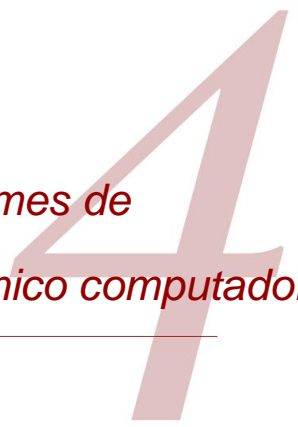


Lidando com grandes volumes de dados em um único computador



Este capítulo cobre

- Trabalhando com grandes conjuntos de dados em um único computador
- Trabalhar com bibliotecas Python adequadas para conjuntos de dados maiores
- Compreender a importância de escolher algoritmos e estruturas de dados corretos
- Compreender como você pode adaptar algoritmos para trabalhar dentro de bancos de dados

E se você tivesse tantos dados que parecessem superá-lo e suas técnicas não parecessem mais suficientes? O que você faz, se entrega ou se adapta?

Felizmente você optou por se adaptar, porque ainda está lendo. Este capítulo apresenta técnicas e ferramentas para lidar com conjuntos de dados maiores que ainda podem ser gerenciados por um único computador se você adotar as técnicas corretas.

Este capítulo fornece as ferramentas para realizar classificações e regressões quando os dados não cabem mais na RAM (memória de acesso aleatório) do seu computador, enquanto o capítulo 3 se concentra nos conjuntos de dados na memória. O Capítulo 5 irá um passo além e ensinará como lidar com conjuntos de dados que exigem vários computadores para

ser processado. Quando nos referimos a *grandes dados* neste capítulo, queremos dizer dados que causam problemas de trabalho em termos de memória ou velocidade, mas que ainda podem ser manipulados por um único computador.

Começamos este capítulo com uma visão geral dos problemas que você enfrenta ao lidar com grandes conjuntos de dados. Em seguida, oferecemos três tipos de soluções para superar esses problemas: adaptar seus algoritmos, escolher as estruturas de dados corretas e escolher as ferramentas certas. Os cientistas de dados não são os únicos que precisam lidar com grandes volumes de dados, portanto, você pode aplicar as melhores práticas gerais para resolver o problema de grandes volumes de dados. Finalmente, aplicamos esse conhecimento a dois estudos de caso. O primeiro caso mostra como detectar URLs maliciosas e o segundo caso demonstra como construir um mecanismo de recomendação dentro de um banco de dados.

4.1 Os problemas que você enfrenta ao lidar com grandes volumes de dados

Um grande volume de dados apresenta novos desafios, como memória sobrecarregada e algoritmos que nunca param de funcionar. Obriga você a adaptar e expandir seu repertório de técnicas. Mas mesmo quando você pode realizar sua análise, você deve cuidar de questões como E/S (entrada/saída) e falta de CPU , porque elas podem causar problemas de velocidade. A Figura 4.1 mostra um mapa mental que se desdobrará gradualmente à medida que avançamos pelas etapas: problemas, soluções e dicas.

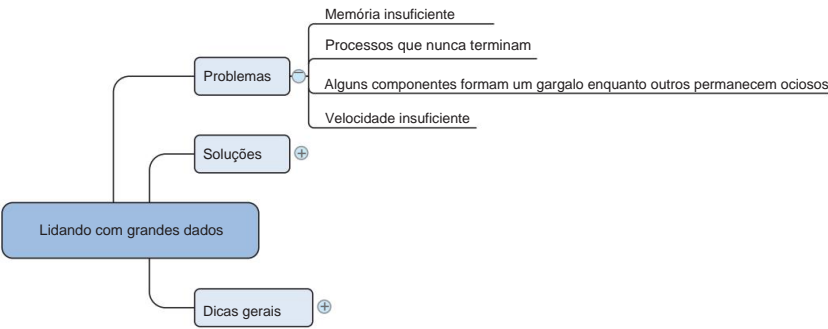


Figura 4.1 Visão geral dos problemas encontrados ao trabalhar com mais dados do que cabem na memória

Um computador possui apenas uma quantidade limitada de RAM. Quando você tenta inserir mais dados nesta memória do que realmente cabe, o sistema operacional começará a trocar blocos de memória por discos, o que é muito menos eficiente do que ter tudo na memória. Mas apenas alguns algoritmos são projetados para lidar com grandes conjuntos de dados; a maioria deles carrega todo o conjunto de dados na memória de uma só vez, o que causa o erro de falta de memória. Outros algoritmos precisam manter múltiplas cópias dos dados na memória ou armazenar resultados intermediários. Tudo isso agrava o problema.

Mesmo quando você cura os problemas de memória, pode precisar lidar com outro recurso limitado: o *tempo*. Embora um computador possa pensar que você vive milhões de anos, em realidade, você não o fará (a menos que entre em criostase até que seu PC esteja pronto). Certos algoritmos não levam em conta o tempo; eles continuarão funcionando para sempre. Outros algoritmos não podem terminar em um período de tempo razoável quando precisam processar apenas alguns megabytes de dados.

Uma terceira coisa que você observará ao lidar com grandes conjuntos de dados é que os componentes do seu computador pode começar a formar um gargalo enquanto deixa outros sistemas ociosos. Embora isso não seja tão grave quanto um algoritmo sem fim ou erros de falta de memória, ainda incorre em custos sérios. Pense na economia de custos em termos de dias de trabalho e infraestrutura de computação para a falta de CPU. Certos programas não alimentam dados rapidamente suficiente para o processador porque eles precisam ler dados do disco rígido, que é um dos componentes mais lentos de um computador. Isto foi abordado com o introdução de unidades de estado sólido (SSD), mas os SSDs ainda são muito mais caros do que a tecnologia de unidade de disco rígido (HDD) mais lenta e difundida.

4.2 Técnicas gerais para lidar com grandes volumes de dados

Algoritmos intermináveis, erros de falta de memória e problemas de velocidade são os desafios mais comuns que você enfrenta ao trabalhar com grandes volumes de dados. Nesta seção, investigaremos soluções para superar ou aliviar esses problemas.

As soluções podem ser divididas em três categorias: usar os algoritmos corretos, escolher a estrutura de dados correta e usar as ferramentas certas (figura 4.2).

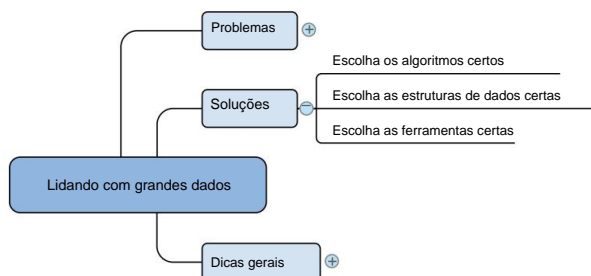


Figura 4.2 Visão geral de soluções para lidar com grandes conjuntos de dados

Não existe um mapeamento claro e individual entre os problemas e as soluções porque muitas soluções abordam a falta de memória e o desempenho computacional. Para Por exemplo, a compactação do conjunto de dados ajudará você a resolver problemas de memória porque o conjunto de dados fica menor. Mas isso também afeta a velocidade de computação com uma mudança do modo lento disco rígido para a CPU rápida. Ao contrário da RAM (memória de acesso aleatório), o disco rígido armazenará tudo mesmo depois que a energia cair, mas gravar em disco custa mais tempo do que alterar informações na RAM fugaz. Ao mudar constantemente o informações, a RAM é, portanto, preferível ao disco rígido (mais durável). Com um

conjunto de dados descompactado, inúmeras operações de leitura e gravação (E/S) estão ocorrendo, mas a CPU permanece praticamente ociosa, enquanto que com o conjunto de dados compactados a CPU obtém sua parte justa da carga de trabalho. Tenha isso em mente enquanto exploramos algumas soluções.

4.2.1 Escolhendo o algoritmo certo

Escolher o algoritmo certo pode resolver mais problemas do que adicionar mais ou melhor hardware. Um algoritmo adequado para lidar com grandes dados não precisa carregar todo o conjunto de dados na memória para fazer previsões. Idealmente, o algoritmo também suporta cálculos paralelizados. Nesta seção, examinaremos três tipos de algoritmos que podem fazer isso: *algoritmos online*, *algoritmos de bloco* e *algoritmos MapReduce*, conforme mostrado na figura 4.3.

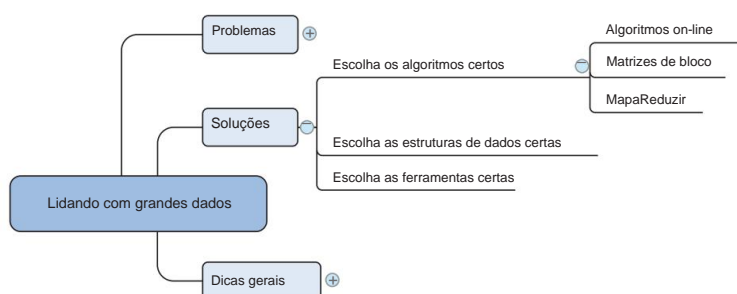


Figura 4.3 Visão geral das técnicas para adaptar algoritmos a grandes conjuntos de dados

ALGORITMOS DE APRENDIZAGEM

ON-LINE Vários, mas não todos, algoritmos de aprendizagem de máquina podem ser treinados usando uma observação por vez, em vez de armazenar todos os dados na memória. Ao chegar um novo ponto de dados, o modelo é treinado e a observação pode ser esquecida; seu efeito agora está incorporado aos parâmetros do modelo. Por exemplo, um modelo utilizado para prever o tempo pode utilizar diferentes parâmetros (como pressão atmosférica ou temperatura) em diferentes regiões. Quando os dados de uma região são carregados no algoritmo, ele esquece esses dados brutos e passa para a próxima região. Esta forma de trabalhar “use e esqueça” é a solução perfeita para o problema da memória, já que é improvável que uma única observação seja grande o suficiente para preencher toda a memória de um computador moderno.

A Listagem 4.1 mostra como aplicar este princípio a um perceptron com aprendizagem online. Um *perceptron* é um dos algoritmos de aprendizagem de máquina menos complexos usados para classificação binária (0 ou 1); por exemplo, o cliente comprará ou não?

Listagem 4.1 Treinando um perceptron por observação

A taxa de aprendizagem de um algoritmo é o ajuste que ele faz cada vez que uma nova observação chega. Se for alta, o modelo se ajustará rapidamente às novas observações, mas poderá "ultrapassar" e nunca ser preciso. Um exemplo simplificado: o peso ideal (e desconhecido) para uma variável $x = 0,75$. A estimativa atual é de 0,4 com uma taxa de aprendizagem de 0,5; o ajuste = $0,5$ (taxa de aprendizagem) * 1 (tamanho do erro) * 1 (valor de x) = $0,5 \cdot 0,4$ (peso atual) + $0,5$ (ajuste) = $0,9$ (novo peso), em vez de 0,75. O ajuste foi muito grande para obter o resultado correto.

```
importar numpy como np
classe perceptron(): def
    __init__(self, X,y, limite = 0,5,
taxa_de_aprendizagem = 0,1, épocas_máx = 10):
    self.threshold = limite
    self.learning_rate = learning_rate self.X = X
```

Configura
a classe perceptron.

O método `__init__` de qualquer classe Python é sempre executado ao criar uma instância da classe. Vários valores padrão são definidos aqui.

```
self.y = y
self.max_épocas = max_épocas
```

O limite é um corte arbitrário entre 0 e 1 para decidir se a previsão se torna 0 ou 1. Geralmente é 0,5, bem no meio, mas depende do caso de uso.

Uma época é uma análise de todos os dados. Permitimos um máximo de 10 execuções até pararmos o perceptron.

As variáveis X e y são atribuídas à classe.

```
def inicializar(self, tipo_inicial = 'zeros'): se tipo_inicial == 'aleatório':
```

Cada observação terminará com um peso. A função de inicialização define esses pesos para cada observação recebida. Permitimos 2 opções: todos os pesos começam em 0 ou recebem um peso aleatório pequeno (entre 0 e 0,05).

```
self.weights = np.random.rand(len(self.X[0])) * 0,05
se tipo_inicial == 'zeros': self.weights =
    np.zeros(len(self.X[0]))
```

Começamos na primeira época.

A função de treinamento.

```
def trem (self): época = 0
```

Verdade é sempre verdade, então tecnicamente este é um loop sem fim, mas construímos várias condições de parada (interrupção).

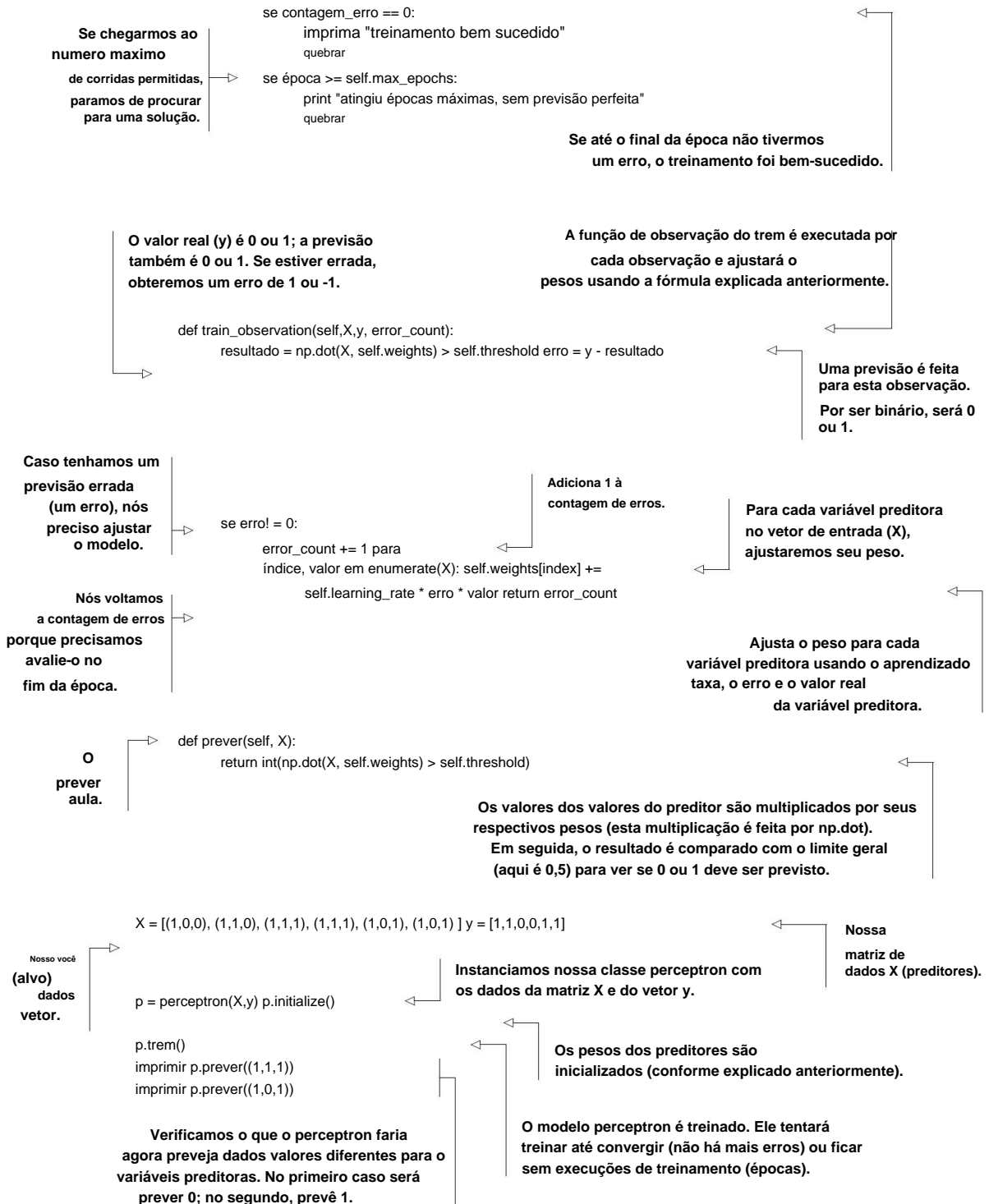
```
enquanto Verdadeiro:
```

Adiciona um ao número atual de épocas.

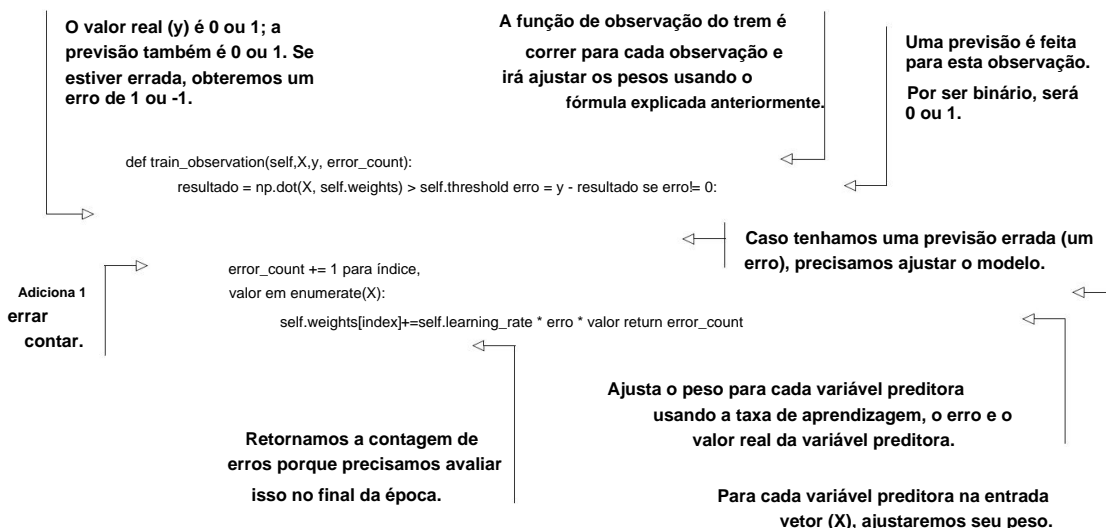
```
error_count = 0 época
+= 1 para (X,y)
em zip(self.X, self.y):
    contagem_erro += self.train_observation(X,y,contagem_erro)
```

Inicia o número de erros encontrados em 0 para cada época. Isso é importante; se uma época terminar sem erros, o algoritmo convergiu e pronto.

Percorremos os dados e o feed para a função de observação do trem, uma observação de cada vez.



Iremos ampliar partes do código que podem não ser tão evidentes de entender sem maiores explicações. Começaremos explicando como a função `train_observation()` funciona. Esta função tem duas partes grandes. A primeira é calcular a previsão de um observação e compare-o com o valor real. A segunda parte é mudar o pesos se a previsão parecer errada.



A previsão (y) é calculada multiplicando o vetor de entrada das variáveis independentes pelos seus respectivos pesos e somando os termos (como na regressão linear). Então esse valor é comparado com o limite. Se for maior que o limite, o algoritmo fornecerá 1 como saída e, se for menor que o limite, o algoritmo dá 0 como saída. Definir o limite é algo subjetivo e depende do seu caso de negócio. Digamos que você esteja prevendo se alguém tem uma determinada doença letal, sendo 1 positivo e 0 negativo. Neste caso é melhor ter um limite mais baixo: não é tão ruim ser considerado positivo e fazer uma segunda investigação do que ignorar a doença e deixar o paciente morrer. É calculado o erro que dará o sentido da mudança dos pesos.

```
resultado = np.dot(X, self.weights) > self.threshold
erro = y - resultado
```

Os pesos são alterados de acordo com o sinal do erro. A atualização é feita com a regra de aprendizagem para perceptrons. Para cada peso no vetor de pesos, você atualiza seu valor com a seguinte regra:

$$\tilde{y}w_i = \tilde{y}\tilde{y}x_i$$

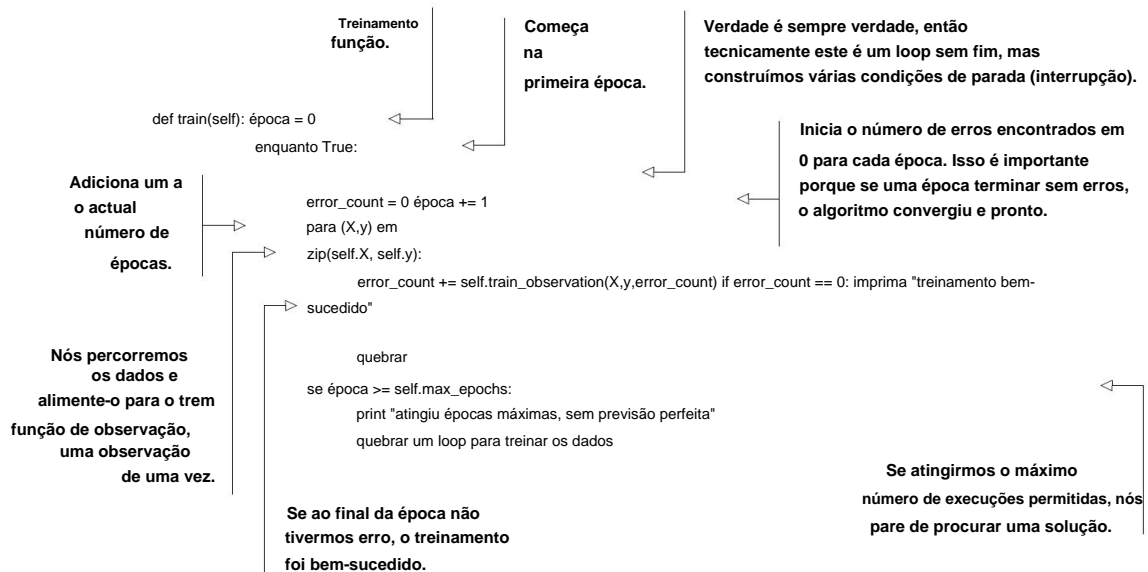
onde $\tilde{y}w_i$ é a quantidade que o peso precisa ser alterado, \tilde{y} é a taxa de aprendizagem, \tilde{y} é o erro e x_i é o i-ésimo valor no vetor de entrada (a i-ésima variável preditora). O

contagem de erros é uma variável para controlar quantas observações foram previstas incorretamente nesta época e é retornado para a função de chamada. Você adiciona uma observação ao contador de erros se a previsão original estava errada. Uma *época* é uma única corrida de treinamento através de todas as observações.

```
se erro! = 0:
    contagem_erro += 1
    para índice, valor em enumerate(X):
        self.weights[índice] += self.learning_rate * erro * valor
```

A segunda função que discutiremos com mais detalhes é a função `train()`. Esse função tem um loop interno que continua treinando o perceptron até que ele possa prever perfeitamente ou até atingir um certo número de rodadas de treinamento (épocas), conforme mostrado na listagem a seguir.

Listagem 4.2 Usando funções de trem



A maioria dos algoritmos online também pode lidar com minilotes; desta forma, você pode alimentá-los lotes de 10 a 1.000 observações de uma só vez, usando uma janela deslizante para examinar seus dados. Você tem três opções:

- *Aprendizado em lote completo* (também chamado de *aprendizado estatístico*) — *alimenta* o algoritmo com todos os dados de uma vez só. Foi isso que fizemos no capítulo 3.
- *Aprendizado em minilote* — *Alimente* o algoritmo com uma colherada (100, 1000,..., dependendo sobre o que seu hardware pode suportar) de observações por vez.
- *Aprendizado on-line* — *Alimente* o algoritmo com uma observação por vez.

As técnicas de aprendizagem online estão relacionadas a *algoritmos de streaming*, onde você vê cada ponto de dados apenas uma vez. Pense nos dados recebidos do Twitter: eles são carregados nos algoritmos e então a observação (tweet) é descartada porque o grande número de tweets de dados recebidos podem em breve sobrecarregar o hardware. Os algoritmos de aprendizagem online diferem dos algoritmos de streaming porque podem ver as mesmas observações várias vezes. É verdade que os algoritmos de aprendizagem on-line e os algoritmos de streaming podem *ambos* aprender com as observações, uma por uma. A diferença é que os *algoritmos online* são também usado em uma fonte de dados estática, bem como em uma fonte de dados de streaming, apresentando os dados em pequenos lotes (tão pequenos quanto uma única observação), o que permite que você vá sobre os dados várias vezes. Este não é o caso de um *algoritmo de streaming*, onde os dados flui para o sistema e você precisa fazer os cálculos normalmente imediatamente. Eles são semelhantes porque lidam com apenas alguns de cada vez.

DIVIDINDO UMA MATRIZ GRANDE EM MUITAS PEQUENAS

Embora no capítulo anterior quase não tenhamos precisado lidar com a forma exata como o algoritmo estima os parâmetros, mergulhar nisso às vezes pode ajudar. Ao cortar um grande tabela de dados em pequenas matrizes, por exemplo, ainda podemos fazer uma regressão linear. O lógica por trás dessa divisão de matriz e como uma regressão linear pode ser calculada com matrizes podem ser encontradas na barra lateral. Basta saber por enquanto que o Python bibliotecas que estamos prestes a usar cuidarão da divisão da matriz e da regressão linear pesos variáveis podem ser calculados usando cálculo matricial.

Matrizes de bloco e fórmula matricial de estimativa do coeficiente de regressão linear

Certos algoritmos podem ser traduzidos em algoritmos que usam blocos de matrizes em vez de matrizes completas. Ao particionar uma matriz em uma matriz de bloco, você divide a matriz completa em partes e trabalhe com as partes menores em vez da matriz completa. Em neste caso você pode carregar matrizes menores na memória e realizar cálculos, evitando assim um erro de falta de memória. A Figura 4.4 mostra como você pode reescrever a matriz adição $A + B$ em submatrizes.

$$\begin{aligned}
 A + B &= \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \cdots & b_{1,m} \\ \vdots & & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{bmatrix} \\
 &= \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \cdots & b_{1,m} \\ \vdots & & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{bmatrix} = \begin{bmatrix} a_{1,1} \\ a_{2,1} \end{bmatrix} + \begin{bmatrix} b_{1,1} \\ b_{2,1} \end{bmatrix}
 \end{aligned}$$

Figura 4.4 Matrizes de bloco podem ser usadas para calcular a soma das matrizes A e B.

(contínuo)

A fórmula da figura 4.4 mostra que não há diferença entre somar as matrizes A e B em uma única etapa ou somar primeiro a metade superior das matrizes e depois somar a metade inferior.

Todas as operações comuns de matrizes e vetores, como multiplicação, inversão e decomposição de valores singulares (uma técnica de redução de variável como PCA), podem ser escritas em termos de matrizes de bloco.¹ As operações de matriz de bloco economizam memória ao dividir o problema em blocos menores e são fáceis de paralelizar.

Embora a maioria dos pacotes numéricos tenha código altamente otimizado, eles funcionam apenas com matrizes que cabem na memória e usarão matrizes de blocos na memória quando for vantajoso. Com matrizes sem memória, eles não otimizam isso para você e cabe a você particionar a matriz em matrizes menores e implementar a versão da matriz de bloco.

Uma regressão linear é uma forma de prever variáveis contínuas com uma combinação linear de seus preditores; uma das maneiras mais básicas de realizar os cálculos é com uma técnica chamada mínimos quadrados ordinários. A fórmula em forma de matriz é

$$\tilde{y} = \tilde{y}^T X^T X \tilde{y}^{-1} X^T y$$

onde \tilde{y} são os coeficientes que você deseja recuperar, X são os preditores e y é a variável de destino.

As ferramentas Python que temos à nossa disposição para realizar nossa tarefa são as seguintes:¹

\tilde{y} *bcolz* é uma biblioteca Python que pode armazenar arrays de dados de forma compacta e usa o disco rígido quando o array não cabe mais na memória principal. \tilde{y}

Dask é uma biblioteca que permite otimizar o fluxo de cálculos e facilita a execução de cálculos em paralelo. Ele não vem com a configuração padrão do Anaconda, portanto, certifique-se de usar *conda* para instalar *dask* em seu ambiente virtual antes de executar o código abaixo. Nota: alguns erros foram relatados na importação do *Dask* ao usar Python de 64 bits. *Dask* depende de algumas outras bibliotecas (como *toolz*), mas as dependências devem ser resolvidas automaticamente por *pip* ou *conda*.

A listagem a seguir demonstra cálculos de matriz de blocos com essas bibliotecas.

¹ Para aqueles que querem tentar, as transformações Dadas são mais fáceis de alcançar do que as transformações Householder ao calcular decomposições de valores singulares.

Listagem 4.3 Cálculos de matrizes de blocos com bibliotecas bcolz e Dask

Número de observações

(notação científica).

1e4 = 10.000. Sinta-se

à vontade para mudar isso.

```
importar dask.array como da
importar bcolz como bc
importar numpy como np
importar dinamarquês
```

```
n = 1e4
```

```
ar = bc.carray(np.arange(n).reshape(n/2,2) , dtype='float64',
              rootdir = 'ar.bcolz', modo = 'w')
y = bc.carray(np.arange(n/2), dtype='float64', rootdir =
              'yy.bcolz', modo = 'w')
```

```
dax = da.from_array(ar, pedaços=(5,5)) dy =
da.from_array(y,pedaços=(5,5))
```

O XTX é definido (definindo-o como “preguiçoso”) como a matriz X multiplicada pela sua versão transposta. Este é um bloco de construção da fórmula para fazer regressão linear usando cálculo matricial.

```
XTX = dax.T.ponto(dax)
Xy = dax.T.ponto(dy)
```

```
coeficientes = np.linalg.inv(XTX.compute()).dot(Xy.compute())
```

```
coef = da.from_array(coeficientes,pedaços=(5,5))
```

```
ar.flush() e.flush()
```

Libere os dados da memória. Não é mais necessário ter matrizes grandes na memória.

```
previsões = dax.dot(coef).compute() imprimir previsões
```

Os coeficientes são calculados utilizando a função de regressão linear matricial. `np.linalg.inv()` é o $^{-1}$ nesta função, ou “inversão” da matriz. `X.dot(y)` -> multiplica a matriz X por outra matriz y.

Cria dados falsos: `np.arange(n).reshape(n/2,2)` cria uma matriz de 5.000 por 2 (porque definimos n como 10.000). `bc.carray = numpy` é uma extensão de array que pode trocar para disco. Isso também é armazenado de forma compactada. `rootdir = 'ar.bcolz'` -> cria um arquivo no disco caso esteja fora de BATER. Você pode verificar isso em seu sistema de arquivos ao lado deste arquivo ipython ou qualquer local de onde você executou esse código. `modo = 'w'` -> é o modo de gravação. `dtype = 'float64'` -> é o tipo de armazenamento dos dados (que são números flutuantes).

Matrizes de bloco são criadas para as variáveis preditoras (ar) e alvo (y). Uma matriz de bloco é uma matriz cortada em pedaços (blocos). `da.from_array()` lê dados do disco ou RAM (onde quer que residam atualmente). `chunks=(5,5)`: cada bloco é uma matriz 5x5 (a menos que restem <5 observações ou variáveis).

Xy é o vetor y multiplicado pela matriz X transposta. Novamente, a matriz está apenas definida, ainda não calculada. Este também é um alicerce da fórmula para fazer regressão linear usando cálculo matricial (ver fórmula).

Os coeficientes também são colocados em uma matriz de blocos. Recebemos um array numpy da última etapa, então precisamos convertê-lo explicitamente de volta em um “array da”

Pontue o modelo (fazer previsões).

Observe que você não precisa usar uma inversão de matriz de bloco porque XTX é um quadrado matriz com tamanho nr. de preditores * nr. de preditores. Isso é uma sorte porque Dask

ainda não suporta inversão de matriz de bloco. Você pode encontrar informações mais gerais sobre aritmética matricial na página da Wikipedia em [https://en.wikipedia.org/wiki/Matrix_\(matemática\)](https://en.wikipedia.org/wiki/Matrix_(matemática)).

MAPREDUCE

Os algoritmos MapReduce são fáceis de entender com uma analogia: imagine que você fosse solicitado a contar todos os votos para as eleições nacionais. O seu país tem 25 partidos, 1.500 locais de votação e 2 milhões de pessoas. Você pode optar por reunir todos os bilhetes de votação de cada escritório individualmente e contá-los centralmente, ou pode pedir aos escritórios locais que contem os votos dos 25 partidos e entreguem os resultados a você, e você pode então agregá-los por partido.

Os redutores de mapa seguem um processo semelhante ao da segunda forma de trabalhar. Eles primeiro mapeiam valores para uma chave e depois fazem uma agregação nessa chave durante a fase de redução. Dê uma olhada no pseudocódigo da listagem a seguir para ter uma ideia melhor disso.

Listagem 4.4 Exemplo de pseudocódigo MapReduce

Para cada pessoa no escritório de votação:

 Rendimento (voted_party, 1)

Para cada voto no cartório: add_vote_to_party()

Uma das vantagens dos algoritmos MapReduce é que eles são fáceis de paralelizar e distribuir. Isso explica seu sucesso em ambientes distribuídos como o Hadoop, mas também podem ser usados em computadores individuais. Iremos examiná-los mais detalhadamente no próximo capítulo, e um exemplo de implementação (JavaScript) também é fornecido no capítulo 9. Ao implementar MapReduce em Python, você não precisa começar do zero. Várias bibliotecas fizeram a maior parte do trabalho para você, como Hadoopy, Octopy, Disco ou Dumbo.

4.2.2 Escolhendo a estrutura de dados correta

Algoritmos podem fazer ou quebrar seu programa, mas a maneira como você armazena seus dados é de igual importância. As estruturas de dados têm diferentes requisitos de armazenamento, mas também influenciam o desempenho do *CRUD* (criar, ler, atualizar e excluir) e outras operações no conjunto de dados.

A Figura 4.5 mostra que você tem muitas estruturas de dados diferentes para escolher, três das quais discutiremos aqui: dados esparsos, dados em árvore e dados hash. Vamos primeiro dar uma olhada nos conjuntos de dados esparsos.

DADOS ESPAROSOS

Um conjunto de dados esparsos contém relativamente pouca informação em comparação com suas entradas (observações). Veja a figura 4.6: quase tudo é “0” com apenas um único “1” presente na segunda observação da variável 9.

Dados como esses podem parecer ridículos, mas geralmente é isso que você obtém ao converter dados textuais em dados binários. Imagine um conjunto de 100.000 Twitter completamente não relacionados

árvore biológica e a maneira como ela se divide em galhos, galhos e folhas. Regras de decisão simples facilitam a localização da árvore filho na qual residem seus dados. Observe a figura 4.7 para ver como uma estrutura em árvore permite chegar rapidamente às informações relevantes.

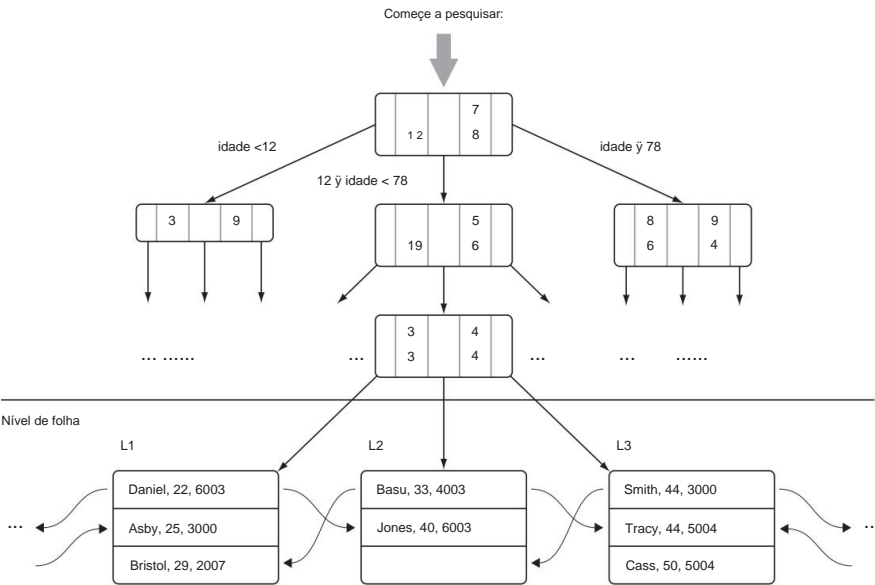


Figura 4.7 Exemplo de estrutura de dados em árvore: regras de decisão como categorias de idade podem ser usadas para localizar rapidamente uma pessoa em uma árvore genealógica

Na figura 4.7 você inicia sua busca no topo e escolhe primeiro uma categoria de idade, porque aparentemente esse é o fator que elimina o maior número de alternativas. Isso continua até você conseguir o que procura. Para quem não conhece o Akinator, recomendamos visitar <http://en.akinator.com/>. O Akinator é um djinn em uma lâmpada mágica que tenta adivinhar uma pessoa em sua mente fazendo algumas perguntas sobre ela. Experimente e surpreenda-se. . . ou veja como essa mágica é uma busca em árvore.

As árvores também são populares em bancos de dados. Os bancos de dados preferem não varrer a tabela da primeira até a última linha, mas usar um dispositivo chamado *índice* para evitar isso. Os índices geralmente são baseados em estruturas de dados, como árvores e tabelas hash, para encontrar observações mais rapidamente.

O uso de um índice acelera enormemente o processo de localização de dados. Vejamos essas tabelas hash.

TABELAS DE

HASH As tabelas de hash são estruturas de dados que calculam uma chave para cada valor em seus dados e colocam as chaves em um bucket. Dessa forma, você pode recuperar rapidamente as informações olhando no intervalo certo ao encontrar os dados. Os dicionários em Python são uma implementação de tabela hash e são parentes próximos dos armazenamentos de valores-chave. Você encontrará

no último exemplo deste capítulo, ao construir um sistema de recomendação dentro de um banco de dados. As tabelas hash são amplamente utilizadas em bancos de dados como índices para recuperação rápida de informações.

4.2.3 Selecionando as ferramentas certas

Com a classe certa de algoritmos e estruturas de dados implementadas, é hora de escolher a ferramenta certa para o trabalho. A ferramenta certa pode ser uma biblioteca Python ou pelo menos uma ferramenta controlada pelo Python, como mostra a figura 4.8. O número de ferramentas úteis disponíveis é enorme, por isso veremos apenas algumas delas.

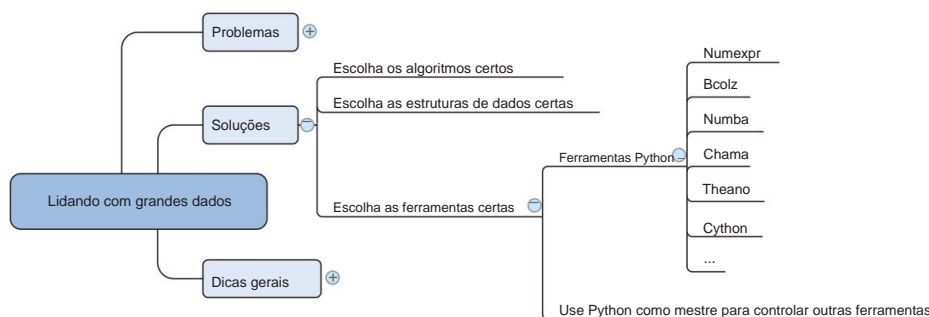


Figura 4.8 Visão geral das ferramentas que podem ser usadas ao trabalhar com grandes volumes de dados

FERRAMENTAS PITÔN

Python possui várias bibliotecas que podem ajudá-lo a lidar com grandes dados. Eles variam de estruturas de dados mais inteligentes a otimizadores de código até compiladores just-in-time. A seguir está uma lista de bibliotecas que gostamos de usar quando confrontados com grandes volumes de dados:

- *Cython* — Quanto mais próximo você estiver do hardware real de um computador, mais vital será para o computador saber quais tipos de dados ele deve processar. Para um computador, somar $1 + 1$ é diferente de somar $1,00 + 1,00$. O primeiro exemplo consiste em números inteiros e o segundo em números flutuantes, e esses cálculos são realizados por diferentes partes da CPU. Em Python você não precisa especificar quais tipos de dados está usando, então o compilador Python precisa inferi-los. Mas inferir tipos de dados é uma operação lenta e é parcialmente por isso que o Python não é uma das linguagens mais rápidas disponíveis. Cython, um superconjunto do Python, resolve esse problema forçando o programador a especificar o tipo de dados durante o desenvolvimento do programa. Depois que o compilador tiver essas informações, ele executará os programas com muito mais rapidez. Consulte <http://cython.org/> para obter mais informações sobre Cython.
- *Numexpr* — O Numexpr está no centro de muitos dos pacotes de big data, assim como o NumPy para pacotes na memória. Numexpr é um avaliador de expressões numéricas para NumPy, mas pode ser muitas vezes mais rápido que o NumPy original. Alcançar

isso, ele reescreve sua expressão e usa um compilador interno (just-in-time). Consulte <https://github.com/pydata/numexpr> para obter detalhes sobre Numexpr.

• Numba — O Numba ajuda você a obter maior velocidade compilando seu código logo antes de executá-lo, também conhecido como *compilação just-in-time*. Isso lhe dá a vantagem de escrever código de alto nível, mas atingir velocidades semelhantes às do código C. Usar o Numba é simples; consulte <http://numba.pydata.org/>. • Bcolz — Bcolz ajuda a superar o problema de falta de memória que pode ocorrer ao usar o NumPy. Ele pode armazenar e trabalhar com arrays em uma forma compactada ideal. Ele não apenas reduz a necessidade de dados, mas também usa Numexpr em segundo plano para reduzir os cálculos necessários ao realizar cálculos com matrizes bcolz. Consulte <http://bcolz.blosc.org/>.

• Blaze — O Blaze é ideal se você deseja usar o poder de um back-end de banco de dados, mas gosta do “modo Pythonic” de trabalhar com dados. Blaze traduzirá seu código Python em SQL, mas pode lidar com muito mais armazenamentos de dados do que bancos de dados relacionais, como CSV, Spark e outros. Blaze oferece uma maneira unificada de trabalhar com muitos bancos de dados e bibliotecas de dados. Porém, o Blaze ainda está em desenvolvimento e muitos recursos ainda não foram implementados. Consulte <http://blaze.readthedocs.org/en/latest/index.html>.

• Theano — Theano permite trabalhar diretamente com a unidade de processamento gráfico (GPU) e fazer simplificações simbólicas sempre que possível, e vem com um excelente compilador just-in-time. Além disso, é uma ótima biblioteca para lidar com um conceito matemático avançado, mas útil: tensores. Consulte <http://deeplearning.net/software/theano/>. • Dask — O Dask permite otimizar seu fluxo de cálculos e executá-los

com eficiência. Também permite distribuir cálculos. Consulte <http://dask.pydata.org/en/latest/>.

Essas bibliotecas tratam principalmente do uso do próprio Python para processamento de dados (além do Blaze, que também se conecta a bancos de dados). Para obter desempenho de ponta, você pode usar Python para se comunicar com todos os tipos de bancos de dados ou outros softwares.

USE PYTHON COMO MESTRE PARA CONTROLAR OUTRAS

FERRAMENTAS A maioria dos produtores de software e ferramentas oferece suporte a uma interface Python para seus softwares. Isso permite que você acesse softwares especializados com a facilidade e a produtividade que acompanham o Python. Dessa forma, o Python se diferencia de outras linguagens populares de ciência de dados, como R e SAS. Você deve aproveitar esse luxo e explorar ao máximo o poder das ferramentas especializadas. O Capítulo 6 apresenta um estudo de caso usando Python para conectar-se a um banco de dados NoSQL, assim como o Capítulo 7 com dados gráficos.

Vamos agora dar uma olhada em dicas úteis mais gerais ao lidar com grandes volumes de dados.

4.3 Dicas gerais de programação para lidar com grandes conjuntos de

dados Os truques que funcionam em um contexto geral de programação ainda se aplicam à ciência de dados. Vários podem ter palavras ligeiramente diferentes, mas os princípios são essencialmente os mesmos para todos os programadores. Esta seção recapitula os truques que são importantes no contexto da ciência de dados.

Você pode dividir os truques gerais em três partes, conforme mostrado no mapa mental da figura 4.9:

- *Não reinvente a roda*. Use ferramentas e bibliotecas desenvolvidas por terceiros.
- *Aproveite ao máximo seu hardware*. Sua máquina nunca é usada em todo o seu potencial; com adaptações simples você pode fazer com que funcione mais.
- *Reduza a necessidade de computação*. Reduza ao máximo suas necessidades de memória e processamento.

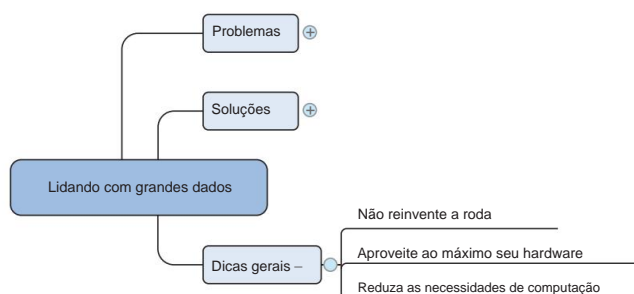


Figura 4.9 Visão geral das melhores práticas gerais de programação ao trabalhar com grandes volumes de dados

“Não reinventar a roda” é mais fácil de dizer do que fazer quando confrontado com um problema específico, mas o seu primeiro pensamento deve ser sempre: 'Alguém deve ter enfrentado este mesmo problema antes de mim.'

4.3.1 Não reinvente a roda

“Não repita ninguém” é provavelmente ainda melhor do que “não se repita”. Agregue valor às suas ações: certifique-se de que elas são importantes. Resolver um problema que já foi resolvido é perda de tempo. Como cientista de dados, você tem duas regras importantes que podem ajudá-lo a lidar com grandes dados e torná-lo muito mais produtivo:

- *Explorar o poder dos bancos de dados*. A primeira reação que a maioria dos cientistas de dados tem ao trabalhar com grandes conjuntos de dados é preparar suas tabelas base analíticas dentro de um banco de dados. Este método funciona bem quando os recursos que você deseja preparar são bastante simples. Quando esta preparação envolver modelagem avançada, descubra se é possível empregar funções e procedimentos definidos pelo usuário. O último exemplo deste capítulo trata da integração de um banco de dados ao seu fluxo de trabalho.
- *Use bibliotecas otimizadas*. A criação de bibliotecas como Mahout, Weka e outros algoritmos de aprendizado de máquina requer tempo e conhecimento. Eles são altamente otimizados

e incorporar as melhores práticas e tecnologias de ponta. Gaste seu tempo fazendo as coisas, não reinventando e repetindo os esforços de outras pessoas, a menos que seja para entender como as coisas funcionam.

Então você deve considerar sua limitação de hardware.

4.3.2 Aproveite ao máximo seu hardware

Os recursos de um computador podem ficar ociosos, enquanto outros recursos são utilizados em excesso. Isso retarda os programas e pode até mesmo fazê-los falhar. Às vezes é possível (e necessário) transferir a carga de trabalho de um recurso sobrecarregado para um recurso subutilizado usando as seguintes técnicas:

• *Alimentar a CPU com dados*

compactados. Um truque simples para evitar a falta de CPU é alimentar os dados compactados da CPU em vez dos dados inflados (brutos). Isso transferirá mais trabalho do disco rígido para a CPU, que é exatamente o que você deseja fazer, porque um disco rígido não consegue acompanhar a CPU na maioria das arquiteturas de computadores modernas. • *Faça uso da GPU*. Às vezes, sua CPU e não sua memória é o gargalo. Se seus cálculos forem paralelizáveis, você poderá se beneficiar ao mudar para a GPU. Isso tem um rendimento muito maior para cálculos do que uma CPU.

A GPU é extremamente eficiente em trabalhos paralelizáveis, mas possui menos cache que a CPU. Mas é inútil mudar para a GPU quando o problema é o disco rígido. Vários pacotes Python, como Theano e NumbaPro, usarão a GPU sem muito esforço de programação. Se isso não for suficiente, você pode usar um pacote CUDA (Compute Unified Device Architecture), como PyCUDA.

Também é um truque bem conhecido na mineração de bitcoin, se você estiver interessado em criar seu próprio dinheiro.

• *Use vários threads*. Ainda é possível paralelizar cálculos em sua CPU.

Você pode conseguir isso com threads Python normais.

4.3.3 Reduza suas necessidades de computação

“Trabalhar de forma inteligente + árduo = realização”. Isso também se aplica aos programas que você escreve. A melhor maneira de evitar grandes problemas com dados é remover o máximo possível do trabalho antecipadamente e deixar o computador trabalhar apenas na parte que não pode ser ignorada. A lista a seguir contém métodos para ajudá-lo a conseguir isso:

• *Crie o perfil do seu código e corrija partes lentas do código*. Nem todas as partes do seu código precisam ser otimizadas; use um criador de perfil para detectar partes lentas dentro do seu programa e corrigir essas partes.

• *Use código compilado sempre que possível, especialmente quando houver loops envolvidos*. Sempre que possível, use funções de pacotes otimizados para cálculos numéricos, em vez de implementar tudo sozinho. O código nesses pacotes costuma ser altamente otimizado e compilado.

• *Caso contrário, compile o código você mesmo*. Se você não puder usar um pacote existente, use um compilador just-in-time ou implemente as partes mais lentas do seu código em um

linguagem de nível inferior, como C ou Fortran, e integre-a à sua base de código.

Se você migrar para *linguagens de nível inferior* (linguagens mais próximas do bytecode universal do computador), aprenda a trabalhar com bibliotecas computacionais como LAPACK, BLAST, Intel MKL e ATLAS. Eles são altamente otimizados e é difícil obter desempenho semelhante a eles.

• *Evite colocar dados na memória.* Ao trabalhar com dados que não cabem na sua memória, evite colocar tudo na memória. Uma maneira simples de fazer isso é ler os dados em partes e analisá-los dinamicamente. Isso não funcionará em todos os algoritmos, mas permite cálculos em conjuntos de dados extremamente grandes. • *Use geradores para evitar armazenamento intermediário de dados.* Os geradores ajudam você a retornar dados por observação, em vez de em lotes. Dessa forma você evita armazenar resultados intermediários.

• *Use o mínimo de dados possível.* Se nenhum algoritmo em grande escala estiver disponível e você não estiver disposto a implementar tal técnica sozinho, ainda poderá treinar seus dados apenas em uma amostra dos dados originais.

• *Use suas habilidades matemáticas para simplificar os cálculos tanto quanto possível.* Tomemos a seguinte equação, por exemplo: $(a + b)^2 = a^2 + 2ab + b^2$. O lado esquerdo será calculado muito mais rápido que o lado direito da equação; mesmo para este exemplo trivial, pode fazer diferença quando se trata de grandes blocos de dados.

4.4 Estudo de caso 1: Predição de URLs maliciosos A Internet é

provavelmente uma das maiores invenções dos tempos modernos. Impulsionou o desenvolvimento da humanidade, mas nem todos usam esta grande invenção com intenções honrosas. Muitas empresas (Google, por exemplo) tentam nos proteger contra fraudes detectando sites maliciosos para nós. Fazer isso não é uma tarefa fácil, porque a Internet tem bilhões de páginas para digitalizar. Neste estudo de caso mostraremos como trabalhar com um conjunto de dados que não cabe mais na memória.

O que usaremos

• *Dados*—Os dados deste estudo de caso foram disponibilizados como parte de um projeto de pesquisa.

O projeto contém dados de 120 dias e cada observação possui aproximadamente 3.200.000 características. A variável de destino contém 1 se for um site malicioso e -1 caso contrário. Para obter mais informações, consulte “Além das listas negras: aprendendo a detectar sites maliciosos a partir de URLs suspeitos”.²

• *A biblioteca Scikit-learn* – você deve ter esta biblioteca instalada em seu ambiente Python neste momento, porque a usamos no capítulo anterior.

Como você pode ver, não precisaremos de muito neste caso, então vamos nos aprofundar nisso.

² Justin Ma, Lawrence K. Saul, Stefan Savage e Geoffrey M. Voelker, “Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs”, Anais da Conferência ACM SIGKDD, Paris (junho de 2009), 1245–53.

4.4.1 Passo 1: Definição do objetivo da pesquisa

O objetivo do nosso projeto é detectar se determinadas URLs são confiáveis ou não.

Como os dados são muito grandes, pretendemos fazer isso de uma forma que facilite a memória. Na próxima passo, primeiro veremos o que acontece se não nos preocuparmos com a memória (RAM).

4.4.2 Passo 2: Adquirindo os dados da URL

Comece baixando os dados de <http://sysnet.ucsd.edu/projects/url/#datasets>

e coloque-o em uma pasta. Escolha os dados no formato SVMLight. SVMLight é um software baseado em texto formato com uma observação por linha. Para economizar espaço, omite os zeros.

```
-----
MemoryError                                Traceback (most recent call last)
<ipython-input-532-d196c05088ce> in <module>()
      5 print "there are %d files" % len(files)
      6 X,y = load_svmlight_file(files[0], n_features=3500000)
----> 7 X.todense()
```

Figura 4.10 Erro de memória ao tentar colocar um grande conjunto de dados na memória

A listagem a seguir e a figura 4.10 mostram o que acontece quando você tenta ler um arquivo dos 120 e crie a matriz normal como a maioria dos algoritmos espera. O `todense()` método altera os dados de um formato de arquivo especial para uma matriz normal onde cada entrada contém um valor.

Listagem 4.5 Gerando um erro de falta de memória

```
importar globo
de sklearn.datasets importar load_svmlight_file
arquivos = glob.glob('C:\Usuários\Usuário\Downloads\
url_svmlight.tar?url_svmlight\*.svm')
arquivos = glob.glob('C:\Usuários\Usuário\Downloads\
url_svmlight?url_svmlight\*.svm') print "existem %d
arquivos" % len(arquivos)
X,y = load_svmlight_file(arquivos[0], n_features=3231952)
X.todense()
```

Os dados são uma matriz grande, mas esparsa. Ao transformá-la em uma matriz densa (cada 0 é representado no arquivo), criamos um erro de falta de memória.

Aponta para arquivos (Linux).

Aponta para arquivos (Windows: o arquivo tar precisa ser descompactado primeiro).

Indicação do número de arquivos.

Carrega arquivos.

Surpresa, surpresa, temos um erro de falta de memória. Isto é, a menos que você execute este código em uma máquina enorme. Depois de alguns truques você não terá mais esses problemas de memória e detectará 97% dos sites maliciosos.

FERRAMENTAS E TÉCNICAS

Encontramos um erro de memória ao carregar um único arquivo – ainda faltam 119. Felizmente, nós temos alguns truques na manga. Vamos tentar essas técnicas ao longo do estudo de caso:

- Use uma representação esparsa de dados.
- Alimente o algoritmo com dados compactados em vez de dados brutos.
- Use um algoritmo online para fazer previsões.

Iremos nos aprofundar em cada “truque” quando começarmos a usá-lo. Agora que temos nossos dados localmente, vamos acessá-lo. A etapa 3 do processo de ciência de dados, preparação e limpeza de dados, não é necessária neste caso porque os URLs vêm pré-limpos. Precisaremos de um formulário de exploração antes de liberar nosso algoritmo de aprendizagem.

4.4.3 Etapa 4: Exploração de dados

Para ver se podemos aplicar nosso primeiro truque (representação esparsa), precisamos descobrir se os dados realmente contêm muitos zeros. Podemos verificar isso com o seguinte

lendo um pedaço de código:

```
imprimir "número de entradas diferentes de zero" % 2.6f" % float((X.nnz)/(float(X.shape[0]) * float(X.shape[1])))
```

Isso gera o seguinte:

```
número de entradas diferentes de zero 0,000033
```

Dados que contêm pouca informação em comparação com zeros são chamados de *dados esparsos*. Isso pode ser salvo de forma mais compacta se você armazenar os dados como `[(0,0,1),(4,4,1)]` em vez de

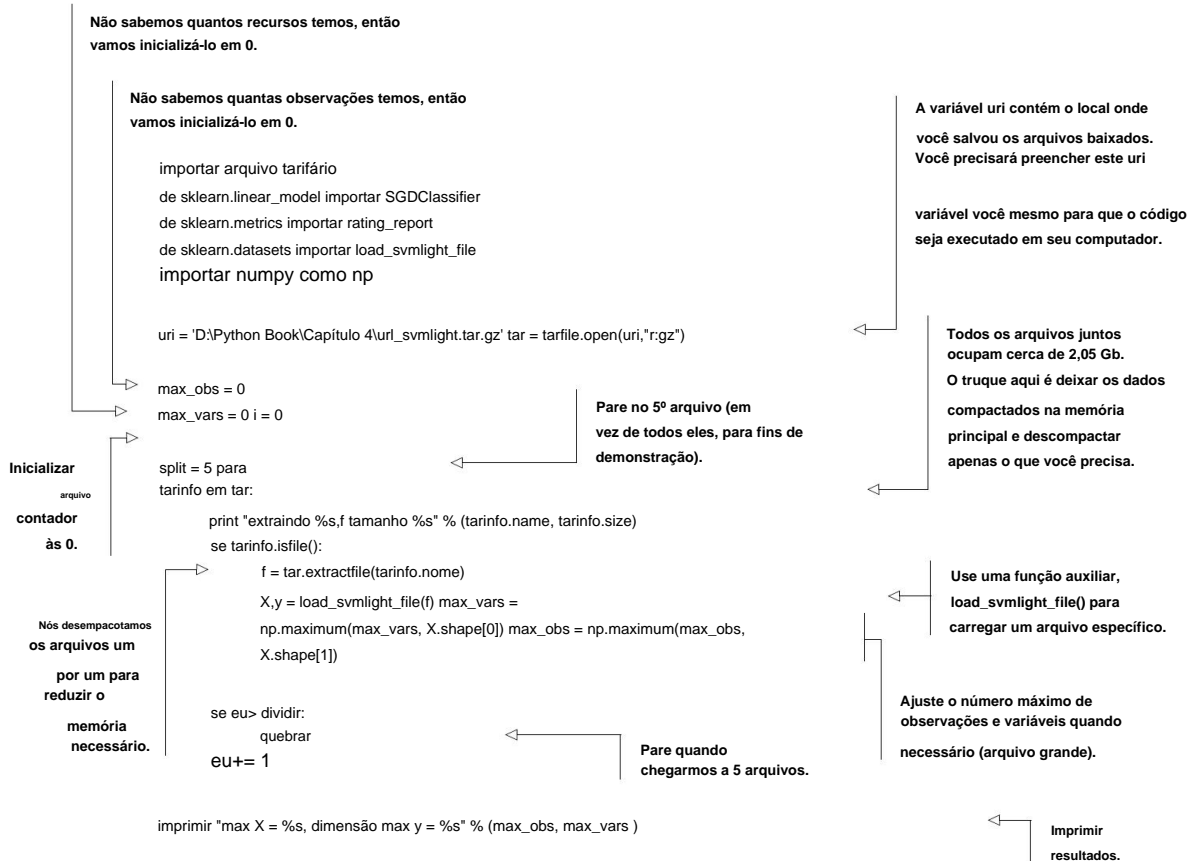
```
[[1,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,1]]
```

Um dos formatos de arquivo que implementa isso é o `SVMLight`, e é exatamente por isso que baixou os dados neste formato. Ainda não terminamos, porque precisamos para ter uma ideia das dimensões dos dados.

Para obter esta informação já precisamos manter os dados comprimidos enquanto verificamos o número máximo de observações e variáveis. Também precisamos *ler* *arquivo de dados por arquivo*. Dessa forma você consome ainda menos memória. Um segundo truque é alimentar o Arquivos compactados pela CPU. No nosso exemplo, já está compactado no formato `tar.gz`. Você descompacte um arquivo somente quando precisar dele, sem gravá-lo no disco rígido (o mais lento parte do seu computador).

Para nosso exemplo, mostrado na listagem 4.6, trabalharemos apenas nos primeiros 5 arquivos, mas parece livre para usar todos eles.

Listagem 4.6 Verificando o tamanho dos dados

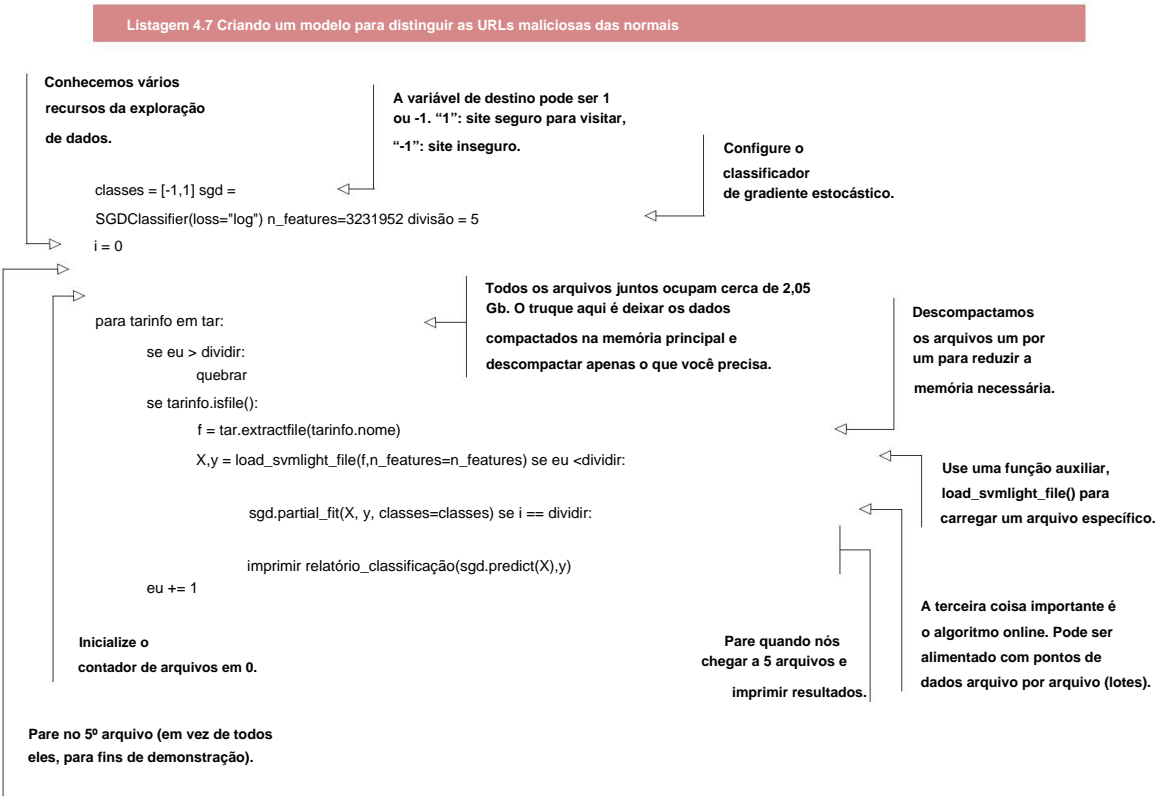


Parte do código precisa de alguma explicação extra. Neste código, percorremos o svm arquivos dentro do arquivo tar. Descompactamos os arquivos um por um para reduzir a memória necessário. Como esses arquivos estão no formato SVM , usamos um auxiliar, `functionload_svmlight_file()` para carregar um arquivo específico. Então podemos ver quantas observações e variáveis o arquivo verificando a forma do conjunto de dados resultante.

Munidos dessas informações, podemos prosseguir para a construção do modelo.

4.4.4 Passo 5: Construção do modelo

Agora que estamos cientes das dimensões dos nossos dados, podemos aplicar os mesmos dois truques (representação esparsa do arquivo compactado) e adicione o terceiro (usando um algoritmo online), na listagem a seguir. Vamos encontrar esses sites prejudiciais!



O código na listagem anterior é bastante semelhante ao que fizemos antes, exceto o classificador estocástico de descida gradiente SGDClassifier().

Aqui, treinamos o algoritmo iterativamente, apresentando as observações em um arquivo com a função parcial_fit() .

Percorrer apenas os primeiros 5 arquivos aqui fornece a saída mostrada na tabela 4.1. O a tabela mostra medidas de diagnóstico de classificação: precisão, recall, pontuação F1 e suporte.

Tabela 4.1 Problema de classificação: Um site pode ser confiável ou não?

	precisão	lembrar	pontuação f1	apoiar
-1	0,97	0,99	0,98	14045
1	0,97	0,94	0,96	5955
média/total	0,97	0,97	0,97	20.000

Apenas 3% (1 - 0,97) dos sites maliciosos não são detectados (precisão) e 6% (1 - 0,94) dos sites detectados são falsamente acusados (recall). Este é um resultado decente, portanto podemos concluir que a metodologia funciona. Se reexecutarmos a análise, o resultado poderá ser ligeiramente

diferente, porque o algoritmo poderia convergir de maneira ligeiramente diferente. Se você não se importa esperando um pouco, você pode obter o conjunto de dados completo. Agora você pode lidar com todos os dados sem problemas. Não teremos uma sexta etapa (apresentação ou automação) neste estudo de caso.

Vejam agora uma segunda aplicação de nossas técnicas; desta vez você construirá um sistema de recomendação dentro de um banco de dados. Para um exemplo bem conhecido de recomendador sistemas, visite o site da Amazon. Ao navegar, você logo se deparará com recomendações: “Quem comprou este produto também comprou...”

4.5 Estudo de caso 2: Construindo um sistema de recomendação dentro de um banco de dados

Na realidade, a maioria dos dados com os quais você trabalha são armazenados em um banco de dados relacional, mas a maioria bancos de dados não são adequados para mineração de dados. Mas, como mostrado neste exemplo, é possível adaptar nossas técnicas para que você possa fazer grande parte da análise dentro do banco de dados em si, aproveitando assim o otimizador de consultas do banco de dados, que otimizará o código para você. Neste exemplo, veremos como usar a estrutura de dados da tabela hash e como usar Python para controlar outras ferramentas.

4.5.1 Ferramentas e técnicas necessárias

Antes de entrar no estudo de caso, precisamos dar uma olhada rápida nas ferramentas necessárias e base teórica para o que estamos prestes a fazer aqui.

FERRAMENTAS

• Banco de dados *MySQL* – *precisa* de um banco de dados MySQL para trabalhar. Se você não instalou um servidor comunitário MySQL, você pode baixar um em www.mysql.com.
 Apêndice C: “Instalando um servidor MySQL” explica como configurá-lo.

• *Biblioteca Python de conexão com banco de dados MySQL* — Para conectar-se a este servidor a partir do Python você também precisará instalar o SQLAlchemy ou outra biblioteca capaz de se comunicar com o MySQL. Estamos usando MySQLdb. No Windows você não pode usar o Conda corretamente imediatamente para instalá-lo. Primeiro instale o Binstar (outro serviço de gerenciamento de pacotes) e procure o pacote mysql-python apropriado para sua configuração do Python.

```
conda instalar binstar
pesquisa binstar -t conda mysql-python
```

O seguinte comando inserido na linha de comando do Windows funcionou para nós (após ativar o ambiente Python):

```
instalação conda --canal https://conda.binstar.org/krisvanneste mysql-python
```

Novamente, sinta-se à vontade para usar a biblioteca SQLAlchemy se isso for algo com o qual você se sinta mais confortável.

• Também precisaremos da biblioteca *pandas* python, mas ela já deve estar instalada por enquanto.

Com a infraestrutura instalada, vamos mergulhar em algumas das técnicas.

TÉCNICAS

Um sistema de recomendação simples procurará clientes que alugaram filmes semelhantes aos você assistiu e depois sugere aqueles que os outros assistiram, mas você não viu ainda. Essa técnica é chamada de *k-vizinhos mais próximos* no aprendizado de máquina.

Um cliente que se comporta de maneira semelhante a você não é necessariamente o cliente mais semelhante. Você usará uma técnica para garantir que encontrará clientes semelhantes (ótimos locais) sem garantias de que você encontrou o melhor cliente (ótimo global). Uma técnica comum usada para resolver isso é chamada de *Hashing Sensível à Localidade*. Uma boa visão geral artigos sobre este tópico podem ser encontrados em <http://www.mit.edu/~andoni/LSH/>.

A ideia por trás do hash sensível à localidade é simples: construir funções que mapeie clientes semelhantes próximos uns dos outros (eles são colocados em um intervalo com o mesmo rótulo) e certifique-se de que objetos diferentes sejam colocados em baldes diferentes.

Central para esta ideia é uma função que realiza o mapeamento. Esta função é chamada de função hash: uma função que mapeia qualquer intervalo de entrada para uma saída fixa. O a função hash mais simples concatena os valores de várias colunas aleatórias. Isto não importa quantas colunas (entrada escalável); ele o traz de volta para uma única coluna (saída fixa).

Você configurará três funções hash para encontrar clientes semelhantes. As três funções pegue os valores de três filmes:

- A primeira função assume os valores dos filmes 10, 15 e 28.
- A segunda função assume os valores dos filmes 7, 18 e 22.
- A última função assume os valores dos filmes 16, 19 e 30.

Isso garantirá que os clientes que estão no mesmo intervalo compartilhem pelo menos vários filmes. Mas os clientes dentro de um mesmo grupo ainda podem diferir nos filmes que não foram incluídos nas funções de hash. Para resolver isso você ainda precisa comparar o clientes dentro do intervalo entre si. Para isso você precisa criar um novo dis-
medida de resistência.

A distância que você usará para comparar clientes é chamada de distância de Hamming. A distância de hamming é usada para calcular a diferença entre duas cordas. A distancia é definido como o número de caracteres diferentes em uma string. A Tabela 4.2 oferece alguns exemplos da distância de Hamming.

Tabela 4.2 Exemplos de cálculo da distância de hamming

Sequência 1	Corda 2	Distância de Hamming
Tem	Gato	1
Tem	Louco	2
Tigre	tigre	2
Paris	Roma	5

Comparar múltiplas colunas é uma operação cara, então você precisará de um truque para acelerar isso. Como as colunas contêm uma variável binária (0 ou 1) para indicar independentemente de o cliente ter comprado um filme ou não, você pode concatenar as informações para que a mesma informação esteja contida em uma nova coluna. A Tabela 4.3 mostra o “movimento” variável que contém tantas informações quanto todas as colunas do filme combinadas.

Tabela 4.3 Combinando as informações de diferentes colunas na coluna de filmes. É assim também
O DNA funciona: todas as informações em uma longa sequência.

Coluna 1	Filme 1	Filme 2	Filme 3	Filme 4	filmes
Cliente 1	1	0	1	1	1011
Cliente 2	0	0	0	1	0001

Isso permite calcular a distância de hamming com muito mais eficiência. Ao manusear este operador um pouco, você pode explorar o operador XOR . O resultado do XOR operador (^) é o seguinte:

$1 \wedge 1 = 0$
 $1 \wedge 0 = 1$
 $0 \wedge 1 = 1$
 $0 \wedge 0 = 0$

Com isso implementado, o processo para encontrar clientes semelhantes torna-se muito simples. Vamos primeiro olhe para ele em pseudocódigo:

Pré-processando:

- 1 Defina *p* (por exemplo, 3) funções que selecionam *k* (por exemplo, 3) entradas de o vetor de filmes. Aqui pegamos 3 funções (*p*), cada uma com 3 (*k*) filmes.
- 2 Aplique essas funções a cada ponto e armazene-as em uma coluna separada. (Em literatura, cada função é chamada de função hash e cada coluna armazenará um balde.)

Ponto de consulta q:

- 1 Aplique as mesmas funções *p* ao ponto (observação) *q* que deseja consultar.
- 2 Recupere para cada função os pontos que correspondem ao resultado no intervalo correspondente.

Pare quando você recuperar todos os pontos nos baldes ou atingir 2*p* pontos (por exemplo 10 se você tiver 5 funções).

- 3 Calcule a distância para cada ponto e retorne os pontos com o mini-distância mãe.

Vejamos uma implementação real em Python para deixar tudo mais claro.

4.5.2 Etapa 1: Questão de pesquisa

Digamos que você esteja trabalhando em uma locadora e o gerente lhe pergunte se é possível usar as informações sobre quais filmes as pessoas alugam para prever quais outros filmes elas Talvez goste. Seu chefe armazenou os dados em um banco de dados MySQL e cabe a você fazer isso a análise. Ele está se referindo a um sistema de recomendação, um sistema automatizado que conhece as preferências das pessoas e recomenda filmes e outros produtos que os clientes ainda não experimentaram. O objetivo do nosso estudo de caso é criar um sistema de recomendação amigável à memória. Conseguiremos isso usando um banco de dados e alguns truques extras. Eram vamos criar nós mesmos os dados para este estudo de caso para que possamos pular a recuperação de dados passo e vá direto para a preparação de dados. E depois disso podemos pular a etapa de exploração de dados e passar direto para a construção do modelo.

4.5.3 Etapa 3: Preparação dos dados

Os dados que seu chefe coletou são mostrados na tabela 4.4. Nós mesmos criaremos esses dados por uma questão de demonstração.

Tabela 4.4 Trecho da base de dados de clientes e dos filmes alugados pelos clientes

Cliente	Filme 1	Filme 2	Filme 3	...	Filme 32
Jack Daniel	1	0	0		1
Guilherme	1	1	0		1
...					
Jane Dane	0	0	1		0
Xi Liu	0	0	0		1
Eros Mazo	1	1	0		1
...					

Para cada cliente você recebe uma indicação se ele já alugou o filme antes

(1) ou não (0). Vamos ver o que mais você precisa para dar ao seu chefe o sistema de recomendação que ele deseja.

Primeiro vamos conectar o Python ao MySQL para criar nossos dados. Faça uma conexão com MySQL usando seu nome de usuário e senha. Na listagem a seguir usamos um banco de dados chamado de "teste". Substitua o usuário, a senha e o nome do banco de dados pelos valores apropriados para sua configuração e recupere a conexão e o cursor. Um cursor de banco de dados é um estrutura de controle que lembra onde você está atualmente no banco de dados.

Listagem 4.8 Criando clientes no banco de dados

```

importar MySQLdb
importar pandas como pd

usuário = "*****"
senha = banco "*****"
de dados = 'teste'

mc = MySQLdb.connect("localhost",usuário,senha,banco de dados) cursor = mc.cursor()

```

Primeiro estabelecemos a conexão; você precisará preencher seu próprio nome de usuário, senha e nome do esquema (variável "banco de dados").

```

nr_clientes = 100
colnames = ["movie%d" %i para i no intervalo(1,33)]
pd.np.random.seed(2015)
clientes_gerados = pd.np.random.randint(0,2,32 *
nr_customers).reshape(nr_customers,32)

```

A seguir simulamos um banco de dados com clientes e criamos algumas observações.

```

dados = pd.DataFrame (generated_customers, colunas = lista (nomes de colunas)) data.to_sql ('cust', mc, sabor
= 'mysql', índice = True, if_exists =
'substituir', index_label = 'cust_id')

```

Armazene os dados dentro de um data frame do Pandas e escreva o quadro de dados em uma tabela MySQL chamada "custo". Se esta tabela já existir, substitua-a.

Criamos 100 clientes e atribuímos aleatoriamente se eles viram ou não um determinado filme, e temos 32 filmes no total. Os dados são criados primeiro em um data frame do Pandas mas é então transformado em código SQL. Nota: você pode encontrar um aviso ao executar este código. O aviso afirma: *O sabor "mysql" com conexão DBAPI está obsoleto e será removido em versões futuras. O MySQL terá suporte adicional com mecanismos SQLAlchemy.* Sinta-se à vontade para mudar para SQLAlchemy ou outra biblioteca. Usaremos SQLAlchemy em outros capítulos, mas usei MySQLdb aqui para ampliar os exemplos.

Para consultar nosso banco de dados com eficiência posteriormente, precisaremos de preparação adicional de dados, incluindo as seguintes coisas:

- Criação de sequências de bits. As cadeias de bits são versões compactadas das colunas conteúdo (valores 0 e 1). Primeiro estes valores binários são concatenados; então o a sequência de bits resultante é reinterpretada como um número. Isso pode parecer abstrato agora mas ficará mais claro no código.
- Definição de funções hash. As funções hash criam de fato as cadeias de bits.
- Adicionar um índice à tabela para acelerar a recuperação de dados.

CRIANDO BIT STRINGS

Agora você cria uma tabela intermediária adequada para consulta, aplica as funções hash, e representar a sequência de bits como um número decimal. Finalmente, você pode colocá-los em uma mesa.

Primeiro, você precisa criar sequências de bits. Você precisa converter a string "11111111" para um binário ou um valor numérico para fazer a função hamming funcionar. Optamos por um número representação, conforme mostrado na próxima listagem.

Listagem 4.9 Criando sequências de bits

Representamos a string como um valor numérico. A string será uma concatenação de zeros (0) e uns (1) porque indicam se alguém viu um determinado filme ou não. As strings são então consideradas como código de bits. Por exemplo: 0011 é igual ao número 3. O que def createNum() faz: recebe 8 valores, concatena esses 8 valores de coluna e os transforma em uma string, então transforma o código de bytes da string em um número.

```
def createNum(x1,x2,x3,x4,x5,x6,x7,x8): return
    [int('%d%d%d%d%d%d%d%d' % (i1,i2,i3, i4,i5,i6,i7,i8),2)
    para (i1,i2,i3,i4,i5,i6,i7,i8) em zip(x1,x2,x3,x4,x5,x6,x7,x8)]
```

```
afirmar int('1111',2) == 15
afirmar int('1100',2) == 12
afirmar criarNum([1,1],[1,1],[1,1],[1,1],[1,1],[1,0],[1,0]) == [255.252]
```

```
armazenar=pd.DataFrame()
store['bit1'] = createNum(data.movie1,
    dados.movie2, dados.movie3, dados.movie4, dados.movie5,
    dados.movie6, dados.movie7, dados.movie8)
store['bit2'] = createNum(data.movie9,
    dados.movie10, dados.movie11, dados.movie12, dados.movie13,
    dados.movie14, dados.movie15, dados.movie16)
store['bit3'] = createNum(data.movie17,
    dados.movie18, dados.movie19, dados.movie20, dados.movie21,
    dados.movie22, dados.movie23, dados.movie24)
store['bit4'] = createNum(data.movie25,
    dados.movie26, dados.movie27, dados.movie28, dados.movie29,
    dados.movie30, dados.movie31, dados.movie32)
```

Teste se a função funciona corretamente. O código binário 1111 é igual a 15 (=1*8+1*4+1*2+1*1). Se a afirmação falhar, ocorrerá um erro de afirmação; caso contrário, nada acontecerá.

Traduza a
coluna do filme
em strings de 4
bits em formato numérico.
Cada sequência
de bits
representa 8 filmes.
4*8 = 32 filmes.
Nota: você pode
usar uma string
de 32 bits em vez
de 4*8 para manter o código curto.

Ao converter as informações de 32 colunas em 4 números, nós as compactamos para pesquisa posterior. A Figura 4.11 mostra o que obtemos quando solicitamos as 2 primeiras observações (histórico de visualização de filmes do cliente) neste novo formato.

```
armazenar[0:2]
```

O próximo passo é criar as funções hash, porque elas nos permitirão provar o dados que usaremos para determinar se dois clientes têm comportamento semelhante.

	bit1	bit2	bit3	bit4
0	10	62	42	182
1	23	28	223	180

Figura 4.11 Informações dos dois primeiros clientes sobre todos os 32 filmes após a conversão de sequência de bits para numérico

CRIANDO UMA FUNÇÃO HASH

As funções hash que criamos assumem os valores dos filmes para um cliente. Decidimos em a parte teórica deste estudo de caso para criar 3 funções hash: a primeira função combina os filmes 10, 5 e 18; o segundo combina os filmes 7, 18 e 22; e a o terceiro combina 16, 19 e 30. Depende de você escolher outros; isso pode ser escolhido aleatoriamente. A listagem de código a seguir mostra como isso é feito.

Listagem 4.10 Criando funções hash

```
def hash_fn(x1,x2,x3):  
    retornar [b'%d%d%d'% (i,j,k) para (i,j,k) em zip(x1,x2,x3)]  
  
afirmar hash_fn([1,0],[1,1],[0,0]) == [b'110',b'010']  
  
armazenar['bucket1'] = hash_fn(data.movie10, data.movie15,data.movie28)  
armazenar['bucket2'] = hash_fn(data.movie7, data.movie18,data.movie22)  
armazenar['bucket3'] = hash_fn(data.movie16, data.movie19,data.movie30)  
store.to_sql('movie_comparison',mc, sabor = 'mysql', index = True,  
             index_label = 'cust_id', if_exists = 'substituir')
```

Defina a função hash (é exatamente como a função createNum() sem a conversão final para um número e para 3 colunas em vez de 8).

Teste se funciona corretamente (se nenhum erro for gerado, funciona). É uma amostragem em colunas, mas todas as observações serão selecionadas.

Armazene essas informações no banco de dados.

Crie valores hash do cliente filmes, respectivamente [10,15, 28], [7,18, 22], [16,19, 30].

A função hash concatena os valores dos diferentes filmes em um binário valor como o que aconteceu antes na função createNum() , só que desta vez não convertemos para números e pegamos apenas 3 filmes em vez de 8 como entrada. A função de afirmação mostra como concatena os 3 valores para cada observação. Quando o cliente tiver comprou o filme 10, mas não os filmes 15 e 28, ele retornará b'100' para o intervalo 1. Quando o cliente comprou os filmes 7 e 18, mas não 22, ele retornará b'110' para o intervalo 2. Se olharmos o resultado atual vemos as 4 variáveis que criamos anteriormente (bit1, bit2, bit3, bit4) dos 9 filmes escolhidos a dedo (figura 4.12).

	bit1	bit2	bit3	bit4	bucket1	bucket2	bucket3
0	10	62	42	182	011	100	011
1	23	28	223	180	001	111	001

Figura 4.12 Informações da compressão da sequência de bits e dos 9 filmes amostrados

O último truque que aplicaremos é indexar a tabela de clientes para que as pesquisas aconteçam mais rapidamente.

ADICIONANDO UM ÍNDICE À TABELA

Agora você deve adicionar índices para acelerar a recuperação conforme necessário em um sistema em tempo real. Isso é mostrado na próxima listagem.

Listagem 4.11 Criando um índice

```
def createIndex(coluna, cursor):
    sql = 'CRIAR ÍNDICE %s ON movie_comparison (%s);' % (coluna, coluna)
    cursor.execute(sql)

createIndex('bucket1', cursor)
createIndex('bucket2', cursor)
createIndex('bucket3', cursor)
```

Crie função para criar índices facilmente. Os índices irão acelerar a recuperação.

Coloque índice em baldes de bits.

Com os dados indexados podemos agora passar para a “parte de construção do modelo”. Nesse caso estudo, nenhum aprendizado de máquina real ou modelo estatístico é implementado. Em vez disso, usaremos um técnica muito mais simples: cálculo da distância da string. Duas strings podem ser comparadas usando a distância de Hamming conforme explicado anteriormente na introdução teórica do estudo de caso.

4.5.4 Etapa 5: Construção do modelo

Para usar a distância de Hamming no banco de dados precisamos defini-la como uma função.

CRIANDO A FUNÇÃO DE DISTÂNCIA DE HAMMING

Implementamos isso como uma função definida pelo usuário. Esta função pode calcular a distância para um número inteiro de 32 bits (na verdade 4*8), conforme mostrado na listagem a seguir.

Listagem 4.12 Criando a distância de impacto

```
SQL = """
CRIAR FUNÇÃO HAMMINGDISTANCE(
    A0 BIGINT, A1 BIGINT, A2 BIGINT, A3 BIGINT,
    B0 BIGINT, B1 BIGINT, B2 BIGINT, B3 BIGINT
)

RETORNOS INT DETERMINÍSTICOS
RETORNAR
    BIT_COUNT(A0 ^ B0) +
    BIT_COUNT(A1 ^ B1) +
    BIT_COUNT(A2 ^ B2) +
    BIT_COUNT(A3 ^ B3); """

cursor.execute(Sql)

Sql = """Selecione hammingdistance(
    b'11111111',b'00000000',b'11011111',b'11111111'
    ,b'11111111',b'10001001',b'11011111',b'11111111'
)"""pd.read_sql(Sql,mc)
```

A função é armazenada em um banco de dados. Você só pode fazer isso uma vez; executar este código uma segunda vez resultará em um erro: OperationalError: (1304, 'FUNCTION HAMMING-DISTANCE já existe')

Definir função. São necessários 8 argumentos de entrada: 4 strings de comprimento 8 para o primeiro cliente e outras 4 strings de comprimento 8 para o segundo cliente. Desta forma podemos comparar 2 clientes lado a lado para 32 filmes.

Para verificar esta função você pode executar esta instrução SQL com 8 strings fixas. Observe o “b” antes de cada string, indicando que você está passando valores de bits. O resultado deste teste específico deve ser 3, o que indica que a série de strings difere em apenas 3 lugares.

Esse corre o consulta.

Se tudo estiver bem, a saída deste código deverá ser 3.

Agora que temos nossa função de distância de Hamming em posição, podemos usá-la para encontrar clientes semelhantes a um determinado cliente, e é exatamente isso que queremos que nosso aplicativo pendência. Vamos para a última parte: utilizar nosso setup como uma espécie de aplicativo.

4.5.5 Passo 6: Apresentação e automação

Agora que temos tudo configurado, nossa aplicação precisa realizar duas etapas ao se deparar com um determinado cliente:

- Procure clientes semelhantes.
- Sugira filmes que o cliente ainda não viu com base no que ele já assistiu visualizados e o histórico de visualização de clientes semelhantes.

Comecemos pelo princípio: selecione um cliente sortudo.

ENCONTRANDO UM CLIENTE SEMELHANTE

É hora de realizar consultas em tempo real. Na listagem a seguir, o cliente 27 é o feliz aquele que terá seus próximos filmes selecionados para ele. Mas primeiro precisamos selecionar os clientes com um histórico de visualização semelhante.

Listagem 4.13 Encontrando clientes semelhantes

Fazemos amostragem em duas etapas. Primeira amostragem: o índice deve ser exatamente igual ao do cliente selecionado (é baseado em 9 filmes). As pessoas selecionadas devem ter visto (ou não) esses 9 filmes exatamente como nosso cliente viu.

A segunda amostragem é uma classificação baseada nas strings de 4 bits. Eles levam em consideração todos os filmes do banco de dados.

Escolha o cliente do banco de dados.

```
ID_do_cliente = 27
sql = "selecione * de movie_comparison onde cust_id = %s" % customer_id
cust_data = pd.read_sql(sql,mc)

sql = """ selecione cust_id,hammingdistance(bit1,
bit2,bit3,bit4,%s,%s,%s,%s) como distância
de movie_comparison onde bucket1 = '%s' ou bucket2 = '%s' ou bucket3='%s' ordena por limite
de distância 3""" % (cust_data.bit1[0],cust_data.bit2[0],
cust_data.bit3[0], cust_data.bit4[0],
cust_data.bucket1[0], cust_data.bucket2[0],cust_data.bucket3[0])
lista = pd.read_sql(sql,mc)
```

Mostramos os 3 clientes que mais se assemelham ao cliente 27.

O cliente 27 termina primeiro.

A Tabela 4.5 mostra que os clientes 2 e 97 são os mais semelhantes ao cliente 27. Não se esqueça que os dados foram gerados aleatoriamente, portanto, qualquer pessoa que repita este exemplo poderá receber resultados diferentes.

Agora podemos finalmente selecionar um filme para o cliente 27 assistir.

Tabela 4.5 Os clientes mais semelhantes ao cliente 27

	ID_do_cliente	distância
0	27	0
12 8		
2	97	9

ENCONTRANDO UM NOVO FILME

Precisamos ver os filmes que o cliente 27 ainda não viu, mas o cliente mais próximo viu, conforme mostrado na listagem a seguir. Esta também é uma boa verificação para ver se a função de distância funcionou corretamente. Embora este possa não ser o cliente mais próximo, é uma boa combinar com o cliente 27. Ao usar os índices hash, você ganhou enorme velocidade ao consultar grandes bancos de dados.

Listagem 4.14 Encontrando um filme não visto

```
cust = pd.read_sql('selecione * de cust onde cust_id em (27,2,97)',mc)
dif = custo.T
dif[dif[0] != dif[1]]
```

Selecione os filmes que o cliente 27 ainda não viu.

Transponha por conveniência.

Selecione filmes clientes 27, 2, 97 viram.

A Tabela 4.6 mostra que você pode recomendar o filme 12, 15 ou 31 com base nas avaliações do cliente 2. comportamento.

Tabela 4.6 Filmes do cliente 2 podem ser usados como sugestões para o cliente 27.

	0	1	2
ID_cliente	2	27	97
Filme3	0	1	1
Filme9	0	1	1
Filme11	0	1	1
Filme12	1	0	0
Filme15	1	0	0
Filme16	0	1	1
Filme25	0	1	1
Filme31	1	0	0

Missão cumprida. Nosso feliz viciado em cinema agora pode se deliciar com um novo filme, adaptado às suas preferências.

No próximo capítulo, examinaremos dados ainda maiores e veremos como podemos lidar com isso usando o Horton Sandbox que baixamos no capítulo 1.

4.6 Resumo Este capítulo

discutiu os seguintes tópicos:

• Os principais *problemas* que você pode encontrar ao trabalhar com grandes conjuntos de dados são estes:

- Memória insuficiente
- Programas de longa duração
- Recursos que formam gargalos e causam problemas de velocidade

• Existem três tipos principais de *soluções* para estes problemas: – Adapte os seus algoritmos.

- Use diferentes estruturas de dados.
- Confie em ferramentas e bibliotecas.

• Três técnicas principais podem ser usadas para *adaptar um algoritmo*:

- Apresentar os dados do algoritmo, *uma observação por vez*, em vez de carregar os dados completos definidos de uma vez.
- *Divida as matrizes em matrizes menores* e use-as para fazer seus cálculos.
- Implementar o algoritmo *MapReduce* (usando bibliotecas Python como Hadoopy, Octopy, Disco ou Dumbo).

• Três *estruturas de dados* principais são usadas na ciência de dados. A primeira é um tipo de matriz que contém relativamente pouca informação, a *matriz esparsa*. A segunda e a terceira são estruturas de dados que permitem recuperar informações rapidamente em um grande conjunto de dados: a *função hash* e a *estrutura em árvore*.

• Python possui muitas *ferramentas* que podem ajudá-lo a lidar com grandes conjuntos de dados. Várias ferramentas irão ajudá-lo com o tamanho do volume, outras irão ajudá-lo a paralelizar os cálculos e outras ainda a superar a velocidade relativamente lenta do próprio Python. Também é fácil usar Python como uma ferramenta para controlar outras ferramentas de ciência de dados porque Python é frequentemente escolhido como linguagem para implementar uma API. • As *melhores práticas* da ciência da computação também são válidas no contexto da ciência de dados, portanto, aplicá-las pode ajudá-lo a superar os problemas enfrentados em um contexto de big data.