

(O Python geralmente vem com uma função `lazy_range` chamada `xrange` e, em Python 3, `range` é, em si, preguiçoso (lazy).) Isso significa que você poderia criar uma sequência infinita:

```
def natural_numbers():
    """retorna 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

embora você não deveria iterar sobre ela sem usar algum tipo de lógica `break`.

O outro lado da preguiça é que você somente pode iterar com um gerador uma vez. Se você precisar iterar múltiplas vezes, você terá que recriar um gerador todas as vezes ou usar uma lista.

Uma segunda forma de criar geradores é usar compreensões de `for` dentro de parênteses:

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
```

Lembre-se de que cada `dict` possui um método `items()` que retorna uma lista de seus pares valores-chave. Veremos com mais frequência o método `iteritems()`, que preguiçosamente `yields` (chama) os pares de valor-chave um de cada vez conforme iteramos sobre ele.

Aleatoriedade

Conforme aprendemos data science, precisaremos gerar números aleatórios com uma certa frequência, o que pode ser feito com o módulo `random`:

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.8444218515250481,      # random.random() produz números
#  0.7579544029403025,      # uniformemente entre 0 e 1
#  0.420571580830845,       # é a função aleatória que usaremos
#  0.25891675029296335]     # com mais frequência
```

O módulo `random` de fato produz números pseudoaleatórios (ou seja, determinísticos) baseado em um estado interno que você pode configurar com `random.seed` se quiser obter resultados reproduzíveis:

```
random.seed(10)          # configura seed para 10
print random.random()    # 0.57140259469
random.seed(10)          # reinicia seed para 10
print random.random()    # 0.57140259469 novamente
```

Às vezes usaremos `random.randrange`, que leva um ou dois argumentos e retorna um elemento escolhido aleatoriamente do `range()` correspondente:

```
random.randrange(10)     # escolhe aleatoriamente de range(10) = [0, 1, ..., 9]
random.randrange(3, 6)   # escolhe aleatoriamente de range(3, 6) = [3, 4, 5]
```

Existem mais alguns métodos que achamos convenientes em certas ocasiões. `random.shuffle` reordena os elementos de uma lista aleatoriamente:

```
up_to_ten = range(10)
random.shuffle(up_to_ten)
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0]   (seus resultados podem ser diferentes)
```

Se você precisar escolher um elemento randomicamente de uma lista, você pode usar `random.choice`:

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])     # "Bob" for me
```

E se você precisar escolher aleatoriamente uma amostra dos elementos sem substituição (por exemplo, sem duplicatas), você pode usar `random.sample`:

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

Para escolher uma amostra de elementos com substituição (por exemplo, permitindo duplicatas), você pode fazer múltiplas chamadas para `random.choice`:

```
four_with_replacement = [random.choice(range(10))
                         for _ in range(4)]

# [9, 4, 4, 2]
```

Expressões Regulares

As expressões regulares fornecem uma maneira de procurar por texto. São incrivelmente úteis mas um pouco complicadas, tanto que até existem livros sobre elas. Explicaremos mais detalhes nas poucas vezes que as encontrarmos; estes são alguns exemplos de como usá-las em Python:

```
import re

print all([                                    # todos são verdadeiros porque
    not re.match("a", "cat"),                  # * 'cat' não começa com 'a'
    re.search("a", "cat"),                     # * 'cat' possui um 'a'
    not re.search("c", "dog"),                 # * 'dog' não possui um 'c'
    3 == len(re.split("[ab]", "carbs")),       # * divide em a ou b para ['c','r','s']
    "R-D-" == re.sub("[0-9]", "-", "R2D2") # * substitui dígitos por traços
    ])  # imprime True
```

Programação Orientada a Objeto

Como muitas linguagens, o Python permite que você defina *classes* que encapsulam dados e as funções que as operam. As usaremos algumas vezes para tornar nosso código mais limpo e simples. Provavelmente é mais fácil explicá-las ao construir um exemplo repleto de anotações.

Imagine que não tivéssemos o `set` embutido em Python. Portanto, talvez quiséssemos criar nossa própria classe `Set`.

Qual comportamento nossa classe deveria ter? Dado um exemplo de `Set`, deveremos ser capazes de `add` (adicionar) itens nele, `remove` (remover) itens dele e verificar se ele `contains` (contém) um determinado valor. Criaremos todos eles como *funções de membro*, o que significa que os acessaremos com um ponto depois de um objeto `Set`:

```
# por convenção, damos nomes PascalCase às classes
class Set:

    # estas são as funções de membro
    # cada uma pega um parâmetro "self" (outra convenção)
    # que se refere ao objeto set sendo usado em questão

    def __init__(self, values=None):
        """este é o construtor.
        Ele é chamado quando você cria um novo Set.
        Você deveria usá-lo como
```

```
        s1 = Set()          # conjunto vazio
        s2 = Set([1,2,2,3]) # inicializa com valores"""

        self.dict = {} # cada instância de set possui sua própria propriedade dict
                       # que é o que usaremos para rastrear as associações
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """esta é a representação da string de um objeto Set
        se você digitá-la no prompt do Python ou passá-la para str()"""
        return "Set: " + str(self.dict.keys())

    # representaremos a associação como uma chave em self.dict com valor True
    def add(self, value):
        self.dict[value] = True

    # valor está no Set se ele for uma chave no dicionário
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

Que poderíamos usar desta forma:

```
s = Set([1,2,3])
s.add(4)
print s.contains(4)     # True
s.remove(3)
print s.contains(3)     # False
```

Ferramentas Funcionais

Ao passar as funções, algumas vezes queremos aplicá-las parcialmente para criar funções novas. Em um simples exemplo, imagine que temos uma função com duas variáveis:

```
def exp(base, power):
    return base ** power
```

e queremos usá-la para criar uma função de uma variável `two_to_the` cuja entrada é um `power` e cuja saída é o resultado de `exp(2, power)`.

Podemos, é claro, fazer isso com `def`, mas pode ser um pouco complicado:

```
def two_to_the(power):
    return exp(2, power)
```

Uma abordagem diferente é usar `functools.partial`:

```
from functools import partial
two_to_the = partial(exp, 2)        # agora é uma função de uma variável
print two_to_the(3)                 # 8
```

Você também pode usar `partial` para preencher os argumentos que virão depois se você especificar seus nomes:

```
square_of = partial(exp, power=2)
print square_of(3)                  # 9
```

Começa a ficar bagunçado quando você adiciona argumentos no meio da função, portanto tentaremos evitar isso.

Ocasionalmente usaremos `map`, `reduce` e `filter`, que fornecem alternativas funcionais para as compreensões de lista:

```
def double(x):
    return 2 * x

xs = [1, 2, 3, 4]
twice_xs = [double(x) for x in xs]          # [2, 4, 6, 8]
twice_xs = map(double, xs)                  # igual ao de cima
list_doubler = partial(map, double)         # função que duplica a lista
twice_xs = list_doubler(xs)                 # novamente [2, 4, 6, 8]
```

Você pode usar `map` com funções de múltiplos argumentos se fornecer múltiplas listas:

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

Igualmente, `filter` faz o trabalho de uma compreensão de lista `if`:

```
def is_even(x):
    """True se x for par, False se x for impar"""
    return x % 2 == 0

x_evens = [x for x in xs if is_even(x)]     # [2, 4]
x_evens = filter(is_even, xs)               # igual ao de cima
list_evener = partial(filter, is_even)      # função que filtra a lista
x_evens = list_evener(xs)                   # novamente [2, 4]
```

E `reduce` combina os dois primeiros elementos de uma lista, então esse resultado com o terceiro e esse resultado com o quarto; e assim por diante, produzindo um único resultado:

```
x_product = reduce(multiply, xs)            # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)    # função que reduz uma lista
x_product = list_product(xs)                # novamente = 24
```

Enumeração (enumerate)

Com alguma frequência, você vai querer iterar por uma lista e usar seus elementos e seus índices:

```
# não é Pythonic
for i in range(len(documents)):
    document = documents[i]
    do_something(i, document)

# também não é Pythonic
i = 0
for document in documents:
    do_something(i, document)
    i += 1
```

A solução Pythonic é `enumerate` (enumerar), que produz tuplas (`index`, `element`):

```
for i, document in enumerate(documents):
    do_something(i, document)
```

Da mesma forma, se apenas quisermos os índices:

```
for i in range(len(documents)): do_something(i)      # não é Pythonic
for i, _ in enumerate(documents): do_something(i)    # Pythonic
```

Usaremos isso bastante.

Descompactação de Zip e Argumentos

Com uma certa frequência, precisaremos `zip` (compactar) duas ou mais listas juntas. `zip` transforma listas múltiplas em uma única lista de tuplas de elementos correspondentes:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # é [('a', 1), ('b', 2), ('c', 3)]
```

Se as listas são de tamanhos diferentes, `zip` para assim que a primeira lista acaba.

Você também pode descompactar uma lista usando um truque curioso:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

O asterisco desempenha a *descompactação de argumento*, que usa os elementos de `pairs` como argumentos individuais para `zip`. Dá no mesmo se você chamasse:

```
zip(('a', 1), ('b', 2), ('c', 3))
```

que retorna `[('a','b','c'), ('1','2','3')]`.

Você pode usar a descompactação de argumento com qualquer função:

```
def add(a, b): return a + b

add(1, 2)       # retorna 3
add([1, 2])     # TypeError!
add(*[1, 2])    # retorna 3
```

É raro acharmos isso útil, mas quando fazemos é um truque engenhoso.

args e kwargs

Digamos que queremos criar uma função de ordem alta que tem como entrada uma função f e retorna uma função nova que retorna duas vezes o valor de f para qualquer entrada:

```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```

Isto funciona em alguns casos:

```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)          # 8 (== ( 3 + 1) * 2)
print g(-1)         # 0 (== (-1 + 1) * 2)
```

No entanto, ele falha com funções que possuem mais de um único argumento:

```
def f2(x, y):
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g() pega exatamente 1 argumento (2 dados)
```

O que precisamos é de alguma maneira de especificar uma função que leva argumentos arbitrários. Podemos fazer isso com a descompactação de argumento e um pouco de mágica:

```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")

# imprime
# unnamed args: (1, 2)
# keyword args:{'key2': 'word2', 'key': 'word'}
```

Ou seja, quando definimos uma função como essa, `args` é uma tupla dos seus argumentos sem nome e `kwargs` é um `dict` dos seus argumentos com nome. Funciona da forma contrária também, se você quiser usar uma `list` (ou `tuple`) e `dict` para *fornecer* argumentos para uma função:

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = { "z" : 3 }
print other_way_magic(*x_y_list, **z_dict)   # 6
```

Você poderia fazer todos os tipos de truques com isso; apenas usaremos para produzir outras funções de ordem alta cujas entradas podem aceitar argumentos arbitrários:

```
def doubler_correct(f):
    """funciona não importa que tipo de entradas f espera"""
    def g(*args, **kwargs):
        """quaisquer argumentos com os quais g é fornecido, os passa para f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
print g(1, 2) # 6
```

Bem-vindo à DataSciencester!

Concluímos aqui o treinamento dos funcionários novos. Ah, e também, tente não surrupiar nada.

Para Mais Esclarecimentos

- Não há escassez de tutoriais de Python no mundo. O site oficial (*https://docs.python.org/2/tutorial/*) não é um lugar ruim para começar.
- O tutorial oficial IPython (*http://ipython.org/ipython-doc/2/interactive/tutorial.html*) não é tão bom. Você talvez prefira assistir aos vídeos e às apresentações (*http://ipython.org/videos.html*). Como alternativa, *Python for Data Analysis* (O'Reilly) do Wes McKinney possui um ótimo capítulo sobre IPython.

CAPÍTULO 3

Visualizando Dados

Acredito que a visualização seja um dos meios mais poderosos de atingir metas pessoais.
—Harvey Mackay

Uma parte fundamental do kit de ferramentas do cientista de dados é a visualização de dados. Embora seja muito fácil criar visualizações, é bem mais difícil produzir algumas *boas*.

Existem dois usos primários para a visualização de dados:

- Para *explorar* dados
- Para *comunicar* dados

Neste capítulo, nos concentraremos em construir habilidades das quais você precisará para começar a explorar seus próprios dados e produzir as visualizações que usaremos no decorrer do livro. Como a maioria dos nossos tópicos do capítulo, a visualização de dados é uma rica área de estudos que merece seu próprio livro. Mas, mesmo assim, tentaremos mostrar o que é preciso e o que não é para uma boa visualização.

matplotlib

Existe uma grande variedade de ferramentas para visualizar dados. Usaremos a biblioteca `matplotlib` *(http://matplotlib.org/)*, que é altamente usada (apesar de sua idade). Se você estiver interessado em produzir elaboradas visualizações interativas para a web, provavelmente não será a melhor escolha mas, para simples gráficos de barras, de linhas e de dispersão, funciona muito bem.

Na verdade, usaremos o módulo `matplotlib.pyplot`. Em seu simples uso, `pyplot` mantém um estado interno em que você constrói uma visualização passo a passo. Ao terminar, você pode salvá-la (com `savefig()`) ou exibi-la (com `show()`).

Por exemplo, fazer um gráfico simples (como a Figura 3-1) é bem fácil:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# cria um gráfico de linha, anos no eixo x, gdp no eixo y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# adiciona um título
plt.title("GDP Nominal")

# adiciona um selo no eixo y
plt.ylabel("Bilhões de $")
plt.show()
```

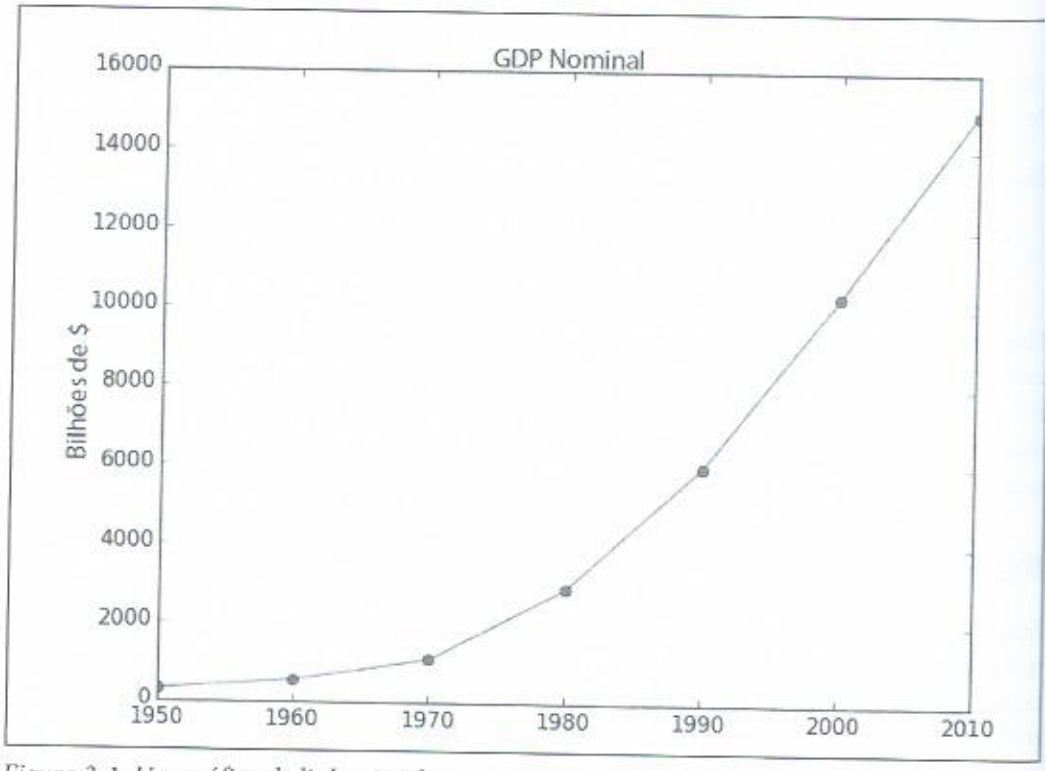

Figura 3-1. Um gráfico de linha simples

Construir gráficos que possuam uma boa qualidade de imagem é mais complicado e está além do escopo deste capítulo. Existem diversas maneiras de personalizar seus gráficos com rótulos de eixos, estilos de linha e marcadores de ponto, por exemplo. Em vez de passar por um tratamento de compreensão com essas opções, apenas usaremos (e chamaremos à atenção) alguns deles em nossos exemplos.

Embora não usemos muito dessa funcionalidade, `matplotlib` é capaz de produzir gráficos complicados dentro de gráficos, formatação sofisticada e visualizações interativas. Verifique sua documentação caso queira se aprofundar mais do que apresentamos neste livro.

Gráficos de Barra

Um gráfico de barra é uma boa escolha quando você quer mostrar como algumas quantidades variam entre um conjunto *particular* de itens. Por exemplo, a Figura 3-2 mostra quantas premiações do Oscar cada uma das variedades dos filmes ganharam:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# barras possuem o tamanho padrão de 0.8, então adicionaremos 0.1 às
# coordenadas à esquerda para que cada barra seja centralizada
xs = [i + 0.1 for i, _ in enumerate(movies)]

# as barras do gráfico com as coordenadas x à esquerda [xs], alturas [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("# de Premiações")
plt.title("Meus Filmes Favoritos")

# nomeia o eixo x com nomes de filmes na barra central
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

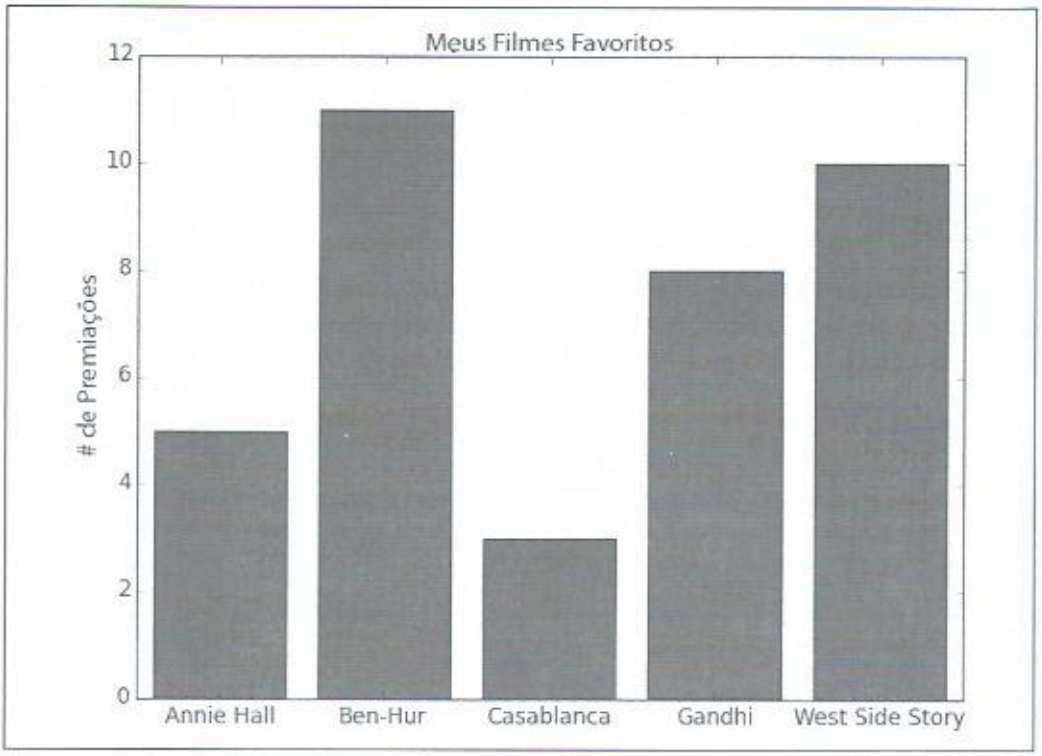

Figura 3-2. Um gráfico de barra simples

Um gráfico de barra também pode ser uma boa escolha para criar gráficos de histogramas de valores numéricos carregados, a fim de explorar visualmente como os valores são *distribuídos*, como na Figura 3-3:

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # move cada barra para a esquerda em 4
        histogram.values(),                # dá para cada barra sua altura correta
        8)                                 # dá para cada barra a largura de 8

plt.axis([-5, 105, 0, 5])                  # eixo x de -5 até 105,
                                           # eixo y de 0 até 5

plt.xticks([10 * i for i in range(11)])    # rótulos do eixo x em 0, 10, …, 100
plt.xlabel("Decil")
plt.ylabel("# de Alunos")
plt.title("Distribuição das Notas do Teste 1")
plt.show()
```

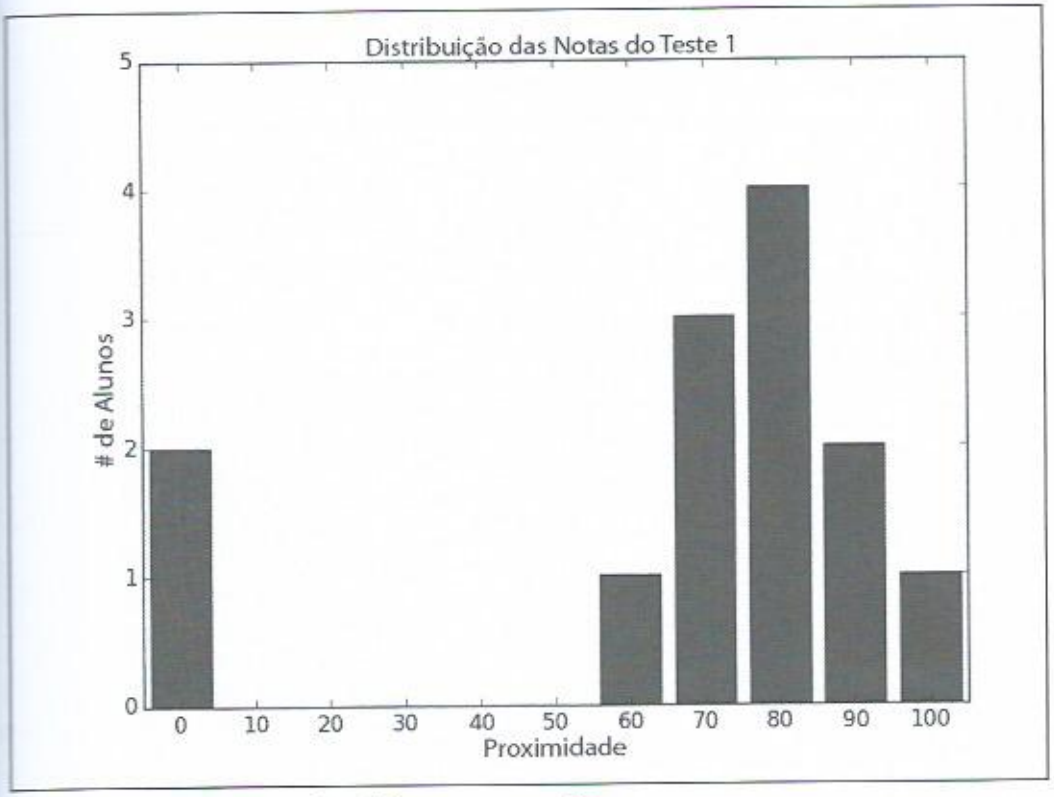

Figura 3-3. Usando um gráfico de barra para um histograma

O terceiro argumento para `plt.bar` especifica a largura da barra. Aqui, escolhemos a largura 8 (o que deixa um espaço pequeno entre as barras, já que nosso agrupamento possui o tamanho 10). Andamos com a barra em 4, para que (por exemplo) a barra "80" tenha seu lado esquerdo e direito em 76 e 84, e (portanto) seu centro em 80.

A chamada para `plt.axis` indica que queremos que o eixo x varie entre –5 até 105 (para que as barras "0" e "100" sejam mostradas por completo), e que o eixo y deveria variar de 0 até 5. A chamada para `plt.xticks` coloca os rótulos do eixo x em 0, 10, 20, ..., 100.

Seja criterioso quando usar `plt.axis()`. Ao criar gráficos de barra, não começar o eixo y em 0 é considerado ruim, já que essa é uma maneira fácil de enganar as pessoas (Figura 3-4):

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("# de vezes que ouvimos alguém dizer 'data science'"

# se você não fizer isso, matplotlib nomeará o eixo x de 0, 1
```

```
# e então adiciona a +2.013e3 para fora do canto (matplotlib feio!)
plt.ticklabel_format(useOffset=False)

# enganar o eixo y mostra apenas a parte acima de 500
plt.axis([2012.5,2014.5,499,506])
plt.title("Olhe o "Grande" Aumento!")
plt.show()
```

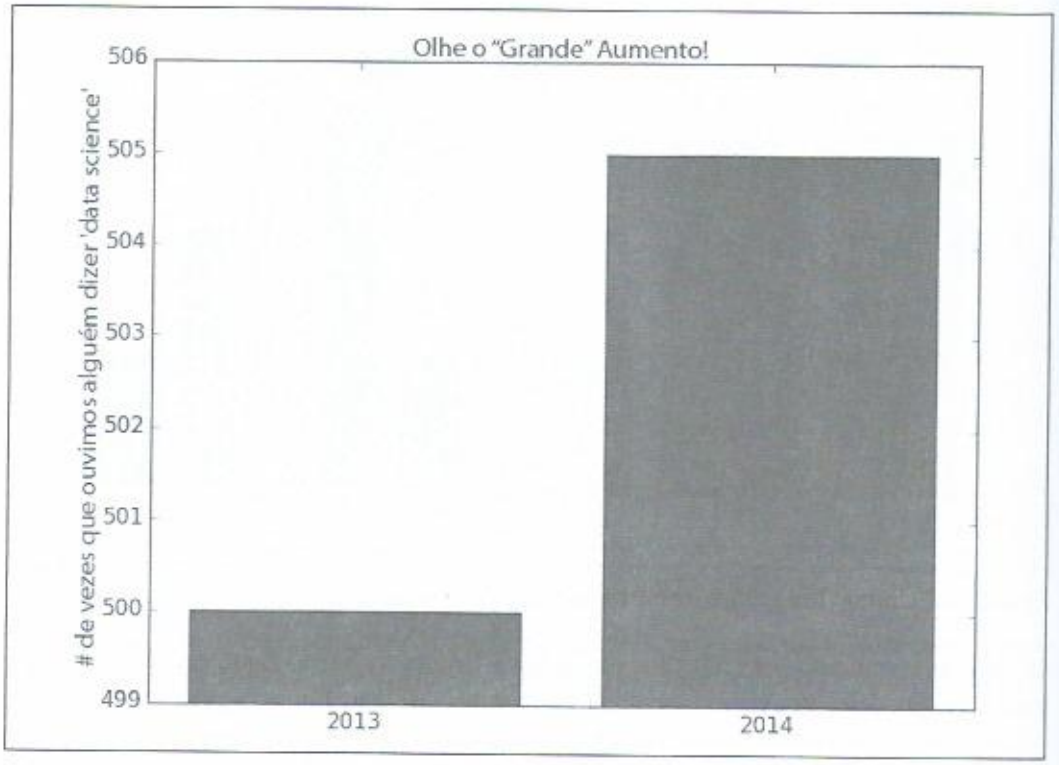

Figura 3-4. Um gráfico com um eixo y enganador

Na Figura 3-5, usamos eixos mais sensatos e, agora, parece menos impressionante:

```
plt.axis([2012.5,2014.5,0,550])
plt.title("Não Tão Grande Agora")
plt.show()
```

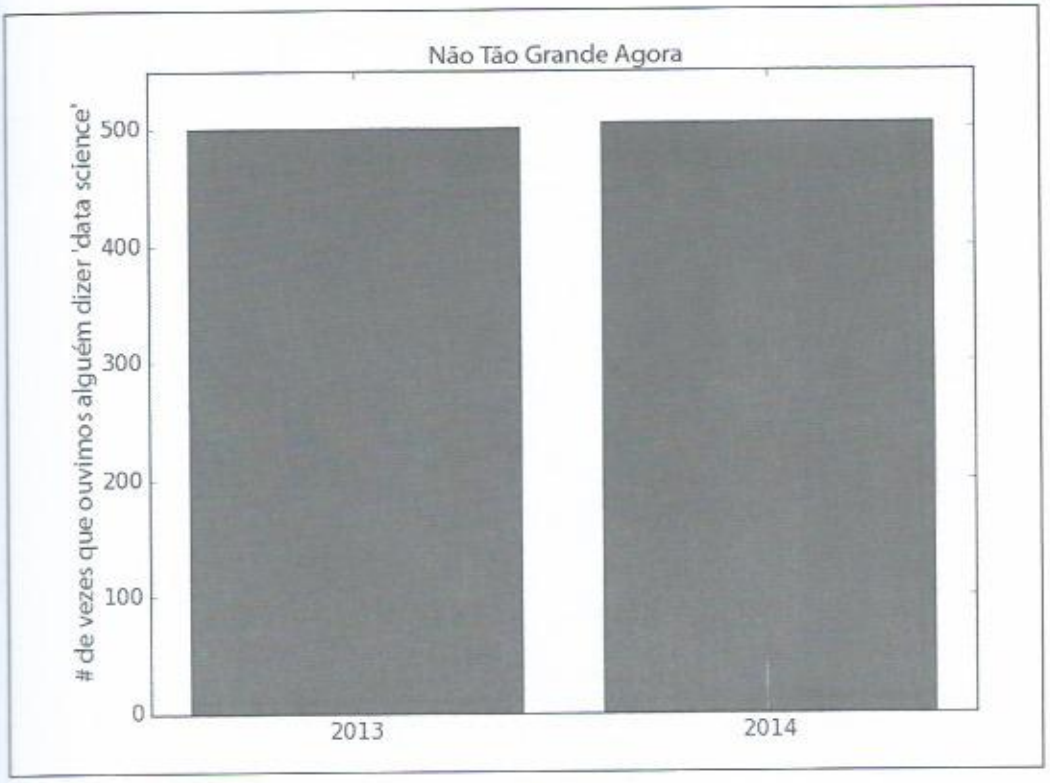

Figura 3-5. O mesmo gráfico sem um eixo y enganador

Gráficos de Linhas

Como já havíamos dito, podemos construir gráficos de linha usando `plt.plot()`. Eles são uma boa escolha ao mostrar tendências, como a Figura 3-6 ilustra:

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# podemos fazer múltiplas chamadas para plt.plot
# para mostrar múltiplas séries no mesmo gráfico
plt.plot(xs, variance,     'g-',  label='variance')    # linha verde sólida
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # linha com linha de ponto tracejado vermelho
plt.plot(xs, total_error,  'b:',  label='total error') # linha com pontilhado azul

# porque atribuimos rótulos para cada série
# podemos obter uma legenda gratuita
# loc=9 significa "top center"
plt.legend(loc=9)
plt.xlabel("complexidade do modelo")
plt.title("Compromisso entre Polarização e Variância")
plt.show()
```

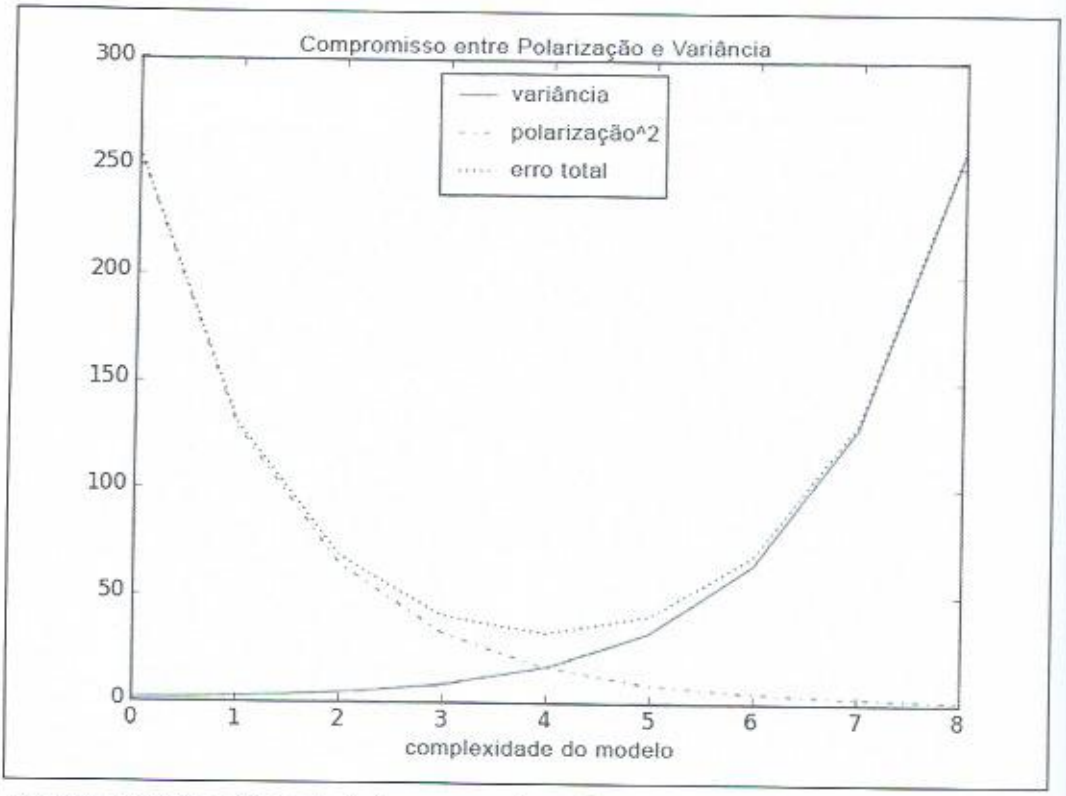

Figura 3-6. Vários gráficos de linha com uma legenda

Gráficos de Dispersão

Um gráfico de dispersão é a escolha certa para visualizar o relacionamento entre dois pares de conjuntos de dados. Por exemplo, a Figura 3-7 ilustra o relacionamento entre o número de amigos que seus usuários têm e o número de minutos que eles passam no site por dia:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# nomeia cada posição
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # coloca o rótulo com sua posição
        xytext=(5, -5),                  # mas compensa um pouco
        textcoords='offset points')

plt.title("Minutos Diários vs. Número de Amigos")
```

```
plt.xlabel("# de amigos")
plt.ylabel("minutos diários passados no site")
plt.show()
```

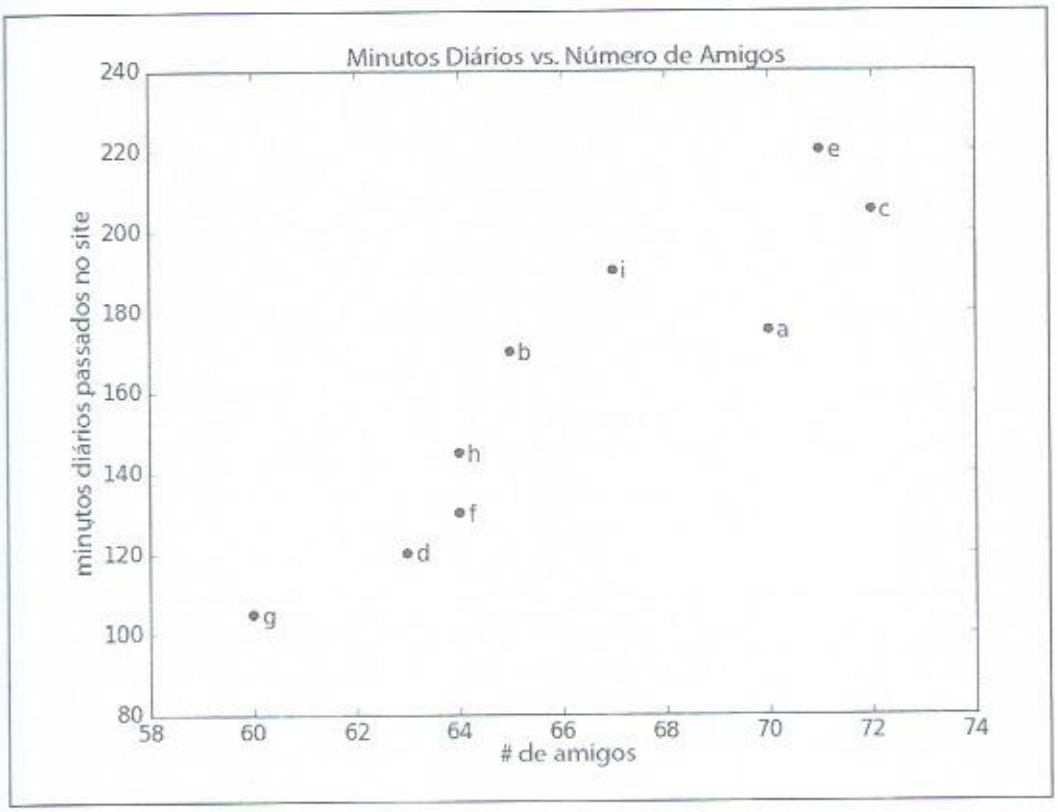

Figura 3-7. Uma dispersão de amigos e tempo no site

Se você está espalhando variáveis comparáveis, talvez você obtenha uma imagem enganosa se deixar matplotlib escolher a escala, como na Figura 3-8:

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Os eixos não são compatíveis")
plt.xlabel("nota do teste 2")
plt.ylabel("nota do teste 1")
plt.show()
```

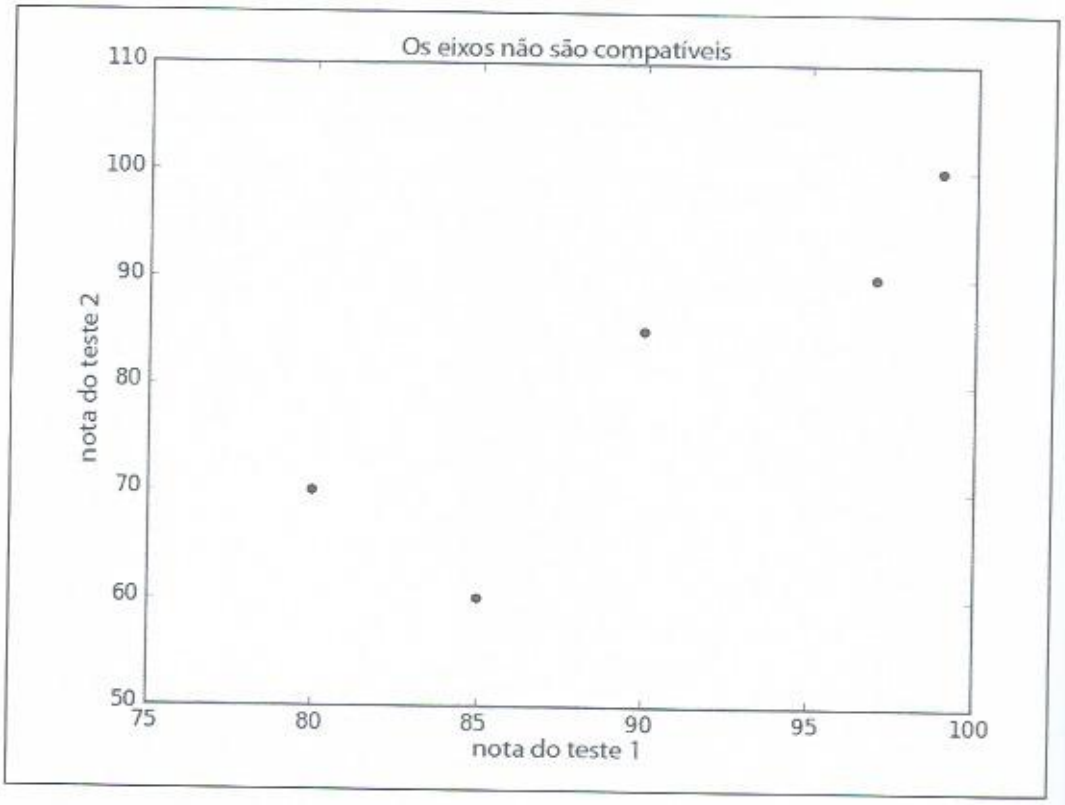

Figura 3-8. Um gráfico de dispersão com eixos incompatíveis

Se incluirmos uma chamada para `plt.axis("equals")`, o gráfico (Figura 3-9) mais preciso mostra que a maior parte da variação acontece no teste 2.

É o suficiente para você começar a criar visualizações. Aprenderemos muito mais sobre visualização no decorrer do livro.

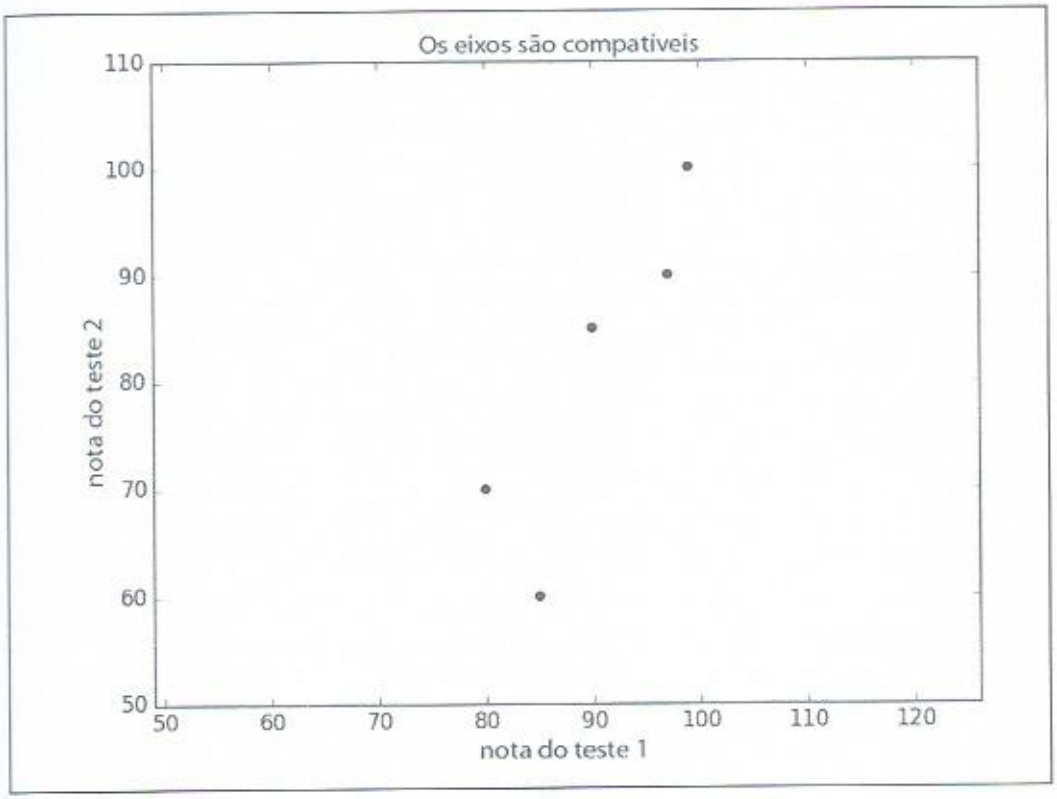

Figura 3-9. O mesmo gráfico de dispersão com eixos iguais

Para Mais Esclarecimentos

- seaborn (*http://stanford.io/1ycOjdI*) é construído no topo de `matplotlib` e permite que você produza facilmente visualizações mais bonitas (e mais complexas).
- D3.js (*http://d3js.org*) é uma biblioteca JavaScript que produz visualizações interativas sofisticadas para a web. Embora não esteja em Python, é uma tendência e amplamente usada, e vale a pena se familiarizar com ela.
- Bokeh (*http://bokeh.pydata.org*) é a biblioteca mais nova que traz o estilo de visualização D3 para Python.
- ggplot (*http://bit.ly/1ycOk1u*) é uma portagem Python da popular biblioteca `ggplot2`, que é amplamente usada para criar gráficos e diagramas de "qualidade de publicação". Provavelmente, é mais interessante se você já é um usuário voraz de `ggplot2` e um pouco opaco se não é.

CAPÍTULO 4

Álgebra Linear

Existe algo mais inútil ou menos útil que Álgebra?
—Billy Connolly

A Álgebra Linear é o ramo da matemática que lida com *espaços vetoriais*. Apesar de eu não achar que vou conseguir ensinar álgebra linear em um capítulo, ela sustenta um grande número de conceitos e técnicas de data science, o que significa que eu devo a você, ao menos, uma tentativa. O que aprenderemos neste capítulo, usaremos excessivamente no decorrer do livro.

Vetores

Abstratamente, os *vetores* são objetos que podem ser somados juntos (para formar vetores novos) e que podem ser multiplicados pelos *escalares* (por exemplo, números), também para formar vetores novos.

Concretamente (para nós), os vetores são pontos em algum espaço de dimensão finita. Apesar de você não pensar em seus dados como vetores, eles são uma ótima maneira de representar dados numéricos.

Por exemplo, se você tiver as alturas, pesos e idades de uma grande quantidade de pessoas, pode tratar seus dados como vetores tridimensionais (`height`, `weight`, `age`). Se você estiver ensinando uma turma com quatro testes, pode tratar as notas dos alunos como vetores quadridimensionais (`exam1`, `exam2`, `exam3`, `exam4`).

A abordagem inicial mais simples é representar vetores como listas de números. Uma lista de três números corresponde a um vetor em um espaço tridimensional, e vice-versa:

```
height_weight_age = [70,  # polegadas,
                     170, # quilos,
                     40 ] # anos

grades = [95,   # teste1
          80,   # teste2
```

```
    75,   # teste3
    62 ]  # teste4
```

Um problema com essa abordagem é que queremos realizar *aritmética* nos vetores. Como as listas de Python não são vetores (e, portanto, não facilita a aritmética com o vetor), precisaremos construir essas ferramentas aritméticas nós mesmos. Então, vamos começar por aí.

Para começar, frequentemente precisaremos de dois vetores. Os vetores se adicionam componente a componente. Isso significa que, se dois vetores `v` e `w` possuem o mesmo tamanho, sua soma é somente o vetor cujo primeiro elemento seja `v[0] + w[0]`, cujo segundo elemento seja `v[1] + w[1]`, e assim por diante. (Se eles não possuírem o mesmo tamanho, então não poderemos somá-los.)

Por exemplo, somar os vetores `[1, 2]` e `[2, 1]` resulta em `[1 + 2, 2 + 1]` ou `[3, 3]`, como mostra a Figura 4-1.

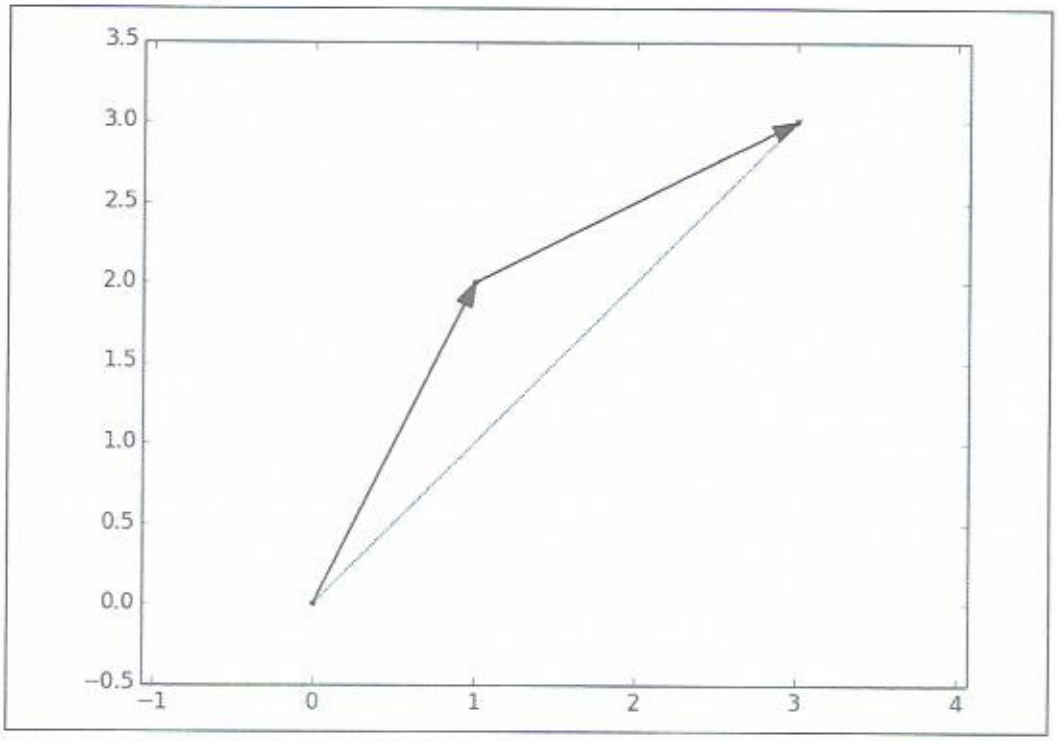

Figura 4-1. Somando dois vetores

Podemos facilmente implementar isso com vetores `zip` juntos e usar uma compreensão de lista para adicionar os elementos correspondentes:

```
def vector_add(v, w):
    """soma elementos correspondentes"""
    return [v_i + w_i
            for v_i, w_i in zip(v, w)]
```

Da mesma forma, para subtrair dois vetores, apenas subtraia os elementos correspondentes:

```
def vector_subtract(v, w):
    """subtrai elementos correspondentes"""
    return [v_i - w_i
            for v_i, w_i in zip(v, w)]
```

Às vezes queremos somar uma lista de vetores. Ou seja, criar um vetor novo cujo primeiro elemento seja a soma de todos os primeiros elementos, cujo segundo elemento seja a soma de todos os segundos elementos, e assim por diante. A maneira mais fácil de fazer isso é adicionar um vetor de cada vez:

```
def vector_sum(vectors):
    """soma toda os elementos correspondentes"""
    result = vectors[0]                          # começa com o primeiro vetor
    for vector in vectors[1:]:                   # depois passa por todos os outros
        result = vector_add(result, vector)      # e os adiciona ao resultado
    return result
```

Se você pensar a respeito, estamos apenas reduzindo (*reducing*) a lista de vetores usando `vector_add`, o que significa que podemos reescrever de forma reduzida usando funções de alta ordem:

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

ou até mesmo:

```
vector_sum = partial(reduce, vector_add)
```

embora esse último seja mais esperto do que útil.

Também precisaremos ser capazes de multiplicar um vetor por um escalar, que simplesmente fazemos ao multiplicar cada elemento do vetor por aquele número:

```
def scalar_multiply(c, v):
    """c é um número, v é um vetor"""
    return [c * v_i for v_i in v]
```

Isso permite que computemos a média de uma lista de vetores (do mesmo tamanho):

```
def vector_mean(vectors):
    """computar o vetor cujo i-ésimo elemento seja a média dos
    i-ésimos elementos dos vetores inclusos"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

Uma ferramenta menos óbvia é o *produto escalar* (*dot product*). O produto escalar de dois vetores é a soma de seus produtos componente a componente:

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

O produto escalar mede a distância a qual o vetor `v` se estende na direção de `w`. Por exemplo, se `w = [1, 0]` então `dot(v, w)` é o primeiro componente de `v`. Outra forma de dizer isso é que esse é o tamanho do vetor que você teria se projetasse `v` em `w` (Figura 4-2).

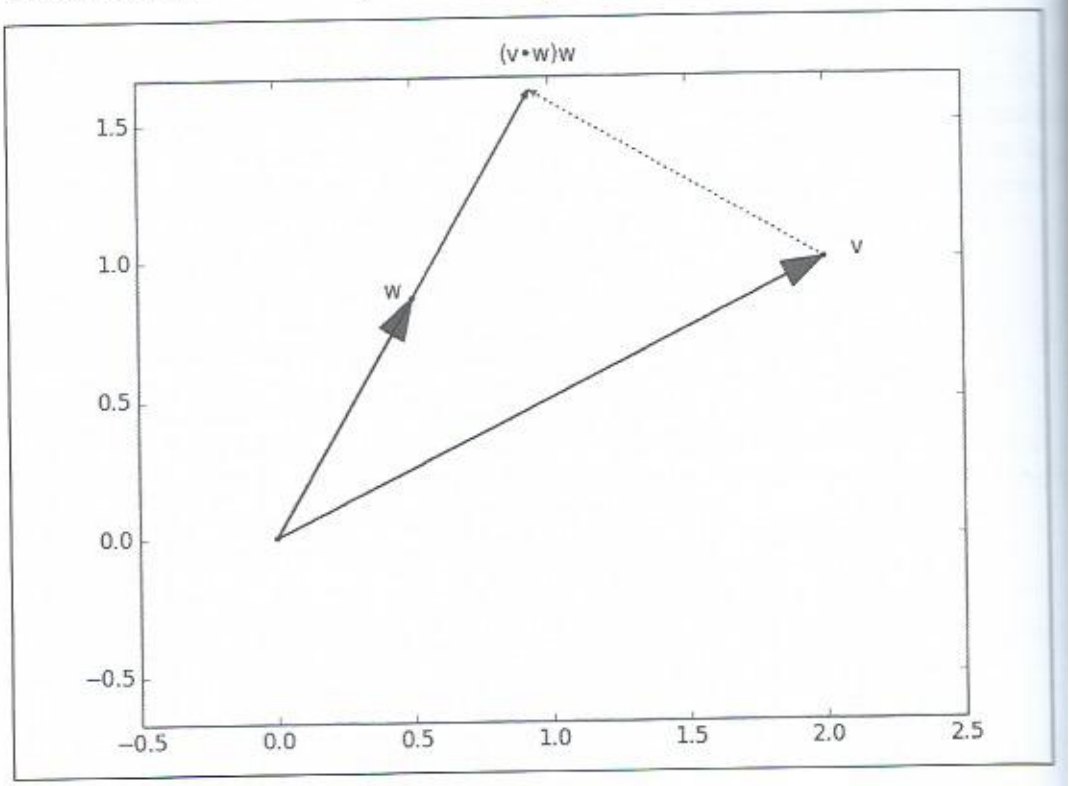

Figura 4-2. O produto escalar como projeção de vetor

Assim, é fácil computar a *soma dos quadrados* de um vetor:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

Que podemos usar para computar sua *magnitude* (ou tamanho):

```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))   # math.sqrt é a função da raiz quadrada
```

Agora temos todas as peças das quais precisamos para computar a distância entre dois vetores, definida como:

$$\sqrt{(v_1 - w_1)^2 + \ldots + (v_n - w_n)^2}$$

```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
   return math.sqrt(squared_distance(v, w))
```

Que fica mais claro se escrevermos como (o equivalente):

```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

Isso deve ser o suficiente para começarmos; usaremos essas funções constantemente no decorrer do livro.

Usar listas como vetores é bom para a exposição, mas terrível para o desempenho.

Na produção de código, você pode querer usar a biblioteca NumPy, que inclui uma classe de array de alta performance com todos os tipos de operações matemáticas inclusas.

Matrizes

Uma *matriz* é uma coleção de números bidimensional. Representaremos as matrizes como `listas` de `listas`, com cada lista interior possuindo o mesmo tamanho e representando uma *linha* da matriz. Se `A` é uma matriz, logo `A[i][j]` é o elemento da i-ésima linha e j-ésima da coluna. Por convenção matemática, geralmente usaremos letras maiúsculas para representar matrizes. Por exemplo:

```
A = [[1, 2, 3],  # A possui duas linhas e três colunas
     [4, 5, 6]]

B = [[1, 2],     # B possui três linhas e duas colunas
     [3, 4],
     [5, 6]]
```

Na matemática, normalmente nomearíamos a primeira linha da matriz de "linha 1" e a primeira coluna de "coluna 1". Já que estamos representando matrizes com as `listas` de Python, que são indexadas em zero, chamaremos a primeira linha de uma matriz de "linha 0" e a primeira coluna de "coluna 0".

Dada esta representação de lista-das-listas, a matriz `A` possui as linhas `len(A)` e colunas `len(A[0])`, que consideramos desta forma (shape):

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0   # número de elementos na primeira linha
    return num_rows, num_cols
```

Se uma matriz possui n linhas e k colunas, nos referiremos a ela como uma matriz $n \times k$. Podemos (e, às vezes, iremos) pensar em cada linha de uma matriz $n \times k$ como um vetor de tamanho k, e cada coluna como um vetor de tamanho n:

```
def get_row(A, i):
    return A[i]              # A[i] já é da linha A[i] é linha i-ésimo

def get_column(A, j):
    return [A_i[j]           # j-ésimo elemento da linha A_i
            for A_i in A]    # para cada linha A_i
```

Também queremos saber como criar matrizes dadas sua forma e uma função para produzir seus elementos. Podemos fazer isso usando uma compreensão de lista aninhada:

```
def make_matrix(num_rows, num_cols, entry_fn):
    """retorna a matriz num_rows X num_cols
    cuja entrada (i,j)th é entry_fn(i, j)"""
    return [[entry_fn(i, j)              # dado i, cria uma lista
             for j in range(num_cols)]   #   [entry_fn(i, 0), ... ]
            for i in range(num_rows)]    # cria uma lista para cada i
```

Dada esta função, você poderia fazer uma *matriz de identidade* 5×5 (com 1s na diagonal e 0s nos demais lugares) com:

```
def is_diagonal(i, j):
    """1's na diagonal, 0's nos demais lugares"""
    return 1 if i == j else 0

# [[1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 0],
#  [0, 0, 0, 0, 1]]
```

As matrizes serão importantes para nós de diversas formas.

Primeiro, podemos usar uma matriz para representar um conjunto de dados consistindo de múltiplos vetores, simplesmente considerando cada vetor como uma linha da matriz. Por exemplo, se você tivesse a altura, o peso e a idade de 1000 pessoas, você poderia colocá-los em uma matriz 1000×3:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

Segundo, como veremos mais tarde, podemos usar uma matriz $n \times k$ para representar uma função linear que mapeia vetores dimensionais k para vetores dimensionais n. Nossas várias técnicas e conceitos englobarão tais funções.

Terceiro, as matrizes podem ser usadas para representar relações binárias. No Capítulo 1, representamos as extremidades de uma rede como uma coleção de pares `(i, j)`. Uma representação alternativa seria criar uma matriz `A` tal que `A[i][j]` seja 1 se os nodos *i* e *j* estejam conectados e 0 de outra forma.

Lembre-se de que tínhamos antes:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Também poderíamos representar desta forma:

```
    #     usuário 0  1  2  3  4  5  6  7  8  9
    #
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # user 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # user 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # user 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # user 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # user 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # user 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # user 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # user 9
```

Se tivermos poucas conexões, essa será uma representação muto fraca já que você terá que completar o armazenamento com diversos zeros. No entanto, com a representação da matriz é bem mais fácil verificar se os dois nodos estão conectados — você apenas tem que procurar na matriz em vez de procurar (possivelmente) cada extremidade:

```
friendships[0][2] == 1   # True, 0 e 2 são amigos
friendships[0][8] == 1   # False, 0 e 8 não são amigos
```

Da mesma forma, para encontrar as conexões que um nodo possui, você precisa apenas inspecionar a coluna (ou a linha) correspondente àquele nodo:

```
friends_of_five = [i                                          # somente precisamos
                   for i, is_friend in enumerate(friendships[5])  # olhar para
                   if is_friend]                              # uma linha
```

Anteriormente, adicionamos uma lista de conexões para cada objeto de nodo para aumentar sua velocidade de processo, mas para um gráfico grande e em evolução, seria muito caro e de difícil manutenção.

Veremos matrizes novamente no decorrer do livro.

Para Mais Esclarecimentos

- A álgebra linear é amplamente usada por cientistas de dados (implícita com frequência, e não raramente por pessoas que não a entendem). Não seria uma má ideia ler um livro didático. Você pode encontrar vários disponíveis online:

  - *Linear Algebra*, da UC Davis (*http://bit.ly/1ycOq96*)
  - *Linear Algebra*, do Saint Michael's College (*http://bit.ly/1ycOpSF*)
  - Se gostar de aventuras, *Linear Algebra Done Wrong (http://bit.ly/1ycOt4W)* é um livro com uma introdução mais avançada

- Todos os exemplos construídos aqui você pode ter gratuitamente se usar NumPy (*http://www.numpy.org*). (E mais exemplos também.)

CAPÍTULO 5

Estatística

Os fatos são teimosos, mas as estatísticas são mais maleáveis.
—Mark Twain

A *estatística* se refere à matemática e às técnicas com as quais entendemos os dados. É um campo rico, amplo, mais adequado a uma prateleira (ou sala) em uma biblioteca em vez de um capítulo em um livro, portanto, nossa abordagem não será muito profunda. Em vez disso, tentarei explicar apenas o suficiente para ensinar a ser aventureiro e captar seu interesse para que você dê continuidade e aprenda mais.

Descrevendo um Conjunto Único de Dados

Por meio de uma combinação de discurso oral e sorte, a DataSciencester ampliou para dúzias de membros e o vice-presidente da Captação de Recursos solicita um relatório de quantos amigos seus membros possuem a fim de incluí-los em seus discursos no elevador.

Ao usar as técnicas do Capítulo 1, você é plenamente capaz de produzir dados. Mas, agora, você está diante do problema de como *descrevê-los*.

Uma descrição evidente de qualquer dado é simplesmente o dado em si:

```
num_friends = [100, 49, 41, 40, 25,
               # _ e muitos mais
              ]
```

Para um conjunto pequeno de dados, essa pode até ser a melhor representação. Mas, para um conjunto maior, ela é complicada e confusa. (Imagine olhar para uma lista de um milhão de números.) Por essa razão, usamos a estatística para destilar e comunicar os aspectos relevantes dos nossos dados.

Na primeira abordagem, colocamos a contagem de amigos em um histograma usando `Counter` e `plt.bar()` (Figura 5-1):

```
friend_counts = Counter(num_friends)
xs = range(101)                         # o valor maior é 100
ys = [friend_counts[x] for x in xs]     # a altura é somente # de amigos
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histograma da Contagem de Amigos")
plt.xlabel("# de amigos")
plt.ylabel("# de pessoas")
plt.show()
```

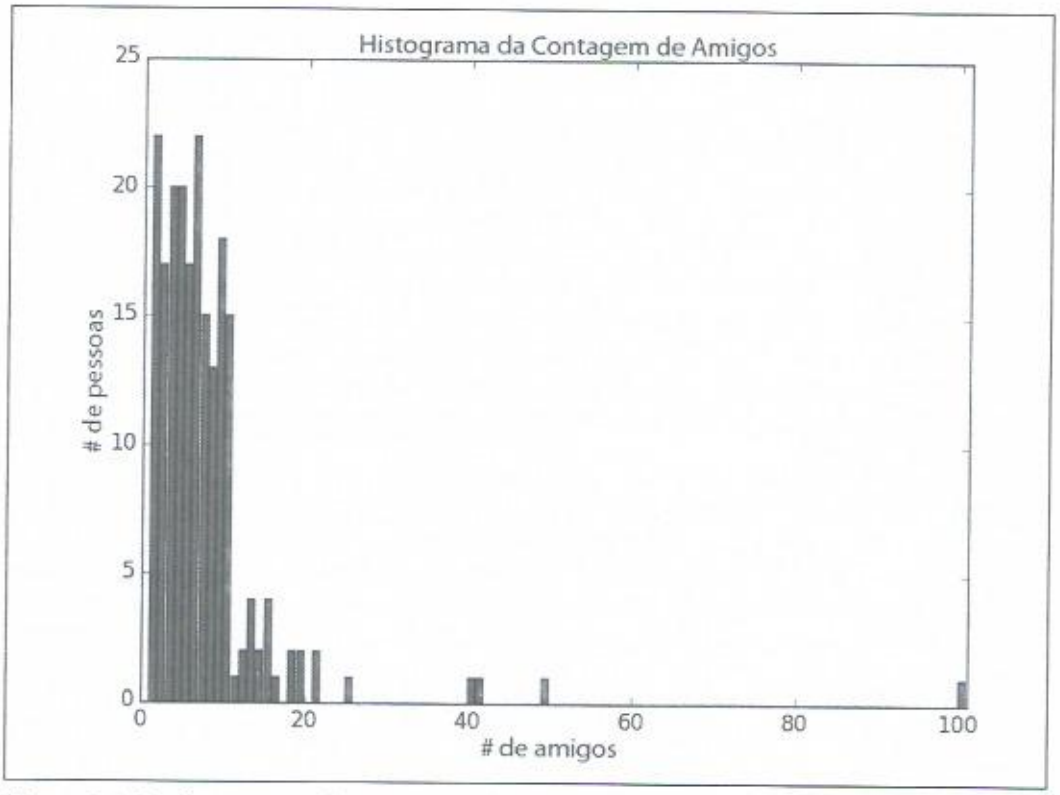

Figura 5-1. Um histograma da contagem de amigos

Infelizmente, esse gráfico ainda é muito difícil para inserir em discussões. Portanto, é melhor começar a gerar algumas estatísticas. Provavelmente, a estatística mais simples é o número de pontos nos dados:

```
num_points = len(num_friends)           # 204
```

Possivelmente, você também está interessado nos maiores e menores valores:

```
largest_value = max(num_friends)        # 100
smallest_value = min(num_friends)       # 1
```

que são apenas casos especiais de querer saber os valores em posições específicas:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]          # 1
second_smallest_value = sorted_values[1]   # 1
second_largest_value = sorted_values[-2]   # 49
```

Mas estamos apenas começando.

Tendências Centrais

Geralmente, queremos ter alguma noção de onde nossos dados estão centrados. A *média* será mais utilizada pois ela é soma dos dados dividido pela sua contagem:

```
# não está certo se você não importar a divisão de __future__
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

Se você possui dois pontos de dados, a média é o ponto no meio do caminho entres eles. Conforme você acrescenta mais pontos, a média se move, mas sempre depende do valor de cada ponto.

Algumas vezes também nos interessaremos pela *mediana*, que é o valor maior do meio (se o número de pontos de dados for ímpar) ou a média dos dois valores que estiverem bem no meio (se o número de pontos de dados for par).

Por exemplo, se tivermos cinco pontos de dados em um vetor variado `x`, a mediana é `x[5//2]` ou `x[2]`. Se tivermos seis pontos de dados, queremos a média de `x[2]` (o terceiro ponto) e `x[3]` (o quarto ponto).

Repare que — diferente da média — a mediana não depende de cada valor nos seus dados. Por exemplo, se você aumentar o maior ponto (ou diminuir o menor ponto), os pontos do meio permanecem intactos, logo, a mediana também.

A função `median` é um pouco mais complicada do que você pensa, principalmente por causa do caso "par":

```
def median(v):
    """encontra o valor mais ao meio de v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # se for ímpar, retorna o valor do meio
        return sorted_v[midpoint]
    else:
        # se for par, retorna a média dos valores do meio
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2
```

```
median(num_friends) # 6.0
```

Evidentemente, a média é mais fácil de computar e varia de modo mais suave conforme os dados mudam. Se tivermos *n* pontos de dados e um deles crescer em uma quantidade *e*, logo, necessariamente, a média aumentará de *e* / *n*. (Isso faz da média a responsável por todos os truques de cálculo.) Mas, para encontrar a mediana, temos que organizar nossos dados. E, ao mudar um dos pontos de dados em uma pequena quantidade *e*, talvez a mediana aumente de *e*, ou algum número menor que *e*, ou nenhum deles (dependendo do número de dados).

Há, na verdade, alguns truques não tão óbvios para computar medianas (*http://en.wikipedia.org/wiki/Quickselect*) sem sem organizar os dados. No entanto, eles estão além do escopo deste livro, então *temos* que organizá-los.

Ao mesmo tempo, a média é muito sensível aos valores discrepantes em nossos dados. Se nosso usuário mais amigável possui duzentos amigos (em vez de cem), então a média subiria para 7.82, enquanto a mediana permaneceria a mesma. Se os valores discrepantes têm a possibilidade de serem dados ruins (ou, de outro modo, não representativos de qualquer fenômeno que estejamos tentando entender), então a média pode nos levar a um engano. Por exemplo, conta-se que, em meados da década de 1980, a graduação da Universidade da Carolina do Norte com a maior média de salário inicial era geografia, principalmente por causa da estrela do NBA (e um valor discrepante) Michael Jordan.

Uma generalização da média é o *quantil*, que representa o valor abaixo do qual a uma certa porcentagem dos dados se encontra (a mediana representa o valor abaixo do qual 50% dos dados se encontram).

```
def quantile(x, p):
    """retorna o valor percentual p-ésimo em x"""
    p_index = int(p * len(x))
    return sorted(x)[p_index]

quantile(num_friends, 0.10) # 1
quantile(num_friends, 0.25) # 3
quantile(num_friends, 0.75) # 9
quantile(num_friends, 0.90) # 13
```

De modo menos comum, você talvez queira olhar para a *moda* ou os valores mais comuns:

```
def mode(x):
    """retorna uma lista, pode haver mais de uma moda"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.iteritems()
            if count == max_count]

mode(num_friends)       # 1 and 6
```

Mas usaremos a média com mais frequência.

Dispersão

A *dispersão* se refere à medida de como os nossos dados estão espalhados. Tipicamente, eles são estatísticas em que valores perto de zero significam *não estão espalhados de forma alguma* e para valores maiores (o que quer que isso signifique) significa *muito espalhados*. Por exemplo, uma simples medida é a *amplitude*, que é a diferença entre o maior e o menor elemento:

```
# "amplitude" já possui significado em Python, então usaremos um nome diferente
def data_range(x):
    return max(x) - min(x)

data_range(num_friends) # 99
```

A amplitude é zero quando o `max` e o `min` são iguais, o que acontece apenas se os elementos de `x` forem todos iguais, o que significa que *os dados estão o menos dispersos possível*. Por outro lado, se a amplitude é ampla, então o `max` é bem maior do que o `min` e os dados estão *mais espalhados*.

Assim como a mediana, a amplitude não depende de fato de todo o conjunto de dados. Um conjunto de dados cujos pontos estão todos entre 0 ou 100 possui a mesma amplitude que um cujos valores são 0, 100 e muitos 50s. Mas parece que o primeiro conjunto de dados "deveria" estar mais espalhado.

Uma medida de dispersão mais complexa é a *variância*, computada desta forma:

```
def de_mean(x):
    """desloca x ao subtrair sua média (então o resultado tem a média 0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]

def variance(x):
    """presume que x tem ao menos dois elementos"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)

variance(num_friends) # 81.54
```

Parece que é quase o desvio do quadrado médio da média, exceto que estamos dividindo por `n-1` em vez de `n`. Na verdade, quando com uma amostra de uma população maior, `x_bar` é apenas uma *estimativa* da média real, o que significa que na média `(x_i - x_bar)**2` há um subestimado quadrado médio da média de `x_i` da média, e é por isso que dividimos por `n-1` em vez de `n`. Veja a Wikipédia em *http://bit.ly/1L2EapI*.

Agora, qualquer que seja a unidade na qual nossos dados estão (por exemplo, "friends"), todas as nossas medidas de tendências centrais estão na mesma unidade. A amplitude estará naquela mesma unidade também. A variância, por outro lado, possui unidades que são os *quadrados* das unidades originais (por exemplo, "friends squared"). Como pode ser difícil entender isso, geralmente olhamos para o *desvio padrão*:

```
def standard_deviation(x):
    return math.sqrt(variance(x))

standard_deviation(num_friends) # 9.03
```

Tanto a amplitude quanto o desvio padrão possuem o mesmo problema de valor discrepante que vimos com a média. Usando o mesmo exemplo, se nosso usuário mais amigável tivesse duzentos amigos, o desvio padrão seria de 14,89, mais do que 60% a mais!

E uma alternativa mais robusta computa a diferença entre os percentos (quantos) 75% e 25% do valor:

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)

interquartile_range(num_friends) # 6
```

que não é afetado por uma pequena quantidade de valores discrepantes.

Correlação

A vice-presidente de Crescimento na DataSciencester tem uma teoria que a quantidade de tempo gasto pelas pessoas no site é relacionada ao número de amigos que elas possuem (ela não é uma vice-presidente à toa), e ela pediu para você verificar isso.

Após examinar os registros do tráfego, você desenvolve uma lista `daily_minutes` que mostra quantos minutos por dia cada usuário passa na DataSciencester e você havia ordenado essa lista para que seus elementos correspondessem aos elementos da lista anterior `num_friends`. Gostaríamos de investigar a relação entre essas duas métricas.

Primeiro, investigaremos a *covariância*, o equivalente pareado da variância. Enquanto a variância mede como uma única variável desvia de sua média, a covariância mede como duas variáveis variam em conjunto de suas médias:

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

Lembre-se que o `dot` resume os produtos dos pares correspondentes dos elementos. Quando os elementos correspondentes de `x` e `y` estão acima ou abaixo de suas médias, um número positivo entra na soma. Quando um está acima de sua média e o outro está abaixo, um

número negativo entra na soma. Na mesma proporção, uma covariância positiva "grande" significa que x tende a ser grande quando y é grande e pequeno quando y é pequeno. Uma covariância negativa "grande" significa o oposto — que x tende a ser pequeno quando y é grande e vice-versa. Uma covariância perto de zero significa que tal relação não existe.

Mesmo assim, esse número pode ser difícil de ser interpretado por dois motivos:

- Suas unidades são o produto das unidades de entrada (por exemplo, minutos-amigo-por-dia), o que pode ser difícil de entender. (O que é um "minutos-amigo-por-dia"?)
- Se cada usuário tiver duas vezes mais amigos (mas o mesmo número de minutos), a covariância seria duas vezes maior. Mas, por algum motivo, as variáveis seriam apenas inter-relacionadas. Visto de outra maneira, é arriscado dizer o que conta como uma covariância "grande".

Por tais motivos, é mais comum considerar a *correlação*, que divide os desvios padrões das duas variáveis:

```
def correlation(x, y):
    stdev_x = standard_deviation(x)
    stdev_y = standard_deviation(y)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(x, y) / stdev_x / stdev_y
    else:
        return 0    # se não houver amplitude, a correlação é zero

correlation(num_friends, daily_minutes) # 0.25
```

A `correlation` não possui unidade e sempre permanece entre –1 (anticorrelação perfeita) e 1 (correlação perfeita). Um número como 0,25 representa uma correlação positiva relativamente fraca.

No entanto, algo que esquecemos de fazer foi examinar nossos dados. Dê uma olhada na Figura 5-2.

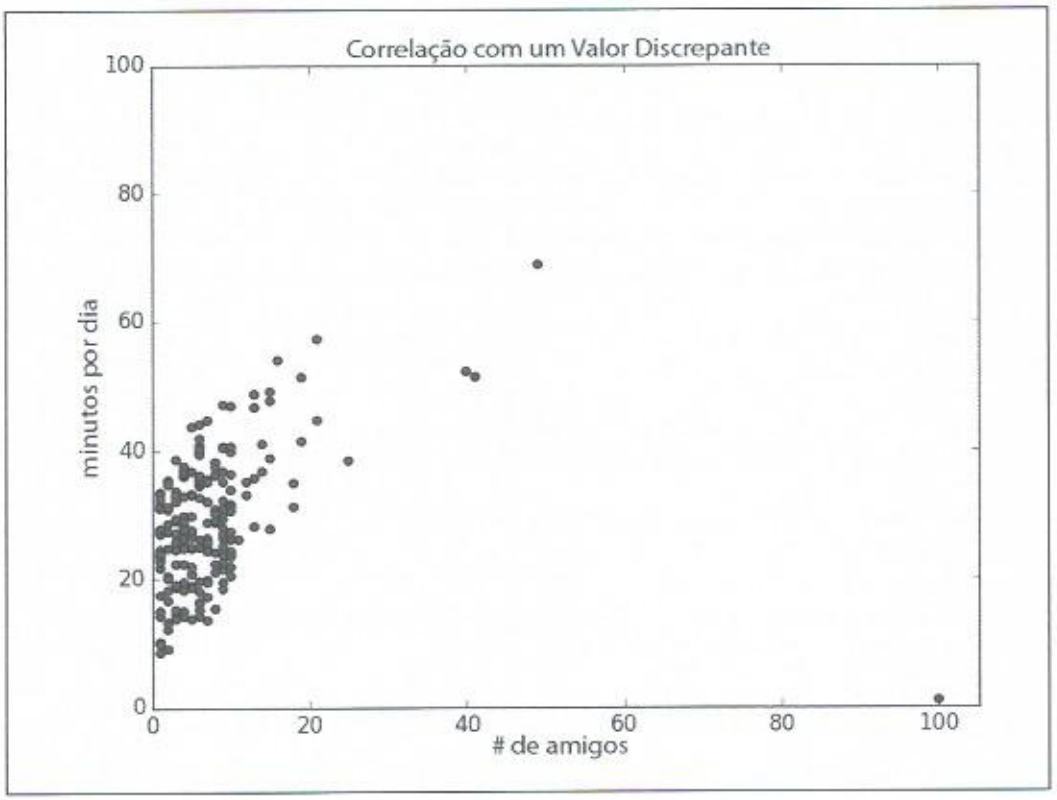

Figura 5-2. Correlação com um valor discrepante

A pessoa com 100 amigos (que passa apenas um minuto por dia no site) é um grande valor discrepante e a correlação pode ser muito sensível para valores discrepantes. O que acontece se o ignorarmos?

```
outlier = num_friends.index(100)    # índice do valor discrepante

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

correlation(num_friends_good, daily_minutes_good) # 0.57
```

Sem o valor discrepante, há uma correlação bem mais forte (Figura 5-3).

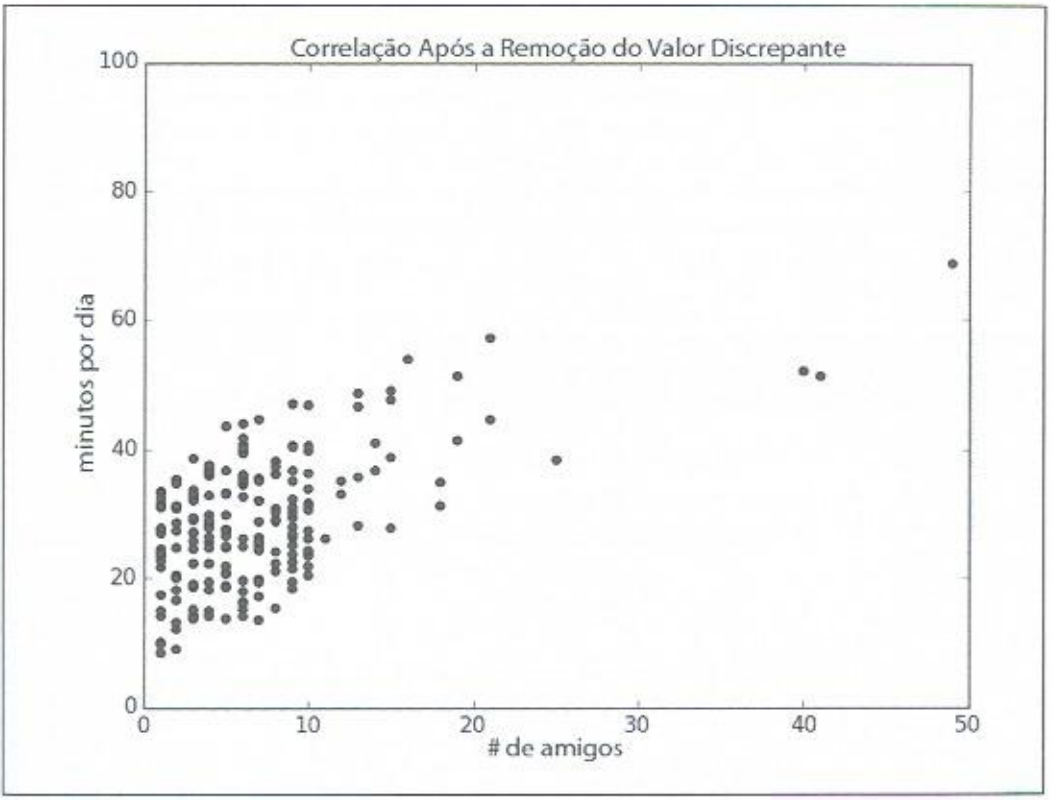

Figura 5-3. Correlação após a remoção do valor discrepante

Você averígua e descobre que o valor discrepante era, na verdade, uma conta *teste* interna que ninguém se preocupou em remover. Então sinta-se bem ao excluí-la.

Paradoxo de Simpson

Uma surpresa incomum ao analisar dados é o Paradoxo de Simpson, em que as correlações podem ser enganosas quando as variáveis *de confusão* são ignoradas.

Por exemplo, imagine que você possa identificar todos os seus membros como cientistas de dados da Costa Leste e da Costa Oeste. Você decide examinar quais são os mais amigáveis:

costa	quantidade de membros	média da quantidade de amigos
Costa Oeste	101	8.2
Costa Leste	103	6.5

Certamente parece que os cientistas de dados da Costa Oeste são mais amigáveis do que os da Costa Leste. Seus colegas de trabalho investem em todo o tipo de teorias no motivo pelo qual isso talvez aconteça: talvez seja o sol, o café, a produção orgânica ou a brisa descontraída do Pacífico.

Ao brincar com os dados, você descobre algo muito estranho. Se você olhar somente para as pessoas com PhDs, os cientistas de dados da Costa Leste possuem uma média maior de amigos. E, se você olhar para as pessoas sem PhDs, os cientistas de dados da Costa Leste também possuem uma média maior de amigos!

costa	grau	quantidade de membros	média da quantidade de amigos
Costa Oeste	PhD	35	3.1
Costa Leste	PhD	70	3.2
Costa Oeste	sem PhD	66	10.9
Costa Leste	sem PhD	33	13.4

Uma vez que você verifica os diplomas dos usuários, a correlação vai em direção oposta! Agrupando os dados como Costa Leste/Oeste mascarou o fato de que os cientistas de dados da Costa Leste se distorcem mais intensamente com os tipos de PhDs.

Tal fenômeno surge no mundo real com alguma regularidade. O ponto chave é que a correlação é medir a relação entre suas duas variáveis *com tudo o mais sendo igual*. Se as suas aulas de dados fossem atribuídas aleatoriamente, como se fossem classificadas como um experimento bem projetado, "por mais que sejam iguais" pode não ser uma premissa terrível. Mas quando há um padrão mais profundo na atribuição de classe, "por mais que sejam iguais" pode ser uma premissa terrível.

O único modo real de evitar isso é *conhecendo seus dados* e fazendo o que puder para ter certeza de que verificou pelos possíveis fatores de confusão. Evidentemente, nem sempre é possível. Se você não tivesse a informação educacional desses 200 cientistas de dados, você talvez concluísse que havia algo inerente e mais sociável sobre a Costa Oeste.

Alguns Outros Pontos de Atenção sobre Correlação

Uma correlação de zero indica que não há uma relação linear entre as duas variáveis. Porém, podem haver vários tipos de relações. Por exemplo, se:

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

então x e y possuem uma correlação zero. Mas, certamente, têm uma relação — cada elemento de y é igual ao valor absoluto do elemento correspondente de x. O que eles não têm é

uma relação em que saber como `x_i` se compara a `mean(x)` nos dá informações sobre como `y_i` se compara a `mean(y)`. Esse é o tipo de relação que a correlação procura.

Além do mais, a correlação não diz nada sobre o tamanho das relações. As variáveis:

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

estão perfeitamente correlacionadas, mas (dependendo do que você está medindo) é bem possível que essa relação não seja muito interessante.

Correlação e Causalidade

Você já deve ter escutado alguma vez que "correlação não é causalidade", mais possivelmente de uma pessoa pesquisando dados que impuseram desafios às partes da visão de mundo que ele estava relutante em questionar. Apesar disso, este é um ponto importante — se `x` e `y` possuem uma forte correlação, isso talvez signifique que `x` causa `y`, que `y` causa `x` e que cada um causa o outro, que algum terceiro fator causa ambos ou pode não significar nada.

Considere a relação entre `num_friends` e `daily_minutes`. É possível que ter mais amigos *faça com que* os usuários da DataSciencester passem mais tempo no site. Esse pode ser o caso se cada amigo postar uma certa quantidade de conteúdo diariamente pois, quanto mais amigos você tem, mais tempo você leva para pôr em dia suas atualizações.

Porém, também é possível que, quanto mais tempo você passe discutindo nos fóruns da DataSciencester, mais você encontrará e fará amizade com pessoas parecidas com você. Ou seja, passar mais tempo no site *faz com que* os usuários tenham mais amigos.

Uma terceira possibilidade seria que os usuários mais dedicados com data science passassem mais tempo no site (porque eles acham mais interessante) e ativamente colecionassem mais amigos data science (porque eles não querem se associar com mais ninguém).

Uma maneira de se sentir mais confiante sobre causalidade é conduzir experimentos aleatórios. Se você pode dividir seus usuários aleatoriamente em dois grupos com demografia parecida e dar a um dos grupos uma experiência um pouco diferente, logo você verá que experiências diferentes estão causando resultados diferentes.

Por exemplo, se você não se importar de ser acusado de fazer experimentos com seus usuários (*http://nyti.ms/1L2DzEg*), você pode escolher um subconjunto aleatório de usuários e mostrar a eles o conteúdo de somente uma parte dos seus amigos. Se esse subconjunto subsequentemente passar menos tempo no site, isso lhe dará mais confiança de que ter mais amigos *faz passar* mais tempo no site.

Para Mais Esclarecimentos

- SciPy (*http://bit.ly/1L2H0Lj*), pandas (*http://pandas.pydata.org*), e StatsModels (*http://bit.ly/1L2GQnc*) vêm com uma grande variedade de funções estatísticas.
- Estatística é *importante*. (Ou, talvez, estatísticas *são* importantes?) Se você quiser um bom cientista de dados, seria uma boa ideia ler um livro didático de estatística. Muitos estão disponíveis online. Gosto muito destes dois:
  - *OpenIntro Statistics (http://bit.ly/1L2GKvG)*
  - *OpenStax Introductory Statistics (http://bit.ly/1L2GJrM)*

CAPÍTULO 6

Probabilidade

As leis da probabilidade, no geral tão verdadeiras, no particular tão enganosas.
—Edward Gibbon

É difícil praticar data science sem algum entendimento de probabilidade e sua matemática. Igualmente à nossa abordagem sobre estatística no Capítulo 5, nos dedicaremos muito a eliminar muitas das tecnicalidades.

Para os nossos propósitos, você deveria pensar em probabilidade como uma forma de quantificar a incerteza associada com *eventos* escolhidos a partir de um *universo* deles. Em vez de estudar tecnicamente esses métodos, pense em jogar um dado. O universo consiste de todos os resultados possíveis. Cada subconjunto desses resultados é um evento; por exemplo, "o dado mostra o número um" ou "o dado mostra um número ímpar".

Desta forma, escrevemos P(E) para significar "a probabilidade do evento E".

Usaremos a teoria da probabilidade para construir modelos. Usaremos a probabilidade para avaliar modelos. Usaremos a teoria da probabilidade em todos os lugares.

Alguém poderia, se tivesse vontade, ir bem a fundo na filosofia do que a teoria da probabilidade *significa*. (Melhor fazer isso com algumas cervejas.) Não faremos isso.

Dependência e Independência

A grosso modo, dizemos que dois eventos E e F são *dependentes* se soubermos algo sobre se E ocorre nos der informações sobre se F ocorre (e vice-versa). Do contrário, são independentes.

Por exemplo, se jogarmos uma moeda honesta duas vezes, sabendo que a primeira jogada é coroa, não temos como saber se a segunda jogada vai dar o mesmo resultado. Esses eventos são independentes. Por outro lado, se soubéssemos que a primeira jogada fosse coroa, certamente teríamos a informação sobre se ambas as jogadas seriam cara. (Se a primeira joga-

da é coroa, então, definitivamente, não é o caso de que as duas jogadas são cara.) Esses dois eventos são dependentes.

Matematicamente, dizemos que os dois eventos E e F são independentes se a probabilidade deles acontecerem é o produto da probabilidade de que cada um deles aconteça:

$$P(E, F) = P(E)P(F)$$

No exemplo anterior, a probabilidade da "primeira jogada ser coroa" é de 1/2, e a probabilidade de "ambas serem cara" é de 1/4, mas a probabilidade de "a primeira jogada ser coroa e ambas serem cara" é 0.

Probabilidade Condicional

Quando os dois eventos E e F são independentes, por definição, temos:

$$P(E, F) = P(E)P(F)$$

Se não são necessariamente independentes (e a probabilidade de F não for 0), logo definimos a probabilidade de E "condicionada a F" assim:

$$P(E|F) = P(E, F)/P(F)$$

Você deveria entender isso como a probabilidade de E acontecer uma vez que sabemos que F acontece.

Geralmente reescrevemos desta forma:

$$P(E, F) = P(E|F)P(F)$$

Quando E e F são independentes, você pode verificar que isso resulta em:

$$P(E|F) = P(E)$$

que é a maneira matemática de explicar que saber que F ocorreu não nos dá nenhuma informação adicional sobre se E ocorreu.

Um exemplo comum e traiçoeiro envolve uma família com dois filhos (desconhecidos).

Se presumirmos que:

1. É igualmente possível que cada criança seja menino ou menina
2. O gênero da segunda criança é independente do gênero da primeira, então o evento

"nenhuma menina" tem a probabilidade de 1/4, o evento "uma menina, um menino" tem a probabilidade de 1/2 e o evento "duas meninas" tem a probabilidade de 1/4.

Agora, podemos perguntar: qual a probabilidade de o evento "as duas crianças são meninas" (B) ser condicionado pelo evento "a criança mais velha é uma menina" (G)? Usando a definição de probabilidade condicional:

$$P(B|G) = P(B, G)/P(G) = P(B)/P(G) = 1/2$$

uma vez que o evento B e G ("ambas as crianças são meninas *e* a criança mais velha é uma menina") é apenas o evento B. (Já sabendo que as duas crianças são meninas, é obrigatoriamente verdade que a criança mais velha seja menina.)

Esse resultado talvez corresponda a sua intuição.

Também poderíamos perguntar sobre a probabilidade do evento "as duas crianças são meninas" ser condicional ao evento "ao menos uma das crianças é menina" (L). Surpreendentemente, a resposta é diferente de antes!

Anteriormente, os eventos B e L ("as duas crianças são meninas *e* ao menos uma delas é uma menina") é apenas o evento B. Isso significa que temos:

$$P(B|L) = P(B, L)/P(L) = P(B)/P(L) = 1/3$$

Como pode ser esse o caso? Bem, se tudo que você sabe é que ao menos uma das crianças é menina, então é duas vezes mais provável que a família tenha um menino e uma menina do que duas meninas.

Podemos verificar isso ao "gerar" várias famílias:

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1
```

```
print "P(both | older):", both_girls / older_girl        # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl     # 0.342 ~ 1/3
```

Teorema de Bayes

Um dos melhores amigos do cientista de dados é o Teorema de Bayes, o qual é uma maneira de "reverter" as probabilidades condicionais. Digamos que precisamos saber a probabilidade de algum evento E ser condicionado à ocorrência de outro evento F. Mas apenas temos a informação sobre a probabilidade da ocorrência de F sendo condicionado a E. Usando a definição de probabilidade condicional duas vezes, podemos dizer que:

$$P(E|F) = P(E,F)/P(F) = P(F|E)P(E)/P(F)$$

O evento F pode ser dividido em dois eventos mutuamente exclusivos "F e E" e "F e não E". Se escrevermos E para "não E" (por exemplo, "E não acontece"), logo:

$$P(F) = P(F,E) + P(F,\neg E)$$

então temos:

$$P(E|F) = P(F|E)P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)]$$

que é como o Teorema de Bayes é estabelecido.

Esse teorema é usado com frequência para demonstrar porque os cientistas de dados são mais espertos do que médicos. imagine que uma determinada doença afete 1 a cada 10.000 pessoas. E imagine que haja um teste para essa doença que mostra o resultado correto ("doente" se você tem a doença e "não-doente" se não) 99% das vezes.

O que significa um teste positivo? Vamos usar T para o evento "seu teste é positivo" e D para o evento "você tem a doença". O Teorema de Bayes diz que a probabilidade de você ter a doença, condicional ao teste positivo, é:

$$P(D|T) = P(T|D)P(D)/[P(T|D)P(D) + P(T|\neg D)P(\neg D)]$$

Aqui vemos que P(T | D), a probabilidade de que alguém com a doença obtenha um teste positivo, é 0,99. P(D), a probabilidade de que qualquer pessoa tenha a doença é 1/10.000 = 0.0001. P (T|¬D), a probabilidade de que alguém sem a doença obtenha um teste positivo é 0,01. E P(¬D), a probabilidade de que qualquer pessoa não tenha a doença é 0,9999. Se você substituir esses números no Teorema de Bayes você encontrará

$$P(D|T) = 0.98\%$$

Ou seja, menos de 1% das pessoas que obtém um teste positivo realmente possuem a doença.

Isso presume que as pessoas fazem o teste de forma aleatória. Se apenas as pessoas que possuíssem alguns sintomas fizessem o teste, teríamos como condição o evento "teste positivo *e* sintomas" e o número teria a possibilidade de ser bem maior.

Enquanto esse é um cálculo simples para os cientistas de dados, a maioria dos médicos chutariam que P(D|T) seria perto de 2.

Uma forma mais intuitiva de ver isso é imaginar uma população de um milhão de pessoas. Você esperaria que 100 delas tivessem a doença, e que 99 dessas 100 obtivessem um teste positivo. Por outro lado, você esperaria que 999.900 delas não tivessem a doença, e que 9,999 delas obtivessem um teste positivo. O que significa que você esperaria que somente 99 de (99 + 9999) testes positivos realmente possuíssem a doença.

Variáveis Aleatórias

Uma *variável aleatória* é a variável cujos valores possíveis possuem uma distribuição de probabilidade associada. Uma variável aleatória bem simples é igual a 1 se um lançamento de moeda for cara e 0 se for coroa. Uma maneira mais complicada seria medir o número de caras observadas ao lançar a moeda dez vezes ou um valor escolhido de `range(10)`, no qual todos os números têm a mesma probabilidade.

A distribuição associada dá as probabilidades que a variável possui em cada um de seus valores possíveis. A variável do lançamento de moeda é igual a 0 com a probabilidade de 0,5 e 1 com a probabilidade de 0,5. A variável `range(10)` tem uma distribuição que atribui a probabilidade 0,1 para cada um dos números de 0 a 9.

Às vezes falaremos sobre o *valor esperado* da variável aleatória, o qual é a média de seus valores ponderados por suas probabilidades. A variável de lançamento da moeda tem um valor esperado de 1/2 (= 0 * 1/2 + 1 * 1/2), e a variável range(10) tem um valor esperado de 4,5.

As variáveis aleatórias podem ser condicionadas a eventos assim como outros eventos. Voltando ao exemplo das duas crianças da "Probabilidade Condicional" na página 70, se X for a variável randômica representando o número de meninas, X é igual a 0 com probabilidade de ¼, 1 com probabilidade de 1/2 e 2 com probabilidade de ¼.

Podemos definir uma nova variável Y que diz o número de meninas condicionado a, pelo menos, uma das crianças ser uma menina. Logo, Y é igual a 1 com a probabilidade de 2/3 e 2 com probabilidade de 1/3. A variável Z é o número de meninas que é condicionado ao filho mais velho sendo uma menina igual a 1 com probabilidade de 1/2 e 2 com probabilidade de 1/2.

Na maioria das vezes, usaremos as variáveis aleatórias implícitas ao que fazemos sem chamar a atenção para elas. Mas, se você olhar mais atentamente, você as verá.

Distribuições Contínuas

Um lançamento de moeda corresponde a uma *distribuição discreta* — uma que associa probabilidade positiva com resultados discretos. Frequentemente, vamos querer modelar as distribuições por meio de um contínuo de resultados. (Para nossos propósitos, esses resultados sempre serão números reais, embora não seja o caso na vida real.) Por exemplo, a *distribuição uniforme* coloca *peso igual* em todos os números entre 0 e 1.

Como existem infinitos números entre 0 e 1, isso significa que o peso que ele atribui aos pontos individuais precisa ser exatamente 0. Por esse motivo, representamos uma distribuição contínua com uma *função de densidade de probabilidade* (pdf, do inglês *probability density function*) tal que a probabilidade de ver um valor em um determinado intervalo é igual à integral da função de densidade sobre o intervalo.

Se seu cálculo integral estiver enferrujado, a melhor maneira de entender isso é se a distribuição tem a função de densidade f, logo a probabilidade de ver um valor entre x e $x + h$ é aproximadamente $h * f(x)$ se h for pequeno.

A função de densidade para a distribuição uniforme é:

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

A probabilidade de um valor aleatório seguido de distribuição estar entre 0,2 e 0,3 é 1/10, como era de se esperar. `random.random()` de Python é uma variável (pseudo) aleatória com uma densidade uniforme.

Estaremos frequentemente mais interessados na *função de distribuição cumulativa* (cdf, do inglês *cumulative distribution function*) que fornece a probabilidade de uma variável aleatória ser menor ou igual a um determinado valor. Não é difícil criar uma função de distribuição cumulativa para a distribuição uniforme (Figura 6-1):

```
def uniform_cdf(x):
    "retorna a probabilidade de uma variável aleatória uniforme ser <= x"
    if x < 0:   return 0    # a aleatória uniforme nunca é menor do que 0
    elif x < 1: return x    # por exemplo P(X <= 0.4) = 0.4
    else:       return 1    # a aleatória uniforme sempre é menor do que 1
```

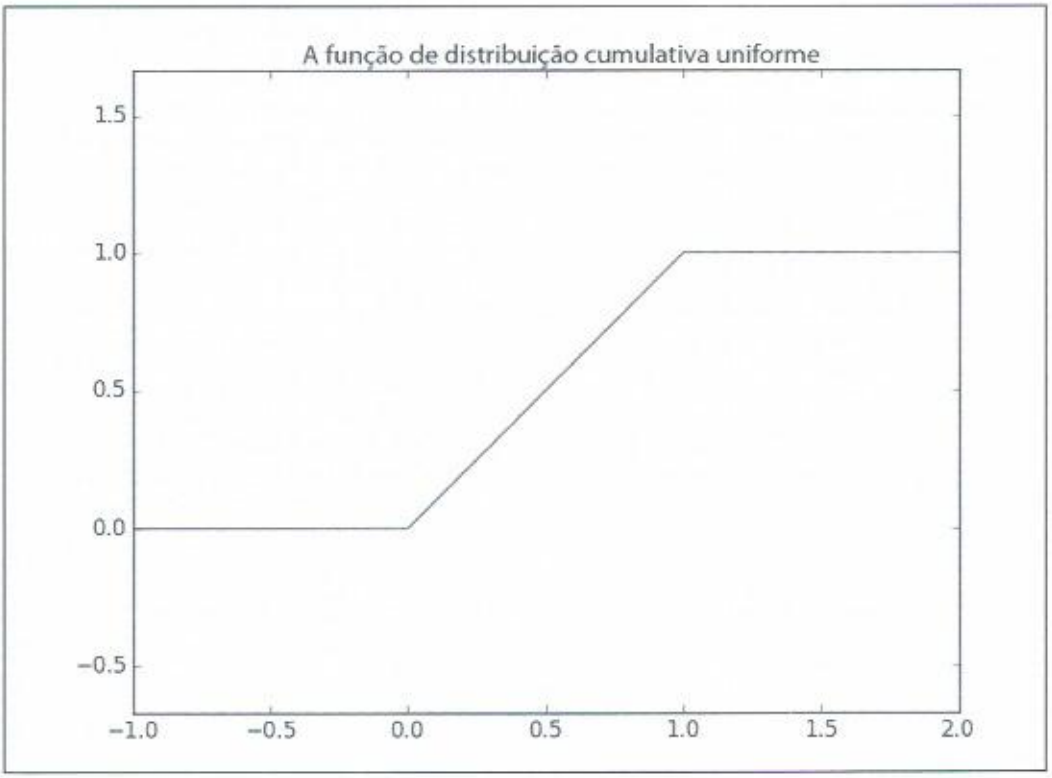

Figura 6-1. A função de distribuição cumulativa uniforme

A Distribuição Normal

A distribuição normal é a rainha das distribuições. É uma clássica distribuição de curva em forma de sino e é determinada por dois parâmetros: sua média μ (mi) e o desvio padrão σ (sigma). A média indica onde o sino é centralizado e o desvio padrão indica a largura do sino.

Ela possui a função de distribuição:

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

que podemos implementar como:

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

Na Figura 6-2, analisamos algumas dessas funções de densidade de probabilidade para ver como eles ficam:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Diversas Funções de Densidade de Probabilidade Normais")
plt.show()
```

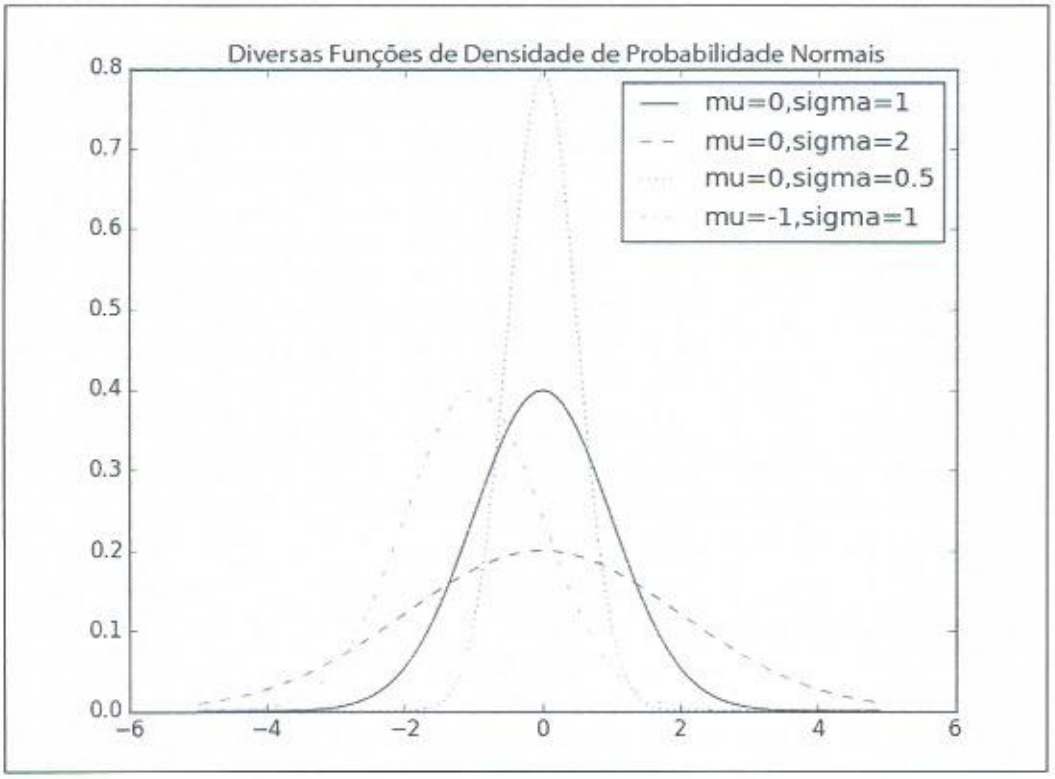

Figura 6-2. Diversas funções de densidade de probabilidade normais

É chamada de *distribuição normal padrão* quando $\mu = 0$ e $\sigma = 1$. Se Z é uma variável aleatória normal padrão, então:

$$X = \sigma Z + \mu$$

também é normal mas com a média μ e o desvio padrão σ. Por outro lado, se X é uma variável aleatória normal com média μ e desvio padrão σ,

$$Z = (X - \mu)/\sigma$$

é uma variável normal padrão.

A função de distribuição cumulativa para a distribuição normal não pode ser escrita de maneira "elementar", mas podemos escrever usando `math.erf` (*http://en.wikipedia.org/wiki/Error_function*) do Python:

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Novamente, na Figura 6-3, vemos alguns:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Diversas Funções de Densidade de Distribuição Cumulativa")
plt.show()
```

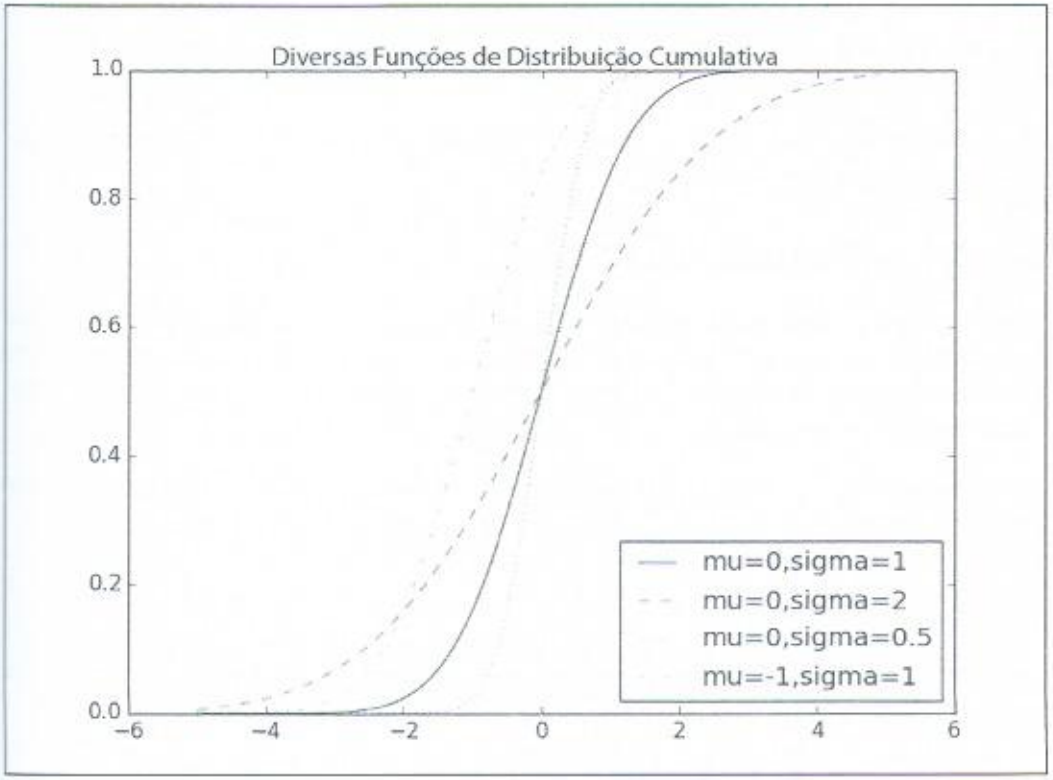

Figura 6-3. Diversas funções de distribuição cumulativa

Algumas vezes teremos que inverter `normal_cdf` para encontrar o valor correspondente à probabilidade especificada. Não existe uma forma simples de computar esse inverso, mas `normal_cdf` é contínuo e em crescimento, portanto podemos usar uma busca binária (*http://en.wikipedia.org/wiki/Binary_search_algorithm*):

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """encontra o inverso mais próximo usando a busca binária"""

    # se não for padrão, computa o padrão e redimensiona
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z, low_p = -10.0, 0            # normal_cdf(-10) está (muito perto de) 0
    hi_z,  hi_p  =  10.0, 1            # normal_cdf(10) está (muito perto de) 1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2     # considera o ponto do meio e o valor da
        mid_p = normal_cdf(mid_z)      # função de distribuição cumulativa lá
        if mid_p < p:
            # o ponto do meio ainda está baixo, procura acima
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            # o ponto do meio ainda está alto, procura abaixo
            hi_z, hi_p = mid_z, mid_p
        else:
            break

    return mid_z
```

A função divide em dois intervalos repetidamente até diminuir para um Z próximo o suficiente da probabilidade desejada.

O Teorema do Limite Central

Um motivo para a distribuição normal ser tão útil é o *teorema do limite central*, que diz que (em essência) uma variável aleatória definida como a média de uma grande quantidade de variáveis aleatórias distribuídas independente e identicamente é ela mesma aproximadamente distribuída normalmente.

Em especial, se x1,...,xn são variáveis aleatórias com média μ e desvio padrão σ, e se n for grande, então:

$$\frac{1}{n}(x_1 + \ldots + x_n)$$

está aproximadamente distribuída normalmente com a média μ e o desvio padrão $\sigma\sqrt{n}$. Da mesma forma (mas bem mais útil),

$$\frac{(x_1 + \ldots + x_n) - \mu n}{\sigma\sqrt{n}}$$

está aproximadamente distribuída normalmente com média 0 e desvio padrão 1.

Uma maneira simples de ilustrar isso é considerando as variáveis aleatórias *binomiais*, as quais possuem dois parâmetros n e p. Uma variável aleatória Binomial(n,p) é apenas a soma de n variáveis aleatórias independentes Bernoulli(p), e cada uma delas é igual a 1 com probabilidade p e 0 com probabilidade 1− p:

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0

def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

A média de uma variável Bernoulli(p) é p, e seu desvio padrão é $\sqrt{p(1-p)}$. O teorema do limite central diz que, conforme n aumenta, a variável Binomial(n,p) é aproximadamente uma variável aleatória normal com a média μ = np e desvio padrão $\sigma = \sqrt{np(1-p)}$. Se analisarmos os dois, pode-se ver claramente a semelhança:

```
def make_hist(p, n, num_points):

    data = [binomial(n, p) for _ in range(num_points)]

    # usa um gráfico de barras para exibir as amostrar binomiais atuais
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # usa um gráfico de linhas para exibir uma aproximação da normal
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("Distribuição Binomial vs. Aproximação Normal"  )
    plt.show()
```

Por exemplo, quando você chama `make_hist(0.75, 100, 10000)`, você obtém o gráfico da Figura 6-4.

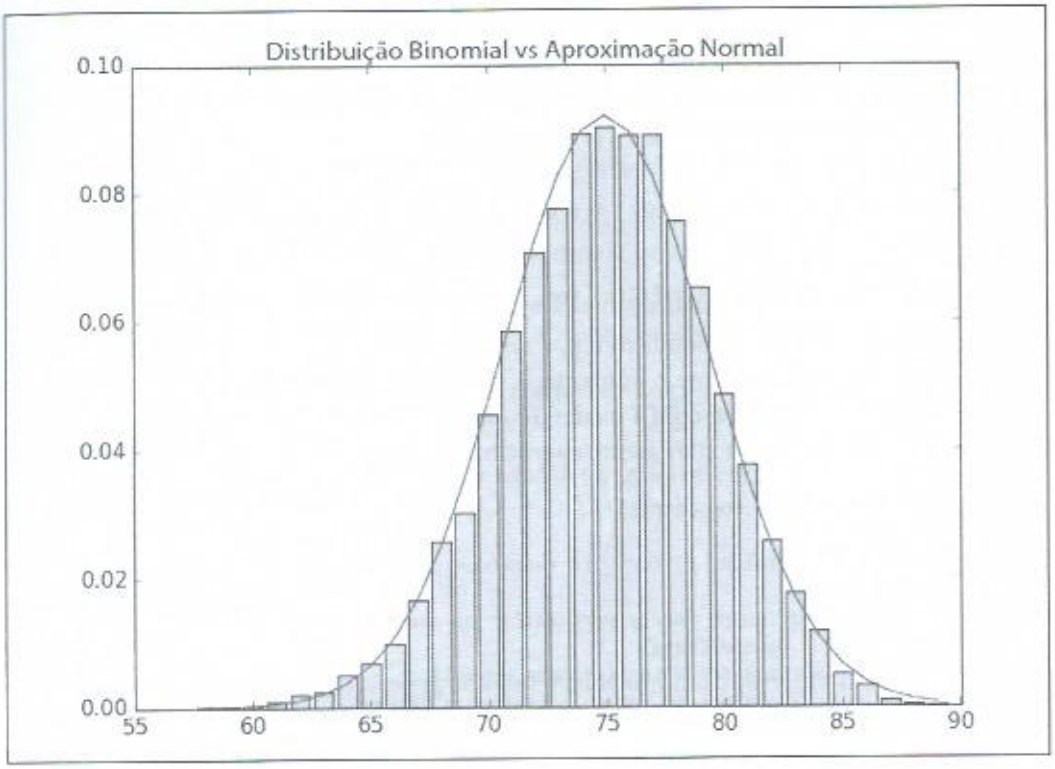

Figura 6-4. A saída de `make_hist`

O principal dessa aproximação é que se você quiser saber a probabilidade de (digamos) uma moeda honesta cair 60 coroas em 100 lançamentos, você pode estimar a probabilidade de uma Normal(50,5) ser maior que 60, o que é mais fácil do que computar a função de distribuição cumulativa Binomial(100,0.5). (Embora na maioria das aplicações é possível usar um software estatístico que computa quaisquer probabilidades que você quiser.)

Para Mais Esclarecimentos

- scipy.stats (*http://bit.ly/1L2H0Lj*) contém as funções de distribuição cumulativa e de densidade de probabilidade para a maioria das distribuições de probabilidade populares.
- Lembre-se como, no final do Capítulo 5, eu disse que seria uma boa ideia estudar com um livro didático de estatística? Também seria uma boa ideia estudar com um livro didático de probabilidade. O melhor que eu conheço e está disponível online é o *Introduction to Probability (http://bit.ly/1L2MTYI)*.

CAPÍTULO 7

Hipótese e Inferência

É característica de uma pessoa realmente inteligente ser movida pela estatística.
—George Bernard Shaw

O que faremos com todas essas teorias de estatística e probabilidade? A parte *ciência* de data science frequentemente envolve formar e testar *hipóteses* sobre nossos dados e os processos que os geram.

Teste Estatístico de Hipótese

Com frequência, como cientistas de dados, vamos querer testar se uma determinada hipótese é verdadeira. Para os nossos propósitos, as hipóteses são afirmações como "esta é uma moeda honesta" ou "os cientistas de dados preferem Python a R" ou "seria mais provável que as pessoas saíssem da página sem ler o conteúdo se nós mostrássemos um anúncio pop-up com um botão de fechar pequeno e difícil de encontrar" que possam ser traduzidas em estatísticas sobre dados. Sob diversas premissas, tais estatísticas podem ser vistas como observações de variáveis aleatórias a partir de distribuições conhecidas, o que permite que façamos declarações sobre as premissas mais prováveis de serem corretas.

Em uma configuração clássica, temos a *hipótese nula* H_0 que representa uma posição padrão, e alguma hipótese H_1 com a qual gostaríamos de compará-la. Usamos a estatística para decidir se rejeitamos H_0 como falso ou não. Provavelmente, fará mais sentido por meio de um exemplo.

Exemplo: Lançar Uma Moeda

Imagine que temos uma moeda e queremos testar para confirmar se ela é honesta. Temos a premissa de que a moeda possui a probabilidade p de cair cara, então nossa hipótese nula é

que a moeda seja honesta — ou seja, que $p = 0{,}5$. Testaremos novamente contra a hipótese alternativa $p \neq 0{,}5$.

Em especial, nosso teste envolverá o lançamento da moeda em número n de vezes e contando o número de caras X. Cada lançamento da moeda é uma Tentativa de Bernoulli, o que significa que X é uma variável aleatória Binomial(n, p), que (como vimos no Capítulo 6) podemos aproximar usando a distribuição normal:

```
def normal_approximation_to_binomial(n, p):
    """encontra mi e sigma correspondendo ao Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Sempre que uma variável aleatória segue uma distribuição normal, podemos usar `normal_cdf` para descobrir a probabilidade dos seus valores resultantes serem internos (ou externos) em um intervalo especial:

```
# o cdf normal é a probabilidade que a variável esteja abaixo de um limite
normal_probability_below = normal_cdf

# está acima do limite se não estiver abaixo
def normal_probability_above(lo, mu=0, sigma=1):
    return 1 - normal_cdf(lo, mu, sigma)

# está entre se for menos do que hi, mas não menor do que lo
def normal_probability_between(lo, hi, mu=0, sigma=1):
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# está fora se não estiver entre
def normal_probability_outside(lo, hi, mu=0, sigma=1):
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

Também podemos fazer o contrário — encontrar a região sem aba ou o intervalo (simétrico) em torno da média que contribui para o nível de probabilidade. Por exemplo, se quisermos encontrar um intervalo centrado na média contendo 60% de probabilidade, então encontraremos os cortes onde as abas inferiores e superiores contêm 20% de probabilidade cada (deixando 60%):

```
def normal_upper_bound(probability, mu=0, sigma=1):
    """retorna z para que p(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
    """retorna z para que p(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
    """retorna os limites simétricos (sobre a média)
    que contêm a probabilidade específica"""
    tail_probability = (1 - probability) / 2
```

```
    # limite superior deveria ter tail_probability acima
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # limite inferior deveria ter tail_probability abaixo
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

Em especial, digamos que escolhemos lançar uma moeda $n = 1000$ vezes. Se nossa hipótese de honestidade for verdadeira, X deveria ser distribuído normalmente com média 500 e desvio padrão de 15,8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

Precisamos tomar uma decisão sobre *significância* — de quanto é a vontade que temos de fazer um *erro tipo 1* ("falso positivo"), em que rejeitamos H_0 mesmo se for verdadeiro. Por motivos perdidos pelas memórias da história, essa vontade é configurada para 5% ou 1%, geralmente. Vamos escolher 5%.

Considere o teste que rejeita H_0 se X cair fora dos limites dados por:

```
normal_two_sided_bounds(0.95, mu_0, sigma_0)   # (469, 531)
```

Presumindo que p seja igual a 0,5 (por exemplo, H_0 é verdadeiro), há apenas 5% de chance de observarmos que um X permanece fora desse intervalo, pois é exatamente a significância que queríamos. De outra forma, se H_0 for verdadeiro, esse teste apresentará o resultado correto aproximadamente em 19 de 20 vezes.

Também estamos interessados no *poder* de um teste, que é a probabilidade de não cometer um *erro tipo 2*, no qual falhamos em rejeitar H_0 mesmo ele sendo falso. A fim de medir esse procedimento, temos que especificar o que *realmente* significa H_0 ser falso. (Sabendo ao menos que p *não* é 0,5 não lhe dá uma informação significativa sobre a distribuição de X.) Em especial, verificaremos o que acontece se p realmente for 0,55, a fim de que a moeda esteja levemente inclinada a ser cara.

Nesse caso, podemos calcular o poder do teste com:

```
# 95% dos limites baseados na premissa p é 0,5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# mi e sigma reais baseados em p = 0,55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# um erro tipo 2 significa que falhamos ao rejeitar a hipótese nula
# que acontecerá quando X ainda estiver em nosso intervalo original
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability       # 0.887
```

Agora, imagine que a nossa hipótese nula fosse que a moeda não seria inclinada a cara, ou que $p \leq 0{,}5$. Nesse caso, queríamos um *teste unilateral* que rejeitasse a hipótese nula quando X fosse muito maior que 50 mas não quando X fosse menor. Portanto, um teste de signifi-

cância de 5% envolveria usar `normal_probability_below` para encontrar o corte abaixo dos 95% em que a probabilidade ficaria:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# é 526 (< 531, já que precisamos de mais probabilidade na aba superior)

type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.936
```

Esse teste é mais poderoso, visto que ele não mais rejeita H_0 quando X está abaixo de 469 (improvável de acontecer se H_1 for verdadeiro) e, ao invés, rejeita H_0 quando X está entre 526 e 531 (provável de acontecer se H_1 for verdadeiro).

p-values

Uma outra maneira de pensar sobre o teste anterior envolve *p-values*. Em vez de escolher limites com base em alguma probabilidade de corte, nós computamos a probabilidade — presumindo que H_0 seja verdadeiro — que podemos ver um valor ao menos tão extremo quanto ao que realmente observamos.

Para o nosso teste bilateral para a moeda honesta, computamos:

```
def two_sided_p_value(x, mu=0, sigma=1):
    if x >= mu:
        # se x for maior do que a média, a coroa será o que for maior do que x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # se x for menor do que a média, a coroa será o que for menor do que x
        return 2 * normal_probability_below(x, mu, sigma)
```

Se víssemos 530 caras, computaríamos:

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

Por que usamos 529,5 em vez de 530? Isso é o que chamamos de *correção de continuidade (http://en.wikipedia.org/wiki/Continuity_correction)*. Reflete o fato de que `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` é a melhor estimativa da probabilidade de ver 530 caras do que `normal_probability_between(530, 531, mu_0, sigma_0)` seria.

Desta forma, `normal_probability_above(529.5, mu_0, sigma_0)` é a melhor estimativa da probabilidade de ver ao menos 530 caras. Você deve ter notado que também usamos isso no código produzido na Figura 6-4.

Uma forma de se convencer da relevância dessa estimativa é por meio de uma simulação:

```
extreme_value_count = 0
for _ in range(100000):
    num_heads = sum(1 if random.random() < 0.5 else 0     # contagem do # de caras
                    for _ in range(1000))                 # em 1000 lançamentos
```

```
    if num_heads >= 530 or num_heads <= 470:            # e contagem da frequência
        extreme_value_count += 1                        # que # é 'extrema'

print extreme_value_count / 100000   # 0.062
```

Desde que p-value seja maior do que a significância de 5%, não rejeitamos a hipótese nula. Se víssemos 532 caras, o p-value seria:

```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

que é menor do que a significância de 5%, logo, rejeitaríamos a hipótese nula. É exatamente o mesmo teste de antes. É apenas uma forma diferente de abordar a estatística.

Da mesma maneira, teríamos:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

Se nós víssemos 525 caras para o teste unilateral, computaríamos:

```
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

mostrando que não rejeitaríamos a hipótese nula. Se nós víssemos 527 caras, a computação seria:

```
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

e nós rejeitaríamos a hipótese nula.

Certifique-se de que seu dado está distribuído normalmente antes de usar `normal_probability_above` para computar p-values. As memórias ruins de data science estão repletas de exemplos de pessoas opinando que a chance de algum evento observado ocorrer aleatoriamente é uma em um milhão, quando o que eles realmente querem dizer é "a chance, presumindo que o dado seja distribuído normalmente" e é bem inútil se o dado não o for.

Existem diversos testes estatísticos para a normalidade, no entanto, até mesmo a elaboração de gráfico dos dados é uma boa ideia.

Intervalos de Confiança

Temos testado hipóteses sobre o valor da probabilidade p, do resultado cara que é um *parâmetro* da desconhecida distribuição "cara". Quando o caso é esse, uma terceira abordagem é construir um *intervalo de confiança* em torno do valor observado do parâmetro.

Por exemplo, podemos estimar a probabilidade de uma moeda viciada ao analisar o valor médio das variáveis Bernoulli correspondentes a cada lançamento — 1 se cara, 0 se coroa. Se nós observarmos 525 caras de 1000 lançamentos, logo estimamos p em 0,525.

Quão *confiantes* podemos ser nessa estimativa? Bem, se soubéssemos o valor exato de p, o teorema de limite central (lembre-se de "O Teorema do Limite Central" na página 78) nos diz que a média daquelas variáveis Bernoulli deveriam ser quase normais, com a média p e desvio padrão:

```
math.sqrt(p * (1 - p) / 1000)
```

Aqui não conhecemos p, portanto usamos nossa estimativa:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

Isso não é inteiramente comprovado, mas as pessoas parecem fazê-lo de qualquer forma. Ao usar a aproximação normal, concluímos que somos "95% confiantes" de que parâmetro seguinte contém o verdadeiro parâmetro p:

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

Essa declaração é sobre o *intervalo* e não sobre p. Você deveria entender como uma premissa se fosse repetir o experimento muitas vezes, 95% das vezes o parâmetro "verdadeiro" (que é o mesmo todas as vezes) ficaria dentro do intervalo de confiança observada (que pode ser diferente a cada vez).

Em especial não concluímos que a moeda seja viciada, já que 0,5 cai dentro de nosso intervalo de confiança.

Se, em vez disso, tivéssemos visto 540 caras, teríamos:

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Aqui, a "moeda honesta" não fica no intervalo de confiança. (A hipótese da "moeda honesta" não passaria no teste esperado 95% das vezes se fosse verdade.)

P-Hacking

Um procedimento que rejeita a hipótese nula erroneamente somente 5% das vezes vai — por definição — rejeitar erroneamente 5% das vezes a hipótese nula:

```
def run_experiment():
    """lança uma moeda 1000 vezes, True = cara, False = coroa"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
    """usando 5% dos níveis de significância"""
    num_heads = len([flip for flip in experiment if flip])
```

```
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

print num_rejections   # 46
```

O que isso quer dizer é que você está tentando encontrar resultados "significativos" e geralmente você consegue. Teste hipóteses suficientes contra o seu conjunto de dados e um deles, certamente, parecerá significante. Remova os valores discrepantes certos, e será provável conseguir seu p-value abaixo de 0,05. (Fizemos algo vagamente parecido em "Correlação" na página 62; percebeu?)

Isto é, por reter, chamado P-hacking (*http://bit.ly/1L2QtCr*), e é, de certa forma, uma consequência da "inferência a partir da estrutura dos p-values". Um artigo ótimo criticando essa abordagem é "The Earth is Round" (*http://bit.ly/1L2QJ4a*).

Se você quer praticar uma boa *ciência*, você deveria determinar suas hipóteses antes de verificar os dados, deveria limpar seus dados sem pensar nas hipóteses, e deveria ter em mente que p-values não são substitutos para o senso comum. (Uma abordagem alternativa seria "Inferência Bayesiana" na página 88.)

Exemplo: Executando um Teste A/B

Uma de nossas responsabilidades iniciais na DataSciencester é testar a otimização, um eufemismo para tentar fazer com que as pessoas cliquem nos anúncios. Um de seus anunciantes desenvolveu uma bebida energética voltada para os cientistas de dados, e o vice-presidente de Publicidade quer a sua ajuda para escolher entre a propaganda A ("bom sabor!") ou propaganda B ("menos polarização!").

Por ser um *cientista*, você decide executar um *experimento* mostrando aos visitantes do site uma das duas propagandas e registrando quantas pessoas clicam em cada um.

Se 990 de 1000 visualizadores do anúncio A clicam na propaganda enquanto que 10 de 1000 visualizadores do B clicam, você pode ficar confiante de que A é melhor do que B. Mas e se as diferenças não são tão graves? Aqui é onde você usaria inferência estatística.

Digamos que pessoas N_A vejam o anúncio A, e que n_A cliquem nele. Podemos pensar em cada visualização do anúncio como uma Tentativa de Bernoulli em que p_A é a probabilidade de alguém clicar no anúncio A. Então (se N_A for grande, e é aqui) sabemos que n_A/N_A é aproximadamente uma variável aleatória com média p_A e desvio padrão de $\sigma_A = \sqrt{p_A(1 - p_A)/N_A}$.

Igualmente, n_B/N_B é aproximadamente uma variável aleatória com média p_B e desvio padrão de $\sigma_B = \sqrt{p_B(1 - p_B)/N_B}$:

```
def estimated_parameters(N, n):
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

Se presumirmos que as duas normais são independentes (parece razoável, já que a Tentativa de Bernoulli deveria ser), então suas diferenças também deveriam ser normais com a média $p_B - p_A$ e o desvio padrão $\sqrt{\sigma_A^2 + \sigma_B^2}$.

É quase uma trapaça. A matemática só funciona exatamente desse jeito se você *conhece* os desvios padrões. Aqui, estamos estimando-os a partir dos dados, o que significa que realmente deveríamos usar a distribuição *t*. Mas para conjuntos de dados suficientemente grandes não faz tanta diferença.

Isso significa que podemos testar a *hipótese nula* que p_A e p_B são a mesma (ou seja, que $p_B - p_A$ é zero) usando a estatística:

```
def a_b_test_statistic(N_A, n_A, N_B, n_B):
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

que deveria ser uma normal padrão.

Por exemplo, se "bom sabor" recebe 200 cliques de 1000 visualizações e "menos polarização" recebe 180 cliques de 1000 visualizações, a estatística é esta:

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

A probabilidade de ver tal diferença se a média fosse realmente igual seria:

```
two_sided_p_value(z)                            # 0.254
```

que é grande o suficiente e você não consegue saber se tal diferença existe. Por outro lado, se "menos polarização" recebesse somente 150 cliques, teríamos:

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

que significa que há somente a probabilidade de 0,0003 que você veria tal diferença se as propagandas fossem igualmente eficazes.

Inferência Bayesiana

Os procedimentos que vimos se dedicaram a fazer as declarações de probabilidade sobre nossos *testes*: "há apenas uma chance de 3% de você ter observado tal estatística extrema se nossa hipótese nula fosse verdadeira".

Uma abordagem alternativa para a inferência envolve tratar os parâmetros desconhecidos como variáveis aleatórias. O analista (ou seja, você) começa com uma *distribuição anterior* (*a priori*) para os parâmetros e usa os dados observados e o Teorema de Bayes para receber uma atualização da *distribuição posterior* (*a posteriori*) para os parâmetros. Em vez de julgar a probabilidade sobre os testes, julgue a probabilidade sobre os próprios parâmetros.

Por exemplo, quando o parâmetro desconhecido é uma probabilidade (como no nosso exemplo de lançamento de moeda), frequentemente usamos uma anterior a partir da *distribuição Beta*, colocando todas as probabilidades entre 0 e 1:

```
def B(alpha, beta):
    """uma constante normalizada para que a probabilidade total seja 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x, alpha, beta):
    if x < 0 or x > 1:          # sem peso fora de [0, 1]
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

Em geral, essa distribuição centraliza seu peso em:

```
alpha / (alpha + beta)
```

e quanto maiores os `alpha` e `beta` são, mais "estreita" é a distribuição.

Por exemplo, se `alpha` e `beta` forem 1, é apenas a distribuição uniforme (centrada em 0,5, muito dispersa). Se `alpha` for muito maior do que `beta`, a maioria do peso fica perto de 1. E, se `alpha` for muito menor do que `beta`, a maioria do peso fica perto de 0. A Figura 7-1 mostra várias distribuições Betas diferentes.

Então, digamos que presumimos uma distribuição anterior em p. Talvez não queremos tomar uma posição se a moeda for honesta e nós escolhermos `alpha` e `beta` para ambas serem 1. Ou, talvez, tenhamos uma forte certeza de que dará cara 55% das vezes, e escolhemos `alpha` igual a 55, `beta` igual a 45.

Lançamos nossa moeda muitas vezes e vemos h para *heads* (cara) e t para *tails* (coroa). O Teorema de Bayes (e um pouco de matemática que é muito entediante para eu tocar nesse assunto) nos diz que a distribuição posterior para p é novamente uma distribuição beta mas com parâmetros `alpha + h` e `beta + t`.

Não é coincidência que a distribuição posterior seja novamente uma distribuição beta. O número de caras é fornecido pela distribuição binomial, e a Beta é *conjugada anterior (http://www.johndcook.com/blog/conjugate_prior_diagram/)* dela. Isso significa que a qualquer momento que você atualizar uma Beta anterior usando observações a partir do correspondente binomial, você receberá uma Beta posterior.

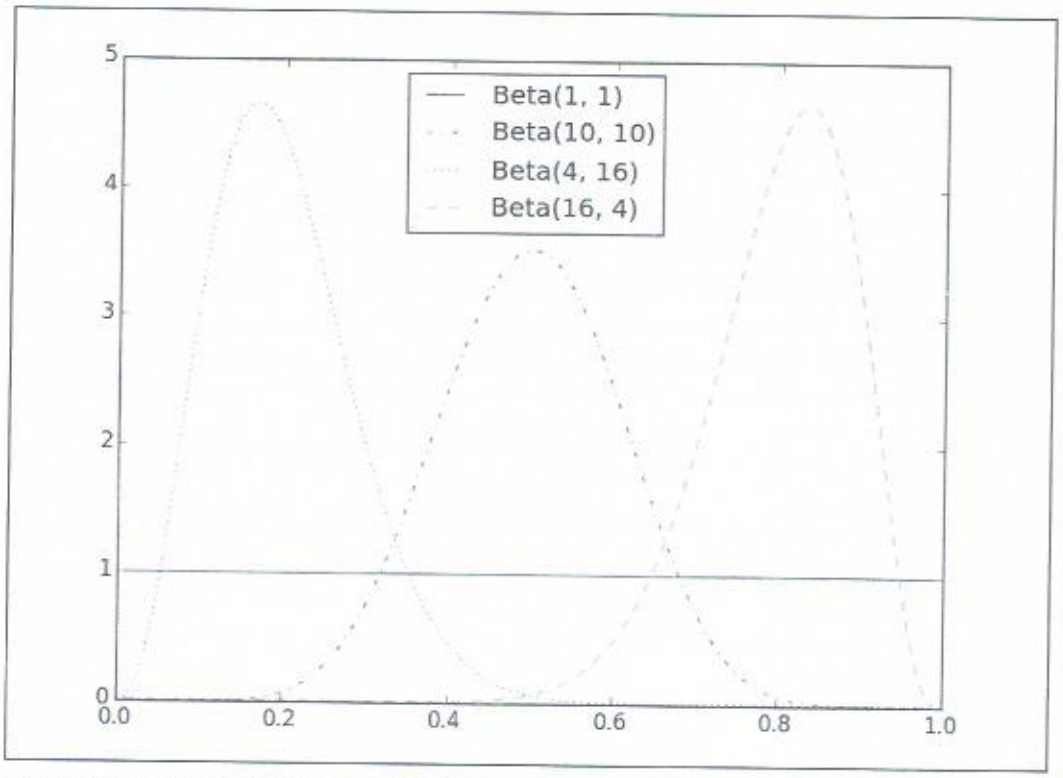

Figura 7-1. Exemplos de distribuições Beta

Digamos que você lance a moeda 10 vezes mas só veja 3 caras.

Se você tivesse começado com uma distribuição anterior uniforme (de certa forma se recusando a tomar uma posição sobre a honestidade da moeda), sua distribuição posterior seria uma Beta(4, 8), centrada próximo de 0,33. Já que você considerou todas as probabilidades igualmente possíveis, seu melhor palpite é algo bem perto da probabilidade observada.

Se você tivesse começado com um Beta(20, 20) (acreditando que a moeda era mais ou menos honesta), sua distribuição posterior seria um Beta(23, 27) centrada próximo de 0,46, indicando uma segurança que talvez a moeda seja levemente inclinada para coroa.

E se você começasse com um Beta(30, 10) (acreditando que a moeda estava inclinada a lançar cara em 75%), sua distribuição posterior seria de um Beta(33, 17), centrado próximo de 0,66. Nesse caso, você ainda acreditaria na inclinação para cara, mas menos do que acreditaria no início. Essas três distribuições posteriores diferentes estão exibidas na Figura 7-2.

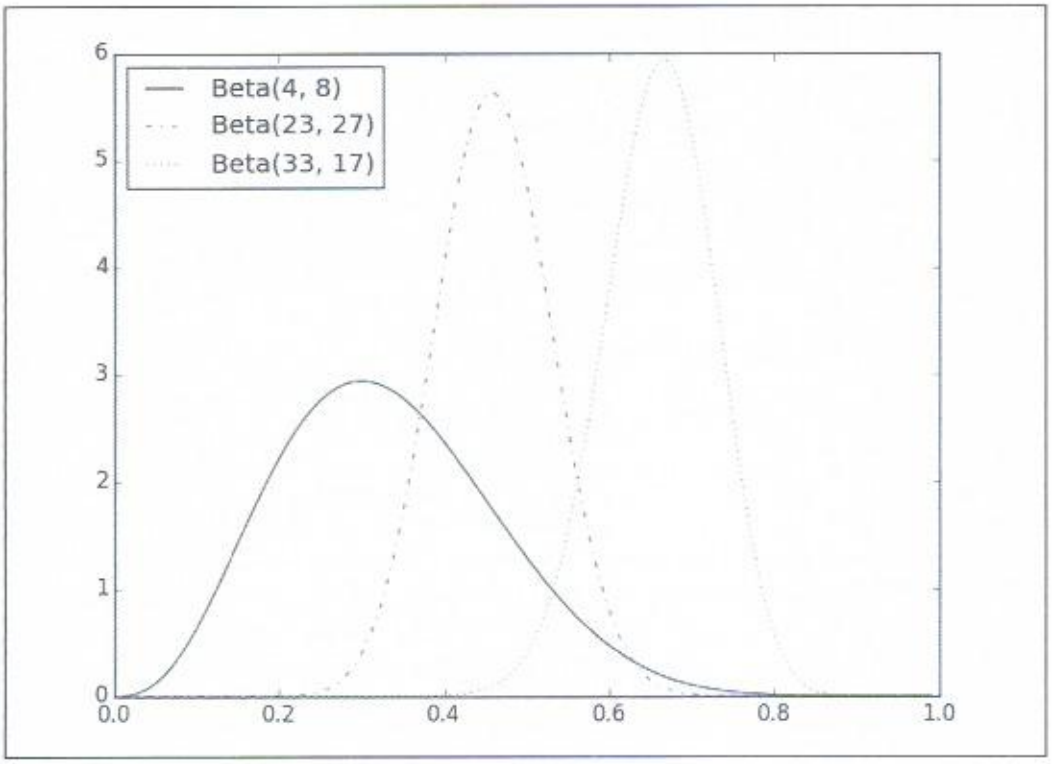

Figura 7-2. Posteriores surgindo de anteriores diferentes

Se você lançasse uma moeda mais e mais vezes, a anterior teria menos importância até eventualmente ter (quase) a mesma distribuição posterior, sem importar em qual anterior você começou.

Por exemplo, não importa a inclinação que você pensou que a moeda tinha, seria difícil acreditar nisso depois de ver 1000 caras de 2000 lançamentos (a menos que você seja um lunático que escolhe uma anterior tipo Beta(1000000,1)).

O interessante é que isso permite que façamos declarações de probabilidade sobre hipóteses: "baseado na anterior e nos dados observados, há apenas 5% de probabilidade que as caras da moeda estejam entre 49% e 51%". Filosoficamente, é muito diferente de uma declaração como "se a moeda fosse honesta, esperaríamos observar dados tão extremos somente 5% das vezes".

O uso da inferência Bayesiana para testar hipóteses é considerado um pouco controverso — em parte porque sua matemática pode se tornar complicada e, em parte, por causa da natureza subjetiva de se escolher uma anterior. Não usaremos isso em mais nenhum lugar deste livro, mas é bom saber sobre isso.

Para Mais Esclarecimentos

- Quase nem tocamos na superfície do que você deveria saber sobre inferência estatística. Os livros recomendados no final do Capítulo 5 entram em muito mais detalhes.
- A Coursera oferece o curso Análise de Dados e Inferência Estatística (em inglês) que aborda muitos desses tópicos.

CAPÍTULO 8

Gradiente Descendente

Aqueles que se gabam de seus descendentes, contam vantagem do que devem aos outros.
—Seneca

Frequentemente, ao praticar data science, tentamos encontrar o melhor modelo para uma determinada situação. E, geralmente, a "melhor" significará algo como "minimiza o erro do modelo" ou "maximiza a probabilidade do dado". Em outras palavras, representa a solução para algum tipo de problema de otimização.

Isso significa que precisaremos resolver uma quantidade de problemas de otimização. E, em especial, precisaremos resolvê-los do zero. Nossa abordagem será uma técnica chamada *gradiente descendente*, que se dispõe muito bem para um tratamento do zero. Você talvez não ache muito animador, mas ela nos permitirá fazer coisas empolgantes no decorrer do livro, portanto, tenha paciência.

A Ideia Por Trás do Gradiente Descendente

Suponha que tenhamos a função f que tem como entrada um vetor de números reais e exibe, como saída, um único número real. Tal função simples é:

```
def sum_of_squares(v):
    """computa a soma dos elementos ao quadrado em v"""
    return sum(v_i ** 2 for v_i in v)
```

Com frequência, precisaremos maximizar (ou minimizar) tais funções. Ou seja, precisamos encontrar a entrada v que produz o maior (ou menor) valor possível.

Para funções como a nossa, o *gradiente* (se você se lembra dos seus estudos de cálculo, ele é o vetor das derivadas parciais) mostra a direção da entrada em que a função cresce mais rapidamente. (Se você não se lembra dos seus estudos de cálculo, acredite em mim ou procure na internet.)

Igualmente, uma abordagem para maximizar uma função é pegar um ponto de início aleatório, computar o gradiente, andar um pequeno passo na direção do gradiente (por exemplo, a direção que faz com que a função cresça mais), e repetir com o novo ponto de início. Da mesma forma, você pode tentar minimizar uma função ao andar poucos passos na direção *oposta*, como mostra a Figura 8-1.

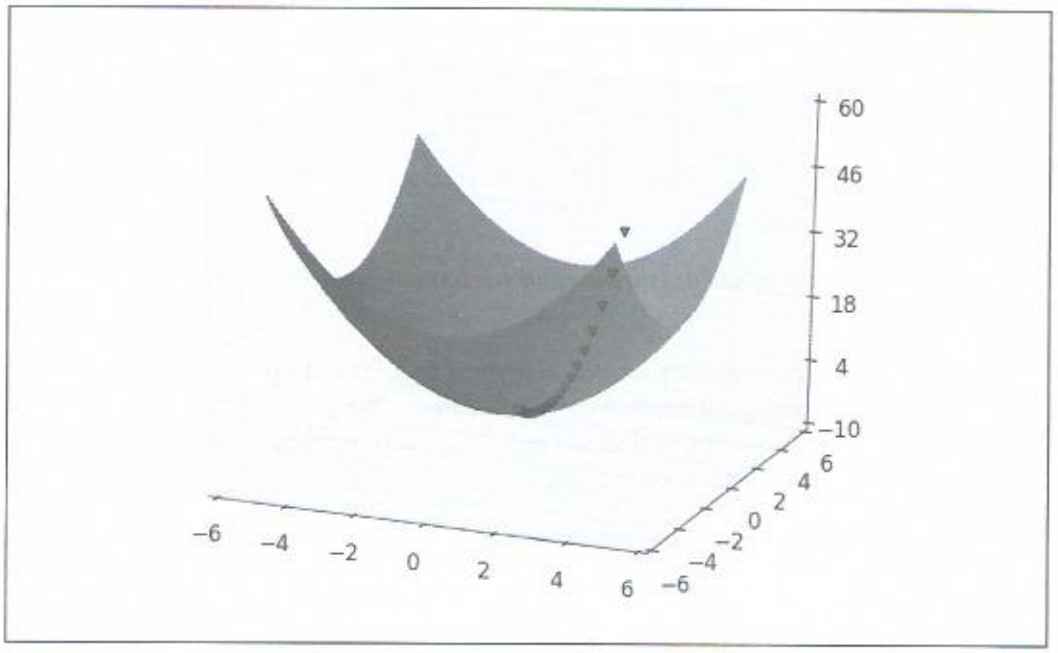

Figura 8-1. Encontrando uma mínima usando um gradiente descendente

Se uma função possui uma mínima global única, é provável que esse procedimento a encontre. Se uma função possui mínimas múltiplas (locais), esse procedimento talvez "encontre" a errada e, nesse caso, você talvez tenha que retomar o procedimento a partir de vários pontos de início. Se uma função não possui mínima, então é possível que o procedimento dure para sempre.

Estimando o Gradiente

Se `f` é uma função de uma variável, sua derivada em um ponto `x` indica como `f(x)` muda quando fazemos uma mudança bem pequena em `x`. É definida como o limite de quocientes diferenciais:

```
def difference_quotient(f, x, h):
    return (f(x + h) - f(x)) / h
```

conforme `h` se aproxima de zero.

(Muitos alunos de cálculo em potencial foram atrapalhados pela definição matemática de limite. Aqui vamos trapacear e simplesmente dizer que ela significa o que você acha que significa.)

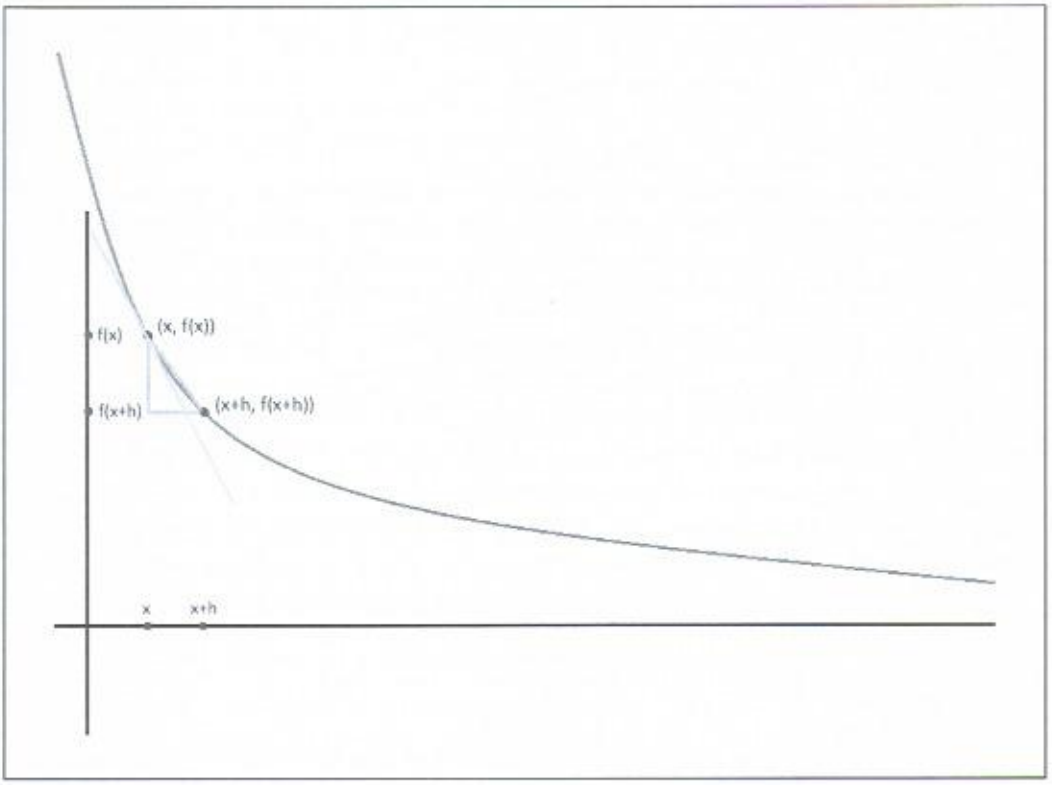

Figura 8-2. Aproximando uma derivada com um quociente diferencial

A derivada é a inclinação da linha tangente em *(x, f(x))*, enquanto o quociente diferencial é a inclinação da linha-não-tão-tangente que passa por *(x+h, f(x+h))*. Conforme *h* vai ficando menor, a linha-não-tão-tangente chega cada vez mais perto da linha tangente (Figura 8-2).

Para muitas funções é fácil calcular as derivadas com exatidão. Por exemplo, a função `square`:

```
def square(x):
    return x * x
```

tem a derivada:

```
def derivative(x):
    return 2 * x
```

que você pode verificar — se você estiver com vontade — ao explicitamente computar o quociente diferencial e tomando o limite.

E se você não pudesse (ou não quisesse) encontrar o gradiente? Embora não possamos ter limites com Python, podemos estimar derivadas ao avaliar o quociente diferencial por um pequeno e. A Figura 8-3 mostra os resultados para tal estimativa:

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)

# planeja mostrar que são basicamente o mesmo
import matplotlib.pyplot as plt
x = range(-10,10)
plt.title("Actual Derivatives vs. Estimates")
plt.plot(x, map(derivative, x), 'rx', label='Actual')            # vermelho x
plt.plot(x, map(derivative_estimate, x), 'b+', label='Estimate')  # azul +
plt.legend(loc=9)
plt.show()
```

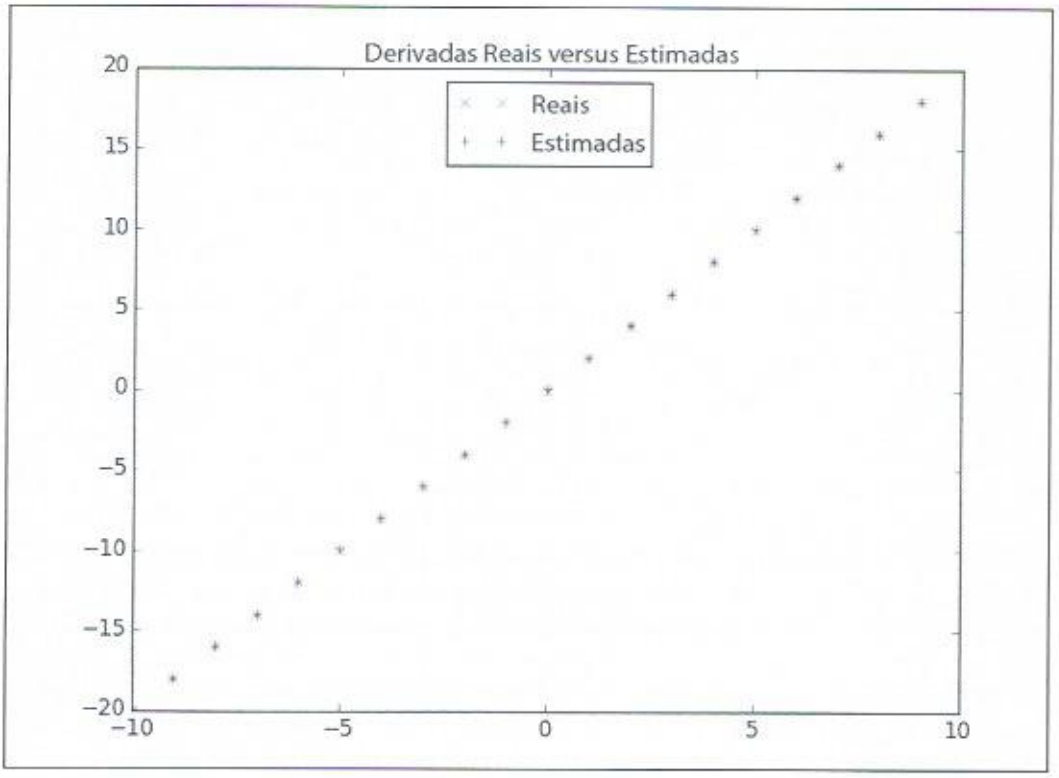

Figura 8-3. A bondade da aproximação do quociente diferencial

Quando `f` é uma função de muitas variáveis, possui múltiplas *derivadas parciais*, cada uma indicando como `f` muda quando fazemos pequenas mudanças em apenas uma das variáveis de entrada.

Calculamos sua derivada parcial *i*-ésimo ao tratá-la como uma função de apenas a *i*-ésima variável, contendo as outras variáveis fixas:

```
def partial_difference_quotient(f, v, i, h):
    """computa o i-ésimo quociente diferencial parcial de f em v"""
    w = [v_j + (h if j == i else 0)    # adiciona h ao elemento i-ésimo de v
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

depois do que podemos estimar o gradiente do mesmo jeito:

```
def estimate_gradient(f, v, h=0.00001):
    return [partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]
```

A maior desvantagem da abordagem "estimar usando os quocientes diferenciais" é sair caro em termos de computação. Se *v* tem o tamanho *n*, `estimate_gradient` tem que avaliar `f` em *2n* entradas diferentes. Se você está estimando gradientes um após o outro, está fazendo muito mais trabalho extra.

Usando o Gradiente

É fácil ver que a função `sum_of_squares` tem seu mínimo valor quando sua entrada `v` é um vetor de zeros. Mas imagine que não sabíamos disso. Usaremos os gradientes para encontrar o mínimo entre todos os vetores tridimensionais. Pegaremos um ponto inicial aleatório e andaremos pequenos passos na direção oposta do gradiente, até chegarmos em um ponto em que o gradiente seja muito pequeno:

```
def step(v, direction, step_size):
    """move step_size na direção a partir de v"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
    return [2 * v_i for v_i in v]

# escolhe um ponto inicial aleatório
v = [random.randint(-10,10) for i in range(3)]

tolerance = 0.0000001

while True:
    gradient = sum_of_squares_gradient(v)   # computa o gradiente em v
    next_v = step(v, gradient, -0.01)       # pega um passo gradiente negativo
    if distance(next_v, v) < tolerance:     # para se estivermos convergindo
        break
    v = next_v                              # continua se não estivermos
```

Se você codificar isso, saberá que ele sempre termina com um `v` muito próximo a `[0,0,0]`. Quanto menor for a `tolerance`, mais próximo ele será.

Escolhendo o Tamanho do Próximo Passo

Embora a lógica de se mover em direção ao gradiente esteja clara, a distância não está. De fato, escolher o tamanho do próximo passo é mais uma arte do que uma ciência. As opções mais populares são:

- Usar um passo de tamanho fixo
- Diminuir gradualmente o tamanho do passo a cada vez
- A cada passo, escolher o tamanho do passo que minimize o valor da função objetiva

A última opção parece perfeita mas é, na prática, uma computação custosa. Podemos aproximá-la ao tentar uma variedade de tamanhos de passos e escolher um que resulte no menor valor da função objetiva:

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

É possível que determinados tamanhos de passos resultarão em entradas inválidas para nossa função. Então, você precisará criar uma função "aplicação segura" que retorna infinito (que nunca deveria ser o mínimo de nada) para entradas inválidas:

```
def safe(f):
    """retorna uma nova função que é igual a f,
    exceto que ele exibe infinito como saída toda vez que f produz um erro"""
    def safe_f(*args, **kwargs):
        try:
            return f(*args, **kwargs)
        except:
            return float('inf')         # isso significa "infinito" em Python
    return safe_f
```

Juntando Tudo

No geral, temos alguma `target_fn` que queremos minimizar, e também temos o seu `gradient_fn`. Por exemplo, `target_fn` poderia representar erros em um modelo como uma função dos seus parâmetros, e talvez queiramos encontrar os parâmetros que produzem os menores erros possíveis.

Além do mais, digamos que escolhemos (de alguma forma) um valor inicial para os parâmetros `theta_0`. Logo, podemos implementar o gradiente descendente como:

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    """usa o gradiente descendente para encontrar theta que minimize a função alvo"""

    step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

    theta = theta_0                         # ajusta theta para o valor inicial
    target_fn = safe(target_fn)             # versão segura de target_fn
    value = target_fn(theta)                # valor que estamos minimizando

    while True:
        gradient = gradient_fn(theta)
        next_thetas = [step(theta, gradient, -step_size)
                       for step_size in step_sizes]
```

```
        # escolhe aquele que minimiza a função de erro
        next_theta = min(next_thetas, key=target_fn)
        next_value = target_fn(next_theta)

        # para se estivermos "convergindo"
        if abs(value - next_value) < tolerance:
            return theta
        else:
            theta, value = next_theta, next_value
```

Chamamos de `minimize_batch` porque, para cada passo do gradiente, ele considera o conjunto inteiro de dados (devido ao `target_fn` retornar o erro no conjunto de dados inteiro). Na próxima seção, veremos uma abordagem alternativa que considera apenas um ponto de cada vez.

Às vezes vamos querer *maximizar* uma função e podemos fazê-la ao minimizar seu negativo (que possui um gradiente negativo correspondente):

```
def negate(f):
    """retorna uma função que, para qualquer entrada, x retorna -f(x)"""
    return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
    """o mesmo quando f retorna uma lista de números"""
    return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)]

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    return minimize_batch(negate(target_fn),
                          negate_all(gradient_fn),
                          theta_0,
                          tolerance)
```

Gradiente Descendente Estocástico

Conforme mencionamos anteriormente, usaremos com frequência o gradiente descendente para escolher os parâmetros de um modelo de modo que minimize alguma noção de erro. Ao usar o grupo de abordagens anteriores, cada passo gradiente requer que nós façamos uma previsão e computemos o gradiente para o conjunto de dados inteiro, fazendo com que cada passo leve mais tempo.

Normalmente, essas funções de erro são *aditivas*, o que significa que o erro previsto no conjunto de dados inteiro é simplesmente a soma dos erros preditivos para cada ponto.

Quando o caso é esse, podemos aplicar uma técnica chamada *gradiente descendente estocástico*, que computa o gradiente (e anda um passo) apenas um ponto de cada vez. Ele circula sobre nossos dados repetidamente até alcançar um ponto de parada.

Durante cada ciclo, vamos querer iterar sobre nossos dados em ordem aleatória:

```
def in_random_order(data):
    """gerador retorna os elementos do dado em ordem aleatória"""
    indexes = [i for i, _ in enumerate(data)]  # cria uma lista de índices
    random.shuffle(indexes)                    # os embaralha
```

```
    for i in indexes:                          # retorna os dados naquela ordem
        yield data[i]
```

Andaremos um passo gradiente para cada ponto de dados. Esse método deixa a possibilidade de circularmos próximos a um mínimo para sempre, então quando pararmos de obter melhorias, diminuiremos o tamanho do passo e, eventualmente, pararemos:

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):

    data = zip(x, y)
    theta = theta_0                            # palpite inicial
    alpha = alpha_0                            # tamanho do passo inicial
    min_theta, min_value = None, float("inf")  # o mínimo até agora
    iterations_with_no_improvement = 0

    # se formos até 100 iterações sem melhorias, paramos
    while iterations_with_no_improvement < 100:
        value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )

        if value < min_value:
            # se achou um novo mínimo, lembre-se
            # e volte para o tamanho do passo original
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # do contrário, não estamos melhorando, portanto tente
            # diminuir o tamanho do passo
            iterations_with_no_improvement += 1
            alpha *= 0.9

        # e ande um passo gradiente para todos os pontos de dados
        for x_i, y_i in in_random_order(data):
            gradient_i = gradient_fn(x_i, y_i, theta)
            theta = vector_subtract(theta, scalar_multiply(alpha, gradient_i))

    return min_theta
```

A versão estocástica será tipicamente mais rápida do que a versão batch. Naturalmente, vamos querer uma versão que maximize da mesma forma:

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):
    return minimize_stochastic(negate(target_fn),
                               negate_all(gradient_fn),
                               x, y, theta_0, alpha_0)
```

Para Mais Esclarecimentos

- Continue lendo! Usaremos gradiente descendente para resolver problemas pelo restante do livro.

- Neste ponto, você já está cansado de me ver recomendar a leitura de livros didáticos. Se servir de consolo, *Active Calculus (http://gvsu.edu/s/xr/)* parece mais legal do que os livros didáticos de cálculo com que eu aprendi.
- scikit-learn possui um módulo de Gradiente Descendente Estocástico (*http://scikit-learn.org/stable/modules/sgd.html*). Não é tão geral quanto o nosso em alguns pontos e mais geral em outros. Apesar de que, na maioria das situações do mundo real, você usará bibliotecas nas quais a otimização já estará pronta, então você não terá que se preocupar com elas (exato quando não funcionar corretamente, o que, um dia, inevitavelmente, acontecerá).

CAPÍTULO 9

Obtendo Dados

Para escrevê-lo, levou três meses; para concebê-lo, três minutos; para coletar os dados nele, toda a minha vida.
—F. Scott Fitzgerald

Para se tornar um cientista de dados, você precisa de dados. Na verdade, como um cientista de dados, você passará uma embaraçosa grande fração do seu tempo adquirindo, limpando e transformando dados. Em uma emergência, você sempre pode inserir os dados você mesmo (ou, se você tiver minions, faça os fazê-lo) mas, geralmente, não é um bom uso do seu tempo. Neste capítulo, consideraremos maneiras diferentes de obter dados para Python e nos formatos adequados.

stdin e stdout

Se você executar seus scripts de Python na linha de comando, você pode *canalizar* (*pipe*) os dados por meio deles usando `sys.stdin` e `sys.stdout`. Por exemplo, este é um script que lê linhas de texto e devolve as que combinarem com uma expressão regular:

```
# egrep.py
import sys, re

# sys.argv é a lista dos argumentos da linha de comandos
# sys.argv [0] é o nome do programa em si
# sys.argv [1] será o regex especificado na linha de comandos
regex = sys.argv[1]

# para cada linha passada pelo script
for line in sys.stdin:
    # se combinar com o regex, escreva-o para o stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

E este é um que conta as linhas recebidas e exibe a contagem:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print vai para sys.stdout
print count
```

Você poderia usá-los para contar quantas linhas de um arquivo contêm números. No Windows, você usaria:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

enquanto que no sistema Unix:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

Este é o caractere pipe |, que significa "use a saída do comando da esquerda como a entrada do comando da direita". Você pode construir encadeamentos (*pipelines*) elaborados de processamento de dados dessa forma.

Se você está usando o Windows, pode deixar a parte Python fora deste comando:

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

Se você está no sistema Unix, tal comando talvez requeira um pouco mais de trabalho para ser feito (*http://bit.ly/1L2Wgb7*).

Igualmente, este script conta as palavras em sua entrada e exibe as mais comuns:

```
# most_common_words.py
import sys
from collections import Counter

# passa o número de palavras como primeiro argumento
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)   # código de saída não-zero indica erro

counter = Counter(word.lower()                      # palavras em minúsculas
                  for line in sys.stdin             #
                  for word in line.strip().split()  # se separam por espaços
                  if word)                          # pula as 'palavras' vazias

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
```

```
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

depois disso, você poderia fazer algo como:

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

Se você for um programador familiarizado com Unix, provavelmente você está familiarizado com uma grande variedade de ferramentas de linhas de comando (por exemplo, `egrep`) que são construídos dentro do seu sistema operacional e provavelmente vocês as prefere do que construir a sua própria do zero. Ainda assim, é bom saber que você pode se precisar.

Lendo Arquivos

Você também pode ler a partir de e escrever nos arquivos diretamente no seu código. Python facilita o trabalho com arquivos.

O Básico de Arquivos Texto

O primeiro passo para trabalhar com arquivos de texto é obter um *objeto de arquivo* usando `open`:

```
# 'r' significa somente leitura
file_for_reading = open('reading_file.txt', 'r')

# 'w' é escrever - - destruirá o arquivo se ele já existir!
file_for_writing = open('writing_file.txt', 'w')

# 'a' é anexar - - para adicionar ao final do arquivo
file_for_appending = open('appending_file.txt', 'a')

# não se esqueça de fechar os arquivos ao terminar
file_for_writing.close()
```

Como é muito fácil esquecer de fechar os arquivos, você deveria sempre usá-los em um bloco `with`, pois no término de cada um eles serão fechados automaticamente:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)
```

```
# neste ponto f já foi fechado, não tente usá-lo
process(data)
```

Se você precisar ler um arquivo de texto inteiro, você pode apenas iterar sobre as linhas do arquivo usando `for`:

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:                # observe cada linha do arquivo
        if re.match("^#",line):      # use um regex para ver se começa com '#'
            starts_with_hash += 1    # se começar, adicione 1 à contagem
```

Toda linha que você obtém desse modo termina em um caractere de nova linha (*newline*), logo, você vai querer `strip()` removê-lo frequentemente antes de fazer qualquer coisa.

Por exemplo, imagine que você tenha um arquivo cheio de endereços de e-mail, um por linha e que você precisa gerar um histograma de domínios. As regras para extrair os domínios corretamente são sutis (por exemplo, a Lista Pública de Sufixo em *https://publicsuffix.org*), uma boa primeira aproximação é pegar as partes dos endereços de e-mails que vêm depois do `@`. (O que nos dá uma relação errada para endereços de e-mail como `joel@mail.datasciencester.com`.)

```
def get_domain(email_address):
    """separa no '@' e retorna na última parte"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Arquivos delimitados

O endereço de e-mail hipotético que acabamos de processar tem um endereço por linha. Com mais frequência, você trabalhará com arquivos com muitos dados em cada linha. Tais arquivos são *separados por vírgula* (*comma-separated*) ou *por tabulação* (*tab-separated*). Cada linha possui diversos campos, com uma vírgula (ou uma tabulação) indicando onde um campo termina e o outro começa.

Começa a ficar complicado quando você tem campos com vírgulas, tabulações e newlines neles (inevitavelmente você terá). Por esse motivo, é quase sempre um erro tentar analisar sozinho. Em vez disso, você deveria usar o modulo `csv` do Python (ou as bibliotecas `pandas`). Por razões técnicas — fique à vontade para culpar a Microsoft — você deve sempre trabalhar com arquivos `csv` no *modo binário* incluindo um `b` depois de `r` ou `w` (veja Stack Overflow em *http://bit.ly/1L2Y7wl*).

Se seu arquivo não possuir cabeçalho (o que significa que você quer que cada linha seja como uma `list` que deposita em você o fardo de saber o conteúdo de cada coluna), você pode usar `csv.reader` para iterar sobre as linhas, cada qual será uma lista apropriadamente separada.

Por exemplo, se tivéssemos um arquivo delimitado por tabulação de preços de ações:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

poderíamos processá-los com:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

Se seu arquivo possui cabeçalhos:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

você pode pular a linha (com uma chamada inicial para `reader.next()`) ou obter cada linha como um `dict` (com os cabeçalhos como chaves) usando `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

Mesmo se seu arquivo não tiver cabeçalhos você ainda pode usar `DictReader` ao passar a chave a eles como o parâmetro `fieldnames`.

Da mesma forma, você pode transcrever os dados delimitados usando `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

`csv.writer` fará a coisa certa se seus campos possuírem vírgulas. Seu escritor feito à mão provavelmente não. Por exemplo, se você tentar:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# não faça isso!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # talvez tenha muitas vírgulas nele!
        f.write("\n")                    # a linha pode ter newlines também!
```

Você acabará com um arquivo csv que se parece com:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

e ninguém mais conseguirá entender.

Extraindo Dados da Internet

Outra maneira de se obter dados é extraindo-os das webpages. Pesquisar páginas da web é muito fácil; extrair informações estruturadas e significativas não é tão fácil.

HTML e Sua Subsequente Pesquisa

As páginas na internet são escritas em HTML, na qual o texto (idealmente) é marcado em elementos e atributos:

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

Em um mundo perfeito, em que todas as páginas da Internet são marcadas semanticamente a nosso favor, seríamos capazes de extrair dados usando regras como "encontre o elemento <p> cujo `id` é `subject` e retorne o texto que ele contém". No mundo real, HTML não é muito bem formulado, muito menos comentado. Isso significa que precisaremos de ajuda para entender tudo isso.

Para extrair dados do HTML, usaremos a biblioteca BeautifulSoup (*http://www.crummy.com/software/BeautifulSoup/*), pois ela constrói uma árvore a partir de vários elementos de uma página e fornece uma simples interface para acessá-los. Enquanto eu escrevo este livro, a versão mais recente é BeautifulSoup 4.3.2 (`pip install beautifulsoup4`), que é a que

usaremos. Também usaremos as bibliotecas de pedidos (`pip install requests`), que é uma maneira mais simpática de fazer pedidos (*http://docs.python-requests.org/en/latest/*) ao HTTP do que para qualquer coisa construída dentro de Python.

O interpretador HTML embutido em Python não é tão flexível, o que significa que nem sempre ele se dá bem com o HTML que não possui boa formação. Por esse motivo, usaremos um interpretador diferente, que precisamos instalar:

```
pip install html5lib
```

Para usar o Beautiful Soup, teremos que passar um pouco de HTML para a função `BeautifulSoup()`. Em nossos exemplos, este será o resultado de uma chamada para `requests.get`:

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

depois disso, iremos longe usando poucos métodos simples.

Trabalharemos, na maior parte, com objetos `Tag` que correspondem às marcações (*tags*) representando a estrutura de uma página HTML.

Por exemplo, para encontrar a primeira marcação <p> você pode usar:

```
first_paragraph = soup.find('p')        # ou somente soup.p
```

Você pode obter o conteúdo do texto de uma `Tag` usando a propriedade `text`:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

E você pode extrair os atributos de uma marcação tratando-a como um `dict`:

```
first_paragraph_id = soup.p['id']       # surge KeyError se não tiver 'id'
first_paragraph_id2 = soup.p.get('id')  # retorna None se não tiver 'id'
```

Você pode obter múltiplas marcações ao mesmo tempo:

```
all_paragraphs = soup.find_all('p')  # ou apenas soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Frequentemente, você encontrará marcações com uma `class` específica:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

Você pode combiná-los para implementar uma lógica mais elaborada. Por exemplo, se você quiser encontrar todo elemento que está contido dentro de um elemento , você poderia fazer assim:

```
# atenção, vai retornar o mesmo span múltiplas vezes
# se ele ficar dentro de múltiplos divs
# seja mais esperto se esse for o caso
```

```
spans_inside_divs = [span
                     for div in soup('div')      # para cada <div> na página
                     for span in div('span')]    # encontra cada <span> dentro
```

Esse punhado de recursos permitirá que façamos bastante coisa. Se você precisar fazer coisas mais complicadas (ou se é curioso), verifique a documentação.

Naturalmente, qualquer dado que seja importante pode não ser classificado como `class="important"`. Você terá que inspecionar com cuidado a fonte do HTML, raciocinar com sua lógica seletiva, e se preocupar com casos extremos para se certificar que seu dado está correto. Veremos um exemplo.

Exemplo: Livros O'Reilly Sobre Dados

Um investidor em potencial para a DataSciencester acha que dados são apenas uma moda passageira. Para provar que ele está errado, você decide examinar quantos livros sobre dados a O'Reilly publicou ao longo dos anos. Depois de fuçar pelo site, você encontra muitas páginas de livros sobre dados (e vídeos) e 30 itens exploráveis em uma página de diretórios com URLs como esta:

```
http://shop.oreilly.com/category/browse-subjects/data.do?
sortby=publicationDate&page=1
```

A menos que você seja um idiota (e a menos que você queira que seu extrator seja banido), sempre que você quiser extrair dados de um website, você deve verificar primeiro se ele possui algum tipo de política de acesso. Olhando em:

```
http://oreilly.com/terms/
```

não parece haver nada proibindo esse projeto. A fim de sermos bons cidadãos, também devemos procurar pelo arquivo *robots.txt* que dizem aos *webcrawlers* (programas que coletam dados da internet) como se comportar. As partes importantes em *`http://shop.oreilly.com/robots.txt`* são:

```
Crawl-delay: 30
Request-rate: 1/30
```

A primeira linha nos diz que devemos esperar 30 segundos entre um pedido e outro; a segunda, que devemos pedir uma página a cada 30 segundos. Então, basicamente, elas são duas formas diferentes de dizer a mesma coisa. (Existem outras linhas que indicam diretórios para não serem extraídos, mas elas não incluem nossa URL, portanto estamos bem.)

Há sempre a possibilidade que a O'Reilly vá remodelar seu site e quebrar toda a lógica desta seção. Farei o que eu puder para prevenir isso, claro, mas eu não possuo muita influência sobre eles. Embora, se cada um de vocês pudesse convencer cada um que vocês conhecem a comprar um exemplar deste livro…

Para descobrir como extrair os dados, vamos baixar uma daquelas páginas e jogá-las no Beautiful Soup:

```
# você não precisa dividir a url desta forma a menos que ela precise caber em um livro
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

Se você vir a fonte da página (no seu navegador, clique com o botão direito e selecione ("Exibir código-fonte" ou "Exibir código-fonte da página" ou qualquer outra opção que se pareça com isso), você verá que cada livro (ou vídeo) parece estar contido unicamente em um elemento `<td>` da célula da tabela cuja `class` é `thumbtext`. Esta é (em versão resumida) o HTML relevante para um livro:

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014    </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                              Ebook:

                              <span class="price"> $29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

Um bom primeiro passo é encontrar todos os elementos de marcação `td thumbtext`:

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

Gostaríamos de filtrar os vídeos. (Esse aspirante a investidor só se impressiona com livros.) Se inspecionarmos o HTML um pouco mais além, vemos que cada `td` contém um ou mais elementos `span` cuja `class` é `pricelabel`, e cujo texto se parece com `Ebook:` ou `Video:` ou `Print:`. Parece que os vídeos contêm apenas um `pricelabel`, cujo texto começa com `Video` (após remover os espaços importantes). Desse modo, podemos testar para vídeos com:

```
def is_video(td):
    """é um video se tiver exatamente um pricelabel, e se
```

```
        o texto corrido dentro do pricelabel começar com 'Video'"""
        pricelabels = td('span', 'pricelabel')
        return (len(pricelabels) == 1 and
                pricelabels[0].text.strip().startswith("Video"))

print len([td for td in tds if not is_video(td)])
# 21 para mim, talvez seja diferente para você
```

Agora estamos prontos para começar a puxar os dados para fora dos elementos td. Parece que o título do livro é o texto dentro da marcação <a> dentro de :

```
title = td.find("div", "thumbheader").a.text
```

Os autores estão no texto de AuthorName . Eles são introduzidos por um By (do qual queremos nos livrar) e separados por vírgulas (que queremos separar, depois teremos que nos livrar dos espaços):

```
author_name = td.find('div', 'AuthorName').text
authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

O ISBN parece estar contido no link que está em thumbheader :

```
isbn_link = td.find("div", "thumbheader").a.get("href")

# re.match captura a parte do regex em parênteses
isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

E a data é só o conteúdo de :

```
date = td.find("span", "directorydate").text.strip()
```

Vamos colocar tudo junto em uma função:

```
def book_info(td):
    """dado uma marcação BeautifulSoup <td> representando um livro,
    extrai os detalhes do livro e retorna um dict"""

    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
        "title" : title,
        "authors" : authors,
        "isbn" : isbn,
        "date" : date
    }
```

Agora estamos prontos para extrair:

```
from bs4 import BeautifulSoup
import requests
```

```
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # na época da escrita deste livro, provavelmente mais agora

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

    # agora seja um bom cidadão e respeite os robots.txt!
    sleep(30)
```

Extrair dados de HTML desta forma é mais uma arte do que uma ciência dos dados. Existem outras incontáveis lógicas encontre-os-livros e encontre-o-título que teriam funcionado bem também.

Agora que coletamos os dados, podemos traçar o número de livros publicados a cada ano (Figura 9-1):

```
def get_year(book):
    """book["date"] se parece com 'November 2014' então precisamos
    dividir no espaço e então pegar o segundo pedaço"""
    return int(book["date"].split()[1])

# 2014 é o último ano completo de dados (quando eu fiz isso)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("# de livros de dados")
plt.title("A Área de Dados É Grande!")
plt.show()
```

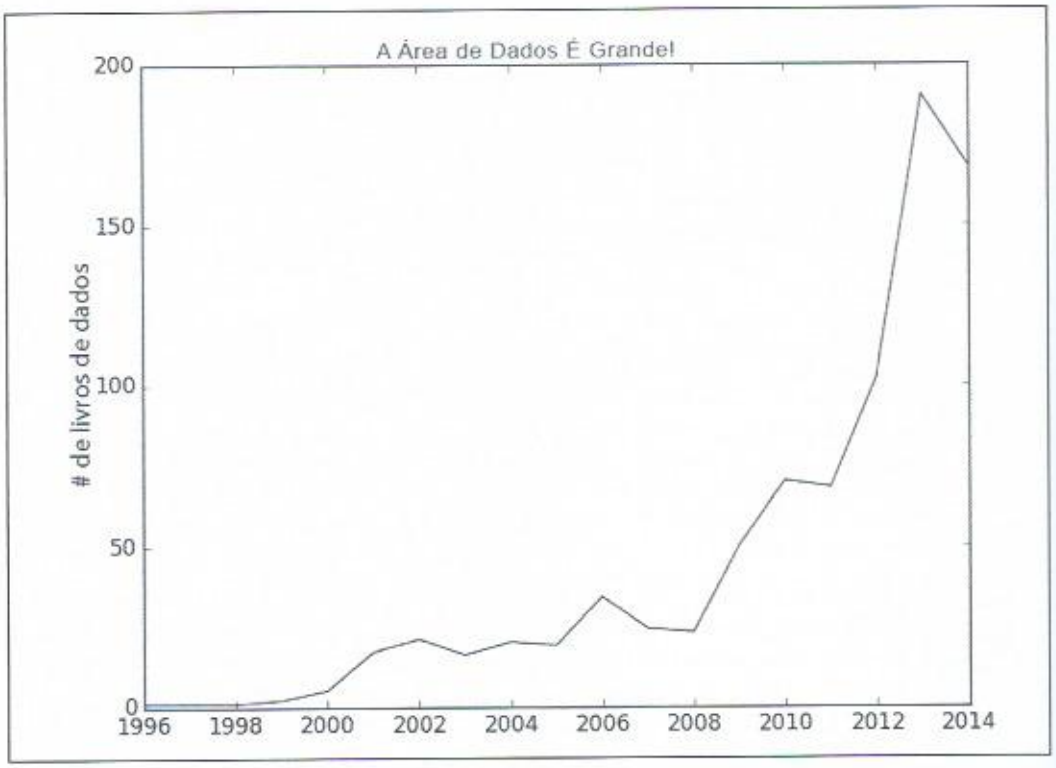

Figura 9-1. Número de livros de dados por ano

Infelizmente, o aspirante a investidor olha para o gráfico e decide que 2013 foi a "taxa máxima de dados".

Usando APIs

Muitos websites e serviços web fornecem interfaces de programação de aplicativos (APIs), permitindo que você solicite os dados em formato estruturado. Isso poupa o trabalho de ter que extraí-los!

JSON (e XML)

Como o HTTP é um protocolo para a transferência de *texto*, os dados que você solicita por meio de uma API da web deve ser *serializada* em formato de string. Geralmente, essa serialização usa o JavaScript Object Notation (JSON). Os objetos JavaScript se parecem bastante com os `dicts` do Python, o que facilita a interpretação de suas strings:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

Podemos analisar JSON usando o módulo `json` do Python. Em especial, usaremos a função `loads`, que desserializa uma string representando um objeto JSON em um objeto Python:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""

# analisa o json para criar um dict do Python
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

Às vezes, um provedor API te odeia e fornece apenas respostas em XML:

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

Você pode usar `BeautifulSoup` para obter os dados do XML da mesma forma como usamos para obter do HTML; verifique a sua documentação para detalhes.

Usando Uma API Não Autenticada

A maioria das APIs de hoje em dia requer que você primeiro as autentique a fim de poder usá-las. Enquanto não nos resignamos a essa política, ela cria muitos padrões extras que distorcem a nossa exposição. Assim, daremos uma primeira olhada na API do GitHub (*http://developer.github.com/v3/*), com a qual você pode praticar coisas simples não-autenticadas:

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"

repos = json.loads(requests.get(endpoint).text)
```

Nesse momento, `repos` é uma `list` de `dicts` do Python, cada uma representando um repositório público na minha conta GitHub. (Sinta-se à vontade para substituir seu nome de usuário e pegar seu repositório de dados GitHub. Você possui um conta Github, certo?)

Podemos usar isso para descobrir em quais meses e dias da semana tenho mais tendências para criar um repositório. O único problema é que as datas na resposta são strings (Unicode):

```
u'created_at': u'2013-07-05T02:02:28Z'
```

O Python não vem com um bom analisador de datas, então teremos que instalar um:

```
pip install python-dateutil
```

do qual você provavelmente só precisará da função `dateutil.parser.parse`:

```
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Da mesma forma, você pode obter as linguagens dos meus cinco últimos repositórios:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Basicamente, não trabalharemos com APIs nesse nível baixo de "faça os pedidos e analise as respostas você mesmo". Um dos benefícios de se usar Python é que alguém já construiu uma biblioteca para quase qualquer API que você esteja interessado em acessar. Quando elas são bem-feitas, elas podem te poupar de muitos problemas ao tentar entender os detalhes mais cabeludos do acesso API. (Quando elas não são bem-feitas, ou quando são baseadas em versões extintas das APIs correspondentes, elas podem dar enormes dores de cabeça.)

Apesar disso, você talvez tenha que implantar seu próprio acesso à biblioteca API (ou, mais provável, depurar porque a de alguém não está funcionando), portanto, é bom saber de alguns detalhes.

Encontrando APIs

Se você precisa de dados de um site específico, procure por desenvolvedores ou a seção de API do site para detalhes, e tente procurar na web por "python_api" para encontrar uma biblioteca. Existe uma API Rotten Tomatoes para Python. Existem múltiplas camadas (wrappers) para a API Klout, para a API Yelp, para a API IMDB, e assim por diante.

Se você está procurando por listas de APIs que tenham as camadas Python, dois diretórios estão em Python API (*http://www.pythonapi.com*) e Python for Beginners (*http://bit.ly/1L35VOR*).

Se você quer um diretório de APIs web mais abrangente (sem necessariamente as camadas Python), um bom recurso é o Programmable Web (*http://www.programmableweb.com*), o qual possui um enorme diretório de APIs categorizados.

E se depois de tudo isso você não encontrar o que precisa, há sempre a opção de extração, o último refúgio de um cientista de dados.

Exemplo: Usando as APIs do Twitter

O Twitter é um fonte fantástica de dados com o qual trabalhar. Você pode usá-lo para ver as notícias em tempo real. Você pode usá-lo para medir as reações aos eventos atuais. Você pode usá-lo para encontrar links relacionados a tópicos específicos. Você pode usá-lo para praticamente tudo que você possa imaginar, contanto que você consiga acesso aos seus dados. E você pode ter esse acesso por meio das APIs.

Para interagir com as APIs do Twitter, usaremos a biblioteca Twython em *https://github.com/ryanmcgrath/twython*, (`pip install twython`). Existem algumas bibliotecas Python para o Twitter por aí, mas foi com essa que eu obtive mais sucesso. Sinta-se encorajado a explorar as outras também!

Obtendo Credenciais

Para usar as APIs do Twitter, você deve ter algumas credenciais (logo, você precisa de uma conta no Twitter, e deveria ter de qualquer forma para fazer parte da viva e amigável comunidade `#datascience` no Twitter). Como todas as instruções que se relacionam com os websites que eu não controlo, talvez isto se torne obsoleto em algum momento, mas espero que funcione por um tempo. (Apesar de eles já terem mudado pelo menos uma vez enquanto eu estava escrevendo este livro, então boa sorte!)

1. Vá para *https://apps.twitter.com/*.
2. Se você não estiver logado, clique em Entrar e insira seu nome de usuário e senha do Twitter.
3. Clique em Criar um Novo Aplicativo.
4. Dê a ele um nome (como "Data Science") e uma descrição, e coloque qualquer URL como website (não importa qual). Deixe a URL de retorno em branco.
5. Concorde com os Termos de Serviço e clique em Criar.
6. Anote a chave e o segredo do consumidor.
7. Clique em "Criar meu token de acesso".
8. Anote o token de acesso e o de segredo (talvez você tenha que atualizar a página).

A chave e o segredo do consumidor dizem ao Twitter qual aplicação está acessando suas APIs, enquanto que o token de acesso e o token de acesso do segredo dizem ao Twitter *quem* está acessando suas APIs. Se você já usou sua conta do Twitter para entrar em outro site,

a página "clique para autorizar" estava gerando um token de acesso para aquele site usar a fim de convencer o Twitter que foi você (ou, ao menos, agindo como se fosse você). Como não precisamos da funcionalidade "deixar qualquer um entrar", podemos sobreviver com o token de acesso e o token de acesso do segredo gerados.

A chave/segredo do consumidor e a chave/segredo do token de acesso devem ser tratadas como *senhas*. Você não deve compartilhá-las, publicá-las em seu livro ou acessá-las no repositório público Github. Uma solução simples é armazená-las em um arquivo *credentials.json* que não é acessado e usar seu código `json.loads` para recuperá-la.

Usando Twython

Primeiro observaremos o Search API (*https://dev.twitter.com/docs/api/1.1/get/search/tweets*), o qual requer apenas a chave e o segredo do consumidor e não o token de acesso ou segredo:

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

O `.encode("utf-8")` é necessário para lidar com o fato de que os tweets geralmente contêm caracteres Unicode com os quais `print` não pode lidar. (Se você deixar de lado, você deve receber um `UnicodeEncodeError`.)

É quase certo que em algum momento de sua carreira de data science você vai se deparar com sérios problemas Unicode e, em algum momento, você precisará se referir à documentação Python (*http://bit.ly/1ycODJw*) ou relutantemente começar a usar Python 3, o qual lida muito melhor com texto Unicode.

Se você executar isso, você deverá receber tweets como:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

Isso não é muito interessante, principalmente porque o Search API do Twitter apenas mostra a parte dos resultados que ele quer. Quando você está praticando data science, você vai querer cada vez mais tweets. É aí que o Streaming API (*http://bit.ly/1ycOEgG*) é útil. Ele permite que você se conecte à (uma amostra de) avalanche Twitter. Para usá-lo, você precisará se identificar usando seus tokens de acesso.

A fim de acessar o Streaming API com Twython, precisamos definir uma classe que herde `TwythonStreamer` e que anule o método `on_success` (e possivelmente seu método `on_error`):

```
from twython import TwythonStreamer

# anexar dados à variável global é bem pobre
# mas simplifica o exemplo
tweets = []

class MyStreamer(TwythonStreamer):
    """nossa própria subclasse de TwythonStreamer que especifica
    como interagir com o stream"""

    def on_success(self, data):
        """o que fazemos quando o twitter nos envia dados?
        aqui os dados serão um dict de Python representando um tweet"""

        # quer coletar apenas tweets da língua inglesa
        if data['lang'] == 'en':
            tweets.append(data)
            print "received tweet #", len(tweets)

        # para quando coleta o suficiente
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
        print status_code, data
        self.disconnect()
```

MyStreamer vai se conectar o stream do Twitter e esperar que o Twitter o abasteça de dados. Toda vez que ele receber dados (aqui, um tweet é representado por um objeto de Python), ele passa para o método `on_success`, que o anexa à nossa lista de `tweets` se sua língua for inglesa, e então desconecta o streamer após ter coletado 1000 tweets.

Tudo o que resta para inicializá-lo e fazê-lo funcionar:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# começa a consumir status públicos que contenham a palavra-chave 'data'
stream.statuses.filter(track='data')
```

```
# se quiséssemos começar a consumir uma amostra de *all* status públicos
# stream.statuses.sample()
```

Ele executará até coletar 1000 tweets (ou até encontrar um erro) e irá parar e é nesse momento que você pode começar a inicializar tais tweets. Por exemplo, você poderia encontrar as hashtags mais famosas com:

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print top_hashtags.most_common(5)
```

Cada tweet contém muitos dados. Você pode passear ou mergulhar fundo na documentação API do Twitter (*https://dev.twitter.com/overview/api/tweets*).

Em um projeto que não seja de brincadeira, você não gostaria de depender de uma `list` in-memory para armazenar seus tweets. Em vez disso, você os salvaria em um arquivo ou banco de dados, para que você possa tê-los permanentemente.

Para Mais Esclarecimentos

- pandas (*http://pandas.pydata.org/*) é a biblioteca primária com a qual os tipos de data science usam para trabalhar (e, em especial, importar) dados.
- Scrapy (*http://scrapy.org/*) é uma biblioteca cheia de recursos para construir extratores da web mais complicados, que fazem coisas como seguir links desconhecidos.

CAPÍTULO 10

Trabalhando com Dados

Os especialistas, muitas vezes, possuem mais dados do que juízo.
—Colin Powell

Trabalhar com dados é uma arte e uma ciência. Temos discutido mais sobre a parte científica, mas neste capítulo consideraremos um pouco de arte.

Explorando Seus Dados

Após identificar as questões que você tem tentado responder e ter posto as mãos em alguns dados, você pode ficar tentado a ir mais fundo, começar a construir modelos e obter respostas. Mas você deveria resistir a esse impulso. Seu primeiro passo deveria ser *explorar* seus dados.

Explorando Dados Unidimensionais

O caso mais simples é quando você tem um conjunto de dados unidimensional, apenas uma coleção de números. Por exemplo, eles poderiam ser a media diária de minutos que cada usuário passa no seu site, o número de vezes que cada coleção de vídeos tutoriais de data science foi vista ou o número de páginas de cada livro de data science na sua biblioteca.

Um primeiro passo inevitável é computar algumas estatísticas sumárias. Você gostaria de saber quantos pontos de dados você tem, o menor, o maior, a média e o desvio padrão.

Mas nem isso tudo fornece, necessariamente, um bom entendimento. Um próximo passo adequado é criar um histograma para agrupar seus dados em agrupamentos (*buckets*) discretos e contar quantos pontos vão para cada um:

```
def bucketize(point, bucket_size):
    """reduza o ponto para o próximo múltiplo mais baixo de bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """agrupa os pontos e conta quantos em cada bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()
```

Por exemplo, considere os seguintes conjuntos de dados:

```
random.seed(0)

# uniforme entre -100 e 100
uniform = [200 * random.random() - 100 for _ in range(10000)]

# distribuição normal com média 0, desvio padrão 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

Ambos possuem médias próximas a 0 e desvios padrões próximos a 58. No entanto, possuem distribuições bem diferentes. A Figura 10-1 mostra a distribuição de `uniform`:

```
plot_histogram(uniform, 10, "Histograma de Uniform")
```

já a Figura 10-2 mostra a distribuição de `normal`:

```
plot_histogram(normal, 10, "Histograma Normal")
```

Nesse caso, as duas distribuições possuem `max` e `min` muito diferentes, mas mesmo sabendo isso não seria suficiente para entender *como* elas diferem.

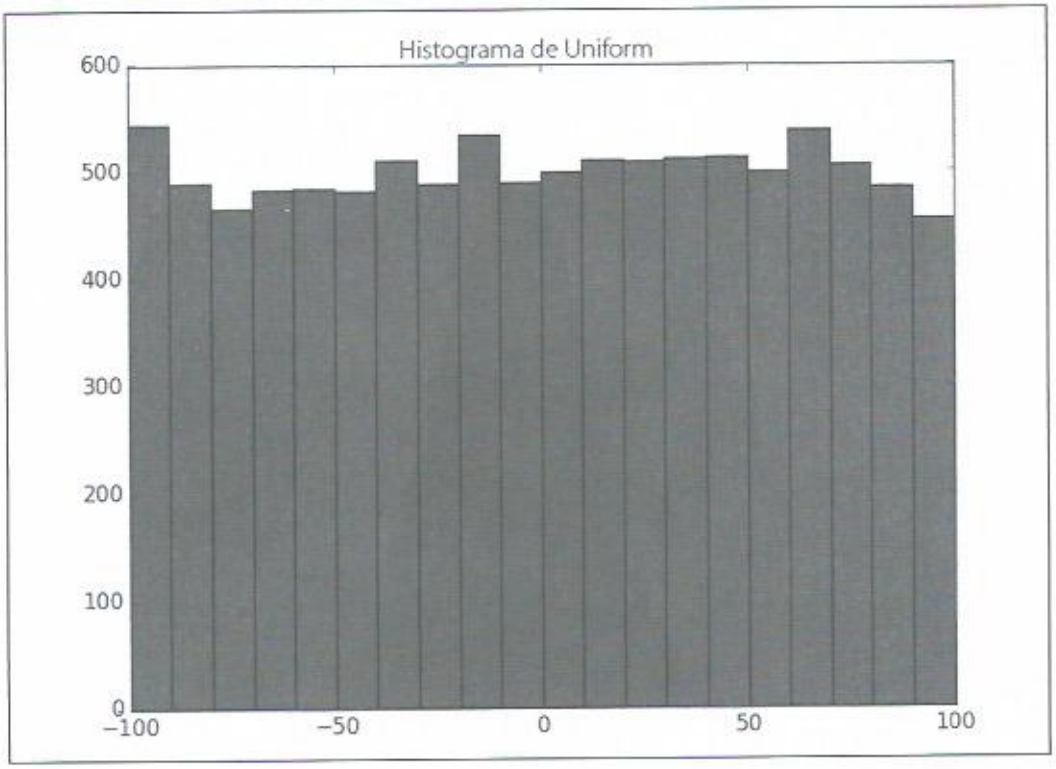

Figura 10-1. Histograma de `uniform`

Duas Dimensões

Agora imagine que você tenha um conjunto de dados com duas dimensões. Talvez, além de minutos diários, você também tenha anos de experiência em data science. Certamente, você gostaria de entender cada dimensão individualmente. Mas você também deve querer dispersar os dados.

Por exemplo, considere outro conjunto de dados falso:

```
def random_normal():
    """retorna um desenho aleatório de uma distribuição normal padrão"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

Se você fosse executar plot_histogram em ys1 e ys2, você teria gráficos muito parecidos (aliás, ambos são distribuídos normalmente com a mesma média e desvio padrão).

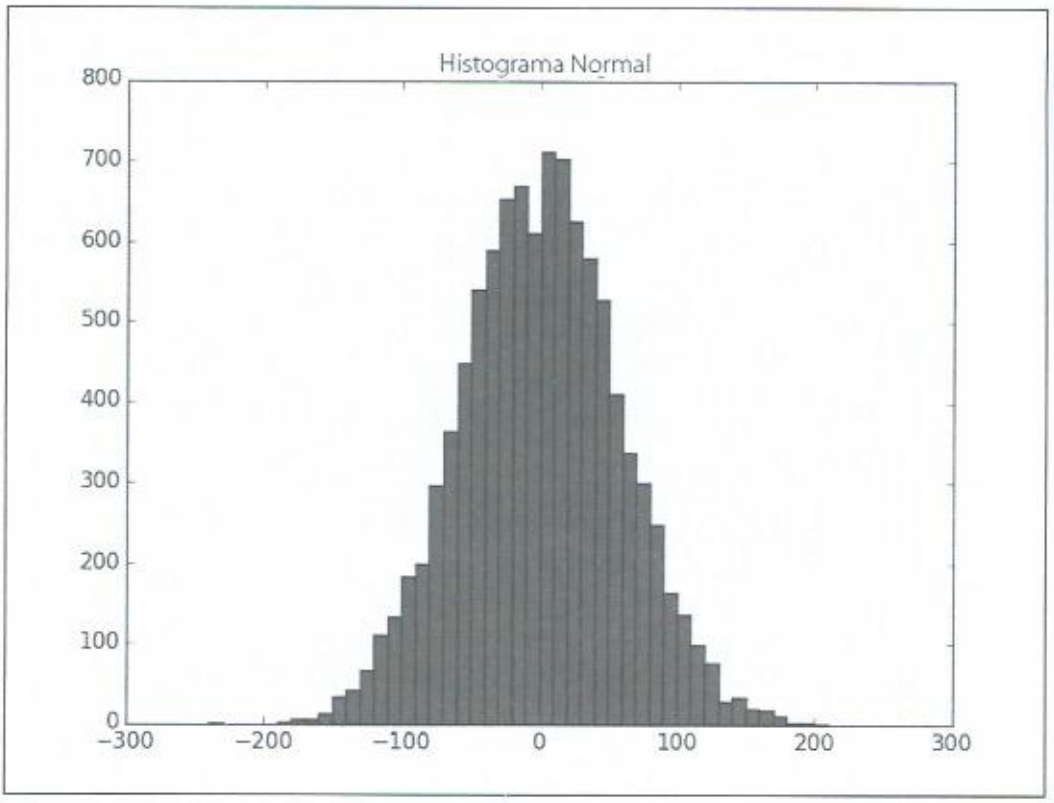

Figura 10-2. Histograma de `normal`

Mas cada um teria distribuições conjuntas diferentes com xs, como mostra a Figura 10-3:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray',  label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Distribuições Conjuntas Muito Diferentes")
plt.show()
```

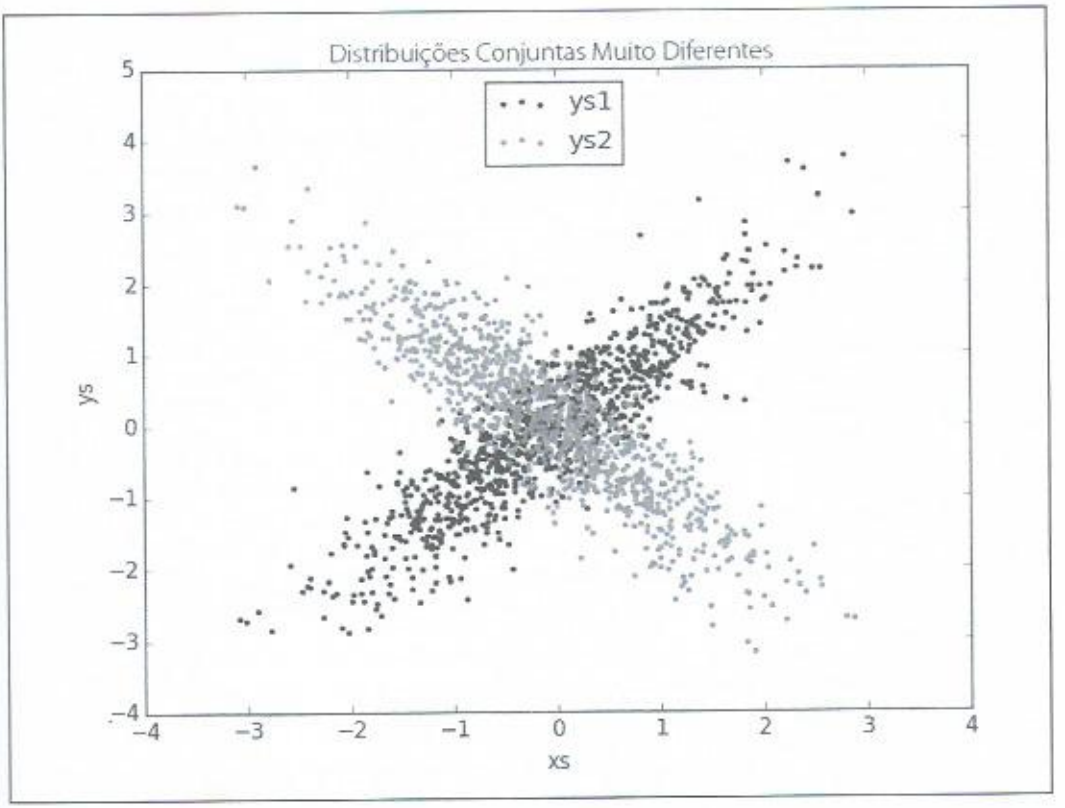

Figura 10-3. Dispersão de dois ys diferentes

Essa diferença também seria aparente se você observasse as correlações:

```
print correlation(xs, ys1)      # 0.9
print correlation(xs, ys2)      # -0.9
```

Muitas Dimensões

Com muitas dimensões, você gostaria de saber como todas as dimensões se relacionam umas com as outras. Uma abordagem simples é observar a *matriz correlacional* (*correlation matrix*), na qual a entrada na linha *i* e na coluna *j* é a correlação entre as dimensões i-ésima e j-ésima dos dados:

```
def correlation_matrix(data):
    """retorna o num_columns x num_columns matrix cuja entrada (i, j)-ésima
    é a correlação entre as colunas de dados i e j"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))

    return make_matrix(num_columns, num_columns, matrix_entry)
```

Uma abordagem mais visual (se você não tiver muitas dimensões) é fazer uma *matriz de gráfico de dispersão* (*scatterplot matrix* — Figura 10-4) mostrando todos os pareamentos dos gráficos de dispersão. Para fazer isso, usaremos `plt.subplots()`, que permite que criemos uma subparcela do nosso gráfico. Nós fornecemos o número de linhas e colunas, e ele retorna um objeto `figure` (que não usaremos) e um array bidimensional de objetos `axes` (cada qual com seu gráfico):

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # dispersa a column_j no eixo x versus column_i no eixo y
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # a menos que i ==j, em cujo caso mostra o nome da série
        else: ax[i][j].annotate("série" + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # então esconde as etiquetas dos eixos exceto
        # os gráficos inferiores e da esquerda
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# conserta as etiquetas inferiores à direita e superiores à esquerda dos eixos,
# que está errado pois seus gráficos somente possuem textos
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

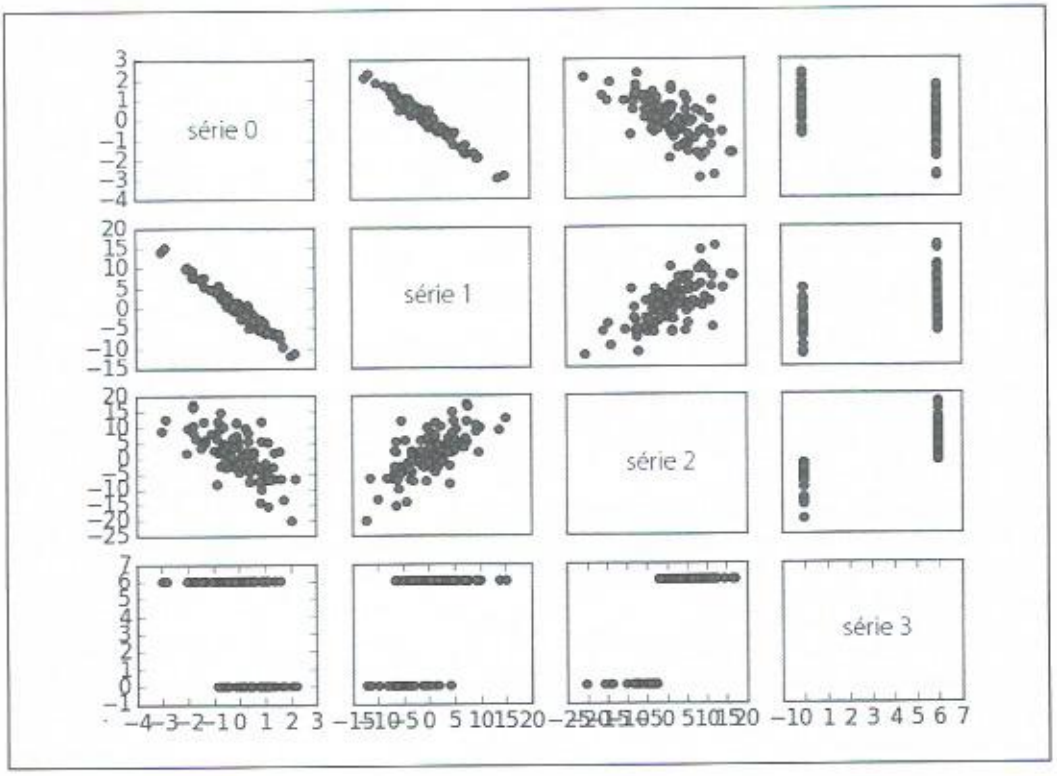

Figura 10-4. Matriz de Gráfico de Dispersão

Ao observar os gráficos de dispersão, você pode ver que a série 1 é muito negativamente correlacionada com a série 0, a série 2 é positivamente correlacionada com a série 1 e a série 3 somente aceita os valores 0 e 6, com 0 correspondendo aos valores menores da série 2 e 6 correspondendo aos maiores.

Essa é uma maneira rápida de ter uma ideia de como as suas variáveis são correlacionadas (a menos que você passe horas ajustando `matplotlib` para exibir as coisas exatamente do jeito que você quer e, nesse caso, não é muito rápido).

Limpando e Transformando

Os dados do mundo real são *sujos*. Muitas vezes, você terá que trabalhar neles antes de usá-los. Vimos alguns exemplos disso no Capítulo 9. Temos que converter strings para `floats` ou `ints` antes de usá-las. Anteriormente, fizemos isso um pouco antes de usarmos os dados:

```
closing_price = float(row[2])
```

Mas é menos propício ao erro fazer a análise no fluxo de entrada, o que podemos fazer ao criar uma função que envolva `csv.reader`. Forneceremos uma lista de interpretadores a ele, cada um especificando como analisar uma das colunas. Usaremos `None` para representar "não faça nada com esta coluna":

```
def parse_row(input_row, parsers):
    """dada uma lista de interpretadores (alguns podem ser None)
    aplique o apropriado a cada elemento de input_row"""

    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """envolve um reader para aplicar os interpretadores em
    cada uma de suas linhas"""
    for row in reader:
        yield parse_row(row, parsers)
```

E se tiver algum dado ruim? Um valor "float" que não represente um número? Preferiríamos receber um None do que travar nosso programa. Podemos fazer isso com uma função auxiliadora:

```
def try_or_none(f):
    """envolve f para retornar None se f levantar uma exceção
    presume que f leve apenas uma entrada"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none
```

depois disso podemos reescrever parse_row para usá-lo:

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]
```

Por exemplo, se tivermos os preços das ações separados por vírgulas com dados ruins:

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

podemos ler e analisar em um único passo agora:

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)
```

depois disso somente precisamos checar por linhas None:

```
for row in data:
    if any(x is None for x in row):
        print row
```

e decidir o que queremos fazer com eles. (De modo geral, as três opções são jogá-los fora, voltar para a fonte e tentar consertar o dado ruim/faltoso, ou não fazer nada e cruzar os dedos.)

Poderíamos criar auxiliadores semelhantes para `csv.DictReader`. Nesse caso, você apenas teria que fornecer um `dict` de analisadores por meio de um nome de campo. Por exemplo:

```
def try_parse_field(field_name, value, parser_dict):
    """tenta analisar o valor usando a função adequada a partir de parser_dict"""
    parser = parser_dict.get(field_name)   # None se não tiver tal entrada
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

Uma próxima etapa seria checar por valores discrepantes, usando técnicas do "Explorando Seus Dados" na página 121 ou por investigação ad hoc. Por exemplo, você reparou que uma das datas no arquivo das ações tinha o ano 3014? Isso não gerará nenhum erro (possivelmente), mas é claramente errado, e você terá resultados ruins se você não consertá-lo. Os dados do mundo real têm pontos decimais faltosos, zeros extras, erros tipográficos e outros problemas incontáveis e é o seu trabalho capturá-los. Talvez não seja seu trabalho oficial, mas quem mais vai fazer?

Manipulando Dados

Uma das habilidades mais importantes de um cientista de dados é *manipular dados*. É mais uma visão geral do que uma técnica específica, portanto trabalharemos com um grupo de exemplos para mostrar um pouco.

Imagine que estamos trabalhando com `dicts` dos preços das ações que se parecem com:

```
data = [
    {'closing_price': 102.06,
     'date': datetime.datetime(2014, 8, 29, 0, 0),
     'symbol': 'AAPL'},
    # ...
]
```

Conceitualmente, pensaremos neles como linhas (como em uma planilha).

Vamos começar perguntando sobre esses dados. Pelo caminho, perceberemos padrões no que estamos fazendo e abstrairemos algumas ferramentas para facilitar a manipulação.

Por exemplo, suponha que queremos saber o preço mais alto para a AAPL. Vamos separar em etapas concretas:

1. Restringir em linhas AAPL.
2. Pegar o `closing_price` de cada linha.
3. Levar o `max` de tais preços.

Podemos fazer os três de uma só vez usando uma compreensão de lista:

```
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

Com mais frequência, talvez queiramos saber o preço mais alto para cada ação em nosso conjunto de dados. Uma maneira de fazer é:

1. Agrupar todas as linhas com o mesmo `symbol` (símbolo).
2. Dentro de cada grupo, fazer o mesmo de antes:

```
# agrupa as linhas por símbolo
by_symbol = defaultdict(list)
for row in data:
    by_symbol[row["symbol"]].append(row)

# usa a compreensão do dict para encontrar o max para cada símbolo
max_price_by_symbol = { symbol : max(row["closing_price"]
                                     for row in grouped_rows)
                        for symbol, grouped_rows in by_symbol.iteritems() }
```

Já existem alguns padrões aqui. Nos dois exemplos, tivemos que puxar o valor `closing_price` para fora de cada `dict`. Então vamos criar uma função para recolher um campo de um `dict`, e outra função para arrancar esse mesmo campo de uma coleção de `dicts`:

```
def picker(field_name):
    """retorna uma função que recolhe um campo de um dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """transforma uma lista de dicts em uma lista de valores field_name"""
    return map(picker(field_name), rows)
```

Também podemos criar uma função para agrupar as linhas pelo resultado de uma função `grouper` e aplicar, por opção, um tipo de `value_transform` em cada grupo:

```
def group_by(grouper, rows, value_transform=None):
    # a chave é a saída de grouper, o valor é uma lista de linhas
    grouped = defaultdict(list)
    for row in rows:
        grouped[grouper(row)].append(row)

    if value_transform is None:
        return grouped
    else:
```

```
        return { key : value_transform(rows)
                 for key, rows in grouped.iteritems() }
```

Isso permite que nós reescrevamos os exemplos anteriores de forma simples. Por exemplo:

```
max_price_by_symbol = group_by(picker("symbol"),
                               data,
                               lambda rows: max(pluck("closing_price", rows)))
```

Agora podemos começar a perguntar assuntos mais complicados, como quais são as maiores e menores mudanças de porcentagem ao dia em nosso conjunto de dados. A mudança na porcentagem é `price_today / price_yesterday - 1`, logo precisamos de alguma forma de associar o preço de hoje com o de ontem. Um método possível é agrupar os preços por símbolo e, então, dentro de cada grupo:

1. Ordenar os preços por data.
2. Usar `zip` para ter pares (anteriores, atuais).
3. Transformar os pares em linhas novas de "mudança de percentual".

Começaremos escrevendo uma função que faça o trabalho dentro de cada grupo:

```
def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] - 1

def day_over_day_changes(grouped_rows):
    # organiza as linhas por data
    ordered = sorted(grouped_rows, key=picker("date"))

    # compacta com uma compensação para ter pares de dias consecutivos
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]
```

Então podemos usá-lo como o `value_transform` em um `group_by`:

```
# a chave é symbol, o valor é uma change de dicts
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# coleta todas as changes de dicts para uma lista grande
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]
```

Nesse ponto, fica fácil de encontrar o maior e o menor:

```
max(all_changes, key=picker("change"))
# {'change': 0.3283582089552237,
#  'date': datetime.datetime(1997, 8, 6, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://news.cnet.com/2100-1001-202143.html
```

```
min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
#  'date': datetime.datetime(2000, 9, 29, 0, 0),
#  'symbol': 'AAPL'}
# veja por exemplo http://money.cnn.com/2000/09/29/markets/techwrap/
```

Podemos usar agora o conjunto de dados novo `all_changes` para encontrar qual mês é o melhor para investir em ações tecnológicas. Primeiro, agrupamos as mudanças por mês; então computamos a mudança geral dentro de cada grupo.

Mais uma vez, escrevemos um `value_transform` adequado e usamos `group_by`:

```
# para combinar as mudanças percentuais, adicionamos 1 a cada um, os multiplicamos
# e subtraímos 1 por exemplo, se combinarmos +10% e -20%, a mudança geral é
#    (1 + 10%) * (1 - 20%) - 1 = 1.1 * .8 - 1 = -12%
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)
```

Faremos esses tipos de manipulações no decorrer do livro, geralmente sem chamar muita atenção direta para elas.

Redimensionando

Muitas técnicas são sensíveis à escala dos seus dados. Por exemplo, imagine um conjunto de dados que consiste de altura e pesos de centenas de cientistas de dados e que você está tentando identificar o *agrupamento* (*cluster*) dos tamanhos dos corpos.

Intuitivamente, gostaríamos que os agrupamentos representassem pontos próximos uns aos outros, o que significa que precisamos de alguma noção de distância entre eles. Nós já temos a função `distance` (distância) Euclideana, portanto uma abordagem natural seria tratar os pares (altura, peso) como pontos em um espaço bidimensional. Observe as pessoas na lista da Tabela 10-1.

Tabela 10-1. Alturas e Pesos

Pessoa	Altura (polegadas)	Altura (centímetros)	Peso (libras)
A	63 inches	160 cm	150 pounds
B	67 inches	170.2 cm	160 pounds
C	70 inches	177.8 cm	171 pounds

Se medirmos a altura em polegadas, o vizinho mais próximo de B é A:

```
a_to_b = distance([63, 150], [67, 160])        # 10.77
a_to_c = distance([63, 150], [70, 171])        # 22.14
b_to_c = distance([67, 160], [70, 171])        # 11.40
```

Porém, se medirmos em centímetros, o vizinho mais próximo de B é C:

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])  # 13.37
```

É uma problemática evidente se a mudança das unidades mudam os resultados dessa forma. Por esse motivo, quando as dimensões não são comparáveis umas com as outras, às vezes *redimensionamos* nossos dados a fim de que cada dimensão tenha média 0 e desvio padrão 1. Isso nos livra das unidades, convertendo cada dimensão para "desvios padrões a partir da média".

A partir dessa explicação, precisaremos computar a `mean` e o `standard_deviation` para cada coluna:

```
def scale(data_matrix):
    """retorna a média e os desvios padrões de cada coluna"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

E então os usa para criar uma nova matriz de dados:

```
def rescale(data_matrix):
    """redimensiona os dados de entrada para que cada coluna
    tenha média 0 e desvio padrão 1
    deixa intactas colunas sem desvio"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

Como sempre, você precisa utilizar de seu bom senso. Se você fosse pegar um conjunto de dados enorme de alturas e pesos e filtrá-los somente para as pessoas que possuíssem entre 69,5 e 70,5 polegadas, seria bem provável (dependendo da questão que você esteja tentando responder) que a variação permanecesse apenas como um ruído e você poderia não querer colocar tal desvio padrão em uma relação de igualdade com os desvios das outras dimensões.

Redução da Dimensionalidade

Às vezes, as dimensões "reais" (ou úteis) dos dados podem não corresponder às dimensões que temos. Por exemplo, observe o conjunto de dados na Figura 10-5.

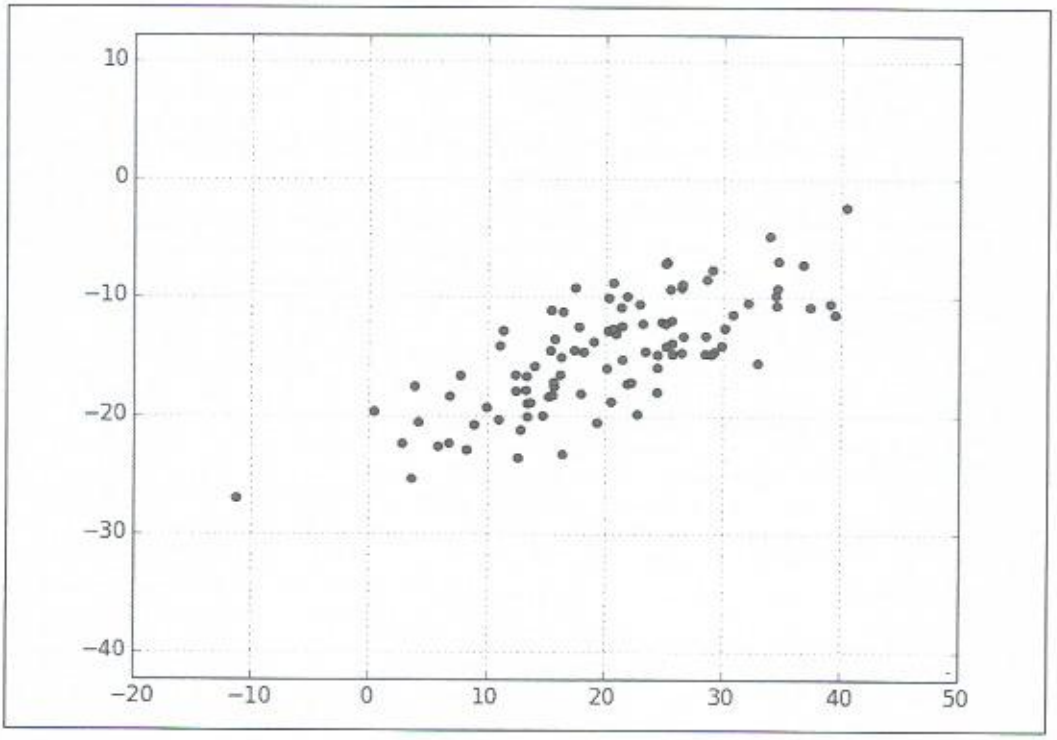

Figura 10-5. Dados com os eixos "errados"

A maioria das variações nos dados parecem ser de uma única dimensão que não corresponde ao eixo x nem ao eixo y.

Quando esse é o caso, podemos usar uma técnica chamada de *análise de componentes principais* para extrair uma ou mais dimensões que capturem a maior variação dos dados possível.

Na prática, você não usaria essa técnica em um conjunto de dados com o dimensional tão baixo. A redução de dimensionalidade é mais útil quando seu conjunto de dados possui um grande número de dimensões e você quer encontrar uma subparcela que captura a maior parte da variação. Infelizmente, esse caso é difícil de ser ilustrado em livro com formato bidimensional.

Na primeira etapa, precisaremos transformar os dados para que cada dimensão tenha média zero:

```
def de_mean_matrix(A):
    """retorna o resultado de subtrair de cada valor em A o valor
    da média da sua coluna. a matriz resultante tem média 0 em cada coluna"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j: A[i][j] - column_means[j])
```

(Se não fizermos isso, é possível que nossas técnicas identificarão a média em si em vez de identificar a variação nos dados.)

A Figura 10-6 mostra os exemplos de dados após o desconto da média.

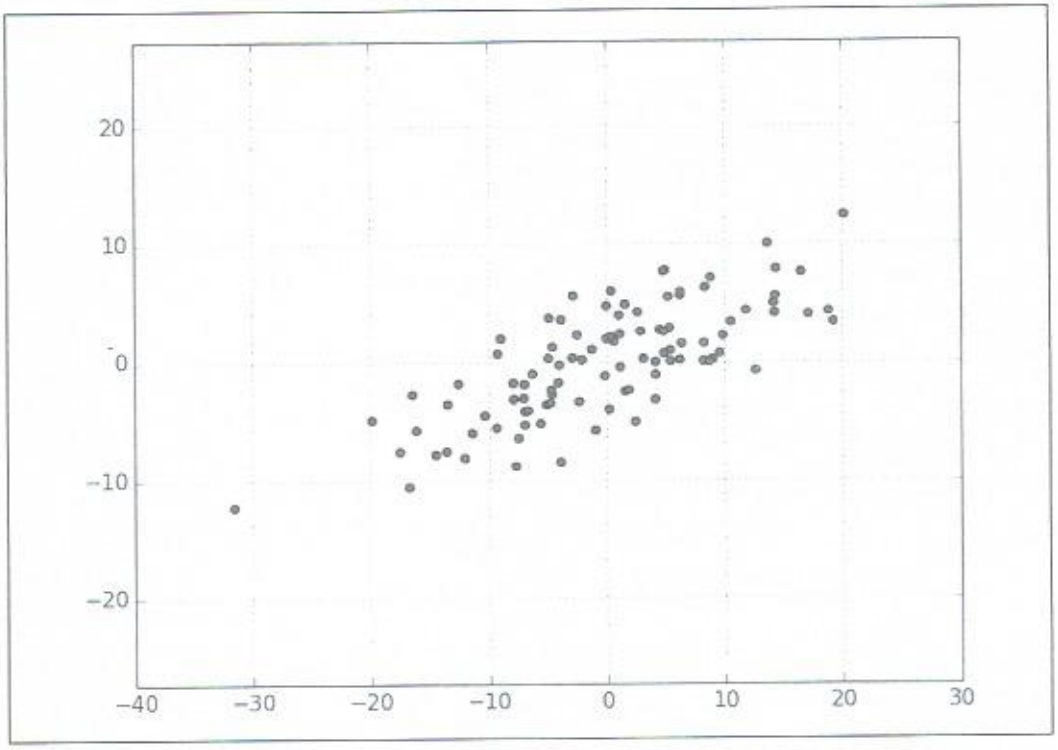

Figura 10-6. Dados após o desconto da média

Agora, dada uma matriz descontada de média X, podemos perguntar: qual é a direção que captura a maior variação nos dados?

Especificamente, dada uma direção d (um vetor de magnitude 1), cada linha x na matriz se estende `dot(x, d)` na direção de d. Cada vetor não-zero w determina uma direção se os redimensionarmos para ter magnitude 1:

```
def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

Portanto, dado um vetor não-zero *w*, podemos computar a variância do nosso conjunto de dados na direção determinada por *w*:

```
def directional_variance_i(x_i, w):
    """a variância na linha x_i na direção determinada por w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """a variância dos dados na direção determinada por w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

Gostaríamos de encontrar a direção que maximiza essa variância. Podemos fazer isso usando o gradiente descente, assim que tivermos a função gradiente:

```
def directional_variance_gradient_i(x_i, w):
    """a contribuição da linha x_1 para o gradiente da
    variância da direção w"""
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return vector_sum(directional_variance_gradient_i(x_i,w)
                      for x_i in X)
```

O componente principal é somente a direção que maximiza a função `directional_variance`:

```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),          # agora é uma função de w
        partial(directional_variance_gradient, X), # agora é uma função de w
        guess)
    return direction(unscaled_maximizer)
```

Ou, se você preferir o gradiente descendente estocástico:

```
# aqui não há "y", então passamos um vetor de Nones
# e funções que ignoram aquela entrada
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
        lambda x, _, w: directional_variance_i(x, w),
        lambda x, _, w: directional_variance_gradient_i(x, w),
        X,
        [None for _ in X],   # o "y" falso
        guess)
    return direction(unscaled_maximizer)
```

No conjunto de dados descontado da média, isso retorna a direção `[0.924, 0.383]`, que parece para capturar o eixo primário ao longo do qual a variação dos dados (Figura 10-7).

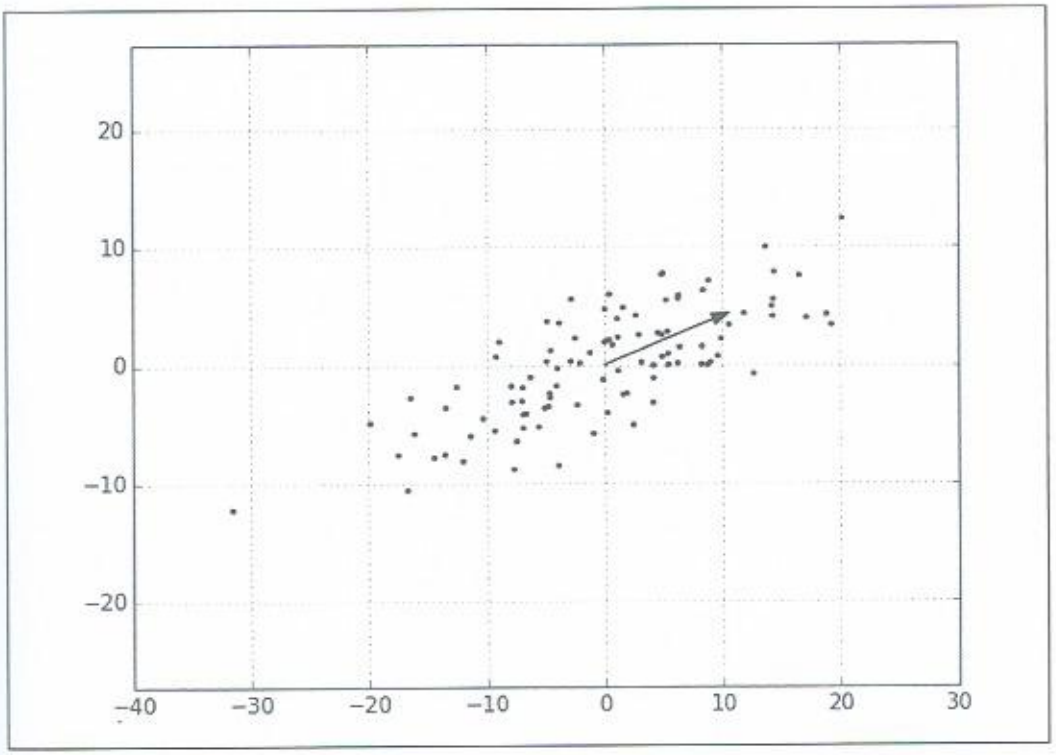

Figura 10-7. O primeiro componente principal

Uma vez que achamos a direção, que é o principal componente, podemos projetar nossos dados para encontrar os valores daquele componente:

```
def project(v, w):
    """retorna a projeção de v na direção w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

Se quisermos encontrar componentes mais distantes, primeiro temos que remover as projeções a partir dos dados:

```
def remove_projection_from_vector(v, w):
    """projeta v em w e subtrai o resultado de v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """para cada linha de X
    projeta a linha em w, e subtrai o resultado da linha"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

Como esse exemplo de conjunto de dados é bidimensional, após removermos o primeiro componente, o que sobra será efetivamente unidimensional (Figura 10-8).

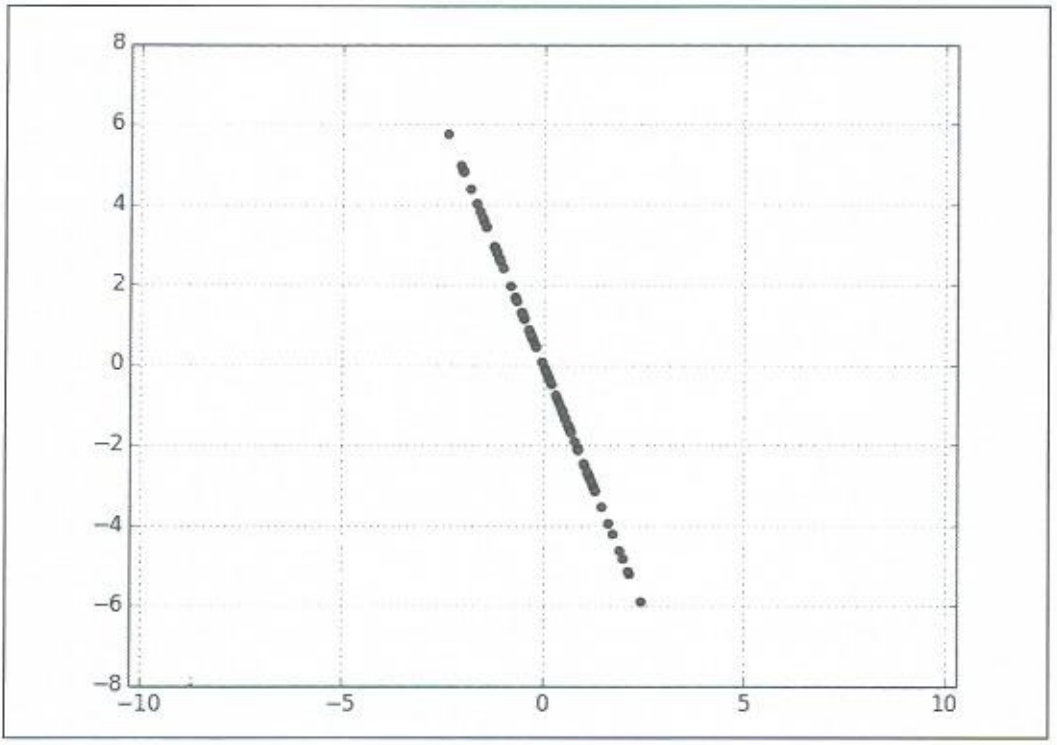

Figura 10-8. Dados após a remoção da primeira componente principal

Nesse ponto, podemos encontrar o próximo componente principal ao repetir o processo sobre o resultado `remove_projection` (Figura 10-9).

Em um conjunto de dados de dimensão mais alta, podemos encontrar tantos componentes quanto quisermos:

```
def principal_component_analysis(X, num_components):
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

Podemos então *transformar* nossos dados no espaço de dimensão mais baixa coberto pelos componentes:

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

Essa técnica é valiosa por dois motivos. Primeiro, ela pode nos ajudar a limpar nossos dados ao eliminar dimensões que são mero ruído e consolidar dimensões que são altamente correlacionadas.

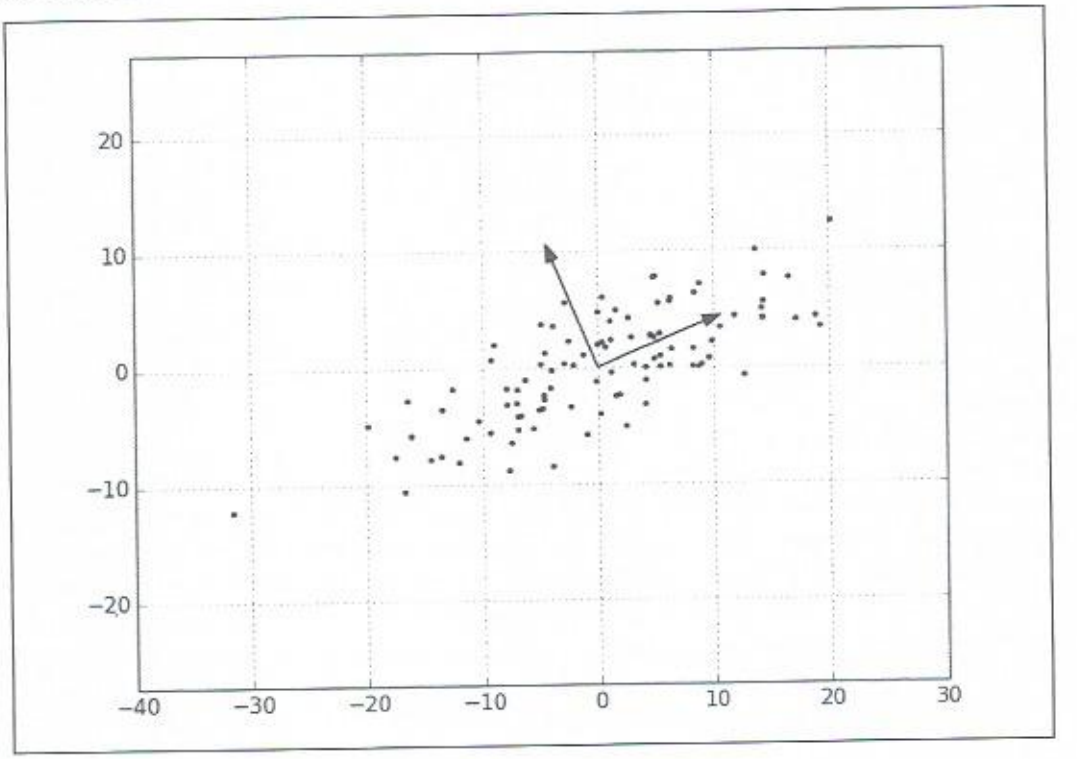

Figura 10-9. Os primeiros dois componentes principais

Segundo, após extrair uma representação de dimensão mais baixa dos nossos dados, podemos usar uma variedade de técnicas que não funcionam bem em dados de alta dimensão. Veremos alguns exemplos de tais técnicas no decorrer do livro.

Ao mesmo tempo, enquanto pode ajudar a construir modelos melhores, também pode torná-los mais difíceis de serem interpretados. É fácil entender conclusões como "cada ano extra de experiência adiciona uma média de $10k no salário". É mais difícil de entender que "cada aumento de 0,1 no terceiro componente principal adiciona uma média de $10k no salário".

Para Mais Esclarecimentos

- Como mencionamos no final do Capítulo 9, *pandas (http://pandas.pydata.org/)* provavelmente é a ferramenta primária de Python para limpar, transformar, manipular e trabalhar com dados. Todos os exemplos que fizemos à mão neste

capítulo poderiam ter sido feitos com mais simplicidade usando *pandas*. *Python for Data Analysis* (O'Reilly) é a melhor maneira de aprender *pandas*.

- scikit-learn possui uma grande variedade de funções de decomposição de matriz (*http://bit.ly/1ycOLJd*), incluindo análise de componente principal (Principal Component Analyses – PCA).

CAPÍTULO 11

Aprendizado de Máquina

Estou sempre disposto a aprender apesar de nem sempre gostar de ser ensinado.
—Winston Churchill

Muitas pessoas imaginam que data science é, em maior parte, aprendizado de máquina e que os cientistas de dados constroem, praticam e ajustam modelos de aprendizado de máquina o dia inteiro. E, novamente, muitas desas pessoas não sabem o que *é* aprendizado de máquina. Na verdade, data science é mais transformar problemas empresariais em problemas de dados e coletar, entender, limpar e formatar os dados, após o que aprendizado de máquina é praticamente uma consideração subsequente. Mesmo assim, é uma referência interessante e essencial que você deve saber a fim de praticar data science.

Modelagem

Antes de podemos falar sobre o aprendizado de máquina, precisamos falar sobre *modelos*.

O que é um modelo? É simplesmente a especificação de uma relação matemática (ou probabilística) existente entre variáveis diferentes.

Por exemplo, se você está tentado a levantar dinheiro para o seu site de rede social, talvez você precise de um *modelo de negócios* (possivelmente em uma planilha) que receba entradas como "número de usuários", "rendimento de propaganda por usuário" e "número de funcionários" e exiba como saída seu lucro anual pelos próximos anos. Um livro de receitas implica um modelo que relaciona entradas como "número de comensais" e "apetite" para as quantidades dos ingredientes necessários. E, se você já assistiu pôquer na televisão, você sabe que eles estimam a "probabilidade de ganhar" de cada jogador em tempo real baseado em um modelo que leva em consideração as cartas que foram reveladas até então e a distribuição de cartas no baralho.

O modelo de negócios é, provavelmente, baseado em relações simples de matemática: lucro é o rendimento menos as despesas, o rendimento é soma das unidades vendidas vezes o preço médio e assim por diante. O modelo do livro de receitas é baseado em tentativas e erros

— alguém foi na cozinha e experimentou combinações diferentes de ingredientes até encontrar uma que gostasse. O modelo do pôquer é baseado na teoria da probabilidade, as regras do pôquer e algumas premissas razoavelmente inócuos sobre o processo aleatório pelo qual as cartas são distribuídas.

O Que É Aprendizado de Máquina?

Todo mundo possui sua própria definição, mas usaremos *aprendizado de máquina* para nos referir à criação e ao uso de modelos que são aprendidos *a partir dos dados*. Em outros contextos isso pode ser chamado de *modelo preditivo* ou *mineração de dados*, mas vamos manter o aprendizado de máquina. Normalmente, nosso objetivo será usar os dados existentes para desenvolver modelos que possamos usar para *prever* possíveis saídas para os dados novos, como:

- Prever se uma mensagem de e-mail é spam ou não
- Prever se uma transação do cartão de crédito é fraudulenta
- Prever qual a probabilidade de um comprador clicar em uma propaganda
- Prever qual time de futebol ganhará o Super Bowl

Consideraremos os modelos *supervisionados* (nos quais existe um conjunto de dados etiquetados com a resposta correta para aprendizagem) e modelos *sem supervisão* (nos quais não existem tais etiquetas). Existem vários outros tipos como *semisupervisionados* (nos quais apenas alguns dados são etiquetados) e *online* (nos quais o modelo precisa ter um ajuste contínuo em face da chegada de novos dados), mas não serão abordados neste livro.

Agora, até mesmo na situação mais simples existe um universo inteiro de modelos que podem descrever a relação na qual estamos interessados. Na maioria dos casos, nós mesmos escolheremos uma família *parametrizada* de modelos e então usaremos os dados para aprender parâmetros que são, de certa forma, ótimos.

Por exemplo, podemos presumir que a altura de uma pessoa é (mais ou menos) uma função linear do seu peso e então usar os dados para descobrir qual função linear é essa. Ou, podemos presumir que uma árvore de decisão é uma boa maneira de identificar quais doenças nossos pacientes possuem e então usar os dados para descobrir a ótima árvore de decisão. Pelo restante do livro investigaremos famílias diferentes de modelos que podemos aprender.

Mas antes disso, precisamos entender melhor os fundamentos do aprendizado de máquina. Pelo resto do capítulo, discutiremos alguns conceitos básicos antes de chegarmos nos modelos propriamente ditos.

Sobreajuste e Sub-Ajuste

Um perigo comum em aprendizado de máquina é o *sobreajuste* — produzir um modelo de bom desempenho com os dados que você treina, mas que não lide muito bem com novos dados.

Isso pode implicar o *aprender* com base no ruído dos dados. Ou, pode implicar em aprender a identificar entradas específicas em vez de qualquer fator que sejam de fato preditivos da saída desejada.

O outro lado é o *sub-ajuste*, produzindo um modelo que não desempenha bem nem com os dados usados no treino, apesar de que, quando acontece isso, você decide que seu modelo não é bom o suficiente e continua a procurar por melhores.

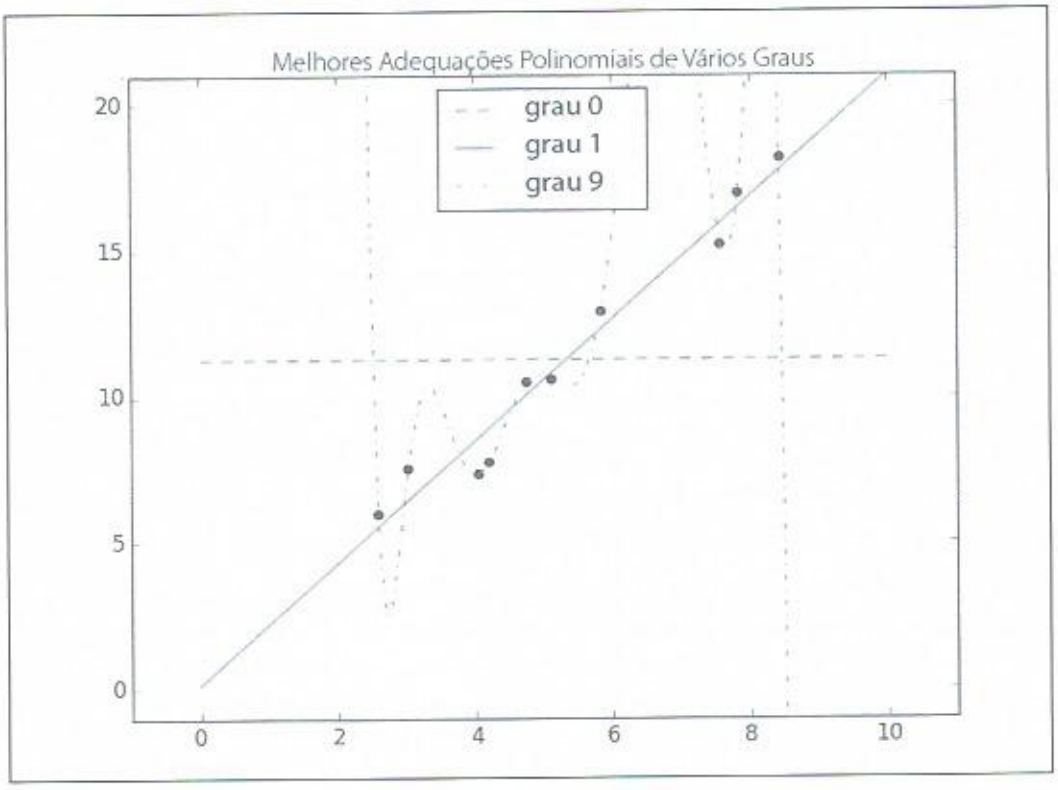

Figura 11-1. Sobreajuste e sub-ajuste

Na Figura 11-1, encaixei três polinômios em uma amostra de dados. (Não se preocupe em como, chegaremos lá nos capítulos posteriores.)

A linha horizontal mostra o melhor grau adequado polinomial 0 (isto é, constante). Ele *sub-ajusta* o dado em treinamento intensamente. O melhor ajuste por um polinômio do 9º grau (isto é, parâmetro 10) passa por todos os pontos de dados em treinamento, mas *sobreajusta* gravemente — se fossemos adquirir um pouco mais de pontos de dados, provavelmente sairiam bem errados. E a linha de 1º grau tem um bom equilíbrio — é bem próximo a cada ponto, e (se esses dados são representativos) a linha estará próxima dos novos pontos de dados também.

Evidentemente, os modelos que são muito complexos tendem ao sobreajuste e não lidam bem com dados além daqueles com os quais foram treinados. Então, como temos certeza que nossos modelos não são muito complexos? O método mais fundamental envolve o uso de dados diferentes para treinar e testar o modelo.

A maneira mais fácil de fazer isso é dividir seu conjunto de dados, a fim de que, por exemplo, dois terços dele sejam usados para treinar o modelo e depois medir o desempenho do modelo com a parte restante:

```
def split_data(data, prob):
    """divide os dados em frações [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

Com frequência, teremos uma matriz x de variáveis de entrada e um vetor y de variáveis de saída. Nesse caso, precisamos nos certificar de colocar os valores correspondentes tanto nos dados em treinamento como nos dados de teste:

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                            # par de valores correspondentes
    train, test = split_data(data, 1 - test_pct)  # divide o conjunto de pares de dados
    x_train, y_train = zip(*train)              # truque mágico de un-zip (descompactação)
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

a fim de fazer algo como:

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

Se o modelo foi sobreajustado para os dados em treinamento, então ele deve desempenhar mal sobre os dados de teste (completamente separados). De outra maneira, se ele desempenha bem sobre os dados de teste, então você pode ficar mais confiante que ele está *ajustado* em vez de *sobreajustado*.

Porém, existem duas formas de dar tudo errado.

A primeira é se existirem padrões comuns aos dados de teste e de treinamento que não seriam generalizados em um conjunto maior de dados.

Por exemplo, imagine que seu conjunto de dados consiste da atividade do usuário, uma linha por usuário por semana. Em tal caso, a maioria dos usuários aparecerá em ambos os dados de teste e de treinamento e alguns modelos talvez aprendessem a *identificar* os usuários em vez de descobrir relações envolvendo *atributos*. Não é de grande preocupação, apesar de ter acontecido comigo uma vez.

Um problema maior é se você usar a divisão de testes/treinamento não apenas para avaliar um modelo mas, também, para *escolher* entre os vários modelos. Nesse caso, embora cada modelo individual possa não ser sobreajustado, o "escolha um modelo que desempenhe melhor nos dados de teste" é um quase treinamento que faz com que o conjunto de testes funcione como um segundo conjunto de treinamento. (É claro que o modelo que tiver melhor desempenho no teste terá um melhor desempenho no conjunto de teste.)

Em uma situação como essa, você deveria dividir os dados em três partes: um conjunto de *treinamento* para construir modelos, um conjunto de *validação* para escolher entre os modelos treinados e um conjunto de *teste* para avaliar o modelo final.

Precisão

Quando não estou praticando data science, eu me aventuro na medicina. E, no meu tempo livre, eu inventei um teste simples, não-invasivo que pode ser feito com um recém-nascido que prediz — com uma precisão maior que 98% — se o recém-nascido desenvolverá leucemia. Meu advogado me convenceu de que o teste não é patenteável, portanto compartilharei com você os detalhes aqui: prever a leucemia se e somente se o nome do bebê for Luke (que se parece um pouco com o som de *leukemia*, leucemia em inglês).

Como veremos a seguir, esse teste possui mesmo mais de 98% de precisão. Apesar disso, é um teste incrivelmente estúpido e uma boa ilustração do motivo pelo qual nós não usamos "precisão" para medir a eficiência de um modelo.

Imagine construir um modelo para fazer uma avaliação *binária*. Esse e-mail é spam? Deveríamos contratar este candidato? Este viajante é um terrorista em segredo?

Dado um conjunto de dados etiquetados e um modelo preditivo, cada ponto de dados se estabelece em quatro categorias:

- Positivo verdadeiro: "Esta mensagem é spam e previmos spam corretamente."
- Positivo falso (Erro Tipo 1): "Esta mensagem não é spam, mas previmos que era."
- Negativo falso (Erro Tipo 2): "Esta mensagem é spam, mas previmos que não era."
- Negativo verdadeiro: "Esta mensagem não é spam e previmos que não era."

Representamos essa contagem em uma *matriz de confusão*:

	Spam	Não é Spam
premissa "Spam"	Positivo Verdadeiro	Positivo Falso
premissa "Não é Spam"	Negativo Falso	Negativo Verdadeiro

Vamos ver como meu teste de leucemia se encaixa nessa estrutura. Por agora, aproximadamente 5 bebês de 1000 se chamam Luke (*http://bit.ly/1CchAqt*). A incidência de sempre da leucemia é de aproximadamente 1,4%, ou 14 de cada 1000 pessoas (*http://1.usa.gov/1ycORjO*).

Se acreditarmos que esses dois fatores são independentes e aplicar meu teste "Luke para leucemia" em um milhão de pessoas, esperaríamos ver uma matriz de confusão com esta:

	leucemia	sem leucemia	total
"Luke"	70	4.930	5.000
não "Luke"	13.930	981.070	995.000
total	14.000	986.000	1.000.000

Podemos usar isso então para computar diversas estatísticas sobre o desempenho do modelo. Por exemplo, a *acurácia* é definida como a fração de premissas corretas:

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

Parece um número bem interessante. Mas, evidentemente, não é um bom teste, o que significa que não deveríamos colocar muita crença na acurácia bruta.

É comum considerar a combinação de *precisão e sensibilidade*. Exatidão significa o quão precisas nossas previsões *positivas* eram:

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

A sensibilidade mede qual fração dos positivos nossos modelos identificam:

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)       # 0.005
```

Ambos são números terríveis, refletindo um modelo terrível.

Às vezes, precisão e sensibilidade são combinados ao *F1 Score*, definido assim:

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

Essa é a *média harmônica (http://en.wikipedia.org/wiki/Harmonic_mean)* da acurácia e sensibilidade e se acha necessariamente encontrada entre elas.

Geralmente, a escolha de um modelo implica em um compromisso entre acurácia e sensibilidade. Um modelo que prevê "sim" quando se está um pouco confiante provavelmente terá

uma sensibilidade alta mas uma acurácia baixa; um modelo que prevê "sim" somente quando está extremamente confiante provavelmente terá uma sensibilidade baixa e uma acurácia alta.

Como alternativa, você pode pensar nisso como uma troca entre positivos e negativos falsos. Dizer "sim" com muita frequência trará muitos positivos falsos; dizer "não" com muita frequência trará muitos negativos falsos.

Imagine que existiram dez fatores de risco para a leucemia e que, quanto mais você os tenha, mais você estará propenso a desenvolver leucemia. Nesse caso, você pode imaginar uma continuidade de testes: "prever leucemia se houver ao menos um fator de risco", "prever leucemia se houver ao menos dois fatores de risco" e assim por diante. Conforme o limite aumenta, você aumenta a exatidão do teste (desde que as pessoas com mais fatores de risco sejam mais propensas a desenvolver a doença) e diminui a confirmação do teste (uma vez que cada vez menos pacientes da doença chegarão ao limite). Em casos como esse, escolher o limite certo é uma questão de encontrar o compromisso certo.

Compromisso entre Polarização e Variância

Outra maneira de pensar sobre o problema de sobreajuste é um compromisso entre polarização e variância.

Ambas são medidas do que aconteceria se você fosse treinar seu modelo novamente muitas vezes em diferentes conjuntos de dados de treinamento (da mesma população).

Por exemplo, o modelo polinomial de grau 0 em "Sobreajustando e Sub-Ajustando" na página 142 cometerá muitos erros para qualquer conjunto de dados em treinamento (tirados da mesma população), o que significa que ele possui uma polarização alta. Porém, quaisquer dois conjuntos de treinamento escolhidos aleatoriamente deveriam fornecer modelos similares (uma vez que quaisquer dois conjuntos de treinamento escolhidos aleatoriamente deveriam ter valores médios bem similares). Então dizemos que ele possui uma baixa *variância*. Polarização alta e variância baixa geralmente pertencem ao sub-ajuste.

Por outro lado, o modelo polinomial de grau 9 se encaixa no conjunto de treinamento com perfeição. Possui polarização baixa, mas variância muito alta (quaisquer dois conjuntos em treinamento dariam origem a modelos bem diferentes). Eles correspondem ao sobreajuste.

Pensar sobre problemas de modelos dessa forma pode ajudar a descobrir o que fazer quando seu modelo não funciona muito bem.

Se o seu modelo possui a polarização alta (o que significa que ele não possui um bom desempenho no seu conjunto em treinamento), algo mais a tentar é adicionar mais características. Ir do modelo polinomial de grau 0 em "Sobreajustando e Sub-Ajustando" na página 142 para o modelo polinomial de grau 1 foi uma grande melhoria.

Se o seu modelo tem variância alta, então você pode de modo similar *remover* características. Mas outra solução seria obter mais dados (se puder).

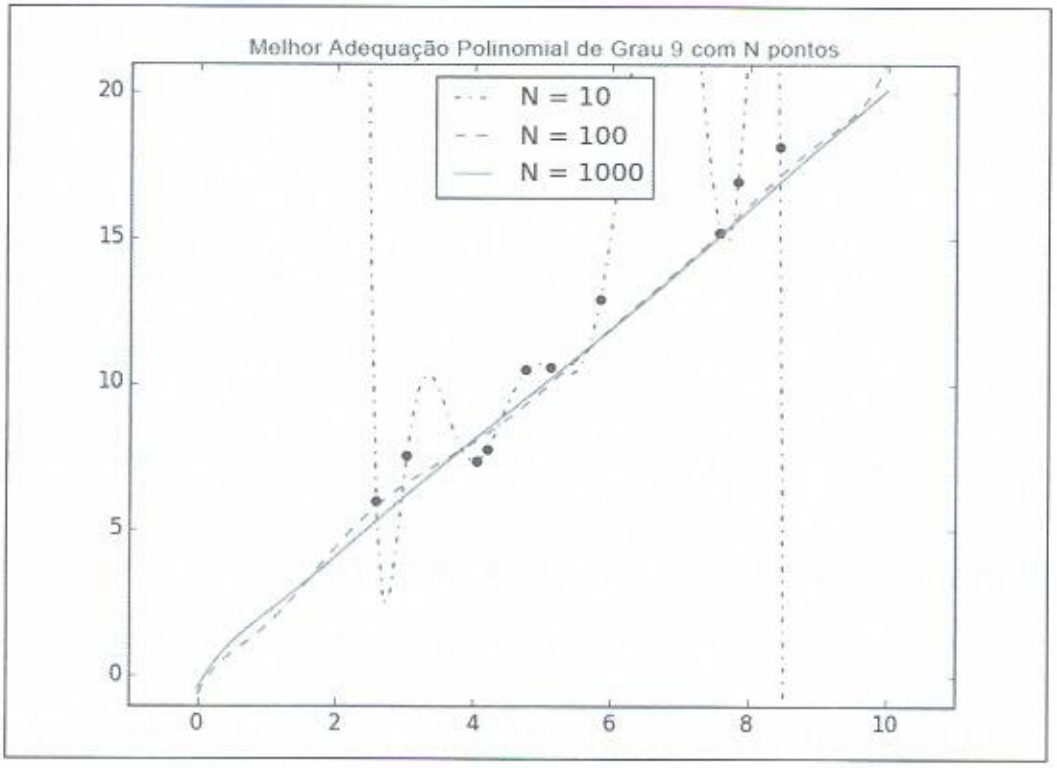

Figura 11-2. Reduzindo a variância com mais dados

Na Figura 11-2, ajustamos um polinômio de grau 9 para diferentes amostras. O modelo ajustado com base nos 10 pontos de dados está em todo lugar, como vimos anteriormente Se treinássemos com 100 pontos de dados, haveria muito menos sobreajuste. E o modelo treinado a partir dos 1000 pontos de dados é muito parecido com o modelo de grau 1.

Mantendo uma constante na complexidade do modelo, quanto mais dados há, mais difícil é para sobreajustar.

Por outro lado, mais dados não ajuda com a polarização. Se seu modelo não usa recursos suficientes para capturar regularidades nos dados, colocar mais dados não ajudará.

Recursos Extração e Seleção de Característica

Como mencionamos, quando os seus dados não tiverem características suficientes, é possível que seu modelo sub-ajuste. E quando seus dados possuem muitas características, fica fácil de sobreajustar. Mas o que são características e de onde elas vêm?

Características são quaisquer entradas que fornecemos ao nosso modelo.

No caso mais simples, as características são fornecidas apenas a você. Se você quiser prever o salário de alguém baseado em seus anos de experiência, então anos de experiência é a única característica que você possui.

(Apesar de que, como vimos em "Sobreajuste e Sub-Ajuste" na página 142, você também pode considerar adicionar anos como experiência ao quadrado, ao cubo, e assim por diante se isso ajudar a construir um modelo melhor.)

As coisas ficam mais interessantes conforme seus dados ficam mais complicados. Imagine tentar construir um filtro de spam para prever se um e-mail é lixo ou não. A maioria dos modelos não saberá o que fazer com o e-mail em si, cru, que é apenas uma coleção de texto. Você terá que extrair as características. Por exemplo:

- Esse e-mail contém a palavra "Viagra"?
- Quantas vezes a letra d aparece?
- Qual era o domínio do remetente?

A primeira é somente sim ou não, o que remete a 1 ou 0. A segunda é um número e a terceira é uma escolha de um leque de opções.

Quase sempre, extrairemos características dos nossos dados que cairão em uma dessas três categorias. Além do mais, o tipo de características que temos restringe o tipo de modelos que podemos usar.

O classificador Naive Bayes que construiremos no Capítulo 13 é destinado às características sim-ou-não, como o primeiro na lista anterior.

Modelos de regressão, como estudaremos nos Capítulos 14 e 16, requerem características numéricas (incluindo variáveis postiças (dummy) de 0s e 1s).

E as árvores de decisão, as quais veremos no Capítulo 17, podem lidar com dados numéricos ou categóricos.

Apesar de tentarmos criar características no exemplo do filtro de spam, algumas vezes tentaremos removê-las.

Por exemplo, suas entradas podem ser vetores de várias centenas de números. Dependendo da situação, talvez seja apropriado diminuir para apenas as dimensões importantes (como em "Redução da Dimensionalidade" na página 134) e usar somente um número pequeno de características. Ou talvez seria apropriado usar uma técnica (como regularização em "Regularização" na página 186) que penaliza os modelos quanto mais características eles usam.

Como escolhemos essas características? Aqui é onde uma combinação de *experiência* e *domínio de entendimento* entra em jogo. Se você recebe muitos e-mails, logo é possível que você tenha percebido a presença de certas palavras como um indicador de spam. Também

pode ter percebido que o número de d pode não ser um indicador de spam. Mas, no geral, você terá que tentar métodos diferentes, o que faz parte da diversão.

Para Mais Esclarecimentos

- Continue lendo! Os próximos capítulos são sobre famílias diferentes de modelos de aprendizado de máquina.
- O curso Machine Learning da Coursera é o MOOC (do inglês *Massive Open Online Course*) original e é um bom lugar para um entendimento mais profundo sobre aprendizado de máquina. O MOOC Machine Learning da Caltech também é bom.
- The *Elements of Statistical Learning* é um livro didático canônico que pode ser baixado gratuitamente (*http://stanford.io/1ycOXbo*). Mas esteja avisado: tem *muita* matemática.

CAPÍTULO 12

K–Vizinhos Mais Próximos

Se você quiser perturbar seus vizinhos, diga a verdade sobre eles.
—Pietro Aretino

Imagine que você está tentando prever como eu vou votar nas próximas eleições presidenciais. Se você não sabe mais nada sobre mim (e se você tiver os dados), uma abordagem lógica é considerar como meus *vizinhos* estão planejando votar. Como eu moro no centro de Seattle, meus vizinhos estão planejando votar no candidato democrata, o que sugere que o "candidato Democrata" é um bom palpite pra mim também.

Agora, imagine que você saiba mais sobre mim do que somente onde eu moro — talvez você saiba minha idade, meus rendimentos, quantos filhos eu tenho, e assim por diante. Considerando que o meu comportamento é influenciado (ou caracterizado) por tais coisas ao máximo, considerar apenas meus vizinhos que estão próximos de mim em todas essas dimensões parece ser uma classificação melhor do que considerar todos os meus vizinhos. Essa é a ideia por trás da *classificação dos vizinhos mais próximos.*

O Modelo

Os vizinhos mais próximos é um dos modelos preditivos mais simples que existe. Ele não possui premissas matemáticas e não requer nenhum tipo de maquinário pesado. Ele apenas requer:

- Uma noção de distância
- Uma premissa de que pontos que estão perto um do outro são similares

A maioria das técnicas que veremos neste livro consideram o conjunto de dados como um todo a fim de aprender padrões nos dados. Os vizinhos mais próximos, por outro lado, rejeitam muitas informações conscientemente, uma vez que a previsão para cada ponto novo depende somente de alguns pontos mais próximos.

Mais ainda, os vizinhos mais próximos provavelmente não vão lhe ajudar a entender os fatores determinantes de quaisquer fenômenos os quais você esteja considerando. Prever os meus votos baseados nos votos dos meus vizinhos não lhe diz muito sobre o que me faz votar do meu jeito, enquanto que algum modelo alternativo que prevê meu voto baseado (digamos) no meu salário e no meu estado civil talvez possa dizer.

Em uma situação geral, temos alguns pontos de dados e um conjunto de rótulos correspondentes. Os rótulos podem ser `True` e `False`, indicando se cada entrada satisfaz algumas condições como "é spam?" ou "é venenoso?" ou "seria prazeroso assistir?" Ou eles poderiam ser categorias, como classificações indicativas de filmes (L, 10, 12, 14, 16, 18). Ou eles poderiam ser nomes dos candidatos à presidência. Ou eles poderiam ser linguagens de programação preferidas.

No nosso caso, os pontos de dados serão vetores, o que significa que podemos usar a função `distance` do Capítulo 4.

Digamos que escolhemos um número *k* como 3 ou 5. Então, quando queremos classificar alguns novos pontos de dados, encontramos os pontos rotulados k mais próximos e os deixamos votar na nova saída.

Para fazer isso, precisaremos de uma função que conte os votos. Uma possibilidade é:

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

Mas isso não faz nada de inteligente com as relações. Por exemplo, imagine que estamos classificando os filmes e os cinco filmes mais próximos são classificados em L, L, 10, 10, 12. L e 10 têm dois votos. Nesse caso, temos várias opções:

- Escolher um dos vencedores aleatoriamente.
- Ponderar os votos à distância e escolher o vencedor mais votado.
- Reduzir *k* até encontrarmos um vencedor único.

Implementaremos o terceiro:

```
def majority_vote(labels):
    """presume que as etiquetas são ordenadas do mais próximo para o mais distante"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # vencedor único, então o devolve
    else:
        return majority_vote(labels[:-1]) # tenta novamente sem o mais distante
```

É certeza que esse método funcionará em algum momento, já que na pior das hipóteses reduziríamos para somente um rótulo e, nesse caso, ele vence.

Com essa função é fácil criar um classificador:

```
def knn_classify(k, labeled_points, new_point):
    """cada ponto rotulado deveria ser um par (point, label)"""

    # organiza os pontos rotulados do mais próximo para o mais distante
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # encontra os rótulos para os k mais próximos
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # e os deixa votar
    return majority_vote(k_nearest_labels)
```

Vamos ver como isso funciona.

Exemplo: Linguagens Favoritas

O resultado da primeira pesquisa de usuários da DataSciencester está de volta, e descobrimos as linguagens de programação preferidas dos nossos usuários em algumas cidades grandes:

```
# cada entrada é ([longitude, latitude], favorite_language)

cities = [([-122.3 , 47.53], "Python"),  # Seattle
          ([ -96.85, 32.85], "Java"),    # Austin
          ([ -89.33, 43.13], "R"),       # Madison
          # … e assim por diante
]
```

O vice-presidente do Envolvimento Comunitário quer saber se podemos usar esses resultados para prever a linguagem de programação preferida para lugares que não fizeram parte da pesquisa.

Como sempre, um primeiro bom passo é demarcar os dados (Figura 12-1):

```
# a chave é a linguagem, o valor é o par (longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# queremos que cada linguagem tenha marcador e cor diferentes
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# cria uma série de dispersão para cada linguagem
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
```

```
                    label=language, zorder=10)

plot_state_borders(plt)        # finge que temos uma função que faça isso

plt.legend(loc=0)              # deixa matplotlib escolher o local
plt.axis([-130,-60,20,55])     # ajusta os eixos

plt.title("Linguagens de Programação Preferidas")
plt.show()
```

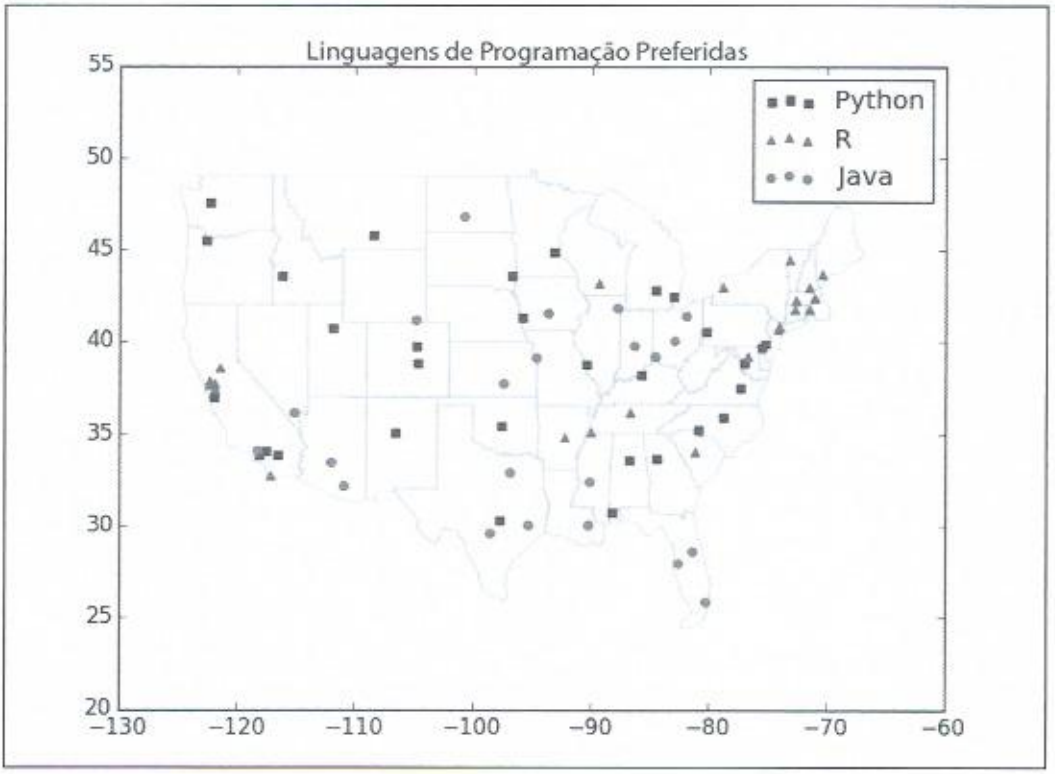

Figura 12-1. Linguagens de programação preferidas

Você deve ter notado a chamada para `plot_state_borders()`, uma função que ainda não definimos. Há uma implementação na página do livro no GitHub (*http://bit.ly/1ycP2M8*), e é um bom exercício para tentar fazer sozinho:

1. Procure na internet por algo como *fronteiras dos estados latitude longitude*.
2. Converta quaisquer dados que você encontrar em uma lista de segmentos [(long1, lat1), (long2, lat2)].
3. Use `plt.plot()` para desenhar os segmentos.

Já que os lugares mais perto tendem a gostar da mesma linguagem, os *k*-vizinhos mais próximos parecem ser uma boa escolha para um modelo preditivo.

Para começar, vamos ver o que acontece se tentarmos prever a linguagem preferida de cada cidade usando seus vizinhos em vez da própria cidade:

```
# tenta vários valores diferentes para k
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)

        if predicted_language == actual_language:
            num_correct += 1

    print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

Parece que três vizinhos mais próximos desempenham melhor, mostrando o resultado correto em 59% das vezes:

```
1 vizinho[s]: 40 certos de 75
3 vizinho[s]: 44 certos de 75
5 vizinho[s]: 41 certos de 75
7 vizinho[s]: 35 certos de 75
```

Agora podemos ver quais regiões seriam classificadas para quais linguagens dentro do esquema dos vizinhos mais próximos. Podemos fazer isso ao classificar uma rede inteira cheia de pontos, e então demarcá-las como fizemos com as cidades:

```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # or 3, or 5, or ...

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

Por exemplo, a Figura 12-2 mostra o que acontece quando olhamos apenas o vizinho mais próximo ($k = 1$).

Vemos muitas mudanças abruptas de uma linguagem para outra com limites bem acentuados. Conforme aumentamos o número de vizinhos para três, vemos regiões mais flexíveis para cada linguagem (Figura 12-3).

E conforme aumentamos os vizinhos para cinco, os limites ficam cada vez mais acentuados (Figura 12-4).

Aqui, nossas dimensões são bastante comparáveis, mas se elas não fossem você talvez quisesse redimensionar os dados como fizemos em "Redimensionando" na página 132.

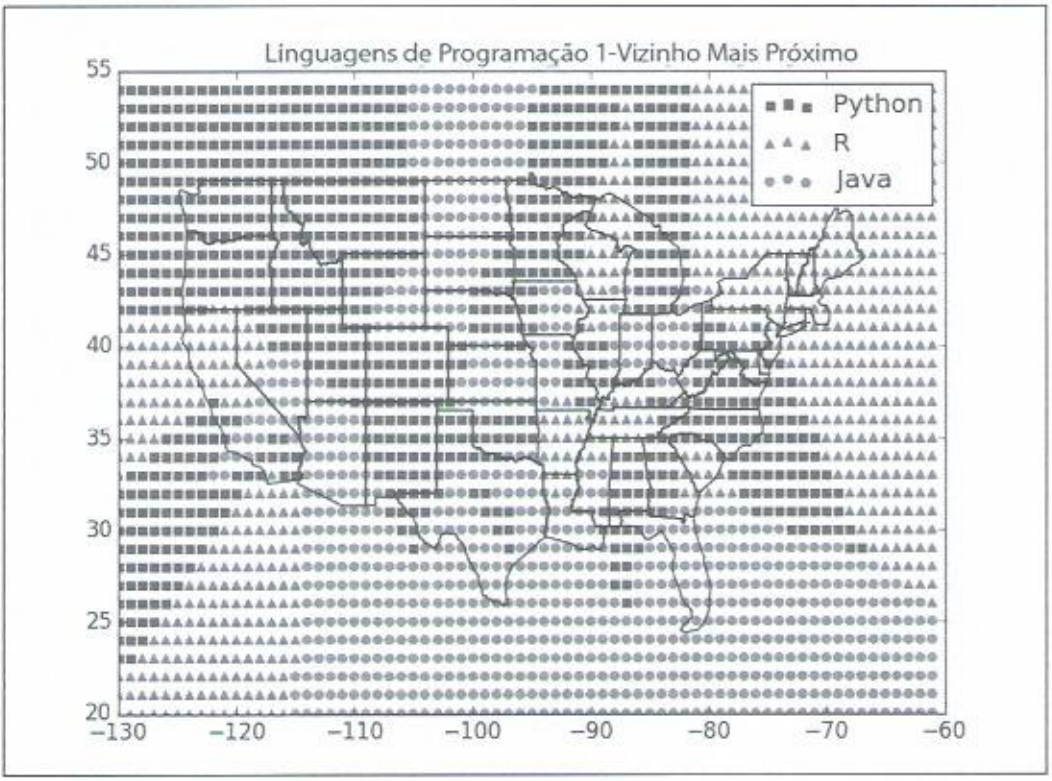

Figura 12-2. Linguagens de programação 1-vizinho mais próximo

A Maldição da Dimensionalidade

Os *k*-vizinhos mais próximos entram em perigo em dimensões mais altas graças à "maldição da dimensionalidade", que se resume ao fato de que espaços de alta dimensão são *vastos*. Os pontos em espaços de alta dimensão tendem a não ser próximos uns dos outros. Uma maneira de observar isso é gerar pares de pontos aleatórios na "unidade cubo" d-dimensional em uma variedade de dimensões e calcular a distância entre eles.

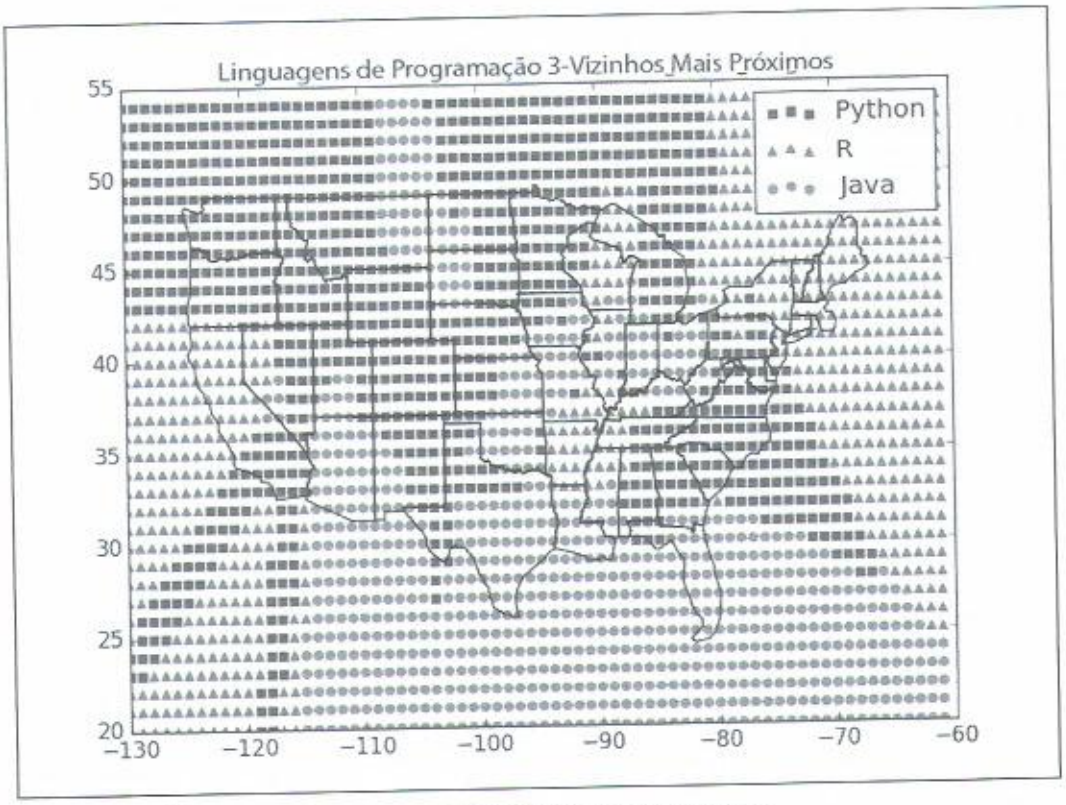

Figura 12-3. Linguagens de programação 3-vizinhos mais próximos

Gerar pontos aleatórios deve ser automático agora:

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

assim como é escrever uma função que gera as distâncias:

```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

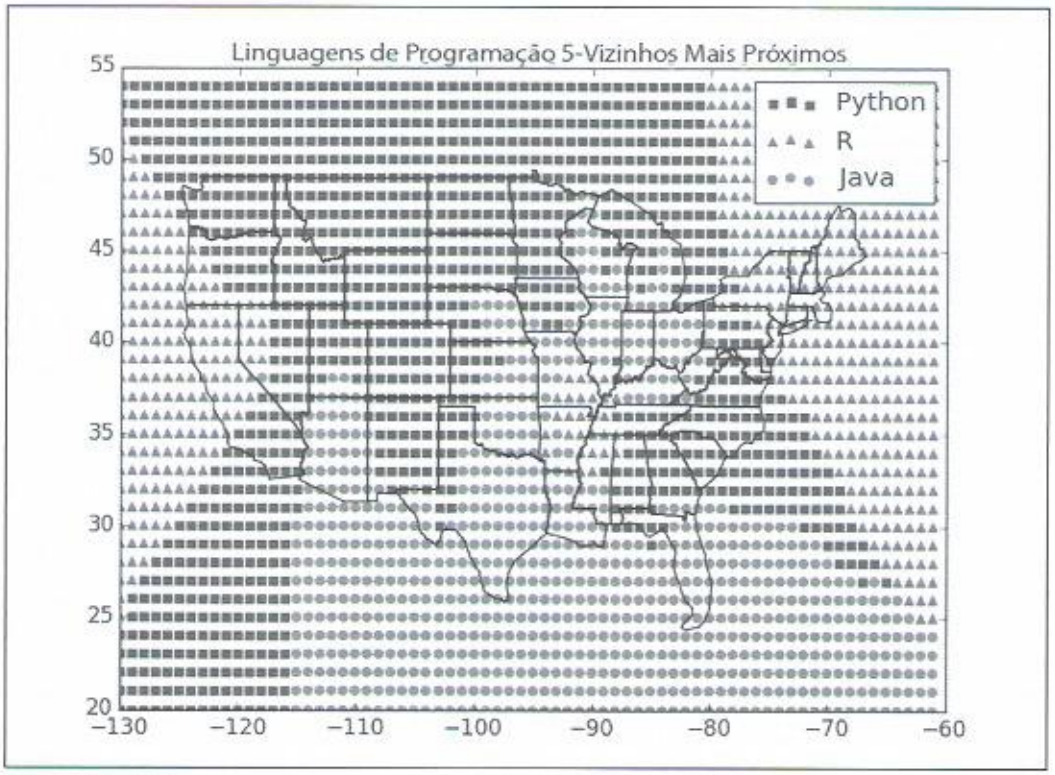

Figura 12-4. Linguagens de programação 5-vizinhos mais próximos

Para cada dimensão de 1 até 100, computaremos 10.000 distâncias e as usaremos para computar a distância média entre os pontos e a distância mínima entre os pontos de cada dimensão (Figura 12-5):

```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
  distances = random_distances(dim, 10000)  # 10.000 pares aleatórios
  avg_distances.append(mean(distances))     # rastreia a média
  min_distances.append(min(distances))      # rastreia o mínimo
```

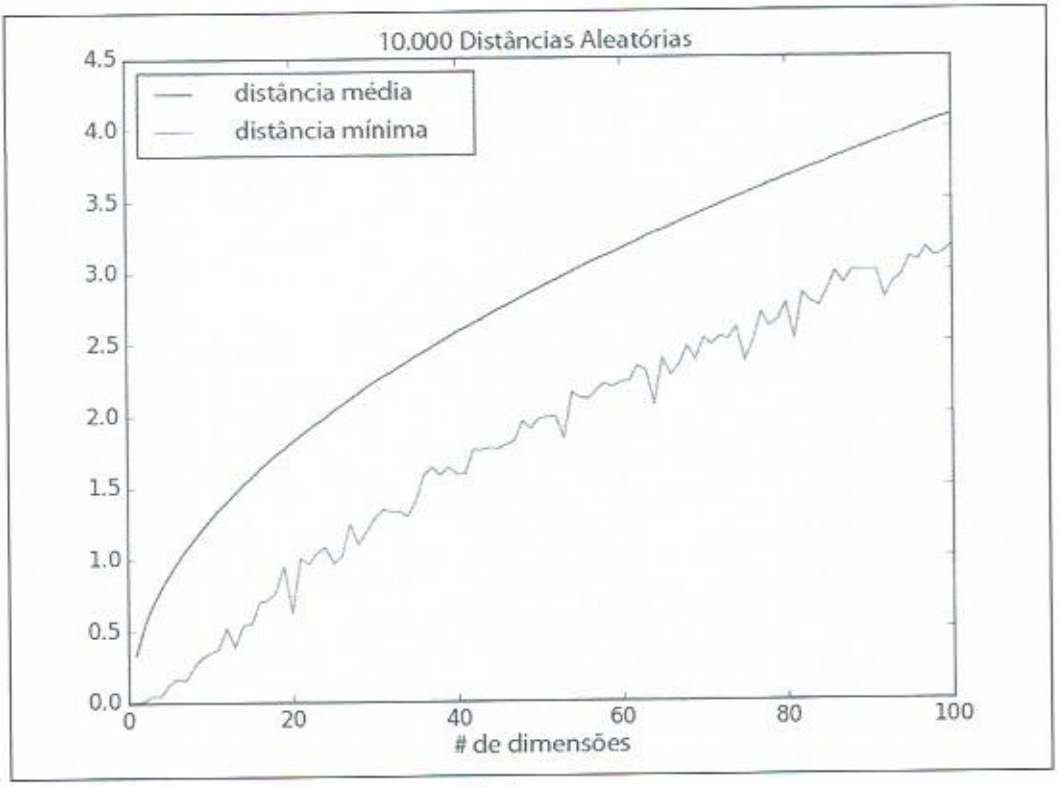

Figura 12-5. A maldição da dimensionalidade

Conforme o número de dimensões aumenta, a distância média entre os pontos também aumenta. Mas o que é mais problemático é a relação entre a distância mais próxima e a distância média (Figura 12-6):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

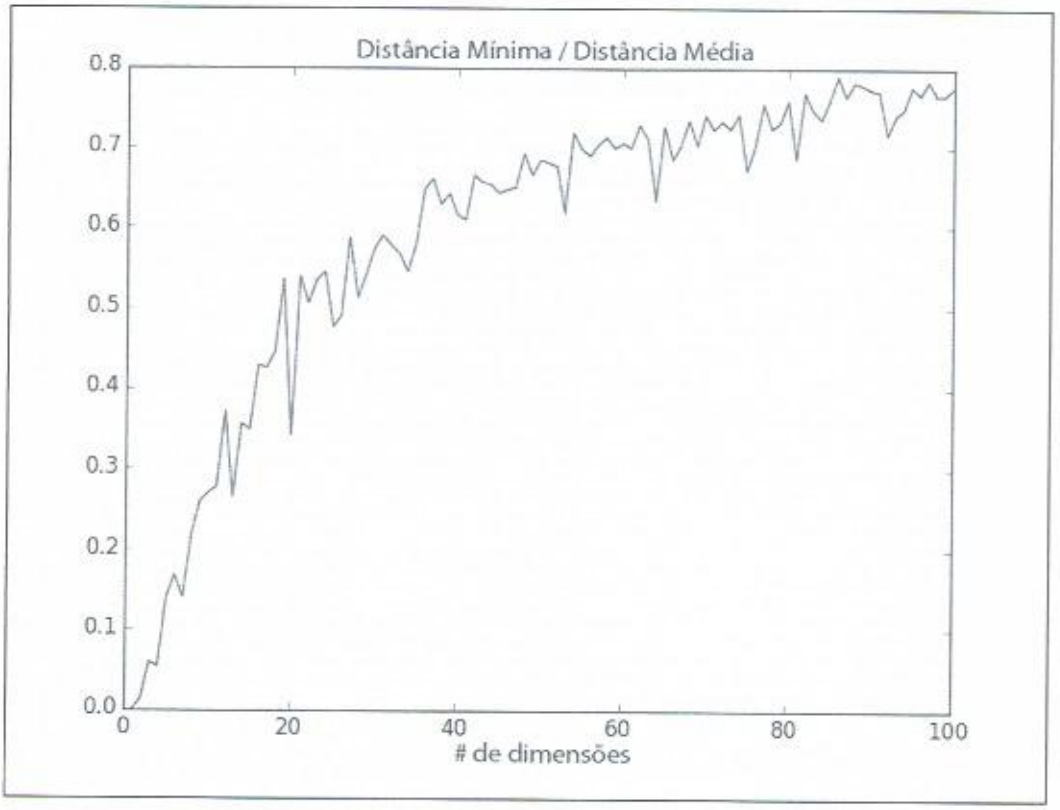

Figura 12-6. A maldição da dimensionalidade novamente

Em conjuntos de dados de baixa dimensão, os pontos mais próximos tendem a ser mais próximos do que a média. Mas os dois pontos estão próximos somente se eles estiverem próximos em todas as dimensões e cada dimensão extra — mesmo se somente um ruído — é outra oportunidade para cada ponto ser mais distante dos outros. Quando há muitas dimensões, é provável que os pontos mais próximos não sejam tão próximos quanto a média, o que significa que dois pontos estarem próximos não significa muita coisa (a menos que haja bastante estrutura em seus dados que faça com que eles se comportem como se estivessem em uma dimensão muito mais baixa).

Uma forma diferente de pensar sobre o problema envolve a dispersão de espaços de alta dimensão.

Se você escolher 50 números aleatórios entre 0 e 1, é provável que você tenha uma boa parte do intervalo unitário (Figura 12-7).

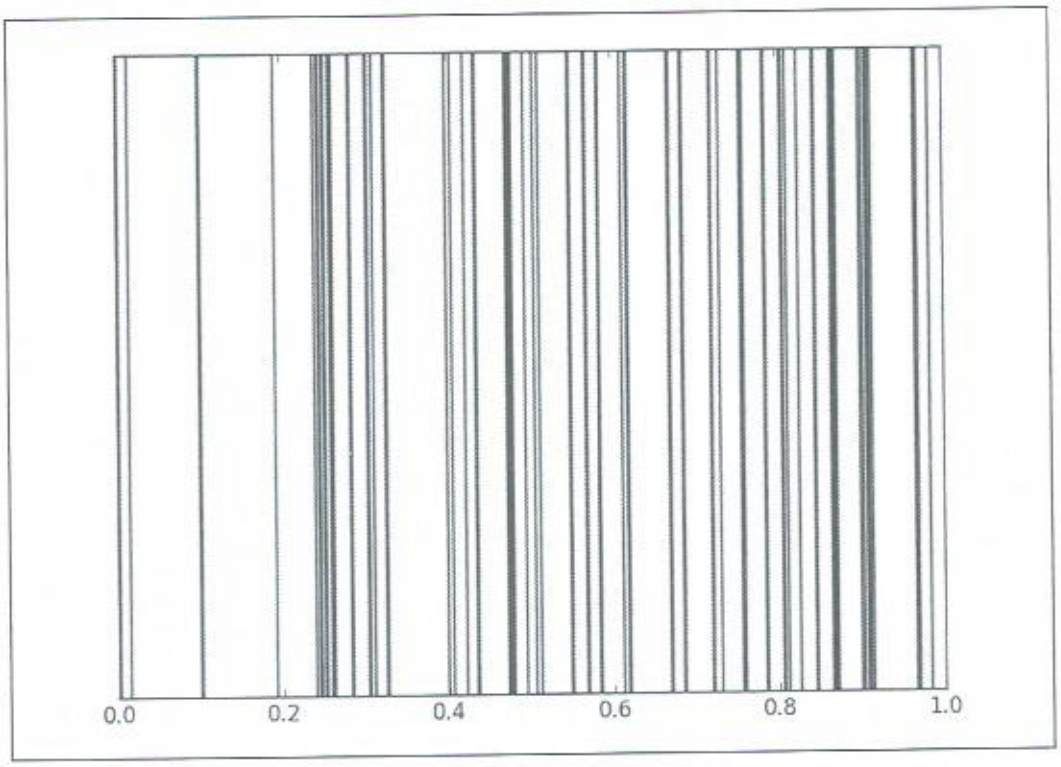

Figura 12-7. Cinquenta pontos aleatórios em uma dimensão

Se você escolher 50 pontos aleatórios no quadrado unitário, você terá menos cobertura (Figura 12-8).

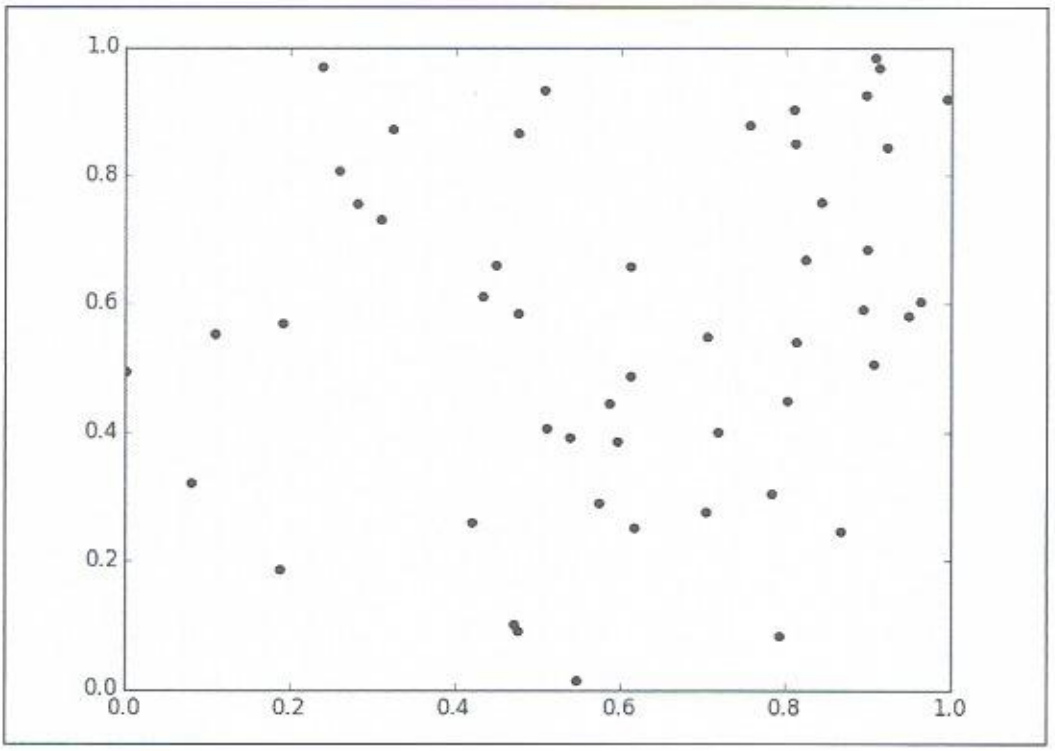

Figura 12-8. Cinquenta pontos aleatórios em duas dimensões

E em três dimensões menos ainda (Figura 12-9).

`matplotlib` não permite gráficos de quatro dimensões muito bem, portanto este é o máximo que iremos, mas você já pode ver que estão começando a ter grandes espaços vazios sem pontos perto deles. Em mais dimensões — a menos que você tenha muito mais dados — esses espaços grandes e vazios representam regiões distantes de todos os pontos que você quer usar nas suas previsões.

Então, se você estiver tentando usar os vizinhos mais próximos em uma dimensão mais alta, é provavelmente uma boa ideia fazer uma redução de dimensionalidade primeiro.

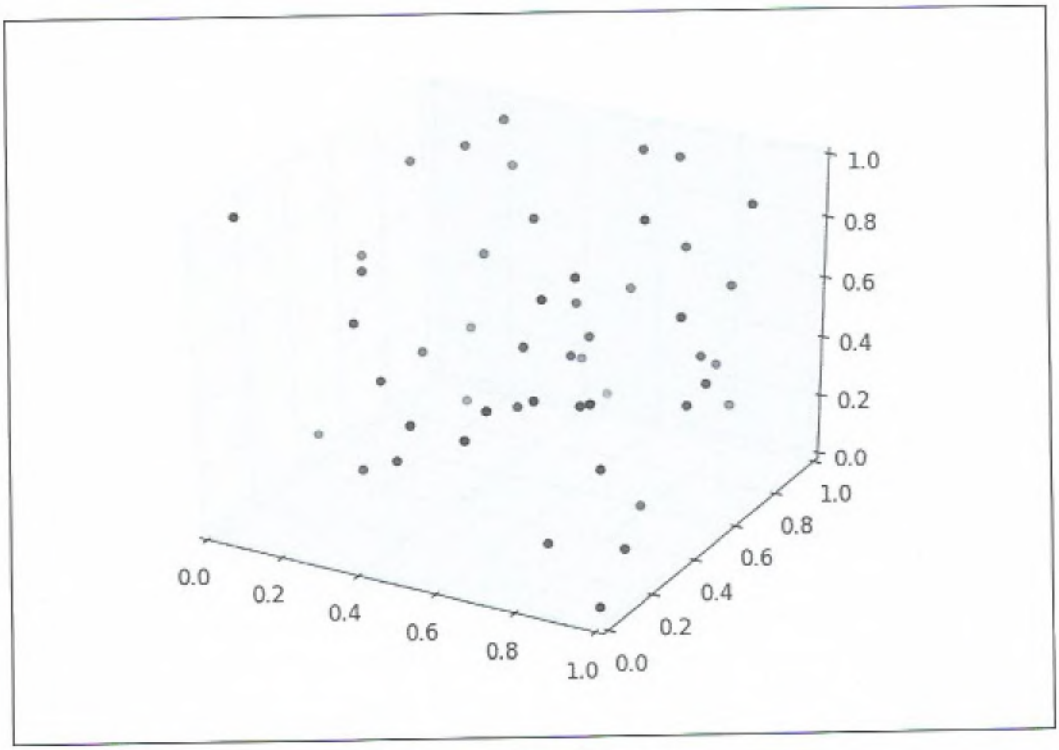

Figura 12-9. Cinquenta pontos aleatórios em três dimensões

Para Mais Esclarecimentos

scikit-learn possui muitos modelos de vizinhos mais próximos (*http://bit.ly/1ycP5rj*).

CAPÍTULO 13

Naive Bayes

"Eu prefiro o erro do entusiasmo à indiferença do bom senso."
—Anatole France

Uma rede social não é tão boa se as pessoas não conseguem se conectar. Portanto, a DataSciencester possui um atributo popular que permite que membros enviem mensagens uns aos outros. E enquanto a maioria dos membros são cidadãos responsáveis que somente enviam mensagens de "como você está?", alguns são canalhas e enviam mensagens de spam sobre esquemas para ficarem ricos, medicamentos sem receita e programas de credenciamento de data science. Seus usuários começaram a reclamar e a Vice-presidente de Mensagem pediu que você usasse data science para descobrir como filtrar essas mensagens de spam.

Um Filtro de Spam Muito Estúpido

Imagine um "universo" que consiste em receber uma mensagem escolhida ao acaso entre todas as possíveis. Deixe S ser o evento "a mensagem é spam" e V ser o evento "a mensagem contém a palavra *viagra*". Logo, o Teorema de Bayes nos diz que a probabilidade de a mensagem ser spam depende de conter a palavra *viagra*:

$$P(S \mid V) = [P(V \mid S)P(S)] / [P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)]$$

O numerador é a probabilidade de a mensagem ser spam *e* conter *viagra*, enquanto o denominador é apenas a probabilidade de a mensagem conter *viagra*. Logo, você pode pensar nesse cálculo como uma simples representação da proporção de mensagens *viagra* que são spam.

Se nós temos uma grande coleção de mensagens que sabemos que são spam, e uma grande coleção que não é spam, nós podemos facilmente calcular $P(V \mid S)$ e $P(V \mid \neg S)$. Se presu-

mirmos que qualquer mensagem é igualmente provável de ser spam ou não-spam (assim $P(S) = P(\neg S) = 0.5$), então:

$$P(S|V) = P(V|S)/[P(V|S) + P(V|\neg S)]$$

Por exemplo, se 50% das mensagens spam possuem a palavra *viagra*, mas apenas 1% das mensagens não-spam possuem, então a probabilidade de que qualquer e-mail que contenha *viagra* seja spam é:

$$0.5/(0.5 + 0.01) = 98\%$$

Um Filtro de Spam Mais Sofisticado

Agora imagine que temos um vocabulário de muitas palavras $w_1, \ldots, w_n$. Para chegar neste reino da teoria da probabilidade, escreveremos X_i para o evento "uma mensagem contém a palavra w_i". Imagine também que (por meio de um processo não-especificado-nesse-ponto) encontramos um $P(X_i|S)$ como estimativa para a probabilidade de a mensagem de spam conter a palavra *i*-ésimo, e como estimativa parecida $P(X_i|\neg S)$ para a probabilidade de uma mensagem não-spam conter a palavra *i*-ésimo.

A chave para Naive Bayes é fazer a (grande) suposição de que as presenças (ou ausências) de cada palavra são independentes umas das outras, condição para uma mensagem ser spam ou não. Intuitivamente, essa suposição significa que saber se uma certa mensagem de spam contém a palavra "viagra" ou não, não lhe dá nenhuma informação sobre a mesma mensagem conter ou não palavra "rolex". Em termos matemáticos, isso significa que:

$$P(X_1 = x_1, \ldots, X_n = x_n | S) = P(X_1 = x_1 | S) \times \ldots \times P(X_n = x_n | S)$$

Essa é uma hipótese extrema. (Há um motivo para conter "naive" (inocente) no nome da técnica.) Imagine que todo o nosso vocabulário consista *apenas* das palavras "viagra" e "rolex", e que metade de todas as mensagens de spam sejam "viagra barato" e que a outra metade seja "rolex autêntico". Nesse caso, a estimativa Naive Bayes de que uma mensagem de spam contenha ambos, "viagra" e "rolex" é:

$$P(X_1 = 1, X_2 = 1 | S) = P(X_1 = 1 | S) P(X_2 = 1 | S) = .5 \times .5 = .25$$

uma vez que afastamos a teoria de que "viagra" e "rolex" nunca acontecem juntos. Apesar da irrealidade dessa suposição, tal modelo geralmente é bem-sucedido e é usado em filtros de spam reais.

O mesmo Teorema de Bayes usado para nosso filtro de spam "apenas_viagra" nos diz que podemos calcular a probabilidade de uma mensagem ser spam usando a equação:

$$P(S|X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$$

A suposição Naive Bayes permite que computemos cada uma das probabilidades à direita simplesmente multiplicando junto as estimativas de probabilidade individual para cada palavra do vocabulário.

Na prática, você geralmente quer evitar a multiplicação de muitas probabilidades ao mesmo tempo, para evitar um problema chamado *underflow*, no qual computadores não lidam bem com números de pontos flutuantes muito próximos a zero. Relembrando da álgebra em que $\log(ab) = \log a + \log b$ e que $\exp(\log x) = x$, nós geralmente computamos $p_1 * \ldots * p_n$ como o equivalente (mas mais amigável ao ponto flutuante):

$$\exp\left(\log(p_1) + \ldots + \log(p_n)\right)$$

O único desafio restante vem com as estimativas para $P(X_i|S)$ e $P(X_i|\neg S)$, as probabilidades de uma mensagem de spam (ou não-spam) conter a palavra w_i. Se temos um número justo de mensagens rotuladas como spam ou não, a primeira tentativa óbvia é calcular $P(X_i|S)$ simplesmente como uma fração de mensagens spam contendo a palavra w_i.

No entanto, isso causa um grande problema. Imagine que em nosso vocabulário de treinamento a palavra "dado" ocorra apenas em mensagens não-spam. Então nós calcularíamos P("dado"|S)=0. O resultado é que nosso classificador Naive Bayes sempre atribuiria a probabilidade de spam 0 a *qualquer* mensagem contendo a palavra "dado", mesmo uma mensagem como "dado em viagra barato e relógios rolex autênticos". Para evitar esse problema, nós usaríamos algum tipo de suavizador.

Particularmente, escolheríamos uma *pseudocount*—k—e calcularíamos a probabilidade de ver a palavra i-ésimo em um spam como:

$$P(X_i|S) = (k + \text{número de spams contendo } w_i) / (2k + \text{número de spams})$$

Igualmente para $P(X_i|\neg S)$. Isso é, quando computamos as probabilidades de spam para a palavra i-ésimo, nós presumimos que também vimos spams adicionais k contendo a palavra e k spams adicionais não contendo.

Por exemplo, se "dado" ocorre em 0/98 documentos spam e se k é 1, nós calculamos P("dado"|S) como 1/100 = 0,001, o que permite que nosso classificador atribua alguma probabilidade spam diferente de zero para mensagens que contenham a palavra "dado".

Implementação

Agora temos todos os pedaços dos quais precisamos para o nosso classificador. Primeiro, vamos criar uma função simples para quebrar (ou *tokenize*) mensagens em palavras distintas. Primeiro converteremos cada mensagem para caixa baixa; use `re.findall()` para extrair "palavras" consistentes de letras, números e apóstrofo; e finalmente, use `set()` para pegar apenas palavras distintas:

```
def tokenize(message):
    message = message.lower()                       # converte para minúsculas
    all_words = re.findall("[a-z0-9']+", message)   # extrai as palavras
    return set(all_words)                           # remove duplicadas
```

Nossa segunda função contará as palavras em um conjunto de mensagens rotuladas para treino. Será retornado um dicionário no qual as chaves são palavras, e cujos valores são listas de dois elementos `[spam_count, non_spam_count]`, correspondentes a quantidade de vezes que vimos aquela palavra em ambas mensagens, spam e não-spam:

```
def count_words(training_set):
    """o conjunto em treinamento consiste de pares (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

Nosso próximo passo é transformar tais contas em probabilidades estimadas usando o suavizador descrito anteriormente. Nossa função retornará uma lista de triplas contendo cada palavra, a probabilidade de ver tal palavra em uma mensagem de spam e a probabilidade de vê-la em uma não-spam:

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """transforma o word_counts em uma lista de triplas
    w, p(w | spam) e p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

A última parte é usar essas probabilidades de palavras (e nossas hipóteses Naive Bayes) para atribuir probabilidades a mensagens:

```
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # itera cada palavra em nosso vocabulário
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # se "word" aparecer na mensagem,
        # adicione a probabilidade log de vê-la
        if word in message_words:
```

$$P(S|X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$$

A suposição Naive Bayes permite que computemos cada uma das probabilidades à direita simplesmente multiplicando junto as estimativas de probabilidade individual para cada palavra do vocabulário.

Na prática, você geralmente quer evitar a multiplicação de muitas probabilidades ao mesmo tempo, para evitar um problema chamado *underflow*, no qual computadores não lidam bem com números de pontos flutuantes muito próximos a zero. Relembrando da álgebra em que $\log(ab) = \log a + \log b$ e que $\exp(\log x) = x$, nós geralmente computamos $p_1 * \ldots * p_n$ como o equivalente (mas mais amigável ao ponto flutuante):

$$\exp(\log(p_1) + \ldots + \log(p_n))$$

O único desafio restante vem com as estimativas para $P(X_i|S)$ e $P(X_i|\neg S)$, as probabilidades de uma mensagem de spam (ou não-spam) conter a palavra w_i. Se temos um número justo de mensagens rotuladas como spam ou não, a primeira tentativa óbvia é calcular $P(X_i|S)$ simplesmente como uma fração de mensagens spam contendo a palavra w_i.

No entanto, isso causa um grande problema. Imagine que em nosso vocabulário de treinamento a palavra "dado" ocorra apenas em mensagens não-spam. Então nós calcularíamos P("dado"|S)=0. O resultado é que nosso classificador Naive Bayes sempre atribuiria a probabilidade de spam 0 a *qualquer* mensagem contendo a palavra "dado", mesmo uma mensagem como "dado em viagra barato e relógios rolex autênticos". Para evitar esse problema, nós usaríamos algum tipo de suavizador.

Particularmente, escolheríamos uma *pseudocount*—*k*—e calcularíamos a probabilidade de ver a palavra *i*-ésimo em um spam como:

$$P(X_i|S) = (k + \text{número de spams contendo } w_i) / (2k + \text{número de spams})$$

Igualmente para $P(X_i|\neg S)$. Isso é, quando computamos as probabilidades de spam para a palavra *i*-ésimo, nós presumimos que também vimos spams adicionais *k* contendo a palavra e *k* spams adicionais não contendo.

Por exemplo, se "dado" ocorre em 0/98 documentos spam e se *k* é 1, nós calculamos P("dado"|S) como 1/100 = 0,001, o que permite que nosso classificador atribua alguma probabilidade spam diferente de zero para mensagens que contenham a palavra "dado".

Implementação

Agora temos todos os pedaços dos quais precisamos para o nosso classificador. Primeiro, vamos criar uma função simples para quebrar (ou *tokenize*) mensagens em palavras distintas. Primeiro converteremos cada mensagem para caixa baixa; use `re.findall()` para extrair "palavras" consistentes de letras, números e apóstrofo; e finalmente, use `set()` para pegar apenas palavras distintas:

```
def tokenize(message):
    message = message.lower()                       # converte para minúsculas
    all_words = re.findall("[a-z0-9']+", message)   # extrai as palavras
    return set(all_words)                           # remove duplicadas
```

Nossa segunda função contará as palavras em um conjunto de mensagens rotuladas para treino. Será retornado um dicionário no qual as chaves são palavras, e cujos valores são listas de dois elementos `[spam_count, non_spam_count]`, correspondentes a quantidade de vezes que vimos aquela palavra em ambas mensagens, spam e não-spam:

```
def count_words(training_set):
    """o conjunto em treinamento consiste de pares (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

Nosso próximo passo é transformar tais contas em probabilidades estimadas usando o suavizador descrito anteriormente. Nossa função retornará uma lista de triplas contendo cada palavra, a probabilidade de ver tal palavra em uma mensagem de spam e a probabilidade de vê-la em uma não-spam:

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """transforma o word_counts em uma lista de triplas
    w, p(w | spam) e p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

A última parte é usar essas probabilidades de palavras (e nossas hipóteses Naive Bayes) para atribuir probabilidades a mensagens:

```
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # itera cada palavra em nosso vocabulário
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # se "word" aparecer na mensagem,
        # adicione a probabilidade log de vê-la
        if word in message_words:
```

```
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_not_spam += math.log(prob_if_not_spam)

        # se "word" não aparecer na mensagem
        # adicione a probabilidade log de não vê-la
        # que é log(1 - probabilidade de vê-la)
        else:
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

Podemos colocar tudo isso junto no nosso classificador Naive Bayes:

```
class NaiveBayesClassifier:

    def __init__(self, k=0.5):
        self.k = k
        self.word_probs = []

    def train(self, training_set):

        # conta mensagens spam e não-spam
        num_spams = len([is_spam
                         for message, is_spam in training_set
                         if is_spam])
        num_non_spams = len(training_set) - num_spams

        # roda dados de treinamento pela nossa "pipeline"
        word_counts = count_words(training_set)
        self.word_probs = word_probabilities(word_counts,
                                             num_spams,
                                             num_non_spams,
                                             self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

Testando Nosso Modelo

Um bom (e de certa forma velho) conjunto de dados é o SpamAssassin public corpus (*https://spamassassin.apache.org/publiccorpus/*). Nós veremos os arquivos prefixados com *20021010*. (No Windows, você precisará de um programa como 7-Zip (*http://www.7-zip.org/*) para descompactar e extrair os arquivos.)

Após extrair os dados (para, digamos, *C:\spam*), você deve ter três pastas: *spam*, *easy_ham* e *hard_ham*. Cada pasta contém muitos e-mails, cada qual contido em um único arquivo. Para manter tudo *bem simples*, olharemos apenas o assunto de cada e-mail.

Como identificamos a linha de assunto? Olhando pelos arquivos, todos parecem começar com "Subject:". Logo, procuraremos por isto:

```
import glob, re

# modifique com o caminho no qual você colocou os arquivos
path = r"C:\spam\*\*"

data = []

# glob.glob retorna todo nome de arquivo que combine com o caminho determinado
for fn in glob.glob(path):
    is_spam = "ham" not in fn

    with open(fn,'r') as file:
        for line in file:
            if line.startswith("Subject:"):
                # remove o primeiro "Subject:" e mantém o que sobrou
                subject = re.sub(r"^Subject: ", "", line).strip()
                data.append((subject, is_spam))
```

Agora podemos dividir os dados em dados de treinamentos e dados de teste e, então, estaremos prontos para construir um classificador:

```
random.seed(0)      # só para que receba a mesma resposta que eu
train_data, test_data = split_data(data, 0.75)

classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

E agora podemos verificar como o nosso modelo faz:

```
# triplas (subject, is_spam real, probabilidade de spam previsto)
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]

# presuma que spam_probability > 0.5 corresponde à previsão de spam
# e conta as combinações de (is_spam real, is_spam previsto)
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

Isso dá 101 positivos verdadeiros (spam classificado como "spam"), 33 positivos falsos (ham classificados como "spam"), 704 negativos verdadeiros (ham classificados como "ham") e 38 negativos falsos (spam classificados como "ham"). Isso significa que nossa acurácia é 101 / (101 + 33) = 75%, e nossa sensibilidade é 101 / (101 + 38) = 73%, que não são números ruins para um modelo tão simples.

Também é interessante olhar para os mais mal classificados:

```
# ordena spam_probability do menor para o maior
classified.sort(key=lambda row: row[2])

# as maiores probabilidades de spam previstos entre os não-spams
```

```
spammiest_hams = filter(lambda row: not row[1], classified)[-5:]

# as menores probabilidades de spam previstos entre os spams
hammiest_spams = filter(lambda row: row[1], classified)[:5]
```

As duas hams com mais jeito de spam possuem as palavras "precisa" (77 vezes mais provável de aparecer em spam), "seguro" (30 vezes mais provável de aparecer em spam) e "importante" (10 vezes mais provável de aparecer em spam).

O spam com mais jeito de ham é muito curto ("Re: garotas") para julgarmos e o segundo é uma solicitação de cartão de crédito em que a maioria das palavras não estava no conjunto de treinamento.

Podemos ver as palavras que possuem mais jeito de spam:

```
def p_spam_given_word(word_prob):
    """usa o teorema de bayes para computar p(spam | message contains word)"""

    # word_prob é uma das triplas produzidas por word_probabilities
    word, prob_if_spam, prob_if_not_spam = word_prob
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)

words = sorted(classifier.word_probs, key=p_spam_given_word)

spammiest_words = words[-5:]
hammiest_words = words[:5]
```

As palavras "money", "systemworks", "rates", "sale" e "year" são as que possuem mais spams, todas parecem relacionadas a tentar fazer as pessoas comprarem coisas. E as palavras "spamBayes", "users", "razor", "zzzzteana" e "sadev" são do tipo ham, em que a maioria parece relacionada com prevenção de spam, por mais estranho que seja.

Como poderíamos obter uma performance melhor? Uma maneira óbvia seria pegar mais dados para treinar. Existem várias maneiras de melhorar o modelo. Estas são algumas possibilidades que você pode tentar:

- Olhe o conteúdo da mensagem, não olhe somente a linha do assunto. Você deve ser cauteloso ao ver os títulos das mensagens.
- Nosso classificador leva em consideração cada palavra que aparece no conjunto de treinamento, até mesmo as palavras que só aparecem uma vez. Modifique o classificador para aceitar um limite opcional `min_count` e ignore os símbolos que não aparecem tantas vezes.
- O *tokenizer* não tem percepção de palavras similares (por exemplo, "cheap" e "cheapest"). Modifique o classificador para ter uma função `stemmer` que converte palavras para as *classes equivalentes* de palavras. Por exemplo, uma função stemmer simples pode ser:

```
def drop_final_s(word):
    return re.sub("s$", "", word)
```

Criar uma boa função stemmer é difícil. As pessoas geralmente usam a Porter Stemmer (*http://tartarus.org/martin/PorterStemmer/*).

- Mesmo que todas as nossas características sejam "mensagens contendo a palavra w_i.", não há motivo para tal. Em nossa implementação, nós pudemos acrescentar características extras como "mensagem contendo um número" criando tokens fictícios como *contains:number* e modificando o `tokenizer` para emiti-los quando necessário.

Para Mais Esclarecimentos

- Os artigos de Paul Graham, "A Plan for Spam" (*http://bit.ly/1ycPcmA*) e "Better Bayesian Filtering" (*http://bit.ly/1ycPbiy*) (são interessantes e) dão uma maior compreensão sobre a construção de filtros de spam.
- `scikit-learn` (*http://bit.ly/1ycP9ar*) contém um modelo `BernoulliNB` que implementa o mesmo algoritmo Naive Bayes que implementamos aqui, bem como outras variações do modelo.

CAPÍTULO 14

Regressão Linear Simples

A arte, como a moralidade, consiste em estabelecer um limite em algum lugar.
— G.K. Cherterton

No Capítulo 5, nós usamos a função `correlation` para medir a força do relacionamento linear entre duas variáveis. Para a maioria das aplicações, saber que tal relacionamento linear existe não é o bastante. Nós queremos conseguir entender a natureza do relacionamento. É aí que usamos regressão linear simples.

O Modelo

Lembre-se de que nós estávamos investigando o relacionamento entre o número de amigos de um usuário da DataSciencester e o tempo que ele passa no site por dia. Vamos supor que você se convenceu de que ter mais amigos *faz* as pessoas passarem mais tempo no site, mas não das explicações alternativas que discutimos.

A vice-presidente de Relacionamentos (*Engagement*) pede para você construir um modelo descrevendo essa relação. Já que você encontrou um relacionamento linear forte, um bom lugar para começar é o modelo linear.

Em particular, você cria uma hipótese de que há constantes α (alfa) e β (beta) tais que:

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

em que y_i é o número de minutos que o usuário i passa no site diariamente, x_i é o número de amigos que o usuário i possui e ε_i é um termo de erro (esperamos que pequeno) representando o fato de existirem outros fatores não contabilizados para esse simples modelo.

Supondo que determinamos tais alpha e beta, podemos fazer previsões simplesmente com:

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

Como escolhemos alpha e beta? Bom, qualquer escolha de alpha e beta nos dá uma saída prevista para cada entrada x_i. Como sabemos a verdadeira saída y_i, podemos computar o erro para cada par:

```
def error(alpha, beta, x_i, y_i):
    """erro de prever beta * x_i + alpha
    quando o valor real é y_i"""
    return y_i - predict(alpha, beta, x_i)
```

O que realmente gostaríamos de saber é o erro total sobre todo o conjunto de dados. Mas não queremos apenas adicionar erros — se a previsão para x_1 for alta demais e a previsão para x_2 for baixa demais, os erros podem apenas ser anulados.

Então, em vez disso, nós adicionamos os erros ao *quadrado*:

```
def sum_of_squared_errors(alpha, beta, x, y):
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

A *solução mínima dos quadrados* é escolher o alpha e o beta que tornarão a soma sum_of_squared_errors a menor possível.

Usando cálculo (ou a tediosa álgebra), a minimização de erro alpha e beta é dada por:

```
def least_squares_fit(x, y):
    """dados os valores em treinamento para x e y,
    encontra os valores mínimos dos quadrados de alfa e beta"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

Sem percorrer a matemática exata, vamos pensar no porquê essa pode ser uma solução razoável. A escolha de alpha simplesmente diz que quando vemos a média da variável independente x, fazemos uma previsão da média da variável dependente y.

A escolha de beta significa que, quando o valor de entrada aumenta por standard_deviation(x), a previsão aumenta por correlation(x, y) * standard_deviation(y). Quando x e y são perfeitamente correlacionados, um aumento de *um desvio padrão* em x resulta em uma divergência de um padrão de y na previsão. Quando eles são perfeitamente não correlacionados, o aumento em x resulta em uma *diminuição* da previsão. E quando a correlação é zero, beta é zero, o que significa que mudanças em x não afetarão a previsão.

É fácil aplicar isso aos dados dos valores discrepantes do Capítulo 5:

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

Isso dá valores de alfa = 22,95 e beta = 0,903. Portanto, nosso modelo diz que esperamos que um usuário com `n` amigos passe `22,95 + n * 0,903` minutos no site por dia. Ou seja, nós previmos que um usuário sem amigos na DataSciencester ainda assim passaria cerca de 23 minutos no site por dia. E, para cada amigo adicional, nós esperamos que o usuário gaste quase um minuto a mais no site por dia.

Na Figura 14-1, nós assinalamos a linha de previsão para entender como esse modelo é adequado aos dados observados.

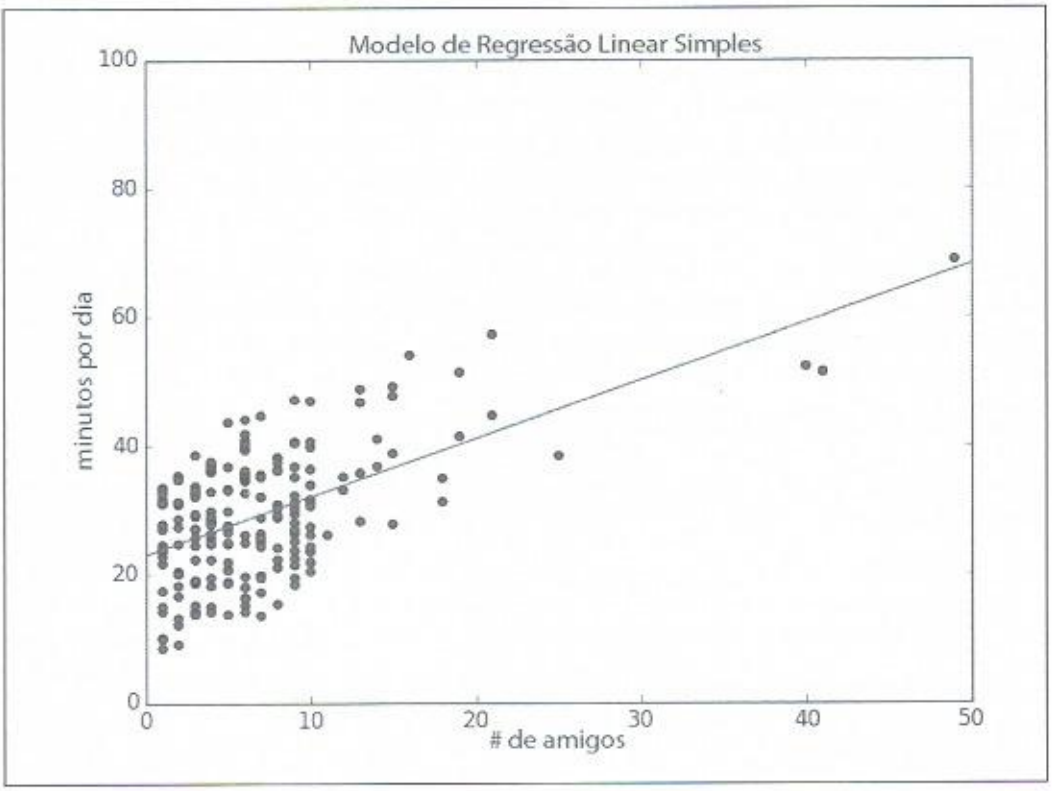

Figura 14-1. Nosso modelo linear simples

Claro, nós precisamos de uma forma melhor de descobrir quão bem conseguimos ajustar os dados em vez de olhar para o gráfico. Uma medida comum é o *coeficiente de determinação* (ou R^2) que mede a fração da variação total na variável dependente que é capturada pelo modelo:

```
def total_sum_of_squares(y):
    """a soma total dos quadrados das variações de y_i a partir de suas médias"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
```

```
    """a fração da variação em y capturada pelo modelo, que é igual a
    1 - a fração da variação em y não capturada pelo modelo"""

    return 1.0 - (sum_of_squared_errors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good)      # 0.329
```

Agora, nós escolhemos um `alpha` e `beta` que minimizaram a soma da previsão quadrada de erros. Um modelo linear que poderíamos ter escolhido é "sempre prever `mean(y)`" (correspondendo a `alpha = mean(y)` e `beta = 0`), cuja soma dos erros ao quadrado é exatamente igual à soma dos quadrados. Isso significa um R^2 de zero, o que indica um modelo que (obviamente, nesse caso) não possui um desempenho melhor do que simplesmente predizer a média.

Claramente, o modelo do menor quadrado deve ao menos ser tão bom quando aquele, o que significa que a soma dos erros ao quadrado é *no máximo* a soma dos quadrados, o que significa que R^2 deve ser pelo menos zero. E a soma dos erros ao quadrado deve ser pelo menos 0, o que significa que R^2 pode ser no máximo 1.

Quanto maior o número, melhor nosso modelo se encaixa aos dados. Aqui calculamos um R^2 de 0.329, o que nos diz que nosso modelo é apenas mais ou menos bom em ajustar os dados, e que claramente há outros fatores em jogo.

Usando o Gradiente Descendente

Se escrevermos `theta = [alpha, beta]`, também podemos resolver isso usando o gradiente descendente:

```
def squared_error(x_i, y_i, theta):
    alpha, beta = theta
    return error(alpha, beta, x_i, y_i) ** 2

def squared_error_gradient(x_i, y_i, theta):
    alpha, beta = theta
    return [-2 * error(alpha, beta, x_i, y_i),       # derivada de alpha parcial
            -2 * error(alpha, beta, x_i, y_i) * x_i] # derivada de beta parcial

# escolhe um valor aleatório para começar
random.seed(0)
theta = [random.random(), random.random()]
alpha, beta = minimize_stochastic(squared_error,
                                  squared_error_gradient,
                                  num_friends_good,
                                  daily_minutes_good,
                                  theta,
                                  0.0001)
print alpha, beta
```

Usando os mesmos dados nós conseguimos alpha = 22,93, beta = 0,905, que são muito próximos das respostas corretas.

Estimativa Máxima da Probabilidade

Por que escolhemos mínimos quadrados? Uma justificativa envolve a *estimativa máxima da probabilidade*.

Imagine que temos um modelo de dados $v_1, \ldots, v_n$ que vem de uma distribuição que depende de algum parâmetro desconhecido θ:

$$p(v_1, \ldots, v_n | \theta)$$

Se não soubéssemos teta, poderíamos voltar e pensar nessa quantidade como a *probabilidade* de θ dada a amostra:

$$L(\theta | v_1, \ldots, v_n)$$

Nessa abordagem, o θ mais provável é o valor que maximiza essa função de probabilidade; isto é, o valor que faz com que os dados observados sejam os mais prováveis. No caso de uma distribuição contínua, na qual temos uma função de distribuição de probabilidade no lugar de uma função de massa de probabilidade, nós podemos fazer a mesma coisa.

De volta à regressão. Uma suposição que geralmente é feita sobre o modelo de regressão simples é que os erros de regressão são normalmente distribuídos com média 0 e algum (conhecido) desvio padrão σ. Se esse for o caso, então a probabilidade baseada em ver um par `(x_i, y_i)` é:

$$L(\alpha, \beta | x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-(y_i - \alpha - \beta x_i)^2 / 2\sigma^2\right)$$

A probabilidade baseada em todo o conjunto de dados é o produto de probabilidades individuais, que é maior precisamente quando `alpha` e `beta` são escolhidos para minimizar a soma dos erros quadrados. Isto é, nesse caso, minimizar a soma dos erros quadrados é equivalente a maximizar a probabilidade dos dados observados.

Para Mais Esclarecimentos

Continue lendo sobre regressão múltipla do Capítulo 15!

CAPÍTULO 15

Regressão Múltipla

Eu não olho para um problema e coloco variáveis que não o afetam.
—Bill Parcells

Mesmo que a vice-presidente esteja impressionada com seu modelo preditivo, ela acha que você pode fazer melhor. Para isso, você colheu dados adicionais: para cada um dos seus usuários, você sabe quantas horas ele trabalha por dia e se ele tem um PhD. Você gostaria de usar esses dados adicionais para melhorar seu modelo.

Desta forma, você cria uma hipótese de um modelo linear com mais variáveis independentes:

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \beta_3 \text{phd} + \varepsilon$$

Obviamente, se um usuário tem PhD não é um número, mas — como falamos no Capítulo 11 — nós podemos introduzir uma *variável fictícia* igual a 1 para usuários com PhD e 0 para usuários sem PhD, o que é tão numérico quanto as outras variáveis.

O Modelo

Lembre-se de que no Capítulo 14 nós adaptamos um modelo da forma:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Agora imagine que cada entrada x_i não é um único número mas um vetor de k números $xi1$, ..., xik. O modelo de regressão múltipla presume que:

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

Em regressão múltipla o vetor de parâmetros geralmente é chamado de β. Nós queremos que isso inclua o termo constante também, o que nós podemos atingir adicionando uma coluna de uns aos nossos dados:

```
beta = [alpha, beta_1, ..., beta_k]
```

e:

```
x_i = [1, x_i1, ..., x_ik]
```

Então nosso modelo é:

```
def predict(x_i, beta):
    """presume que o primeiro elemento de cada x_i é 1"""
    return dot(x_i, beta)
```

Nesse caso em particular, nossa variável independente `x` será uma lista de vetores, cada um se parecendo com:

```
[1,    # termo constante
 49,   # número de amigos
 4,    # horas de trabalho por dia
 0]    # não tem PhD
```

Mais Suposições do Modelo dos Mínimos Quadrados

Há mais algumas suposições que são exigidas para que esse modelo (e nossa solução) faça sentido.

A primeira é que as colunas de x sejam *linearmente independentes* — que não haja como escrever qualquer um como uma soma ponderada dos outros. Se esta suposição falhar, é impossível estimar `beta`. Para ver isso em um caso extremo, imagine que temos um campo extra `num_acquaintances` em nossos dados que para cada usuário fosse exatamente igual a `num_friends`.

Então, começando com qualquer `beta`, se adicionarmos *qualquer* quantidade ao coeficiente `num_friends` e subtrair a mesma quantidade ao coeficiente `num_acquaintances`, as previsões do modelo permanecerão as mesmas. O que significa que não há como encontrar *o* coeficiente para `num_friends`. (Geralmente, violações a esta suposição não são tão óbvias.)

A segunda hipótese importante é que as colunas de x não estão correlacionadas com os erros ε. Se isto calhar de acontecer, nossas estimativas de `beta` estarão sistematicamente erradas.

Por exemplo, no Capítulo 14, nós construímos um modelo preditivo em que cada amigo adicional estava associado com 0,90 minutos extras no site.

Imagine que também é o caso que:

- Pessoas que trabalham mais horas passam menos tempo no site.

- Pessoas com mais amigos tendem a trabalhar mais horas.

Isto é, imagine que o modelo real é:

$$\text{minutos} = \alpha + \beta_1 \text{ amigos} + \beta_2 \text{ horas de trabalho} + \varepsilon$$

e que horas de trabalho e amigos são positivamente correlacionados. Neste caso, quando minimizarmos os erros de um modelo variável:

$$\text{minutos} = \alpha + \beta_1 \text{ amigos} + \varepsilon$$

nós subestimaremos β_1.

Pense no que aconteceria se nós fizéssemos previsões usando um modelo de uma única variável com o valor "real" de β_1. (Isto é, o valor que aparece a partir da minimização de erros é o que chamamos de modelo "real".) As previsões tenderiam a ser muito pequenas para usuários que trabalham muitas horas e muito grandes para os que trabalham poucas horas, pois $\beta_2 > 0$ e nós "esquecemos" de inclui-lo. Porque horas de trabalho é positivamente correlacionada com o número de amigos, isto significa que as previsões tendem a ser muito pequenas para usuários com muitos amigos e muito grandes para usuários com poucos amigos.

O resultado disso é que podemos reduzir os erros (no modelo de única variável) diminuindo nossa estimativa de β_1, que significa que o β_1 que minimiza o erro é menor do que o valor "real". E, em geral, quando variáveis independentes são correlacionadas com erros como aqui, nossa solução dos mínimos quadrados nos dará uma estimativa polarizada de β.

Ajustando o Modelo

Como fizemos no modelo linear simples, escolheremos `beta` para minimizar a soma dos erros quadrados. Encontrar a solução exata não é tão simples de fazer a mão, o que significa que precisamos usar o gradiente descendente. Começaremos criando uma função de erro a minimizar. Para o gradiente descendente aleatório, queremos apenas o erro quadrado correspondente a uma simples previsão:

```
def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2
```

Se você sabe cálculo, você pode computar:

```
def squared_error_gradient(x_i, y_i, beta):
    """o gradiente (com respeito a beta)
    correspondente ao i-ésimo termo de erro quadrado"""
    return [-2 * x_ij * error(x_i, y_i, beta)
            for x_ij in x_i]
```

Caso contrário, você precisará acreditar na minha palavra.

Neste momento, estamos prontos para encontrar o beta ótimo usando o gradiente descendente aleatório:

```
def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)

random.seed(0)
beta = estimate_beta(x, daily_minutes_good) # [30.63, 0.972, -1.868, 0.911]
```

Isso significa que nosso modelo se parece com isso:

minutos = 30,63 + 0,972 amigos – 1,868 horas de trabalho + 0,911 PhD

Interpretando o Modelo

Você deveria pensar nos coeficientes dos modelos como representantes de estimativas dos impactos de cada fator com-todos-os-demais-mantidos-constantes. O restante sendo igual, cada amigo adicional corresponde a um minuto extra passado no site a cada dia. O restante sendo igual, cada hora adicional no trabalho de um usuário corresponde a aproximadamente dois minutos menos gastos no site por dia. O restante sendo igual, ter um PhD é associado a passar um minuto extra no site a cada dia.

Isso não nos diz (diretamente) nada a respeito das interações entre as variáveis. É possível que o efeito de horas de trabalho seja diferente para pessoas com muitos amigos do que para pessoas com poucos amigos. Este modelo não captura isso. Uma forma de lidar com tal caso é inserir uma nova variável que seja o *produto* de "amigos" e "horas de trabalho". Isso efetivamente permite que o coeficiente de "horas de trabalho" aumente (ou diminua) conforme o número de amigos aumenta.

Ou é possível que quanto mais amigos você tem mais tempo você passe no site *até certo ponto*, e após isso mais amigos fazem com que você passe menos tempo no site. (Talvez com muitos amigos a experiência seja demais?) Nós poderíamos tentar capturar isso em nosso modelo adicionando outra variável que seja o *quadrado* do número de amigos.

Uma vez que começamos a adicionar variáveis, precisamos nos preocupar se os coeficientes são "importantes". Não há limites para os números de produtos, logs, quadrados e potências de Louis Grace que podemos adicionar.

O Benefício do Ajuste

Mais uma vez, nós podemos olhar para R^2, que agora aumentou para 0,68:

```
def multiple_r_squared(x, y, beta):
    sum_of_squared_errors = sum(error(x_i, y_i, beta) ** 2
                                for x_i, y_i in zip(x, y))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

Lembre-se, entretanto, que adicionar novas variáveis a uma regressão irá *necessariamente* aumentar R^2. Afinal, o modelo de regressão simples é apenas o caso especial do modelo de regressão múltipla em que os coeficientes de "horas de trabalho" e "PhD" são iguais a 0. O melhor modelo de regressão múltipla terá necessariamente um erro pelo menos tão pequeno quanto aquele.

Por isso, em regressão múltipla, nós também precisamos ver os *erros padrões* dos coeficientes, que medem quão certos estamos em nossas estimativas para cada β_i. A regressão como um todo pode ajustar nossos dados muito bem, mas se algumas das variáveis independentes forem correlacionadas (ou irrelevantes), seus coeficientes podem não *significar* tanto.

A abordagem típica para medir estes erros começa com outra suposição — que erros ε_i são variáveis independentes aleatórias normais com média 0 e algum (desconhecido) desvio padrão σ compartilhado. Neste caso, nós (ou, mais provavelmente, nosso software de estatística) podemos usar a álgebra linear para encontrar o erro padrão para cada coeficiente. Quanto maior for, menos certeza tem nosso modelo sobre o coeficiente. Infelizmente, não podemos fazer esse tipo de álgebra linear do zero.

Digressão: A Inicialização

Imagine que temos um modelo de n pontos de dados gerados por alguma distribuição (desconhecida por nós:

```
data = get_sample(num_points=n)
```

No Capítulo 5, escrevemos uma função `median` para computar a mediana dos dados observados, a qual podemos usar como estimativa da própria mediana da distribuição.

Mas quão confiantes podemos estar com relação à nossa estimativa? Se todos os dados no exemplo são próximos a 100, parece que a mediana é próxima a 100. Se aproximadamente metade dos dados no modelo forem próximos de 0 e a outra metade próxima de 200, então não podemos estar tão certos sobre a mediana.

Se pudéssemos pegar novos modelos repetidamente, poderíamos computar a mediana para cada e ver a distribuição delas. Geralmente não podemos. O que podemos fazer é inicializar novos conjuntos de dados escolhendo *n* pontos de dados *com substituição* a partir de nossos dados e então computar as medianas destes conjuntos de dados sintéticos:

```
def bootstrap_sample(data):
    """amostra aleatoriamente len(dados) elementos com substituição"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data, stats_fn, num_samples):
    """avalia stats_fn em num_samples amostra de inicialização a partir dos dados"""
    return [stats_fn(bootstrap_sample(data))
            for _ in range(num_samples)]
```

Por exemplo, considere os dois seguintes conjuntos de dados:

```
# 101 pontos todos muito próximos de 100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# 101 pontos, 50 próximos de 0, 50 próximos de 200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

Se você computar a mediana para cada, ambas serão muitos próximas de 100. Entretanto, se você olhar para:

```
bootstrap_statistic(close_to_100, median, 100)
```

você verá em sua maioria números próximos de 100. Enquanto que, se você olhar para:

```
bootstrap_statistic(far_from_100, median, 100)
```

você verá muitos números próximos de 0 e muitos próximos de 200.

O desvio padrão do primeiro conjunto de medianas é próximo de 0 enquanto que o do segundo é próximo de 100. Este caso extremo seria muto fácil de perceber inspecionando manualmente os dados mas, no geral, isso não é verdade.

Erros Padrões de Coeficientes de Regressão

Nós podemos usar a mesma abordagem para calcular os erros padrões dos nossos coeficientes de regressão. Nós repetidamente tiramos uma amostra de inicialização dos nossos dados e calculamos `beta` baseado naquela amostra. Se o coeficiente correspondente a uma das variáveis independentes (digamos `num_friends`) não variar muitos pelos modelos, nós podemos ficar confiantes que nossa estimativa está relativamente correta. Se o coeficiente variar muito, não podemos ficar tão confiantes assim.

O único detalhe é que, antes da amostragem, precisamos compactar (zip) nossos dados x e y para certificar-nos de que valores correspondentes de variáveis independentes e dependentes sejam amostrados juntos. Isso significa que `bootstrap_sample` retornará uma lista de pares (`x_i, y_i`), que precisará se reagrupar em `x_sample` e `y_sample`.

```
def estimate_sample_beta(sample):
    """amostra é uma lista de pares (x_i, y_i)"""
    x_sample, y_sample = zip(*sample) # truque mágico para descompactar
    return estimate_beta(x_sample, y_sample)

random.seed(0) # para que você consiga o mesmo resultado que eu

bootstrap_betas = bootstrap_statistic(zip(x, daily_minutes_good),
                                      estimate_sample_beta,
                                      100)
```

E após isso podemos calcular o desvio padrão de cada coeficiente:

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

# [1,174,    # termo constante, erro real = 1,19
#  0,079,    # num_friends,     erro real = 0,080
#  0,131,    # desempregado,    erro real = 0,127
#  0,990]    # phd,             erro real = 0,998
```

Nós podemos usar isso para testar hipóteses, como "β_i é igual a zero?" de acordo com a hipótese nula $\beta_i = 0$ (e com nossas outras premissas sobre a distribuição de ε_i), a estatística:

$$t_j = \widehat{\beta}_j / \widehat{\sigma}_j$$

que é nossa estimativa de β_i, dividida pela nossa estimativa de erro padrão, segue uma *distribuição t de student* com "$n - k$ pontos de liberdade".

Se tivéssemos a função `students_t_cdf` poderíamos computar valores p para cada coeficiente mínimo quadrado para indicar a probabilidade de observarmos tal valor se o coeficiente real fosse zero. Infelizmente, nós não temos tal função. (Mas teríamos se não estivéssemos trabalhando a partir do zero.)

Entretanto, conforme os graus de liberdade aumentam, a distribuição t fica mais perto de um padrão normal. Em uma situação como essa, em que n é muito maior que k, nós podemos usar `normal_cdf` e ainda nos sentir bem:

```
def p_value(beta_hat_j, sigma_hat_j):
    if beta_hat_j > 0:
        # se o coeficiente é positivo, precisamos computar duas vezes a
        # probabilidade de ver um valor ainda *maior*
```

```
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # caso contrário, duas vezes a probabilidade de ver um valor *menor*
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

p_value(30.63, 1.174)    # ~0   (termo constante)
p_value(0.972, 0.079)    # ~0   (num_friends)
p_value(-1.868, 0.131)   # ~0   (work_hours)
p_value(0.911, 0.990)    # 0.36 (phd)
```

(Em uma situação não como essa, nós estaríamos usando um programa estatístico que sabe tanto computar a distribuição t quanto os erros padrões exatos.)

Enquanto a maioria dos coeficientes possuem valores p pequenos (sugerindo que eles realmente não são zeros), o coeficiente para "PhD" não é "significantemente" diferente de zero, o que torna possível que o coeficiente de "PhD" seja aleatório.

Em cenários de regressão mais elaborados, às vezes você quer testar hipóteses mais elaboradas sobre os dados, como "pelo menos um de β_1 não é zero" ou "β_1 é igual a β_2 e β_3 é igual a β_4", que você pode fazer com um *teste F*, que está fora do escopo deste livro.

Regularização

Na prática, você geralmente aplicará regressão linear em conjuntos de dados com grandes quantidades de variáveis. Isso cria algumas rugas extras. Primeiro, quanto mais variáveis você usar, maior a probabilidade de você sobreajustar seu modelo ao conjunto de treinamento. E segundo, quanto mais coeficientes não-zero você tiver, mais difícil será entendê-los. Se o objetivo é *explicar* algum fenômeno, um modelo pequeno com três fatores pode ser mais útil do que um modelo um pouco melhor com centenas.

A *regularização* é uma abordagem na qual nós adicionamos ao termo de erro uma penalidade que aumenta `beta` aumenta. Então nós minimizamos o erro e a penalidade combinadas. Quanto mais importância dermos em termos de penalidade, mais desencorajamos coeficientes grandes.

Por exemplo, em *regressão de cumeeira* (*ridge*), adicionamos uma penalidade proporcional à soma dos quadrados de `beta_i`. (Exceto que nós tipicamente não penalizamos `beta_0`, o termo constante.)

```
# alpha é um *hiperparâmetro* que controla quão severa a penalidade é
# às vezes é chamado de "lambda" mas isso já tem um significado em Python
def ridge_penalty(beta, alpha):
  return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x_i, y_i, beta, alpha):
```

```
    """estimativa de erro mais a penalidade ridge sobre beta"""
    return error(x_i, y_i, beta) ** 2 + ridge_penalty(beta, alpha)
```

que você pode então colocar o gradiente descendente como sempre:

```
def ridge_penalty_gradient(beta, alpha):
    """gradiente somente de penalidade ridge"""
    return [0] + [2 * alpha * beta_j for beta_j in beta[1:]]

def squared_error_ridge_gradient(x_i, y_i, beta, alpha):
    """gradiente correspondente ao i-ésimo termo de erro quadrado
    incluindo a penalidade ridge"""
    return vector_add(squared_error_gradient(x_i, y_i, beta),
                      ridge_penalty_gradient(beta, alpha))

def estimate_beta_ridge(x, y, alpha):
    """usa o gradiente descendente para encaixar uma regressão ridge
    com penalidade alfa"""
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(partial(squared_error_ridge, alpha=alpha),
                               partial(squared_error_ridge_gradient,
                                       alpha=alpha),
                               x, y,
                               beta_initial,
                               0.001)
```

Com `alpha` em zero, não há penalidade e conseguimos os mesmos resultados de antes:

```
random.seed(0)
beta_0 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.0)
# [30.6, 0.97, -1.87, 0.91]
dot(beta_0[1:], beta_0[1:]) # 5.26
multiple_r_squared(x, daily_minutes_good, beta_0) # 0.680
```

Conforme aumentamos `alpha`, o benefício do ajuste piora, mas o tamanho de `beta` diminui:

```
beta_0_01 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.01)
# [30.6, 0.97, -1.86, 0.89]
dot(beta_0_01[1:], beta_0_01[1:])  # 5.19
multiple_r_squared(x, daily_minutes_good, beta_0_01)  # 0.680

beta_0_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.1)
# [30.8, 0.95, -1.84, 0.54]
dot(beta_0_1[1:], beta_0_1[1:])  # 4.60
multiple_r_squared(x, daily_minutes_good, beta_0_1)  # 0.680

beta_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=1)
# [30.7, 0.90, -1.69, 0.085]
dot(beta_1[1:], beta_1[1:])  # 3.69
multiple_r_squared(x, daily_minutes_good, beta_1)  # 0.676

beta_10 = estimate_beta_ridge(x, daily_minutes_good, alpha=10)
# [28.3, 0.72, -0.91, -0.017]
```

```
dot(beta_10[1:], beta_10[1:])   # 1.36
multiple_r_squared(x, daily_minutes_good, beta_10)   # 0.573
```

Em particular, o coeficiente em "PhD" some conforme aumentamos a penalidade, o que está de acordo com nosso resultado anterior que não foi significantemente diferente de zero.

Geralmente, você quereria reescalar seus dados antes de usar essa abordagem. Afinal, se você muda anos de experiência para séculos de experiência, seu coeficiente de mínimo quadrado aumentar por um fator de 100 e inesperadamente será penalizado muito mais, mesmo sendo o mesmo modelo.

Outra abordagem é a regressão laço (*lasso*), que usa a penalidade:

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Enquanto a penalidade de cumeeira diminui os coeficientes no geral, a penalidade laço tende a forçar os coeficientes a serem zero, o que a torna boa para aprender modelos esparsos. Infelizmente, não é agradável para o gradiente descendente, o que significa que nós não conseguiremos resolvê-la do zero.

Para Mais Esclarecimentos

- A regressão possui uma teoria rica e expansiva. Este é outro assunto que você deveria considerar ler um livro didático ou, pelo menos, artigos do Wikipédia.
- scikit-learn possui um módulo linear_model (*http://bit.ly/1ycPg63*) que fornece um modelo `LinearRegression` similar ao nosso, bem como uma regressão `Ridge`, regressão `Lasso` e outros tipos de regularização.
- Statsmodel (*http://statsmodels.sourceforge.net*) é outro tipo de módulo Python que contém (entre outras coisas) modelos de regressão linear.

CAPÍTULO 16

Regressão Logística

Muitas pessoas dizem que há uma linha tênue entre a genialidade e a loucura. Não acho que exista uma linha tênue, eu acho que há um abismo.
—Bill Bailey

No Capítulo 1, nós demos uma pequena olhada no problema de tentar prever quais usuários da DataSciencester pagavam por contas premium. Revisitaremos esse problema neste capítulo.

O Problema

Nós temos um conjunto de dados anônimos de aproximadamente 200 usuários, contendo o salário de cada usuário, seus anos de experiência como cientistas de dados e se pagam por uma conta premium (Figura 16-1). Como é comum com variáveis categóricas, nós representamos as variáveis dependentes como 0 (sem conta premium) ou 1 (conta premium).

Como de costume, nossos dados estão em uma matriz na qual cada fileira é uma lista `[experience, salary, paid_account]`. Vamos transformá-la no formato do qual precisamos:

```
x = [[1] + row[:2] for row in data]  # cada elemento é [1, experience, salary]
y = [row[2] for row in data]         # cada elemento é paid_account
```

Uma primeira tentativa óbvia é usar a regressão linear e encontrar o melhor modelo:

$$\text{conta paga} = \beta_0 + \beta_1 \text{ experiência} + \beta_2 \text{ salário} + \varepsilon$$

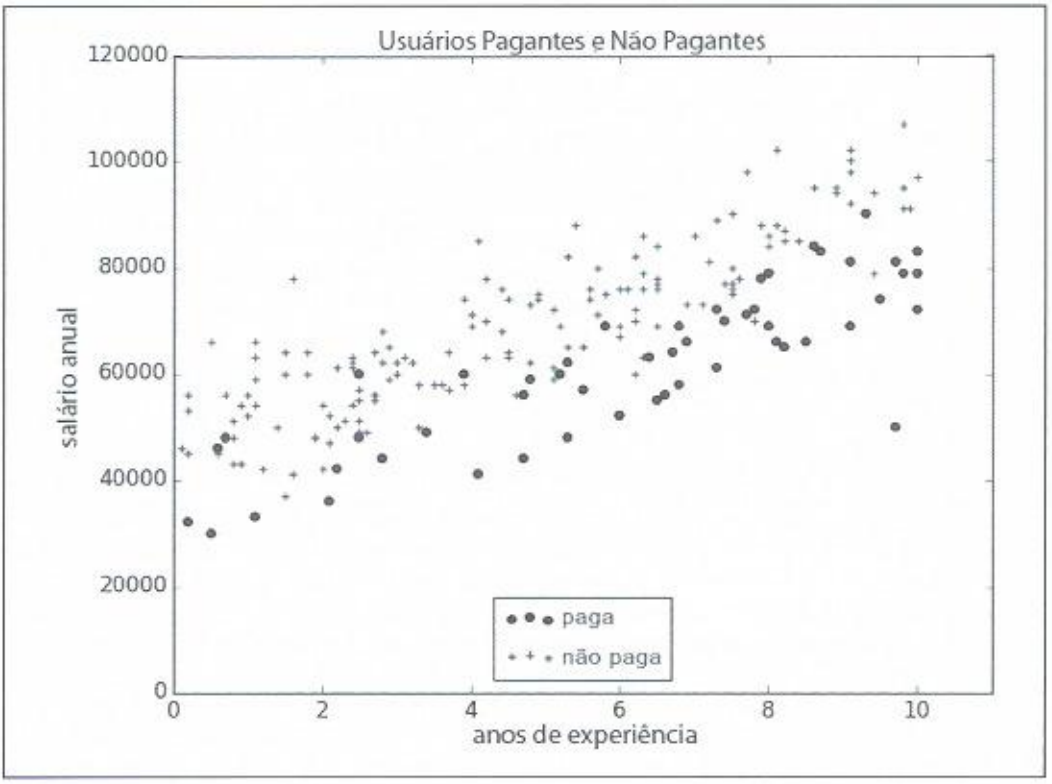

Figura 16-1. Usuários pagantes e não pagantes

E certamente não há nada que nos impeça de modelar o problema dessa forma. Os resultados são exibidos na Figura 16-2.

```
rescaled_x = rescale(x)
beta = estimate_beta(rescaled_x, y)  # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]

plt.scatter(predictions, y)
plt.xlabel("prevista")
plt.ylabel("realizada")
plt.show()
```

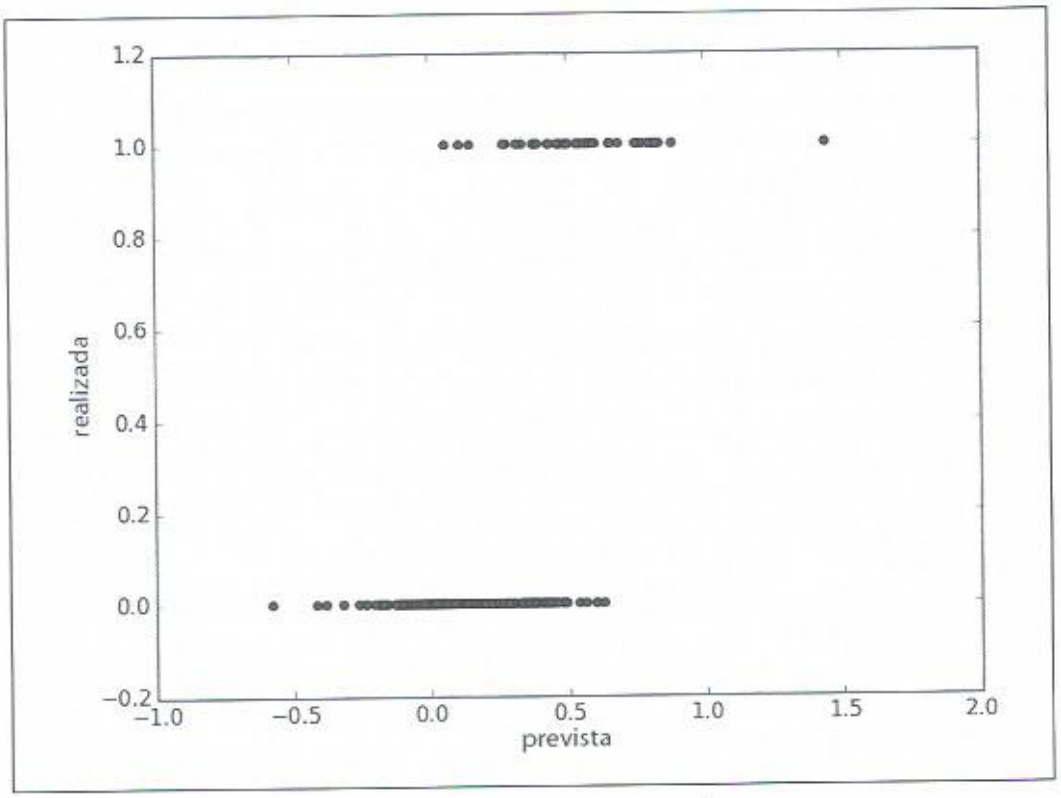

Figura 16-2. Usando regressão linear para prever contas premium

Mas essa abordagem leva a alguns problemas imediatos:

- Nós gostaríamos que as saídas da nossa previsão fossem 0 ou 1, para indicar a membresia da classe. Estaria tudo bem se eles estivessem entre 0 e 1, uma vez que nós podemos interpretar essas probabilidades — uma saída de 0,25 poderia significar 25% de chance de ser um sócio pagante. Mas saídas do modelo linear podem ser grandes números positivos ou até mesmo números negativos, fazendo com que a interpretação não seja clara. De fato, aqui muitas das nossas previsões foram negativas.
- O modelo de regressão linear presumiu que os erros não eram correlacionados com as colunas de x. Mas aqui, o coeficiente de regressão para `experience` é 0,43, indicando que mais experiência leva a maiores probabilidades de uma conta premium. Isso significa que nosso modelo apresenta valores muito grandes para pessoas com muita experiência. Mas nós sabemos que os valores reais devem ser, no máximo 1, o que significa que saídas muito grandes correspondem a valores negativos muito altos nos termos de erros. Sendo esse o caso, nossa estimativa para beta é polarizada.

O que nós gostaríamos é que grandes valores positivos de `dot(x_i, beta)` correspondessem às probabilidades próximas a 1 e que grandes valores negativos correspondessem às probabilidades próximas a 0. Nós podemos realizar isso aplicando outra função no resultado.

A Função Logística

No caso da regressão logística, usamos a *função logística*, exibida na Figura 16-3:

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

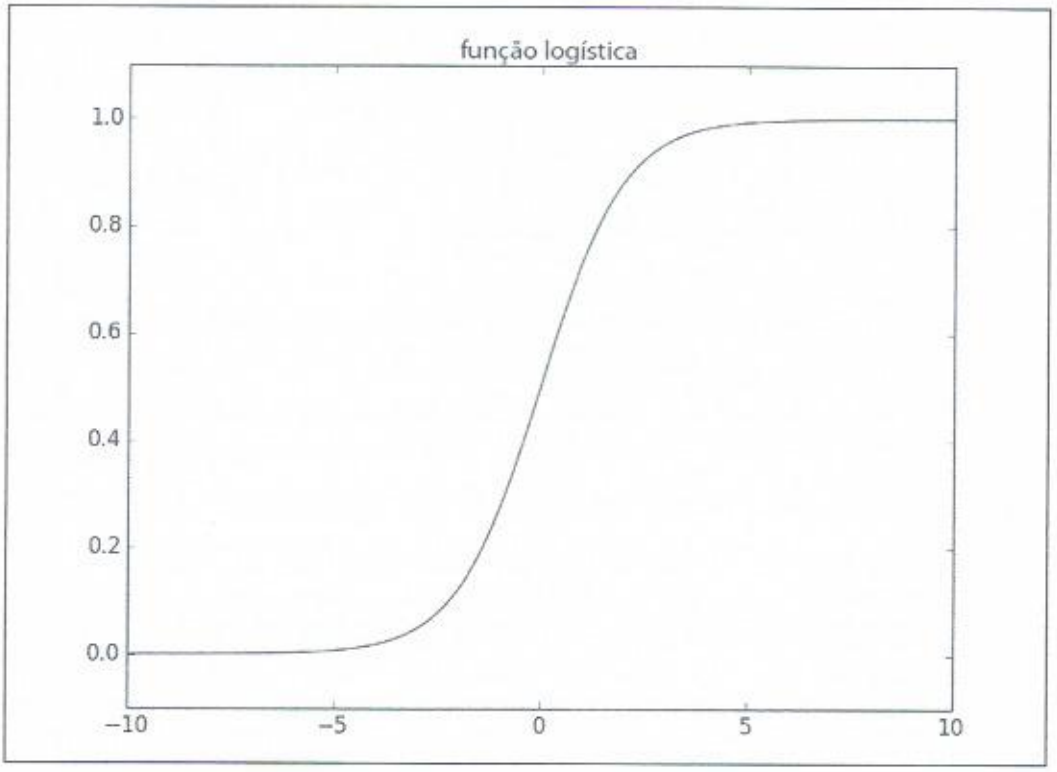

Figura 16-3. A função logística

Conforme sua entrada fica grande e positiva, ela se aproxima cada vez mais de 1. Conforme sua entrada fica grande e negativa, se aproxima mais de 0. Além disso, ela tem a propriedade conveniente da sua derivada ser dada por:

```
def logistic_prime(x):
    return logistic(x) * (1 - logistic(x))
```

a qual nós utilizaremos em um instante. Nós usaremos isto para ajustar um modelo:

$$y_i = f(x_i\beta) + \varepsilon_i$$

em que f é a função `logistic`.

Lembre-se que, para a regressão linear, nós ajustamos o modelos minimizando a soma dos erros quadrados, o que acabou escolhendo o β que maximizou a probabilidade dos dados.

Aqui as duas não são equivalentes, então usaremos o gradiente descendente para maximizar a probabilidade diretamente. Isso significa que precisamos calcular a função de probabilidade e seu gradiente.

Dado algum β, nosso modelo diz que cada y_i deveria ser igual a 1 com probabilidade $f(x_i\beta)$ e 0 com probabilidade $1 - f(x_i\beta)$.

O pdf para y_i pode ser escrito como:

$$p(y_i|x_i,\beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1-y_i}$$

se y_i é 0, isso é igual a:

$$1 - f(x_i\beta)$$

e se y_i é 1, é igual a:

$$f(x_i\beta)$$

Acaba sendo mais simples maximizar o *log da probabilidade*:

$$\log L(\beta|x_i, y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log (1 - f(x_i\beta))$$

Porque a função log é monotonamente crescente, qualquer `beta` que maximize o log da probabilidade também maximiza a probabilidade e vice-versa:

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

Se supormos que pontos de dados diferentes são independentes uns dos outros, a probabilidade total é o produto das probabilidades individuais. O que significa que o log total da probabilidade é a soma das probabilidades dos log individuais;

```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

Um pouco de cálculo nos fornece o gradiente:

```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """aqui i é o índice do ponto de dados,
    j é o índice da derivada"""

    return (y_i - logistic(dot(x_i, beta))) * x_i[j]

def logistic_log_gradient_i(x_i, y_i, beta):
    """o gradiente do log da probabilidade
    correspondente ao i-ésimo ponto de dados"""

    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

neste ponto nós temos todos os pedaços que precisamos.

Aplicando o Modelo

Queremos dividir nossos dados em conjunto de treinamento e conjunto de teste:

```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# queremos maximizar o log da probabilidade em dados de treinamento
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# escolhemos um ponto de partida aleatório
beta_0 = [random.random() for _ in range(3)]

# e maximizamos usando o gradiente descendente
beta_hat = maximize_batch(fn, gradient_fn, beta_0)
```

Como alternativa, você poderia usar o gradiente descendente estocástico:

```
beta_hat = maximize_stochastic(logistic_log_likelihood_i,
                               logistic_log_gradient_i,
                               x_train, y_train, beta_0)
```

De qualquer forma, encontramos aproximadamente:

```
beta_hat = [-1.90, 4.05, -3.87]
```

Esses são os coeficientes para os dados `rescaled` (redimensionados), mas podemos transformá-los de volta aos dados originais:

```
beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

Infelizmente, eles não são tão fáceis de interpretar como os coeficientes de regressão linear. O restante sendo igual, um ano a mais de experiência acrescenta 1,42 à entrada de `logistic`. O restante sendo igual, 10.000 a mais no salário diminui 2,49 da entrada de `logistic`.

No entanto, o impacto na saída depende de outras entradas também. Se `dot(beta, x_i)` já é grande (correspondente a uma probabilidade próxima de 1), aumentá-lo mesmo que muito não pode afetar muito a probabilidade. Se for próxima de 0, aumentá-lo um pouco pode aumentar bastante a probabilidade.

O que podemos dizer é que — todo o restante sendo igual — as pessoas com mais experiência têm mais probabilidade de pagar pela assinatura. E também, — todo o restante sendo igual — as pessoas com os salários mais altos são menos prováveis de pagar por assinaturas. (Estava um tanto aparente quando montamos o gráfico dos dados.)

O Benefício do Ajuste

Nós ainda não utilizamos os dados de teste. Vamos ver o que acontece se fizermos a previsão de *conta paga* quando a probabilidade exceder 0,5:

```
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))

    if y_i == 1 and predict >= 0.5:  # PV: paga e previmos paga
        true_positives += 1
    elif y_i == 1:                   # FN: paga e previmos não pagantes
        false_negatives += 1
    elif predict >= 0.5:             # VF: não paga e previmos pagantes
        false_positives += 1
    else:                            # VP: não paga e previmos não paga
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

Isso dá uma acurácia de 93% ("quando previmos *conta paga estamos certos* 93% do tempo") e uma sensibilidade de 82% ("quando um usuário pagou uma conta e previmos *conta paga* 82% do tempo"), em que ambos são números bem respeitáveis.

Nós também podemos assinalar as previsões versus as reais (Figura 16-4), que também mostra que o modelo funciona bem:

```
predictions = [logistic(dot(beta_hat, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test)
plt.xlabel("probabilidade prevista")
plt.ylabel("resultado real")
plt.title("Regressão Logística Prevista vs. Real")
plt.show()
```

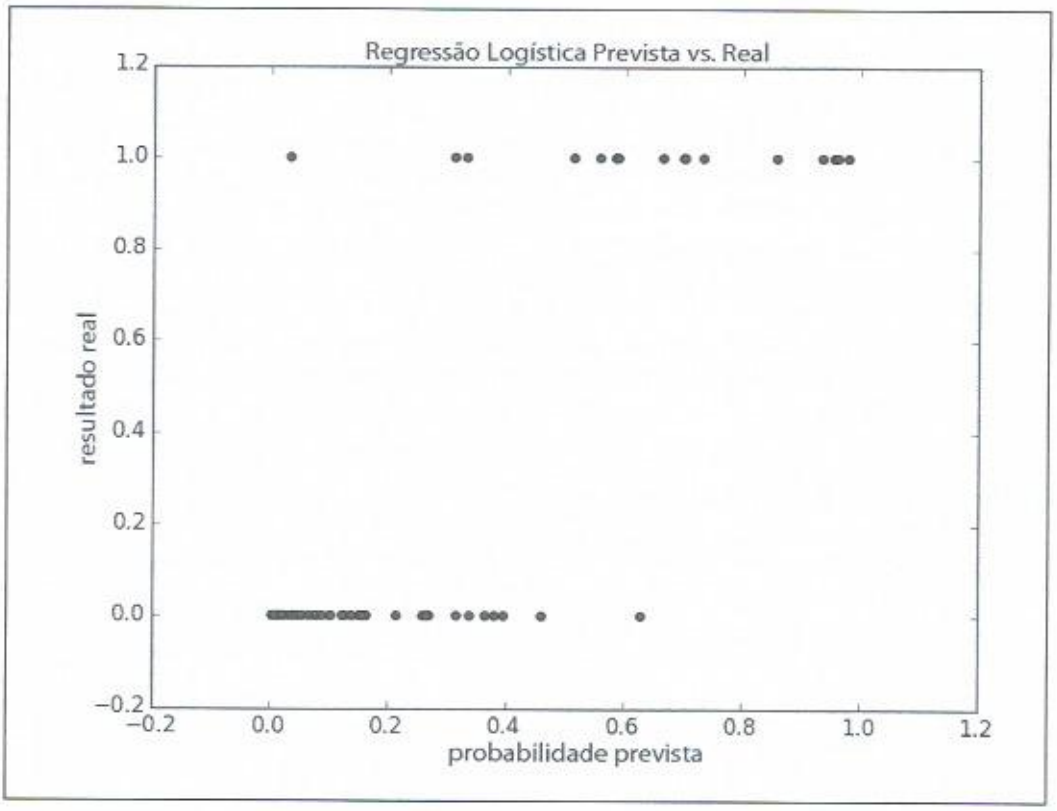

Figura 16-4. Regressão logística prevista versus real.

Máquina de Vetor de Suporte

Esse conjunto de pontos em que `dot(beta_hat, x_i)` é igual a 0 é o limite entre nossas classes. Nós podemos assinalá-lo para ver exatamente o que nosso modelo está fazendo (Figura 16-5).

Esse limite é um *hiperplano* que divide o espaço de parâmetro entre duas partes de espaço correspondentes a *prever pagos* e *prever não pagos*. Nós encontramos isso como um efeito colateral de encontrar o modelo logístico mais provável.

Uma abordagem alternativa para a classificação é apenas procurar o hiperplano que "melhor" separe as classes nos dados de treinamento. Essa é a ideia por trás da *máquina de vetor de suporte*, que encontra o hiperplano que maximiza a distância para o ponto mais próximo em cada classe (Figura 16-6).

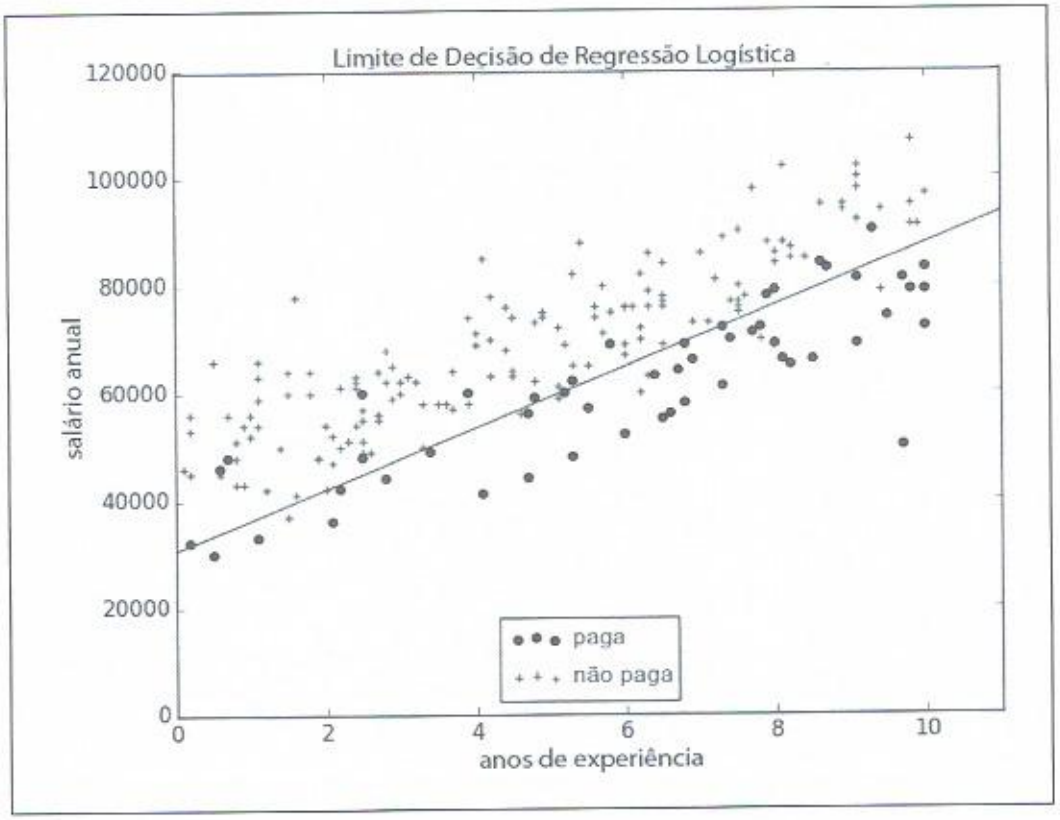

Figura 16-5. Usuários pagantes e não pagantes com limite de decisão

Encontrar tal hiperplano é um problema de otimização que envolve técnicas que são avançadas demais para nós. Um problema diferente é que um hiperplano de separação pode nem mesmo existir. Em nosso conjunto de dados "quem paga?" simplesmente não há linha que separe perfeitamente os usuários pagantes dos não pagantes.

Nós podemos,às vezes, contornar essa situação transformando os dados em um espaço dimensional superior. Por exemplo, considere o simples conjunto unidimensional de dados exibido na Figura 16-7.

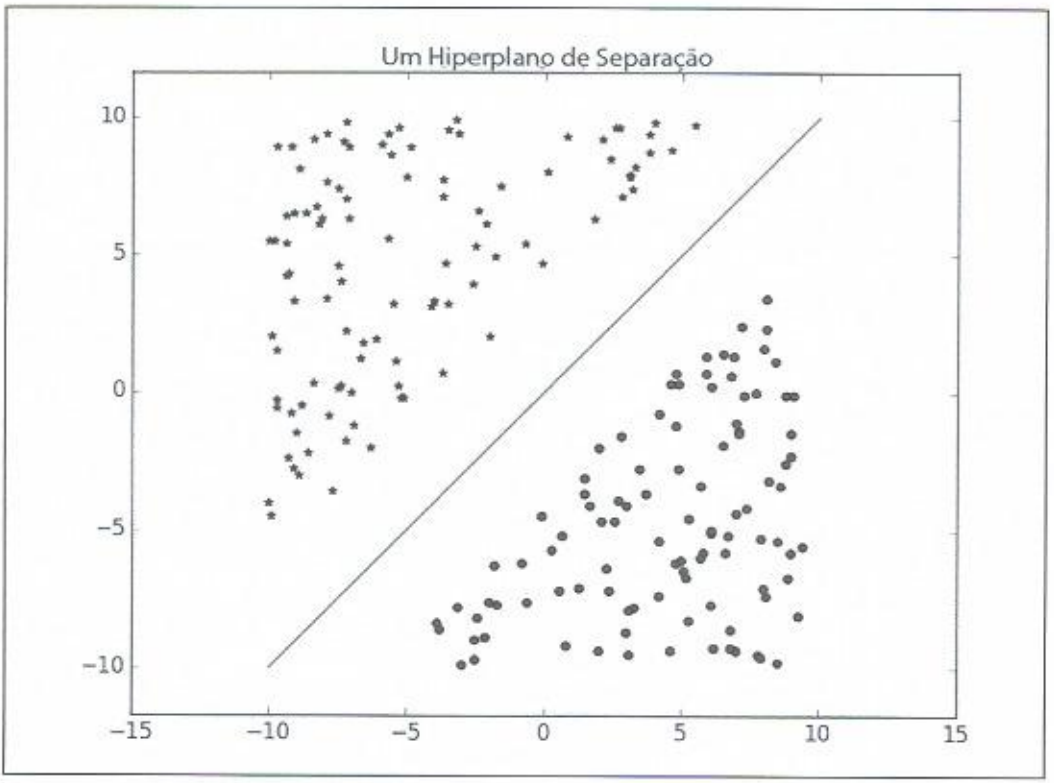

Figura 16-6. Um hiperplano de separação

Está claro que não há hiperplano que separe exemplos positivos dos negativos. Entretanto, olhe o que acontece quando mapeamos esse conjunto de dados em duas dimensões diferentes enviando o ponto `x` para `(x, x**2)`. De repente, é possível encontrar um hiperplano que divide os dados (Figura 16-8).

Isso é geralmente chamado de *truque do kernel* porque, em vez de mapear os pontos num espaço dimensional maior (o que pode ser caro se houver muitos pontos e o mapeamento for complicado), nós usamos uma função "kernel" para computar produtos escalares no espaço dimensional maior e usá-los para encontrar o hiperplano.

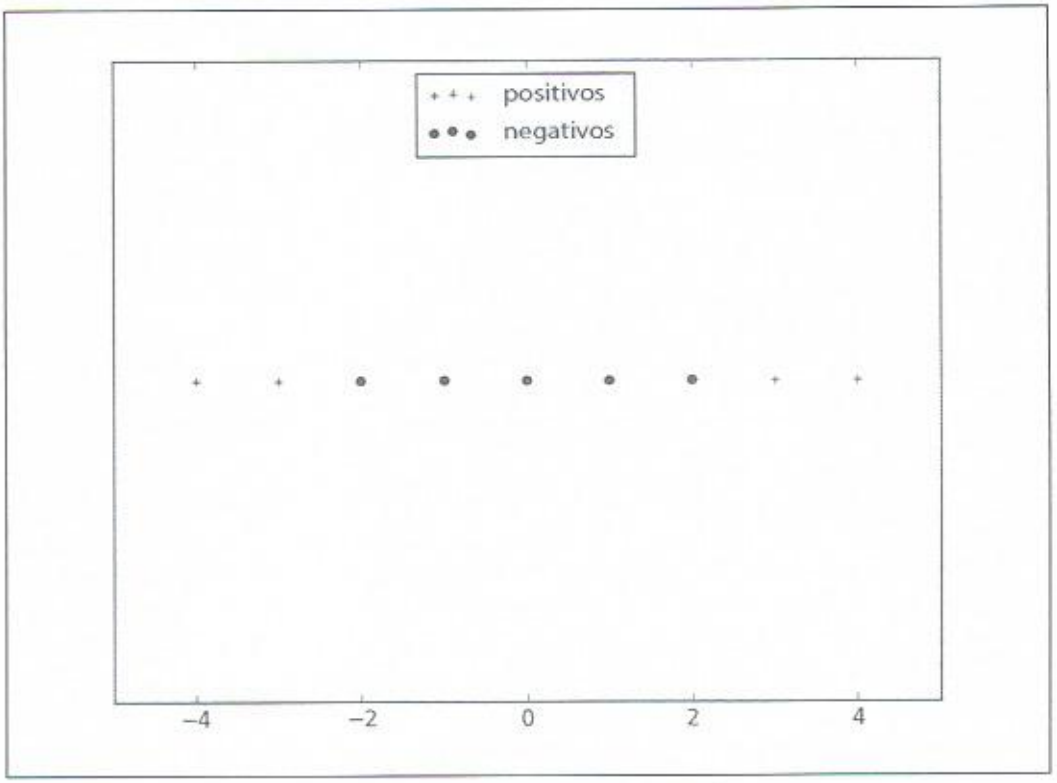

Figura 16-7. Um conjunto unidimensional de dados inseparável

É difícil (e, provavelmente, uma má ideia) usar máquinas de vetor de suporte sem depender de um software especializado em otimização escrito por pessoas com o conhecimento apropriado, então vamos deixar nosso tratamento aqui.

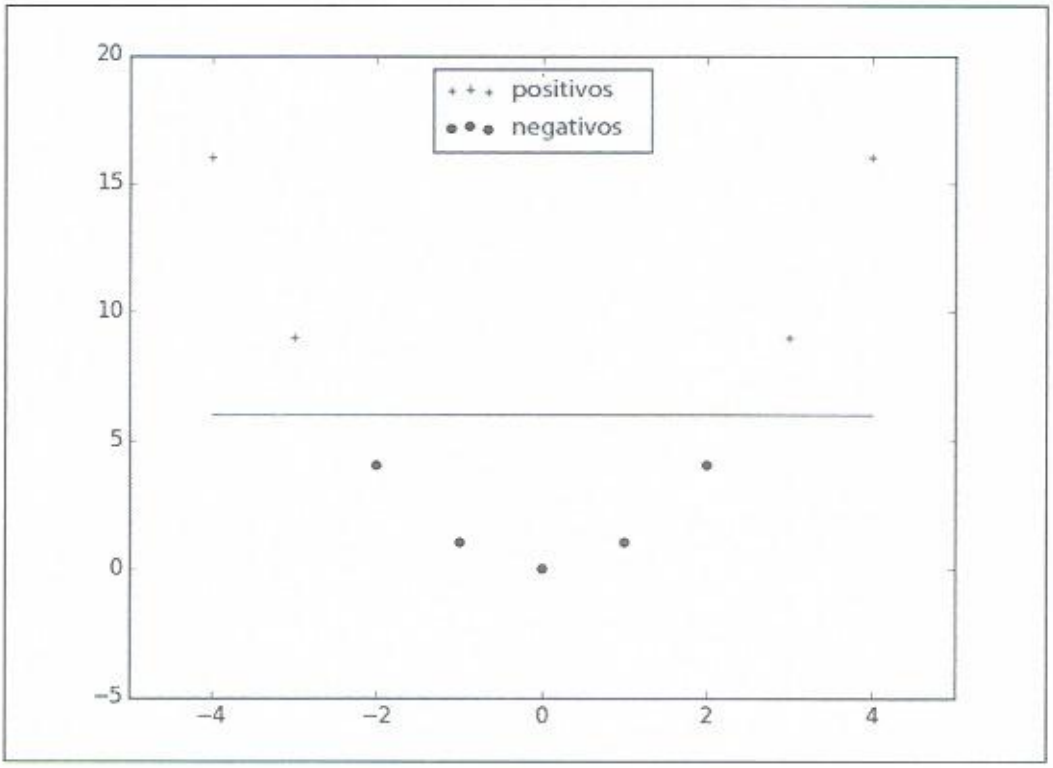

Figura 16-8. O conjunto de dados se torna separável em dimensões maiores

Para Mais Esclarecimentos

- scikit-learn possui modelos para Regressão Logística (*http://bit.ly/1xkbywA*) e Máquinas de Vetor de Suporte (*http://bit.ly/1xkbBZj*).
- libsvm (*http://bit.ly/1xkbA7t*) é a implementação de máquina de vetor de suporte que scikit-learn usa. Em seu website há uma variedade de documentação sobre máquinas de vetor de suporte.

CAPÍTULO 17

Árvores de Decisão

Uma árvore é um mistério incompreensível.
—Jim Wooddring

O vice-presidente de Talentos da DataSciencester entrevistou um número de candidatos para emprego do site, com níveis de sucesso variados. Ele coletou um conjunto de dados com vários atributos (qualitativos) de cada candidato, bem como se o candidato de saiu bem ou mal na entrevista. Você poderia usar esses dados para construir um modelo identificando quais candidatos farão boas entrevistas, para que ele não precise perder tempo fazendo entrevistas?

Isso parece ser perfeito para uma *árvore de decisão*, outra ferramente de modelagem de previsão no kit de um cientista de dados.

O Que É uma Árvore de Decisão?

Uma árvore de decisão usa uma estrutura de árvore para representar um número de possíveis *caminhos de decisão* e um resultado para cada caminho.

Se você já jogou Vinte Perguntas, já está familiarizado com árvores de decisão. Por exemplo:

- "Estou pensando em um animal."
- "Ele possui mais de cinco pernas?"
- "Não."
- "É delicioso?"
- "Não."

- "Ele aparece na parte de trás da moeda de cinco centavos australiana?"
- "Sim."
- "É um equidna?"
- "Sim!"

Isso corresponde ao caminho:

"Não mais do que 5 pernas" → "Não delicioso" → "Na moeda de 5 centavos" → "Equidna!"

em uma idiossincrática (e não muito abrangente) árvore de decisão "adivinhe o animal" (Figura 17-1).

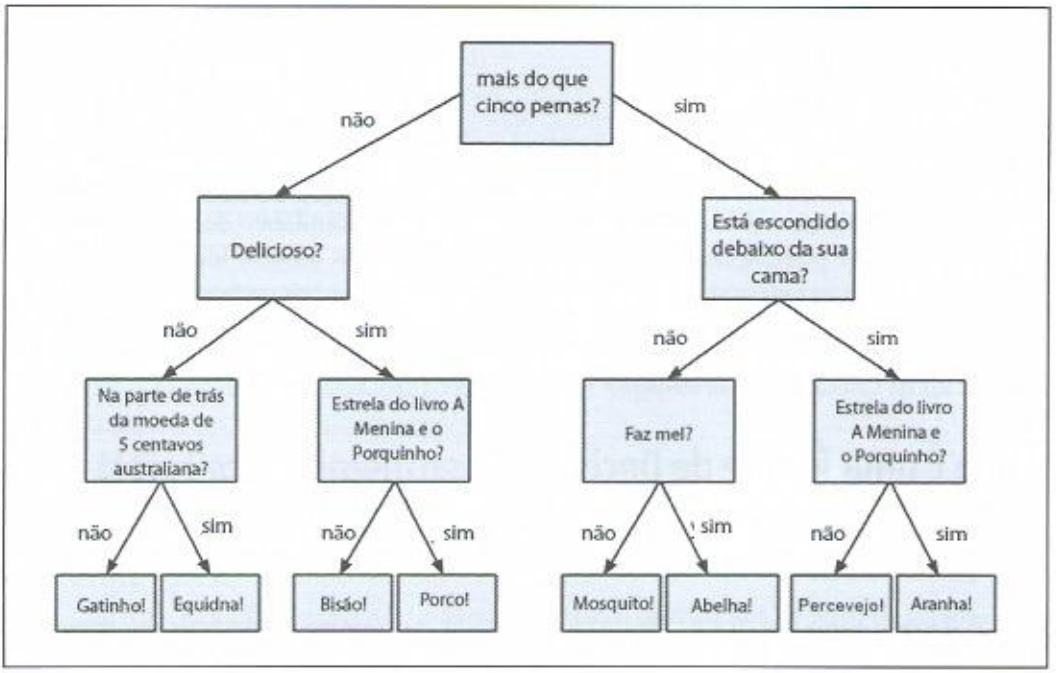

Figura 17-1. Uma árvore de decisão "adivinhe o animal"

As árvores de decisão possuem muitas recomendações. Elas são muito fáceis de entender e interpretar, e o processo por onde chegam numa previsão é completamente transparante. Diferente de outros modelos que vimos até agora, as árvores de decisão podem lidar facilmente com uma mistura de atributos numéricos (exemplo: número de pernas) e categóricos (exemplo: delicioso/não delicioso) e podem até classificar os dados para os atributos que estão faltando.

Ao mesmo tempo, encontrar a árvore de decisão perfeita para um conjunto de dados em treinamento é computacionalmente um problema muito difícil. (Contornaremos isso tentando construir uma árvore boa o bastante em vez de uma perfeita, apesar de que, para uma boa parte

de conjuntos de dados isso ainda pode ser muito trabalhoso.) Mais importante, é muito fácil (e muito ruim) construir árvores de decisão que são *sobreajustadas* aos dados em treinamento e que não generalizem bem para dados desconhecidos. Nós veremos formas de lidar com isso.

A maioria das pessoas dividem árvores de decisão em *árvores de classificação* (que produzem saídas categóricas) e *árvores de regressão* (que produzem saídas numéricas). Neste capítulo, focaremos em árvores de classificação e trabalharemos com o algoritmo ID3 para aprender uma árvore de decisão a partir de um conjunto de dados rotulados, o que talvez nos ajude a entender como árvores de decisão realmente funcionam. Para simplificar as coisas, nos restringiremos a problemas com saídas binárias como "eu deveria contratar esse candidato?" ou "eu deveria exibir o anúncio A ou o B para visitantes do site?" ou "comer essa comida que encontrei na geladeira do escritório me fará mal?"

Entropia

Pra construir uma árvore de decisão, precisaremos decidir quais perguntas fazer e em qual ordem. Em cada etapa de uma árvore há algumas possibilidades que eliminamos e outras que não. Após ter a informação de que um animal não possui mais do que cinco pernas, eliminamos a possibilidade de ele ser um gafanhoto. Não eliminamos a possibilidade de ser um pato. Cada pergunta possível separa as possibilidades restantes de acordo com as respostas.

Nós gostaríamos de escolher perguntas cujas respostas nos dessem muita informação sobre o que nossa árvore deveria prever. Se houver uma simples pergunta sim/não para cada respostas "sim" sempre correspondente a saídas `True` (Verdadeiros) e respostas "não" a saídas `False` (Falsos), essa seria uma pergunta perfeita para fazer. Contrariamente, uma pergunta sim/não para a qual nenhuma resposta lhe dá muita informação sobre o que previsão deveria ser não é uma boa escolha.

Nós chegamos nessa noção de "quanta informação" com *entropia*. Você já deve ter escutado isso com o significado de desordem. Nós usamos para representar a incerteza associada com os dados.

Imagine que temos um conjunto S de dados, no qual cada membro é rotulado como pertencente a uma classe dentre o finito número de classes $C_1, \ldots, C_n$. Se todos os pontos de dados pertencem a uma única classe, então não há incerteza real, o que significa que deve haver baixa entropia. Se os pontos de dados estão separados de forma igual nas classes, há muita incerteza e deve haver alta entropia.

Em termos matemáticos, se p_i é a proporção de dados definidos como classes c_i, nós definimos a entropia como:

$$H(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$

com a convenção (padrão) de que $0 \log 0 = 0$.

Sem muita preocupação com os detalhes, cada termo $-p_i \log_2 p_i$ é não negativo e próximo de zero precisamente quando p_i ou é próximo de zero ou de 1 (Figura 17-2).

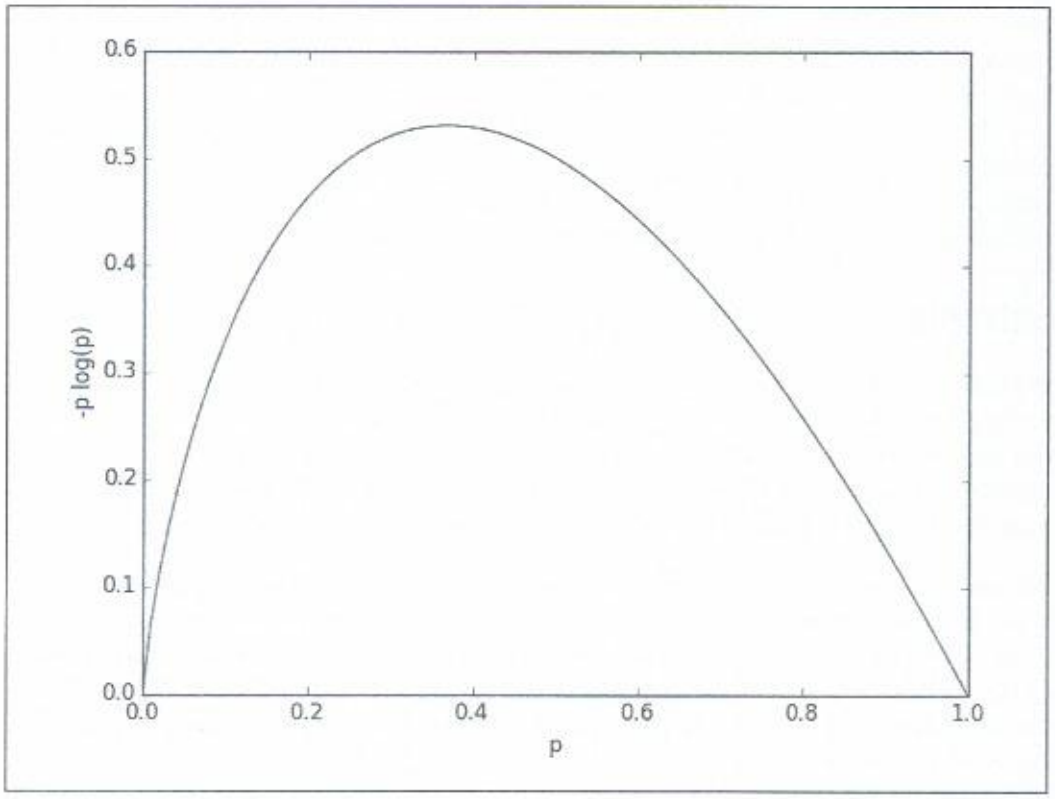

Figura 17-2. Um gráfico de –p log p

Isso significa que a entropia será pequena quando cada p_i é próximo de 0 ou 1 (por exemplo: quando a maioria dos dados está em uma única classe) e será maior quando muitos dos p_i não estiverem próximos de 0 (por exemplo, quando os dados estão espalhados por múltiplas classes). Esse é exatamente o comportamento esperado.

É muito fácil jogar tudo isso em uma função:

```
def entropy(class_probabilities):
    """dada uma lista de probabilidades de classe, compute a entropia"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p)                          # ignora probabilidades zero
```

Nossos dados consistirão de pares (`input, label`), o que significa que precisaremos computar sozinhos as probabilidades de classe. Observe que não nos importa qual rótulo é associada com qual probabilidade, apenas quais são as probabilidades:

```
def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

A Entropia de uma Partição

O que fizemos até agora foi computar a entropia (pense "incerteza") de um conjunto de dados rotulados. Agora, cada etapa da árvore de decisão envolve fazer uma pergunta cuja resposta particiona os dados em um ou (esperamos) mais subconjuntos. Por exemplo, nossa pergunta "tem mais de cinco pernas?" divide animais em aqueles que têm mais de cinco pernas (por exemplo, aranhas) e os que não (por exemplo, equidnas).

Da mesma forma, nós gostaríamos de ter alguma noção de entropia que resulte em particionar um conjunto de dados em uma certa forma. Nós queremos uma divisão que tenha baixa entropia se dividir os dados em subconjuntos que tenham baixa entropia (por exemplo, são altamente certos) e alta entropia se possuir subconjuntos com (grandes e com) alta entropia (por exemplo, são altamente incertos).

Por exemplo, minha pergunta "moeda de cinco centavos australiana" foi muito boba (e bem sortuda!), à medida que dividiu os animais restantes até aquele momento em S_1 = {equidna} e S_2 = {o restante}, em que S_2 é grande e possui alta entropia. (S_1 não tem entropia mas representa uma pequena fração das "classes" restantes.)

Matematicamente, se dividirmos nossos dados S em subconjuntos $S_1, \ldots, S_m$ contendo proporções de dados $q1, \ldots, q_m$, então nós computamos a entropia da partição como uma soma ponderada:

$$H = q_1 H(S_1) + \ldots + q_m H(S_m)$$

que podemos implementar como:

```
def partition_entropy(subsets):
    """encontre a entropia desta divisão de dados em subconjuntos
    subconjunto é uma lista de listas de dados rotulados"""

    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )
```

Um problema com essa abordagem é que particionar por um atributo (característica) com muitos valores diferentes resultará em um entropia muito baixa devido ao sobreajuste. Por exemplo, imagine que você trabalha para um banco e está tentando construir uma árvore de decisão para prever quais clientes provavelmente serão inadimplentes com o financiamento usando alguns dados históricos como seu conjunto de treinamento. Imagine ainda que os dados contêm o número do CPF de cada cliente. Dividir em SSN produzirá subconjuntos de uma pessoa, em que cada uma delas necessariamente possui zero entropia. Mas um modelo que depende de SSN *certamente* não generaliza além do conjunto de treinamento. Por isso, você deveria evitar (ou agrupar, se apropriado) atributos com muitos valores possíveis ao criar árvores de decisão.

Criando uma Árvore de Decisão

A vice-presidente forneceu dados dos entrevistados, que consistem de (por sua especificação) pares (`input, label`) em que cada `input` é um `dict` de características de candidatos e cada rótulo é `True` (o candidato fez boa entrevista) ou `False` (o candidato fez entrevista ruim). Em específico, você possui o nível de cada candidato, sua linguagem favorita, se é ativo no Twitter e se possui PhD:

```
inputs = [
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'},    False),
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'},   False),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'},      True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'},   True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'},     False),
    ({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'},         True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'},  False),
    ({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'},  True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'}, True),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'},     True),
    ({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, False)
]
```

Nossa árvore consistirá de *nós de decisão* (que fazem uma pergunta e direcionam de forma diferente dependendo da resposta) e *nós folha* (que nos dão uma previsão). Nós a construiremos usando um algoritmo *ID3* relativamente simples que opera da seguinte forma. Digamos que nos deram alguns dados rotulados e uma lista de características que deveríamos considerar para ramificar.

- Se os dados possuem a mesmo rótulo, crie um nó folha que prevê esse rótulo e então pare.

- Se a lista de características está vazia (por exemplo: não existem mais perguntas possíveis), crie um nó folha que prevê o rótulo mais comum e pare.
- Caso contrário, tente particionar os dados por todas as características.
- Escolha a divisão com a entropia de partição mais baixa.
- Adicione um nó de decisão baseado na característica escolhida.
- Retorne a cada subconjunto particionado usando as características remanescentes.

Isso é conhecido como um algoritmo "ganancioso" porque, a cada passo, ele escolhe a opção imediatamente melhor. Dado um conjunto de dados, pode existir uma árvore melhor com um primeiro movimento pior de ver. Caso exista, esse algoritmo não irá encontrá-la. Contudo, é relativamente mais fácil de entender e implementar, o que o torna muito bom para começar a explorar árvores de decisão.

Vamos percorrer manualmente esses passos no conjunto de dados dos entrevistados. O conjunto de dados possui rótulos `True` e `False`, e nós temos quatro características pelas quais podemos dividi-los. Então, nosso primeiro passo será encontrar a partição com a menor entropia. Começaremos escrevendo uma função que faz a divisão:

```
def partition_by(inputs, attribute):
    """cada entrada é um par (attribute_dict, label).
    retorna uma dict: attribute_value -> inputs"""
    groups = defaultdict(list)
    for input in inputs:
        key = input[0][attribute]   # pega o valor do atributo especificado
        groups[key].append(input)   # então adiciona essa entrada à lista correta
    return groups
```

e uma que usa isso para computar a entropia:

```
def partition_entropy_by(inputs, attribute):
    """computa a entropia correspondente à partição dada"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

Então só precisamos encontrar a partição com entropia mínima para todo o conjunto de dados:

```
for key in ['level','lang','tweets','phd']:
    print key, partition_entropy_by(inputs, key)

# level 0.693536138896
# lang 0.860131712855
# tweets 0.788450457308
# phd 0.892158928262
```

A menor entropia vem da divisão baseada em `level`, então precisamos fazer uma sub-árvore para cada valor `level` possível. Cada candidato `Mid` é rotulado com `True`, o que significa que a sub-árvore `Mid` é simplesmente um nó folha que prevê `True`. Para candidatos `Senior`, temos uma mistura de `True`s e `False`s, logo precisamos dividir novamente:

```
senior_inputs = [(input, label)
                 for input, label in inputs if input["level"] == "Senior"]

for key in ['lang', 'tweets', 'phd']:
    print key, partition_entropy_by(senior_inputs, key)

# lang 0.4
# tweets 0.0
# phd 0.950977500433
```

Isso nos mostra que nossa próxima divisão deveria ser com base em `tweets`, que resulta em uma partição de entropia zero. Para os candidatos `Senior`, tweets "sim" sempre resultam em `True` enquanto tweets "não" sempre resultam em `False`.

Finalmente, se fizermos a mesma coisa para os candidatos `Junior`, dividiremos em `phd`, após o que descobrimos que não PhD sempre resulta em `True` e PhD sempre resulta em `False`.

A Figura 17-3 mostra a árvore de decisão completa.

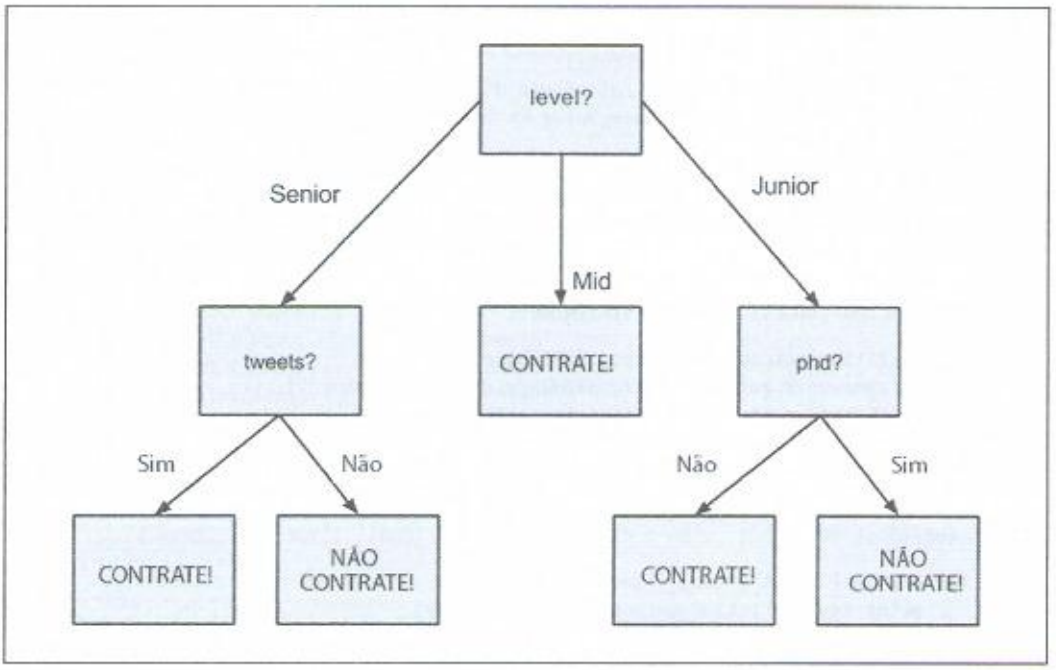

Figura 17-3. A árvore de decisão para contratação

Juntando Tudo

Agora que vimos como o algoritmo funciona, gostaríamos de implementá-lo de forma mais geral. Isso significa que precisamos decidir como queremos representar as árvores. Usaremos a representação mais leve possível. Nós definimos as *árvores* como:

- `True`
- `False`
- uma tupla (`attribute, subtree_dict`)

Aqui `True` representa um nó folha que retorna `True` para qualquer entrada, `False` representa um nó folha que retorna `False` para qualquer entrada, e uma tupla representa um nó de decisão que, para qualquer entrada, encontra seu valor `attribute` e classifica a entrada usando a sub-árvore correspondente.

Com essa representação, nossa árvore de contratação se pareceria com isso:

```
('level',
 {'Junior': ('phd', {'no': True, 'yes': False}),
  'Mid': True,
  'Senior': ('tweets', {'no': False, 'yes': True})})
```

Ainda há a questão do que fazer se encontrarmos um valor de característica inesperada (ou faltante). O que nossa árvore deveria fazer se encontrasse um candidato cujo `level` é "estagiário"? Lidaremos com esse caso acrescentando uma chave `None` que prevê o rótulo mais comum. (Apesar de que isso seria uma péssima ideia se `None` fosse na verdade um valor que aparecesse nos dados.)

Dada tal representação, podemos classificar uma entrada com:

```
def classify(tree, input):
    """classifica a entrada usando a árvore de decisão fornecida"""

    # se for um nó folha, retorna seu valor
    if tree in [True, False]:
        return tree

    # senão, esta árvore consiste de uma característica para dividir
    # e um dicionário cujas chaves são valores daquela característica
    # e cujos valores são sub-árvores para considerar depois
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)    # None se estiver faltando característica

    if subtree_key not in subtree_dict:   # se não há sub-árvore para chave,
        subtree_key = None                # usaremos a sub-árvore None

    subtree = subtree_dict[subtree_key]   # escolha a sub-árvore apropriada
    return classify(subtree, input)       # e use para classificar a entrada
```

E tudo o que restou é construir a representação da árvore a partir dos nossos dados em treinamento:

```
def build_tree_id3(inputs, split_candidates=None):

    # se este é nosso primeiro passo,
    # todas as chaves da primeira entrada são candidatos divididos
```

```
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # conta Trues e Falses nas entradas
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0: return False     # nenhum True? Retorne uma folha "False"
    if num_falses == 0: return True     # nenhum False? Retorne uma folha "True"

    if not split_candidates:            # se não houver mais candidatos a dividir
        return num_trues >= num_falses  # retorne a folha majoritária

    # senão, divida com base na melhor caracteristica
    best_attribute = min(split_candidates,
                         key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # recursivamente constrói as sub-árvores
    subtrees = { attribute_value : build_tree_id3(subset, new_candidates)
                 for attribute_value, subset in partitions.iteritems() }

    subtrees[None] = num_trues > num_falses      # caso padrão

    return (best_attribute, subtrees)
```

Na árvore que construímos, cada folha consistia inteiramente de entradas `True` ou inteiramente de entradas `False`. Isso significa que a árvore prevê perfeitamente em um conjunto de dados em treinamento. Mas nós também podemos aplicar isso a novos dados que não estavam no conjunto de treinamento:

```
tree = build_tree_id3(inputs)

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "no"} )        # True

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "yes"} )       # False
```

E também em dados com valores faltando ou inesperados:

```
classify(tree, { "level" : "Intern" } ) # True
classify(tree, { "level" : "Senior" } ) # False
```

Como nosso objetivo era demonstrar *como* construir uma árvore, nós construímos a árvore usado todo o conjunto de dados. Como sempre, se realmente estivéssemos tentando criar um bom modelo para alguma coisa, nós teríamos (coletado mais dados e) dividido os dados em subconjuntos de treinamento/validação/teste.

Florestas Aleatórias

Levando em consideração como árvores de decisão podem se ajustar quase perfeitamente a seus dados em treinamento, não nos surpreende que elas tendem a sobreajustar. Uma forma de evitar isso é a técnica chamada *florestas aleatórias*, na qual podemos construir várias árvores de decisão e deixá-las escolher como classificar entradas:

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

Nosso processo de construção de árvores era determinista, então como conseguimos árvores aleatórias?

Uma parte envolve dados *inicialização* (lembre-se de "Digressão: A Inicialização", na página 183). Em vez de treinar cada árvore em todas as entradas no conjunto de treinamento, nós treinamos cada árvore no resultado de `bootstrap_sample(inputs)`. Uma vez que cada árvore é construída usando dados diferentes, cada árvore será diferente da outra. (Um benefício é que usar dados não-amostrados para testar cada árvore é um método justo, o que significa que você pode continuar usando todos os dados como o conjunto de treinamento se você souber medir rendimento.) Esta técnica é conhecida como *bootstrap aggregating* ou *bagging* (empacotamento).

Uma segunda forma envolve mudar como escolhemos a forma como `best_attribute` divide-se. Em vez de olhar para todos os atributos remanescentes, nós primeiro escolhemos um subconjunto aleatório e o dividimos no que for melhor:

```
# se já há candidatos o bastante, olhe para todos eles
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# senão escolha uma amostra aleatória
else:
    sampled_split_candidates = random.sample(split_candidates,
                                             self.num_split_candidates)

# agora escolha a melhor característica a partir apenas daqueles candidatos
best_attribute = min(sampled_split_candidates,
    key=partial(partition_entropy_by, inputs))

partitions = partition_by(inputs, best_attribute)
```

Esse é um exemplo e uma técnica mais ampla chamada *ensemble learning* na qual combinamos vários *weak learners* (tipicamente modelos de alta polarização, e baixa variância) para produzir um modelo forte global.

Florestas aleatórias são um dos modelos mais populares e versáteis disponíveis.

Para Maiores Esclarecimentos

- scikit-learn possui muitos modelos de Árvores de Decisão (*http://bit.ly/1ycPmuq*). Também possui um módulo *ensemble (http://bit.ly/1ycPom1)* que inclui `RandomForestClassifier` e outros métodos.
- Nós quase não arranhamos a superfície do tópico de árvores de decisão e seus algoritmos. A Wikipédia (*http://bit.ly/1ycPn1j*) é um bom ponto de partida para uma explicação mais ampla.

CAPÍTULO 18

Redes Neurais

Eu gosto de coisas sem sentido; elas acordam as células do cérebro.
—Dr. Seuss

Uma *rede neural artificial* (ou rede neural) é um modelo preditivo motivado pela forma como o cérebro funciona. Pense no cérebro como uma coleção de neurônios conectados. Cada neurônio olha para a saída de outros neurônios que o alimentam, faz um cálculo e então ele dispara (se o cálculo exceder algum limite) ou não (se não exceder).

Redes neurais artificiais consistem de neurônios artificiais, que desenvolvem cálculos similares sobre suas entradas. Redes neurais podem resolver uma variedade de problemas como reconhecimento de caligrafia e detecção facial, e elas são muito usadas em *deep learning* (aprendizado profundo), uma das subáreas mais populares de data science. Entretanto, a maioria das redes neurais são "caixas-pretas" — inspecionar seus detalhes não lhe fornece muito entendimento de *como* elas estão resolvendo um problema. E grandes redes neurais podem ser difíceis de treinar. Para a maioria dos problemas que você encontrará como um cientista de dados, elas provavelmente não são a melhor opção. Algum dia, quando você estiver tentando construir uma inteligência artificial para tornar real a Singularidade, elas podem ser.

Perceptrons

A rede neural mais simples é a *perceptron*, que aproxima um único neurônio com *n* entradas binárias. Ela computa a soma ponderada de suas entradas e "dispara" se essa soma for zero ou maior:

```
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights, bias, x):
    """retorna 1 se a perceptron 'disparar', 0 se não"""
```

```
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

Perceptron é simplesmente a distinção entre espaços separados pelo hiperplano de pontos x, pelo qual:

```
dot(weights,x) + bias == 0
```

Com pesos propriamente escolhidos, perceptrons podem solucionar alguns problemas simples (Figura 18-1). Por exemplo, podemos criar uma *porta AND* (que retorna 1 se ambas entradas forem 1 mas retorna 0 se uma das entradas for 0) com:

```
weights = [2, 2]
bias = -3
```

Se ambas as entradas forem 1, o cálculo (`calculation`) será igual a 2 + 2 – 3 = 1, e a saída será 1. Se apenas uma das entradas for 1, o `cálculo` será igual a 2 + 0 – 3 = –1, e a saída será 0. E se ambas entradas forem 0, o `cálculo` será –3 e a saída será 0.

Similarmente, poderíamos construir uma *porta OR* com:

```
weights = [2, 2]
bias = -1
```

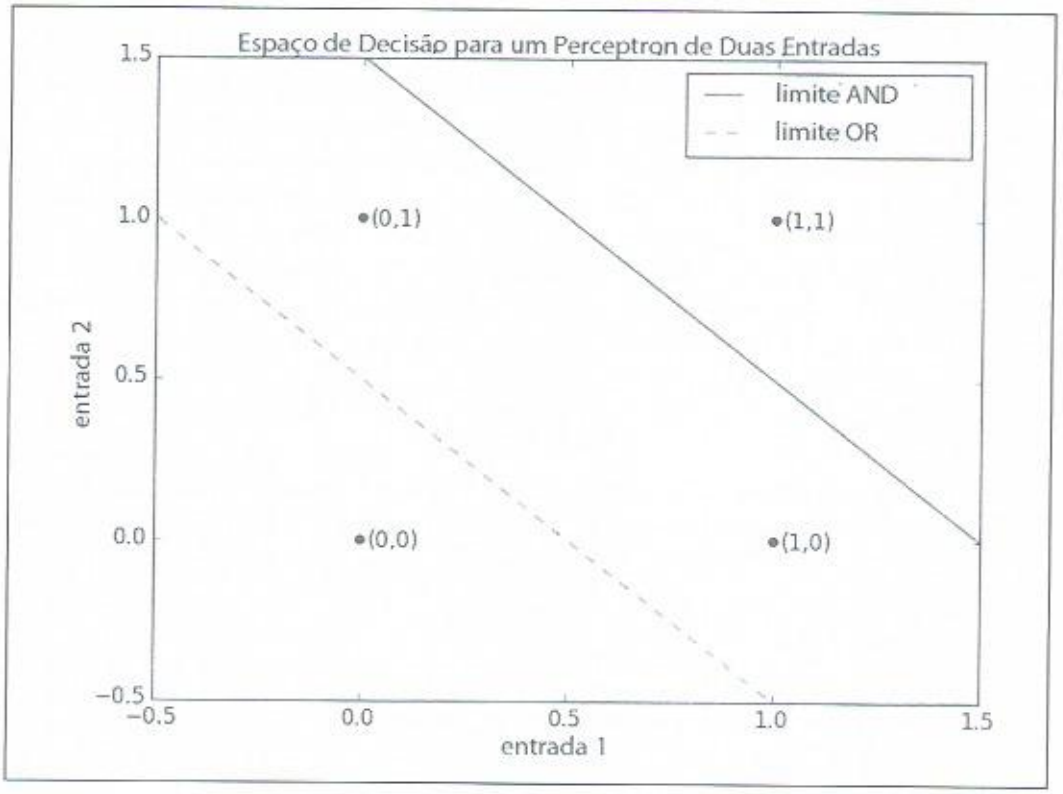

Figura 18-1. Espaço de decisão para um perceptron de duas entradas

E poderíamos construir uma *porta NOT* (que teria uma entrada e converteria 1 para 0 e 0 para 1) com:

```
weights = [-2]
bias = 1
```

Entretanto, existem alguns problemas que simplesmente não podem ser resolvidos com apenas um perceptron. Por exemplo, não importa o quanto você tente, você não pode usar um perceptron para construir um a *porta XOR* com saída 1 se exatamente uma de suas entradas for 1 ou então 0. É aí que começamos a precisar de redes neurais mais complicadas.

Claro, você não precisa da aproximação de um neurônio para construir uma porta lógica:

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Como neurônios reais, neurônios artificiais começam a ficar mais interessantes quando você começa a conectá-los.

Redes Neurais Feed-Forward

A topologia do cérebro é demasiadamente complicada, então é normal aproximá-la com uma rede neural *feed-forward* idealizada que consiste de *camadas* discretas de neurônios, cada uma conectada à seguinte. Isso tipicamente envolve uma camada de entrada (que recebe entradas e as transmite sem modificações), uma ou mais "camadas ocultas" (em que cada uma consiste de neurônios que pegam saídas da camada anterior, fazem algum cálculo e passam o resultado para a próxima camada), e uma camada de saída (que produz as saídas finais).

Assim como o perceptron, cada neurônio (não de entrada) possui o peso correspondente a cada uma de suas entradas e uma polarização (tendência). Para simplificar nossa representação, adicionaremos a polarização (bias) no final do nosso vetor de pesos e daremos a cada neurônio uma *entrada polarizada* que é sempre igual a 1.

Como com o perceptron, para cada neurônio somaremos os produtos de suas entradas e seus pesos. Mas aqui, em vez de gerar `step_function` aplicada àquele produto, exibiremos uma aproximação suave da função step. Usaremos a função `sigmoid` (Figura 18-2):

```
def sigmoid(t):
    return 1 / (1 + math.exp(-t))
```

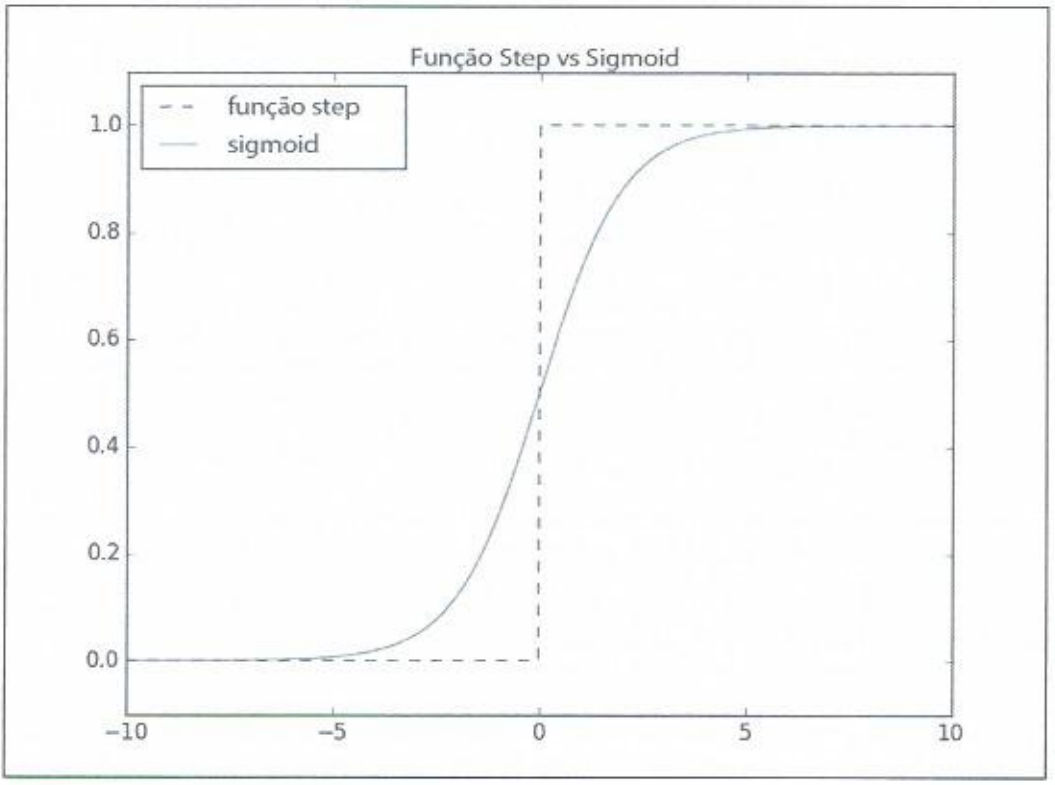

Figura 18-2. A função sigmoid

Por que usar `sigmoid` em vez de uma mais simples `step_function`? Para treinar uma rede neural, precisaremos usar cálculo, e para usar cálculo, precisaremos de funções *suaves*. A função step não é contínua, e sigmoid é uma boa aproximação suave dela.

Você deve se lembrar de `sigmoid` do Capítulo 16, onde era chamada de `logistic`. Tecnicamente, "sigmoid" se refere ao *formato* da função, "logística" a esta função específica embora as pessoas geralmente usem os termos indistintamente.

Nós podemos então calcular a saída como:

```
def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))
```

Dada essa função, nós podemos representar um neurônio simplesmente como uma lista de pesos cujo tamanho é mais do que o número de entradas daquele neurônio (por causa do peso bias). Então, podemos representar uma rede neural como uma lista de *camadas* (não de entrada), em que cada camada é apenas uma lista de neurônios naquela camada.

Isto é, representaremos uma rede neural como uma lista (camadas) de listas (neurônios) de listas (pesos).

Dada tal representação, usar a rede neural é bem simples:

```
def feed_forward(neural_network, input_vector):
    """recebe a rede neural
    (representada como uma lista de listas de listas de pesos)
    e retorna a saída a partir da entrada a se propagar"""

    outputs = []

    # processa uma camada por vez
    for layer in neural_network:
        input_with_bias = input_vector + [1]              # adiciona uma entrada polarizada
        output = [neuron_output(neuron, input_with_bias)  # computa a saída
                  for neuron in layer]                    # para cada neurônio
        outputs.append(output)                            # e memoriza

        # então a entrada para a próxima camada é a saída desta
        input_vector = output

    return outputs
```

Agora é fácil construir a porta XOR que não podíamos construir com um único perceptron. Só precisamos ajustar os pesos para que `neuron_outputs` seja bem próximo de 0 ou de 1:

```
xor_network = [# camada oculta
               [[20, 20, -30],      # neurônio 'and'
                [20, 20, -10]],     # neurônio 'or'
               # output layer
               [[-60, 60, -30]]]    # neurônio 'segunda entrada,
                                    # mas não a primeira entrada'

for x in [0, 1]:
    for y in [0, 1]:
        # feed_forward produz as saídas para todos os neurônios
        # feed_forward[-1] é a saída da camada de saída de neurônios
        print x, y, feed_forward(xor_network,[x, y])[-1]

# 0 0 [9.38314668300676e-14]
# 0 1 [0.9999999999999059]
# 1 0 [0.9999999999999059]
# 1 1 [9.383146683006828e-14]
```

Ao usar uma camada oculta, podemos transmitir a saída de um neurônio "and" e a saída de um neurônio "or" em um neurônio "segunda entrada mas não primeira entrada". O resultado é uma rede que realiza "or, mas não and", que é precisamente XOR (Figura 18-3).

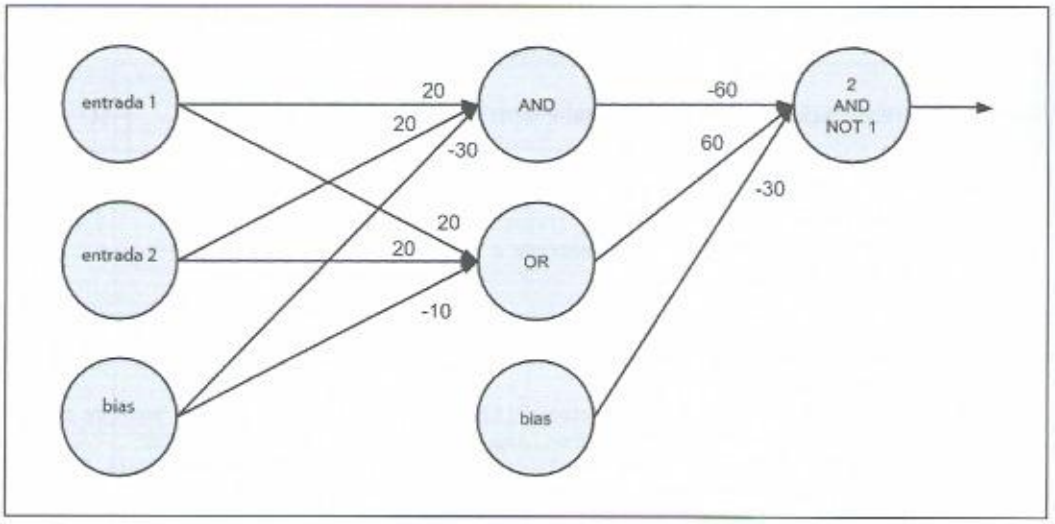

Figura 18-3. Uma rede neural para XOR

Backpropagation

Geralmente nós não construímos redes neurais manualmente. Isso se dá, em parte, porque as usamos para resolver problemas muito maiores — um problema de reconhecimento da imagem pode envolver dezenas ou milhares de neurônios. E em parte porque nós geralmente não conseguimos "raciocinar" sobre o que neurônios deveriam ser.

Em vez disso, nós usamos dados para *treinar* redes neurais. Uma abordagem popular é um algoritmo chamado *backpropagation* que possui semelhanças com o algoritmo gradiente descendente que vimos anteriormente.

Imagine que temos um conjunto de treinamento que consiste de vetores de entradas e correspondentes vetores alvos de saída. Por exemplo, em nosso exemplo anterior `xor_network`, o vetor de entrada `[1,0]` correspondia ao alvo de saída `[1]`. E imagine que nossa rede tem algum conjunto de pesos. Nós ajustamos os pesos usando o seguinte algoritmo:

1. Execute `feed_forward` em um vetor de entrada para produzir saídas de todos os neurônios na rede.
2. Isso resulta em um erro para cada neurônio de saída — a diferença entre sua saída e seu alvo.
3. Compute o gradiente para esse erro como uma função de pesos de neurônios e ajuste seus pesos na direção que mais diminui o erro.
4. "Propague" esses erros de saída de volta para inferir erros para as camadas ocultas.
5. Compute os gradientes desses erros e ajuste os pesos da camada oculta da mesma maneira.

Tipicamente, nós executamos o algoritmo muitas vezes para todo o nosso conjunto de treinamento até que a rede convirja:

```
def backpropagate(network, input_vector, targets):

    hidden_outputs, outputs = feed_forward(network, input_vector)

    # a saída * (1 - output) é da derivada da sigmoid
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, targets)]

    # ajusta os pesos para a camada de saída, um neurônio por vez
    for i, output_neuron in enumerate(network[-1]):
        # foca no i-ésimo neurônio da camada de saída
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            # ajusta o j-ésimo peso baseado em ambos
            # o delta deste neurônio e sua j-ésima entrada
            output_neuron[j] -= output_deltas[i] * hidden_output

    # erros de backpropagation para a camada oculta
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                     dot(output_deltas, [n[i] for n in output_layer])
                     for i, hidden_output in enumerate(hidden_outputs)]

    # ajusta os pesos para a camada oculta, um neurônio por vez
    for i, hidden_neuron in enumerate(network[0]):
        for j, input in enumerate(input_vector + [1]):
            hidden_neuron[j] -= hidden_deltas[i] * input
```

Isso é praticamente escrever explicitamente o erro ao quadrado como uma função de pesos e usar a função `minimize_stochastic` que construímos no Capítulo 8.

Neste caso, escrever explicitamente a função gradiente acaba sendo um tipo de dor. Se você sabe cálculo e a regra da cadeia, os detalhes matemáticos são relativamente diretos, mas manter a notação direta ("a derivada parcial da função de erro do peso que aquele neurônio i atribui à entrada vinda do neurônio j") não é tão divertido.

Exemplo: Derrotando um CAPTCHA

Para certificar que pessoas que estão se registrando em seu site são realmente pessoas, a vice-presidente da Gerência de Produtos quer que você implemente um CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) como parte do processo de registro. Em particular, ele gostaria de exibir aos usuários uma imagem de um dígito e exigir que eles forneçam aquele dígito para provar que são humanos.

Ele não acreditou quando você disse que computadores podem facilmente resolver esse problema, então você decide convencê-lo criando um programa que faça isso.

Representaremos cada dígito como uma imagem 5 × 5:

```
@@@@@  ..@..  @@@@@  @@@@@  @...@  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@
@...@  ..@..  ....@  ....@  @...@  @....  @....  ....@  @...@  @...@
@...@  ..@..  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
@...@  ..@..  @....  ....@  ....@  ....@  @...@  ....@  @...@  ....@
@@@@@  ..@..  @@@@@  @@@@@  ....@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
```

Nossa rede neural quer que uma entrada seja um vetor de números. Então transformaremos cada imagem em um vetor de tamanho 25, cujos elementos são 1 ("este pixel está na imagem") ou 0 ("este pixel não está na imagem").

Por exemplo, o dígito zero seria representado como:

```
zero_digit = [1,1,1,1,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,1,1,1,1]
```

Nós queremos que nossa saída indique qual dígito a rede neural pensa que é, então precisaremos de 10 saídas. A saída correta para o dígito 4, por exemplo, seria:

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Então, presumindo que nossas entradas estão ordenadas corretamente de 0 a 9, nossos alvos serão:

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]
```

para que (por exemplo) `targets[4]` seja a saída correta para o dígito 4.

Nesse ponto estamos prontos para construir nossa rede neural:

```
random.seed(0)      # para pegar resultados repetidos
input_size = 25     # cada entrada é um vetor de tamanho 25
num_hidden = 5      # teremos 5 neurônios na camada oculta
output_size = 10    # precisamos de 10 saídas para cada entrada

# cada neurônio oculto tem um peso por entrada, mais um peso bias
hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

# cada neurônio de saída tem um peso por neurônio oculto, mais o peso bias
output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]

# a rede começa com pesos aleatórios
network = [hidden_layer, output_layer]
```

E podemos treinar o algoritmo backpropagation:

```
# 10.000 iterações parecem ser o suficiente para convergir
for __ in range(10000):
    for input_vector, target_vector in zip(inputs, targets):
        backpropagate(network, input_vector, target_vector)
```

Isso funciona bem no conjunto de treinamento, obviamente:

```
def predict(input):
    return feed_forward(network, input)[-1]

predict(inputs[7])
# [0.026, 0.0, 0.0, 0.018, 0.001, 0.0, 0.0, 0.967, 0.0, 0.0]
```

O que indica que a saída de neurônio de dígito 7 produz 0,97, enquanto que todas as outras saídas de neurônios produzem números muito pequenos.

Mas também podemos aplicar isso a dígitos desenhados diferentes, como meu 3 estilizado:

```
predict([0,1,1,1,0,  # .@@@.
         0,0,0,1,1,  # ...@@
         0,0,1,1,0,  # ..@@.
         0,0,0,1,1,  # ...@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.92, 0.0, 0.0, 0.0, 0.01, 0.0, 0.12]
```

A rede ainda pensa que ele parece com um 3, enquanto meu 8 estilizado recebe votos para ser um 5, um 8 e um 9:

```
predict([0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.0, 0.0, 0.55, 0.0, 0.0, 0.93, 1.0]
```

Ter um conjunto de treinamento maior provavelmente ajudaria.

Embora a operação da rede não seja exatamente transparente, podemos inspecionar os pesos da camada oculta para entender o que estão reconhecendo. Podemos assinalar os pesos para cada neurônio como uma grade 5×5 correspondente às entradas 5×5.

Na vida real, você provavelmente marcaria pesos zero como brancos, com pesos maiores positivos mais e mais (digamos) verdes e negativos com (digamos) vermelho. Infelizmente, é muito difícil fazer isso em um livro preto e branco.

Em vez disso, marcaremos pesos zero com branco e pesos mais e mais distantes de zero cada vez mais escuros. E usaremos hachurado para indicar pesos negativos.

Para fazer isso, usaremos `pyplot.imshow`, que não vimos antes. Com isso, podemos assinalar imagens pixel por pixel. Normalmente isso não é usado para data science, mas aqui é uma boa escolha:

```
import matplotlib
weights = network[0][0]              # primeiro neurônio na camada oculta
abs_weights = map(abs, weights)      # a escuridão depende somente do valor absoluto
```

```
grid = [abs_weights[row:(row+5)]        # transforma os pesos em uma grade 5x5
        for row in range(0,25,5)]       # [pesos[0:5], …, pesos[20:25]]

ax = plt.gca()                          # para usar hachuras, precisamos de eixos

ax.imshow(grid,                         # aqui o mesmo que plt.imshow
          cmap=matplotlib.cm.binary,    # use a escala de cores preto e branco
          interpolation='none')         # assinala blocos como blocos

def patch(x, y, hatch, color):
    """retorna um objeto matplotlib 'patch' com a localização
    especificada, padrão de hachuras e cor"""
    return matplotlib.patches.Rectangle((x - 0.5, y - 0.5), 1, 1,
                                        hatch=hatch, fill=False, color=color)

# hachuras pesos negativos
for i in range(5):                      # linha
    for j in range(5):                  # coluna
        if weights[5*i + j] < 0:        # linha i, coluna j = pesos[5*i + j]
            # adiciona hachuras preto e brancas, visíveis sejam claras ou escuras
            ax.add_patch(patch(j, i, '/',  "white"))
            ax.add_patch(patch(j, i, '\\', "black"))

plt.show()
```

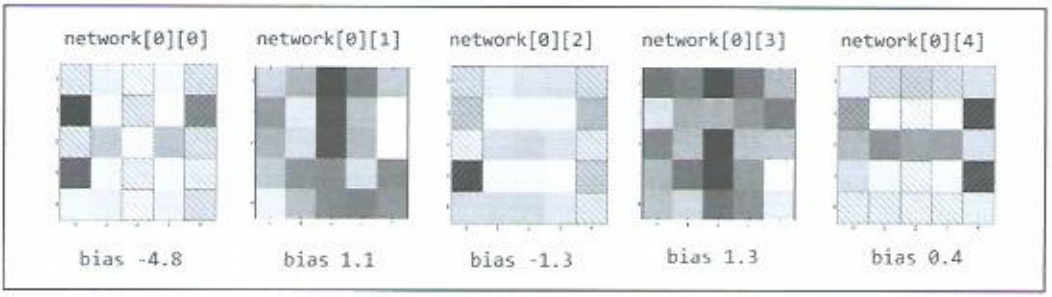

Figura 18-4. Pesos para a camada oculta

Na Figura 18-4, podemos ver que o primeiro neurônio oculto possui grandes pesos positivos na coluna da esquerda e no centro da fileira no meio, enquanto possui grandes pesos negativos na coluna da direita. (E você pode ver que possui grandes bias negativos, o que significa que não disparará a não ser que consiga precisamente as entradas positivas que está "procurando".)

Sem dúvidas, nessas entradas, ele faz o que esperamos:

```
left_column_only = [1, 0, 0, 0, 0] * 5
print feed_forward(network, left_column_only)[0][0]  # 1.0

center_middle_row = [0, 0, 0, 0, 0] * 2 + [0, 1, 1, 1, 0] + [0, 0, 0, 0, 0] * 2
print feed_forward(network, center_middle_row)[0][0]  # 0.95
```

```
right_column_only = [0, 0, 0, 0, 1] * 5
print feed_forward(network, right_column_only)[0][0]  # 0.0
```

Similarmente, o neurônio oculto do meio parece "gostar" de linhas horizontais mas não de linhas diagonais, e o último neurônio oculto parece "gostar" da fileira do centro mas não da coluna do meio. (É difícil de interpretar os outros dois neurônios.)

O que acontece quando executamos meu 3 estilizado na rede?

```
my_three =  [0,1,1,1,0,  # .@@@.
             0,0,0,1,1,  # ...@@
             0,0,1,1,0,  # ..@@.
             0,0,0,1,1,  # ...@@
             0,1,1,1,0]  # .@@@.

hidden, output = feed_forward(network, my_three)
```

As saídas `hidden` são:

```
0.121080  # from network[0][0], provavelmente excedido por (1, 4)
0.999979  # from network[0][1], grandes contribuições de (0, 2) e (2,2)
0.999999  # from network[0][2], positivo em todos os lugares menos (3,4)
0.999992  # from network[0][3], mais uma vez grandes contribuições de (0,2) e (2,2)
0.000000  # from network[0][4], negativo ou zero em todos os lugares,
                                menos na fileira do centro
```

que entra no neurônio de saída "three"(três) com pesos `network[-1][3]`:

```
-11.61  # peso para oculto[0]
 -2.17  # peso para oculto[1]
  9.31  # peso para oculto[2]
 -1.38  # peso para oculto[3]
-11.47  # peso para oculto[4]
- 1.92  # peso da entrada polarizada
```

De modo que o neurônio compute:

```
sigmoid(.121 * -11.61 + 1 * -2.17 + 1 * 9.31 - 1.38 * 1 - 0 * 11.47 - 1.92)
```

que é 0,92, como vimos. Na essência, a camada oculta está computando cinco divisões diferentes de espaço dimensional 25, mapeando cada entrada dimensional 25 para cinco números. E então cada neurônio de saída olha apenas para os resultados daquelas cinco divisões.

Como vimos, `my_three` cai levemente na parte "inferior" da partição 0 (isto é, apenas ativa levemente o neurônio oculto 0), longe da parte "superior" das partições 1, 2 e 3 (isto é, ativa fortemente aqueles neurônios ocultos), e longe da parte inferior da partição 4 (isto é, não ativa nenhum neurônio).

E cada um dos 10 neurônios de saída usa apenas aquelas cinco ativações para decidir se `my_three` é seu dígito ou não.

Para Mais Esclarecimentos

- A Coursera tem um curso gratuito sobre Neural Networks for Machine Learning (*https://www.coursera.org/course/neuralnets*). O último curso foi em 2012, mas os materiais do curso ainda estão disponíveis.
- Michael Nielsen está escrevendo um livro online gratuito sobre Neural Networks and Deep Learning (*http://neuralnetworksanddeeplearning.com/*). Quando você ler este livro, ele já deve ter terminado.
- PyBrain (*http://pybrain.org*) é uma biblioteca Python simples de rede neural.
- Pylearn2 (*http://deeplearning.net/software/pylearn2/*) é uma biblioteca de rede neural muito mais avançada (e muito mais difícil de usar).

CAPÍTULO 19

Agrupamento

Onde tínhamos tais agrupamentos
Nos tornou nobremente selvagens, não insanos
— Robert Herrick

A maioria dos algoritmos neste livro são o que é conhecido por aprendizado supervisionado, no que começam com um conjunto de dados *rotulados* e os usam como base para fazer previsões sobre novos dados, não rotulados. Agrupamento, entretanto, é um exemplo de aprendizado não supervisionado, em que nós trabalhamos com dados completamente não rotulados (ou no qual nosso dado possui rótulo mas nós o ignoramos).

A Ideia

Quando você olha para alguma fonte de dados é normal que os dados, de alguma forma, formem *agrupamentos*. Um conjunto de dados que mostre onde milionários moram provavelmente possui agrupamentos em lugares como Beverly Hills e Manhattan. Um conjunto de dados que mostre quantas horas as pessoas trabalham semanalmente provavelmente possui um agrupamento por volta de 40 (e se for tirado de um estado com leis exigindo benefícios especiais para pessoas que trabalham pelo menos 20 horas por semana, provavelmente terá outro agrupamento por volta de 19). Um conjunto de dados demográficos de eleitores registrados provavelmente forma uma variedade de agrupamentos (por exemplo: "mães de praticantes de futebol", "aposentados entediados", "jovens desempregados") que pesquisadores de opinião pública e consultores políticos devem considerar relevantes.

Diferente de alguns dos problemas que vimos, geralmente não há agrupamento "correto". Um esquema de agrupamento alternativo pode agrupar alguns dos "jovens desempregados" com "estudantes de pós-graduação", outros com "moradores do porão dos pais". Nenhum esquema é necessariamente mais correto — pelo contrário, cada um é melhor no que diz respeito à sua própria métrica "quão bons são os agrupamentos?"

Além disso, os agrupamentos não se rotulam sozinhos. Você terá que fazer isso vendo os dados contidos em cada um.

O Modelo

Para nós, cada entrada (*input*) será um vetor em espaço dimensional *d* (que representaremos como uma lista de números). Nosso objetivo será identificar agrupamentos de entradas similares e,às vezes, encontrar um valor representativo para cada agrupamento.

Por exemplo, cada entrada poderia ser (um vetor numérico que de alguma forma representa) o título de um post de um blog, em cujo caso o objetivo poderia ser encontrar agrupamentos de posts similares, talvez para entender sobre o que nossos usuários estão falando no blog. Ou imagine que temos uma imagem contendo milhares de cores (`red, green, blue`) e que nós precisamos tirar uma cópia de uma versão de 10 cores dela. O agrupamento nos ajuda a escolher 10 cores que minimizarão o "erro de cor" total.

Um dos métodos de agrupamento mais simples é a *k-means*, na qual um número de agrupamentos *k* é escolhido antecipadamente, depois do que o objetivo é particionar as entradas em conjuntos $S_1, \ldots, S_k$ de uma forma que minimize a soma total das distâncias quadradas de cada ponto para a média de seu agrupamento designado.

Há muitas formas de definir pontos *n* para agrupamentos *k*, o que significa que encontrar o melhor agrupamento é um problema bem difícil. Nós aceitaremos um algoritmo iterativo que usualmente encontra um bom agrupamento:

1. Comece com um conjunto de *k-means*, que são pontos em espaço dimensional *d*.
2. Associe cada ponto com a média (k-means) mais próxima.
3. Se nenhuma associação de ponto de atribuição mudou, pare e mantenha os agrupamentos.
4. Se alguma associação mudar, compute novamente as médias e volte ao passo 2.

Usando a função `vector_mean` do Capítulo 4, é bem fácil criar uma classe que faça isso:

```
class KMeans:
    """executa agrupamentos k-means"""

    def __init__(self, k):
        self.k = k          # número de agrupamentos
        self.means = None   # ponto médio de agrupamentos

    def classify(self, input):
        """retorna o índice do agrupamento mais próximo da entrada"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs):
        # escolha pontos k aleatórios como média inicial
        self.means = random.sample(inputs, self.k)
        assignments = None
```

```
        while True:
            # encontre novas associações
            new_assignments = map(self.classify, inputs)

            # se nenhuma associação mudou, terminamos.
            if assignments == new_assignments:
                return

            # senão, mantenha as novas associações,
            assignments = new_assignments

            # e compute novas médias, baseado nas novas associações
            for i in range(self.k):
                # encontre todos os pontos associados ao agrupamento i
                i_points = [p for p, a in zip(inputs, assignments) if a == i]

                # certifique-se que i_points não está vazio,
                # para não dividir por 0
                if i_points:
                    self.means[i] = vector_mean(i_points)
```

Vamos ver como isso funciona.

Exemplo: Encontros

Para celebrar o crescimento da DataSciencester, a vice-presidente de Recompensas para Usuário quer organizar vários encontros presenciais para os usuários de sua cidade natal, completos com cerveja, pizza e camisetas DataSciencester. Você sabe a localização de todos os seus usuários locais (Figura 19-1), e ela gostaria que você escolhesse locais de encontro para que fique mais fácil para todos comparecerem.

Dependendo de como você enxerga, verá dois ou três agrupamentos. (É fácil fazer isso visualmente porque os dados estão apenas em duas dimensões. Com mais dimensões, seria mais difícil de visualizar.)

Primeiro imagine que ela possui orçamento o suficiente para três encontros. Você vai até seu computador e tenta isso:

```
random.seed(0)          # para que você consiga os mesmos
clusterer = KMeans(3)   # resultados que eu
clusterer.train(inputs)
print clusterer.means
```

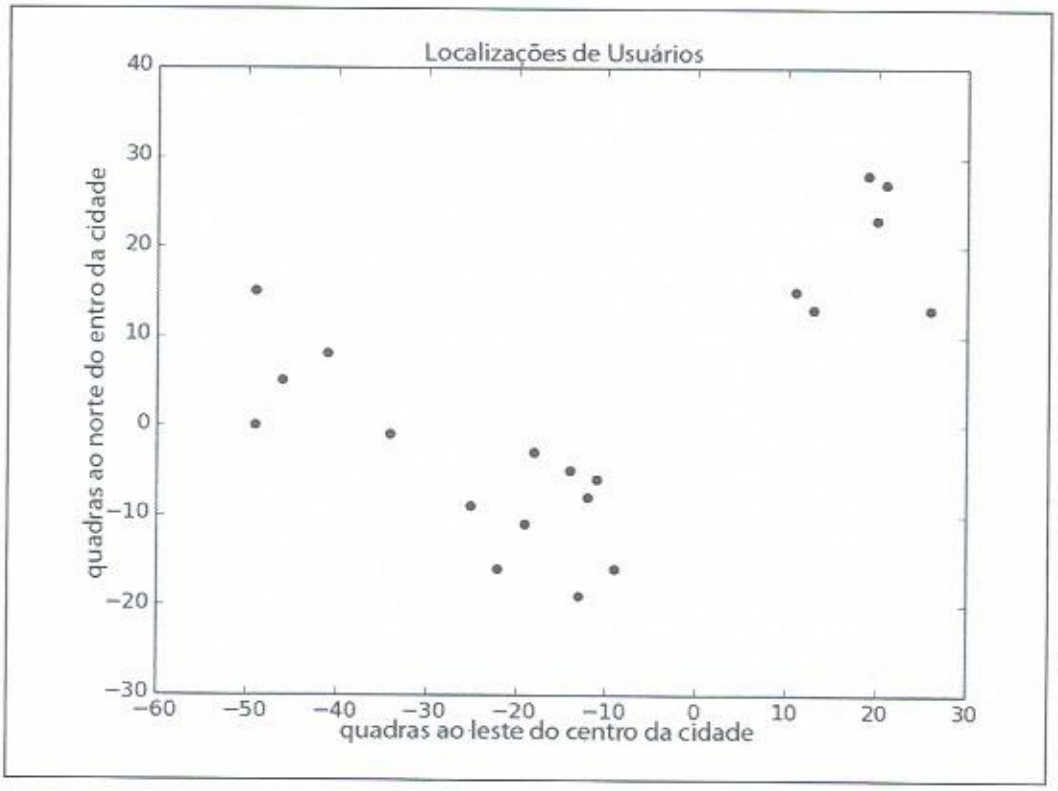

Figura 19-1: As localizações dos usuários de sua cidade natal

Você encontra três agrupamentos centralizados em [-45,4], [-16,10], e [18,20], e você procura locais de encontro perto dessas localizações (Figura 19-2).

Você mostra isso à vice-presidente, que o informa que agora ela só tem orçamento para *dois* encontros.

"Sem problemas", você diz:

```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

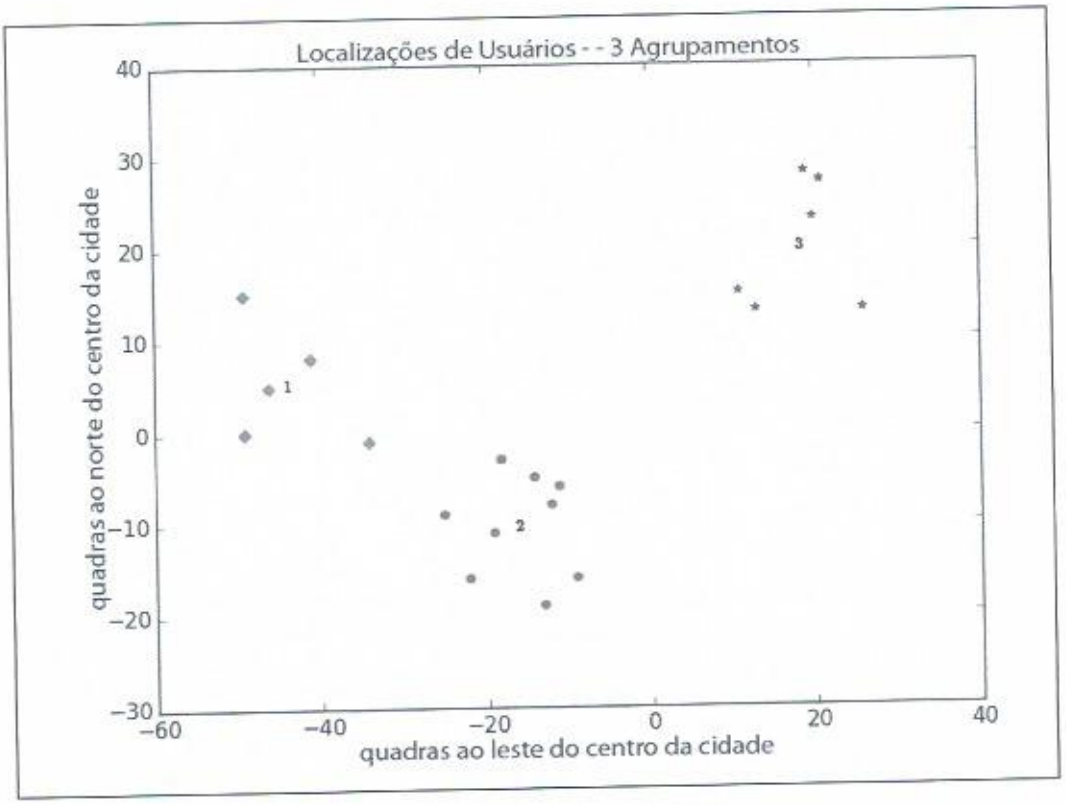

Figura 19-2: As localizações de usuários agrupadas em três agrupamentos

Como exibido na Figura 19-3, um encontro ainda deveria estar perto [18,20], mas agora o outro deve estar perto [-26,-5].

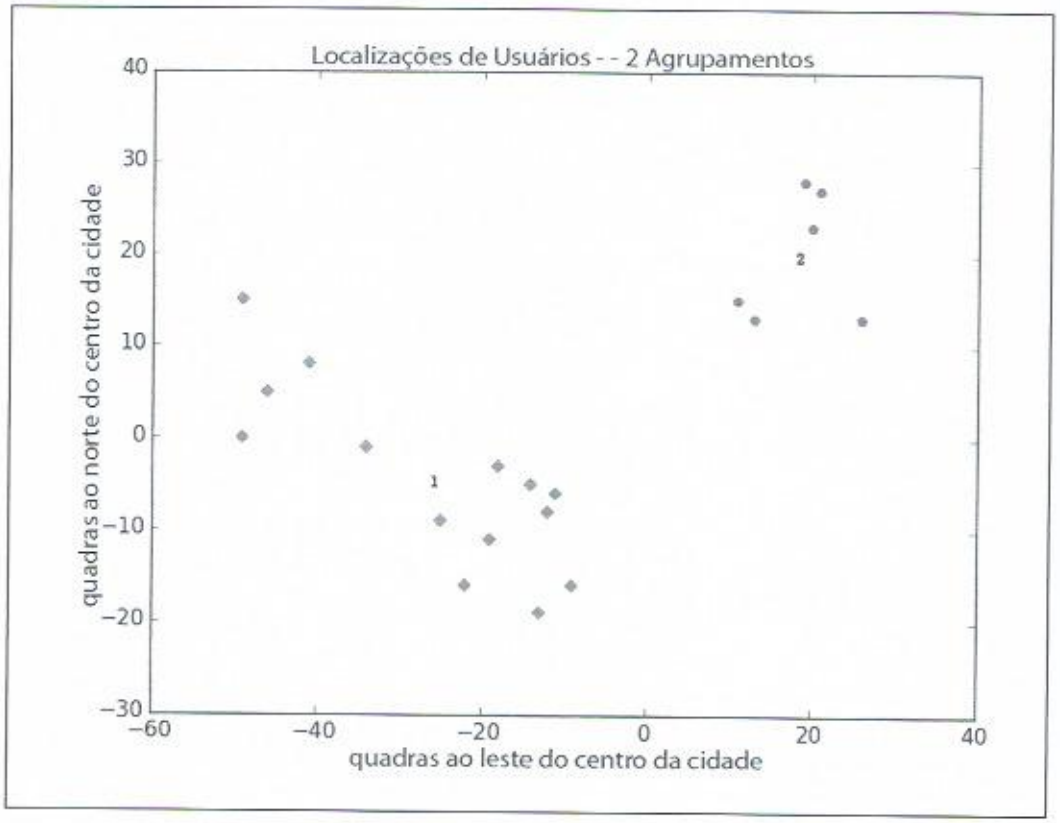

Figura 19-3: As localizações de usuários agrupadas em dois agrupamentos

Escolhendo k

No exemplo anterior, a escolha de *k* foi levada por fatores fora do nosso controle. No geral, esse não seria o caso. Há uma grande variedade de caminhos para escolher um *k*. Uma que é razoavelmente fácil de entender envolve marcar a soma dos erros ao quadrado (entre cada ponto e a média de seu agrupamento) como uma função de *k* e olhar para onde o gráfico "dobra":

```
def squared_clustering_errors(inputs, k):
    """encontra o erro ao quadrado total de k-means agrupando as entradas"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))

# agora faça o gráfico de 1 até len(inputs) agrupamentos

ks = range(1, len(inputs) + 1)
```

```
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("total de erros ao quadrado")
plt.title("Erro Total vs. Número de Agrupamentos")
plt.show()
```

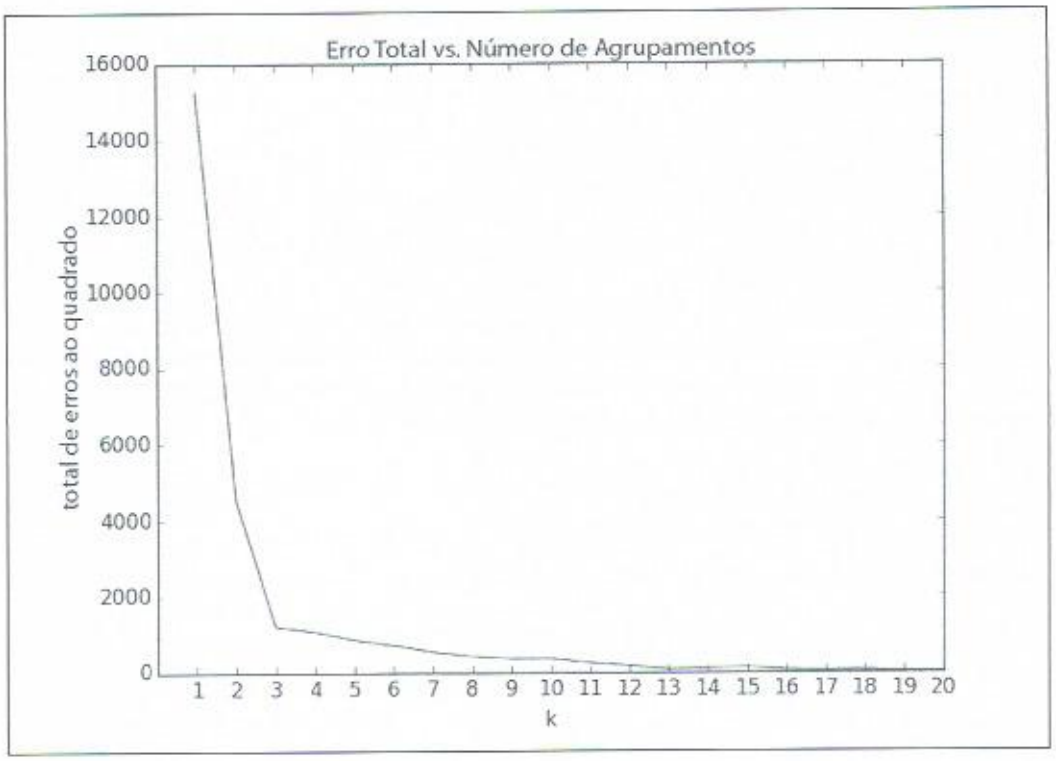

Figura 19-4. Escolhendo um k

Olhando para a Figura 19-4, esse método coincide com sua visão original que 3 é o número "certo" de agrupamentos.

Exemplo: Agrupando Cores

A vice-presidente da Swag criou adesivos atraentes DataSciencester que eles gostariam que você entregasse nos encontros. Infelizmente, sua impressora de adesivos pode imprimir no máximo cinco cores por adesivo. E como a vice-presidente de Arte está de licença, a vice-presidente da Swag perguntou se há alguma forma de você modificar o design para que contenha cinco cores.

As imagens de computador podem ser representadas como um array de pixels de duas dimensões, onde cada pixel possui um vetor de três dimensões (`red, green, blue`) indicando sua cor.

Criar uma versão de cinco cores da imagem requer:

1. Escolher cinco cores
2. Designar uma destas cores para cada pixel

Essa é uma excelente tarefa para agrupar a k-means, que pode particionar os pixels em cinco agrupamentos em um espaço vermelho, verde e azul. Se então recolorirmos os pixels em cada agrupamento para a cor média, terminamos.

Para começar, precisaremos de uma maneira de carregar uma imagem em Python. Podemos fazer isso com `matplotlib`:

```
path_to_png_file = r"C:\images\image.png"   # onde sua imagem está
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

Por trás das cenas, `img` é um array NumPy, mas para nossos objetivos, podemos tratá-lo como uma lista de listas de listas.

`img[i][j]` é o pixel na i-ésima linha e na coluna j-ésima, e cada pixel é uma lista `[red, green, blue]` de números entre 0 e 1 indicando a cor para aquele pixel (*http://en.wikipedia.org/wiki/RGB_color_model*):

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

Em particular, podemos conseguir uma lista estável para todos os pixels, como:

```
pixels = [pixel for row in img for pixel in row]
```

e então abastecê-las ao nosso agrupamento:

```
clusterer = KMeans(5)
clusterer.train(pixels)   # isso pode demorar um pouco
```

Uma vez terminado, apenas construímos uma imagem nova com o mesmo formato:

```
def recolor(pixel):
    cluster = clusterer.classify(pixel)        # índice do agrupamento mais próximo
    return clusterer.means[cluster]            # ponto médio do agrupamento mais próximo

new_img = [[recolor(pixel) for pixel in row]   # recolore esta linha de pixels
           for row in img]                     # para cada linha na imagem
```

e exibe usando `plt.imshow()`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

É difícil exibir os resultados de cores em um livro preto e branco, mas a Figura 19-5 mostra versões em escala de cinza de uma imagem em cores e a saída para usar esse processo para reduzi-la para cinco cores:

Figura 19-5. Imagem original e sua descoloração de média 5

Agrupamento Hierárquico Bottom-up

Uma abordagem alternativa para agrupamento é "criar" agrupamentos bottom-up. Podemos fazer isso da seguinte forma:

1. Faça de cada entrada seu próprio agrupamento de um.
2. Enquanto houver múltiplos agrupamentos sobrando encontre os dois agrupamentos mais próximos e os junte.

No final, teremos um agrupamento gigante contendo todas as entradas. Se quisermos acompanhar a ordem de junção, podemos recriar qualquer número de agrupamentos desfazendo junções. Por exemplo, se quisermos três agrupamentos, podemos simplesmente desfazer as duas últimas junções.

Nós usaremos uma representação muito simples de agrupamento. Nossos valores estarão em agrupamentos *folha*, que representaremos como tuplas de 1:

```
leaf1 = ([10, 20],)   # para fazer uma tupla de 1 você precisa de virgulas
leaf2 = ([30, -15],)  # senão Python interpreta os parênteses como parênteses
```

Usaremos estes para criar agrupamentos *fundidos*, os quais representaremos como tuplas de 2 (ordem de junção, filhos):

```
merged = (1, [leaf1, leaf2])
```

Falaremos da ordem de junção daqui a pouco, mas, enquanto isso, vamos criar algumas funções auxiliares:

```
def is_leaf(cluster):
    """um agrupamento é uma folha se tiver tamanho 1"""
    return len(cluster) == 1

def get_children(cluster):
    """retorna os dois filhos desse agrupamento se for um agrupamento fundido;
    cria uma exceção se for um agrupamento folha"""
    if is_leaf(cluster):
        raise TypeError("um agrupamento folha não tem filhos")
    else:
        return cluster[1]

def get_values(cluster):
    """retorna o valor neste agrupamento (se for um agrupamento folha)
    ou todos os valores nos agrupamentos folha abaixo dele (se não for)"""
    if is_leaf(cluster):
        return cluster      # já é uma tupla de 1 contendo valor
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

A fim de fundir os agrupamentos mais próximos, precisamos de alguma noção de distância entre agrupamentos. Usaremos a distância *mínima* entre elementos de dois agrupamentos, que funde os dois agrupamentos mais próximos (mas, às vezes, produzirá grandes agrupamentos em cadeia que não são tão próximos). Se quiséssemos ajustar dois agrupamentos esféricos, usaríamos a distância *máxima*, pois ela funde dois agrupamentos que se encaixam na menor bola. Ambas escolhas são comuns, assim como é a distância *média:*

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """computa todas as distâncias entre cluster1 e cluster2
    e aplica _distance_agg_ na lista resultante"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in get_values(cluster2)])
```

Usaremos a ordem de junção para acompanhar a ordem que fizemos o agrupamento. Números menores representarão junções *tardias*. Isso significa que quando quisermos desfazer a junção de agrupamentos, o fazemos da menor para a maior junção. Como agrupamentos folha nunca foram fundidos, iremos atribuir infinito (inf) a eles:

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0]  # a ordem de junção é o primeiro elemento de tupla de 2
```

Agora estamos prontos para criar o algoritmo de agrupamentos:

```
def bottom_up_cluster(inputs, distance_agg=min):
    # começa com cada entrada como um agrupamento folha / tupla de 1
    clusters = [(input,) for input in inputs]
```

```
    # enquanto tivermos mais de um agrupamento folha restante…
    while len(clusters) > 1:
        # encontra os dois agrupamentos mais próximos
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                      key=lambda (x, y): cluster_distance(x, y, distance_agg))

        # remove-os da lista de agrupamentos
        clusters = [c for c in clusters if c != c1 and c != c2]

        # faz a junção deles, usando a ordem de junção = números de agrupamentos restantes
        merged_cluster = (len(clusters), [c1, c2])

        # e adiciona a junção deles
        clusters.append(merged_cluster)

    # quando sobrar apenas um agrupamento, retorne-o
    return clusters[0]
```

Seu uso é bem simples:

```
base_cluster = bottom_up_cluster(inputs)
```

Isso produz um agrupamento cuja representação estranha é:

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                       ([20, 23],)]),
                 ([26, 13],)]),
            (16, [([11, 15],),
                  ([13, 13],)])]),
       (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                                 ([-46, 5],)]),
                           ([-41, 8],)]),
                       ([-49, 15],)]),
                 ([-34, -1],)]),
            (6, [(7, [(8, [(10, [([-22, -16],),
                                 ([-19, -11],)]),
                           ([-25, -9],)]),
                       (13, [(15, [(17, [([-11, -6],),
                                         ([-12, -8],)]),
                                  ([-14, -5],)]),
                            ([-18, -3],)])]),
                 (12, [([-13, -19],),
                       ([-9, -16],)])])])])
```

Para cada agrupamento fundido, eu alinhei seus filhos verticalmente. Se dissermos "agrupamento 0" para o agrupamento com ordem de junção 0, você pode interpretar como:

- Agrupamento 0 é a junção do agrupamento 1 e do agrupamento 2.
- Agrupamento 1 é a junção do agrupamento 3 e do agrupamento 16.

- Agrupamento 16 é a junção da folha[11, 15] e da folha[13, 13].
- E assim por diante...

Como tínhamos 20 entradas, foram necessárias 19 junções para conseguir esse agrupamento. A primeira junção criou o agrupamento 18 combinando as folhas [19, 28] e [21, 27]. E a última junção criou o agrupamento 0.

No entanto, geralmente não queremos representações ruins como essa. (Mesmo que esse possa ser um exercício interessante para criar visualizações amigáveis ao usuário de hierarquia de agrupamento.) Em vez disso, vamos escrever uma função que gera qualquer número de agrupamentos desfazendo o número apropriado de junções:

```
def generate_clusters(base_cluster, num_clusters):
    # comece com uma lista apenas com o agrupamento base
    clusters = [base_cluster]

    # desde que ainda não tenhamos agrupamentos o suficiente…
    while len(clusters) < num_clusters:
        # escolha o que foi fundido por último
        next_cluster = min(clusters, key=get_merge_order)
        # remova-o da lista
        clusters = [c for c in clusters if c != next_cluster]
        # e adicione seus filhos à lista, isto é, desfaça a função
        clusters.extend(get_children(next_cluster))

    # uma vez que temos agrupamentos o suficiente…
    return clusters
```

Então, por exemplo, se queremos gerar três agrupamentos, só precisamos fazer:

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

que podemos representar facilmente:

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # truque mágico de descompactar
    plt.scatter(xs, ys, color=color, marker=marker)

    # coloca um número no ponto médio do agrupamento
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')

plt.title("Localizações de Usuários - 3 Agrupamentos Bottom-up, Min")
plt.xlabel("quadras ao leste do centro da cidade")
plt.ylabel("quadras ao norte do centro da cidade")
plt.show()
```

Isso dá resultados muito diferentes dos que a k-means, como exibido na Figura 19-6.

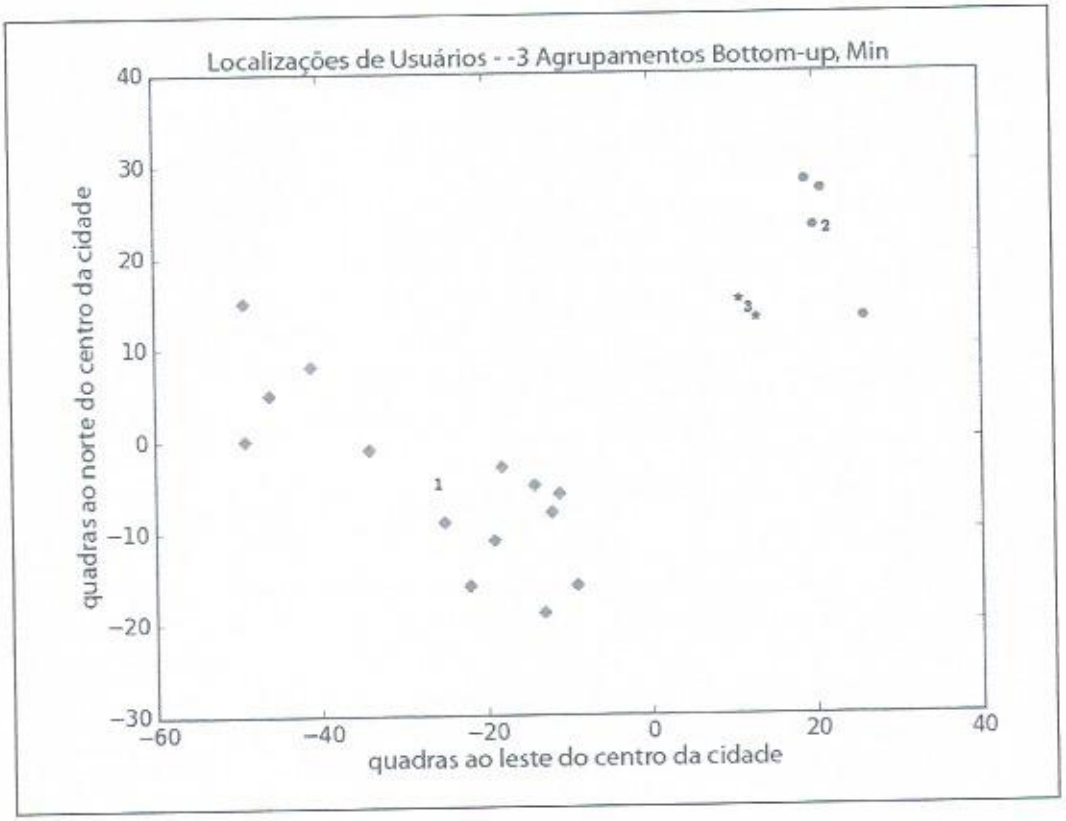

Figura 19-6. Três agrupamentos bottom-up usando distância mínima

Como mencionamos anteriormente, isso se dá porque usar `min` em `cluster_distance` tende a criar agrupamentos em cadeia. Se usarmos `max` parece (nos fornece agrupamentos próximos) igual ao resultado de média 3 (Figura 19-7).

A implementação `bottom_up_clustering` acima é relativamente simples, mas também é chocantemente ineficiente. Ela computa a distância entre cada par de entrada em cada passo. Uma implementação mais eficiente poderia pré-computar as distâncias entre cada par de entradas e então dar uma olhada dentro de `cluster_distance`. Uma implementação *realmente* eficiente também lembraria de `cluster_distances` do passo anterior.

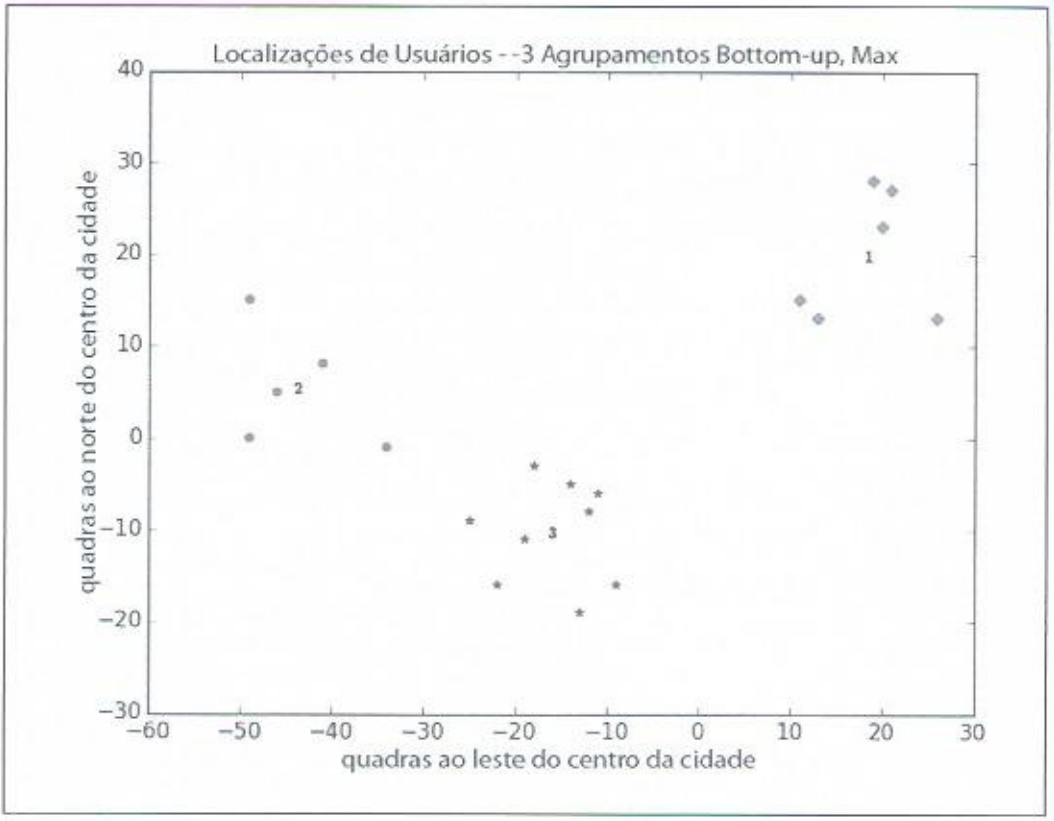

Figura 19-7. Três Agrupamentos bottom-up usando distância máxima

Para Mais Esclarecimentos

- `scikit-learn` possui um módulo completo `sklearn.cluster` (*http://scikit-learn.org/stable/modules/clustering.html*) que contém vários algoritmos de agrupamento incluindo `KMeans` e o algoritmo hierárquico `Ward` de agrupamento (que usa um critério diferente para unir os agrupamentos do que os nossos módulos).
- SciPy (*http://www.scipy.org/*) possui dois modelos de agrupamentos `scipy.cluster.vq` (que faz a k-means) e `scipy.cluster.hierarchy` (que possui uma variedade de algoritmos de agrupamento hierárquicos).

CAPÍTULO 20

Processamento de Linguagem Natural

Eles foram a um grande banquete de linguagens e roubaram as sobras.
—William Shakespeare

Processamento de Linguagem Natural (Natural Language Processing — NLP) refere-se a técnicas computacionais envolvendo linguagem. É um campo amplo, mas veremos algumas técnicas simples e outras não.

Nuvens de Palavras

No Capítulo 1, nós computamos contagem de palavras de interesse de usuários. Uma técnica para visualizar e contar palavras é nuvem de palavras, que é artisticamente desenhar as palavras com tamanhos proporcionais às suas contagens.

No geral, os cientistas de dados não penam muito em nuvens de palavras, em grande parte porque a colocação das palavras não significa nada além de "este é um espaço onde eu consegui encaixar uma palavra".

Se você for forçado a criar uma nuvem de palavras, pense se quer fazer os eixos transmitirem alguma coisa. Por exemplo, imagine que para cada coleção de dados de jargões relacionados à ciência você tenha dois números entre 0 e 100 — o primeiro representando a frequência que ele aparece em postagens de empregos e o segundo a frequência que aparece em currículos:

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
         ("actionable insights", 40, 30), ("think out of the box", 45, 10),
         ("self-starter", 30, 50), ("customer focus", 65, 15),
         ("thought leadership", 35, 35)]
```

A abordagem nuvem de palavras é apenas para organizar as palavras na página usando uma fonte bonita (Figura 20-1).

Figura 20-1. Nuvem de Jargões

Isso parece legal mas não nos diz nada. Uma abordagem mais interessante poderia ser dispersá-las para que a posição horizontal indicasse popularidade de postagens e a vertical popularidade de currículos, o que produziria uma visualização que transmitiria alguns *insights* (Figura 20-2):

```
def text_size(total):
    """igual a 8 se o total for 0, 28 se o total for 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularidade em Postagens de Empregos")
plt.ylabel("Popularidade em Curriculos")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

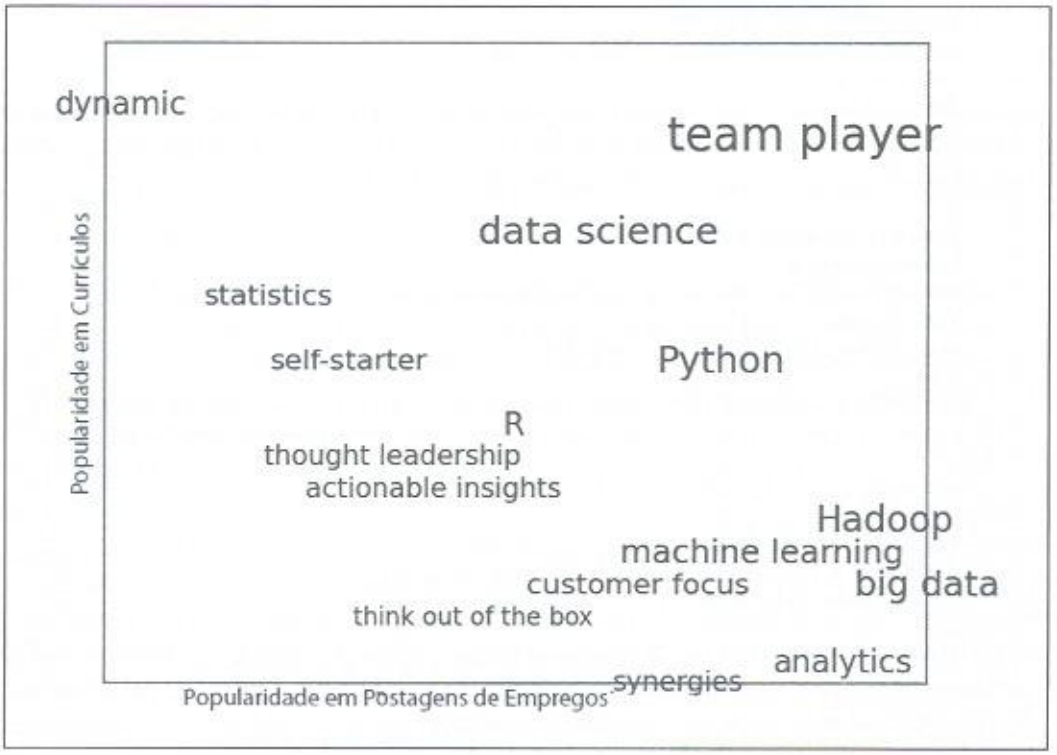

Figura 20-2. Uma nuvem de palavras mais significativa (se menos atrativa)

Modelos n-gramas

A vice-presidente de Marketing de Pesquisa quer que você crie milhares de páginas web sobre data science para que seu site seja classificado no topo dos resultados de pesquisa para os termos relacionados. (Você tenta explicar que os algoritmos dos mecanismos de pesquisas são espertos o bastante e que isso não funcionará, mas ela se recusa a escutar.)

Claro, ela não quer escrever milhares de páginas web, nem quer pagar uma horda de "estrategistas de conteúdo" para fazê-lo. Em vez disso, ela pergunta se você pode, de alguma forma, gerar estas páginas. Para fazer isso, precisaremos de alguma linguagem de modelagem.

Uma abordagem é começar com um corpo de documentos e aprender um modelo estatístico de linguagem. No nosso caso, começaremos com o ensaio de Mike Loukidess "What is data science?" (*http://oreil.ly/1Cd6ykN*)

Como no Capítulo 9, usaremos `requests` e `BeautifulSoup` para recuperar os dados. Há alguns problemas nos quais precisamos prestar atenção.

O primeiro é que os apóstrofos no texto são na verdade o caractere Unicode `u"\u2019"`. Criaremos uma função auxiliar para substituí-las por apóstrofos normais.

```
def fix_unicode(text):
    return text.replace(u"\u2019", "'")
```

O segundo problema é que uma vez que conseguirmos o texto da página da web, vamos querer dividi-lo em uma sequência de palavras e pontos (para que possamos dizer onde as sentenças terminam). Podemos fazer isso usando `re.findall()`:

```
from bs4 import BeautifulSoup
import requests
url = "http://radar.oreilly.com/2010/06/what-is-data-science.html"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "entry-content")   # encontra conteúdo de entrada div
regex = r"[\w']+|[\.]"                        # combina uma palavra ou um ponto

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

Nós certamente poderíamos (e deveríamos) limpar um pouco mais esses dados. Ainda há uma quantidade de texto extrínseco no documento (por exemplo, a primeira palavra é "section"), nós dividimos em pontos no meio da sentença (por exemplo, em "Web 2.0) e existem legendas úteis e listas espalhadas por todo lado. Dito isso, trabalharemos com o `document` como ele está.

Agora que nós temos o texto como uma sequência de palavras, nós podemos modelar uma linguagem da seguinte forma: dada alguma palavra inicial (como "book") olhamos para todas as palavras seguintes nos documentos fonte (aqui "isn't", "a", "shows", "demonstrates", e "teaches"). Nós escolhemos aleatoriamente umas dessas para ser a próxima palavra e repetimos o processo até chegar no ponto, que significa o final da sentença. Nós chamamos isso de *modelo bigrama*, por ser determinado completamente por sequências de bigramas (pares de palavra) nos dados originais.

Mas e a palavra inicial? Nós podemos apenas escolher aleatoriamente das palavras que *seguem* o ponto. Para começar, vamos pré-computar as possíveis transições de palavras. Lembre-se que `zip` para quando qualquer uma de suas entradas termina, para que `zip(document, document[1:])` nos dê precisamente os pares de elementos consecutivos do documento:

```
bigrams = zip(document, document[1:])
transitions = defaultdict(list)
for prev, current in bigrams:
    transitions[prev].append(current)
```

Agora estamos prontos para gerar sentenças:

```
def generate_using_bigrams():
    current = "."   # isso significa que a próxima palavra começará uma sentença
    result = []
```

```
    while True:
        next_word_candidates = transitions[current]    # bigramas (current, _)
        current = random.choice(next_word_candidates)  # escolhe um aleatoriamente
        result.append(current)                         # anexa-o aos resultados
        if current == ".": return " ".join(result)     # se "." terminamos
```

As sentenças que produz são besteiras, mas são o tipo de besteira que você deveria colocar no seu web site se está tentando fazer parecer com data science. Por exemplo:

> Você deve saber quais são você quer dados ordenar dados abastecer web amigo alguém em tópicos de tendência como os dados em Hadoop é data science requer um livro demonstrar por que visualizações são mas nós fazemos correlações massivas através muitos comerciais disco rígido em linguagem Python e cria forma mais manejável fazendo conexões então usa e usa isso para resolver os dados.
>
> —Modelo Bigrama

Nós podemos tornar as sentenças menos bobas olhando para *trigrams*, trios de palavras consecutivas. (De modo mais geral, você pode olhar para *n-grams* com *n* palavras consecutivas, mas três serão o bastante para nós.) Agora as transições dependerão das *duas* palavras anteriores:

```
trigrams = zip(document, document[1:], document[2:])
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in trigrams:

    if prev == ".":              # se a "palavra" anterior era um ponto
        starts.append(current)   # então esta é uma palavra inicial

    trigram_transitions[(prev, current)].append(next)
```

Note que agora temos que acompanhar as palavras inicias separadamente. Podemos gerar sentenças praticamente da mesma forma:

```
def generate_using_trigrams():
    current = random.choice(starts)   # escolha uma palavra inicial aleatória
    prev = "."                        # e a preceda com um '.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)

        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

Isso produz sentenças melhores como:

> Em retrospecto MapReduce parece com uma epidemia e caso seja ele nos dá novos insights em como a economia funciona Isso não é uma pergunta nós poderíamos até ter perguntado a alguns anos houve instrumentação.
>
> — Modelo Trigrama

Claro, elas parecem melhor porque em cada passo o processo de geração possui menos escolhas e em muitos passos apenas uma escolha. Isso significa que você frequentemente gera sentenças (ao menos frases longas) que foram vistas literalmente nos dados originais. Mais dados ajudariam; até funcionaria melhor se você coletasse *n*-grams de vários artigos sobre data science.

Gramáticas

Uma abordagem diferente para modelar linguagem é com *gramáticas*, regras para gerar sentenças aceitáveis. No ensino fundamental, você provavelmente aprendeu sobre partes do discurso e como combiná-las. Por exemplo, se você tinha um professor de inglês muito ruim, você pode dizer que uma sentença necessariamente consiste de um *substantivo* seguido de um *verbo*. Se, então, você tem uma lista de substantivos e verbos, você pode gerar sentenças de acordo com a regra.

Definiremos uma gramática um pouco mais complicada:

```
grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

Eu inventei a convenção de que nomes que comecem com sublinhados referem-se a *regras* que precisam de maior explicação, e que outros nomes são *terminais* que não precisam de mais processamento.

Então, por exemplo, "_S" é a regra de "sentença", que produz uma regra "_NP" ("frase nominal") seguida de uma regra "_VP" ("frase verbal").

A regra da frase verbal pode produzir a regra "_V" ("verbo") ou a regra verbo seguida da regra frase nominal.

Note que a regra "_NP" contém ela mesma em uma de suas produções. As gramáticas podem ser recursivas, o que permite que até gramáticas infinitas como esta gerem sentenças infinitamente diferentes.

Como geramos sentenças a partir desta gramática? Começaremos com uma lista contendo a regra sentença ["_S"]. E então expandiremos repetidamente cada regra substituindo-a por uma escolha aleatória de suas produções. Nós pararemos quando tivermos uma lista contendo apenas terminais.

Por exemplo, uma tal progressão pode parecer com:

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

Como implementamos isso? Bom, para começar, criaremos uma simples função auxiliar para identificar terminais:

```
def is_terminal(token):
    return token[0] != "_"
```

Em seguida, precisamos escrever uma função para transformar uma lista de símbolos em uma sentença. Procuraremos pelo primeiro símbolo não terminal. Se não conseguimos encontrar um, significa que completamos a sentença e terminamos.

Se encontrarmos um não terminal, escolhemos aleatoriamente uma de suas produções. Se essa produção é um terminal (por exemplo: uma palavra), nós simplesmente substituímos o símbolo por ela. Caso contrário, será uma sequência de símbolos não terminais separados por espaço que precisamos separar (`split`) e então encaixar em símbolos atuais. De qualquer forma, repetimos o processo no novo conjunto de símbolos.

Colocando tudo junto conseguimos:

```
def expand(grammar, tokens):
    for i, token in enumerate(tokens):

        # pula os terminais
        if is_terminal(token): continue

        # se chegamos aqui, encontramos um símbolo não terminal
        # então precisamos escolher um substituto aleatório
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
```

```
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # agora chama expand da nova lista de símbolos
        return expand(grammar, tokens)

    # se chegamos aqui, temos todos os terminais e acabamos
    return tokens
```

Agora podemos começar a gerar sentenças:

```
def generate_sentence(grammar):
    return expand(grammar, ["_S"])
```

Tente mudar a gramática — acrescente mais palavras, mais regras e suas próprias partes do discurso— até que você esteja pronto para gerar tantas páginas web quanto sua empresa precisa.

As gramáticas são mais interessantes quando usadas em outra direção. Dada uma sentença podemos usar uma gramática para *analisar* a sentença. Isso permite que identifiquemos sujeitos e verbos e nos ajuda a entender a sentença.

Usar data science para gerar texto é um truque esperto; usá-la para *entender* o texto é mais mágico. (Veja "Para Mais Esclarecimentos" na página 200 as bibliotecas que você poderia usar.)

Um Adendo: Amostragem de Gibbs

Gerar amostras de algumas distribuições é fácil. Nós podemos conseguir variáveis aleatórias uniformes com:

```
random.random()
```

e variáveis aleatórias normais com:

```
inverse_normal_cdf(random.random())
```

Mas algumas distribuições são mais difíceis de criar amostras. A *amostragem de Gibbs* é uma técnica para gerar amostras de distribuições multidimensionais quando apenas conhecemos algumas das distribuições condicionais.

Por exemplo, imagine que está jogando dois dados. Deixe x ser o valor do primeiro dado e y a soma dos dados, e imagine que você queria gerar muitos pares (x, y). Neste caso é fácil gerar amostras diretamente:

```
def roll_a_die():
    return random.choice([1,2,3,4,5,6])

def direct_sample():
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

Mas imagine que você só conhecia as distribuições condicionais. A distribuição de y condicionado a x é fácil — se você sabe o valor de x, y é igualmente possível de ser $x + 1$, $x + 2$, $x + 3$, $x + 4$, $x + 5$ ou $x + 6$:

```
def random_y_given_x(x):
    """igualmente possível de ser x + 1, x + 2, … , x + 6"""
    return x + roll_a_die()
```

A outra direção é mais complicada. Por exemplo, se você sabe que y é 2, então necessariamente x é 1 (pois a única forma de dois dados somarem 2 é se ambos forem 1). Se você sabe que y é 3, então x é igualmente possível de ser 1 ou 2. Similarmente, se y é 11, então x tem que ser 5 ou 6:

```
def random_x_given_y(y):
    if y <= 7:
        # se o total é 7 ou menos, o primeiro dado é igualmente
        # possível de ser 1, 2, ..., (total - 1)
        return random.randrange(1, y)
    else:
        # se o total é 7 ou mais, o primeiro dado é igualmente
        # possível de ser (total - 6), (total - 5), …, 6
        return random.randrange(y - 6, 7)
```

A forma como a amostragem de Gibbs funciona é que se começamos com qualquer valor (válido) para x e y e então repetida e alternadamente substituímos x por um valor aleatório escolhido condicionado a y e substituímos y por um valor aleatório escolhido condicionado a x. Após um número de iterações, os valores de x e y representarão uma amostra de uma distribuição conjunta incondicional:

```
def gibbs_sample(num_iters=100):
    x, y = 1, 2 # não importa
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

Você pode verificar que isso fornece resultados parecidos aos da amostra direta:

```
def compare_distributions(num_samples=1000):
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

Usaremos essa técnica na próxima seção.

Modelagem de Tópicos

Quando construímos nossa recomendação Cientistas de Dados que Você Deveria Conhecer no Capítulo 1, nós simplesmente procuramos combinações exatas dos interesses declarados das pessoas.

Uma abordagem mais sofisticada para entender como os interesses dos nossos usuários pode ser tentar identificar os *tópicos* que sustentam esses tópicos. Uma técnica chamada *Análise Latente de Dirichlet* (Latent Dirichlet's Analysis – LDA) é comumente usada para identificar tópicos comuns em uma coleção de documentos. Aplicaremos isso a documentos que consistem em interesses de cada usuário.

A LDA possui algumas semelhanças com o Classificador Naive Bayes que construímos no Capítulo 13, no que assume um modelo probabilístico para documentos. Nós maquiaremos os detalhes matemáticos pesados, mas para nossos objetivos o modelo presume que:

- Há um número fixo K de tópicos.
- Existe uma variável aleatória que atribui a cada tópico uma probabilidade de distribuição associada às palavras. Você deveria pensar nessa distribuição como a probabilidade de ver a palavra w dado o tópico k.
- Ainda há outra variável aleatória que atribui a cada documento a probabilidade de distribuição de tópicos. Você deveria pensar nessa distribuição como uma mistura de tópicos no documento d.
- Cada palavra em um documento foi gerada primeiro pela escolha aleatória de um tópico (da distribuição de tópicos do documento) e então uma palavra (da distribuição de palavras dos tópicos).

Nós temos uma coleção de `documents` em que cada uma é uma `list` de palavras. E temos uma coleção correspondente de `document_topics` que atribui um tópico (um número entre 0 e $K - 1$) a cada palavra em cada documento.

Para que a quinta palavra no quarto documento seja:

```
documents[3][4]
```

e o tópico de onde aquela palavra foi escolhida:

```
document_topics[3][4]
```

Isso explicitamente define a distribuição de tópicos de cada documento e implicitamente define a distribuição de palavras de cada tópico.

Nós podemos estimar a probabilidade de o tópico 1 produzir uma certa palavra comparando quantas vezes o tópico 1 produz *qualquer* palavra. Da mesma forma, quando construímos um filtro para spam no Capítulo 13, comparamos quantas vezes uma palavra aparecia em spams com o número total de palavras que aparecem em spams.

Apesar de esses tópicos serem apenas números, podemos dar a eles nomes descritivos olhando para as palavras nas quais eles colocam mais peso. Nós só precisamos gerar de alguma forma o `document_topics`. É aqui que a amostragem de Gibbs entra em ação.

Nós começamos atribuindo a cada palavra em cada documento um tópico completamente aleatório. Então passamos por cada documento, uma palavra por vez. Para aquela palavra e documento, nós construímos pesos para cada tópico que dependem da distribuição (atual) de tópicos naquele documento e da distribuição (atual) de palavras para aquele tópico. Então, usamos tais pesos para amostrar um novo tópico para aquela palavra. Se iterarmos esse processo muitas vezes, acabaremos com uma amostra conjunta da distribuição tópico-palavra e da distribuição documento-tópico.

Para começar, precisaremos de uma função para escolher aleatoriamente um índice baseado no conjunto arbitrário de pesos:

```
def sample_from(weights):
    """retorna i com probabilidade de weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()       # uniforme entre 0 e total
    for i, w in enumerate(weights):
        rnd -= w                        # retorna o menor i tal que
        if rnd <= 0: return i           # weights [0] + _ + weights[i] >= rnd
```

Por exemplo, se você dá pesos `[1, 1, 3]` então um quinto do tempo ele retornará 0, um quinto do tempo ele retornará 1 e três quintos do tempo retornará 2.

Nossos documentos são os interesses de nossos usuários, que parecem com:

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

E tentaremos encontrar `K = 4` tópicos.

Para calcular os pesos da amostragem, precisaremos acompanhar várias contagens. Primeiro vamos criar as estruturas de dados para elas.

Quantas vezes cada tópico é atribuído a cada documento:

```
# uma lista de Contadores, uma para cada documento
document_topic_counts = [Counter() for _ in documents]
```

Quantas vezes cada palavra é atribuída para cada tópico:

```
# uma lista de Contadores, uma para cada tópico
topic_word_counts = [Counter() for _ in range(K)]
```

O número total de palavras atribuídas a cada tópico:

```
# uma lista de números, uma para cada tópico
topic_counts = [0 for _ in range(K)]
```

O número total de palavras contidas em cada documento:

```
# uma lista de números, uma para cada documento
document_lengths = map(len, documents)
```

O número de palavras distintas:

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

E o número de documentos:

```
D = len(documents)
```

Por exemplo, uma vez que populamos estes, podemos encontrar o número de palavras em `documents[3]` associado com o tópico 1 como:

```
document_topic_counts[3][1]
```

E podemos encontrar o número de vezes que *nlp* é associado com o tópico 2 como:

```
topic_word_counts[2]["nlp"]
```

Agora estamos prontos para definir nossas funções de probabilidade condicional. Como no Capítulo 13, cada uma possui um termo suavizador que garante que todo tópico possua uma chance diferente de zero de ser escolhido em qualquer documento e que cada palavra possua uma chance diferente de zero de ser escolhida em qualquer tópico:

```
def p_topic_given_document(topic, d, alpha=0.1):
    """a fração de palavras no documento _d_
    que são atribuídas a _topic_ (mais alguma coisa)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """a fração de palavras atribuídas a _topic_
    que é igual a _ word _ (mais alguma coisa)"""

    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

Usaremos estes para criar pesos para atualizar os tópicos:

```
def topic_weight(d, word, k):
    """dado um documento e uma palavra naquele documento,
    retorne o peso para o k-ésimo tópico"""
```

```
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

Existem sólidos motivos matemáticos para o porquê de `topic_weight` ter sido definido de tal forma, mas seus detalhes nos levariam muito fora do escopo. Felizmente, faz sentido que — dada uma palavra e seu documento — a probabilidade de escolha de qualquer tópico dependa de quão provável aquele tópico é para o documento e quão provável aquela palavra é para o tópico.

Isso é todo o maquinário do qual precisamos. Começamos atribuindo cada palavra a um tópico aleatório e populando nossos contadores apropriadamente:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Nosso objetivo é conseguir uma amostra auxiliar da distribuição tópicos-palavras e da distribuição documentos-tópicos. Nós fazemos isso usando uma forma de amostragem de Gibbs que usa probabilidades condicionais definidas previamente:

```
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # remova esta palavra / tópico da contagem
            # para que não influencie nos pesos
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # escolha um novo tópico baseado nos pesos
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # e agora coloque-o de volta nas contas
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

O que são os tópicos? Eles são apenas números 0, 1, 2, e 3. Se quisermos nomes para eles temos que fazer isso nós mesmos. Vamos ver as palavras com maiores pesos para cada tópico (Tabela 20-1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print k, word, count
```

Tabela 20-1. Palavras mais comuns por tópico

Tópico 0	Tópico 1	Tópico 2	Tópico 3
Java	R	Hbase	regressão
Big Data	estatística	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probabilidade	Cassandra	aprendizado de máquina
inteligência artificial	pandas	NoSQL	redes neurais

Baseado nisso eu provavelmente atribuiria nomes de tópicos:

```
topic_names = ["Big Data e linguagens de programação",
               "Python e estatística",
               "bases de dados",
               "aprendizado de máquina"]
```

agora podemos ver como o modelo atribui tópicos aos interesses de cada usuário:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print document
    for topic, count in topic_counts.most_common():
        if count > 0:
            print topic_names[topic], count,
    print
```

que nos dá:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data e linguagem de programação 4 bases de dados 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
bases de dados 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python e estatística 5 aprendizado de máquina 1
```

e assim por diante. Dados os "e" que precisamos em alguns de nossos nomes de tópicos, é possível que usemos mais tópicos, embora seja provável que não tenhamos dados o suficiente para aprendê-los com sucesso.

Para Mais Esclarecimentos

- Natural Language Toolkit (*http://www.nltk.org/*) é uma biblioteca popular de ferramentas NLP para Python. Ela possui seu próprio livro completo (*http://www.nltk.org/book/*), que está disponível para leitura online.
- gensim (*http://radimrehurek.com/gensim/*) é uma biblioteca Python para modelagem de tópicos, que é uma aposta melhor do que nosso modelo a partir do zero.

CAPÍTULO 21

Análise de Rede

Suas conexões a todas as coisas ao seu redor, literalmente, definem quem você é.
—Aaron O'Connell

Muitos problemas de dados interessantes podem ser lucrativos pensando em termos de *redes*, consistentes de *nós* de algum tipo e *vínculos* que as juntam.

Por exemplos, seus amigos do Facebook formam os nós de uma rede cujos vínculos são relações de amizade. Um exemplo menos óbvio é a própria World Wide Web, com cada página sendo um nó e cada hiperlink de uma página para outra um vínculo.

Amizade de Facebook é mútua — se eu sou seu amigo no Facebook, você necessariamente é meu amigo. Nesse caso, dizemos que os vínculos são *não direcionados*. Hiperlinks não são — meu website possui links para whitehouse.gov, mas (por razões inexplicáveis) whitehouse.gov se recusa a ter link para o meu website. Nós chamamos esses tipos de vínculos de *direcionados*. Veremos os dois tipos de redes.

Centralidade de Intermediação

No Capítulo 1, nós computamos os conectores chave na rede DataSciencester contando o número de amigos que cada usuário tinha. Agora nós temos maquinário o suficiente para ver outras abordagens. Lembre-se de que a rede (Figura 21-1) englobava usuários:

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
```

```
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

e amizades:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

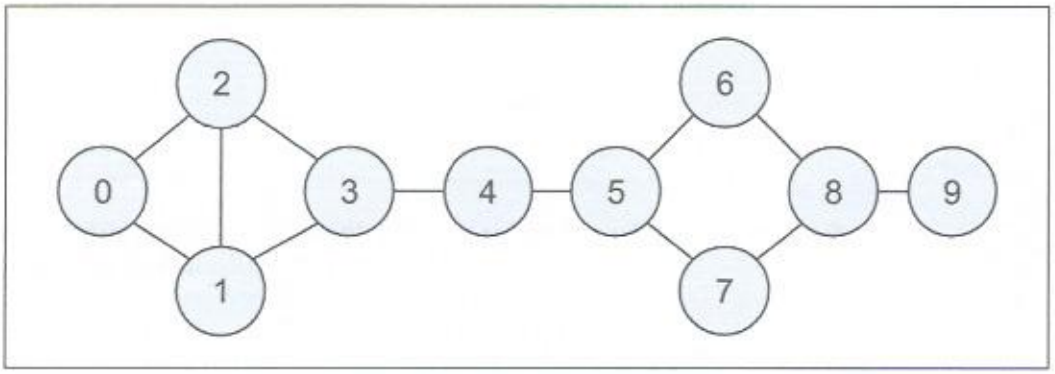

Figura 21-1. A rede DataSciencester

Nós também adicionamos listas de amigos para cada `dict` de usuário:

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # isso funciona porque users[i] é o usuário cuja id é i
    users[i]["friends"].append(users[j]) # adiciona i como amigo de j
    users[j]["friends"].append(users[i]) # adiciona j como amigo de i
```

Quando começamos, estávamos insatisfeitos com nossa noção de *grau de centralidade*, que não concordava com nossa intuição sobre quem eram os conectores-chave da rede.

Uma alternativa métrica é *centralidade de intermediação*, que identifica pessoas que frequentemente estão no menor caminho entre pares de outras pessoas. A centralidade de intermediação de nó *i* é computada adicionando para cada outro par de nós j e k, a proporção dos caminhos mais curtos entre o nó j e o nó k que passa por i.

Isto é, para entender a centralidade de intermediação de Thor, precisamos computar todos os caminhos mais curtos entre todos os pares de pessoas que não são Thor. E então precisaremos contar quantos desses caminhos mais curtos passam por Thor. Por exemplo, o único caminho mais curto entre Chi `(id 3)` e Clive `(id 5)` passa por Thor, enquanto nenhum dos dois caminhos mais curtos entre Hero `(id 0)` e Chi `(id 3)` passa.

Portanto, como primeiro passo, precisaremos descobrir os caminhos mais curtos entre todos os pares de pessoas. Existem alguns algoritmos bem sofisticados para fazer isso eficientemente, mas (como quase sempre acontece) usaremos um menos eficiente e mais fácil de entender.

Este algoritmo (uma implementação de busca em largura) é um dos mais complicados neste livro, então falaremos cuidadosamente sobre ele:

1. Nosso objetivo é uma função que pega `from_user` e encontra *todos* os caminhos mais curtos para todos os outros usuários.
2. Representaremos um caminho como uma `list` de IDs de usuários. Como todo caminho começa em `from_user`, não incluiremos seu ID na lista. Isso significa que o tamanho da lista representando o caminho será o próprio tamanho do caminho.
3. Manteremos um dicionário `shortest_paths_to` onde as chaves são IDs de usuários e os valores são listas de caminhos que terminam no usuário com o ID especificado. Se existe um único menor caminho, a lista conterá apenas aquele caminho. Se existem múltiplos caminhos menores, a lista conterá todos eles.
4. Também manteremos uma fila `frontier` que contém usuários que queremos explorar na ordem em que queremos explorá-los. Os armazenaremos em pares `(prev_user, user)` para que saibamos como chegamos a cada um deles. Inicializaremos a lista com todos os vizinhos de `from_user`. (Ainda não falamos sobre filas, as quais são estruturas otimizadas de operações "adicione ao final" e "remova do começo". Em Python, elas são implementadas como `collections.deque` que na verdade é uma fila duplamente terminada.)
5. À medida que exploramos o gráfico, sempre que encontramos novos vizinhos para os quais ainda não conhecemos os caminhos mais curtos, nós os adicionamos ao final da fila para explorarmos mais tarde, com o usuário atual `prev_user`.
6. Quando tiramos um usuário da fila que nunca encontramos antes, nós, com certeza, encontramos um ou mais caminhos mais curtos para ele — cada menor caminho para `prev_user` com um passo extra.
7. Quando tiramos um usuário da fila que *já* havíamos encontrado aquele usuário antes, então ou encontramos outro menor caminho (em, cujo caso, deveremos adicionar o usuário) ou encontramos um caminho maior (em, cujo caso, não deveríamos adicionar o usuário).
8. Quando não há mais usuários na fila, exploramos todo o gráfico (ou, ao menos, as partes possíveis de serem analisadas a partir do usuário com quem iniciamos) e terminamos.

Podemos colocar tudo junto em uma (grande) função:

```
from collections import deque

def shortest_paths_from(from_user):

    # um dicionário para "user_id" para *todos* os caminhos
    # mais curtos para aquele usuário
    shortest_paths_to = { from_user["id"] : [[]] }

    # uma fila de (previous user, next user) que precisamos verificar.
    # começa com todos os pares (from_user, friend_of_from_user)
```

```
    frontier = deque((from_user, friend)
                     for friend in from_user["friends"])

    # continue até esvaziar a fila
    while frontier:

        prev_user, user = frontier.popleft()   # remova o usuário que é
        user_id = user["id"]                   # o primeiro na fila

        # pela maneira como estamos adicionando na fila,
        # necessariamente já conhecemos alguns dos caminhos mais curtos para prev_user
        paths_to_prev_user = shortest_paths_to[prev_user["id"]]
        new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

        # é possível que já saibamos um menor caminho
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # qual é o menor caminho até aqui que já vimos até agora?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')

        # apenas mantém caminhos que não são longos demais e são novos
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # add never-seen neighbors to the frontier
        frontier.extend((user, friend)
                        for friend in user["friends"]
                        if friend["id"] not in shortest_paths_to)

    return shortest_paths_to
```

Agora podemos armazenar esses `dicts` com cada nó:

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

E finalmente estamos prontos para computar a centralidade de intermediação. Para todo par de nós *i* e *j*, conhecemos os *n* caminhos mais curtos para *i* e *j*. Então, para cada um desses caminhos, apenas adicionamos $1/n$ à centralidade de cada nó naquele caminho:

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():
```

```
        if source_id < target_id:        # não conta duas vezes
            num_paths = len(paths)       # quantos caminhos mais curtos?
            contrib = 1 / num_paths      # contribuição para centralidade
            for path in paths:
                for id in path:
                    if id not in [source_id, target_id]:
                        users[id]["betweenness_centrality"] += contrib
```

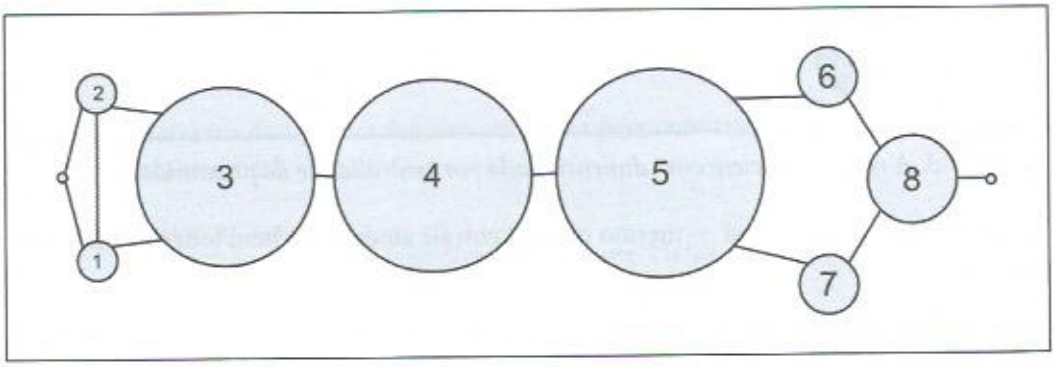

Figura 21-2. A rede DataSciencester dimensionada por centralidade de intermediação

Como exibido na Figura 21-2, os usuários 0 e 9 têm centralidade 0 (porque nenhum deles está em nenhum caminho menor entre outros usuários), mas 3, 4 e 5 têm altas centralidades (porque todos os três estão em muitos caminhos menores).

Geralmente, os números de centralidade não são significativos. O que importa é como os números para cada nó se comparam com os números dos outros nós.

Outra medida que podemos ver é *centralidade de proximidade*. Primeiro, para cada usuário, nós computamos sua *distância*, que é a soma dos tamanhos de seus caminhos mais curtos para cada outro usuário. Como já computamos os caminhos mais curtos entre cada par de nós, é fácil calcular seus tamanhos. (Se houver múltiplos caminhos curtos, todos eles possuem o mesmo tamanho, logo podemos analisar o primeiro.)

```
def farness(user):
    """a soma dos tamanhos dos caminhos mais curtos para cada outro usuário"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

após isso temos pouquíssimo trabalho para computar a centralidade de proximidade (Figura 21-3):

```
for user in users:
    user["closeness_centrality"] = 1 / farness(user)
```

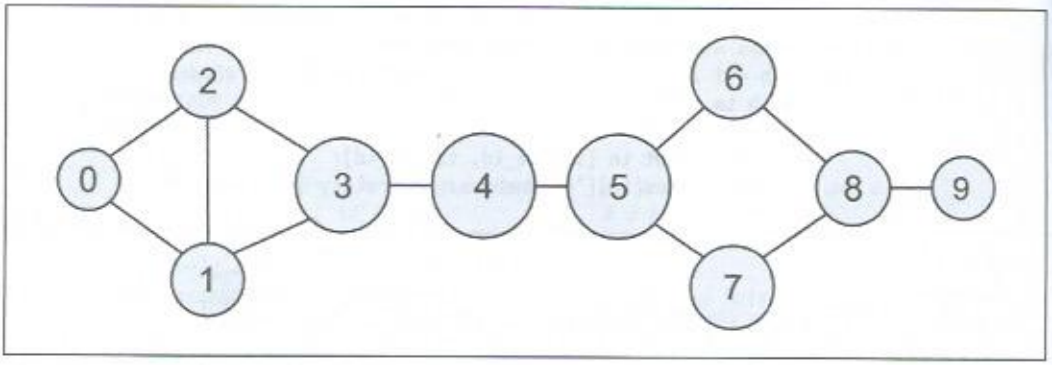

Figura 21-3. A rede DataSciencester dimensionada por centralidade de proximidade

Há bem menos variação aqui — mesmo os nós centrais ainda estão bem longe dos nós nos arredores.

Como vimos, computar os caminhos mais curtos dá uma certa chateação. Por isso, centralidades de intermediação e de proximidade não são usadas com frequência em grandes redes. A centralidade menos intuitiva (mas mais fácil de computar), *centralidade de vetor próprio* é usada com mais frequência.

Centralidade de Vetor Próprio

Para falar sobre centralidade de vetor próprio, temos que falar sobre vetores próprios, e para falar sobre vetores próprios, temos que falar sobre multiplicação de matrizes.

Multiplicação de Matrizes

Se A é uma matriz n_1 x k_1 e B é uma matriz $n_2 \times k_2$, e se $k_1 = n_2$ então o produto AB deles é a matriz $n_1 \times k_2$ cuja entrada (i,j) é:

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{ik}B_{kj}$$

Que é apenas o produto `dot` da i-ésima linha de A com a j-ésima coluna de B *(também chamados de vetores)*:

```
def matrix_product_entry(A, B, i, j):
    return dot(get_row(A, i), get_column(B, j))
```

depois do quê, temos:

```
def matrix_multiply(A, B):
    n1, k1 = shape(A)
    n2, k2 = shape(B)
    if k1 != n2:
        raise ArithmeticError("formatos incompatíveis!")

    return make_matrix(n1, k2, partial(matrix_product_entry, A, B))
```

Note que, se A é uma matriz $n \times k$ e B é uma matriz $k \times 1$, então AB é uma matriz $n \times 1$. Se tratamos um vetor como uma matriz de uma coluna, podemos pensar em A como uma função que mapeia vetores dimensionais k para vetores dimensionais n, em que a função é apenas a multiplicação da matriz.

Anteriormente, representamos vetores apenas como listas, mas não são a mesma coisa:

```
v = [1, 2, 3]
v_as_matrix = [[1],
               [2],
               [3]]
```

Então precisaremos de algumas funções auxiliares para converter de um lado para o outro as duas representações:

```
def vector_as_matrix(v):
    """retorna o vetor v (representado como uma lista) como uma matriz n × 1"""
    return [[v_i] for v_i in v]

def vector_from_matrix(v_as_matrix):
    """retorna a matriz n x 1 como uma lista de valores"""
    return [row[0] for row in v_as_matrix]
```

e após podemos definir a operação de matriz usando `matrix_multiply`:

```
def matrix_operate(A, v):
    v_as_matrix = vector_as_matrix(v)
    product = matrix_multiply(A, v_as_matrix)
    return vector_from_matrix(product)
```

Quando A é uma matriz *quadrada*, esta operação mapeia vetores dimensionais n para outros vetores dimensionais n. É possível que, para alguma matriz A e vetor v, quando A opera em v conseguimos um múltiplo de v. Isso é, o resultado é um vetor que aponta para a mesma direção que v. Quando isso acontece (e quando, além disso, v não é um vetor de zeros), nós chamamos v de *vetor próprio (autovetor)* de A. E chamamos o multiplicador de *valor próprio (autovalor)*.

Uma possível maneira de encontrar o vetor próprio de A é escolher um vetor inicial v, aplicar `matrix_operate`, reescalar o resultado para ter magnitude 1 e repetir até o processo convergir:

```
def find_eigenvector(A, tolerance=0.00001):
    guess = [random.random() for __ in A]

    while True:
        result = matrix_operate(A, guess)
        length = magnitude(result)
        next_guess = scalar_multiply(1/length, result)

        if distance(guess, next_guess) < tolerance:
            return next_guess, length   # vetor próprio, valor próprio
```

```
        guess = next_guess
```

Por construção, o `guess` retornado é um vetor tal que, quando você aplica `matrix_operate` a ele e o reescala para ter tamanho 1, você recebe de volta (um vetor muito próximo a) ele mesmo. O que significa que ele é um vetor próprio.

Nem todas as matrizes de números reais possuem vetores e valores próprios. Por exemplo, a matriz:

```
rotate = [[ 0, 1],
          [-1, 0]]
```

gira vetores a 90 graus no sentido horário, o que significa que o único vetor que ela mapeia para um múltiplo em escala de si mesmo é um vetor de zeros. Se você tentou `find_eigenvector(rotate)` ele poderia rodar para sempre. Até mesmo matrizes com vetores próprios podem, às vezes, ficar presas em ciclos. Considere a matriz:

```
flip = [[0, 1],
        [1, 0]]
```

Essa matriz mapeia qualquer vetor `[x, y]` para `[y, x]`. Isso significa que, por exemplo, `[1, 1]` é um vetor próprio com valor próprio 1. Entretanto, se você começa com um vetor aleatório com coordenadas deferentes, `find_eigenvector` irá repetidamente trocar as coordenadas para sempre. Bibliotecas que não são do zero como NumPy usam métodos diferentes que funcionariam nesse caso. Todavia, quando `find_eigenvector` retorna um resultado, tal resultado é sem dúvidas um vetor próprio.

Centralidade

Como isso nos ajuda a entender a rede DataSciencester?

Para começar, precisaremos representar as conexões em nossa rede como `adjacency_matrix`, cuja entrada (i,j)th ou é 1 (se usuários *i* e *j* forem amigos) ou 0 (se não forem):

```
def entry_fn(i, j):
    return 1 if (i, j) in friendships or (j, i) in friendships else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

A centralidade de vetor próprio para cada usuário é a entrada correspondente àquele usuário no vetor próprio retornado por `find_eigenvector` (Figura 21-4):

Por motivos técnicos que estão além do escopo deste livro, qualquer matriz adjacente diferente de zero necessariamente possui um vetor próprio para todas aquelas cujos valores não são negativos. E felizmente para nós, para essa `adjacency_matrix` nossa função `find_eigenvector` o encontra.

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

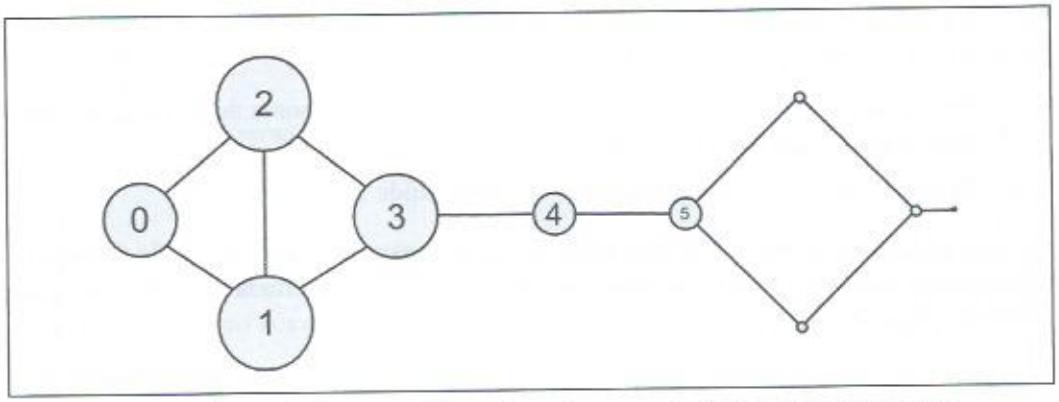

Figura 21-4. A rede DataSciencester dimensionada por centralidade de vetor próprio

Usuários com alta centralidade de vetor próprio deveriam ser aqueles que têm muitas conexões e conexões com pessoas que têm alta centralidade.

Aqui os usuários 1 e 2 são os mais centrais, pois ambos possuem três conexões com pessoas que são altamente centrais. Conforme nos afastamos deles, as centralidades das pessoas gradualmente diminuem.

Em uma rede pequena assim, a centralidade do vetor próprio comporta-se de forma instável. Se você tentar adicionar ou remover links, descobrirá que pequenas modificações na rede podem mudar dramaticamente os números de centralidade. Em uma rede maior esse não seria o caso.

Nós ainda não chegamos no porquê de um vetor próprio poder levar a uma noção razoável de centralidade. Ser um vetor próprio significa que se você computa:

```
matrix_operate(adjacency_matrix, eigenvector_centralities)
```

o resultado é um múltiplo escalar de `eigen_vector_centralities`.

Se você olhar como a multiplicação de matriz funciona, `matrix_operate` produz um vetor cujo i-ésimo elemento é:

```
dot(get_row(adjacency_matrix, i), eigenvector_centralities)
```

que é precisamente a soma das centralidades de vetor próprio de outros usuários conectados ao usuário *i*.

Em outras palavras, as centralidades de vetor próprio são números, um por usuário, tal que cada valor de usuário é um múltiplo constante da soma dos valores de seus vizinhos. Neste caso, centralidade significa estar conectado a pessoas que são elas mesmas centrais. Quanto mais você estiver diretamente conectado a nós com centralidade, mais central você é. Isso é uma definição circular—vetores próprios são uma forma de sair do círculo.

Outra maneira de entender isso é pensar no quê `find_eigenvector` está fazendo aqui. Ele começa atribuindo cada nó a uma centralidade aleatória. Então repete os dois passos a seguir até o processo convergir:

1. Dá a cada nó um novo valor de centralidade que é igual a soma dos valores das centralidades (antigas) de seus vizinhos.
2. Reescala o vetor de centralidades para ter magnitude 1.

Embora a matemática por trás disso pareça um tanto opaca no início, o cálculo é relativamente direto (diferente de centralidade por intermediação) e é muito fácil de fazer em gráficos muito grandes.

Gráficos Direcionados e PageRank

DataSciencester não está conseguindo deslanchar, então a vice-presidente de Rendimentos considera trocar de um modelo de amizade para um modelo de aprovação. Acontece que ninguém particularmente se importa quais cientistas de dados são *amigos* um do outro, mas recrutadores se importam muito com quais cientistas de dados são respeitados por outros cientistas.

Nesse novo modelo, acompanharemos as aprovações `(source, target)` que não representam mais um relacionamento recíproco, mas sim que `source` aprove `target` como um excelente cientista de dados (Figura 21-5). Precisaremos considerar esta assimetria:

```
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]

for user in users:
    user["endorses"] = []        # adiciona uma lista para acompanhar aprovações dadas
    user["endorsed_by"] = []     # adiciona outra para companhar aprovações recebidas

for source_id, target_id in endorsements:
    users[source_id]["endorses"].append(users[target_id])
    users[target_id]["endorsed_by"].append(users[source_id])
```

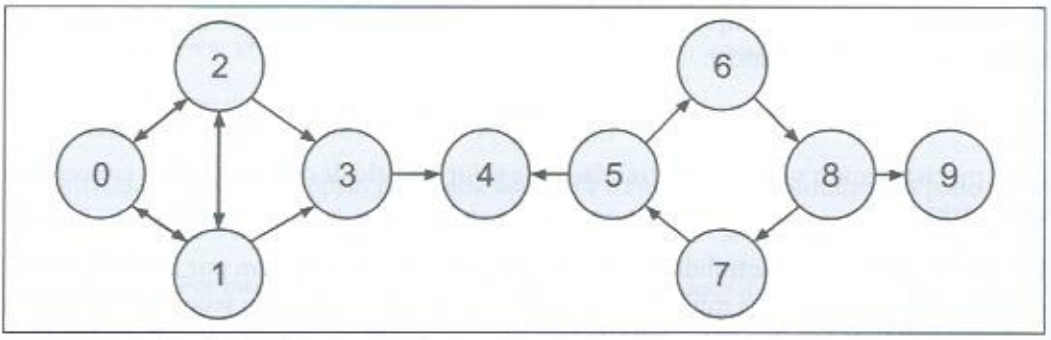

Figura 21-5. A rede DataSciencester de aprovações

após o que podemos facilmente encontrar os cientistas de dados mais aprovados (`most_endorsed`) e vender toda essa informação para os recrutadores:

```
endorsements_by_id = [(user["id"], len(user["endorsed_by"]))
                      for user in users]

sorted(endorsements_by_id,
       key=lambda (user_id, num_endorsements): num_endorsements,
       reverse=True)
```

Entretanto, "número de aprovações" é uma métrica fácil de manipular. Tudo que você precisa fazer é criar contas falsas e fazer com que elas o aprovem. Ou combinar com seus amigos para aprovarem uns aos outros (o que parece que os usuários 0, 1 e 2 fizeram).

Uma métrica melhor para levar em consideração é *quem* aprova você. Aprovações de pessoas com muitas aprovações deveriam de alguma forma contar mais do que aprovações de pessoas com poucas aprovações. Essa é a essência do algoritmo PageRank, usado pelo Google para ordenar web sites baseados em quais outros sites estão conectados a eles, quais a estes e por aí vai.

(Se isso mais ou menos o lembra da ideia por trás da centralidade de vetor próprio, você está certo.)

Uma versão simplificada se parece com isto:

1. Há um total de 1.0 (ou 100%) de PageRank na rede.
2. Inicialmente este PageRank é igualmente distribuído pelos nós.
3. A cada passo, uma grande fração do PageRank do nó é distribuída igualmente entre os seus links de saída.
4. A cada passo, o resto de PageRank do nó é distribuído igualmente entre todos os nós.

```
def page_rank(users, damping = 0.85, num_iters = 100):

    # inicialmente distribui PageRank igualmente
    num_users = len(users)
    pr = { user["id"] : 1 / num_users for user in users }

    # esta é a pequena fração de PageRank
    # que cada nó recebe a cada iteração
    base_pr = (1 - damping) / num_users

    for __ in range(num_iters):
        next_pr = { user["id"] : base_pr for user in users }
        for user in users:
            # distribui PageRank para links de saída
            links_pr = pr[user["id"]] * damping
            for endorsee in user["endorses"]:
                next_pr[endorsee["id"]] += links_pr / len(user["endorses"])
```

```
        pr = next_pr

    return pr
```

PageRank (Figura 21-6) identifica o usuário 4 (Thor) como o cientista de dados com mais classificações.

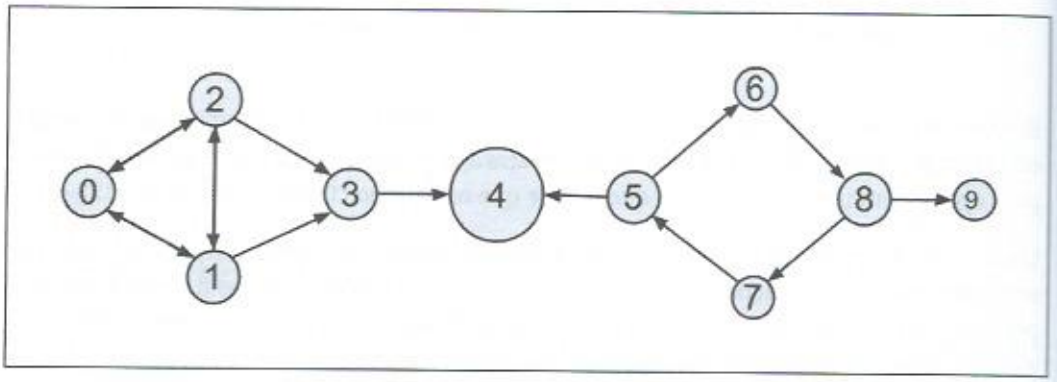

Figura 21-6. A rede DataSciencester dimensionada por PageRank

Mesmo que ele tenha menos aprovações (2) do que usuários 0, 1 e 2, as aprovações dele carregam classificações de suas aprovações. Além disso, ambos apoiadores apoiaram apenas ele, o que significa que ele não precisou dividir a classificação com mais ninguém.

Para Mais Esclarecimentos

- Há muitas outras noções de centralidade (*http://en.wikipedia.org/wiki/Centrality*) além das que usamos (mas essas são as mais populares).
- NetworkX (*http://networkx.github.io/*) é uma biblioteca Python para análise de rede. Ela possui funções para computar centralidades e visualizar gráficos.
- Gephi (*http://gephi.github.io/*) é uma ferramenta de visualização amada/odiada de rede baseada na ferramenta de visualização GUI.

CAPÍTULO 22

Sistemas Recomendadores

Oh natureza, natureza, por que és tão desonesta, para enviares homens com falsas recomendações para o mundo!
—Henry Fielding

Outro problema de dados comum é produzir algum tipo de *recomendação*. A Netflix recomenda filmes que você poderia querer assistir. A Amazon recomenda produtos que você poderia querer comprar. O Twitter recomenda usuários para você seguir. Neste capítulo, veremos várias formas de usar dados para fazer recomendações.

Em particular, veremos o conjunto de dados de `users_interests` que usamos anteriormente:

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

E pensaremos sobre o problema de recomendar novos interesses para um usuário baseado em seus interesses atuais específicos.

Curadoria Manual

Antes da Internet, quando você precisava de recomendações de livros, você ia até a biblioteca onde um bibliotecário estava disponível para sugerir livros que fossem relevantes a seus interesses ou similares a livros que você gostou.

Dado um número limitado de usuários e interesses da DataSciencester, seria fácil para você passar uma tarde recomendando interesses para cada usuário. Mas esse método não escala particularmente bem e é limitado por seu conhecimento pessoal e imaginação. (Não que eu esteja sugerindo que seu conhecimento pessoal e imaginação sejam limitados.) Então vamos pensar sobre o que podemos fazer com *dados*.

Recomendando O Que é Popular

Uma abordagem fácil é simplesmente recomendar o que é popular:

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

que se parece com:

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # ...
]
```

Tendo computado isso, podemos apenas sugerir a um usuário os interesses mais populares pelos quais ele ainda não está interessado:

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

Então, se você é o usuário 1, com interesses:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

recomendaríamos a você:

```
most_popular_new_interests(users_interests[1], 5)

# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

Se você for o usuário 3, que já está interessado em muitas dessas coisas, você receberia:

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

Claro, "muitas pessoas são interessadas em Python então você também deveria ser" não é uma frase de venda muito persuasiva. Se alguém é novo em nosso site e não sabemos nada sobre ele, isso é basicamente o melhor que podemos fazer. Veremos como podemos melhorar baseando as recomendações de cada usuário em seus interesses.

Filtragem Colaborativa Baseada no Usuário

Uma forma de levar em consideração os interesses do usuário é procurar usuários que são *similares* a ele e então sugerir as coisas nas quais aqueles usuários são interessados.

Para fazer isso, precisaremos de uma maneira de medir quão similares os dois usuários são. Aqui usaremos uma métrica chamada *similaridade do cosseno*. Dados dois vetores, `v` e `w`, é definida como:

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

Ela mede o "ângulo" entre `v` e `w`. Se `v` e `w` apontam para a mesma direção, então o numerador e o denominador são iguais e sua similaridade do cosseno é igual a 1. Se `v` e `w` apontam para direções opostas, sua similaridade do cosseno é igual a –1. E se `v` é 0 sempre que `w` não é (e vice-versa) então `dot(v, w)` é 0 e sua similaridade do cosseno será 0.

Quando aplicamos isso a vetores de 0s e 1s, cada vetor `v` representando os interesses de um usuário. `v[i]` será 1 se o usuário é especificado com o i-ésimo interesse, e 0 se não. Da mesma forma, "usuários similares" significará "usuários cujos vetores de interesses quase apontam para a mesma direção". Usuários com interesses idênticos terão similaridade 1. Usuários com interesses não idênticos terão similaridade 0. Senão, a similaridade ficará no meio, com números próximos a 1 indicando "muito similar" e números próximos de 0 indicando "não muito similar".

Um bom lugar para começar é juntar os interesses conhecidos e (implicitamente) atribuir índices a eles. Podemos fazer isso usando um conjunto de compreensão para encontrar os interesses únicos, colocá-los em uma lista e ordená-los. O primeiro interesse na lista resultante será o interesse 0 e por aí vai:

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

Isso nos dá uma lista que começa:

```
['Big Data',
 'C++',
```

```
    'Cassandra',
    'HBase',
    'Hadoop',
    'Haskell',
    # ...
]
```

Em seguida, queremos produzir um vetor "interesse" de 0s e 1s para cada usuário. Nós só precisamos iterar pela lista `unique_interests`, substituindo um 1 se o usuário possui cada interesse, e 0 se não:

```
def make_user_interest_vector(user_interests):
    """dada uma lista de interesses, produza um vetor cujo i-ésimo elemento é 1
    se unique_interests[i] está na lista, 0"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

após isso, podemos criar uma matriz de interesses de usuário simplesmente mapeando (`map_ping`) esta função contra a lista de listas de interesses:

```
user_interest_matrix = map(make_user_interest_vector, users_interests)
```

Agora `user_interest_matrix[i][j]` é igual a 1 se o usuário `i` especificou o interesse `j`, e 0 se não.

Por termos um pequeno conjunto de dados, não é um problema computar as similaridades em pares entre todos os nossos usuários:

```
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_matrix]
                     for interest_vector_i in user_interest_matrix]
```

e depois, `user_similarities[i][j]` nos dá a similaridade entre *i* e *j*.

Por exemplo, `user_similarities[i]` é 0,57, porque aqueles dois usuários demonstram interesse em comum em Hadoop, Java e Big Data. Por outro lado, `user_similarities[0][8]` é somente 0,19, porque os usuários 0 e 8 demonstram apenas um interesse em comum: Big Data;

Especificamente, `user_similarities[i]` é um vetor de similaridades do usuário *i* para cada outro usuário. Podemos usar isso para escrever uma função que encontra os usuários mais similares a um usuário específico. Nos certificaremos de não incluir o próprio usuário, nem qualquer outro usuário com similaridade 0. E ordenaremos os resultados do mais similar para o menos:

```
def most_similar_users_to(user_id):
    pairs = [(other_user_id, similarity)                    # encontra outros
             for other_user_id, similarity in               # usuários com
                enumerate(user_similarities[user_id])       # similaridade
             if user_id != other_user_id and similarity > 0] # não zero

    return sorted(pairs,                                    # ordena
                  key=lambda (_, similarity): similarity,   # mais similares
                  reverse=True)                             # primeiro
```

Por exemplo, se chamarmos `most_similar_user(0)`, conseguimos:

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

Como usamos isso para sugerir novos interesses para um usuário? Para cada interesse, podemos apenas somar similaridades de usuário dos outros usuários interessados:

```
def user_based_suggestions(user_id, include_current_interests=False):
    # soma as similaridades
    suggestions = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # converte-as em uma lista ordenada
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, weight): weight,
                         reverse=True)

    # e (talvez) exclui interesses já existentes
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

Se chamarmos `user_based_suggestions(0)`, os primeiros interesses sugeridos são:

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]
```

Essas parecem ser sugestões decentes para alguém que está interessado em "Big Data" e assuntos relacionados a bancos de dados. (Os pesos não são significativos intrinsecamente; apenas os usamos para ordenar.)

Essa abordagem não funciona bem quando o número de itens fica muito grande. Lembre da maldição da dimensionalidade do Capítulo 12 — em espaços de vetor de grandes dimensões a maioria dos vetores estão distantes (e logo, apontam para direções diferentes). Isto é, quando há um grande número de interesses, os "usuários mais similares" a um usuário específico podem não ser similares no final.

Imagine um site como o da Amazon, em que compramos milhares de itens nas últimas duas décadas. Você poderia tentar identificar usuários similares a mim baseado em padrões de compras, mas é bem provável que não exista ninguém no mundo que compre roupas históricas como as minhas. Quem quer que seja meu comprador "mais similar", provavelmente não é similar a mim e sua compras quase certamente dariam recomendações idiotas para mim.

Filtragem Colaborativa Baseada em Itens

Uma abordagem alternativa é computar similaridades entres interesses diretamente. Nós podemos gerar sugestões para cada usuário agregando interesses que são similares a seus interesses atuais.

Para começar, queremos *transpor* a matriz de interesse do nosso usuário para que fileiras correspondam a interesses e colunas correspondam a usuários:

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_matrix]
                        for j, _ in enumerate(unique_interests)]
```

Com o que isso se parece? A linha `j` de `interest_user_matrix` é coluna `j` de `user_interest_matrix`. Isto é, possui 1 para cada usuário com aquele interesse e 0 para cada usuário sem aquele interesse.

Por exemplo, `unique_interests[0]` é Big Data e `interest_user_matrix[0]` também é:

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

porque usuários 0, 8 e 9 indicaram interesse em Big Bata.

Agora podemos usar similaridade do cosseno novamente. Se, precisamente, os mesmos usuários estão interessados em dois tópicos, suas similaridades serão 1. Se nenhum dos dois usuários estiver interessado em ambos tópicos, a similaridade será 0:

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

Por exemplo, podemos encontrar os interesses mais similares a Big Data (interesse 0) usando:

```
def most_similar_interests_to(interest_id):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda (_, similarity): similarity,
                  reverse=True)
```

o que sugere os seguintes interesses similares:

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

Agora nós podemos criar recomendações para um usuário somando as similaridades de interesses parecidos com os dele:

```
def item_based_suggestions(user_id, include_current_interests=False):
    # soma interesses similares
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_matrix[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

    # ordena por peso
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, similarity): similarity,
                         reverse=True)

    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

Para o usuário 0, isso gera as seguintes (aparentemente razoáveis) recomendações:

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```

Para Mais Esclarecimentos

- Crab (*http://muricoca.github.io/crab/*) é um framework para construção de sistemas recomendadores em Python.
- Graphlab também possui ferramentas recomendadoras (*http://bit.ly/1MF9Tsy*).
- O Prêmio Netflix (*http://www.netflixprize.com*) foi uma competição famosa para construir um melhor sistema para recomendar filmes para usuários Netflix.

CAPÍTULO 23

Bases de Dados e SQL

A memória é a melhor amiga e a pior inimiga do homem.
— Gilbert Parker

Os dados dos quais você precisará geralmente estarão em *bases de dados*, usando sistemas feitos para armazenar e consultar dados eficientemente. A maior parte desses sistemas de bancos de dados são *relacionais*, como Oracle, MySQL e SQL Server, que armazenam dados em *tabelas* e são tipicamente consultados usando Structured Query Language (SQL), uma linguagem declarativa para manipular dados.

SQL é uma parte essencial do kit de ferramentas do cientista de dados. Neste capítulo, criaremos NotQuiteABase, uma implementação Python de algo que não é exatamente um banco de dados. Também cobriremos o básico de SQL enquanto mostramos como ele trabalham em nosso não exatamente banco de dados, que a forma mais "do zero" que pude pensar para ajudar a entender o que tais funcionalidades básicas estão fazendo. Minha esperança é que resolver problemas em NotQuiteABase trará uma boa noção de como você pode resolver os mesmos problemas usando SQL.

CREATE TABLE e INSERT

Uma base de dados relacional é uma coleção de tabelas (e de relacionamentos entre elas). Uma tabela é uma simples coleção de linhas, não diferente das matrizes com as quais trabalhamos. Entretanto, uma tabela também possui associação com um *esquema* fixo constituído por nomes e tipos de colunas.

Por exemplo, imagine um conjunto de dados `users` contendo para cada usuário seu `user_id`, `name` e `num_friends`:

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

Em SQL, poderíamos criar essa tabela com:

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

Note que especificamos que `user_id` e `num_friends` devem ser inteiros (e que `user_id` não pode ser `NULL`, o que indica um valor faltando e é como nosso `None`) e que nome deveria ser uma cadeia de caracteres de tamanho 200 ou menor. NotQuiteABase não leva tipos em consideração mas se comporta como se levasse.

SQL é quase completamente insensível a endentação e capitalização. O estilo de endentação e capitalização aqui é o meu favorito. Se você começa a aprender SQL, você certamente encontrará outros exemplos estilizados diferentemente.

Você pode inserir as linhas com declarações INSERT:

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Note também que declarações SQL precisam terminar com ponto de vírgula e que SQL requer aspa única para suas strings.

Em NotQuiteABase você criará uma `Table` simplesmente especificando os nomes de suas colunas. E para inserir uma linha, você usará o método `insert()` da tabela, que pega uma `list` de valores de linha que precisam estar na mesma ordem dos nomes da coluna da tabela.

Nos bastidores, armazenaremos cada linha como um `dict` (dicionário) de nomes de colunas para valores. Um banco de dados real jamais usaria tal representação mas fazê-la tornará NotQuiteABase mais fácil de trabalhar:

```
class Table:
    def __init__(self, columns):
        self.columns = columns
        self.rows = []

    def __repr__(self):
        """bela representação da tabela: colunas e então linhas"""
        return str(self.columns) + "\n" + "\n".join(map(str, self.rows))

    def insert(self, row_values):
        if len(row_values) != len(self.columns):
            raise TypeError("wrong number of elements")
        row_dict = dict(zip(self.columns, row_values))
        self.rows.append(row_dict)
```

Por exemplo, poderíamos configurar:

```
users = Table(["user_id", "name", "num_friends"])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

Se agora você imprimir `users`, verá:

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

UPDATE

Às vezes, você precisa atualizar os dados que já estão no banco de dados. Por exemplo, se Dunn possuir outro amigo, você pode precisar fazer isso:

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

Os atributos-chave são:

- Qual tabela atualizar
- Quais linhas atualizar
- Quais campos atualizar
- Quais deveriam ser os novos valores

Adicionaremos um método `update` similar em NotQuiteABase. Seu primeiro argumento será um `dict` cujas chaves são colunas para atualizar e cujos valores são os novos valores para aqueles campos. E seu segundo argumento é um predicado que retorna `True` para linhas que deveriam ser atualizadas, `False` se não:

```
def update(self, updates, predicate):
    for row in self.rows:
        if predicate(row):
            for column, new_value in updates.iteritems():
                row[column] = new_value
```

e após isso podemos simplesmente fazer:

```
users.update({'num_friends' : 3},                # estabelece num_friends = 3
             lambda row: row['user_id'] == 1)    # em linhas onde user_id == 1
```

DELETE

Há duas formas de excluir linhas de uma tabela em SQL. A forma perigosa apaga todas as linhas da tabela:

```
DELETE FROM users;
```

A forma menos perigosa adiciona uma cláusula `WHERE` e apenas apaga as colunas que atendem a uma certa condição:

```
DELETE FROM users WHERE user_id = 1;
```

É fácil acrescentar essa funcionalidade a nossa `Table`:

```
def delete(self, predicate=lambda row: True):
    """apaga todas as linhas que combinam com o predicado
    ou todas as linhas se não há predicado"""
    self.rows = [row for row in self.rows if not(predicate(row))]
```

Se você forneceu uma função `predicate` (por exemplo, cláusula `WHERE`), isso apaga apenas as linhas que a satisfazem. Se não forneceu uma, o predicado padrão sempre retorna `True` e você apagará todas as linhas.

Por exemplo:

```
users.delete(lambda row: row["user_id"] == 1)  # apaga linhas com user_id == 1
users.delete()                                 # apaga todas as linhas
```

SELECT

Tipicamente você não inspeciona uma tabela SQL diretamente. Em vez disso, você a consulta com uma declaração SELECT:

```
SELECT * FROM users;                            -- busca todo o conteúdo
SELECT * FROM users LIMIT 2;                    -- busca as duas primeiras linhas
SELECT user_id FROM users;                      -- busca somente colunas especificas
SELECT user_id FROM users WHERE name = 'Dunn':  -- busca somente linhas especificas
```

Você também pode usar declarações SELECT para calcular campos:

```
SELECT LENGTH(name) AS name_length FROM users;
```

Nós daremos a nossa classe `Table` um método `select()` que retorna uma nova `Table`. O método aceita dois argumentos opcionais:

- `keep_columns` especifica o nome das colunas que você quer manter no resultado. Se você não fornecê-la, o resultado contém todas as colunas.

- `additional_columns` é um dicionário cujas chaves são novos nomes de colunas e cujos valores são funções especificando como computar os valores de novas colunas.

Se você não fornecer nenhum dos dois, você simplesmente receberia uma cópia da tabela:

```
def select(self, keep_columns=None, additional_columns=None):

    if keep_columns is None:          # se nenhuma coluna especificada,
        keep_columns = self.columns   # retorna todas as colunas

    if additional_columns is None:
        additional_columns = {}

    # nova tabela para resultados
    result_table = Table(keep_columns + additional_columns.keys())

    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.iteritems():
            new_row.append(calculation(row))
        result_table.insert(new_row)

    return result_table
```

Nosso `select()` retorna uma nova tabela, enquanto o típico SQL `SELECT()` apenas produz algum tipo de resultado transitório (a menos que você explicitamente insira os resultados na tabela).

Nós também precisaremos dos métodos `where()` e `limit()`. Ambos são bem simples:

```
def where(self, predicate=lambda row: True):
    """retorna apenas as linhas que satisfazem o predicado fornecido"""
    where_table = Table(self.columns)
    where_table.rows = filter(predicate, self.rows)
    return where_table

def limit(self, num_rows):
    """retorna apenas as primeiras linhas num_rows"""
    limit_table = Table(self.columns)
    limit_table.rows = self.rows[:num_rows]
    return limit_table
```

e depois podemos facilmente construir equivalentes NotQuiteABase às declarações SQL precedentes:

```
# SELECT * FROM users;
users.select()

# SELECT * FROM users LIMIT 2;
users.limit(2)

# SELECT user_id FROM users;
users.select(keep_columns=["user_id"])
```

```
# SELECT user_id FROM users WHERE name = 'Dunn';
users.where(lambda row: row["name"] == "Dunn") \
     .select(keep_columns=["user_id"])

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row): return len(row["name"])

users.select(keep_columns=[],
             additional_columns = { "name_length" : name_length })
```

Observe que — diferente do resto do livro — aqui eu uso barra invertida \ para continuar declarações em múltiplas linhas. Acredito que isso torna a pesquisa de NotQuiteABase mais fácil de ler do que qualquer outra forma.

GROUP BY

Outra operação SQL comum é GROUP BY, que agrupa linhas com valores idênticos em colunas especificadas e produz valores agregados como MIN e MAX e COUNT e SUM. Isso deveria relembrar você da função group_by de "Manipulando Dados" na página 129.

Por exemplo, você pode querer encontrar o número de usuários e o menor user_id para cada possível tamanho de nome:

```
SELECT LENGTH(name) as name_length,
 MIN(user_id) AS min_user_id,
 COUNT(*) AS num_users
FROM users
GROUP BY LENGTH(name);
```

Cada campo que selecionamos com SELECT precisa estar na cláusula GROUP BY (o que name_length é) ou ser uma computação agregada (o que min_user_id e num_users são).

SQL também suporta uma cláusula HAVING que se comporta de forma similar a cláusula WHERE com exceção de seu filtro, que é aplicado aos agregados (enquanto que um WHERE filtraria todas as linhas antes mesmo da agregação começar).

Você pode querer saber o número médio de amigos de usuários cujos nomes começam com letras específicas mas apenas ver os resultados para letras com média correspondente maior que 1. (Sim, alguns desses exemplos são maquinados.)

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
 AVG(num_friends) AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

(Funções para trabalhar com strings variam pelas implementações SQL; alguns bancos de dados usam SUBSTRING ou outra coisa.)

Você também pode computar o total de agregados. Neste caso, você abandona o GROUP BY:

```
SELECT SUM(user_id) as user_id_sum
FROM users
WHERE user_id > 1;
```

Para acrescentar essa funcionalidade às `Tables` NotQuiteABase, adicionaremos um método `group_by()`. Ele usa os nomes das colunas que queremos agrupar, um dicionário de funções de agregação que você quer executar em cada grupo e um predicado `having` que opera em múltiplas linhas.

Os seguintes passos são realizados:

1. Cria `defaultdict` para mapear `tuples` (de agrupação por valores) para linhas (contendo agrupação por valores). Lembre-se de que não pode usar listas como chaves `dict`; você tem que usar tuplas.
2. Itera pelas linhas da tabela, populando o `defaultdict`.
3. Cria uma nova tabela com as colunas de saída corretas.
4. Itera por `defaultdict` e popula a tabela de saída, aplicando o filtro `having`, se houver.

(Um banco de dados real quase certamente faria isso de uma forma mais eficiente.)

```
def group_by(self, group_by_columns, aggregates, having=None):

    grouped_rows = defaultdict(list)

    # popula grupos
    for row in self.rows:
        key = tuple(row[column] for column in group_by_columns)
        grouped_rows[key].append(row)

    # tabela resultante com colunas group_by e agregados
    result_table = Table(group_by_columns + aggregates.keys())

    for key, rows in grouped_rows.iteritems():
        if having is None or having(rows):
            new_row = list(key)
            for aggregate_name, aggregate_fn in aggregates.iteritems():
                new_row.append(aggregate_fn(rows))
            result_table.insert(new_row)

    return result_table
```

Novamente, deixe-me ver como faríamos o equivalente às declarações SQL precedentes. As métricas `name_length` são:

```
def min_user_id(rows): return min(row["user_id"] for row in rows)
```

```
stats_by_length = users \
    .select(additional_columns={"name_length" : name_length}) \
    .group_by(group_by_columns=["name_length"],
              aggregates={ "min_user_id" : min_user_id,
                           "num_users" : len })
```

As métricas `first_letter`:

```
def first_letter_of_name(row):
    return row["name"][0] if row["name"] else ""

def average_num_friends(rows):
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows):
    return average_num_friends(rows) > 1

avg_friends_by_letter = users \
    .select(additional_columns={'first_letter' : first_letter_of_name}) \
    .group_by(group_by_columns=['first_letter'],
              aggregates={ "avg_num_friends" : average_num_friends },
              having=enough_friends)
```

e `user_id_sum` é:

```
def sum_user_ids(rows): return sum(row["user_id"] for row in rows)

user_id_sum = users \
    .where(lambda row: row["user_id"] > 1) \
    .group_by(group_by_columns=[],
              aggregates={ "user_id_sum" : sum_user_ids })
```

ORDER BY

Frequentemente, você desejará ordenar seus resultados. Por exemplo, você pode querer saber (alfabeticamente) os dois primeiros nomes de seus usuários:

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

Isso é fácil de implementar usando nosso método `order_by()` que pega uma função `order`:

```
def order_by(self, order):
    new_table = self.select()       # cria uma cópia
    new_table.rows.sort(key=order)
    return new_table
```

que podemos usar como segue:

```
friendliest_letters = avg_friends_by_letter \
    .order_by(lambda row: -row["avg_num_friends"]) \
    .limit(4)
```

O ORDER BY do SQL permite que você especifique ASC (ascendente) e DESC (descendentes) para cada campo; aqui teríamos que colocar isso em nossa função order.

JOIN

Os bancos de dados relacionais são frequentemente *normalizados*, o que significa que ele são organizados para minimizar redundâncias. Por exemplo, quando trabalhamos com interesses de nossos usuários em Python podemos apenas dar a cada usuário uma lista contendo seus interesses.

Tabelas SQL tipicamente não podem conter listas, então a solução típica é criar uma segunda tabela user_interests usando o relacionamento um-para-muitos entre user_ids e interesses. Em SQL você poderia fazer:

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

mas em NotQuiteABase você criaria a tabela:

```
user_interests = Table(["user_id", "interest"])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

Ainda há muita redundância — o "SQL" interessado está armazenado em dois lugares diferentes. Em um banco de dados real você talvez armazenasse user_id e interest_id na tabela user_interests, e então criaria uma terceira tabela interests mapeando interest_id para interest a fim de armazenar cada nome de interesse apenas uma vez. Aqui, eles apenas tornariam nossos exemplos mais complicados ainda.

Quando nossos dados estão em diferentes tabelas, como os analisamos? Fazendo a junção das duas tabelas. JOIN combina linhas na tabela à esquerda com linhas correspondentes na tabela da direita, onde o significado de "correspondente" é baseado em como especificamos a junção.

Por exemplo, para encontrar usuários interessados em SQL você consultaria:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

JOIN diz que, para cada linha em user, deveríamos ver o user_id e associar a linha com cada linha em user_interests contendo o mesmo user_id.

Note que tivemos que especificar quais tabelas usar JOIN e quais colunas juntar. Este é um INNER JOIN, que retorna as combinações de linhas (e apenas as combinações de linhas) que correspondem de acordo com o critério de junção especificado.

Também há o LEFT JOIN, que — além das combinações de linhas correspondentes — retorna uma linha para cada linha da tabela à esquerda sem linhas correspondentes (nesse caso, os campos que deveriam vir da tabela a direita seriam todos NULL).

Usando um LEFT JOIN, é fácil contar o número de interesses de cada usuário:

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

LEFT JOIN certifica que usuários sem interesses ainda terão linhas no conjunto de dados unidos (com valores NULL para campos provenientes de user_interests), e COUNT apenas conta valores que não são NULL.

A implementação join() de NotQuiteABase será mais restritiva — ela simplesmente faz a junção de duas tabelas em quaisquer colunas que elas tenham em comum. Mesmo assim, não é trivial escrever:

```
def join(self, other_table, left_join=False):

    join_on_columns = [c for c in self.columns            # colunas em
                       if c in other_table.columns]       # ambas as tabelas

    additional_columns = [c for c in other_table.columns # colunas apenas
                          if c not in join_on_columns]   # na tabela à direita

    # todas as colunas da tabela à esquerda + additional_columns da tabela à direita
    join_table = Table(self.columns + additional_columns)

    for row in self.rows:
        def is_join(other_row):
            return all(other_row[c] == row[c] for c in join_on_columns)

        other_rows = other_table.where(is_join).rows

        # cada linha que corresponda a esta produz uma linha resultado
        for other_row in other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [other_row[c] for c in additional_columns])

        # se nenhuma linha corresponde e há um left join, saída com NONE
        if left_join and not other_rows:
            join_table.insert([row[c] for c in self.columns] +
```

```
                                    [None for c in additional_columns])

        return join_table
```

Então, podemos encontrar usuários interessados em SQL com:

```
sql_users = users \
    .join(user_interests) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=["name"])
```

E conseguimos pegar a contagem de interesses com:

```
def count_interests(rows):
    """conta quantas linhas não têm diferente de interesse None"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = users \
    .join(user_interests, left_join=True) \
    .group_by(group_by_columns=["user_id"],
              aggregates={"num_interests" : count_interests })
```

Em SQL também há um `RIGHT JOIN`, que mantêm linhas da tabela a direita sem correspondentes e um `FULL ORDER JOIN`, que mantêm linhas de ambas tabelas que não possuem correspondentes. Não implementaremos nenhuma dessas.

Subconsultas

Em SQL, você pode selecionar (`SELECT`) de (e juntar (`JOIN`)) a partir de resultados de consultas como se elas fossem tabelas. Então, se você quiser encontrar o menor user_id de alguém interessado em SQL, você poderia usar uma subconsulta. (Claro, você poderia fazer o mesmo cálculo usando `JOIN`, mas não ilustraria subconsultas.)

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

Da forma como criamos NotQuiteABase, conseguimos isso de graça. Nossos resultados de consulta são tabelas.

```
likes_sql_user_ids = user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=['user_id'])

likes_sql_user_ids.group_by(group_by_columns=[],
                            aggregates={ "min_user_id" : min_user_id })
```

Índices

Para encontrar linhas contendo um valor específico (digamos, em que `name` é "Hero"), NotQuiteABase precisa inspecionar cada linha na tabela. Se a tabela possui muitas linhas, isso pode demorar muito.

De maneira parecida, nosso algoritmo `join` é extremamente ineficiente. Para cada linha na tabela a esquerda, ele inspeciona cada linha na tabela a direita para ver se é uma combinação. Com duas grandes tabelas isso pode demorar eternamente.

Também, você geralmente gostaria de aplicar restrições a algumas de suas colunas. Por exemplo, na tabela `users` você provavelmente não quer permitir que dois usuários diferentes tenham o mesmo `user_id`.

Índices resolvem todos esses problemas. Se a tabela `user_interests` tivesse um índice em `user_id`, um algoritmo `join` poderia encontrar combinações diretamente em vez de pesquisar a tabela toda. Se a tabela `users` tivesse um "único" índice em `user_id`, você receberia um erro se tentasse inserir uma cópia.

Cada tabela em um banco de dados pode ter um ou mais índices, o que permite que você rapidamente encontre linhas por colunas-chave, eficientemente junte tabelas e execute restrições únicas em colunas ou combinações de colunas.

Planejar e usar índices bem é um tipo de arte negra (o que varia dependendo do banco de dados específico), mas se você faz muito trabalho de banco de dados, vale a pena aprender.

Otimização de Consulta

Lembre-se da consulta (consulta) para encontrar todos os usuários interessados em SQL:

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

Em NotQuiteABase existe (pelo menos) duas maneiras diferentes para escrever esta consulta. Você poderia filtrar a tabela `user_interests` fazendo a junção:

```
user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .join(users) \
    .select(["name"])
```

Ou poderia filtrar os resultados da junção:

```
user_interests \
    .join(users) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(["name"])
```

Você acabaria com os mesmos resultados de qualquer forma, mas filtrar antes de juntar é quase certamente mais eficiente, porque neste caso `join` tem menos linhas para operar.

Em SQL, você não se preocuparia com isso no geral. Você "declara" os resultados que quer e deixa para o motor de consulta executá-los (e usar índices eficientemente).

NoSQL

Uma tendência recente em bancos de dados está direcionada a bancos de dados não relacionais "NoSQL", que não representam dados em tabelas. Por exemplo, MongoDB é um famoso banco de dados sem diagrama cujos elementos são arbitrariamente documentos JSON complexos em vez de linhas.

Existem bancos de dados de colunas que armazenam dados em colunas em vez de linhas (bons quando os dados têm muitas colunas mas as consultas precisam de poucas delas), armazenados valores-chave que são otimizados para recuperarem valores únicos (complexos) por suas chaves, bancos de dados para armazenar e criar gráficos transversais, bancos de dados que são otimizados para rodar em múltiplos datacenters, bancos de dados que são feitos para rodar na memória, bancos de dados para armazenar dados série temporal e centenas de outros.

A novidade de amanhã pode ainda nem existir, então eu não posso fazer nada além de informar que NoSQL existe. Então agora você sabe. Ele existe.

Para Mais Esclarecimentos

- Se você gostaria de fazer download de um banco de dados relacional para brincar, SQLite (*http://www.sqlite.org*) é rápido e pequeno, enquanto MySQL (*http://www.mysql.com*) e PostgreSQL (*http://www.postgresql.org*) são maiores e mais complicados. Todos são gratuitos e possuem uma grande documentação.
- Se você quiser explorar NoSQL, MongoDB (*http://www.mongodb.org*) é bem simples para começar, o que pode ser uma benção ou uma maldição. Ele também tem uma boa documentação.
- O artigo sobre NoSQL no Wikipédia (*http://en.wikipedia.org/wiki/NoSQL*) quase certamente possui links para bancos de dados que nem existiam quando este livro foi escrito.

CAPÍTULO 24

MapReduce

O futuro já chegou. Só não foi distribuído igualmente ainda.
—William Gibson

MapReduce é um modelo de programação para realizar processamento paralelo em grandes conjuntos de dados. Embora seja uma técnica poderosa, sua base é relativamente simples.

Imagine que temos uma coleção de itens que gostaríamos de processar. Por exemplo, os itens podem ser logs de web site, textos de livros variados, arquivos de imagens ou qualquer outra coisa. Uma versão básica do algoritmo MapReduce consiste dos seguintes passos:

1. Use uma função `mapper` para transformar cada item em zero ou mais pares chave-valor. (É chamado com frequência de função `map`, porém já existe uma função Python chamada `map` e não devemos confundir as duas.)
2. Junte todos os pares com chaves idênticas.
3. Use uma função `reducer` em cada coleção de valores agrupados para produzir valores de saída para a chave correspondente.

Isso tudo é um pouco abstrato, então vamos ver um exemplo específico. Há poucas regras absolutas de data science mas uma delas é que seu primeiro exemplo MapReduce deve envolver contagem de palavras.

Exemplo: Contagem de Palavras

DataSciencester cresceu para milhões de usuários! Isso é ótimo para a segurança do seu emprego mas dificulta um pouco a análise de rotina.

Por exemplo, a vice-presidente de Conteúdo quer que você saiba sobre o que as pessoas estão falando em suas atualizações de status. Como primeira tentativa, você decide contar as palavras que aparecem, para que você possa preparar um relato sobre os mais frequentes.

Quando você tinha poucas centenas de usuários isso era simples de fazer:

```
def word_count_old(documents):
    """contagem de palavras sem usar MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

Com milhões de usuários, o conjunto de `documents` (atualizações de status) é muito grande para caber no seu computador. Se você puder encaixar isso no modelo MapReduce, pode usar alguma infraestrutura "big data" que seus engenheiros implementaram.

Primeiro, queremos uma função que transforme um documento em uma sequência de pares chave-valor. Nós queremos que nossa saída seja agrupada por palavra, o que significa que as chaves deveriam ser palavras. E para cada palavra, apenas emitiremos o valor 1 para indicar que este par corresponde a uma ocorrência da palavra:

```
def wc_mapper(document):
    """para cada palavra no documento, emite (word, 1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Pulando o passo 2 por enquanto, imagine que para alguma palavra nós coletamos uma lista de contas correspondentes às que emitimos. Então para produzir a contagem total para aquela palavra, apenas precisamos:

```
def wc_reducer(word, counts):
    """soma as contagens para uma palavra"""
    yield (word, sum(counts))
```

Retornando ao passo 2, agora precisamos coletar os resultados de `wc_mapper` e fornecê-los a `wc_reducer`. Vamos pensar sobre o que faríamos em apenas um computador:

```
def word_count(documents):
    """conta as palavras nos documentos de saída usando MapReduce"""

    # lugar para armazenar valores agrupados
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

Imagine que temos três documentos `["data science", "big data", "science fiction"]`.

Então `wc_mapper` aplicado ao primeiro documento dá preferência aos dois pares `("data", 1)` e `("science", 1)`. Depois que nós passamos por todos os três documentos, o `collector` contém

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

Então `wc_reducer` produz a conta para cada palavra:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Por que MapReduce?

Como mencionado anteriormente, o principal benefício de MapReduce é que ele permite que distribuamos computações movendo o processamento aos dados. Imagine que queremos contar palavras em bilhões de documentos.

Nossa abordagem original (não MapReduce) requer que a máquina que está fazendo o processamento tenha acesso a todos os documentos. Isso significa que todos os documentos precisam viver na máquina ou ser transferidos para ela durante o processamento. Mais importante, significa que a máquina pode apenas processar um documento por vez.

Possivelmente, ele processa até alguns por vez se tiver múltiplos núcleos e se o código for reescrito para levar vantagem sobre eles. Mas mesmo assim, todos os documentos ainda têm que chegar naquela máquina.

Imagine agora que nossos bilhões de documentos estão espalhados por 100 máquinas. Com a infraestrutura certa, podemos fazer o seguinte:

- Faça a máquina rodar o mapeador (`mapper`) em seus documentos, produzindo muitos pares (chave-valor).
- Distribua aqueles pares (chave, valor) em um número de máquinas redutoras, certificando que os pares correspondentes a qualquer chave terminem na mesma máquina.
- Faça cada máquina redutora agrupar os pares por chave e então rodar o redutor em cada conjunto de valores.
- Retorne cada par (chave, saída).

O que é excelente nisso é que ele escala horizontalmente. Se dobrarmos o número de máquinas (ignorando certos custos fixos de rodar o sistema MapReduce), então nossa computação deveria rodar aproximadamente duas vezes mais rápido. Cada máquina mapeadora precisará fazer apenas metade do trabalho (assumindo que há chaves distintas o suficiente para distribuir ainda mais o trabalho do redutor) e o mesmo é verdade para máquinas redutoras.

MapReduce Mais Generalizado

Se você pensar nisso por um minuto, todo o código específico para contagem de palavra no exemplo anterior está contido nas funções `wc_mapper` e `wc_reducer`. Isso significa que com algumas mudanças nós temos um framework muito mais geral (que ainda roda em uma única máquina):

```
def map_reduce(inputs, mapper, reducer):
    """roda MapReduce nas entradas usando mapper e reducer"""
    collector = defaultdict(list)

    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)

    return [output
            for key, values in collector.iteritems()
            for output in reducer(key,values)]
```

E então podemos contar palavras simplesmente usando:

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

Isso nos dá a flexibilidade de resolver uma grande variedade de problemas.

Antes de continuarmos, observe que `wc_reducer` está apenas somando os valores correspondentes a cada chave. Esse tipo de agregação é tão comum que vale a pena abstrair:

```
def reduce_values_using(aggregation_fn, key, values):
    """reduz um par chave-valor aplicando aggregation_fn aos valores"""
    yield (key, aggregation_fn(values))

def values_reducer(aggregation_fn):
    """transforma uma função (valores -> saída) em uma redutora
    que mapeia (chave, valor) -> (chave, saída)"""
    return partial(reduce_values_using, aggregation_fn)
```

após o que podemos facilmente criar:

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))
```

e assim por diante.

Exemplo: Analisando Atualizações de Status

A vice-presidente de Conteúdo ficou impressionada com a contagem de palavras e pergunta o que mais você pode aprender das atualizações de status das pessoas. Você consegue extrair um conjunto de dados de atualizações de status que se parecem com:

```
{"id": 1,
 "username" : "joelgrus",
 "text" : "Is anyone interested in a data science book?",
 "created_at" : datetime.datetime(2013, 12, 21, 11, 47, 0),
 "liked_by" : ["data_guy", "data_gal", "mike"] }
```

Digamos que precisamos descobrir que dia da semana as pessoas mais falam sobre data science. Para descobrir isso, apenas contaremos quantas atualizações de data science existem em cada dia. Isso significa que precisaremos agrupar por dia da semana, então essa é nossa chave. E se emitirmos um valor de 1 para cada atualização que contém "data science", nós podemos simplesmente conseguir a soma total usando `sum`:

```
def data_science_day_mapper(status_update):
    """produz (day_of_week, 1) se status_update contém "data science" """
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

Como um exemplo um pouco diferente, imagine que precisamos descobrir a palavra mais comum que cada usuário coloca em suas atualizações de status. Há três abordagens possíveis que surgem na mente para `mapper`:

- Coloque o nome de usuário na chave; coloque as palavras e contagens nos valores.
- Coloque a palavra na chave; coloque os nomes de usuários nos valores.
- Coloque o nome de usuário e palavra na chave; coloque as contagens nos valores.

Se você pensar um pouco mais sobre isso, nós definitivamente queremos agrupar por `username`, porque nós queremos considerar as palavras de cada pessoa separadamente. E nós não queremos agrupar por palavra, já que nosso redutor precisará ver todas as palavras para cada pessoa para descobrir qual é a mais popular. Isso significa que a primeira opção é a opção certa:

```
def words_per_user_mapper(status_update):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))
```

```
def most_popular_word_reducer(user, words_and_counts):
    """dada uma sequência de pares (palavra, contagem),
    retorna a palavra com a maior contagem total"""

    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count

    word, count = word_counts.most_common(1)[0]

    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)
```

Ou poderíamos descobrir o número de diferentes curtições de status para cada usuário:

```
def liker_mapper(status_update):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)
```

Exemplo: Multiplicação de Matriz

Lembre-se de "Multiplicação de Matriz" na página 260 que dada uma matriz A $m \times n$ e uma matriz B $m \times k$, podemos multiplicá-las para formar uma matriz C $m \times k$, em que o elemento de C na fileira i e coluna j é dado por:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + ... + A_{in}B_{nj}$$

Como vimos, uma forma "natural" de representar uma matriz $m \times n$ é com uma lista de listas, onde o elemento A_{ij} é o elemento j-ésimo da lista i-ésima.

Mas grandes matrizes, às vezes, são *esparsas*, o que significa que a maioria de seus elementos são iguais a zero. Para grandes matrizes esparsas, uma lista de listas pode ser uma representação inútil. Uma representação mais compacta é uma lista de `tuples` `(name, i, j, value)` onde `name` identifica a matriz e onde `i`, `j`, `value` indica a localização com valor não zero.

Por exemplo, uma matriz bilhão x bilhão possui um *quintilhão* de entradas, que não seriam fáceis de armazenar em um computador. Mas se há apenas algumas entradas não zero em cada fileira, essa representação alternativa é muito menor.

Dado esse tipo de representação, podemos usar MapReduce para executar a multiplicação de matriz de uma maneira distribuída.

Para motivar nosso algoritmo, note que cada elemento A_{ij} é usado apenas para computar elementos de C na fileira i, e cada elemento B_{ij} é usado apenas para computar os elementos de C na coluna j. Nosso objetivo será para cada saída do nosso `reducer` ser uma entrada única de C, o que significa que precisaremos que nosso `mapper` emita chaves identificando uma única entrada de C. Isso sugere o seguinte:

```
def matrix_multiply_mapper(m, element):
    """m é a dimensão comum (colunas de A, linhas de B)
    elemento é uma tupla (matrix_name, i, j, value)"""
    name, i, j, value = element

    if name == "A":
        # A_ij é a j-ésima entrada na soma de cada C_ik, k=1..m
        for k in range(m):
            # agrupada com outras entradas para C_ik
            yield((i, k), (j, value))
    else:
        # B_ij é a i-ésima entrada na soma de cada C_kj
        for k in range(m):
            # agrupada com outras entradas para C_k
            yield((k, j), (i, value))

def matrix_multiply_reducer(m, key, indexed_values):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)

    # soma todos os produtos das posições com dois resultados
    sum_product = sum(results[0] * results[1]
                      for results in results_by_index.values()
                      if len(results) == 2)

    if sum_product != 0.0:
        yield (key, sum_product)
```

Por exemplo, se você tivesse as duas matrizes

```
A = [[3, 2, 0],
     [0, 0, 0]]

B = [[4, -1, 0],
     [10, 0, 0],
     [0, 0, 0]]
```

você poderia reescrevê-las como tuplas:

```
entries = [("A", 0, 0, 3), ("A", 0, 1,  2),
           ("B", 0, 0, 4), ("B", 0, 1, -1), ("B", 1, 0, 10)]
mapper = partial(matrix_multiply_mapper, 3)
```

```
reducer = partial(matrix_multiply_reducer, 3)

map_reduce(entries, mapper, reducer) # [((0, 1), -3), ((0, 0), 32)]
```

Isso não é interessante em matrizes pequenas mas se você tivesse milhões de fileiras e milhões de colunas, isso poderia ajudar muito.

Um Adendo: Combinadores

Uma coisa que você provavelmente notou é que muitos dos nossos mapeadores parecem incluir um monte de informação extra. Por exemplo, ao contar palavras, em vez de emitir `(word, 1)` e somar os valores, poderíamos emitir `(word, None)` e apenas pegar o tamanho.

Uma razão pela qual não fizemos isso é que, na configuração distribuída, às vezes queremos usar *combinadores* para reduzir a quantidade de dados que têm que ser transferidos de máquina para máquina. Se uma de nossas máquinas mapeadoras vêm a palavra "data" 500 vezes, podemos dizer para ela combinar 500 instâncias de `("data", 1)` em uma única `("data", 500)` antes de entregar para a máquina redutora. Isso resulta em muito menos dados sendo movidos, o que pode deixar nosso algoritmo ainda mais rápido.

Pela forma como escrevemos nosso redutor, ele lidaria com esses dados combinados corretamente. (Se tivéssemos escrito usando `len`, não faria.)

Para Mais Esclarecimentos

- O sistema MapReduce mais usado é Hadoop (*http://hadoop.apache.org*), que tem mérito em muitos livros. Há várias distribuições comerciais e não comerciais e um grande ecossistema de ferramentas relacionadas a Hadoop.

  Para usá-lo, você deve configurar seu próprio *cluster* (ou encontrar alguém que deixe você usar o dele), o que não é necessariamente uma tarefa para os fracos de coração. Mapeadores e redutores Hadoop são comumente escritos em Java, embora exista uma facilidade conhecida como "Hadoop streaming" que lhe permite escrevê-las em outras linguagens (incluindo Python).
- A Amazon oferece um serviço Elastic MapReduce (*http://aws.amazon.com/elasticmapreduce/*) que pode criar e destruir clusters, cobrando de você apenas pelo tempo que você os utiliza.
- mrjob é um pacote Python para interface com Hadoop (ou Elastic MapReduce).
- Trabalhos Hadoop são tipicamente de alta latência, o que os torna uma escolha ruim para análises em "tempo real". Há várias ferramentas de "tempo real"

construídas sobre Hadoop, mas também há muito frameworks alternativos que estão ficando mais populares. Dois dos mais populares são Spark (*http://spark.apache.org/*) e Storm (*http://storm.incubator.apache.org/*).

- Agora é bem provável que a novidade do dia seja algum framework que nem existia quando este livro foi escrito. Você terá que descobrir sozinho.

CAPÍTULO 25

Vá em Frente e Pratique Data Science

E agora, mais uma vez, eu ordeno a minha hedionda prole que siga em frente e prospere.
—Mary Shelley

Para onde você vai daqui? Supondo que eu não assustei você com data science, há um grande número de coisas que você deveria aprender em seguida.

IPython

Nós mencionamos IPython (*http://ipython.org/*) anteriormente no livro. Ele fornece um shell com muito mais funcionalidades do que o shell padrão Python e adiciona "funções mágicas" que permitem que você (dentre outras coisas) copie e cole o código (que é normalmente complicado pela combinação de formatação com linhas vazias e espaços em brancos) e rode scripts de dentro do shell.

Tornar-se um especialista em IPython facilitará mais a sua vida. (Até mesmo aprender um pouquinho de IPython tornará sua vida muito mais fácil.)

Além do mais, ele permite que você crie "cadernos" combinando texto, código Python vivo e visualizações que você pode compartilhar com outras pessoas, ou apenas manter como um diário do que você fez (Figura 25-1).

IP[y]: Notebook Stock Prices Last Checkpoint: Jan 25 15:40 (unsaved change

File Edit View Insert Cell Kernel Help

Code Cell Toolbar None

In [1]:
```
import csv
```

Here's where we read from the file

In [2]:
```
with open(r"c:\src\data-science-from-scratch\code\stocks.txt", "rb") as f:
    reader = csv.DictReader(f, delimiter='\t')
    data =  row for row in reader
```

What does this data look like?

In [3]:
```
print data[0]
```

```
{'date': '2015-01-23', 'symbol': 'AAPL', 'closing_price': '112.98'}
```

Now we can find the maximum price for AAPL stock using a list comprehension

In [4]:
```
print max(row["closing_price"] for row in data if row["symbol"] == "AAPL")
```

```
99.68
```

Figura 25-1. Um caderno IPython

Matemática

Ao longo do livro, nós exploramos álgebra linear (Capítulo 4), estatística (Capítulo 5), probabilidade (Capítulo 6) e aspectos variados do aprendizado de máquina.

Para ser um bom cientista de dados, você deve saber muito mais sobre esses tópicos e eu encorajo você a tentar estudar cada um deles, usando os livros recomendados ao final dos capítulos, seus livros preferidos, cursos online ou até mesmo cursos presenciais.

Não Do Zero

Implementar coisas "do zero" é ótimo para entender como elas funcionam. Mas geralmente não é ótimo em performance (a não ser que você os esteja implementando especificamente com performance em mente), facilidade de uso, resposta rápida ou tratamento de erros.

Na prática, você vai querer usar bibliotecas bem projetadas que implementem solidamente os essenciais. (Minha proposta original para este livro envolvia uma segunda metade "agora vamos aprender as bibliotecas" que a O'Reilly, felizmente, vetou.)

NumPy

NumPy (*http://www.numpy.org*) (para "Numeric Python") fornece facilidades para fazer computação científica "real". Ele contém arrays que desempenham melhor do que nossos vetores `list`, matrizes que desempenham melhor do que nossas matrizes `list-of-list` e várias funções numéricas para trabalhar com eles.

NumPy é um pilar para muitas outras bibliotecas, o que torna seu conhecimento especialmente valioso.

pandas

pandas (*http://pandas.pydata.org*) fornece estruturas de dados adicionais para trabalhar com conjuntos de dados em Python. Sua abstração primária é o `DataFrame`, que é conceitualmente similar à classe `NotQuiteABase Table` que construímos no Capítulo 23, mas com muito mais funcionalidades e melhor performance.

Se você usar Python para analisar, dividir, agrupar ou manipular conjuntos de dados, `pandas` uma ferramenta de valor inestimável.

scikit-learn

scikit-learn (*http://scikit-learn.org*) provavelmente é a biblioteca mais popular para fazer aprendizado de máquina em Python. Ela contém todos os modelos que implementamos e muitos mais que não usamos. Em um problema real, você jamais construiria uma árvore de decisão do zero; você faria `scikit-learn` fazer o trabalho pesado. Em um problema real, você jamais escreveria um algoritmo de otimização à mão; você contaria que `scikit-learn` já estivesse usando um muito bom.

Sua documentação contém muitos exemplos (*http://scikit-learn.org/stable/auto_examples/*) do que pode fazer (e o que o aprendizado de máquina pode fazer).

Visualização

Os gráficos `matplotlib` que criamos foram limpos e funcionais mas não particularmente estilosos (e nada interativos). Se você quiser se aprofundar em visualização de dados, você possui muitas opções.

A primeira é explorar mais `matplotlib`, cujas características nós já falamos. Seu web site contém muitos exemplos (*http://matplotlib.org/examples/*) de suas funcionalidades e uma galeria (*http://matplotlib.org/gallery.html*) de alguns de seus exemplos mais interessantes. Se você quiser criar visualizações estáticas, este é, provavelmente, seu próximo passo.

Você também deveria verificar seaborn, que é uma biblioteca que (dentre outras coisas) torna `matplotlib` mais atraente.

Se você quiser criar visualizações *interativas* que você possa compartilhar na Web, a opção óbvia é D3.js, uma biblioteca JavaScript para criar "Documentos Direcionados por Dados" (estes são os três Ds). Mesmo que você não saiba muito JavaScript, é possível pegar exemplos da galeria D3 e ajustá-los para trabalharem com seus dados. (Bons cientistas copiam da galeria D3. Ótimos cientistas *roubam* da galeria D3.)

Mesmo que você não tenha interesse em D3, apenas dar uma olhada na galeria é bem educativo para visualização de dados.

Bokeh é um projeto que traz funcionalidade de estilo D3 para Python.

R

Mesmo que você possa se sair bem sem aprender R, muitos cientistas de dados e projetos de data science usam isso, então vale a pena ao menos se familiarizar.

Em parte, isso é para que possa entender postagens de blogs baseadas em R e exemplos e código; em parte, isso é para lhe ajudar a apreciar a (comparativamente) elegância de Python; e, em parte, isso é para você se tornar um participante melhor informado na eterna guerra "R versus Python".

No mundo não faltam tutorias de R, cursos de R e livros de R. Eu escuto boas coisas sobre *Hands-on Programming with R* e não apenas porque também é um livro O'Reilly. (Ok, em grande parte por ser um livro O'Reilly.)

Encontre Dados

Se você está fazendo data science como parte do seu trabalho, você provavelmente conseguirá dados como parte do seu trabalho (mas não necessariamente). E se você estiver fazendo data science por diversão? Dados estão em todos os lugares, mas estes são alguns pontos de partida:

- Data.gov é o portal de dados do governo. Se você quiser dados de qualquer coisa a respeito do governo (o que parece ser a onda do momento) é um bom lugar para começar.
- reddit possui alguns fóruns, r/datasets e r/data, que são lugares para descobrir e perguntar sobre dados.

- A Amazon mantém uma coleção de conjunto de dados públicos que eles gostariam que você analisasse usando seus produtos (mas que você pode analisar com qualquer produto que quiser).
- Robb Seaton possui uma lista de conjuntos de dados selecionados em seu blog.
- Kaggle é um site que faz competições de data science. Eu nunca consegui entrar em uma (eu não tenho um espírito muito competitivo) mas você pode tentar.

Pratique Data Science

Dar uma olhada em catálogos de dados é bom, mas os melhores projetos (e produtos) são aquele que dão uma certa coceira. Estes foram alguns que eu fiz.

Hacker News

Hacker News é um site de agregação de notícias e de discussão sobre notícias relacionadas a tecnologia. Ele coleciona muitos artigos, muitos dos quais não são interessantes para mim.

Muitos anos atrás, eu construí um classificador de história Hacker News para prever se eu estaria interessado ou não em uma história. Isso não foi muito bom com os usuários de Hacker News, que ficaram magoados com o fato de alguém não estar interessado em todas as histórias no site.

Isso envolveu rotular muitas histórias (para conseguir um pouco de treinamento), escolher atributos de histórias (por exemplo, palavras em um título, e domínios de links) e treinar um classificador Naive Bayes não muito diferente do nosso filtro de spam.

Por razões agora perdidas na história, eu o construí em Ruby. Aprenda com os meus erros.

Carros de Bombeiros

Eu moro em uma grande rua no centro de Seattle, entre uma estação do corpo de bombeiros e a maioria dos incêndios da cidade (ou é o que parece). Com o passar dos anos, eu desenvolvi um interesse recreacional pelo Corpo de Bombeiros de Seattle.

Felizmente (de uma perspectiva de dados) eles mantêm um site em tempo real 911 que lista cada alarme de incêndio com os carros de bombeiros envolvidos.

E então, para satisfazer meu interesse, eu juntei muitos anos de dados de alarmes e realizei uma análise de rede social dos carros de bombeiros. Entre outras coisas, isso exigiu que

eu inventasse uma noção de centralidade específica de carro de bombeiros, que eu chamei de TruckRank.

Camisetas

Eu tenho uma filha jovem e uma fonte incessante de frustração para mim durante sua infância foi que a maioria das "blusas femininas" são sem graça, enquanto "camisas masculinas" são muito divertidas.

Em particular, estava claro para mim que havia uma diferença entre as camisas para bebês meninos e bebês meninas. Então, eu me perguntei se eu poderia treinar um modelo para reconhecer essas diferenças.

Resultado: Eu pude.

Isso envolveu fazer download de imagens de centenas de camisas, minimizando-as para o mesmo tamanho, torná-las em vetores de cores pixel e usar regressão logística para construir um classificador.

Uma abordagem parecia simples onde as cores eram apresentadas em cada camisa; uma segunda abordagem encontrou os 10 componentes principais dos vetores da imagem da camisa e classificou cada camisa usando suas projeções em um espaço dimensional 10 abrangendo as "autocamisetas" (Figura 25-2).

Figura 25-2. Autocamisetas correspondentes ao primeiro componente principal

E Você?

O que interessa você? Quais perguntas tiram seu sono? Procure por um conjunto de dados e faça um pouco de data science.