Simulação Estocástica

Thiago Rodrigo Ramos

25 de agosto de 2025

Sumário

Capítulo 1

Introdução

1.1 Um conselho: a importância de ser ruim antes de ser bom

É natural que, quando começamos a fazer algo, a gente faça essa coisa muito malfeita ou cheia de defeitos. Isso é comum em qualquer processo de aprendizagem, e sempre foi assim, desde o início dos tempos.

Quando comecei a programar em Python, muita coisa sobre a linguagem eu aprendi por conta própria, apesar de já ter feito alguns cursos básicos em C. Programei de forma amadora em Python por muitos anos, até que, no doutorado, precisei aprender a programar de forma mais organizada e profissional. Lembro que, nessa época, um amigo da pós-graduação me apresentou ao "submundo da programação". Foi aí que aprendi muito do que sei hoje sobre terminal do Linux, Git, e foi também quando comecei a usar o Vim.

Uma das coisas que esse amigo me mostrou foi o Pylint, que nada mais é do que um verificador de bugs e qualidade de código para Python. O Pylint é bem rigoroso na análise, e ainda te dá, ao final, uma nota que vai até 10. Nessa fase, apesar de já ter evoluído bastante, meus códigos ainda recebiam notas por volta de 6 ou 7. Resolvi então rodar o Pylint nos meus códigos antigos pra ter uma noção de quão ruins eles eram — e a nota final foi -900. Pois é, existe um limite superior para o quão bem você consegue fazer algo, mas aparentemente o fundo do poço é infinito.

O que eu queria mostrar com essa história é que faz parte do processo de aprendizado ser ruim no começo e melhorar com o tempo. Falo isso porque, hoje em dia, com o crescimento dos LLMs, a gente fica tentado a pular essa etapa de errar muito até acertar, e ir direto pra fase em que escrevemos códigos limpos, bem comentados, identados e organizados. Mas não se enganem: apesar da aparência profissional, depender de LLMs pra escrever tudo atrapalha justamente essa parte essencial de aprender errando.

Neste curso, vários exercícios envolvem escrever códigos em Python. Meu conselho é: não tenham vergonha de errar, de escrever soluções ruins ou confusas. Isso é absolutamente normal. Vocês estão aqui para evoluir — e errar faz parte do processo.

Capítulo 2

Elementos básicos de probabilidade

2.1 Axiomas da probabilidade

Um *espaço de probabilidade* é uma tupla composta por três elementos: o *espaço amostral*, o *conjunto de eventos* e uma *distribuição de probabilidade*:

- **Espaço amostral** Ω: Ω é o conjunto de todos os eventos elementares ou resultados possíveis de um experimento. Por exemplo, ao lançar um dado, $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- **Conjunto de eventos** $\mathcal{F}$: $\mathcal{F}$ é uma σ-álgebra, ou seja, um conjunto de subconjuntos de Ω que contém Ω e é fechado sob complementação e união enumerável (e, consequentemente, também sob interseção enumerável). Um exemplo de evento é: "o dado mostra um número ímpar".
- **Distribuição de probabilidade** $\mathbb{P}$: $\mathbb{P}$ é uma função que associa a cada evento de $\mathcal{F}$ um número em $[0, 1]$, tal que $\mathbb{P}[\Omega] = 1$, $\mathbb{P}[\varnothing] = 0$ e, para eventos mutuamente exclusivos $A_1, \ldots, A_n$, temos:

$$\mathbb{P}\left[A_1 \cup \cdots \cup A_n\right] = \sum_{i=1}^{n} \mathbb{P}[A_i].$$

A distribuição de probabilidade discreta associada ao lançamento de um dado justo pode ser definida como $\mathbb{P}[A_i] = 1/6$ para $i \in \{1, \ldots, 6\}$, onde A_i é o evento "o dado mostra o valor i".

2.1.1 Probabilidade condicional e independência

A probabilidade condicional do evento A dado o evento B é definida como a razão entre a probabilidade da interseção $A \cap B$ e a probabilidade de B, desde que $\mathbb{P}[B] \neq 0$:

$$\mathbb{P}[A \mid B] = \frac{\mathbb{P}[A \cap B]}{\mathbb{P}[B]}.$$

Dois eventos A e B são ditos independentes quando a probabilidade conjunta $\mathbb{P}[A \cap B]$ pode ser fatorada como o produto $\mathbb{P}[A]\mathbb{P}[B]$:

$$\mathbb{P}[A \cap B] = \mathbb{P}[A]\mathbb{P}[B].$$

De forma equivalente, a independência entre A e B pode ser expressa afirmando que $\mathbb{P}[A \mid B] = \mathbb{P}[A]$, sempre que $\mathbb{P}[B] \neq 0$.

Além disso, uma sequência de variáveis aleatórias é dita *i.i.d.* (independentes e identicamente distribuídas) quando todas as variáveis da sequência são mutuamente independentes e seguem a mesma distribuição de probabilidade.

Seguem algumas propriedades importantes:

$$\mathbb{P}[A \cup B] = \mathbb{P}[A] + \mathbb{P}[B] - \mathbb{P}[A \cap B] \qquad \text{(regra da soma)}$$

$$\mathbb{P}\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \mathbb{P}[A_i] \qquad \textbf{(desigualdade da união)}$$

$$\mathbb{P}[A \mid B] = \frac{\mathbb{P}[B \mid A]\mathbb{P}[A]}{\mathbb{P}[B]} \qquad \text{(fórmula de Bayes)}$$

$$\mathbb{P}\left[\bigcap_{i=1}^{n} A_i\right] = \mathbb{P}[A_1]\mathbb{P}[A_2 \mid A_1] \cdots \mathbb{P}\left[A_n \mid \bigcap_{i=1}^{n-1} A_i\right] \qquad \text{(regra da cadeia)}$$

Exercício 1. *Prove os resultados acima.*

2.2 Variáveis aleatórias

Uma *variável aleatória* X é uma função mensurável $X : \Omega \to \mathbb{R}$, ou seja, tal que, para qualquer intervalo $I \subset \mathbb{R}$, o conjunto

$$\{\omega \in \Omega : X(\omega) \in I\}$$

pertence à σ-álgebra de eventos.

No caso discreto, a *função de massa de probabilidade* de X é dada por

$$x \mapsto \mathbb{P}[X = x].$$

Quando a distribuição de X é *absolutamente contínua*, existe uma *função densidade de probabilidade* f tal que, para todo $a, b \in \mathbb{R}$,

$$\mathbb{P}[a \leq X \leq b] = \int_a^b f(x)\, dx.$$

A função f é chamada *função densidade de probabilidade* da variável aleatória X. A relação entre a *função de distribuição acumulada* $F(\cdot)$ e a densidade $f(\cdot)$ é

$$F(a) = \mathbb{P}\{X \leq a\} = \int_{-\infty}^{a} f(x)\, dx.$$

Derivando ambos os lados, obtemos

$$\frac{d}{da} F(a) = f(a),$$

ou seja, a densidade é a derivada da função de distribuição acumulada.

Uma interpretação mais intuitiva de f pode ser obtida observando que, para $\varepsilon > 0$ pequeno,

$$\mathbb{P}\left(a - \frac{\varepsilon}{2} < X < a + \frac{\varepsilon}{2}\right) = \int_{a-\varepsilon/2}^{a+\varepsilon/2} f(x)\, dx \approx \varepsilon f(a).$$

Assim, $f(a)$ quantifica a probabilidade de X assumir valores próximos de a.

Em muitos contextos, o interesse recai não apenas sobre variáveis aleatórias individuais, mas também sobre o relacionamento entre duas ou mais variáveis. Para descrever a dependência entre X e Y, definimos a *função de distribuição acumulada conjunta* como

$$F(x,y) = \mathbb{P}\{X \leq x,\, Y \leq y\},$$

que fornece a probabilidade de X ser menor ou igual a x e, simultaneamente, Y ser menor ou igual a y.

Se X e Y forem variáveis aleatórias discretas, a *função de massa de probabilidade conjunta* é

$$p(x,y) = \mathbb{P}\{X = x,\, Y = y\}.$$

Se forem *conjuntamente contínuas*, existe uma *função densidade de probabilidade conjunta* $f(x,y)$ tal que, para quaisquer conjuntos $C, D \subset \mathbb{R}$,

$$\mathbb{P}\{X \in C,\, Y \in D\} = \iint_{x \in C,\, y \in D} f(x,y)\, dx\, dy.$$

As variáveis X e Y são *independentes* se, para quaisquer $C, D \subset \mathbb{R}$,

$$\mathbb{P}\{X \in C,\, Y \in D\} = \mathbb{P}\{X \in C\}\, \mathbb{P}\{Y \in D\}.$$

De forma intuitiva, isso significa que conhecer o valor de uma das variáveis não altera a distribuição da outra.

No caso discreto, X e Y são independentes se, e somente se, para todo x, y,

$$\mathbb{P}\{X = x,\, Y = y\} = \mathbb{P}\{X = x\}\, \mathbb{P}\{Y = y\}.$$

Se forem conjuntamente contínuas, a independência é equivalente a

$$f(x,y) = f_X(x)\, f_Y(y), \quad \forall x, y,$$

onde f_X e f_Y são as densidades marginais de X e Y, respectivamente.

2.3 Valor esperado

A esperança ou valor esperado de uma variável aleatória X é denotada por $\mathbb{E}[X]$ e, no caso discreto, é definida como

$$\mathbb{E}[X] = \sum_x x\, \mathbb{P}[X = x].$$

Exemplo 1. *Se I é uma variável aleatória indicadora do evento A, isto é,*

$$I = \begin{cases} 1, & \text{se } A \text{ ocorre}, \\ 0, & \text{se } A \text{ não ocorre}, \end{cases}$$

então

$$\mathbb{E}[I] = 1 \cdot \mathbb{P}(A) + 0 \cdot \mathbb{P}(A^c) = \mathbb{P}(A).$$

Portanto, a esperança de uma variável indicadora de um evento A é exatamente a probabilidade de que A ocorra.

No caso contínuo, quando X possui uma função densidade de probabilidade $f(x)$, a esperança é dada por

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x)\, dx.$$

Além disso, dado uma função qualquer g, temos que:

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f(x)\, dx.$$

Uma propriedade fundamental da esperança é sua linearidade. Isto é, para quaisquer variáveis aleatórias X e Y e constantes $a, b \in \mathbb{R}$, temos:

$$\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y].$$

2.4 Variância

A variância de uma variável aleatória X é denotada por $\mathrm{Var}[X]$ e definida como

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2].$$

O desvio padrão de X é denotado por σ_X e definido como

$$\sigma_X = \sqrt{\mathrm{Var}[X]}.$$

Para qualquer variável aleatória X e qualquer constante $a \in \mathbb{R}$, as seguintes propriedades básicas são válidas:

$$\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2,$$

$$\mathrm{Var}[aX] = a^2 \mathrm{Var}[X].$$

Além disso, se X e Y forem independentes, então

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y].$$

Exercício 2. *Prove os resultados anteriores.*

2.4.1 Covariância

A covariância entre duas variáveis aleatórias X e Y é denotada por $\mathrm{Cov}(X, Y)$ e definida por

$$\mathrm{Cov}(X, Y) = \mathbb{E}\left[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])\right].$$

Exercício 3. *Prove que*

$$\mathrm{Cov}(X, Y) = \mathbb{E}\left[XY\right] - \mathbb{E}\left[X\right]\mathbb{E}\left[Y\right].$$

Dizemos que X e Y são *não correlacionadas* quando $\mathrm{Cov}(X, Y) = 0$. Se X e Y forem independentes, então certamente são não correlacionadas, mas a recíproca nem sempre é verdadeira.

Exercício 4. *Seja X uniforme no intervalo $[-1, 1]$ e seja $Y = X^2$. Mostre que $\mathrm{Cov}\,(X, Y) = 0$ mas X, Y não são independentes.*

Observação 1. *Considere uma variável aleatória contínua X centrada em zero, ou seja, $\mathbb{E}[X] = 0$, com densidade de probabilidade par e definida em um intervalo do tipo $(-a, a)$, com $a > 0$. Seja $Y = g(X)$ para uma função g. A questão é: para quais funções $g(X)$ temos $\mathrm{Cov}(X, g(X)) = 0$?*

Sabemos que

$$\mathrm{Cov}(X, g(X)) = \mathbb{E}[Xg(X)] - \mathbb{E}[X]\mathbb{E}[g(X)].$$

Como $\mathbb{E}[X] = 0$, segue que $\mathrm{Cov}(X, g(X)) = \mathbb{E}[Xg(X)]$. Denotando a densidade de X por $f(x)$, temos

$$\mathrm{Cov}(X, g(X)) = \int_{-a}^{a} xg(x)f(x)dx.$$

Uma maneira de garantir que $\mathrm{Cov}(X, g(X)) = 0$ é exigir que $g(x)$ seja uma função par. Assim, $xg(x)f(x)$ será uma função ímpar e a integral em $(-a, a)$ se anulará, ou seja,

$$\int_{-a}^{a} xg(x)f(x)dx = 0.$$

Portanto, $\mathrm{Cov}(X, f(X)) = 0$ e como $Y = g(X)$, teremos que ambas são dependentes.

Dessa forma, podemos concluir que a distribuição precisa de X não afeta a condição, desde que $p(x)$ seja simétrica em torno da origem. Qualquer função par $f(\cdot)$ satisfará $\mathrm{Cov}(X, f(X)) = 0$.

A covariância é uma forma bilinear simétrica e semi-definida positiva, com as seguintes propriedades:

- **Simetria**: $\mathrm{Cov}(X, Y) = \mathrm{Cov}(Y, X)$ para quaisquer variáveis X e Y.
- **Bilinearidade**: $\mathrm{Cov}(X + X', Y) = \mathrm{Cov}(X, Y) + \mathrm{Cov}(X', Y)$ e $\mathrm{Cov}(aX, Y) = a\,\mathrm{Cov}(X, Y)$ para qualquer $a \in \mathbb{R}$.
- **Semi-definida positiva**: $\mathrm{Cov}(X, X) = \mathrm{Var}[X] \geq 0$ para qualquer variável X.

Além disso, vale a desigualdade de Cauchy-Schwarz, que afirma que para variáveis X e Y com variância finita,

$$|\mathrm{Cov}(X, Y)| \leq \sqrt{\mathrm{Var}[X]\,\mathrm{Var}[Y]}.$$

Perceba a semelhança do resultado acima com a desigualdade de Cauchy-Schwarz!

Exercício 5. *Prove os resultados acima.*

A matriz de covariância de um vetor de variáveis aleatórias $\mathbf{X} = (X_1, \ldots, X_p)$ é a matriz em $\mathbb{R}^{n\times n}$ denotada por $\mathbf{C}(\mathbf{X})$ e definida por

$$\mathbf{C}(\mathbf{X}) = \mathbb{E}\left[(\mathbf{X} - \mathbb{E}[\mathbf{X}])(\mathbf{X} - \mathbb{E}[\mathbf{X}])^\top\right].$$

Portanto, $\mathbf{C}(\mathbf{X})$ é a matriz cujos elementos são $\mathrm{Cov}(X_i, X_j)$. Além disso, é imediato mostrar que

$$\mathbf{C}(\mathbf{X}) = \mathbb{E}[\mathbf{X}\mathbf{X}^\top] - \mathbb{E}[\mathbf{X}]\,\mathbb{E}[\mathbf{X}]^\top.$$

2.5 Desigualdades básicas de concentração

Nesta seção, apresentamos duas desigualdades fundamentais que estabelecem limites superiores para a probabilidade de uma variável aleatória assumir valores distantes de sua média. Tais resultados são amplamente utilizados em probabilidade, estatística e teoria da informação para analisar o comportamento de caudas de distribuições.

A primeira delas é a *Desigualdade de Markov*, que fornece um limite simples para variáveis aleatórias não-negativas em função apenas de sua esperança.

Teorema 1 (Desigualdade de Markov)**.** *Seja X uma variável aleatória não-negativa ($X \geq 0$ quase certamente) com valor esperado $\mathbb{E}[X] < \infty$. Então, para todo $t > 0$, temos:*

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}[X]}{t}.$$

Exercício 6. *Prove a desigualdade de Markov.* ***Dica:*** *use o fato de que $\frac{x}{t} \geq \mathbb{I}\{x \geq t\}$.*

A próxima desigualdade é um refinamento da anterior. Conhecida como *Desigualdade de Chebyshev*, ela aplica a desigualdade de Markov à variável aleatória $(X - \mu)^2$ e relaciona o desvio da média com a variância da distribuição.

Teorema 2 (Desigualdade de Chebyshev)**.** *Seja X uma variável aleatória com valor esperado $\mu = \mathbb{E}[X]$ e variância finita $\mathrm{Var}(X) = \sigma^2$. Então, para todo $\varepsilon > 0$, vale:*

$$\mathbb{P}(|X - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{\varepsilon^2}.$$

Exercício 7. *Prove a desigualdade de Chebyshev a partir da desigualdade de Markov aplicada a $(X - \mu)^2$.*

2.6 Teoremas assintóticos

Em muitas aplicações de probabilidade e estatística, estamos interessados no comportamento de sequências de variáveis aleatórias quando o número de observações tende ao infinito. Os *teoremas assintóticos* fornecem resultados fundamentais que descrevem como certos estimadores ou somas de variáveis aleatórias se comportam no limite, ou seja, quando o tamanho da amostra n cresce indefinidamente.

Teorema 3 (Lei Fraca dos Grandes Números)**.** *Seja $(X_n)_{n\in\mathbb{N}}$ uma sequência de variáveis aleatórias independentes, todas com a mesma esperança μ e variância $\sigma^2 < \infty$. Definindo a média amostral por*

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i,$$

então, para qualquer $\varepsilon > 0$,

$$\lim_{n\to\infty} \mathbb{P}\left(\left|\overline{X}_n - \mu\right| \geq \varepsilon\right) = 0.$$

Exercício 8. *Prove a Lei Fraca dos Grandes números utilizando a desigualdade de Chebyshev.*

Teorema 4 (Teorema Central do Limite)**.** *Seja $X_1, \ldots, X_n$ uma sequência de variáveis aleatórias i.i.d. com esperança μ, variância σ^2 e momento de ordem 3 finito. Definimos a média amostral como*

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

Então,

$$\frac{\sqrt{n}(\overline{X}_n - \mu)}{\sigma} \quad \xrightarrow{d} \quad N(0,1).$$

Demonstração. Suponha, sem perda de generalidade, que $\mu = 0$ e $\sigma = 1$. Defina

$$A_n = \frac{1}{\sqrt{n}}\sum_{i=1}^{n} X_i \quad \text{e} \quad B_n = \frac{1}{\sqrt{n}}\sum_{i=1}^{n} N_i,$$

onde $N_i \overset{\text{i.i.d.}}{\sim} N(0,1)$ independentes de tudo. Note que $B_n \sim N(0,1)$ para todo n.

Para provar que $A_n \xrightarrow{d} N(0,1)$, é suficiente mostrar que, para qualquer função de teste f suave e com crescimento controlado,

$$\mathbb{E}[f(A_n)] - \mathbb{E}[f(B_n)] \longrightarrow 0.$$

Passo 1: Construção telescópica. Considere as variáveis intermediárias

$$\begin{aligned}
C_n^{(0)} &= \tfrac{1}{\sqrt{n}}(X_1 + X_2 + \cdots + X_n),\\
C_n^{(1)} &= \tfrac{1}{\sqrt{n}}(N_1 + X_2 + \cdots + X_n),\\
C_n^{(2)} &= \tfrac{1}{\sqrt{n}}(N_1 + N_2 + X_3 + \cdots + X_n),\\
&\vdots\\
C_n^{(n)} &= \tfrac{1}{\sqrt{n}}(N_1 + N_2 + \cdots + N_n).
\end{aligned}$$

Claramente, $C_n^{(0)} = A_n$ e $C_n^{(n)} = B_n$.

Assim,

$$\begin{aligned}
\mathbb{E}[f(A_n)] - \mathbb{E}[f(B_n)] &= \mathbb{E}[f(C_n^{(0)})] - \mathbb{E}[f(C_n^{(n)})]\\
&= \sum_{k=1}^{n} \Delta_k,
\end{aligned}$$

onde

$$\Delta_k := \mathbb{E}[f(C_n^{(k-1)}) - f(C_n^{(k)})].$$

Passo 2: Isolando o termo que difere. Entre $C_n^{(k)}$ e $C_n^{(k-1)}$, o único termo diferente é o k-ésimo. Definamos

$$D_n^{(k)} = \tfrac{1}{\sqrt{n}}\big(N_1 + \cdots + N_{k-1} + 0 + X_{k+1} + \cdots + X_n\big),$$

isto é, a parte comum entre $C_n^{(k)}$ e $C_n^{(k-1)}$, mas com o k-ésimo termo anulado.

Assim,

$$C_n^{(k)} = D_n^{(k)} + \tfrac{N_k}{\sqrt{n}}, \qquad C_n^{(k-1)} = D_n^{(k)} + \tfrac{X_k}{\sqrt{n}}.$$

Portanto,

$$\Delta_k = \mathbb{E}\Big[f\Big(D_n^{(k)} + \tfrac{X_k}{\sqrt{n}}\Big) - f\Big(D_n^{(k)} + \tfrac{N_k}{\sqrt{n}}\Big)\Big].$$

Passo 3: Expansão de Taylor condicional. Fixe $D_n^{(k)} = d$. Aplicando Taylor em torno de d, temos:

$$f\Big(d + \tfrac{X_k}{\sqrt{n}}\Big) = f(d) + \tfrac{X_k}{\sqrt{n}} f'(d) + \tfrac{X_k^2}{2n} f''(d) + \tfrac{X_k^3}{6n^{3/2}} f^{(3)}(d + \xi_X),$$
$$f\Big(d + \tfrac{N_k}{\sqrt{n}}\Big) = f(d) + \tfrac{N_k}{\sqrt{n}} f'(d) + \tfrac{N_k^2}{2n} f''(d) + \tfrac{N_k^3}{6n^{3/2}} f^{(3)}(d + \xi_N),$$

para alguns ξ_X, ξ_N entre 0 e $X_k/\sqrt{n}$ ou $N_k/\sqrt{n}$.

Subtraindo,

$$f\Big(d + \tfrac{X_k}{\sqrt{n}}\Big) - f\Big(d + \tfrac{N_k}{\sqrt{n}}\Big) = \tfrac{1}{\sqrt{n}} f'(d)(X_k - N_k) + \tfrac{1}{2n} f''(d)(X_k^2 - N_k^2) + R_k(d),$$

onde

$$R_k(d) = \tfrac{1}{6n^{3/2}} \left(X_k^3 f^{(3)}(d + \xi_X) - N_k^3 f^{(3)}(d + \xi_N)\right).$$

Passo 4: Tomando esperança condicional. Voltamos para

$$\Delta_k = \mathbb{E}\Big[f\Big(D_n^{(k)} + \tfrac{X_k}{\sqrt{n}}\Big) - f\Big(D_n^{(k)} + \tfrac{N_k}{\sqrt{n}}\Big)\Big].$$

Usando a decomposição anterior e condicionando em $D_n^{(k)}$, temos:

$$\begin{aligned}\Delta_k = {} & \mathbb{E}\Big[\tfrac{1}{\sqrt{n}} f'(D_n^{(k)})(X_k - N_k)\Big] \\ & + \mathbb{E}\Big[\tfrac{1}{2n} f''(D_n^{(k)})(X_k^2 - N_k^2)\Big] \\ & + \mathbb{E}[R_k].\end{aligned}$$

Agora, como X_k e N_k são independentes de $D_n^{(k)}$, obtemos:

$$\mathbb{E}[f'(D_n^{(k)})(X_k - N_k)] = \mathbb{E}[f'(D_n^{(k)})] \cdot (\mathbb{E}[X_k] - \mathbb{E}[N_k]) = 0,$$
$$\mathbb{E}[f''(D_n^{(k)})(X_k^2 - N_k^2)] = \mathbb{E}[f''(D_n^{(k)})] \cdot (\mathbb{E}[X_k^2] - \mathbb{E}[N_k^2]) = 0.$$

Portanto, só resta

$$\Delta_k = \mathbb{E}[R_k].$$

Passo 5: Controle do resto. Do termo R_k, temos

$$|R_k| \le \frac{1}{6n^{3/2}} \Big(|X_k|^3 \sup|f^{(3)}| + |N_k|^3 \sup|f^{(3)}|\Big).$$

Tomando esperança,

$$|\mathbb{E}[R_k]| \le \frac{C}{n^{3/2}} \big(\mathbb{E}[|X_1|^3] + \mathbb{E}[|N_1|^3]\big),$$

onde $C = \frac{1}{6} \sup|f^{(3)}|$.

Somando sobre k,

$$\left|\sum_{k=1}^{n} \Delta_k\right| \le n \cdot \frac{C}{n^{3/2}} \big(\mathbb{E}[|X_1|^3] + \mathbb{E}[|N_1|^3]\big) = O\Big(\tfrac{1}{\sqrt{n}}\Big) \;\to\; 0.$$

Logo,

$$\mathbb{E}[f(A_n)] - \mathbb{E}[f(B_n)] \to 0,$$

e como $B_n \sim N(0,1)$ para todo n, segue que $A_n \xrightarrow{d} N(0,1)$. □

Capítulo 3

Variáveis discretase suas propriedades

Nesta seção, apresentaremos as principais propriedades de algumas distribuições de probabilidade amplamente utilizadas, que servirão de base para o estudo e a implementação de métodos de simulação nas seções posteriores.

Iniciaremos com o estudo de algumas distribuições discretas, isto é, distribuições de probabilidade cuja variável aleatória associada assume apenas valores em um conjunto enumerável (finito ou infinito).

Nesse caso, a distribuição é completamente determinada pela função de probabilidade

$$p_X(x_k) = \mathbb{P}(X = x_k), \quad x_k \in S,$$

onde S é um conjunto de valores possíveis da variável aleatória X.

Essas probabilidades satisfazem

$$p_X(k) \geq 0 \quad \text{para todo } k \in S, \quad \sum_{k \in S} p_X(k) = 1.$$

Exemplo 2. *Seja $S = \{x_1, x_2, \ldots, x_K\}$ um conjunto de K valores distintos. Dizemos que X tem distribuição uniforme discreta em S quando*

$$p_X(x_i) = \frac{1}{K}, \quad i = 1, 2, \ldots, K.$$

Nesse caso, cada valor é igualmente provável e temos

$$\sum_{i=1}^{K} p_X(x_i) = \sum_{i=1}^{K} \frac{1}{K} = 1.$$

3.1 Bernoulli

A distribuição de Bernoulli modela experimentos com dois resultados possíveis, tipicamente denominados "sucesso" (valor 1) e "fracasso" (valor 0). Dizemos que $X \sim \text{Bernoulli}(p)$ se

$$\mathbb{P}(X = 1) = p \quad \text{e} \quad \mathbb{P}(X = 0) = 1 - p,$$

onde $0 \leq p \leq 1$ representa a probabilidade de sucesso.

A função de probabilidade (pmf) pode ser escrita de forma compacta como

$$p_X(k) = p^k(1-p)^{1-k}, \quad k \in \{0,1\}.$$

As principais características dessa distribuição são:

$$\mathbb{E}[X] = p, \quad \text{Var}(X) = p(1-p).$$

Exercício 9. *Prove as propriedades acima.*

3.2 Distribuição Binomial

A distribuição binomial modela o número de sucessos em n repetições independentes de um experimento de Bernoulli com probabilidade de sucesso p.

Sejam $X_1, X_2, \ldots, X_n$ variáveis aleatórias independentes, todas com distribuição Bernoulli(p). Definimos

$$X = \sum_{i=1}^{n} X_i.$$

Nesse caso, dizemos que $X \sim \text{Binomial}(n,p)$, cuja função de probabilidade é

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \ldots, n.$$

As principais propriedades são:

$$\mathbb{E}[X] = np, \quad \text{Var}(X) = np(1-p).$$

Exercício 10. *Prove as propriedades acima.*

3.3 Distribuição Geométrica

A distribuição geométrica modela o número de ensaios de Bernoulli até a ocorrência do primeiro sucesso. Seja p a probabilidade de sucesso em cada tentativa, com $0 < p \leq 1$. Definimos X como o número de ensaios necessários até o primeiro sucesso. Dizemos que $X \sim \text{Geom}(p)$ se

$$\mathbb{P}(X = k) = (1-p)^{k-1} p, \quad k = 1, 2, 3, \ldots$$

Nesse caso:

$$\mathbb{E}[X] = \frac{1}{p}, \quad \text{Var}(X) = \frac{1-p}{p^2}.$$

Exercício 11. *Prove que a função de probabilidade acima satisfaz* $\sum_{k=1}^{\infty} \mathbb{P}(X = k) = 1$.

Exercício 12. *Prove as propriedades acima.*

3.4 Distribuição de Poisson

A distribuição de Poisson modela o número de ocorrências de um evento em um intervalo fixo de tempo ou espaço, assumindo que tais ocorrências sejam raras e independentes.

Dizemos que $X \sim \text{Poisson}(\lambda)$ se sua função de probabilidade for

$$\mathbb{P}(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}, \quad k = 0, 1, 2, \ldots,$$

onde $\lambda > 0$ representa a taxa média de ocorrências no intervalo considerado.

A média e a variância são dadas por

$$\mathbb{E}[X] = \lambda, \quad \text{Var}(X) = \lambda.$$

Exercício 13. *Prove que* $\sum_{k=0}^{\infty} \mathbb{P}(X = k) = 1$.

Exercício 14. *Prove as propriedades acima.* Dica: *para a variância, calcule primeiro* $\mathbb{E}[X(X-1)]$ *e use o fato de que*

$$\text{Var}(X) = \mathbb{E}[X(X-1)] + \mathbb{E}[X] - \big(\mathbb{E}[X]\big)^2.$$

3.5 Distribuição Binomial Negativa

Seja X o número de ensaios necessários para obter um total de r sucessos, considerando que cada ensaio é independente e resulta em sucesso com probabilidade p. Nesse caso, dizemos que X segue uma distribuição *binomial negativa* (também chamada *Pascal*) com parâmetros p e r.

Sua função de probabilidade é dada por:

$$\mathbb{P}(X = n) = \binom{n-1}{r-1} p^r (1-p)^{n-r}, \quad n = r, r+1, r+2, \ldots$$

Essa fórmula é justificada pelo fato de que, para que sejam necessários exatamente n ensaios para obter r sucessos, os primeiros $n-1$ ensaios devem conter exatamente $r-1$ sucessos — o que ocorre com probabilidade

$$\binom{n-1}{r-1} p^{r-1} (1-p)^{n-r}$$

— e, em seguida, o n-ésimo ensaio deve ser um sucesso, com probabilidade p.

Seja X_i, $i = 1, \ldots, r$, o número de ensaios necessários após o $(i-1)$-ésimo sucesso para obter o i-ésimo sucesso. É fácil ver que $X_1, X_2, \ldots, X_r$ são variáveis aleatórias independentes com distribuição $\text{Geom}(p)$. Assim, como

$$X = X_1 + X_2 + \cdots + X_r,$$

temos, usando os resultados da distribuição geométrica:

$$\mathbb{E}[X] = \sum_{i=1}^{r} \mathbb{E}[X_i] = \frac{r}{p}, \quad \text{Var}(X) = \sum_{i=1}^{r} \text{Var}(X_i) = \frac{r(1-p)}{p^2}.$$

Exercício 15. *Prove que a função de probabilidade acima é válida, isto é, que* $\sum_{n=r}^{\infty} \mathbb{P}(X = n) = 1$.

Exercício 16. *Prove as fórmulas da média e variância usando o fato de que* X *é a soma de* r *variáveis independentes com distribuição* $\text{Geom}(p)$.

3.6 Distribuição Hipergeométrica

A distribuição hipergeométrica modela experimentos de seleção *sem reposição* a partir de uma população finita contendo dois tipos de elementos. Por exemplo, suponha uma urna com $N + M$ bolas, das quais N são claras e M são escuras. Retiramos, de forma aleatória e sem reposição, uma amostra de tamanho n. Seja X o número de bolas claras na amostra.

Nesse caso, cada subconjunto de tamanho n é igualmente provável, e a probabilidade de observar exatamente k bolas claras é

$$\mathbb{P}(X = k) = \frac{\binom{N}{k}\binom{M}{n-k}}{\binom{N+M}{n}}, \quad \max(0, n - M) \leq k \leq \min(n, N).$$

Dizemos então que $X \sim \text{Hipergeom}(N, M, n)$.

As principais propriedades dessa distribuição são:

$$\mathbb{E}[X] = n \cdot \frac{N}{N+M}, \quad \text{Var}(X) = n \cdot \frac{N}{N+M} \cdot \frac{M}{N+M} \cdot \frac{N+M-n}{N+M-1}.$$

Exercício 17. *Prove que a função de probabilidade acima é válida, isto é, que*

$$\sum_{k=\max(0,\, n-M)}^{\min(n,\, N)} \mathbb{P}(X = k) = 1.$$

Exercício 18. *Prove as fórmulas da média e variância acima.* Dica: *considere o sorteio sequencial das n bolas e defina X_i como a variável indicadora do evento "a i-ésima bola é clara". Para a variância, use a decomposição*

$$\text{Var}(X) = \sum_{i=1}^{n} \text{Var}(X_i) + 2 \sum_{1 \leq i < j \leq n} \text{Cov}(X_i, X_j).$$

Capítulo 4

Simulação de variáveis discretas via inversão

O truque fundamental para simular variáveis aleatórias a partir de uma variável uniforme $U \sim \text{Uniforme}(0,1)$ é a seguinte propriedade:

$$\mathbb{P}(a < U < b) = b - a, \quad 0 \leq a < b \leq 1.$$

Isto é, a probabilidade de U cair em um subintervalo do intervalo $(0,1)$ é igual ao comprimento desse subintervalo.

Para variáveis discretas, essa ideia pode ser usada da seguinte forma: suponha que X assuma valores $x_1, x_2, \ldots, x_m$ com probabilidades $p_1, p_2, \ldots, p_m$, onde

$$p_k = \mathbb{P}(X = x_k), \quad p_k \geq 0, \quad \sum_{k=1}^{m} p_k = 1.$$

Definimos as probabilidades acumuladas

$$F_k = \sum_{i=1}^{k} p_i, \quad k = 1, \ldots, m.$$

Então, o algoritmo de simulação é:

1. Gerar $U \sim \text{Uniforme}(0,1)$;
2. Encontrar o menor índice k tal que $U \leq F_k$;
3. Retornar $X = x_k$.

A propriedade $\mathbb{P}(a < U < b) = b - a$ garante que

$$\mathbb{P}(X = x_k) = p_k.$$

De forma intuitiva, dividimos o intervalo $(0,1)$ em subintervalos consecutivos de comprimentos p_k. Ao sortearmos $U \sim \text{Uniforme}(0,1)$, o valor de X será aquele correspondente ao subintervalo no qual U cair. Esse procedimento é conhecido como *método da inversão* para variáveis discretas.

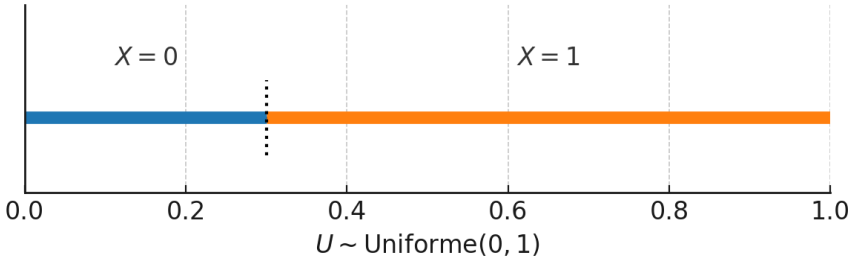


Figura 4.1: Particionamento do intervalo $(0,1)$ para simular uma variável Bernoulli com $p = 0{,}3$. Sorteia-se $U \sim \text{Uniforme}(0,1)$; se U cair na região azul, definimos $X = 0$, e caso contrário, $X = 1$.

A Figura 4.1 ilustra esse processo para uma variável Bernoulli.

A mesma ideia se aplica quando o conjunto de valores possíveis de X é infinito (ou muito grande). Nesse caso, o intervalo $(0,1)$ é particionado em uma sequência de subintervalos, cada um correspondente a um valor de X, como ilustrado na Figura 4.2.

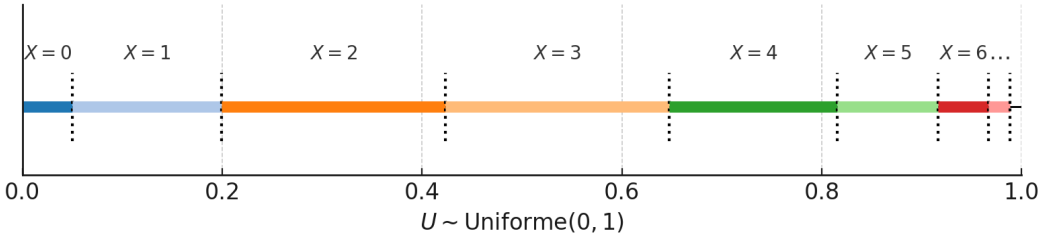


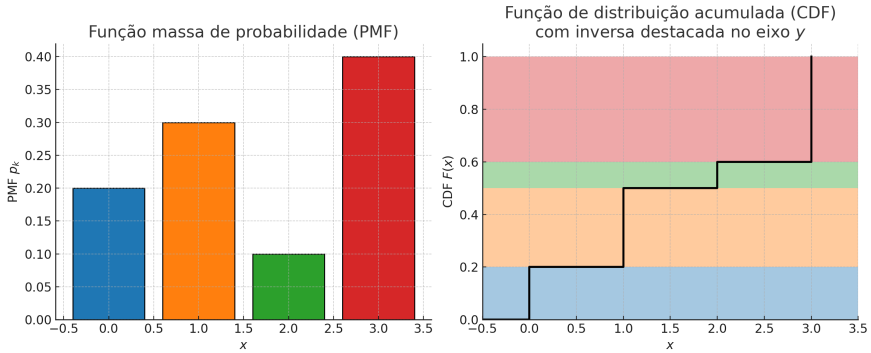
Figura 4.2: Particionamento do intervalo $(0,1)$ para simular uma variável discreta com suporte infinito.

O nome *método da inversão* vem do fato de que a simulação utiliza a *função de distribuição acumulada* (CDF) e sua *inversa generalizada*. Seja X uma variável aleatória com CDF $F(x)$. Então, se $U \sim \text{Uniforme}(0,1)$, vale que

$$X = F^{-1}(U),$$

onde a inversa generalizada é definida por

$$F^{-1}(u) = \min\{x : F(x) \geq u\}, \quad 0 < u < 1.$$



No caso discreto, isto corresponde exatamente ao passo do algoritmo em que escolhemos o menor k tal que $U \leq F_k$. Ou seja, sorteamos U, e depois "invertemos" a CDF para recuperar uma realização de X na sua escala original.

Esse procedimento pode parecer um pouco abstrato neste momento, já que a noção de inversa de uma função acumulada fica mais clara quando lidamos com variáveis contínuas. Por isso, retornaremos a esse método mais adiante, ao estudarmos a simulação de variáveis contínuas via inversão.

4.1 Variáveis com suporte finito

Considere o caso em que X assume um número finito de valores $x_1, x_2, \ldots, x_m$, cada um com probabilidade $p_j = \mathbb{P}(X = x_j)$.

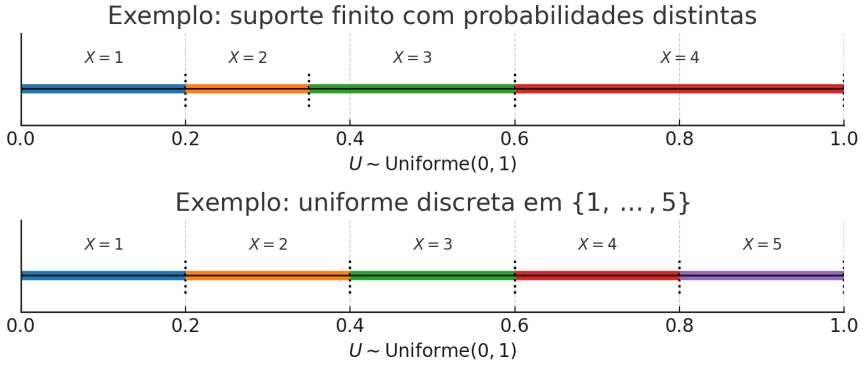
Por exemplo, suponha que

$$p_1 = 0.20, \quad p_2 = 0.15, \quad p_3 = 0.25, \quad p_4 = 0.40.$$

Uma maneira direta de simular X é gerar $U \sim \text{Uniforme}(0, 1)$ e aplicar:

- Se $U < 0.20$, definir $X = 1$ e pare;
- Se $U < 0.35$, definir $X = 2$ e pare;
- Se $U < 0.60$, definir $X = 3$ e pare;
- Caso contrário, definir $X = 4$.

Embora possamos reordenar os testes para tornar a verificação mais eficiente, a ideia central permanece a mesma: dividir o intervalo $(0, 1)$ em partes de comprimentos p_j e identificar onde U caiu.



Um caso especial é o da *uniforme discreta* nos valores $1, 2, \ldots, n$, em que

$$\mathbb{P}(X = j) = \frac{1}{n}, \quad j = 1, \ldots, n.$$

Neste cenário, o método se torna extremamente simples: basta gerar $U \sim \text{Uniforme}(0, 1)$ e definir

$$X = \lfloor nU \rfloor + 1,$$

onde $\lfloor x \rfloor$ indica a parte inteira de x (maior inteiro menor ou igual a x).

De fato, $X = j$ se e somente se $j - 1 \leq nU < j$, o que ocorre com probabilidade $\frac{1}{n}$. Variáveis uniformes discretas são particularmente importantes em simulação, pois permitem gerar inteiros equiprováveis de forma extremamente eficiente.

Um caso ainda mais simples é a simulação de uma variável **Bernoulli** com parâmetro p, que é equivalente a uma uniforme discreta em $\{0, 1\}$ com probabilidades $1 - p$ e p, respectivamente.

4.2 Binomial

A seguir veremos duas formas de simulas variáveis binomiais.

4.2.1 Simulando via Bernoullis

Uma forma simples e direta de simular uma variável aleatória binomial é a partir de variáveis de Bernoulli independentes.

Recorde que se $X \sim \text{Binomial}(n, p)$, então X pode ser escrito como

$$X = \sum_{i=1}^{n} B_i,$$

onde $B_1, B_2, \ldots, B_n$ são variáveis independentes e identicamente distribuídas, cada uma com

$$B_i \sim \text{Bernoulli}(p).$$

Assim, o algoritmo de simulação da binomial segue naturalmente:

1. Para $i = 1, \ldots, n$, gerar $B_i \sim \text{Bernoulli}(p)$;
2. Retornar $X = \sum_{i=1}^{n} B_i$.

Em outras palavras, uma variável binomial conta o número de sucessos em n tentativas independentes, cada uma com probabilidade de sucesso p. Portanto, simular uma binomial se reduz a repetir n vezes o procedimento de simulação da Bernoulli e somar os resultados.

4.2.2 Simulando via identidade recursiva

Uma alternativa mais eficiente utiliza o *método da inversão*, aproveitando a identidade recursiva da função massa de probabilidade da Binomial.

Se $X \sim \text{Binomial}(n, p)$, então

$$\mathbb{P}(X = i) = \binom{n}{i} p^i (1 - p)^{n-i}, \quad i = 0, 1, \ldots, n.$$

Essas probabilidades satisfazem uma relação de recorrência simples. De fato, começando em

$$\mathbb{P}(X = i + 1) = \binom{n}{i+1} p^{i+1} (1 - p)^{n-i-1},$$

observamos que [1]

$$\binom{n}{i+1} = \frac{n!}{(i+1)!\,(n-i-1)!} = \frac{n-i}{i+1}\,\frac{n!}{i!\,(n-i)!} = \frac{n-i}{i+1}\binom{n}{i}.$$

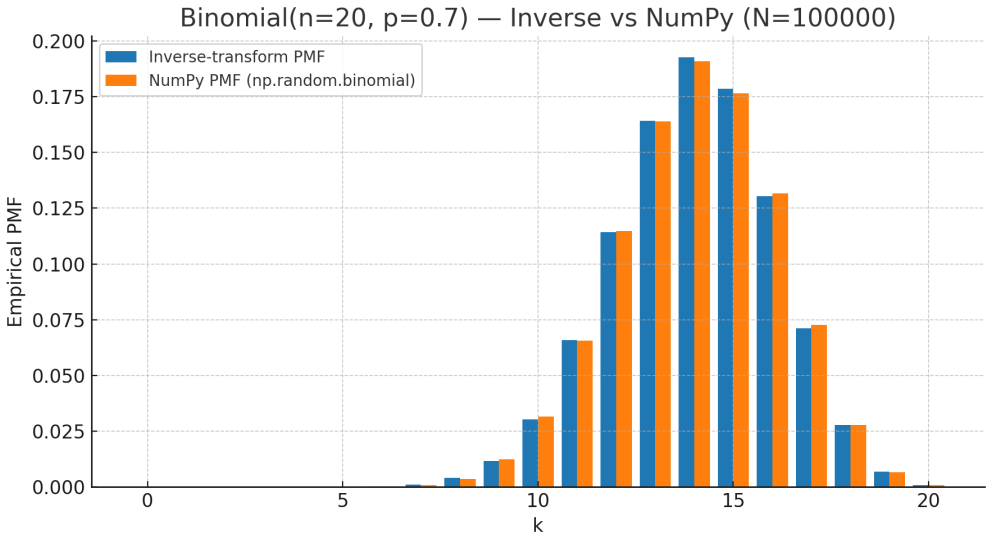
Substituindo essa relação,

$$\mathbb{P}(X = i+1) = \frac{n-i}{i+1}\binom{n}{i}\,p^{i+1}(1-p)^{n-i-1}.$$

Reorganizando,

$$\mathbb{P}(X = i+1) = \frac{n-i}{i+1}\cdot\frac{p}{1-p}\,\mathbb{P}(X = i).$$

Assim, conhecendo $\mathbb{P}(X = 0) = (1-p)^n$, podemos calcular $\mathbb{P}(X = 1), \mathbb{P}(X = 2), \ldots$ de forma recursiva, sem reavaliar coeficientes binomiais nem potências.



Isso leva ao seguinte algoritmo de simulação via inversão:

1. Gerar $U \sim \text{Uniforme}(0,1)$;
2. Inicializar o índice $i = 0$, a probabilidade atual $p_i = (1-p)^n$ e a soma acumulada $F = p_i$;
3. Enquanto $U > F$, atualizar

$$p_{i+1} = \frac{n-i}{i+1}\cdot\frac{p}{1-p}\,p_i, \qquad i \leftarrow i+1, \qquad F \leftarrow F + p_i;$$

4. Retornar $X = i$.

Esse procedimento verifica primeiro se $X = 0$, depois se $X = 1$, e assim por diante, até encontrar o valor de X sorteado. Em média, o número de passos necessários é aproximadamente $1 + np$, o que pode ser bem mais eficiente do que gerar n variáveis de Bernoulli quando n é grande.

[1] Por exemplo, se $n = 10$, $i = 6$ e $i+1 = 7$, então

$$\frac{10!}{7!\,3!} = \frac{10!\cdot 4}{7\cdot 6!\cdot 4\cdot 3!} = \frac{10!}{6!\cdot 4!}\,\frac{4}{7}$$

Exemplo 3. *Considere* $n = 5$ *e* $p = 0.3$. *Temos* $\mathbb{P}(X = 0) = (1 - 0.3)^5 = 0.16807$. *Suponha que geramos* $U = 0.4$. *Como* $U > 0.16807$, *passamos ao próximo valor:*

$$p_1 = \frac{5-0}{1} \cdot \frac{0.3}{0.7} \cdot 0.16807 \approx 0.36015, \quad F = 0.16807 + 0.36015 = 0.52822.$$

Agora $U = 0.4 < F$, *logo o algoritmo retorna* $X = 1$.

Portanto, neste caso específico, o sorteio resultou em exatamente um sucesso entre as cinco tentativas.

4.2.3 Aspectos computacionais

A escolha do método para simular variáveis binomiais tem implicações diretas em termos de eficiência. Dois fatores fundamentais influenciam o desempenho: o número de tentativas n e a probabilidade de sucesso p.

No método da soma de Bernoullis, o custo de cada amostra é proporcional a n, já que é necessário realizar n sorteios independentes. Esse custo não depende do valor de p: tanto para valores pequenos quanto grandes de p, o algoritmo precisa sempre gerar todas as n Bernoullis.

Já no método da inversão recursiva, o número médio de passos é da ordem de $1 + np$, pois o procedimento acumula probabilidades até ultrapassar o valor sorteado U. Quando p é pequeno, o valor típico da variável X também é pequeno, e o algoritmo tende a parar cedo, podendo ser competitivo em relação à soma de Bernoullis. Por outro lado, quando p é moderado ou grande, o valor esperado np cresce e, com ele, o número de passos, tornando a inversão significativamente mais lenta.

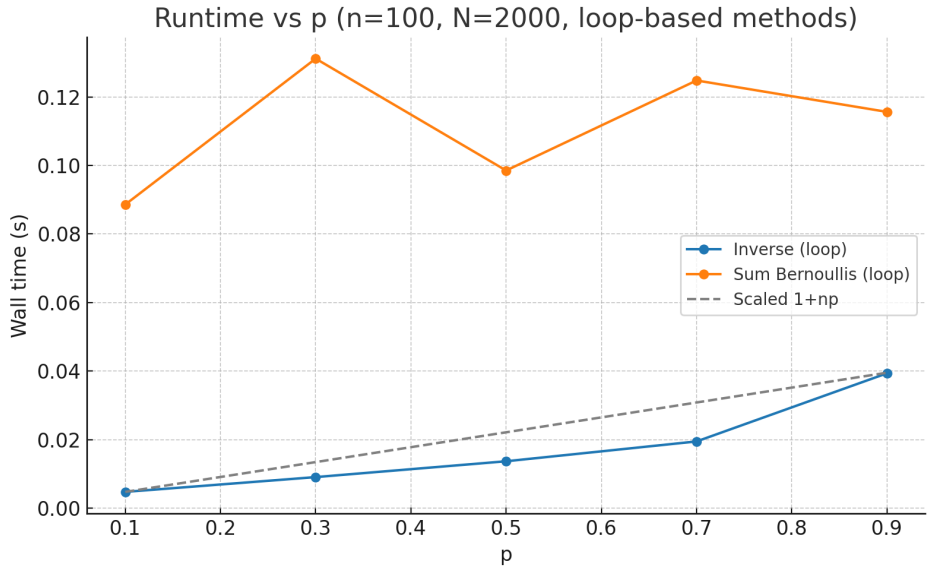


Figura 4.3: Comparação de tempo de execução (em segundos) entre o método da inversão recursiva e a soma de Bernoullis para $n = 100$ e $N = 2000$ amostras, variando p.

A Figura 4.3 ilustra essa comparação em implementações com *loops* explícitos, para $n = 100$ e diferentes valores de p. Enquanto o tempo da soma de Bernoullis cresce linearmente apenas com n e não é afetado por p, o tempo do método da inversão cresce proporcionalmente a np, aumentando de forma acentuada à medida que p se aproxima de 1. Na prática, bibliotecas como

`NumPy` utilizam algoritmos especializados para a binomial, ainda mais rápidos do que ambos os métodos discutidos aqui, de modo que a utilidade principal desses algoritmos é didática e comparativa, permitindo compreender os diferentes custos computacionais associados a cada abordagem.

4.2.4 Número médio de passos em algoritmos de inversão recursiva

Nos algoritmos recursivos de inversão, a lógica é sempre a mesma: dado um número aleatório $U \sim \text{Uniforme}(0,1)$, acumulamos as probabilidades da distribuição até que a soma ultrapasse U. O valor de X sorteado é exatamente o índice k em que essa condição se verifica pela primeira vez.

Assim, se o valor sorteado é $X = k$, o algoritmo precisou verificar todos os valores $0, 1, 2, \ldots, k-1$ e só então aceitou k. Isso significa que o número total de passos é

$$S = k + 1.$$

Como X é a variável aleatória que estamos simulando, temos

$$\mathbb{E}[S] = \mathbb{E}[X + 1] = \mathbb{E}[X] + 1.$$

Esse resultado é geral para qualquer algoritmo de inversão recursiva que inicie a busca no valor mínimo do suporte e avance de forma sequencial. No caso da binomial $X \sim \text{Bin}(n, p)$, por exemplo, o número esperado de passos é

$$\mathbb{E}[S] = 1 + np,$$

uma vez que $\mathbb{E}[X] = np$.

Portanto, o custo médio do algoritmo está diretamente ligado ao valor esperado da distribuição sorteada: distribuições concentradas em valores pequenos produzem simulações muito rápidas, enquanto distribuições centradas em valores grandes exigem proporcionalmente mais passos.

4.3 Simulação da distribuição de Poisson

Seja $X \sim \text{Poisson}(\lambda)$. A função de probabilidade é

$$\mathbb{P}(X = i) = e^{-\lambda} \frac{\lambda^i}{i!}, \quad i = 0, 1, 2, \ldots$$

e essa expressão satisfaz a relação de recorrência

$$\mathbb{P}(X = i + 1) = \frac{\lambda}{i + 1} \mathbb{P}(X = i).$$

Para ver isso, basta observar que $\mathbb{P}(X = i+1) = e^{-\lambda} \frac{\lambda^{i+1}}{(i+1)!}$. Separando um fator $\lambda/(i+1)$, obtemos $\mathbb{P}(X = i+1) = \frac{\lambda}{i+1} e^{-\lambda} \frac{\lambda^i}{i!}$, que nada mais é do que $\frac{\lambda}{i+1} \mathbb{P}(X = i)$. Assim, conhecendo $p_0 = \mathbb{P}(X = 0) = e^{-\lambda}$, é possível calcular recursivamente $p_1 = \lambda p_0$, depois $p_2 = (\lambda/2)p_1$, e assim sucessivamente. Esse raciocínio evita a recomputação de fatoriais a cada passo e fornece um procedimento numericamente mais estável.

4.3.1 Algoritmo de inversão

O algoritmo clássico para gerar uma variável de Poisson com parâmetro λ funciona da seguinte maneira:

1. Gerar $U \sim \text{Uniforme}(0,1)$;
2. Inicializar $i = 0$, $p_0 = e^{-\lambda}$ e $F = p_0$;
3. Enquanto $U > F$, atualizar
$$i \leftarrow i+1, \qquad p_i \leftarrow \frac{\lambda}{i}\, p_{i-1}, \qquad F \leftarrow F + p_i;$$
4. Retornar $X = i$.

Esse procedimento verifica primeiro se $X = 0$, depois se $X = 1$, e assim por diante, até encontrar o valor sorteado. O número médio de passos necessários é $1 + \lambda$, de modo que o algoritmo é eficiente para λ pequeno, mas se torna custoso para valores grandes de λ.

Exemplo 4. *Considere $\lambda = 3$ e suponha que o número aleatório gerado seja $U = 0.35$.*

- *Primeiro, calculamos $p_0 = e^{-3} \approx 0.0498$ e $F = p_0 \approx 0.0498$. Como $U = 0.35 > F$, avançamos para o próximo valor.*
- *Calculamos $p_1 = \frac{3}{1} p_0 \approx 0.1494$ e atualizamos $F \approx 0.0498 + 0.1494 = 0.1992$. Ainda temos $U = 0.35 > F$, logo seguimos adiante.*
- *Agora $p_2 = \frac{3}{2} p_1 \approx 0.2240$ e $F \approx 0.1992 + 0.2240 = 0.4232$. Como $U = 0.35 < F$, o algoritmo para aqui e retornamos $X = 2$.*

Portanto, neste exemplo, o valor simulado da variável aleatória foi $X = 2$.

4.3.2 Algoritmo melhorado

Uma forma mais eficiente de implementar o método é iniciar a busca em torno do valor mais provável da variável, que está próximo de λ. Seja $m = \lfloor \lambda \rfloor$. Calcula-se a probabilidade acumulada

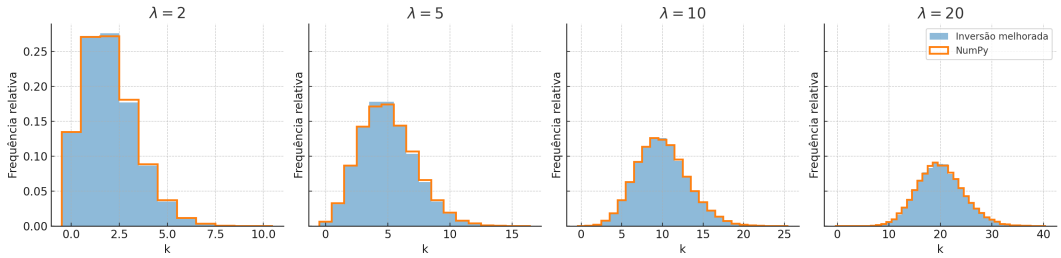
$$F(m) = \mathbb{P}(X \leq m),$$

usando a recorrência das probabilidades. Em seguida, gera-se $U \sim \text{Uniforme}(0,1)$ e procede-se assim:

- se $U < F(m)$, faz-se a busca recursiva para baixo ($m-1, m-2, \ldots$);
- se $U \geq F(m)$, faz-se a busca recursiva para cima ($m+1, m+2, \ldots$).

Note que a mesma identidade recursiva que relaciona $\mathbb{P}(X = i+1)$ a $\mathbb{P}(X = i)$ também pode ser escrita no sentido inverso:

$$\mathbb{P}(X = i) \;=\; \frac{i+1}{\lambda}\, \mathbb{P}(X = i+1).$$



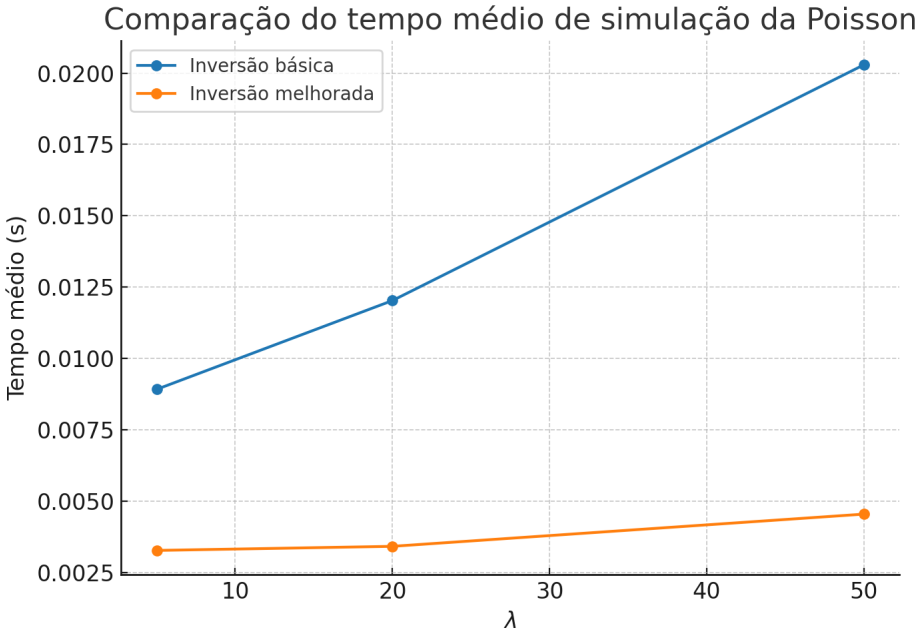
Assim, a partir de $p_m = \mathbb{P}(X = m)$, é possível atualizar as probabilidades tanto para cima quanto para baixo, sem necessidade de recalcular fatoriais. Isso garante que a busca em torno de m seja realizada de forma eficiente, explorando o valor sorteado em ambas as direções.

Neste caso, o número de passos não depende mais diretamente de X, mas sim da distância entre X e λ: para localizar o valor sorteado, precisamos primeiro verificar m, e depois avançar $|X - m|$ passos adicionais. Assim, o número de passos é

$$T = 1 + |X - m|.$$

Como $m \approx \lambda$, o custo médio pode ser aproximado por

$$\mathbb{E}[T] = 1 + \mathbb{E}[|X - \lambda|].$$



4.4 Geométrica

Seja X uma variável aleatória geométrica com parâmetro p, isto é,

$$\mathbb{P}(X = i) = (1 - p)^{i-1} p, \quad i = 1, 2, \ldots.$$

Essa variável pode ser interpretada como o tempo da primeira ocorrência de sucesso em uma sequência de tentativas independentes, cada uma com probabilidade p de sucesso.

A função de distribuição acumulada é

$$\mathbb{P}(X \leq j) \;=\; 1 - \mathbb{P}(X > j) \;=\; 1 - \mathbb{P}(\text{primeiras } j \text{ tentativas são falhas}) \;=\; 1 - (1-p)^j.$$

Assim, podemos usar o método da inversão para gerar X. Seja $U \sim \text{Uniforme}(0,1)$. Definimos $X = j$ se

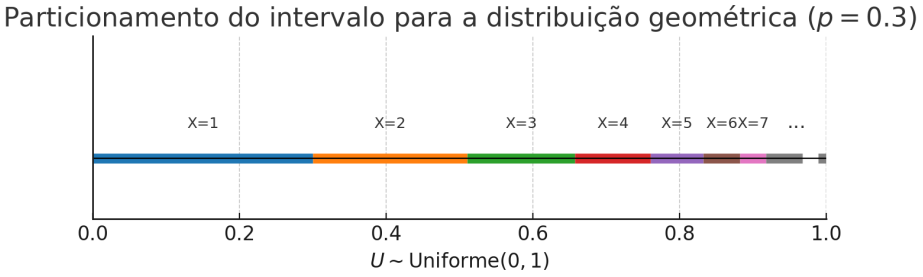
$$1 - (1-p)^{j-1} \leq U < 1 - (1-p)^j,$$

ou seja,

$$(1-p)^j < 1 - U \leq (1-p)^{j-1},$$

o que equivale a

$$X = \min\{j : (1-p)^j < 1 - U\}.$$



Como $0 < 1 - p < 1$, temos $\log(1-p) < 0$. Aplicando logaritmos,

$$(1-p)^j < 1 - U \quad \Longleftrightarrow \quad j \log(1-p) < \log(1-U).$$

Logo,

$$X = \min\left\{ j : j > \frac{\log(1-U)}{\log(1-p)} \right\}.$$

Portanto, obtemos a fórmula fechada

$$X \;=\; \left\lfloor \frac{\log(1-U)}{\log(1-p)} \right\rfloor + 1.$$

Como $1 - U \sim \text{Uniforme}(0,1)$, podemos substituir $1 - U$ por U sem perda de generalidade, resultando em

$$X \;=\; \left\lfloor \frac{\log(U)}{\log(1-p)} \right\rfloor + 1.$$

Exemplo 5. *Considere uma variável geométrica $X \sim \text{Geom}(p)$ com $p = 0.3$ e seja $U = 0.52$ uma realização de uma variável uniforme $(0,1)$. Usando a fórmula fechada da inversão, temos*

$$X \;=\; \left\lfloor \frac{\log(1-U)}{\log(1-p)} \right\rfloor + 1,$$

e, substituindo os valores, obtemos

$$X \;=\; \left\lfloor \frac{\log(0.48)}{\log(0.7)} \right\rfloor + 1 \;\approx\; \lfloor 2.06 \rfloor + 1 \;=\; 3.$$

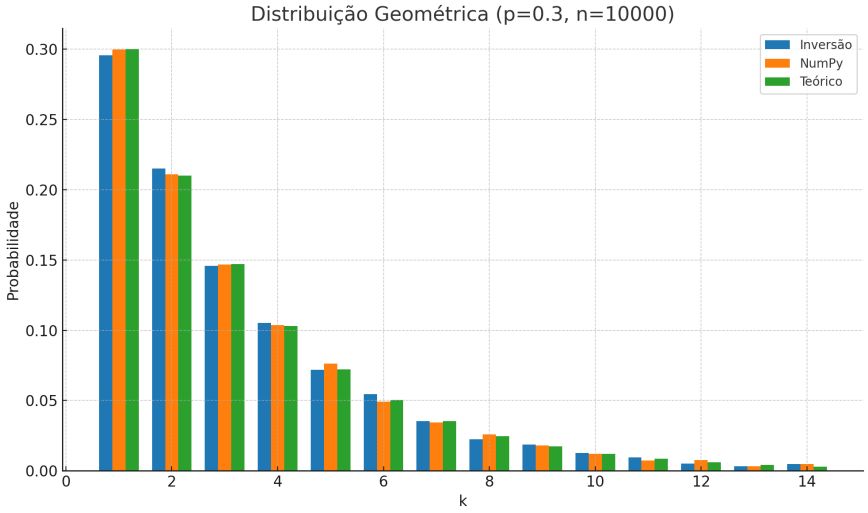
Outra forma é aplicar a inversão direta da CDF, que é dada por

$$F(j) \;=\; \mathbb{P}(X \leq j) = 1 - (1-p)^j.$$

Procuramos o menor j tal que $F(j) \geq U$. Para $p = 0.3$, temos

$$F(1) = 0.3, \quad F(2) = 0.51, \quad F(3) \approx 0.657.$$

Como $F(2) = 0.51 < U = 0.52$ mas $F(3) = 0.657 \geq 0.52$, o menor j que satisfaz é $j = 3$.



4.5 Binomial Negativa

A distribuição Binomial Negativa pode ser vista como o número de ensaios necessários até a ocorrência do r-ésimo sucesso em ensaios independentes com probabilidade de sucesso p.

4.5.1 Simulando via Bernoullis

Uma forma direta de gerar uma variável Binomial Negativa é simular sucessivos ensaios de Bernoulli(p) até obter o r-ésimo sucesso.

De fato, por definição, X representa o número total de ensaios necessários até a ocorrência de r sucessos. Assim, o algoritmo pode ser descrito da seguinte forma:

1. Inicializar $n = 0$ (contador de ensaios) e $s = 0$ (contador de sucessos);
2. Enquanto $s < r$:
   (a) Gerar $B \sim \text{Bernoulli}(p)$;
   (b) Atualizar $n \leftarrow n + 1$;
   (c) Se $B = 1$, atualizar $s \leftarrow s + 1$;
3. Retornar $X = n$.

Esse método é conceitualmente simples e corresponde exatamente à definição da distribuição Binomial Negativa. No entanto, quando p é pequeno e r é grande, o número esperado de ensaios $\mathbb{E}[X] = r/p$ pode ser elevado, tornando o algoritmo computacionalmente mais custoso.

4.5.2 Simulando via soma de geométricas

Recorde que se $X \sim \text{NegBin}(r, p)$, então X pode ser decomposto como

$$X = X_1 + X_2 + \cdots + X_r,$$

onde $X_1, \ldots, X_r$ são variáveis independentes com

$$X_i \sim \text{Geom}(p), \quad i = 1, \ldots, r,$$

no suporte $\{1, 2, \ldots\}$ (número de ensaios até o primeiro sucesso).

Assim, podemos simular uma Binomial Negativa somando r geométricas independentes, cada uma gerada via o método da inversão:

$$X_i = \left\lfloor \frac{\log(1 - U_i)}{\log(1 - p)} \right\rfloor + 1, \quad U_i \sim \text{Uniforme}(0, 1).$$

O algoritmo de simulação segue:

1. Para $i = 1, \ldots, r$, gerar $X_i \sim \text{Geom}(p)$ via inversão;
2. Retornar $X = \sum_{i=1}^{r} X_i$.

4.5.3 Simulando via inversão recursiva

Outra forma de simular a Binomial Negativa é aplicar diretamente o método da inversão, aproveitando a relação de recorrência da sua função de probabilidade.

Se $X \sim \text{NegBin}(r, p)$, então

$$\mathbb{P}(X = n) = \binom{n-1}{r-1} p^r (1-p)^{n-r}, \quad n = r, r+1, r+2, \ldots$$

Essas probabilidades satisfazem a seguinte relação recursiva:

$$\frac{\mathbb{P}(X = n+1)}{\mathbb{P}(X = n)} = \frac{n}{n - r + 1}(1 - p).$$

Exercício 19. *Prove a identidade recursiva acima.*

Portanto, conhecendo $\mathbb{P}(X = r) = p^r$, podemos calcular recursivamente as demais probabilidades. Isso leva ao seguinte algoritmo:

1. Gerar $U \sim \text{Uniforme}(0, 1)$;
2. Inicializar $n = r$, $p_n = p^r$, $F = p_n$;
3. Enquanto $U > F$, atualizar

$$p_{n+1} = p_n \cdot \frac{n}{n - r + 1}(1 - p), \qquad n \leftarrow n + 1, \qquad F \leftarrow F + p_{n+1};$$

4. Retornar $X = n$.

O número esperado de passos T no método recursivo não coincide diretamente com $\mathbb{E}[X]$, pois o algoritmo já inicia em $n = r$, que é o menor valor possível para a variável $X \sim \text{NegBin}(r, p)$.

De fato, se o sorteio resultar em $X = n$, o número de passos dados pelo algoritmo é

$$T = (n - r) + 1,$$

pois começamos verificando o valor $n = r$ (primeiro passo) e avançamos até alcançar n.

Assim, em termos de valor esperado,

$$\mathbb{E}[T] = \mathbb{E}[X - r] + 1 = \frac{r}{p} - r + 1.$$

Esse termo $-r$ aparece porque, embora $\mathbb{E}[X] = r/p$, o procedimento de inversão não percorre todos os valores desde 0, mas já parte de r.

Quando p é pequeno, $\mathbb{E}[T]$ pode ainda ser bastante grande, tornando o método recursivo lento. Nessas situações, a versão ingênua baseada na soma de geométricas pode ser mais eficiente na prática.

4.5.4 Por que o nome "Binomial Negativa"?

O nome Binomial Negativa tem origem na conexão com a expansão binomial para expoentes negativos. Para um inteiro $n \geq 0$, o teorema binomial fornece

$$(1 - p)^n = \sum_{k=0}^{n} \binom{n}{k} (-p)^k.$$

Essa identidade estende-se a expoentes reais (expansão binomial generalizada):

$$(1 - p)^{-\alpha} = \sum_{k=0}^{\infty} \binom{\alpha + k - 1}{k} p^k, \qquad |p| < 1.$$

Tomando $\alpha = r \in \{1, 2, \ldots\}$,

$$(1 - p)^{-r} = \sum_{k=0}^{\infty} \binom{r + k - 1}{k} p^k.$$

Os coeficientes $\binom{r+k-1}{k}$ são precisamente os que aparecem na parametrização da Binomial Negativa em termos do número de falhas k antes do r-ésimo sucesso:

$$\mathbb{P}(Y = k) = \binom{r + k - 1}{k} p^r (1 - p)^k, \qquad k = 0, 1, 2, \ldots$$

Para ver a equivalência com a forma escrita em função do número total de ensaios n, detalhamos a reparametrização. Defina $n = r + k$ (isto é, $k = n - r$). Então

$$\binom{r + k - 1}{k} = \frac{(r + k - 1)!}{k!\,(r - 1)!} = \frac{(n - 1)!}{(n - r)!\,(r - 1)!} = \binom{n - 1}{n - r}.$$

Pela simetria binomial, $\binom{a}{b} = \binom{a}{a-b}$; aplicando com $a = n - 1$ e $b = n - r$ obtemos

$$\binom{n - 1}{n - r} = \binom{n - 1}{(n - 1) - (n - r)} = \binom{n - 1}{r - 1}.$$

Substituindo $k = n - r$ em $\mathbb{P}(Y = k)$ e usando as igualdades acima,

$$\mathbb{P}(X = n) = \mathbb{P}(Y = n - r) = \binom{r + (n - r) - 1}{n - r} p^r (1 - p)^{n-r} = \binom{n - 1}{r - 1} p^r (1 - p)^{n-r}, \qquad n = r, r + 1, \ldots$$

Mostramos, assim, passo a passo, que as duas formas da PMF — em função de k (falhas) ou de n (ensaios) — são exatamente equivalentes; trata-se apenas de uma reparametrização.

Tabela de referência

Distribuição	Técnica utilizada	Dica/Obs
Suporte finito	Inversão simples	Separar em intervalos
Bernoulli	Inversão simples	Caso particular do suporte finito com $m = 2$
Binomial	Soma de Bernoullis ou inversão recursiva	Relação de recorrência evita coeficientes binomiais
Poisson	Inversão recursiva	Relação $p_{i+1} = \frac{\lambda}{i+1} p_i$ evita fatoriais
Geométrica	Inversão direta (CDF)	Retornar $X = \lfloor \frac{\log(1-U)}{\log(1-p)} \rfloor + 1$
Binomial negativa	Soma de geométricas ou inversão recursiva	Soma de r geométricas independentes
Hipergeométrica		

Capítulo 5

Técnicas computacionais

5.1 Profiling com `cProfile`

Antes de otimizar, é essencial medir onde o tempo realmente está sendo gasto. O módulo `cProfile`, da biblioteca padrão do Python, permite gerar um perfil de execução mostrando quantas vezes cada função foi chamada e quanto tempo ela consumiu.

O uso mais simples é direto pelo terminal, aplicando o profiler a um script:

```
# Executa o script inteiro e mostra estatisticas
python -m cProfile meu_script.py

# Salva os resultados em arquivo para analise posterior
python -m cProfile -o saida.prof meu_script.py
```

Rodando pelo terminal

O arquivo gerado pode ser inspecionado com o módulo `pstats`, que permite ordenar e filtrar resultados:

```
python -m pstats saida.prof
# Comandos uteis no prompt do pstats:
#   sort time       (ordena pelo tempo interno da funcao)
#   sort cumtime    (ordena pelo tempo acumulado)
#   stats 20        (mostra as 20 funcoes mais custosas)
#   callers func    (quem chama 'func')
#   callees func    (quem 'func' chama)
```

Explorando com pstats (terminal)

Também é possível usar `cProfile` dentro do código, o que facilita em notebooks ou quando queremos medir apenas um trecho específico:

```
import cProfile, pstats, io

pr = cProfile.Profile()
pr.enable()

# --- codigo a ser medido ---
```

```
resultado = algoritmo_pesado()
# ----------------------------

pr.disable()
s = io.StringIO()
ps = pstats.Stats(pr, stream=s).sort_stats("cumtime")
ps.print_stats(10)   # mostra as 10 funcoes mais custosas
print(s.getvalue())
```

Usando cProfile dentro do código

As duas métricas principais são:

- `time`: tempo gasto apenas dentro da função, sem contar chamadas internas.
- `cumtime`: tempo acumulado, incluindo todas as funções chamadas.

Em geral, começa-se ordenando por `cumtime` para encontrar o caminho mais caro da execução. Depois, olhar o `time` ajuda a identificar funções individuais que valem otimização.

5.1.1 Exemplo guiado: medindo gargalos com `cProfile`

A seguir montamos um experimento simples para evidenciar como o `cProfile` ajuda a localizar gargalos: comparamos uma multiplicação de matrizes feita de forma ingênua em Python (três laços) com a versão vetorizada do NumPy (delegada à BLAS).

O código abaixo implementa as duas versões e usa uma função auxiliar para rodar o profiler em cada uma delas, exibindo as funções mais custosas. O script pode ser salvo como `profile_matmul.py`.

```
import numpy as np
import math
import cProfile, pstats, io
import time

# Versao ingenua: 3 loops em Python
def matmul_naive(A, B):
    n, m = A.shape
    m2, p = B.shape
    assert m == m2
    C = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            s = 0.0
            for k in range(m):
                s += A[i, k] * B[k, j]
            C[i, j] = s
    return C

# Versao NumPy (vetorizada/BLAS)
```

```
def matmul_numpy(A, B):
    return A @ B

def profile_func(func, *args, top=15):
    pr = cProfile.Profile()
    pr.enable()
    t0 = time.perf_counter()
    result = func(*args)
    t1 = time.perf_counter()
    pr.disable()
    s = io.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats("cumtime")
    ps.print_stats(top)
    print(f"\n>>>> Tempo total (parede): {t1 - t0:.3f} s\n")
    return s.getvalue()

A = np.random.rand(n, n)
B = np.random.rand(n, n)

print("==== Profiling matmul_naive ====")
out_naive = profile_func(matmul_naive, A, B)
print(out_naive)

print("==== Profiling matmul_numpy ====")
out_np = profile_func(matmul_numpy, A, B)
print(out_np)
```

Esse script pode ser executado normalmente com `python profile_matmul.py`. Outra forma é rodar o profiler diretamente no terminal, usando `python -m cProfile -o saida.prof profile_matmul.py`. Nesse caso o resultado fica salvo em `saida.prof`, e podemos explorá-lo depois com o módulo `pstats` de forma interativa, usando comandos como `sort cumtime`, `stats 20` ou `callers matmul_naive`.

Rodando a versão ingênua, a saída típica mostra que praticamente todo o tempo foi consumido dentro de `matmul_naive`:

```
         7 function calls in 8.532 seconds

   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    8.523    8.523    8.523    8.523 profile_matmul.py:11(
           matmul_naive)
        1    0.009    0.009    0.009    0.009 {built-in method builtins.
           print}
        ...
```

Ao comparar com a versão vetorizada, vemos que a execução termina em milésimos de segundo, com o tempo todo acumulado em `matmul_numpy`:

```
         7 function calls in 0.020 seconds

   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.019    0.019    0.019    0.019 profile_matmul.py:28(
           matmul_numpy)
        ...
```

Os números exatos variam conforme o tamanho das matrizes e a biblioteca BLAS instalada, mas o padrão é claro: a implementação ingênua em Python puro consome segundos de CPU, enquanto a versão NumPy é milhares de vezes mais rápida.

As colunas do profiler têm significados diferentes. O campo `ncalls` mostra o número de chamadas à função. O `tottime` corresponde ao tempo gasto apenas dentro da função, sem contar chamadas internas. Já o `cumtime` indica o tempo acumulado incluindo funções chamadas dentro dela. Em geral, ordenar por `cumtime` ajuda a encontrar o caminho mais custoso da execução, enquanto olhar para `tottime` revela funções "folha" particularmente lentas.

Quando esse mesmo código é rodado em um notebook Jupyter, o output tende a ficar mais "poluído", aparecendo referências a `asyncio`, `zmq` e outros componentes do kernel. Isso acontece porque o profiler mede tudo o que roda no processo, não apenas a nossa função. Para uma visão limpa e didática, vale a pena executar o script direto no terminal.

5.2 Paralelização com `joblib.Parallel`

A biblioteca `joblib` fornece uma forma simples de paralelizar *loops embaraçosamente paralelos* em Python, isto é, situações em que várias tarefas independentes podem ser executadas ao mesmo tempo. A ideia básica é escrever um laço `for` como uma compreensão preguiçosa de chamadas a uma função via `delayed`, e despachar essas tarefas para `Parallel`, que se encarrega de distribuí-las entre diferentes trabalhadores.

```python
from joblib import Parallel, delayed
from math import sqrt

# aplicar sqrt a 0^2, 1^2, ..., 9^2 em paralelo
res = Parallel(n_jobs=4)(
    delayed(sqrt)(i**2)
    for i in range(10)
)
```

Receita de bolo

No exemplo acima, o parâmetro `n_jobs` define quantos trabalhadores serão usados (tipicamente o número de CPUs lógicas da máquina). A função `delayed` apenas empacota a chamada para que ela possa ser enviada a um worker, enquanto `Parallel` recolhe todas as tarefas e coordena sua execução.

Uma forma intuitiva de entender esse mecanismo é pensar em uma cozinha: se temos apenas um cozinheiro (um `for` sequencial), cada prato é preparado do início ao fim antes do próximo

começar. Já com vários cozinheiros (workers), cada um recebe um prato e trabalha nele independentemente, de modo que vários ficam prontos ao mesmo tempo. Essa estratégia funciona muito bem, mas há alguns cuidados: se uma tarefa demora muito enquanto outras são rápidas, pode haver desequilíbrio entre os workers; por outro lado, se existem milhares de tarefas minúsculas, o custo de despachá-las pode ser maior que o ganho da paralelização. Para reduzir esse problema, o `joblib` agrupa chamadas em lotes (*batching*), enviando várias de uma vez só.

Outro detalhe importante está no backend usado. Em Python, o *Global Interpreter Lock* (GIL) impede que várias threads executem código Python puro ao mesmo tempo. Por isso, o backend padrão (`loky`) cria processos separados, que contornam o GIL e escalam bem em cálculos pesados. Já o backend `threading` mantém as tarefas no mesmo processo, sendo útil em funções que passam a maior parte do tempo esperando I/O ou que já liberam o GIL (como operações NumPy). Existe ainda o `multiprocessing`, mas o `loky` tende a ser mais robusto.

```
# Uso de threads porque a funcao processa_io
# passa a maior parte do tempo esperando rede.
res = Parallel(n_jobs=8, backend="threading")(
    delayed(processa_io)(u) for u in urls
)
```

Exemplo com I/O

Em resumo: use `loky` (padrão) para tarefas CPU-bound, `threading` para tarefas I/O-bound, e sempre ajuste o número de jobs de acordo com o hardware disponível. Paralelizar acelera muito, mas nem sempre compensa: quando as tarefas são pequenas demais, o overhead pode superar o benefício.

Um exemplo clássico de tarefa CPU-bound é calcular números primos ou executar operações pesadas de álgebra linear. Nesses casos, vale usar o backend padrão:

```
from joblib import Parallel, delayed
import math

def eh_primo(n):
    for i in range(2, int(math.sqrt(n))+1):
        if n % i == 0:
            return False
    return True

nums = range(10**6, 10**6+1000)
res = Parallel(n_jobs=4)(delayed(eh_primo)(n) for n in nums)
```

Exemplo CPU-bound

Aqui, cada worker testa um conjunto de números independentemente. Quanto mais núcleos disponíveis, mais rápido o processamento.

Já um exemplo I/O-bound seria baixar várias páginas da web. Cada tarefa fica a maior parte do tempo esperando a rede, e usar processos separados não traz vantagem; nesse caso o backend `threading` é mais leve:

```
import requests
urls = ["https://httpbin.org/delay/1"] * 20

def baixa(url):
    return requests.get(url).status_code

res = Parallel(n_jobs=8, backend="threading")(
    delayed(baixa)(u) for u in urls
)
```

Exemplo I/O-bound

Se cada requisição demora cerca de 1 segundo, com 8 threads as 20 requisições terminam em poucos segundos, em vez de mais de 20.

Por fim, um caso em que a paralelização atrapalha é quando as tarefas são rápidas demais, por exemplo calcular o quadrado de números pequenos:

```
def quadrado(n):
    return n*n

nums = range(1000)
res = Parallel(n_jobs=4)(delayed(quadrado)(n) for n in nums)
```

Exemplo de overhead

Aqui o custo de organizar as tarefas, mandar para os workers e reunir os resultados é maior do que simplesmente rodar um `for` sequencial. Nesse cenário, a paralelização pode ser mais lenta.

5.3 Numba

Numba é um compilador JIT (Just-In-Time) para Python focado em acelerar código numérico. Ele "traduz" funções Python (que operam sobre tipos e arrays compatíveis) para código nativo via LLVM, reduzindo drasticamente o overhead dos laços em Python puro. A ideia prática é simples: decorar funções críticas com `@njit` (ou `@jit(nopython=True)`), evitar objetos Python dentro dessas funções e, quando fizer sentido, ativar paralelização com `parallel=True` e `prange`.

O primeiro cuidado ao medir é lembrar do custo de compilação: na *primeira* chamada de cada assinatura de tipos, Numba compila a função (demora mais). Depois disso, as chamadas seguintes usam o código nativo já gerado.

O exemplo abaixo acelera uma multiplicação de matrizes ingênua (três laços) sem recorrer ao `NumPy @`. Primeiro mostramos a versão *njit* sequencial; em seguida, a variação paralela (`parallel=True` + `prange`). Usamos `perf_counter` para mostrar o tempo da primeira chamada (com compilação) e das chamadas seguintes (sem compilação).

```
import numpy as np
import time
from numba import njit, prange
```

```
4
5  # Versao Python pura (referencia)
6  def matmul_naive(A, B):
7      n, m = A.shape
8      m2, p = B.shape
9      assert m == m2
10     C = np.zeros((n, p))
11     for i in range(n):
12         for j in range(p):
13             s = 0.0
14             for k in range(m):
15                 s += A[i, k] * B[k, j]
16             C[i, j] = s
17     return C
18
19 # Versao Numba: nopython mode (sem objetos Python dentro)
20 @njit
21 def matmul_numba(A, B):
22     n, m = A.shape
23     m2, p = B.shape
24     C = np.zeros((n, p))
25     for i in range(n):
26         for j in range(p):
27             s = 0.0
28             for k in range(m):
29                 s += A[i, k] * B[k, j]
30             C[i, j] = s
31     return C
32
33 # Versao Numba paralela: requer parallel=True e uso de prange
34 @njit(parallel=True)
35 def matmul_numba_parallel(A, B):
36     n, m = A.shape
37     m2, p = B.shape
38     C = np.zeros((n, p))
39     for i in prange(n):  # <-- prange permite paralelizar esse loop
       externo
40         for j in range(p):
41             s = 0.0
42             for k in range(m):
43                 s += A[i, k] * B[k, j]
44             C[i, j] = s
45     return C
46
47 # Benchmark simples: separa "primeira chamada" e "repetidas"
48 def bench(func, *args, repeat=3, label=""):
49     # primeira chamada (inclui compilacao JIT quando aplicavel)
50     t0 = time.perf_counter()
```

```
    out = func(*args)
    t1 = time.perf_counter()
    print(f"{label} [1a chamada]: {t1 - t0:.3f} s")

    # chamadas seguintes (ja compilado)
    best = float("inf")
    for _ in range(repeat):
        t0 = time.perf_counter()
        func(*args)
        t1 = time.perf_counter()
        best = min(best, t1 - t0)
    print(f"{label} [melhor chamada subsequente]: {best:.3f} s")
    return out

if __name__ == "__main__":
    n = 600
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)

    # Referencia Python puro (lento)
    bench(matmul_naive, A, B, label="naive (Python)")

    # Numba sequencial
    bench(matmul_numba, A, B, label="Numba (njit)")

    # Numba paralelo
    bench(matmul_numba_parallel, A, B, label="Numba (parallel)")
```

Acelerando loops com Numba (@njit)

Na prática, você deverá observar algo assim: a versão Python pura leva segundos; a versão `@njit` cai para frações (ou poucos segundos em matrizes grandes) após a compilação; a versão paralela tende a ganhar mais em máquinas com vários núcleos, desde que o tamanho do problema justifique o overhead de criar e sincronizar threads. Nem todo laço se beneficia de `parallel=True`; se o problema é pequeno, o custo extra pode superar o ganho.

Outro modo útil de Numba é compilar funções *elementwise* com `@vectorize`, criando uma ufunc ao estilo NumPy; isso permite aplicar a função diretamente sobre arrays, com broadcast, sem escrever laços em Python. O exemplo a seguir define uma ufunc para uma transformação escalar simples e a aplica a um array grande.

```
import numpy as np
from numba import vectorize, float64

@vectorize([float64(float64)])
def transform(x):
    # alguma transformacao escalar (exemplo)
    return (x * x + 0.5) / (x + 1.0)

```

```
x = np.random.rand(1_000_000)
y = transform(x)  # aplica como ufunc, sem lacos explicitos em Python
```

UFunc com @vectorize (estilo NumPy)

Algumas recomendações práticas ao usar Numba: (i) mantenha dentro das funções JIT apenas operações suportadas (aritmética, indexação NumPy, algumas funções `math`/`numpy`); (ii) evite objetos Python (listas que crescem, dicionários, `set`) e chamadas que exijam o interpretador; (iii) prefira arrays com *dtype* numéricos (float64, int64, etc.) e formatos contíguos; (iv) tome cuidado com alocação excessiva dentro do laço; (v) ative `parallel=True` apenas após confirmar que o gargalo é CPU-bound e que o tamanho do problema compensa a paralelização; (vi) lembre-se do "aquecimento": meça separando a primeira chamada (com compilação) das seguintes; (vii) quando a função estabilizar, `@njit(cache=True)` pode salvar o binário no disco e reduzir o tempo de compilação em execuções futuras (útil em scripts).

Por fim, se você já tem uma versão vetorizada eficiente em NumPy (que usa BLAS), muitas vezes ela será tão rápida quanto (ou mais rápida que) reimplementar em Numba, a menos que o seu padrão de acesso/cálculo seja muito específico. O ponto forte do Numba é acelerar *laços* e lógicas numéricas que seriam lentas em Python puro, mantendo o código próximo ao original, sem partir direto para C/C++.

5.4 Paralelismo simples em Bash

O Bash permite escrever pequenos scripts para automatizar tarefas repetitivas. Um dos recursos mais uteis é a possibilidade de rodar varios comandos em paralelo, sem esperar um terminar para comecar o proximo. Para isso usamos o operador `&`.

No exemplo abaixo, usamos o comando `sleep` (que apenas dorme por alguns segundos) para simular tarefas demoradas. Cada chamada ao `sleep` é enviada ao plano de fundo com `&`, de modo que o laco continua imediatamente para a proxima iteracao.

```
#!/bin/bash

for i in $(seq 1 5)
do
    echo "Iniciando tarefa $i"
    sleep 3 &
done

echo "Todas as tarefas foram lancadas!"
```

Rodando sleeps em paralelo

Nesse script, as cinco tarefas comecam quase ao mesmo tempo e, apos cerca de tres segundos, todas terminam juntas. Se tirassemos o `&`, o script levaria cerca de 15 segundos, pois cada `sleep 3` seria executado em sequencia.

Para visualizar essa diferenca, vejamos primeiro a execucao sequencial:

```
#!/bin/bash
```

```
2
3  for i in $(seq 1 5)
4  do
5      echo "Rodando tarefa $i"
6      sleep 3
7  done
8
9  echo "Todas as tarefas terminaram (sequencial)"
```

Execucao sequencial

E agora a versao em paralelo, onde o tempo total cai para cerca de 3 segundos:

```
1  #!/bin/bash
2
3  for i in $(seq 1 5)
4  do
5      echo "Rodando tarefa $i"
6      sleep 3 &
7  done
8
9  wait
10 echo "Todas as tarefas terminaram (paralelo)"
```

Execucao em paralelo

Para garantir que o script so finalize depois que todas as tarefas concluirem, podemos usar explicitamente o comando `wait`:

```
1  #!/bin/bash
2
3  for i in $(seq 1 5)
4  do
5      sleep 3 &
6  done
7
8  wait
9  echo "Todas as tarefas terminaram!"
```

Sincronizando com wait

Tambem é possivel limitar quantas tarefas rodam em paralelo. Uma tecnica simples é controlar com um contador e usar `wait -n` para esperar pelo menos um job terminar antes de lancar o proximo:

```
1  #!/bin/bash
2
3  N=2  # no maximo 2 processos ao mesmo tempo
4
5  for i in $(seq 1 5)
6  do
```

```
    sleep 3 &
    if (( $(jobs -r | wc -l) >= N )); then
        wait -n
    fi
done

wait
echo "Fim das tarefas"
```

Limitando jobs simultaneos

Esses exemplos usam apenas comandos nativos (`sleep`, `echo`), mas a ideia é exatamente a mesma se quisermos chamar um script Python ou outro programa no lugar.

Referências Bibliográficas