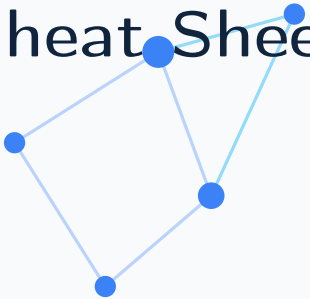
Ciências de Dados Cheat Sheet



Seção

Organização

Ferramentas para manter o trabalho limpo e reproduzível

Subseção

Git + GitHub

Controle de versão e colaboração

Controle de versão: Git + GitHub

Por que usar Git?

- Salva versões do projeto ao longo do tempo.
- Permite voltar atrás sem perder tudo.
- Facilita experimentar ideias em branches.

Por que usar GitHub?

- Guarda o código na nuvem.
- Facilita colaboração e revisão.
- Ajuda a compartilhar projetos e portfólios.

Ideia principal

Git registra a história do projeto; GitHub hospeda essa história online.

Git + GitHub: comandos básicos

Começar e salvar

- `git init`: inicia o repositório.
- `git status`: mostra o que mudou.
- `git add arquivo.py`: seleciona um arquivo.
- `git commit -m "msg"`: salva uma versão.

Sincronizar e organizar

- `git diff`: mostra detalhes das mudanças.
- `git log`: mostra o histórico.
- `git push`: envia para o GitHub.
- `git pull`: baixa atualizações.

Fluxo básico

Editar → `git status` → `git add arquivo.py` → `git commit -m "msg"` → `git push`

Git: branches e versões antigas

Trabalhar em paralelo

- `git branch`: lista branches.
- `git branch teste`: cria uma branch.
- `git checkout teste`: troca de branch.
- `git checkout main`: volta para a principal.

Recuperar uma versão

- `git log`: acha o código do commit.
- `git checkout abc123`: olha uma versão antiga.
- `git checkout main`: volta para o presente.
- `git restore -source abc123 arquivo.py`: recupera um arquivo.

Cuidado

Evite comandos destrutivos no começo. Primeiro olhe o histórico, depois recupere só o arquivo necessário.

Git: cuidado com `git add .`

Por que evitar o ponto?

- Ele adiciona tudo que mudou.
- Pode incluir arquivos temporários.
- Pode vazar dados, senhas ou saídas grandes.

Melhor hábito

- Use `git status` antes.
- Use `git diff` para revisar.
- Adicione arquivos pelo nome.

Regra prática

Commit bom é pequeno e intencional: só entra o que você realmente quer salvar.

Git: o papel do `.gitignore`

O que é?

- Lista arquivos que o Git deve ignorar.
- Evita commitar lixo local.
- Protege o repositório de arquivos grandes.

Comum em Python/ML

- `__pycache__/` e `*.pyc`
- `.venv/` e arquivos `.env`
- `.ipynb_checkpoints/`
- `data/`, `models/`, `checkpoints/`

Ideia principal

Código e configuração entram no Git; arquivos gerados, secretos e muito grandes ficam fora.

Subseção

uv

Ambiente e dependências do projeto

Gerenciamento de pacotes: `uv`

Por que usar `uv` e não `pip`?

- Mais rápido para instalar e resolver dependências.
- Cria e gerencia ambientes virtuais.
- Mantém o projeto mais reproduzível.
- Evita conflitos comuns de `pip` instalado no usuário.

Comandos básicos

- `uv init`: cria um projeto Python.
- `uv add numpy`: adiciona uma dependência.
- `uv sync`: instala o ambiente do projeto.
- `uv run python -m src.models.rf`: roda no ambiente.

`pyproject.toml`

Arquivo central do projeto: guarda metadados, versão do Python e dependências.

Exemplo de pyproject.toml

```
[project]
name = "meu-projeto"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = [
  "numpy",
  "pandas",
  "scikit-learn",
]
```

Seções principais

- `[project]`: nome, versão e metadados.
- `requires-python`: versão esperada do Python.
- `dependencies`: pacotes usados pelo projeto.
- O `uv.lock`: trava versões instaladas.

Ideia principal

O `pyproject.toml` descreve o ambiente; `uv sync` monta e `uv run` usa esse ambiente.

Subseção

Organização de Pastas

Código separado dos arquivos gerados

Organização de pastas: exemplo

```
meu-projeto/
  pyproject.toml
  uv.lock
  .gitignore
  README.md

  src/
    data/
      generate_sintetic_data.py
      process.py
    models/
      rf.py
    eval/
      metrics.py
    utils/
      files.py

  data/
  models/
  reports/
```

Ideia

- `src/`: código do projeto.
- `src/data/`: preparação dos dados.
- `src/models/`: treino dos modelos.
- `src/eval/`: métricas e avaliação.
- `src/utils/`: funções compartilhadas.
- Pastas fora de `src/`: outputs gerados.

Organização de pastas: fluxo do pipeline

Etapas

- `src/data/process.py`: lê dados brutos.
- Salva dados tratados em `data/`.
- `src/models/rf.py`: lê `data/`.
- Salva modelos em `models/`.

Depois do modelo

- `src/eval/metrics.py`: lê dados e modelo.
- Calcula métricas de avaliação.
- Salva tabelas e figuras em `reports/`.
- Cada etapa recebe inputs e salva outputs.

Fluxo

`src/data` → `data/` → `src/models` → `models/` → `src/eval` → `reports/`

Organização de pastas: o que vai para o Git?

Versionar

- `src/`
- `pyproject.toml`
- `uv.lock`
- `.gitignore`
- `README.md`
- Arquivos pequenos de configuração.

Não versionar

- Dados gerados em `data/`.
- Modelos salvos em `models/`.
- Figuras e relatórios grandes.
- Cache, checkpoints e arquivos temporários.

Ideia principal

Git guarda a receita. Os artefatos grandes ou gerados devem ser reconstruíveis pela receita.

Subseção

Scripts

Execução, argumentos e funções reutilizáveis

Scripts: rodar da raiz

Regra

- Abra o terminal na raiz do projeto.
- Rode scripts com `uv run`.
- Use `python -m` para módulos em `src/`.

Por que isso ajuda?

- Caminhos relativos ficam previsíveis.
- Imports do projeto funcionam melhor.
- O ambiente vem do `pyproject.toml`.

Exemplo

```
uv run python -m src.models.rf
```

Scripts: o que faz `-m`?

Sem `-m`

- `python src/models/rf.py`
- Roda um arquivo pelo caminho.
- É simples, mas imports do projeto podem ficar chatos.

Com `-m`

- `python -m src.models.rf`
- Roda um módulo Python.
- Ajuda a importar funções de `src/utils/`.

Regra

Rode da raiz do projeto: `uv run python -m src.models.rf`

Scripts: argumentos com `argparse`

Por que usar?

- Evita editar o código para cada teste.
- Deixa inputs e outputs explícitos.
- Facilita repetir experimentos.

Exemplos

- `-input data/raw.csv`
- `-output reports/metrics.json`
- `-seed 42`
- `-n-estimators 200`

Exemplo

```
uv run python -m src.models.rf -n-estimators 200 -max-depth 6
```

Scripts: salvar modelos com `pickle`

No treino

- Treina o modelo em `src/models/rf.py`.
- Salva o objeto Python em `models/rf.pkl`.
- Usa `pickle.dump`.

Na avaliação

- Lê o modelo salvo em `src/eval/metrics.py`.
- Usa `pickle.load`.
- Calcula métricas e salva em `reports/`.

Cuidado

Arquivos `.pkl` são artefatos gerados: normalmente ficam fora do Git.

Scripts: `utils/` e imports

Para que serve `utils/`?

- Guarda funções usadas em várias etapas.
- Evita copiar e colar código.
- Exemplo: salvar JSON e pickle.

Como chamar?

- `src/utils/files.py`: funções de arquivo.
- `from src.utils.files import save_json`
- Rode com `python -m` da raiz.

Estilo dos scripts

Eu prefiro: scripts lineares, rodando de cima a baixo, sem `if __name__`.

Seção

Fundamentos de ML

Conceitos essenciais para treinar e avaliar modelos

Subseção

Conceitos básicos

Entradas e respostas usadas no aprendizado

Banco de dados

O que temos?

Um banco de dados é formado por pares:

$$(X_1, y_1), (X_2, y_2), \ldots, (X_n, y_n)$$

- $\mathtt{X}$: variáveis de entrada, features ou atributos.
- Ex.: $X_i = (\text{idade}, \text{renda}, \text{área})$, geralmente um vetor.
- $\mathtt{y}$: variável resposta, alvo ou label.
- Assumimos que os pares são i.i.d.: independentes e com a mesma distribuição.
- Em geral, pensamos que os dados vêm de uma distribuição desconhecida $\mathbb{P}$.

Regressão

- $\mathtt{y}$ é real.
- Ex.: preço, temperatura, demanda.

Classificação

- $\mathtt{y}$ é discreto.
- Ex.: classe, categoria, aprovado/reprovado.

Objetivo do aprendizado

O que queremos aprender?

A partir dos dados, queremos aprender uma função g tal que:

$$g(X) \approx y$$

- $g(X)$ é a predição feita pelo modelo para as entradas X.
- y é a resposta observada que queremos prever.
- A noção de $\approx$ é definida por uma função de perda.

Função de perda

Risco populacional

Uma função de perda ℓ mede o erro de uma predição. Buscamos uma função g que minimize a perda esperada $\mathbb{E}[\ell(g(X), Y)]$.

Perda quadrática

$$\mathcal{L}_{\text{quad}}(g) = \mathbb{E}\left[(g(X) - Y)^2\right]$$

- Captura fortemente erros grandes; o ótimo prevê a média condicional.
- Vantagem: é suave e conveniente para otimização.
- Desvantagem: é sensível a outliers.

Perda absoluta

$$\mathcal{L}_{\text{abs}}(g) = \mathbb{E}\left[|g(X) - Y|\right]$$

- Captura o erro típico; o ótimo prevê a mediana condicional.
- Vantagem: é mais robusta a outliers.
- Desvantagem: não é diferenciável no erro zero.

Perdas para classificação binária

Cenário

Para $Y \in \{0, 1\}$, podemos interpretar $g(X) \in [0, 1]$ como a probabilidade prevista de $Y = 1$. Com limiar 0.5, $\widehat{Y}_g(X) = 1$ quando $g(X) \geq 0.5$.

Perda 0–1

$$\mathcal{L}_{0\text{-}1}(g) = \mathbb{P}\left(\widehat{Y}_g(X) \neq Y\right)$$

- Mede diretamente a proporção de classificações erradas.
- Útil para avaliar o modelo após escolher um limiar.
- Não é adequada para treinar: não é suave nem diferenciável.

Entropia cruzada binária

$$\mathcal{L}_{\mathrm{BCE}}(g) = \mathbb{E}\left[-Y \log g(X) - (1 - Y) \log(1 - g(X))\right]$$

- Penaliza previsões confiantes que estão erradas.
- Captura a qualidade das probabilidades previstas.
- É diferenciável e é a escolha usual para treinar classificadores binários.

O que cada perda considera ótimo?

Ótimo populacional

Se pudéssemos minimizar o risco populacional sobre todas as funções possíveis, cada perda levaria a uma predição ótima diferente.

Perda	**Predição ótima** $g^*(X)$	**O que ela captura**
Quadrática	$\mathbb{E}[Y \mid X]$	Média condicional de Y
Absoluta	$\text{med}(Y \mid X)$	Mediana condicional de Y
0–1	$\arg\max_{c \in \{0,1\}} \mathbb{P}(Y = c \mid X)$	Classe mais provável
Entropia cruzada	$\mathbb{P}(Y = 1 \mid X)$	Probabilidade condicional de $Y = 1$

Objetivo dos modelos de ML

Como $\mathbb{P}$ é desconhecida, um modelo de ML aprende uma aproximação de g^* a partir da amostra. A perda escolhida define qual noção de "melhor" queremos aproximar.

Da população à amostra

O que não conhecemos?

Na prática, não conhecemos a distribuição $\mathbb{P}$ nem as expectativas populacionais. Temos apenas uma amostra finita de dados.

Para simplificar, a partir daqui assumiremos a perda quadrática.

Conjunto de treino

Usamos os dados de treino para ajustar a função g, minimizando a perda empírica:

$$\widehat{g} = \arg\min_{g} \frac{1}{n_{\text{tr}}} \sum_{i \in \text{tr}} (g(X_i) - y_i)^2$$

Conjunto de calibração

Reservamos outro conjunto para avaliar o modelo e escolher decisões, como hiperparâmetros ou limiares.

$$\widehat{R}_{\text{cal}}(g) = \frac{1}{n_{\text{cal}}} \sum_{i \in \text{cal}} (g(X_i) - y_i)^2$$

Lei dos grandes números

Quando n_{cal} cresce, $\widehat{R}_{\text{cal}}(g)$ se aproxima do risco populacional $R(g) = \mathbb{E}[(g(X) - Y)^2]$.

Split: treino, calibração e teste

Regra principal

Dividimos os dados em conjuntos disjuntos. Cada conjunto tem uma função diferente; o conjunto de teste deve permanecer intocado até o fim.

Treino

- Ajusta os parâmetros de g.
- Minimiza a perda empírica.
- É usado repetidamente durante o ajuste.

Calibração

- Compara modelos e hiperparâmetros.
- Escolhe transformações, limiares e parada antecipada.
- Também é chamado de conjunto de validação.

Teste

- Avalia o modelo final, após todas as escolhas.
- Estima o desempenho fora da amostra.
- Não deve orientar decisões de modelagem.

Fluxo: treino $\rightarrow$ calibração $\rightarrow$ modelo final $\rightarrow$ teste. Consultar o teste antes da hora torna a avaliação otimista.

Exemplo: escolha do modelo

Candidatos

Suponha que, após treinar cada candidato, medimos a perda quadrática média no conjunto de calibração. Menor valor é melhor.

Modelo	**Hiperparâmetros**	$\widehat{R}_{\text{cal}}$
Floresta aleatória	`max_depth` $= 5$	0.24
Floresta aleatória	`max_depth` $= 15$	0.21
Rede neural	32 neurônios, `lr` $= 10^{-3}$	0.20
Rede neural	64 neurônios, `lr` $= 10^{-4}$	0.17

Decisão e avaliação final

Escolhemos a rede neural com 64 neurônios e `lr` $= 10^{-4}$, pois ela teve a menor perda de calibração. Só então avaliamos esse modelo no teste, obtendo, por exemplo, $\widehat{R}_{\text{teste}} = 0.19$. Esse é o valor reportado.

Alternativa: treino–teste e validação cruzada

Em vez de separar treino, calibração e teste, podemos separar apenas treino e teste. Dentro do treino, o k-fold faz o papel da calibração para escolher o modelo.

Como funciona o k-fold?

- Dividimos o treino em k partes (folds).
- Em cada rodada, treinamos em $k-1$ partes e validamos na parte restante.
- A média das k perdas escolhe a configuração; então retreinamos em todo o treino.

Vantagens

- Aproveita melhor os dados e reduz a dependência de um único split.

Desvantagens

- É mais caro: treinamos cada candidato k vezes.
- Séries temporais ou dados agrupados exigem folds especiais, para evitar vazamento.

Uso do teste: ele continua intocado durante o k-fold e é usado uma única vez, no fim, para reportar o desempenho final.

Exemplo de k-fold: $k = 3$

Passo a passo

Dividimos o conjunto de treino em três partes: F_1, F_2 e F_3. Para cada modelo, repetimos: treinar em duas partes e validar na parte restante.

Rodada	Treino	Calibração	Modelos avaliados
1	$F_2 \cup F_3$	F_1	RF e rede neural
2	$F_1 \cup F_3$	F_2	RF e rede neural
3	$F_1 \cup F_2$	F_3	RF e rede neural

Perda quadrática em cada fold

	F_1	F_2	F_3	média
RF	.24	.22	.25	.24
Rede neural	.20	.19	.21	.20

Decisão

A rede neural tem a menor média. Retreinamos a rede neural em $F_1 \cup F_2 \cup F_3$ e só então a avaliamos uma vez no conjunto de teste.

Subseção

Modelos

Escolha, complexidade e generalização

Modelos, hiperparâmetros e complexidade

O que vamos aprender?

Agora vamos estudar modelos de ML. Todo modelo tem hiperparâmetros: escolhas definidas antes do treino que controlam sua flexibilidade e complexidade.

Exemplos de hiperparâmetros

- Floresta aleatória: profundidade das árvores e número de árvores.
- Rede neural: número de camadas, neurônios e regularização.
- Esses valores não são aprendidos diretamente dos dados.

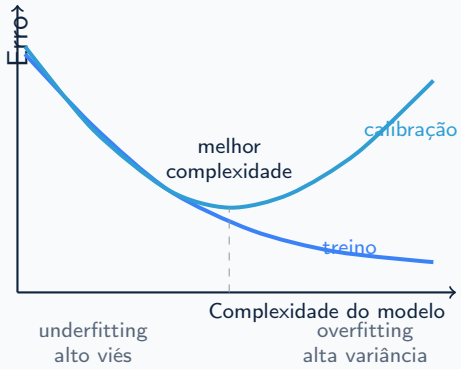
Efeito na complexidade

- Menor complexidade: modelo mais rígido, maior viés.
- Maior complexidade: modelo mais flexível, maior variância.
- A calibração escolhe os hiperparâmetros que melhor equilibram os dois.

Ideia principal

Hiperparâmetros controlam a complexidade; a complexidade determina o equilíbrio entre underfitting e overfitting.

Underfitting, overfitting e viés–variância



Underfitting

- Modelo simples demais para capturar o padrão.
- Erro alto em treino e calibração.

Overfitting

- Modelo se ajusta a ruído do treino.
- Erro de treino cai, mas o de calibração aumenta.

Escolha prática

Escolhemos a complexidade com menor erro de calibração: o equilíbrio entre viés e variância.

OLS: regressão linear por mínimos quadrados

Ideia

No OLS, assumimos que a média condicional da resposta é uma função linear das entradas:

$$\mathbb{E}[Y \mid X = x] = g_\beta(x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

O que o modelo representa?

- $x = (x_1, \ldots, x_p)$ reúne as features.
- β_0 é o intercepto; cada β_j descreve o efeito linear de x_j.
- OLS busca a melhor aproximação linear para a média de Y.

Como ajustamos?

Com a perda quadrática, escolhemos os coeficientes que minimizam o erro no treino:

$$\widehat{\beta} = \arg\min_{\beta} \sum_{i \in \mathrm{tr}} (y_i - g_\beta(X_i))^2 .$$

OLS: hiperparâmetros e uso prático

Ponto importante

OLS puro, via `LinearRegression`, não tem um hiperparâmetro de regularização. Sua complexidade vem principalmente das features que fornecemos ao modelo.

Controles de complexidade

- `PolynomialFeatures(degree=d)`: d maior cria mais termos; viés $\downarrow$, variância $\uparrow$.
- `interaction_only=True`: remove potências como x_j^2; menos features, portanto viés $\uparrow$ e variância $\downarrow$.
- Incluir transformações e interações também aumenta a capacidade do modelo.

Opções de `LinearRegression`

- `fit_intercept=False`: força a reta a passar pela origem; pode aumentar o viés se os dados não estiverem centrados.
- `positive=True`: impõe $\beta_j \geq 0$; reduz flexibilidade, aumentando viés e reduzindo variância.

Padronização: não é essencial para OLS simples, mas é recomendada com termos polinomiais ou escalas muito diferentes, por estabilidade e interpretação dos coeficientes.

OLS com features polinomiais

Estendendo o modelo linear

Podemos criar features como $x^2, x^3, \ldots, x^d$. Assim, a regressão linear passa a modelar curvas:

$$\mathbb{E}[Y \mid X = x] \approx \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d.$$

Ainda é um modelo linear

- A relação com x pode ser não linear.
- O modelo continua linear nos coeficientes $\beta_0, \ldots, \beta_d$.
- Ajustamos os coeficientes com OLS, usando as novas features.

O grau d é um hiperparâmetro

- Grau baixo: curva rígida, maior viés e possível underfitting.
- Grau alto: curva flexível, maior variância e risco de overfitting.
- Escolhemos d por calibração ou validação cruzada.

Exemplo simulado: escolhendo o grau d

Grau $d = 1$

Curva simples demais:
alto viés.

Grau $d = 3$

Boa aproximação:
menor erro de calibração.

Grau $d = 10$

Curva flexível demais:
ajusta também o ruído.

Escolha

Selecionamos o grau com menor erro de calibração; neste exemplo, $d = 3$.

Ridge: regularização L_2

Objetivo

Ridge adiciona ao erro quadrático uma penalidade para coeficientes grandes:

$$\widehat{\beta}^{\text{ridge}} = \arg\min_{\beta} \left\{ \sum_{i \in \text{tr}} (y_i - g_\beta(X_i))^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}.$$

Efeito da penalidade

- $\lambda = 0$: recuperamos OLS.
- Quanto maior λ, mais os coeficientes são encolhidos em direção a zero.
- Reduz variância e aumenta viés de forma controlada.

Normalmente, o intercepto β_0 não é penalizado.

Quando ajuda?

- Quando há muitas features ou features correlacionadas.
- Mantém todas as features, mas com efeitos menores e mais estáveis.
- Escolhemos λ por calibração ou validação cruzada.

Ridge: hiperparâmetros e uso prático

Principal hiperparâmetro: `alpha`

Em `Ridge(alpha=...)`, `alpha` corresponde a λ: valores maiores aumentam a regularização L_2.

Efeito sobre viés e variância

- `alpha=0`: equivale a OLS.
- `alpha` maior: coeficientes menores, viés $\uparrow$ e variância $\downarrow$.
- `positive=True`: restringe $\beta_j \geq 0$, reduzindo ainda mais a flexibilidade.

Outras escolhas relevantes

- `fit_intercept`: normalmente mantemos `True`; desativar sem centralizar os dados pode aumentar viés.
- `solver`, `tol` e `max_iter` controlam a otimização, não a complexidade, se houver convergência.

Padronização: é crucial em Ridge; sem ela, features em escalas maiores recebem uma penalização efetiva diferente. Avaliamos `alpha` por calibração.

Lasso: regularização L_1

Objetivo

Lasso penaliza a soma dos valores absolutos dos coeficientes:

$$\widehat{\beta}^{\text{lasso}} = \arg\min_{\beta} \left\{ \sum_{i \in \text{tr}} (y_i - g_\beta(X_i))^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}.$$

Efeito da penalidade

- $\lambda = 0$: recuperamos OLS.
- À medida que λ cresce, alguns coeficientes podem se tornar exatamente zero.
- Reduz variância e produz um modelo esparso.

Quando ajuda?

- Faz seleção automática de features.
- Pode tornar o modelo mais simples e interpretável.
- Com features muito correlacionadas, pode escolher uma e descartar as demais.

Assim como em Ridge, λ é um hiperparâmetro escolhido por calibração.

Lasso: hiperparâmetros e uso prático

Principal hiperparâmetro: `alpha`

Em `Lasso(alpha=...)`, `alpha` controla a penalidade L_1 e, portanto, o grau de esparsidade do modelo.

Efeito sobre viés e variância

- `alpha=0`: aproxima OLS, mas OLS é preferível numericamente.
- `alpha` maior: mais coeficientes ficam em zero; viés $\uparrow$ e variância $\downarrow$.
- A seleção de features fica mais agressiva à medida que `alpha` aumenta.

Outras escolhas relevantes

- `fit_intercept`: desativar sem centralizar pode aumentar viés.
- `positive=True`: impõe coeficientes não negativos e reduz a flexibilidade.

Padronização: é crucial em Lasso; sem ela, a seleção de features fica enviesada pela escala. `alpha` deve ser escolhido por calibração.

KNN: estimativa pelos vizinhos mais próximos

Ideia

Para uma nova entrada x, procuramos os k pontos de treino mais próximos, $\mathcal{N}_k(x)$, e usamos suas respostas como uma estimativa local.

Regressão

$$\widehat{g}_k(x) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x)} y_i \approx \mathbb{E}[Y \mid X = x].$$

A predição é a média local dos vizinhos.

Classificação binária

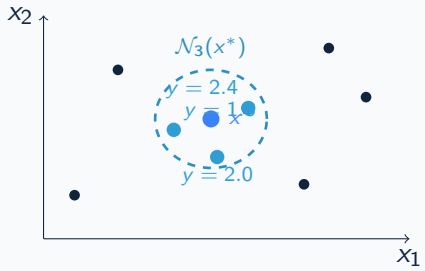
$$\widehat{p}_k(x) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x)} y_i \approx \mathbb{P}(Y = 1 \mid X = x).$$

A classe é definida por votação; por exemplo, classe 1 se $\widehat{p}_k(x) \geq 0.5$.

Modelo não paramétrico

KNN não ajusta uma única função global. Ele guarda os dados de treino e constrói uma estimativa diferente em cada região do espaço de features.

Exemplo simulado: KNN em regressão



Passo a passo

1. Escolhemos $k = 3$.
2. Os três vizinhos destacados têm $y = (1.8, 2.0, 2.4)$.
3. Calculamos a média:

$$\widehat{g}_3(x^*) = \frac{1.8 + 2.0 + 2.4}{3} = 2.07.$$

Se k fosse maior, incluiríamos pontos mais distantes e a estimativa ficaria mais suave.

KNN: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `n_neighbors=k`: k menor deixa o modelo mais flexível: viés $\downarrow$, variância $\uparrow$. k maior faz o oposto.
- `weights='distance'`: dá mais peso aos vizinhos próximos; aumenta a flexibilidade e pode aumentar a variância.
- `metric` e `p`: definem a noção de distância; devem refletir o domínio.

Padronização
É crucial: KNN depende de distâncias. Sem padronizar, uma feature de escala maior pode dominar os vizinhos.

Vantagens e quando usar

- Simples, não paramétrico e capaz de capturar relações locais não lineares.
- Bom para poucos atributos, dados densos e quando há exemplos parecidos perto de cada consulta.

Limitações e quando evitar

- Sofre com alta dimensionalidade, features irrelevantes e regiões pouco povoadas.
- Exige memória e pode ser lento para prever em bases grandes; é sensível a escala e outliers.

Escolhemos `n_neighbors`, `weights` e a métrica por calibração ou validação cruzada.

Árvore de decisão: estimativa por regiões

Ideia

A árvore divide o espaço de features em regiões $R_1, \ldots, R_M$ usando regras como $x_j \leq c$. Em cada folha, ela faz uma predição constante.

Regressão

Para $x \in R_m$, a folha usa:

$$\widehat{g}(x) = \frac{1}{n_m} \sum_{i: X_i \in R_m} y_i \approx \mathbb{E}[Y \mid X = x].$$

É a média dos alvos que caíram naquela região.

Classificação binária

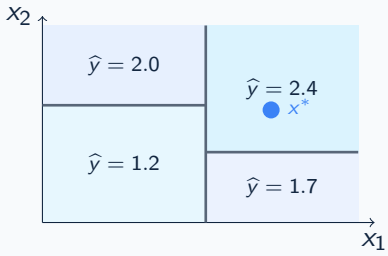
Para $x \in R_m$, a folha estima:

$$\widehat{p}(x) = \frac{1}{n_m} \sum_{i: X_i \in R_m} y_i \approx \mathbb{P}(Y = 1 \mid X = x).$$

A classe é definida pela proporção majoritária na folha.

Aprendizado: n_m é o número de observações na folha. A árvore escolhe cortes que reduzem erro quadrático (regressão) ou impureza, como Gini (classificação).

Exemplo simulado: árvore em duas dimensões



Caminho de x^*

1. $x_1^* = 4.2 > 3$: seguimos para a direita.
2. $x_2^* = 2.4 > 1.5$: seguimos para cima.
3. x^* cai na folha com $\widehat{y} = 2.4$.

A superfície prevista é formada por degraus: cada região recebe uma constante diferente.

Árvore: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `max_depth`: maior profundidade cria mais folhas; viés ↓, variância ↑.
- `min_samples_split` e `min_samples_leaf`: valores maiores impedem folhas pequenas; viés ↑, variância ↓.
- `ccp_alpha`: controla poda; valor maior simplifica a árvore e reduz variância.

Padronização
Não é necessária: os cortes dependem da ordem das features, não de sua escala. Transformações monotônicas preservam os cortes equivalentes.

Vantagens e quando usar

- Interpretável, captura não linearidades e interações sem criar features manualmente.
- Boa escolha para relações com regras e efeitos por faixas de valores.

Limitações e quando evitar

- Árvores profundas têm alta variância e mudam muito com pequenas alterações nos dados.
- Em regressão, não extrapolam além dos valores vistos; fronteiras são em degraus.

Escolhemos profundidade, tamanhos mínimos e poda por calibração ou validação cruzada.

Bagging: média de árvores bootstrap

Problema da árvore individual

Árvores profundas têm baixo viés, mas alta variância: pequenas mudanças nos dados podem produzir cortes e previsões muito diferentes.

Como Bagging funciona?

- Criamos B amostras bootstrap: sorteios com reposição de n observações; algumas repetem e outras ficam de fora.
- Treinamos uma árvore T_b em cada amostra.
- Em regressão, usamos a média das previsões:

$$\widehat{g}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x).$$

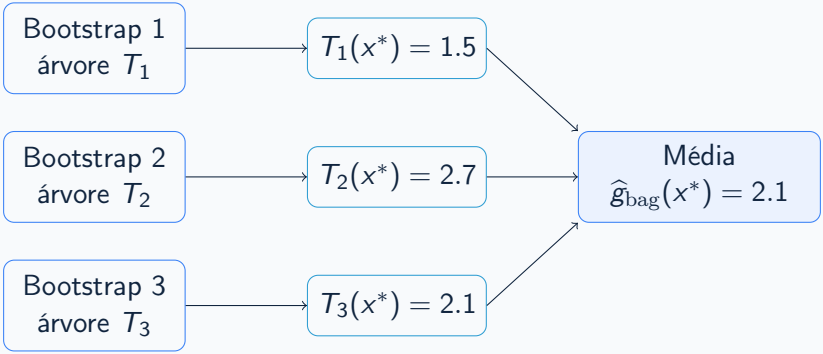
O que melhora?

- A média suaviza as variações das árvores individuais.
- Reduz variância com pouca alteração no viés.

Problema restante

Árvores treinadas em dados parecidos ainda podem ser correlacionadas e cometer erros semelhantes.

Exemplo simulado: Bagging reduz variância



Leitura

Cada árvore é instável, mas a média é mais estável. Quanto menos correlacionadas forem as árvores, maior a redução de variância.

Bagging: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `n_estimators`: mais árvores reduzem variância até estabilizar, com maior custo.
- `max_samples`: amostras menores aumentam diversidade, mas podem aumentar viés.
- `bootstrap`: define se cada estimador usa amostra com reposição.
- A complexidade de cada árvore vem do estimador base, como `max_depth`.

Padronização
Não é necessária quando o estimador base é uma árvore.

Vantagens e quando usar

- Reduz a alta variância de estimadores instáveis, como árvores profundas.
- É paralelizável e funciona bem como melhoria direta de uma árvore.

Limitações

- Árvores correlacionadas limitam o ganho da média.
- Perde interpretabilidade em relação a uma árvore e aumenta custo de memória.

Escolhemos o número de árvores, tamanho da amostra e complexidade do estimador base por calibração.

Random Forest: Bagging com features aleatórias

O que Random Forest resolve?

Bagging reduz a variância das árvores, mas elas podem continuar correlacionadas se todas escolhem a mesma feature forte nos primeiros cortes. Random Forest reduz essa correlação.

Mecanismo

- Para cada árvore, usamos uma amostra bootstrap.
- Em cada nó, sorteamos apenas um subconjunto de features candidatas.
- A árvore escolhe o melhor corte somente dentro desse subconjunto.

Resultado

- As árvores ficam mais diversas e menos correlacionadas.
- A média reduz mais variância do que em Bagging simples.
- Pode aumentar um pouco o viés de cada árvore, mas melhora a generalização do conjunto.

$$\widehat{g}_{\mathrm{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b^{\mathrm{RF}}(x).$$

Exemplo simulado: de Bagging a Random Forest

Bagging

raiz: $x_1 \leq c$

raiz: $x_1 \leq c$

raiz: $x_1 \leq c$

Mesma feature forte
$\Rightarrow$ árvores correlacionadas

Random Forest

raiz: $x_1 \leq c$

raiz: $x_2 \leq c$

raiz: $x_3 \leq c$

Features sorteadas
$\Rightarrow$ árvores menos correlacionadas

Conclusão

Random Forest mantém o bootstrap do Bagging e adiciona aleatoriedade nas features para tornar a média mais eficiente.

Random Forest: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `n_estimators`: mais árvores reduzem variância até estabilizar.
- `max_features`: menos features por corte reduz correlação; pode elevar viés e reduzir variância.
- `max_depth`, `min_samples_leaf` e `ccp_alpha` controlam a complexidade das árvores.
- `bootstrap` e `max_samples` controlam a amostragem das observações.

Padronização
Não é necessária: os cortes de árvores não dependem da escala das features.

Vantagens e quando usar

- Forte baseline para dados tabulares, com não linearidades e interações.
- Geralmente generaliza melhor e é mais estável que uma árvore única.
- Fornece importância natural das features pela redução acumulada da impureza.

Limitações

- Menos interpretável, mais lento e mais pesado que uma árvore individual.
- Em regressão, não extrapola; pode ser superado por boosting em alguns problemas tabulares.

Escolhemos `max_features`, número de árvores e complexidade por calibração ou validação cruzada.

Extra Trees: aleatoriedade adicional nos cortes

Da Random Forest para Extra Trees

Random Forest sorteia features e busca o melhor limiar entre elas. Extra Trees também sorteia as features, mas usa limiares aleatórios para criar cortes ainda mais diversos.

O que muda?

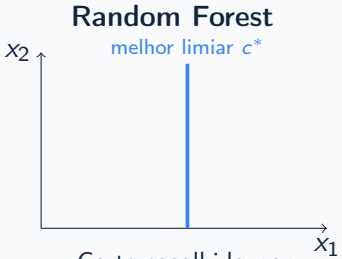
- RF: para cada feature candidata, procura o melhor limiar.
- Extra Trees: sorteia limiares e escolhe o melhor entre esses cortes aleatórios.
- Por padrão, `bootstrap=False`: cada árvore pode usar toda a amostra.

Efeito

- Árvores ainda menos correlacionadas $\Rightarrow$ menor variância do conjunto.
- Cada árvore é um pouco menos otimizada $\Rightarrow$ viés pode aumentar.
- A média das árvores compensa a instabilidade dos cortes aleatórios.

$$\widehat{g}_{\mathrm{ET}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b^{\mathrm{extra}}(x).$$

Exemplo simulado: limiares em RF e Extra Trees



Corte escolhido por otimização local

Cortes aleatórios aumentam a diversidade

Leitura

Com muitas árvores, os cortes aleatórios são suavizados pela média; o ganho vem da menor correlação entre árvores.

Extra Trees: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `n_estimators`: mais árvores reduzem variância até estabilizar.
- `max_features`: menos features por nó aumenta diversidade; pode elevar viés e reduzir variância.
- `max_depth`, `min_samples_leaf` e `ccp_alpha` regulam a complexidade das árvores.
- `bootstrap`: habilita bootstrap; o padrão de Extra Trees é `False`.

Padronização
Não é necessária: árvores continuam baseadas em limiares das features.

Vantagens e quando usar

- Ensemble rápido e robusto para dados tabulares com relações não lineares.
- Útil quando queremos reduzir correlação e variância sem ajustar muitos detalhes.

Limitações

- Menos interpretável que uma árvore e pode introduzir viés maior que RF.
- Em regressão, não extrapola; continua exigindo memória e várias árvores para prever.

Escolhemos número de árvores, features por nó e complexidade por calibração ou validação cruzada.

Gradient Boosting: correção sequencial de erros

Ideia

Em vez de treinar árvores independentes e fazer a média, Gradient Boosting constrói um modelo aditivo: cada nova árvore tenta corrigir os resíduos deixados pelo conjunto anterior.

Em regressão com erro quadrático

- Começamos com uma previsão simples g_0, como a média de Y.
- No estágio m, calculamos resíduos $r_i = y_i - g_{m-1}(X_i)$.
- Ajustamos uma árvore h_m aos resíduos.

Atualização

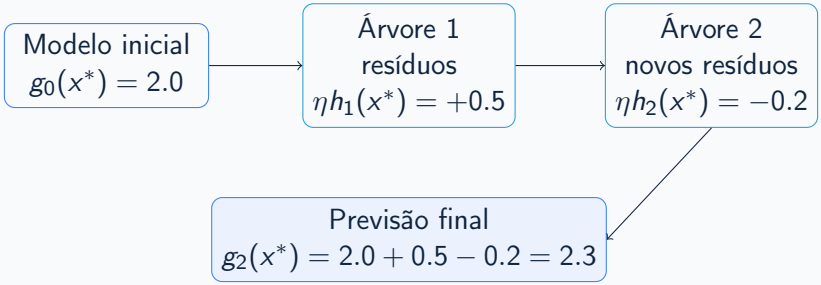
$$g_m(x) = g_{m-1}(x) + \eta h_m(x),$$

onde η é a taxa de aprendizado. Cada árvore reduz o erro que ainda restou; em classificação, o princípio é o mesmo, usando o gradiente da perda logística.

Diferença para Random Forest

RF reduz variância pela média de árvores paralelas; Gradient Boosting reduz viés ao corrigir erros sequencialmente.

Exemplo simulado: estágios do Gradient Boosting



Leitura

Árvores pequenas fazem correções graduais. Muitas correções podem aprender padrões úteis, mas também podem começar a ajustar ruído.

Gradient Boosting: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `n_estimators`: mais estágios reduzem viés, mas podem aumentar variância.
- `learning_rate`: η menor exige mais árvores e regulariza cada correção.
- `max_depth`, `min_samples_leaf` e `ccp_alpha` controlam cada árvore base.
- `subsample<1.0` introduz aleatoriedade e pode reduzir variância.

Vantagens e quando usar

- Muito forte em dados tabulares, aprendendo não linearidades e interações complexas.
- Pode alcançar baixo viés com árvores rasas e muitas etapas bem regularizadas.

Limitações

- Treino sequencial: mais lento e menos paralelizável que RF.
- Sensível a hiperparâmetros; muitos estágios ou taxa alta podem causar overfitting.

Padronização
Não é necessária: as árvores usam cortes por limiares, não distâncias.
Escolhemos taxa de aprendizado, número de estágios e complexidade por calibração ou validação cruzada.

SVM: margem em classificação e tolerância em regressão

Classificação: SVC

- Procura uma fronteira que separe as classes com a maior margem possível.
- Os pontos próximos da fronteira são os vetores de suporte; eles determinam a solução.
- Com kernels, a fronteira pode ser não linear no espaço original.

$$f(x) = \sum_{i \in \mathrm{SV}} \alpha_i y_i K(x_i, x) + b.$$

Regressão: SVR

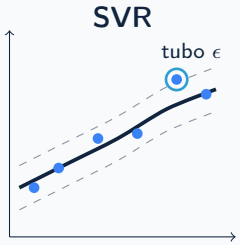
- Procura uma função com um tubo de largura 2ϵ.
- Erros dentro do tubo não recebem penalidade.
- Pontos fora do tubo são vetores de suporte e ajustam a curva.

$$\ell_\epsilon(y, \widehat{y}) = \max\{0, |y - \widehat{y}| - \epsilon\}.$$

Ideia comum

SVM controla a complexidade por margem, regularização e kernel, buscando uma solução definida principalmente pelos pontos mais informativos.

Exemplo simulado: margem e tubo ϵ



Pontos circulados são vetores de suporte: os mais importantes para definir a fronteira ou a curva.

SVM: hiperparâmetros e uso prático

Hiperparâmetros no scikit-learn

- `C`: maior C penaliza mais erros; viés ↓, variância ↑.
- `kernel`: define a forma da fronteira; `rbf` permite não linearidade.
- `gamma` (RBF): maior torna a influência mais local; viés ↓, variância ↑.
- Em SVR, `epsilon` maior amplia o tubo; viés ↑, variância ↓.

Padronização
É crucial: margens, kernels e distâncias dependem da escala das features.

Vantagens e quando usar

- Eficaz em dados de dimensão moderada, especialmente com fronteiras complexas e poucos exemplos.
- A solução depende só de vetores de suporte e pode ser compacta.

Limitações

- Treino pode ficar caro em bases grandes; exige boa escolha de C, kernel e `gamma`.
- Menos interpretável e não fornece probabilidades calibradas por padrão em SVC.

Escolhemos C, kernel, `gamma` e `epsilon` por calibração ou validação cruzada.

Seção

Avaliação

Como medir se um modelo realmente é útil

Subseção

Métricas de classificação

Métricas para decisões, probabilidades e classes desbalanceadas

Acurácia: simples, mas pode enganar

Definição

A acurácia é a fração de previsões corretas:

$$\mathrm{Acc} = \frac{TP + TN}{TP + TN + FP + FN}.$$

- É intuitiva e funciona bem quando as classes têm frequências e custos semelhantes.
- Resume tudo em uma única proporção de acertos.

Problema: classe desbalanceada

Suponha 1.000 pacientes, dos quais apenas 20 têm a doença. Um modelo que sempre prevê *sem doença* alcança

$$\frac{980}{1000} = 98\% \quad \text{de acurácia,}$$

mas encontra **zero** casos positivos.

Pergunta certa

Além de acertar no total, quantos positivos encontramos e quão confiáveis são os alertas?

Matriz de confusão: de onde vêm os acertos e erros?

		Classe verdadeira	
	Predição	Positiva	Negativa
	Positiva	**TP**	**FP**
	Negativa	**FN**	**TN**

- **TP**: detectou corretamente um positivo.
- **TN**: descartou corretamente um negativo.
- **FP**: falso alarme (predisse positivo, mas não era).
- **FN**: perdeu um positivo que importava encontrar.

A matriz torna explícito qual tipo de erro o modelo comete – e permite escolher a métrica adequada ao problema.

Da probabilidade à classe: escolhendo o corte

Muitos classificadores produzem uma probabilidade estimada

$$\widehat{p}(x) = \widehat{\mathbb{P}}(Y = 1 \mid X = x).$$

Para obter uma classe, escolhemos um corte t:

$$\widehat{y}_t(x) = 1\{\widehat{p}(x) \geq t\}\,.$$

O valor $t = 0{,}5$ é apenas uma convenção – não uma regra universal.

O corte decide o compromisso

- **Corte menor**: mais exemplos viram positivos; recall tende a subir e precisão a cair.
- **Corte maior**: alertas são mais seletivos; precisão tende a subir e recall a cair.

Como escolher?

Use o conjunto de calibração e os custos do problema. Em triagem médica, perder um caso (FN) pode ser bem mais caro que um falso alarme (FP).

Precisão, recall e F1: qual erro importa mais?

Precisão

$$\frac{TP}{TP + FP}$$

Entre os alertas positivos, quantos eram corretos?
Sozinha não basta: prever positivo raramente pode dar alta precisão e ainda perder muitos casos.

Recall

$$\frac{TP}{TP + FN}$$

Entre os positivos reais, quantos foram encontrados?
Sozinho não basta: prever tudo como positivo obtém recall de 100%, com muitos falsos alarmes.

F1-score

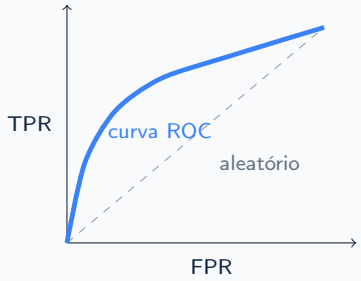
$$F_1 = 2\frac{P \cdot R}{P + R}$$

Média harmônica: só é alta quando **ambos** são altos.
Útil para balancear FP e FN, sobretudo com classes desbalanceadas.

O limiar muda o compromisso

Diminuir o limiar costuma aumentar recall e reduzir precisão; elevar o limiar faz o oposto.

ROC e AUC: avaliar todos os limiares



Curva ROC
Para cada limiar, plota

$$\mathrm{TPR} = \frac{TP}{TP + FN},$$
$$\mathrm{FPR} = \frac{FP}{FP + TN}.$$

AUC é a área sob a curva: a probabilidade de o modelo ranquear um positivo acima de um negativo aleatório. AUC próxima de 1 é boa; 0,5 equivale ao acaso. ROC–AUC mede ordenação, mas não escolhe o limiar nem incorpora custos de erro.