Árvore de Decisão Teoria e Prática com Python

Giseldo Neo

2026-03-11

UMA ABORDAGEM OBJETIVA

ÁRVORE DE DECISÃO COM PYTHON

GISELDO NEO

ALANA NEO

Índice

Capítulo 1

Árvore de decisão com Python

Bem Vindos!

Baixar PDF

1.1 Resumo

As árvores de decisão são um dos métodos de aprendizado de máquina supervisionado mais populares e práticos, sendo amplamente utilizadas principalmente para tarefas de classificação (identificar a qual categoria um elemento pertence). Elas resolvem problemas utilizando uma estratégia de "dividir para conquistar", onde um problema complexo é decomposto recursivamente em subproblemas mais simples.

A representação de uma árvore de decisão é muito natural e intuitiva, sendo composta por três elementos principais:

- Nós internos (incluindo o nó raiz): Cada nó especifica um teste sobre um determinado atributo ou característica da instância (por exemplo, "Perspectiva do tempo", "Idade" ou "Salário").
- Ramos (ou arestas): Cada ramo que desce de um nó corresponde a um dos possíveis resultados ou valores desse teste (por exemplo, "Ensolarado" ou "Nublado").
- Nós folha: Ficam nas extremidades da árvore e especificam o valor final da decisão, ou seja, a classe atribuída à instância (por exemplo, "Sim" ou "Não").

Para classificar um novo exemplo, o processo começa no nó raiz. O sistema avalia o atributo

daquele nó e segue o ramo correspondente ao valor que a instância possui. Esse teste é repetido nos próximos nós descendentes até que se alcance um nó folha, o qual fornecerá a classificação final.

Cada caminho percorrido da raiz até uma folha pode ser traduzido facilmente em uma regra lógica do tipo SE ENTÃO (ex: SE Perspectiva = Ensolarado E Umidade = Alta ENTÃO JogarTênis = Não), o que torna os modelos altamente interpretáveis por seres humanos.

A construção do modelo a partir dos dados de treinamento geralmente é feita de forma descendente (top-down). O passo mais crucial dos algoritmos formadores de árvores (como ID3, C4.5 e CART) é decidir qual atributo testar em cada nó. O objetivo é escolher o atributo que melhor separa ou discrimina os exemplos de acordo com suas classes.

Para fazer essa escolha, utilizam-se cálculos estatísticos:

- Entropia: É uma medida da impureza, desordem ou aleatoriedade de uma coleção de exemplos. Se um nó contém exemplos de várias classes misturadas, a entropia é alta. Se todos os exemplos forem da mesma classe, a entropia é zero.
- Ganho de Informação: Mede a redução esperada na entropia se particionarmos os dados usando um determinado atributo. O algoritmo calcula o ganho de informação para todos os atributos disponíveis e escolhe aquele que apresentar o maior valor (ou seja, o que melhor limpa a "impureza" dos dados) para ser o nó de decisão. Nota: Outras métricas matemáticas também podem ser usadas dependendo do algoritmo, como a Razão de Ganho ou o Índice Gini.

O algoritmo repete esse processo de particionamento e escolha de atributos para cada novo subconjunto de dados criado, até que todos os exemplos de um ramo pertençam à mesma classe ou até que não restem mais atributos para testar.

Um problema comum na indução de árvores de decisão é o sobre-ajustamento (overfitting). Isso ocorre quando a árvore cresce demais e se ajusta perfeitamente aos dados de treinamento, memorizando inclusive ruídos e erros, o que a faz perder a capacidade de generalizar e classificar corretamente novos dados. Para combater isso, aplicam-se técnicas de poda (pruning), que consistem em interromper o crescimento da árvore cedo ou remover ramos inteiros após a sua construção, transformando-os em folhas e deixando a árvore mais simples e confiável.

1.2 Objetivos

Ao final do estudo, espera-se que o leitor seja capaz de:

- explicar o que é uma árvore de decisão e como ela realiza previsões;

- diferenciar classificação, regressão, pureza, impureza e ganho de informação;
- compreender o papel de algoritmos clássicos como ID3, C4.5 e CART;
- treinar, visualizar, avaliar e podar árvores no `scikit-learn`;
- interpretar regras aprendidas e comunicar resultados para público técnico e não técnico;
- usar árvores como modelo final ou como etapa de análise exploratória.

1.3 Público-alvo

Este material foi escrito para:

- estudantes de ciência de dados, estatística, computação e engenharias;
- professores e instrutores que desejam uma base organizada para aulas;
- analistas que precisam justificar previsões com regras compreensíveis;
- profissionais em transição para machine learning que preferem modelos interpretáveis;
- leitores que desejam unir teoria e prática em um mesmo fluxo de aprendizado.

1.4 Conceitos fundamentais

Ao longo do livro, vamos aprofundar alguns conceitos centrais:

- impureza: mede o quanto as classes estão misturadas em um nó;
- critério de divisão: regra matemática usada para escolher a melhor pergunta;
- ganho de informação: quanto a divisão reduz a incerteza;
- pré-poda: limitar o crescimento antes que a árvore fique grande demais;
- pós-poda: reduzir a árvore depois do treino;
- generalização: capacidade de funcionar bem em dados novos.

1.5 Requisitos técnicos

Para acompanhar os exemplos de código, recomenda-se Python 3.10+ e as bibliotecas abaixo.

```
pip install pandas numpy matplotlib seaborn scikit-learn
```

Exemplo de código python.

```
print("Olá Mundo.")
```

```
Olá Mundo.
```

1.6 Orientação de estudo

Aproveite o livro como uma apostila de estudo progressivo.

- Leia os capítulos teóricos com calma antes de executar o código.
- Refaça os cálculos numéricos manualmente quando possível.
- Compare árvores simples e complexas em cada experimento.
- Tente verbalizar as regras aprendidas como se estivesse explicando para outra pessoa.
- Ao final de cada capítulo, pergunte-se não apenas “como fazer”, mas também “por que o modelo fez isso”.

Capítulo 2

Fundamentos de Árvore de Decisão

Árvores de decisão estão entre os modelos mais didáticos de aprendizado supervisionado. Elas aprendem regras do tipo "se… então…" a partir de dados rotulados e organizam essas regras em uma estrutura hierárquica formada por perguntas sucessivas.

A grande vantagem pedagógica desse modelo é que ele permite enxergar com relativa clareza o caminho que levou a uma previsão. Em vez de parecer uma caixa-preta, a árvore explicita quais atributos foram usados, em que ordem eles foram testados e por que determinados grupos foram separados.

2.1 O que é uma árvore de decisão

Uma árvore de decisão recebe exemplos descritos por atributos e produz uma saída. Em problemas de classificação, a saída é uma classe, como `aprovar` ou `reprovar`, `fraude` ou `não fraude`, `doente` ou `saudável`. Em problemas de regressão, a saída é um valor numérico, como preço, demanda ou tempo.

A ideia central é simples:

- começamos com todos os exemplos em um único conjunto;
- escolhemos a pergunta que melhor separa esse conjunto;
- dividimos os dados em subconjuntos menores;
- repetimos o processo recursivamente até que os grupos fiquem suficientemente homogêneos.

A árvore nasce pela raiz, cresce em direção às folhas e, a cada etapa, busca um teste que reduza a mistura de classes nos subconjuntos gerados.

2.2 Intuição com um problema simples

Imagine um sistema de análise de crédito. Cada cliente possui atributos como renda, histórico de inadimplência, tempo de emprego e comprometimento de renda. A tabela abaixo descreve esse conjunto de dados:

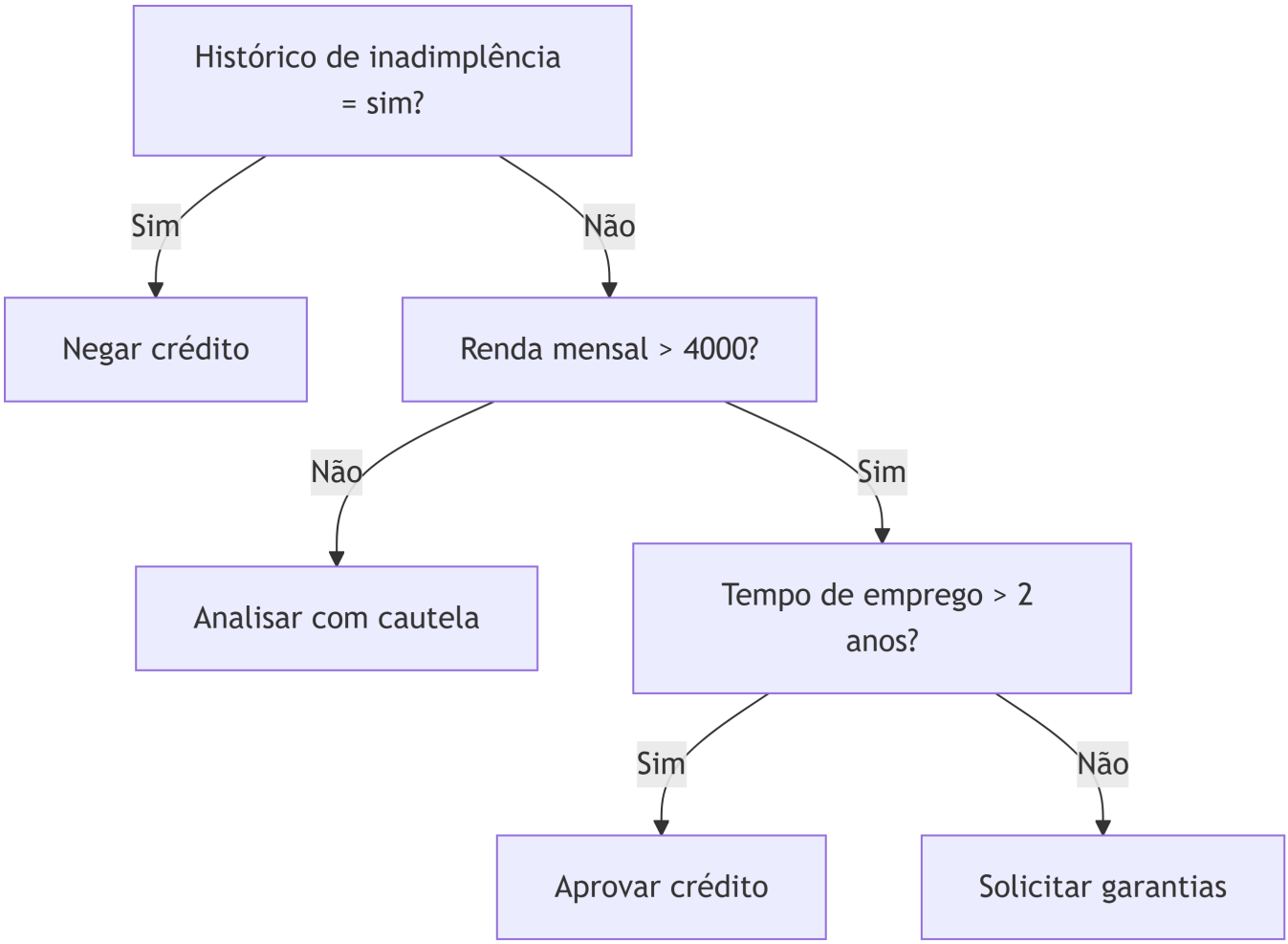
Atributo	Tipo	Valores / Exemplo	Papel
Histórico de inadimplência	Categórico	Sim, Não	Entrada
Renda mensal	Numérico	Ex.: R$ 3.500, R$ 5.200	Entrada
Tempo de emprego	Numérico	Ex.: 1 ano, 3 anos	Entrada
Comprometimento de renda	Numérico	Ex.: 30%, 55%	Entrada
Decisão de crédito	Categórico	Aprovar, Negar, Analisar com cautela, Solicitar garantias	Saída (alvo)

A tabela abaixo mostra exemplos fictícios desse conjunto de dados:

Cliente	Histórico de inadimplência	Renda mensal	Tempo de emprego	Comprometimento de renda	Decisão de crédito
Ana	Não	R$ 5.200	4 anos	28%	Aprovar crédito
Bruno	Sim	R$ 4.800	6 anos	35%	Negar crédito
Carla	Não	R$ 3.100	2 anos	52%	Analisar com cautela
Diego	Não	R$ 4.500	1 ano	40%	Solicitar garantias
Elisa	Não	R$ 6.300	5 anos	22%	Aprovar crédito

Uma árvore pode aprender algo como:

```
Histórico de inadimplência = sim?
   Sim: Negar crédito
   Não: Renda mensal > 4000?
       Não: Analisar com cautela
       Sim: Tempo de emprego > 2 anos?
           Sim: Aprovar crédito
           Não: Solicitar garantias
```

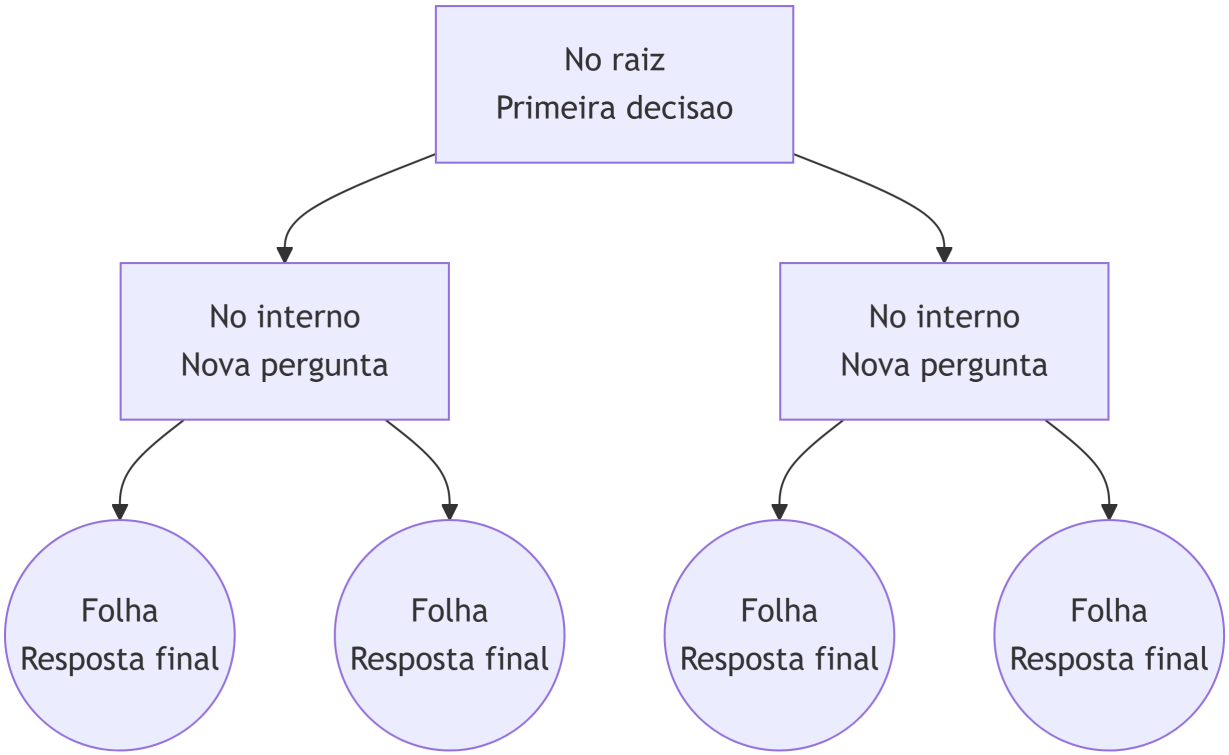


Esse tipo de estrutura aproxima muito o modelo do raciocínio humano. As árvores podem ser apresentadas como sequências de perguntas, ou como os atributos do conjunto de dados. Cada pergunta reduz a incerteza sobre a classe correta.

2.3 Componentes da árvore

Uma árvore possui três elementos fundamentais.

- **Nó raiz**: É o primeiro teste realizado. Em geral, corresponde ao atributo que mais ajuda a separar os dados logo no início.
- **Nós internos**: São testes intermediários. Cada nó interno recebe um subconjunto de exemplos e decide qual nova pergunta deve ser feita naquele ponto.
- **Folhas**: São as respostas finais. Em classificação, a folha guarda a classe prevista. Em regressão, guarda um valor médio ou representativo do grupo.



2.4 Como uma instância percorre a árvore

Classificar um novo exemplo, ou seja realizar uma inferência, significa descer pela árvore.

1. O processo começa no nó raiz.
2. O valor do atributo correspondente é observado.
3. O exemplo segue pelo ramo compatível com esse valor.
4. O processo continua até chegar a uma folha.

Se uma instância chega a uma folha com rótulo `sim`, então a previsão final é `sim`. Se chegar a uma folha com valor numérico, esse valor se torna a previsão.

Essa interpretação operacional é importante porque mostra que uma árvore é, ao mesmo tempo, um modelo de previsão e um conjunto de regras executáveis.

2.5 Representação como regras simbólicas

Um ponto muito forte das árvores de decisão é sua equivalência com regras lógicas. Cada caminho da raiz até uma folha pode ser lido como uma conjunção de condições. O conjunto de caminhos pode ser visto como uma disjunção dessas conjunções.

Exemplo:

```
Se perspectiva = ensolarado e umidade = normal, então jogar = sim
Se perspectiva = nublado, então jogar = sim
Se perspectiva = chuvoso e vento = fraco, então jogar = sim
```

Isso ajuda bastante em domínios nos quais explicabilidade importa, como crédito, saúde, auditoria, diagnóstico e sistemas especialistas.

2.6 Por que árvores são tão populares

Árvores de decisão ficaram famosas por várias razões práticas.

- São fáceis de entender e explicar.
- Exigem menos preparação de dados do que muitos outros modelos.
- Lidam bem com relações não lineares.
- Podem trabalhar com atributos numéricos e categóricos.
- Servem como ponto de partida para modelos mais avançados, como Random Forest e Gradient Boosting.

No contexto educacional, elas também são uma excelente porta de entrada para aprender conceitos centrais de machine learning: treino, teste, generalização, ruído, viés, overfitting e seleção de atributos.

2.7 Tipos de problema em que costumam funcionar bem

Árvores costumam ser especialmente adequadas quando:

- os exemplos podem ser descritos como pares atributo-valor;
- a tarefa é de classificação discreta ou regressão estruturada;
- a interpretação do modelo importa tanto quanto a acurácia;
- existe possibilidade de interações não lineares entre variáveis;
- há necessidade de comunicar regras para pessoas não técnicas.

Elas também aparecem com frequência em triagem médica, marketing, previsão de churn, análise de risco, detecção de inadimplência, classificação de documentos e apoio à decisão em processos internos.

2.8 Vantagens

- **Interpretabilidade**: Talvez seja a principal vantagem. Em muitos cenários, o valor da árvore não está apenas na previsão, mas em explicar por que aquela previsão foi feita.
- **Flexibilidade**: A árvore consegue criar fronteiras de decisão complexas sem exigir normalização, padronização ou engenharia sofisticada logo no início.
- **Pouca preparação inicial**: Modelos lineares costumam pedir mais cuidado com escala e relações funcionais. A árvore pode ser treinada rapidamente para produzir um baseline forte.
- **Captura de interações**: Se um atributo só faz sentido quando combinado com outro, a estrutura hierárquica da árvore naturalmente consegue modelar isso.

2.9 Limitações

Apesar da simplicidade, árvores isoladas também possuem fraquezas importantes.

- **Tendência a overfitting**: Se crescer demais, a árvore pode memorizar peculiaridades do conjunto de treino em vez de aprender padrões gerais.
- **Instabilidade**: Pequenas mudanças nos dados podem gerar estruturas diferentes. Isso acontece porque uma divisão ligeiramente distinta perto da raiz afeta toda a árvore abaixo.
- **Viés de seleção**: Alguns critérios podem favorecer atributos com muitos valores possíveis, caso o algoritmo não trate isso adequadamente.
- **Performance nem sempre é a melhor possível**: Em alguns problemas, métodos de conjunto baseados em árvores superam com folga uma única árvore.

2.10 O princípio da árvore pequena

Uma ideia recorrente no estudo de árvores é a preferência por estruturas menores que expliquem bem os dados. Esse raciocínio conversa com o princípio da parcimônia: entre hipóteses com desempenho semelhante, preferimos a mais simples.

Na prática, isso não quer dizer escolher sempre a menor árvore possível. Quer dizer buscar um equilíbrio entre:

- fidelidade aos dados de treino;
- capacidade de generalização;
- facilidade de interpretação.

2.11 Exemplo intuitivo de separação progressiva

Pense em um conjunto com clientes que podem cancelar ou não um serviço. Se os dados estiverem todos misturados, a previsão é incerta. Ao perguntar primeiro `número de chamados no suporte > 3?`, podemos separar um grupo mais propenso ao cancelamento. Em seguida, dentro de cada grupo, outra pergunta como `tempo de contrato > 12 meses?` pode tornar a previsão ainda mais segura.

A árvore funciona justamente assim: ela não tenta resolver o problema inteiro de uma vez. Ela quebra o problema em subproblemas locais.

2.12 Árvore como modelo e como ferramenta de análise

Mesmo quando a árvore não será o modelo final em produção, ela continua valiosa como instrumento de análise exploratória.

- ajuda a descobrir atributos importantes;
- sugere regras de negócio;
- revela interações entre variáveis;
- mostra regiões do espaço de atributos mais arriscadas ou mais favoráveis.

Por isso, aprender árvores de decisão é útil mesmo para quem depois vai trabalhar com ensembles, redes neurais ou modelos probabilísticos.

> ⚠ Erros comuns
>
> - achar que interpretável significa infalível;
> - confundir uma regra aprendida com verdade causal;
> - supor que a maior árvore é sempre a melhor;
> - ignorar a instabilidade estrutural causada por pequenas mudanças nos dados.

Resumo

- Árvores de decisão organizam a previsão como sequências de testes.
- A estrutura raiz-nós-folhas aproxima o modelo de regras humanas.
- O grande atrativo da árvore é combinar interpretabilidade e utilidade prática.
- O grande risco é crescer demais e perder generalização.

Árvores de decisão são um excelente ponto de encontro entre teoria e prática. Elas permitem estudar conceitos formais de aprendizado supervisionado sem perder a intuição do processo decisório. Nos próximos capítulos, vamos detalhar como uma árvore escolhe suas perguntas, como ela mede a qualidade de uma divisão e como transformar essa teoria em modelos úteis com Python.

Perguntas de revisão

1. O que diferencia raiz, nó interno e folha?
2. Por que a árvore é considerada um modelo interpretável?
3. Em que sentido uma árvore pode representar uma disjunção de conjunções?
4. Por que simplicidade e generalização costumam andar juntas?

Capítulo 3

Critérios de Divisão

Uma árvore de decisão não escolhe perguntas ao acaso. Em cada nó, ela procura o teste que melhor organiza os dados locais. Essa escolha depende de um critério de divisão, isto é, de uma medida numérica que avalia o quanto uma partição melhora a separação entre as classes ou reduz o erro.

3.1 Objetivos do capítulo

Ao final desta leitura, o leitor deve ser capaz de:

- explicar o que significa pureza e impureza em um nó;
- calcular entropia e impureza Gini em exemplos simples;
- interpretar ganho de informação como redução de incerteza;
- entender por que um atributo pode ser escolhido na raiz;
- reconhecer os limites práticos de uma escolha gulosa.

3.2 O problema que o critério resolve

Suponha um conjunto com exemplos das classes `A` e `B`. Se o nó atual possui exemplos muito misturados, a incerteza sobre a classe correta é alta. O objetivo do algoritmo é encontrar uma divisão que gere filhos mais homogêneos.

Em outras palavras, a pergunta ideal é aquela que reduz a desordem do conjunto.

3.3 Pureza e impureza

Um nó puro contém exemplos de uma única classe. Um nó impuro contém classes misturadas.

Exemplos:

- `10 A e 0 B`: nó totalmente puro;
- `9 A e 1 B`: nó bastante puro;
- `6 A e 4 B`: nó moderadamente impuro;
- `5 A e 5 B`: nó altamente impuro.

A árvore prefere testes que levem de um nó impuro para filhos mais puros.

3.4 Entropia

A entropia vem da teoria da informação e mede a incerteza associada a uma distribuição de classes.

Para classificação binária:

$$H(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

onde:

- `p_+` é a proporção de exemplos positivos;
- `p_-` é a proporção de exemplos negativos.
- Interpretação intuitiva
- Se todos os exemplos pertencem à mesma classe, a entropia é `0`.
- Se as classes estão equilibradas, a entropia é máxima.
- Quanto maior a entropia, maior a incerteza sobre a classe de um exemplo escolhido ao acaso.

3.5 Tabela intuitiva de entropia

Antes de calcular um caso completo, vale observar o comportamento da medida em distribuições diferentes.

```
import math

def entropia_binaria(p):
    if p in (0, 1):
        return 0.0
```

```
    return -(p * math.log2(p) + (1 - p) * math.log2(1 - p))

for positivos, total in [(10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (5, 10)]:
    p = positivos / total
    print(f"{positivos}/{total} positivos -> entropia = {entropia_binaria(p):.3f}")
```

```
10/10 positivos -> entropia = 0.000
9/10 positivos -> entropia = 0.469
8/10 positivos -> entropia = 0.722
7/10 positivos -> entropia = 0.881
6/10 positivos -> entropia = 0.971
5/10 positivos -> entropia = 1.000
```

– Leitura da tabela

- `10/10`: nenhuma incerteza, entropia zero;
- `9/10`: ainda há alta previsibilidade, entropia baixa;
- `5/10`: máxima incerteza para o caso binário, entropia alta.

Essa progressão ajuda a entender por que um atributo que produz subconjuntos desequilibrados em favor de uma classe é valioso para a árvore.

3.6 Exemplo numérico detalhado de entropia

Suponha um nó com 14 exemplos, sendo 9 positivos e 5 negativos.

$$p_+ = 9/14 \approx 0{,}643$$

$$p_- = 5/14 \approx 0{,}357$$

Substituindo na fórmula:

$$H(S) = -(0{,}643 \cdot \log_2 0{,}643 + 0{,}357 \cdot \log_2 0{,}357)$$

O resultado é aproximadamente `0.940`.

```
p_pos = 9 / 14
p_neg = 5 / 14
```

```
entropia = -(p_pos * math.log2(p_pos) + p_neg * math.log2(p_neg))
print(round(entropia, 3))
```

```
0.94
```

Esse valor indica que o conjunto ainda possui mistura relevante entre classes. Não está no máximo de incerteza, mas tampouco está próximo da pureza total.

3.7 Ganho de informação

O ganho de informação mede a redução esperada na entropia depois da divisão por um atributo.

$$Ganho(S, A) = H(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} H(S_v)$$

A lógica é direta:

- calculamos a entropia antes da divisão;
- calculamos a média ponderada das entropias dos filhos;
- subtraímos os dois valores.

Quanto maior o ganho, melhor foi a divisão.

3.8 Exemplo passo a passo de ganho de informação

Considere novamente o conjunto com 14 exemplos (`9 positivos`, `5 negativos`). Suponha que um atributo particione esse conjunto em dois filhos:

- Filho 1: `6 positivos` e `2 negativos`;
- Filho 2: `3 positivos` e `3 negativos`.
- Entropia do nó pai

Já vimos que:

$$H(S) \approx 0.940$$

Entropia do Filho 1

$$H(S_1) = -(6/8)\log_2(6/8) - (2/8)\log_2(2/8)$$

Entropia do Filho 2

$$H(S_2) = -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1.0$$

Média ponderada após a divisão

$$H_{aps} = (8/14)H(S_1) + (6/14)H(S_2)$$

Ganho

$$Ganho = H(S) - H_{aps}$$

```
def entropia(pos, neg):
    total = pos + neg
    if pos == 0 or neg == 0:
        return 0.0
    p_pos = pos / total
    p_neg = neg / total
    return -(p_pos * math.log2(p_pos) + p_neg * math.log2(p_neg))

pai = entropia(9, 5)
filho_1 = entropia(6, 2)
filho_2 = entropia(3, 3)
entropia_após = (8/14) * filho_1 + (6/14) * filho_2
ganho = pai - entropia_após

print("Entropia pai:", round(pai, 3))
print("Entropia filho 1:", round(filho_1, 3))
print("Entropia filho 2:", round(filho_2, 3))
print("Entropia após divisão:", round(entropia_após, 3))
print("Ganho de informação:", round(ganho, 3))
```

```
Entropia pai: 0.94
Entropia filho 1: 0.811
Entropia filho 2: 1.0
Entropia após divisão: 0.892
Ganho de informação: 0.048
```

- Interpretação do resultado

O ganho será positivo porque a divisão produziu pelo menos um subconjunto mais organizado do que o conjunto original. Se compararmos vários atributos, a árvore tende a escolher aquele com maior ganho.

3.9 Exemplo comparativo entre dois atributos

Imagine dois atributos candidatos no mesmo nó pai.

- Atributo `A`: gera filhos `(6+, 2-)` e `(3+, 3-)`.
- Atributo `B`: gera filhos `(4+, 1-)` e `(5+, 4-)`.

Embora ambos reduzam alguma incerteza, `A` pode produzir ganho maior se separar melhor as classes. Esse tipo de comparação é exatamente o que o algoritmo faz em cada etapa.

```
def ganho_informacao(pai_pos, pai_neg, filhos):
    total = pai_pos + pai_neg
    entropia_pai = entropia(pai_pos, pai_neg)
    entropia_filhos = 0.0
    for pos, neg in filhos:
        subtotal = pos + neg
        entropia_filhos += (subtotal / total) * entropia(pos, neg)
    return entropia_pai - entropia_filhos

print("Ganho atributo A:", round(ganho_informacao(9, 5, [(6, 2), (3, 3)]), 3))
print("Ganho atributo B:", round(ganho_informacao(9, 5, [(4, 1), (5, 4)]), 3))
```

```
Ganho atributo A: 0.048
Ganho atributo B: 0.045
```

3.10 Exemplo conceitual com o problema do tênis

Nos exemplos clássicos de indução, pergunta-se: qual atributo deve ir para a raiz? Se um atributo separa o conjunto em subconjuntos quase puros, ele tende a produzir alto ganho de informação. Por isso o ID3 adota esse critério para construir a árvore de maneira gulosa.

3.11 Impureza Gini

Outro critério muito usado é a impureza Gini.

$$Gini(S) = 1 - \sum_i p_i^2$$

Em classificação binária:

$$Gini(S) = 1 - (p_+^2 + p_-^2)$$

- Interpretação
- `Gini = 0` indica pureza total;
- valores maiores indicam maior mistura entre classes;
- na prática, é um critério extremamente popular por ser eficiente e funcionar muito bem.

3.12 Tabela intuitiva de Gini

```python
def gini_binario(p):
    return 1 - (p**2 + (1 - p)**2)

for positivos, total in [(10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (5, 10)]:
    p = positivos / total
    print(f"{positivos}/{total} positivos -> gini = {gini_binario(p):.3f}")
```

```
10/10 positivos -> gini = 0.000
9/10 positivos -> gini = 0.180
8/10 positivos -> gini = 0.320
7/10 positivos -> gini = 0.420
6/10 positivos -> gini = 0.480
5/10 positivos -> gini = 0.500
```

Observe que a lógica geral é a mesma da entropia: o valor cresce conforme o nó se aproxima de uma divisão equilibrada entre classes.

3.13 Exemplo detalhado de Gini

Considere 10 amostras: 6 da classe A e 4 da classe B.

$$Gini = 1 - (0{,}6^2 + 0{,}4^2) = 1 - (0{,}36 + 0{,}16) = 0{,}48$$

Agora compare com um nó mais puro: 9 da classe A e 1 da classe B.

$$Gini = 1 - (0{,}9^2 + 0{,}1^2) = 1 - (0{,}81 + 0{,}01) = 0{,}18$$

```
p_a = 6 / 10
p_b = 4 / 10
gini = 1 - (p_a**2 + p_b**2)
print(round(gini, 3))

p_a = 9 / 10
p_b = 1 / 10
gini = 1 - (p_a**2 + p_b**2)
print(round(gini, 3))
```

```
0.48
0.18
```

No segundo caso, o valor cai porque o nó está mais puro.

3.14 Entropia ou Gini?

Na prática, as duas medidas costumam produzir resultados parecidos. As diferenças principais são:

- entropia tem interpretação mais ligada à informação;
- Gini costuma ser computacionalmente simples e muito usado por padrão;
- pequenas mudanças na árvore podem ocorrer dependendo do critério escolhido.

No `scikit-learn`, classificação aceita critérios como `gini`, `entropy` e `log_loss`.

3.15 Atributos categóricos e numéricos

Em problemas reais, nem sempre o atributo é categórico simples. Variáveis numéricas exigem encontrar um ponto de corte.

Exemplo:

- `idade <= 35.5`
- `renda <= 4200`
- `tempo_cliente <= 18.5`

O algoritmo testa candidatos a corte e escolhe o que produz a melhor redução de impureza.

3.16 Por que a escolha é gulosa

Algoritmos clássicos de árvore, como ID3 e CART, fazem escolhas locais. Em cada nó, escolhem o melhor atributo naquele momento. Eles não exploram exaustivamente todas as árvores possíveis, porque isso seria computacionalmente inviável.

Essa estratégia gulosa funciona muito bem na prática, mas também explica por que uma pequena alteração nos dados pode mudar a estrutura da árvore.

3.17 Quando um atributo aparentemente bom engana

Um ponto importante da teoria clássica é que alguns atributos podem parecer bons por criarem muitas partições pequenas. Isso pode gerar ganho elevado no treino, mas baixa generalização. Por isso, a escolha do critério deve ser acompanhada por mecanismos de controle de complexidade, como profundidade máxima e poda.

3.18 Critérios em regressão

Quando a saída é numérica, o problema muda. Em vez de reduzir mistura de classes, queremos reduzir dispersão dos valores do alvo.

Critérios comuns incluem:

- redução da variância;
- erro quadrático médio;
- erro absoluto em algumas variantes.

A ideia continua a mesma: separar os dados em grupos mais coerentes internamente.

3.19 Hiperparâmetros ligados a divisão

Os principais controles no `scikit-learn` são:

- `criterion`: define a medida usada no nó;
- `splitter`: geralmente `best` ou `random`;
- `max_depth`: limita profundidade;
- `min_samples_split`: mínimo de exemplos para tentar dividir;
- `min_samples_leaf`: mínimo de exemplos por folha;

- `max_leaf_nodes`: limita número total de folhas;
- `min_impurity_decrease`: exige ganho mínimo para aceitar divisão.

Erros comuns

- calcular entropia sem ponderar corretamente o tamanho dos filhos;
- comparar divisões olhando apenas um subconjunto e não o ganho total;
- esquecer que atributos com muitos valores podem parecer bons no treino;
- tratar Gini e entropia como se um sempre fosse superior ao outro.

Resumo

Toda divisão em uma árvore de decisão tenta responder à mesma pergunta: "qual teste organiza melhor os dados neste ponto?". Entropia, ganho de informação e Gini são formas matemáticas de transformar essa intuição em algoritmo.

Perguntas de revisão

1. O que significa um nó puro?
2. Como a entropia varia entre um nó puro e um nó equilibrado?
3. O que o ganho de informação mede exatamente?
4. Qual a intuição por trás da impureza Gini?
5. Por que a escolha local do melhor atributo é chamada de gulosa?

Capítulo 4

Primeiro Modelo em Python

Neste capítulo, vamos sair da teoria e construir nossa primeira árvore de decisão com `scikit-learn`. O objetivo não é apenas treinar o modelo, mas observar como os conceitos de raiz, nós internos, folhas, impureza e profundidade aparecem concretamente.

4.1 Escolha do dataset

Vamos usar o conjunto Iris, um clássico da literatura de machine learning. Ele contém medidas de flores de três espécies diferentes e é excelente para demonstrar classificação supervisionada.

As variáveis de entrada são:

- comprimento da sépala;
- largura da sépala;
- comprimento da pétala;
- largura da pétala.

A variável alvo é a espécie da flor.

4.2 Preparando os dados

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

iris = load_iris()
```

```
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42,
    stratify=y
)

print("Formato de X:", X.shape)
print("Classes:", iris.target_names)
```

```
Formato de X: (150, 4)
Classes: ['setosa' 'versicolor' 'virginica']
```

4.3 Treinando uma árvore inicial

```
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)

pred = model.predict(X_test)
print("Acurácia:", accuracy_score(y_test, pred))
print(classification_report(y_test, pred, target_names=iris.target_names))
```

```
Acurácia: 0.9333333333333333
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.80      0.89        15
   virginica       0.83      1.00      0.91        15

    accuracy                           0.93        45
   macro avg       0.94      0.93      0.93        45
weighted avg       0.94      0.93      0.93        45
```

Esse primeiro modelo já nos permite observar um ponto importante: árvores costumam se ajustar muito bem a datasets pequenos e estruturados, mas isso não significa automaticamente que vão

generalizar bem em qualquer contexto.

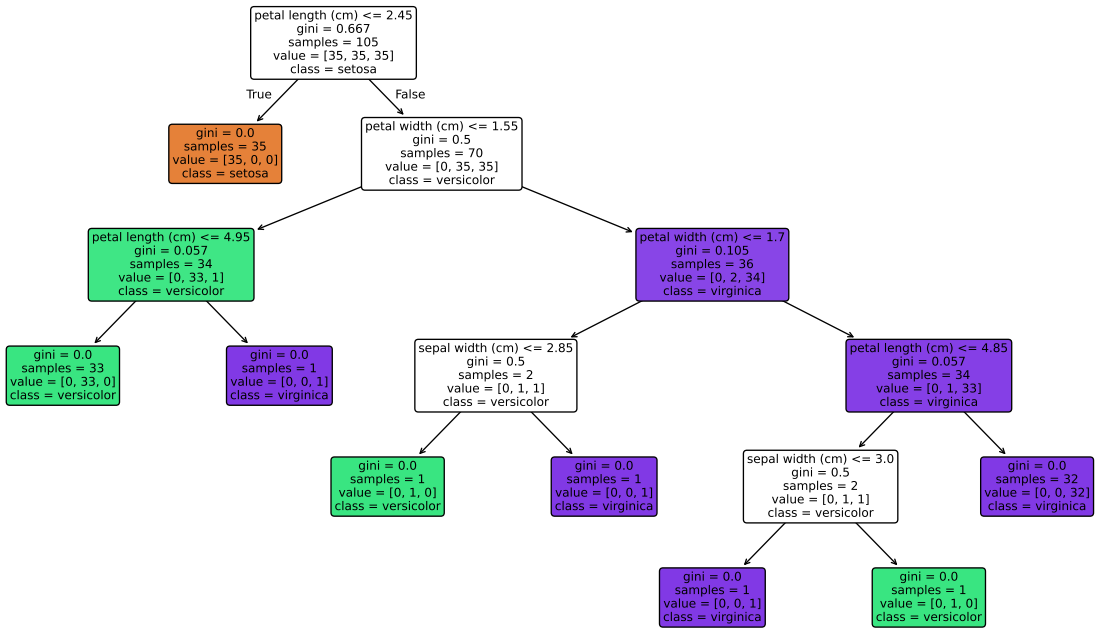
4.4 O que a árvore aprendeu

Depois do treino, a árvore passa a ter uma estrutura interna com informações como:

- qual atributo foi usado na raiz;
- quais pontos de corte foram escolhidos;
- quantas amostras chegaram a cada nó;
- qual foi a impureza observada em cada etapa.

4.5 Visualizando a árvore

```
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

plt.figure(figsize=(16, 9))
plot_tree(
    model,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    filled=True,
    rounded=True,
    fontsize=9
)
plt.show()
```



4.5.1 Como ler o gráfico

Em cada nó, você costuma encontrar:

- a regra de divisão, como `petal length (cm) <= 2.45`;
- a impureza do nó;
- o número de amostras naquele ponto;
- a distribuição das classes;
- a classe majoritária prevista.

4.6 Interpretando a raiz

A raiz normalmente concentra a pergunta mais informativa de todo o problema. No Iris, é comum que medidas da pétala apareçam cedo na árvore, porque elas separam muito bem certas espécies.

Quando um atributo aparece perto da raiz, isso sugere que ele tem grande utilidade discriminativa para aquele conjunto de dados.

4.7 Medindo complexidade da árvore

Podemos inspecionar algumas propriedades do modelo.

```
print("Profundidade da árvore:", model.get_depth())
print("Número de folhas:", model.get_n_leaves())
```

```
Profundidade da árvore: 5
Número de folhas: 8
```

Esses dois números ajudam a entender se a árvore está simples ou excessivamente detalhada.

4.8 Comparando uma árvore mais controlada

Uma boa prática é comparar o modelo livre com uma versão limitada.

```
shallow_model = DecisionTreeClassifier(max_depth=3, random_state=42)
shallow_model.fit(X_train, y_train)

shallow_pred = shallow_model.predict(X_test)
print("Acurácia árvore rasa:", accuracy_score(y_test, shallow_pred))
print("Profundidade:", shallow_model.get_depth())
print("Folhas:", shallow_model.get_n_leaves())
```

```
Acurácia árvore rasa: 0.9777777777777777
Profundidade: 3
Folhas: 5
```

Esse tipo de comparação mostra algo essencial: aumentar complexidade não garante melhora em dados novos.

4.9 Importância das variáveis

Outra análise útil é observar a importância dos atributos.

```
import pandas as pd

importance = pd.Series(model.feature_importances_, index=iris.feature_names)
print(importance.sort_values(ascending=False))
```

```
petal length (cm)    0.541176
petal width (cm)     0.430252
sepal width (cm)     0.028571
sepal length (cm)    0.000000
```

```
dtype: float64
```

4.9.1 Cuidado com a interpretação

Importância de atributo é útil, mas não deve ser lida como verdade absoluta. Ela depende da estrutura aprendida pela árvore e da forma como as variáveis competem entre si para entrar nas divisões.

4.10 Fazendo previsões em novos exemplos

```
novo_exemplo = [[5.1, 3.5, 1.4, 0.2]]
classe_prevista = model.predict(novo_exemplo)[0]
probabilidades = model.predict_proba(novo_exemplo)[0]

print("Classe prevista:", iris.target_names[classe_prevista])
print("Probabilidades:", probabilidades)
```

```
Classe prevista: setosa
Probabilidades: [1. 0. 0.]
```

Isso reforça a diferença entre:

- `predict`: devolve a classe final;
- `predict_proba`: devolve a distribuição estimada entre classes na folha.

4.11 Exportando regras textuais

Quando quisermos uma representação mais textual da árvore, podemos usar `export_text`.

```
from sklearn.tree import export_text

rules = export_text(model, feature_names=list(iris.feature_names))
print(rules)
```

```
|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- petal width (cm) <= 1.55
|   |   |--- petal length (cm) <= 4.95
|   |   |   |--- class: 1
```

```
|   |   |--- petal length (cm) >  4.95
|   |   |   |--- class: 2
|   |--- petal width (cm) >  1.55
|   |   |--- petal width (cm) <= 1.70
|   |   |   |--- sepal width (cm) <= 2.85
|   |   |   |   |--- class: 1
|   |   |   |--- sepal width (cm) >  2.85
|   |   |   |   |--- class: 2
|   |   |--- petal width (cm) >  1.70
|   |   |   |--- petal length (cm) <= 4.85
|   |   |   |   |--- sepal width (cm) <= 3.00
|   |   |   |   |   |--- class: 2
|   |   |   |   |--- sepal width (cm) >  3.00
|   |   |   |   |   |--- class: 1
|   |   |   |--- petal length (cm) >  4.85
|   |   |   |   |--- class: 2
```

Esse formato é excelente para documentação, ensino e discussão com pessoas que preferem texto a gráfico.

4.12 Boas práticas desde o primeiro modelo

Mesmo em exemplos simples, vale manter algumas disciplinas:

- separar treino e teste;
- fixar `random_state` para reprodutibilidade;
- comparar versões mais simples e mais complexas;
- olhar além da acurácia;
- interpretar a árvore aprendida.

> ⚠ Erros comuns
>
> - olhar apenas a acurácia e ignorar estrutura da árvore;
> - treinar sem separar treino e teste;
> - interpretar importância de atributos como causalidade;
> - assumir que a primeira árvore treinada já está pronta para uso final.

Resumo

- O treino em Python materializa conceitos como raiz, profundidade e folhas.
- A visualização da árvore ajuda a interpretar regras e cortes aprendidos.
- Comparar uma árvore livre com uma árvore rasa é um bom hábito didático.
- Importância de atributos e exportação textual ampliam a interpretabilidade.

Nosso primeiro modelo mostrou que árvores de decisão podem ser treinadas rapidamente e lidas com relativa facilidade. No entanto, ainda falta responder a perguntas importantes: como avaliar melhor a qualidade do modelo, como ajustar hiperparâmetros e como evitar que a árvore fique mais complexa do que o necessário. Esse será o foco do próximo capítulo.

Perguntas de revisão

1. O que você observa em cada nó quando usa `plot_tree`?
2. Por que vale a pena comparar árvores com profundidades diferentes?
3. O que `predict_proba` adiciona a interpretação?
4. Em que situações `export_text` pode ser mais útil do que o gráfico?

Capítulo 5

Avaliação e Ajuste

Treinar um modelo é apenas parte do trabalho. Um modelo aparentemente bom pode estar apenas decorando o conjunto de treino. Avaliação adequada significa medir desempenho em dados não vistos e interpretar os resultados com critério.

5.1 Separação entre treino e teste

A primeira barreira contra o autoengano em machine learning é separar os dados em conjuntos distintos.

- treino: usado para ajustar o modelo;
- validação: usado para comparar configurações;
- teste: usado para medir desempenho final de forma mais honesta.

No capítulo anterior usamos apenas treino e teste. Agora vamos ampliar essa avaliação.

5.2 Métricas para classificação

Em classificação, a acurácia é útil, mas não basta em todos os contextos.

Para entender essas métricas, usamos quatro quantidades da matriz de confusão:

- `TP` (true positive): positivo previsto como positivo;
- `TN` (true negative): negativo previsto como negativo;
- `FP` (false positive): negativo previsto como positivo;
- `FN` (false negative): positivo previsto como negativo.

5.2.1 Accuracy

Indica a proporção total de previsões corretas:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

É uma métrica geral e simples de comunicar, mas pode enganar em bases desbalanceadas. Se 95% dos exemplos forem da classe negativa, um modelo que quase sempre prevê "negativo" pode ter alta acurácia e ainda assim ser pouco útil.

5.2.2 Precision

Entre os exemplos previstos como positivos, quantos eram de fato positivos:

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

Precision alta significa poucos falsos positivos. Ela é importante quando o custo de um alarme falso é alto, como em bloqueio indevido de transações legítimas.

5.2.3 Recall

Entre os exemplos realmente positivos, quantos foram encontrados:

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

Recall alta significa poucos falsos negativos. Ela é essencial quando "deixar passar" um caso positivo é crítico, como triagem médica ou detecção de fraude.

5.2.4 F1-score

É a média harmônica entre precision e recall:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

O F1-score cresce quando há equilíbrio entre precision e recall e cai quando uma das duas é baixa. Por isso, é útil quando queremos uma visão única de desempenho em cenários com classes desbalanceadas ou com trade-off entre falsos positivos e falsos negativos.

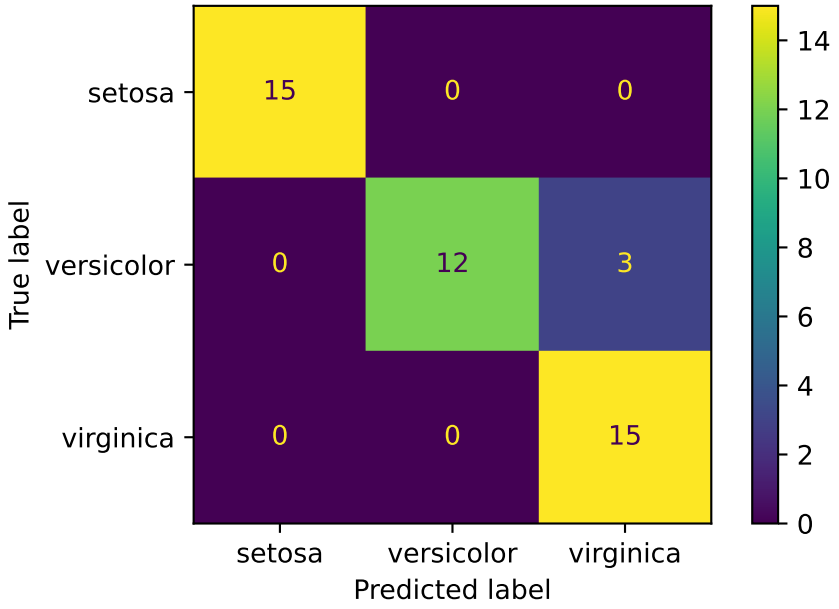
5.2.5 Matriz de confusão

Mostra com mais detalhe onde o modelo está acertando e errando.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42,
    stratify=y
)

print("Formato de X:", X.shape)
print("Classes:", iris.target_names)
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)

pred = model.predict(X_test)


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, pred)
print(cm)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=iris.target_names)
disp.plot()
```

```
Formato de X: (150, 4)
Classes: ['setosa' 'versicolor' 'virginica']
[[15  0  0]
```

```
[ 0 12  3]
[ 0  0 15]]
```



5.3 Relatório de classificação

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42,
    stratify=y
)

print("Formato de X:", X.shape)
print("Classes:", iris.target_names)
```

```
from sklearn.metrics import classification_report

print(classification_report(y_test, pred, target_names=iris.target_names))
```

```
Formato de X: (150, 4)
Classes: ['setosa' 'versicolor' 'virginica']
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.80      0.89        15
   virginica       0.83      1.00      0.91        15

    accuracy                           0.93        45
   macro avg       0.94      0.93      0.93        45
weighted avg       0.94      0.93      0.93        45
```

Esse relatório é útil porque resume várias métricas por classe.

5.4 Por que validação cruzada importa

Uma única divisão treino-teste pode ser injusta com o modelo ou benevolente demais. A validação cruzada reduz essa dependência de uma partição específica.

Na validação cruzada `k-fold`, dividimos os dados em `k` partes. O modelo treina em `k-1` partes e valida na parte restante. Repetimos isso `k` vezes, mudando o bloco de validação.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(
    DecisionTreeClassifier(random_state=42),
    X,
    y,
    cv=5,
    scoring="accuracy"
)

print("Scores:", scores)
print("Média:", scores.mean())
```

```
print("Desvio padrão:", scores.std())
```

```
Scores: [0.96666667 0.96666667 0.9        0.93333333 1.        ]
Média: 0.9533333333333334
Desvio padrão: 0.03399346342395189
```

5.4.1 Como interpretar

- média alta sugere bom desempenho médio;
- desvio padrão alto sugere instabilidade;
- média baixa pode indicar que o modelo não está adequado ou que faltam ajustes.

5.5 Ajustando hiperparâmetros

Uma árvore possui vários controles de complexidade. O papel do ajuste é encontrar configurações que generalizem melhor.

5.5.1 Parâmetros mais importantes

- `max_depth`: limita profundidade total;
- `min_samples_split`: exige número mínimo para dividir um nó;
- `min_samples_leaf`: evita folhas pequenas demais;
- `criterion`: muda a função de qualidade da divisão;
- `ccp_alpha`: controla poda por custo-complexidade.

5.6 Grid Search

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    "max_depth": [2, 3, 4, 5, None],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf": [1, 2, 4],
    "criterion": ["gini", "entropy"]
}

grid = GridSearchCV(
```

```
    DecisionTreeClassifier(random_state=42),
    param_grid=param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1
)

grid.fit(X_train, y_train)
print("Melhores parâmetros:", grid.best_params_)
print("Melhor score CV:", grid.best_score_)
```

```
Melhores parâmetros: {'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 1, 'mi
Melhor score CV: 0.9523809523809523
```

5.7 Avaliando o melhor modelo no teste

```
best_model = grid.best_estimator_
best_pred = best_model.predict(X_test)

print("Accuracy no teste:", accuracy_score(y_test, best_pred))
print(classification_report(y_test, best_pred, target_names=iris.target_names))
```

```
Accuracy no teste: 0.9777777777777777
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.93      0.97        15
   virginica       0.94      1.00      0.97        15

    accuracy                           0.98        45
   macro avg       0.98      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

Esse passo é importante porque a escolha do melhor conjunto de hiperparâmetros foi feita olhando para validação, não para o teste final.

5.8 Vieses de avaliação comuns

5.8.1 Avaliar no treino

Se medirmos o modelo no mesmo conjunto que o treinou, o resultado tende a ser otimista demais.

5.8.2 Ajustar repetidamente olhando o teste

O conjunto de teste deve ser preservado como referência final. Usar o teste para tomar decisões sucessivas vaza informação.

5.8.3 Escolher só pela acurácia

Em dados desbalanceados, acurácia pode esconder um modelo ruim para a classe de maior interesse.

5.9 Diagnóstico prático de uma árvore

Ao avaliar uma árvore, pergunte:

- ela está muito profunda?
- o desempenho no treino está muito acima do teste?
- existem folhas com poucas amostras?
- a estrutura aprendida faz sentido para o domínio?
- a variação entre folds está alta?

5.10 Curva de complexidade

Uma estratégia didática interessante é testar profundidades crescentes e observar o desempenho.

```
from sklearn.metrics import f1_score

for depth in [1, 2, 3, 4, 5, None]:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_train, y_train)
    pred_depth = clf.predict(X_test)
    print(
        f"depth={depth} | acc={accuracy_score(y_test, pred_depth):.3f} | "
```

```
        f"f1_macro={f1_score(y_test, pred_depth, average='macro'):.3f}"
    )
```

```
depth=1 | acc=0.667 | f1_macro=0.556
depth=2 | acc=0.889 | f1_macro=0.889
depth=3 | acc=0.978 | f1_macro=0.978
depth=4 | acc=0.889 | f1_macro=0.889
depth=5 | acc=0.933 | f1_macro=0.933
depth=None | acc=0.933 | f1_macro=0.933
```

Esse experimento ajuda a visualizar a relação entre simplicidade e desempenho.

Resumo

- Avaliar bem é tão importante quanto treinar bem.
- Validação cruzada reduz a dependência de uma única divisão treino-teste.
- Grid search ajuda a explorar hiperparâmetros de forma sistemática.
- O teste final deve ser preservado para avaliação honesta.

Avaliar bem uma árvore significa medir desempenho com honestidade, comparar configurações de maneira sistemática e interpretar os erros com atenção. No próximo capítulo, vamos aprofundar um dos riscos mais importantes desse modelo: o overfitting, e veremos como a poda ajuda a manter a árvore útil e interpretável.

Erros comuns

- usar o conjunto de teste para ajustar decisões repetidas vezes;
- escolher modelo só pela acurácia;
- ignorar variação entre folds;
- confundir score alto no treino com capacidade de generalização.

Perguntas de revisão

1. Qual é a diferença entre treino, validação e teste?
2. Quando a acurácia pode ser enganosa?
3. O que a validação cruzada ajuda a reduzir?
4. Por que o melhor modelo do grid deve ser avaliado depois no teste?

Capítulo 6

Poda e Overfitting

Árvores de decisão aprendem rápido, mas justamente por isso podem crescer demais. Quando a árvore continua criando divisões para capturar detalhes muito específicos do treino, ela deixa de representar padrões gerais e passa a memorizar ruído. Esse fenômeno é o overfitting.

6.1 O que é overfitting

Overfitting acontece quando o modelo se ajusta excessivamente aos dados de treinamento e perde capacidade de generalização. Em uma árvore, isso aparece de maneira muito visível: a estrutura fica profunda, ramificada e cheia de folhas com poucas amostras.

6.2 Sinais típicos

- desempenho excelente no treino;
- desempenho bem pior no teste;
- árvore muito alta ou com muitas folhas;
- regras muito específicas e pouco intuitivas;
- alta sensibilidade a pequenas mudanças nos dados.

6.3 Por que árvores sofrem com isso

Como a construção é gulosa e local, a árvore aceita novas divisões sempre que elas parecem melhorar a pureza dos nós. Sem restrições, esse processo continua até restarem grupos muito pequenos ou totalmente puros no treino.

Em muitos casos, pureza extrema no treino significa apenas memorização.

6.4 Pré-poda

Pré-poda significa impedir que a árvore cresca demais desde o início.

6.4.1 Hiperparâmetros mais usados

- `max_depth`: limita quantos níveis a árvore pode ter;
- `min_samples_split`: exige um mínimo para tentar nova divisão;
- `min_samples_leaf`: garante um mínimo por folha;
- `max_leaf_nodes`: controla o número total de folhas;
- `min_impurity_decrease`: exige ganho mínimo de qualidade.

6.4.2 Exemplo

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42,
    stratify=y
)
```

```
from sklearn.tree import DecisionTreeClassifier

pre_pruned = DecisionTreeClassifier(
    max_depth=4,
    min_samples_split=10,
    min_samples_leaf=5,
    random_state=42
)
```

```
pre_pruned.fit(X_train, y_train)
```

	criterion criterion: {”gini”, ”entropy”, ”log_loss”}, default=”gini” The function to measure the quality of a split. Supported criteria are ”gini” for the Gini impurity and ”log_loss” and ”entropy” both for the Shannon information gain, see :ref:‘tree_mathematical_formulation‘.	'gini'
	splitter splitter: {”best”, ”random”}, default=”best” The strategy used to choose the split at each node. Supported strategies are ”best” to choose the best split and ”random” to choose the best random split.	'best'
	max_depth max_depth: int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.	4

	min_samples_split min_samples_split: int or float, default=2 The minimum number of samples required to split an internal node: - If int, then consider ‘min_samples_split‘ as the minimum number. - If float, then ‘min_samples_split‘ is a fraction and ‘ceil(min_samples_split * n_samples)‘ are the minimum number of samples for each split. .. versionchanged:: 0.18 Added float values for fractions.	10

min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least “min_samples_leaf“ training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. - If int, then consider ‘min_samples_leaf‘ as the minimum number. - If float, then ‘min_samples_leaf‘ is a fraction and ‘ceil(min_samples_leaf * n_samples)‘ are the minimum number of samples for each node. .. versionchanged:: 0.18 Added float values for fractions.	5

	min_weight_fraction_leaf min_weight_fraction_leaf: float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.	0.0

	max_features max_features: int, float or {”sqrt”, ”log2”}, default=None The number of features to consider when looking for the best split: - If int, then consider ‘max_features‘ features at each split. - If float, then ‘max_features‘ is a fraction and ‘max(1, int(max_features * n_features_in_))‘ features are considered at each split. - If ”sqrt”, then ‘max_features=sqrt(n_features)‘. - If ”log2”, then ‘max_features=log2(n_features)‘. - If None, then ‘max_features=n_features‘. .. note:: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than “max_features“ features.	None

	random_state random_state: int, RandomState instance or None, default=None Controls the randomness of the estimator. The features are always randomly permuted at each split, even if “splitter“ is set to ““best”“. When “max_features < n_features“, the algorithm will select “max_features“ at random at each split before finding the best split among them. But the best found split may vary across different runs, even if “max_features=n_features“. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, “random_state“ has to be fixed to an integer. See :term:‘Glossary ‘ for details.	42

	max_leaf_nodes max_leaf_nodes: int, default=None Grow a tree with “max_leaf_nodes“ in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.	None

min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R / N_t * right_impurity - N_t_L / N_t * left_impurity) where “N“ is the total number of samples, “N_t“ is the number of samples at the current node, “N_t_L“ is the number of samples in the left child, and “N_t_R“ is the number of samples in the right child. “N“, “N_t“, “N_t_R“ and “N_t_L“ all refer to the weighted sum, if “sample_weight“ is passed. .. versionadded:: 0.19	0.0

class_weight	class_weight: dict, list of dict or ”balanced”, default=None Weights associated with classes in the form “{class_label: weight}“. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}]. The ”balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as “n_samples / (n_classes * np.bincount(y))“ For multi-output, the weights of each column of y will be multiplied.	None

	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than “ccp_alpha“ will be chosen. By default, no pruning is performed. See :ref:‘minimal_cost_complexity_pruning‘ for details. See :ref:‘sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py‘ for an example of such pruning. .. versionadded:: 0.22	0.0

	monotonic_cst monotonic_cst: array-like of int of shape (n_features), default=None Indicates the monotonicity constraint to enforce on each feature. - 1: monotonic increase - 0: no constraint - -1: monotonic decrease If monotonic_cst is None, no constraints are applied. Monotonicity constraints are not supported for: - multiclass classifications (i.e. when ‘n_classes > 2‘), - multioutput classifications (i.e. when ‘n_outputs_ > 1‘), - classifications trained on data with missing values. The constraints hold over the probability of the positive class. Read more in the :ref:‘User Guide ‘. .. versionadded:: 1.4	None

6.5 Pos-poda

Pós-poda significa deixar a árvore crescer mais e depois remover ramos que não compensam sua complexidade. No `scikit-learn`, isso aparece na poda por custo-complexidade usando

`ccp_alpha`.

A ideia é equilibrar duas forças:

- erro de ajuste;
- tamanho da árvore.

Quanto maior o `ccp_alpha`, maior a penalização da complexidade e menor tende a ficar a árvore.

6.6 Caminho de poda

```
path = DecisionTreeClassifier(random_state=42).cost_complexity_pruning_path(X_train, 
ccp_alphas = path.ccp_alphas

models = []
for alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
    clf.fit(X_train, y_train)
    models.append(clf)

print("Total de modelos testados:", len(models))
print("Primeiros alphas:", ccp_alphas[:10])
```

```
Total de modelos testados: 5
Primeiros alphas: [0.         0.00899471 0.01848739 0.2788671  0.33333333]
```

6.7 Comparando alphas na prática

```
results = []
for alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
    clf.fit(X_train, y_train)
    train_score = clf.score(X_train, y_train)
    test_score = clf.score(X_test, y_test)
    results.append((alpha, clf.get_depth(), clf.get_n_leaves(), train_score, test_sco

for row in results[:10]:
    print(row)
```

```
(np.float64(0.0), 5, np.int64(8), 1.0, 0.9333333333333333)
(np.float64(0.008994708994708996), 3, np.int64(4), 0.9809523809523809, 0.933333333333
(np.float64(0.01848739495798319), 2, np.int64(3), 0.9714285714285714, 0.888888888888
(np.float64(0.27886710239651413), 1, np.int64(2), 0.6666666666666666, 0.666666666666
(np.float64(0.33333333333333354), 0, np.int64(1), 0.3333333333333333, 0.333333333333
```

A comparação acima mostra algo valioso: conforme o alpha cresce, a árvore simplifica. Em algum ponto, simplificar ajuda a generalizar; depois disso, simplificar demais passa a reduzir desempenho.

6.8 Como escolher `ccp_alpha`

O jeito mais seguro é usar validação cruzada. A escolha não deve depender apenas do score no treino ou de uma única divisão teste.

6.9 Complexidade versus interpretabilidade

Poda não serve apenas para melhorar desempenho. Ela também deixa o modelo mais fácil de explicar. Em ambientes corporativos, uma árvore um pouco menos precisa, mas muito mais clara, pode ser preferível a uma estrutura enorme e difícil de justificar.

6.10 Exemplo de leitura de negócio

Uma árvore de churn sem poda pode produzir dezenas de regras microespecíficas. Depois da poda, talvez sobrem algumas regras fortes, como:

- muitos chamados ao suporte aumentam risco de cancelamento;
- pouco tempo de contrato combinado com alto gasto aumenta vulnerabilidade;
- clientes antigos com baixo atrito tendem a permanecer.

Essas regras são muito mais acionáveis.

6.11 Underfitting também existe

Nem toda simplificação é boa. Se limitarmos demais a profundidade ou exigirmos folhas muito grandes, a árvore pode se tornar incapaz de capturar padrões importantes. Isso produz underfitting: desempenho ruim até mesmo no treino.

6.12 Sinais de underfitting

- baixa acurácia no treino e no teste;
- árvore rasa demais;
- regras excessivamente genéricas;
- incapacidade de separar grupos claramente distintos.

6.13 Estratégia prática recomendada

Uma abordagem equilibrada costuma ser:

1. treinar uma árvore inicial para entender o problema;
2. medir profundidade, número de folhas e desempenho em treino e teste;
3. ajustar `max_depth`, `min_samples_leaf` e `min_samples_split`;
4. testar poda com `ccp_alpha`;
5. escolher o ponto em que simplicidade e desempenho ficam bem equilibrados.

Resumo

- Overfitting aparece quando a árvore memoriza detalhes do treino.
- Pré-poda impede crescimento excessivo desde o início.
- Pós-poda simplifica a árvore depois do treino.
- O melhor ponto costuma equilibrar simplicidade, desempenho e interpretabilidade.

Uma boa árvore não é necessariamente a maior nem a mais pura no treino. É a que consegue representar os padrões relevantes sem se prender a detalhes acidentais. No próximo capítulo, vamos ver como essa lógica aparece tanto em classificação quanto em regressão.

Erros comuns

- concluir que mais profundidade sempre melhora o modelo;
- podar demais e gerar underfitting;
- escolher `ccp_alpha` olhando apenas treino;
- ignorar folhas com poucas amostras.

Perguntas de revisão

1. Como reconhecer sinais de overfitting em uma árvore?
2. O que diferencia pre-poda e pos-poda?
3. Qual o papel de `ccp_alpha`?

4. Por que uma árvore mais simples pode ser preferível em negócio?

Capítulo 7

Classificação e Regressão

Árvores de decisão não se limitam a problemas de classificação. A mesma estrutura geral também pode ser usada para prever valores contínuos. O que muda e o critério usado para construir a árvore e a natureza da previsão feita nas folhas.

7.1 Classificação

Na classificação, o objetivo e atribuir uma catégoria.

Exemplos:

- aprovar ou reprovar crédito;
- detectar spam ou não spam;
- prever cancelamento ou permanencia;
- classificar espécie de planta.

O modelo mais comum em Python e `DecisionTreeClassifier`.

7.1.1 Como a previsão e feita

Cada folha armazena a distribuição das classes que chegaram até ela. A previsão final geralmente corresponde a classe majoritária da folha.

7.1.2 Medidas tipicas

- accuracy;
- precision;
- recall;

- f1-score;
- ROC AUC em alguns contextos;
- matriz de confusão.

7.2 Regressão

Na regressão, a saída e numerica.

Exemplos:

- prever preço de imovel;
- estimar consumo de energia;
- projetar tempo de aténdimento;
- calcular demanda futura.

O modelo correspondente e `DecisionTreeRegressor`.

7.2.1 Como a previsão e feita

Em cada folha, o modelo produz um valor numérico, normalmente relacionado a média dos alvos observados naquele grupo.

7.2.2 Medidas tipicas

- MAE;
- MSE;
- RMSE;
- R2.

7.3 O que muda no critério de divisão

Na classificação, a árvore tenta reduzir mistura de classes. Na regressão, ela tenta reduzir dispersão dos valores do alvo. Assim, embora a estrutura pareca a mesma, a função objetivo e diferente.

7.4 Exemplo de regressão com dados sintéticos

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

X_reg, y_reg = make_regression(
    n_samples=1000,
    n_features=6,
    n_informative=4,
    noise=15,
    random_state=42
)

X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg,
    y_reg,
    test_size=0.2,
    random_state=42
)

reg = DecisionTreeRegressor(max_depth=6, random_state=42)
reg.fit(X_train_reg, y_train_reg)
pred_reg = reg.predict(X_test_reg)

print("MAE:", mean_absolute_error(y_test_reg, pred_reg))
print("MSE:", mean_squared_error(y_test_reg, pred_reg))
print("R2:", r2_score(y_test_reg, pred_reg))
```

```
MAE: 24.261617947978333
MSE: 986.8354955471909
R2: 0.6604996247352501
```

7.5 Comparando profundidades na regressão

```
for depth in [2, 4, 6, 10, None]:
    reg = DecisionTreeRegressor(max_depth=depth, random_state=42)
    reg.fit(X_train_reg, y_train_reg)
    pred_reg = reg.predict(X_test_reg)
    print(
```

```
        f"depth={depth} | "
        f"MAE={mean_absolute_error(y_test_reg, pred_reg):.2f} | "
        f"R2={r2_score(y_test_reg, pred_reg):.3f}"
    )
```

```
depth=2 | MAE=31.95 | R2=0.476
depth=4 | MAE=26.49 | R2=0.613
depth=6 | MAE=24.26 | R2=0.660
depth=10 | MAE=22.31 | R2=0.710
depth=None | MAE=22.62 | R2=0.709
```

Esse experimento costuma mostrar que profundidades muito grandes podem criar curvas de previsão fragmentadas demais.

7.6 Visualmente, o que a árvore faz na regressão

Enquanto modelos lineares produzem superficies suaves, árvores de regressão particionam o espaco de atributos em regioes. Em cada regiao, a previsão tende a ser constante ou aproximadamente constante. Isso torna o modelo flexível, mas também sujeito a cortes abruptos.

7.7 Quando usar classificação e quando usar regressão

A resposta depende da variável alvo.

- Se o alvo representa catégorias, use classificação.
- Se o alvo representa quantidade numerica, use regressão.

Parece obvio, mas ha casos de fronteira. Por exemplo, prever faixa de risco pode ser classificação; prever probabilidade de inadimplencia pode ser modelado de outras formas; prever valor de perda e regressão.

7.8 Vantagens compartilhadas

Tanto em classificação quanto em regressão, árvores oferecem:

- interpretabilidade;
- capacidade de modelar não linearidades;
- pouca exigencia de pre-processamento inicial;
- possibilidade de identificar regras locais de comportamento.

7.9 Cuidados compartilhados

As duas variantes exigem aténcao para:

- overfitting;
- instabilidade estrutural;
- necessidade de validação cruzada;
- leitura critica das regras aprendidas.

7.10 Relação com ensembles

Random Forest e Gradient Boosting usam várias árvores combinadas. Entender bem uma única árvore fácilita muito o aprendizado desses modelos mais fortes, porque os principios basicos de divisão, profundidade, folhas e controle de complexidade contínuam relevantes.

Resumo

- A mesma estrutura de árvore pode servir a classificação e regressão.
- O que muda principalmente e o critério usado nas divisoes e o tipo de saída das folhas.
- Em classificação, as folhas representam classes; em regressão, valores numéricos.
- Controle de complexidade contínua sendo essencial nos dois casos.

Árvores de decisão formam uma família versatil. O mesmo mecanismo de particionar o espaco de atributos pode servir tanto para escolher uma classe quanto para estimar um valor numérico. No próximo capítulo, vamos reunir esses conceitos em um estudo de caso mais completo, cobrindo preparação, treino, avaliação e interpretação.

Erros comuns

- usar metricas de classificação em problema de regressão ou vice-versa;
- supor que a interpretabilidade elimina necessidade de validação;
- esquecer que árvores de regressão produzem previsões em degraus;
- comparar modelos sem alinhar a natureza do alvo.

Perguntas de revisão

1. Qual a principal diferenca entre `DecisionTreeClassifier` e `DecisionTreeRegressor`?
2. Que tipo de metrica faz sentido em regressão?
3. Por que a profundidade excessiva também prejudica regressão?

4. O que aproxima uma árvore isolada dos ensembles baseados em árvores?

Capítulo 8

Estudo de Caso Completo

Neste capítulo, vamos organizar um fluxo de trabalho mais próximo do que acontece em um projeto real. O problema escolhido será previsão de churn, isto é, estimar se um cliente tende a cancelar um serviço.

O foco aqui não é apenas treinar um modelo, mas estruturar o pensamento:

- compreender o problema de negócio;
- gerar ou organizar os dados;
- separar treino e teste;
- ajustar uma árvore interpretável;
- avaliar o resultado;
- extrair insight acionável.

8.1 Contexto do problema

Em churn, geralmente queremos identificar clientes com maior risco de cancelamento para agir antes da perda acontecer. Uma árvore de decisão é particularmente útil nesse contexto porque transforma o problema em regras compreensíveis para equipes de negócio.

8.2 Passo 1: gerando uma base sintética plausível

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.metrics import accuracy_score, f1_score, classification_report

np.random.seed(42)
n = 1000

df = pd.DataFrame({
    "idade": np.random.randint(18, 70, n),
    "tempo_cliente": np.random.randint(1, 120, n),
    "gasto_mensal": np.random.normal(120, 40, n).clip(20, 400),
    "suporte_chamados": np.random.poisson(2, n),
    "atrasos_pagamento": np.random.poisson(1, n),
    "usa_app": np.random.binomial(1, 0.7, n),
})

logit = (
    -2.2
    + 0.012 * df["idade"]
    - 0.022 * df["tempo_cliente"]
    + 0.010 * df["gasto_mensal"]
    + 0.35 * df["suporte_chamados"]
    + 0.45 * df["atrasos_pagamento"]
    - 0.60 * df["usa_app"]
)

prob = 1 / (1 + np.exp(-logit))
df["churn"] = (np.random.rand(n) < prob).astype(int)

print(df.head())
print(df["churn"].mean())
```

```
   idade  tempo_cliente  gasto_mensal  suporte_chamados  atrasos_pagamento  \
0     56             99    121.149793                 5                  0
1     69            115    171.138075                 1                  0
2     46             15    127.643963                 1                  1
3     32             64    121.857462                 4                  0
4     60             89     65.605754                 1                  0
```

```
   usa_app  churn
0        1      0
1        1      1
2        1      0
3        1      0
4        0      0
0.307
```

8.3 Passo 2: separando atributos e alvo

```
X = df.drop(columns=["churn"])
y = df["churn"]

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=42,
    stratify=y
)

print(X_train.shape, X_test.shape)
```

```
(750, 6) (250, 6)
```

8.4 Passo 3: treinando uma árvore interpretável

```
clf = DecisionTreeClassifier(
    max_depth=5,
    min_samples_leaf=15,
    random_state=42
)
clf.fit(X_train, y_train)

pred = clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test, pred))
print("F1:", f1_score(y_test, pred))
```

```
print(classification_report(y_test, pred))
```

```
Accuracy: 0.74
F1: 0.4036697247706422
              precision    recall  f1-score   support

           0       0.75      0.94      0.83       173
           1       0.69      0.29      0.40        77

    accuracy                           0.74       250
   macro avg       0.72      0.61      0.62       250
weighted avg       0.73      0.74      0.70       250
```

Esses hiperparâmetros foram escolhidos para equilibrar clareza e desempenho. Em problemas de negócio, uma árvore um pouco mais compacta pode ser mais útil do que uma estrutura enorme com pequenas vantagens marginais.

8.5 Passo 4: lendo a importância das variáveis

```
importances = pd.Series(clf.feature_importances_, index=X.columns)
print(importances.sort_values(ascending=False))
```

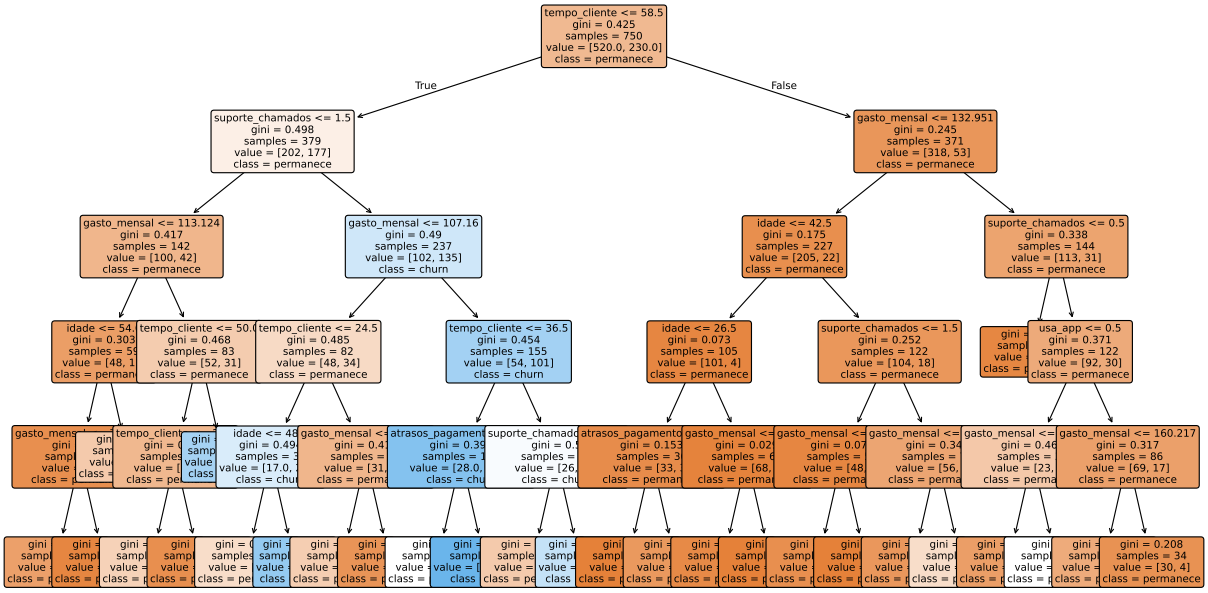
```
tempo_cliente        0.528235
suporte_chamados     0.186245
gasto_mensal         0.180747
idade                0.049257
atrasos_pagamento    0.041453
usa_app              0.014064
dtype: float64
```

8.5.1 Interpretação

Se `suporte_chamados` e `atrasos_pagamento` aparecem no topo, isso sugere que experiência ruim e atrito operacional estão associados ao cancelamento. Se `tempo_cliente` tiver importância alta, o modelo pode estar distinguindo clientes ainda pouco fidelizados.

8.6 Passo 5: visualizando a árvore

```
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

plt.figure(figsize=(18, 10))
plot_tree(
    clf,
    feature_names=X.columns,
    class_names=["permanece", "churn"],
    filled=True,
    rounded=True,
    fontsize=9
)
plt.show()
```



Ao olhar a árvore, tente responder:

- qual variável foi para a raiz?
- quais pontos de corte apareceram?
- existem folhas muito pequenas?
- as regras parecem coerentes com o domínio?

8.7 Passo 6: extraindo regras textuais

```
from sklearn.tree import export_text

print(export_text(clf, feature_names=list(X.columns)))
```

```
|--- tempo_cliente <= 58.50
|   |--- suporte_chamados <= 1.50
|   |   |--- gasto_mensal <= 113.12
|   |   |   |--- idade <= 54.00
|   |   |   |   |--- gasto_mensal <= 74.36
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  74.36
|   |   |   |   |   |--- class: 0
|   |   |   |--- idade >  54.00
|   |   |   |   |--- class: 0
|   |   |--- gasto_mensal >  113.12
|   |   |   |--- tempo_cliente <= 50.00
|   |   |   |   |--- tempo_cliente <= 35.50
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- tempo_cliente >  35.50
|   |   |   |   |   |--- class: 0
|   |   |   |--- tempo_cliente >  50.00
|   |   |   |   |--- class: 1
|   |--- suporte_chamados >  1.50
|   |   |--- gasto_mensal <= 107.16
|   |   |   |--- tempo_cliente <= 24.50
|   |   |   |   |--- idade <= 48.50
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- idade >  48.50
|   |   |   |   |   |--- class: 1
|   |   |   |--- tempo_cliente >  24.50
|   |   |   |   |--- gasto_mensal <= 94.44
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  94.44
|   |   |   |   |   |--- class: 0
|   |   |--- gasto_mensal >  107.16
```

```
|   |   |   |--- tempo_cliente <= 36.50
|   |   |   |   |--- atrasos_pagamento <= 0.50
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- atrasos_pagamento >  0.50
|   |   |   |   |   |--- class: 1
|   |   |   |--- tempo_cliente >  36.50
|   |   |   |   |--- suporte_chamados <= 2.50
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- suporte_chamados >  2.50
|   |   |   |   |   |--- class: 1
|--- tempo_cliente >  58.50
|   |--- gasto_mensal <= 132.95
|   |   |--- idade <= 42.50
|   |   |   |--- idade <= 26.50
|   |   |   |   |--- atrasos_pagamento <= 0.50
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- atrasos_pagamento >  0.50
|   |   |   |   |   |--- class: 0
|   |   |   |--- idade >  26.50
|   |   |   |   |--- gasto_mensal <= 79.51
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  79.51
|   |   |   |   |   |--- class: 0
|   |   |--- idade >  42.50
|   |   |   |--- suporte_chamados <= 1.50
|   |   |   |   |--- gasto_mensal <= 96.07
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  96.07
|   |   |   |   |   |--- class: 0
|   |   |   |--- suporte_chamados >  1.50
|   |   |   |   |--- gasto_mensal <= 118.71
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  118.71
|   |   |   |   |   |--- class: 0
|   |--- gasto_mensal >  132.95
|   |   |--- suporte_chamados <= 0.50
|   |   |   |--- class: 0
```

```
|   |   |--- suporte_chamados >  0.50
|   |   |   |--- usa_app <= 0.50
|   |   |   |   |--- gasto_mensal <= 155.19
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  155.19
|   |   |   |   |   |--- class: 0
|   |   |   |--- usa_app >  0.50
|   |   |   |   |--- gasto_mensal <= 160.22
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- gasto_mensal >  160.22
|   |   |   |   |   |--- class: 0
```

Esse formato ajuda a comunicar o resultado para pessoas não técnicas.

8.8 Passo 7: comparando com uma árvore mais profunda

```
complex_clf = DecisionTreeClassifier(random_state=42)
complex_clf.fit(X_train, y_train)

pred_complex = complex_clf.predict(X_test)
print("Accuracy árvore livre:", accuracy_score(y_test, pred_complex))
print("F1 árvore livre:", f1_score(y_test, pred_complex))
print("Depth árvore livre:", complex_clf.get_depth())
print("Folhas árvore livre:", complex_clf.get_n_leaves())
```

```
Accuracy árvore livre: 0.688
F1 árvore livre: 0.4657534246575342
Depth árvore livre: 16
Folhas árvore livre: 176
```

A comparação evidencia um ponto importante do livro inteiro: nem sempre a árvore mais complexa é a mais interessante.

8.9 Passo 8: validação de negócio

Em projetos reais, score não basta. É preciso verificar se as regras são acionáveis.

Exemplos de perguntas úteis:

- clientes com muitos chamados estão recebendo suporte insuficiente?
- atrasos em pagamento refletem dificuldade financeira ou falha operacional?
- usuários que não usam o app estão menos engajados com o serviço?
- clientes novos precisam de onboarding melhor?

8.10 Passo 9: possíveis ações práticas

Se a árvore identificar grupos de alto risco, algumas ações podem ser desenhadas:

- campanha de retenção para clientes com alto gasto e muitos chamados;
- oferta de suporte proativo para clientes novos;
- automação de cobrança e regularização para perfis com atrasos;
- incentivo ao uso do aplicativo para aumentar engajamento.

8.11 Passo 10: próximos refinamentos

Em um projeto real, poderíamos ainda:

- usar validação cruzada para ajustar hiperparâmetros;
- avaliar desbalanceamento de classes;
- testar poda com `ccp_alpha`;
- comparar com Random Forest e Gradient Boosting;
- calibrar probabilidades, se necessário.

Resumo

- Um estudo de caso mostra a árvore como ferramenta de previsão e de descoberta de regras.
- A interpretabilidade ajuda a ligar score a ação de negócio.
- Comparar uma árvore compacta com uma árvore livre é uma forma concreta de estudar complexidade.
- O valor prático da árvore aumenta quando as regras se tornam acionáveis.

Este estudo de caso mostrou que uma árvore de decisão pode servir ao mesmo tempo como modelo preditivo e ferramenta de descoberta de regras. Em contextos empresariais, essa combinação é poderosa porque permite transformar previsões em ações concretas. No último capítulo, vamos consolidar o aprendizado com exercícios práticos.

Erros comuns

- focar apenas em score e ignorar a utilidade operacional das regras;
- interpretar importância de atributo sem contexto de negócio;
- construir uma árvore excessivamente detalhada para um público não técnico;
- deixar de validar se a regra faz sentido fora da amostra sintética.

Perguntas de revisão

1. Por que churn é um problema interessante para árvores de decisão?
2. O que uma variável importante sugere, e o que ela não prova?
3. Por que uma árvore mais simples pode ser mais útil para negócio?
4. Que tipo de ação prática pode surgir da leitura das regras?

Capítulo 9

Exercícios Práticos

Este capítulo foi pensado para transformar leitura em domínio real do assunto. Tente resolver os exercícios sem consultar imédiatamente o gabarito. O ganho maior vem do processo de pensar nas escolhas do modelo.

9.1 Exercício 1: intuição estrutural

Explique com suas palavras:

- o que representa a raiz de uma árvore;
- o que significa um nó ser puro;
- por que uma folha pode ser entendida como uma decisão final;
- por que uma árvore muito grande pode generalizar pior.

9.2 Exercício 2: Gini e entropia

Considere um nó com 12 exemplos, dos quais 9 pertencem a classe `A` e 3 a classe `B`.

1. Calcule a impureza Gini.
2. Calcule a entropia.
3. Explique o que esses valores dizem sobre a pureza do nó.
4. Compare com um nó contendo 6 exemplos de cada classe.

9.3 Exercício 3: primeira árvore no Iris

Treine uma `DecisionTreeClassifier` no Iris com `max_depth=3`.

- Compare com o modelo sem limite de profundidade.
- Avalie `accuracy` e `f1_macro`.
- Compare profundidade e número de folhas.
- Interprete a diferenca entre simplicidade e desempenho.

9.4 Exercício 4: leitura da árvore

Depois de treinar o modelo no Iris:

- plote a árvore;
- identifique o atributo da raiz;
- escreva duas regras da raiz até folhas;
- diga por que esses atributos parecem informativos.

9.5 Exercício 5: validação cruzada

Use `cross_val_score` com `cv=5` para comparar os seguintes modelos:

- `DecisionTreeClassifier(max_depth=2)`
- `DecisionTreeClassifier(max_depth=3)`
- `DecisionTreeClassifier(max_depth=5)`
- `DecisionTreeClassifier()`

Responda:

- qual profundidade teve melhor média?
- qual configuração pareceu mais estavel?
- ha evidencias de overfitting nas configurações mais profundas?

9.6 Exercício 6: ajuste com Grid Search

Ajuste uma árvore para o Iris usando os hiperparâmetros:

- `max_depth`: `[2, 3, 4, 5, None]`
- `min_samples_split`: `[2, 5, 10]`
- `min_samples_leaf`: `[1, 2, 4]`
- `criterion`: `["gini", "entropy"]`

Depois:

- mostre os melhores parâmetros;

- avalie no conjunto de teste;
- explique por que a melhor combinação pode ter funcionado.

9.7 Exercício 7: poda por custo-complexidade

No mesmo problema:

- gere o caminho de `ccp_alpha`;
- treine várias árvores com alphas diferentes;
- compare profundidade, número de folhas e desempenho;
- tente encontrar o ponto em que a simplificação ainda preserva a qualidade.

9.8 Exercício 8: regressão sintética

Crie um problema com `make_regression` e compare:

- `DecisionTreeRegressor(max_depth=3)`
- `DecisionTreeRegressor(max_depth=6)`
- `DecisionTreeRegressor(max_depth=10)`
- `DecisionTreeRegressor()`

Avalie com:

- `MAE`
- `MSE`
- `R2`

Explique o efeito do aumento da profundidade.

9.9 Exercício 9: estudo de caso de churn

No dataset sintético do capítulo anterior:

- teste `min_samples_leaf` em `[1, 5, 10, 20]`;
- compare `max_depth` em `[3, 4, 5, 6, None]`;
- verifique quais variáveis mais aparecem na parte superior da árvore;
- escreva três interpretações de negócio a partir das regras aprendidas.

9.10 Exercício 10: critica de modelo

Escreva um pequeno parecer técnico respondendo:

- quando uma árvore isolada e uma boa escolha;
- quando ela pode ser insuficiente;
- em que situações voce priorizaria interpretabilidade;
- quando faria sentido migrar para Random Forest ou Gradient Boosting.

9.11 Gabarito inicial para apoio

O objetivo do gabarito abaixo não e entregar todas as respostas, mas fornecer um ponto de partida para implementação.

```
from sklearn.datasets import load_iris, make_regression
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.metrics import accuracy_score, f1_score, mean_absolute_error, r2_score

# Classificação com Iris
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42,
    stratify=y
)

for depth in [None, 3]:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_train, y_train)
    pred = clf.predict(X_test)
    print(
        f"depth={depth} | acc={accuracy_score(y_test, pred):.3f} | "
        f"f1_macro={f1_score(y_test, pred, average='macro'):.3f} | "
        f"depth_real={clf.get_depth()} | leaves={clf.get_n_leaves()}"
    )
```

```
# Validação cruzada
for depth in [2, 3, 5, None]:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    scores = cross_val_score(clf, X, y, cv=5, scoring="accuracy")
    print(f"depth={depth} | média={scores.mean():.3f} | desvio={scores.std():.3f}")

# Grid search
param_grid = {
    "max_depth": [2, 3, 4, 5, None],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf": [1, 2, 4],
    "criterion": ["gini", "entropy"]
}

grid = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    param_grid=param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1
)
grid.fit(X_train, y_train)
print(grid.best_params_)

# Regressão
X_reg, y_reg = make_regression(n_samples=500, n_features=5, noise=20, random_state=42
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg,
    y_reg,
    test_size=0.2,
    random_state=42
)

for depth in [3, 6, 10, None]:
    reg = DecisionTreeRegressor(max_depth=depth, random_state=42)
    reg.fit(X_train_reg, y_train_reg)
```

```
    pred_reg = reg.predict(X_test_reg)
    print(
        f"depth={depth} | MAE={mean_absolute_error(y_test_reg, pred_reg):.2f} | "
        f"R2={r2_score(y_test_reg, pred_reg):.3f}"
    )
```

```
depth=None | acc=0.933 | f1_macro=0.933 | depth_real=5 | leaves=8
depth=3 | acc=0.978 | f1_macro=0.978 | depth_real=3 | leaves=5
depth=2 | média=0.933 | desvio=0.047
depth=3 | média=0.973 | desvio=0.025
depth=5 | média=0.953 | desvio=0.034
depth=None | média=0.953 | desvio=0.034
{'criterion': 'gini', 'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2}
depth=3 | MAE=47.65 | R2=0.689
depth=6 | MAE=53.99 | R2=0.645
depth=10 | MAE=54.69 | R2=0.645
depth=None | MAE=53.80 | R2=0.667
```

Use esse bloco como base e amplie as respostas conceituais por escrito. Em aprendizado de maquina, saber justificar a escolha do modelo e tao importante quanto executar o código.

! Dicas

- resolva primeiro sem consultar o gabarito;
- escreva as respostas conceituais em texto, não apenas em código;
- compare suas solucoes com os principios vistos nos capítulos teóricos;
- revise especialmente os exercícios em que a estrutura da árvore ficou diferente do esperado.

⚠ Erros comuns

- responder apenas com código, sem interpretação;
- comparar modelos sem relatar metricas e profundidade;
- esquecer de justificar por que um hiperparametro funcionou melhor;
- tratar o gabarito como resposta final em vez de ponto de partida.

Perguntas de revisão

1. Voce consegue calcular entropia e Gini manualmente em exemplos simples?
2. Voce consegue explicar diferencas entre ID3, C4.5 e CART?
3. Voce sabe treinar, visualizar e podar uma árvore em Python?
4. Voce consegue transformar uma árvore em regras compreensíveis para negócio?

Capítulo 10

ID3, C4.5 e CART

Depois de entender os critérios de divisão, o passo seguinte é conectar esses critérios aos algoritmos clássicos de indução de árvores. Nesta etapa, três nomes aparecem com frequência na literatura e no ensino de aprendizado de máquina: ID3, C4.5 e CART.

Embora hoje muitas pessoas usem diretamente bibliotecas como `scikit-learn`, compreender esses algoritmos ajuda bastante a entender por que as árvores se comportam da forma que se comportam.

10.1 Objetivos do capítulo

Ao final deste capítulo, o leitor deve ser capaz de:

- descrever a lógica geral de indução top-down;
- explicar o papel do ganho de informação no ID3;
- reconhecer as melhorias introduzidas pelo C4.5;
- compreender a proposta do CART e sua relevância prática;
- diferenciar esses algoritmos em nível conceitual.

10.2 A ideia geral de indução top-down

A maior parte dos algoritmos clássicos constrói a árvore de cima para baixo.

1. Começa-se com todos os exemplos no nó raiz.
2. Escolhe-se o melhor atributo para dividir esse conjunto.
3. Criam-se ramos para os resultados possíveis do teste.
4. O processo é repetido recursivamente em cada subconjunto.

5. A construção para quando os exemplos ficam suficientemente homogêneos, quando faltam atributos ou quando algum critério de parada é atingido.

Essa estratégia é chamada de gulosa porque a escolha do atributo é feita localmente, sem revisar todas as alternativas possíveis para a árvore inteira.

10.3 O algoritmo ID3

O ID3, associado a Quinlan, é um dos algoritmos mais clássicos de aprendizado de árvores de decisão. Seu critério principal é o ganho de informação.

10.3.1 Ideia central

Em cada nó, o ID3 escolhe o atributo que mais reduz a entropia do conjunto local.

10.3.2 Visão conceitual do procedimento

- se todos os exemplos do nó pertencem à mesma classe, cria-se uma folha;
- se não há mais atributos disponíveis, usa-se a classe majoritária;
- caso contrário, escolhe-se o melhor atributo segundo ganho de informação;
- o processo continua recursivamente em cada ramo.

10.3.3 Pseudocódigo conceitual

```
ID3(exemplos, atributos):
  se todos os exemplos são da mesma classe:
    retorna folha
  se não restam atributos:
    retorna folha com classe majoritária
  escolhe o atributo com maior ganho de informação
  cria um nó para esse atributo
  para cada valor do atributo:
    cria um ramo
    chama ID3 para o subconjunto correspondente
```

10.4 Forças do ID3

- alta clareza conceitual;

- forte valor didático;
- boa conexão entre teoria da informação e classificação;
- capacidade de construir regras simbólicas interpretáveis.

10.5 Limitações do ID3

Apesar de elegante, o ID3 possui limitações clássicas.

- foi pensado principalmente para classificação;
- lida de forma mais natural com atributos categóricos;
- pode favorecer atributos com muitos valores distintos;
- não é, por si só, um mecanismo completo de poda robusta, como nas variantes posteriores.

10.6 O problema do viés por muitos valores

Imagine dois atributos:

- `código_cliente`, com muitos valores quase únicos;
- `inadimplencia_previa`, com poucos valores e forte significado.

Um atributo com muitos valores pode gerar partições pequenas demais e aparentar grande ganho no treino, mesmo sem generalizar bem. Esse tipo de problema motivou refinamentos posteriores.

10.7 C4.5

O C4.5 pode ser visto como uma extensão importante do ID3. Ele preserva a ideia de construção top-down, mas introduz mecanismos mais sofisticados para problemas reais.

10.7.1 Melhorias conceituais associadas ao C4.5

- tratamento mais flexível de atributos contínuos;
- melhor tratamento de valores ausentes;
- uso de critérios que reduzem o viés por atributos com muitos valores;
- poda posterior para melhorar generalização.

10.7.2 Gain ratio

Uma ideia frequentemente associada ao C4.5 é o uso do `gain ratio`, que ajusta o ganho de informação pela forma como o atributo divide os dados. Isso reduz a tendência de favorecer atributos com muitos valores.

Em termos intuitivos, o algoritmo pergunta não apenas “quanto esse atributo reduz a incerteza?”, mas também “essa redução veio de uma divisão razoável ou de uma fragmentação excessiva?”.

10.8 Tratamento de atributos contínuos

Em domínios reais, muitos atributos são numéricos: idade, renda, temperatura, saldo, tempo, consumo, score. O C4.5 considera pontos de corte e transforma um atributo contínuo em testes do tipo:

- `idade <= 35.5`
- `saldo <= 1200`
- `tempo_contrato <= 18`

Essa ideia foi crucial para a aplicabilidade prática das árvores.

10.9 Poda no C4.5

A poda entra para reduzir complexidade e melhorar generalização. O algoritmo aceita que a árvore inicial possa crescer mais e, em seguida, remove ramos cuja contribuição não justifica a complexidade adicional.

10.10 CART

CART significa `Classification and Regression Trees`. Esse nome já sinaliza um diferencial importante: o algoritmo foi formulado tanto para classificação quanto para regressão.

10.10.1 Características conceituais do CART

- usa divisões binárias;
- pode ser aplicado à classificação e à regressão;
- popularizou a lógica de poda por custo-complexidade;
- está fortemente ligado a implementações modernas.

10.11 Divisões binárias

Enquanto alguns materiais didáticos apresentam árvores com vários ramos por atributo categórico, o CART trabalha de forma muito natural com divisões binárias. Isso significa que cada nó produz dois filhos.

Exemplos:

- `idade <= 35.5` versus `idade > 35.5`;
- `regiao in {sul, sudeste}` versus `regiao fora desse grupo`.

Essa estrutura binária simplifica vários aspectos computacionais e aparece claramente em muitas bibliotecas atuais.

10.12 CART em classificação e regressão

No caso de classificação, o CART costuma usar medidas como Gini para avaliar a qualidade da divisão. Na regressão, usa critérios associados à redução da variância ou do erro.

Por isso, o CART se tornou particularmente influente na prática moderna e em famílias inteiras de modelos baseados em árvores.

10.13 Relação com o `scikit-learn`

Na prática, quando treinamos uma `DecisionTreeClassifier` ou `DecisionTreeRegressor` no `scikit-learn`, estamos muito mais próximos da família conceitual do CART do que do ID3 puro.

Isso significa que, mesmo estudando ID3 e C4.5, o leitor deve entender que a implementação corrente da biblioteca segue uma linha mais próxima de:

- divisões binárias;
- critérios como Gini e entropia para classificação;
- poda por custo-complexidade com `ccp_alpha`.

10.14 Comparação conceitual resumida

10.14.1 ID3

- fortemente didático;
- usa ganho de informação;
- associado ao estudo clássico de indução;
- mais limitado diante de atributos contínuos e questões práticas modernas.

10.14.2 C4.5

- amplia o ID3;

- lida melhor com atributos contínuos e valores ausentes;
- introduz refinamentos como gain ratio;
- incorpora poda mais estruturada.

10.14.3 CART

- trabalha com divisões binárias;
- cobre classificação e regressão;
- tem enorme relevância prática;
- inspira bastante as implementações de uso corrente.

10.15 Por que estudar algoritmos clássicos se a biblioteca já faz tudo?

Porque a biblioteca treina a árvore, mas não substitui o entendimento.

Quem compreende ID3, C4.5 e CART consegue:

- interpretar melhor o comportamento do modelo;
- explicar por que determinado critério foi escolhido;
- identificar limitações de generalização;
- tomar decisões mais conscientes sobre hiperparâmetros e poda.

Resumo

ID3, C4.5 e CART representam momentos centrais na evolução das árvores de decisão. O ID3 formaliza a indução top-down guiada por ganho de informação. O C4.5 amplia essa proposta para lidar melhor com problemas reais. O CART consolida uma abordagem extremamente prática, binária e aplicável tanto à classificação quanto à regressão.
No próximo capítulo, vamos sair da formulação algorítmica e voltar ao uso concreto em Python, observando como essas ideias aparecem em uma árvore treinada de fato.

Erros comuns

- estudar apenas a API moderna e ignorar a intuição dos algoritmos clássicos;
- imaginar que ID3, C4.5 e CART são equivalentes em todos os detalhes;
- esquecer o problema do viés por atributos com muitos valores;
- supor que bibliotecas atuais implementam ID3 puro.

Perguntas de revisão

1. Qual é a ideia central do ID3?
2. O que o C4.5 melhora em relação ao ID3?
3. Por que o CART é tão importante na prática moderna?
4. Como as divisões binárias influenciam a estrutura da árvore?

Capítulo 11

Interpretação de Regras e Importância de Atributos

Uma das maiores vantagens das árvores de decisão é que elas não servem apenas para prever. Elas também ajudam a explicar, comunicar e organizar conhecimento sobre o problema. Neste capítulo, o foco deixa de ser apenas desempenho e passa a incluir interpretabilidade.

11.1 Objetivos do capítulo

Ao final da leitura, o leitor deve ser capaz de:

- transformar caminhos da árvore em regras compreensíveis;
- interpretar folhas, probabilidades e distribuições de classe;
- usar importância de atributos com critério;
- comunicar resultados para público técnico e não técnico;
- distinguir explicação legítima de interpretação apressada.

11.2 Cada caminho é uma regra

Uma árvore pode ser lida como um conjunto de regras condicionais. Cada caminho da raiz até uma folha corresponde a uma regra do tipo:

```
Se condição 1 e condição 2 e condição 3, então classe prevista = X
```

Exemplo:

```
Se suporte_chamados > 3
 e tempo_cliente <= 12
 e atrasos_pagamento > 1
então churn = alto
```

Essa forma textual é extremamente útil para documentação, auditoria e comunicação com áreas de negócio.

11.3 Como interpretar uma folha

Uma folha não deve ser vista apenas como um rótulo final. Em geral, ela também carrega informações sobre:

- quantidade de amostras que chegaram ali;
- distribuição das classes nesse subconjunto;
- confiança relativa da previsão;
- pureza ou mistura residual daquele grupo.

Uma folha com 300 exemplos e classe majoritária muito clara conta uma história diferente de uma folha com 4 exemplos e distribuição equilibrada entre classes.

11.4 Exemplo de exportação textual

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text

X, y = load_iris(return_X_y=True)
feature_names = load_iris().feature_names

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

clf = DecisionTreeClassifier(max_depth=3, random_state=42)
clf.fit(X_train, y_train)

print(export_text(clf, feature_names=list(feature_names)))
```

```
|--- petal length (cm) <= 2.45
|   |--- class: 0
|--- petal length (cm) >  2.45
|   |--- petal width (cm) <= 1.55
|   |   |--- petal length (cm) <= 4.95
|   |   |   |--- class: 1
|   |   |--- petal length (cm) >  4.95
|   |   |   |--- class: 2
|   |--- petal width (cm) >  1.55
|   |   |--- petal width (cm) <= 1.70
|   |   |   |--- class: 1
|   |   |--- petal width (cm) >  1.70
|   |   |   |--- class: 2
```

Ao ler a saída, procure identificar:

- qual atributo aparece na raiz;
- quais cortes aparecem primeiro;
- quais regras são curtas e quais são mais profundas;
- onde a árvore parece mais confiante ou mais hesitante.

11.5 Importância de atributos

Bibliotecas como `scikit-learn` disponibilizam uma medida de importância de atributo baseada na redução de impureza proporcionada por cada variável ao longo da árvore.

```python
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
clf = DecisionTreeClassifier(random_state=42)
clf.fit(iris.data, iris.target)

importancias = pd.Series(clf.feature_importances_, index=iris.feature_names)
print(importancias.sort_values(ascending=False))
```

```
petal length (cm)    0.564056
petal width (cm)     0.422611
```

```
sepal length (cm)    0.013333
sepal width (cm)     0.000000
dtype: float64
```

11.6 Como ler a importância com responsabilidade

Se uma variável tem importância alta, isso indica que ela contribuiu bastante para as divisões da árvore. No entanto, isso não significa automaticamente causalidade, relevância universal ou independência em relação a outras variáveis.

11.6.1 Cuidados importantes

- importância não é causalidade;
- importância depende do conjunto de dados disponível;
- atributos correlacionados podem disputar espaço entre si;
- pequenas mudanças nos dados podem alterar a árvore e a importância;
- uma variável pouco usada na árvore não é necessariamente irrelevante em outros modelos.

11.7 Regras globais e regras locais

Uma forma útil de pensar é separar dois níveis de interpretação.

11.7.1 Interpretação global

Busca entender a lógica geral da árvore:

- quais atributos aparecem perto da raiz;
- quais grupos principais foram separados;
- como a complexidade foi distribuída.

11.7.2 Interpretação local

Busca entender uma previsão específica:

- por que este exemplo caiu nesta folha?
- quais condições foram satisfeitas?
- qual conjunto de regras levou a essa decisão?

11.8 Exemplo de interpretação local

Suponha um cliente com:

- muitos chamados no suporte;
- pouco tempo de contrato;
- histórico recente de atraso.

Se a árvore conduz esse cliente até uma folha de alto risco de churn, podemos reconstruir o caminho e apresentar essa previsão como uma regra explícita. Isso torna a ação muito mais clara do que simplesmente dizer "o score foi alto".

11.9 Quando a interpretabilidade é mais valiosa que alguns pontos de score

Em muitos contextos, uma árvore ligeiramente menos precisa pode ser preferida a um modelo mais opaco se houver necessidade de:

- auditoria;
- prestação de contas;
- validação por especialistas do domínio;
- comunicação para gestores;
- definição de políticas ou regras internas.

11.10 Comunicando resultados para público não técnico

Ao apresentar uma árvore para público não técnico, evite falar primeiro em entropia, Gini ou hiperparâmetros. Comece por perguntas como:

- quais perfis estão mais associados ao resultado?
- quais sinais aparecem mais cedo na decisão?
- que grupos a árvore separou?
- quais regras podem ser transformadas em ação?

Depois, se necessário, detalhe o mecanismo técnico.

11.11 Boas práticas de documentação

Ao documentar uma árvore para uso prático, inclua:

- objetivo do modelo;
- variável alvo;
- principais atributos usados;
- regras mais relevantes;
- métricas principais;
- limites conhecidos do modelo;
- cuidados de interpretação.

11.12 O perigo da falsa simplicidade

O fato de a árvore ser visualmente intuitiva não significa que toda interpretação será correta. Há riscos de:

- confundir correlação com causa;
- superestimar a robustez de um ramo com poucas amostras;
- ignorar instabilidade estrutural;
- tratar uma regra local como se fosse uma lei geral do domínio.

Interpretar bem exige unir leitura da estrutura com conhecimento do problema.

11.13 Árvore como ferramenta de descoberta

Mesmo quando o modelo final em produção não é uma única árvore, ela continua sendo muito útil para:

- descobrir variáveis promissoras;
- revelar interações entre atributos;
- propor hipóteses de negócio;
- orientar engenharias de variável;
- explicar rapidamente um domínio para a equipe.

Resumo

Interpretar uma árvore de decisão significa transformar estrutura em significado. Cada caminho pode ser lido como regra, cada folha como grupo de comportamento e cada atributo importante como um sinal potencialmente relevante. Quando bem utilizada, a árvore é, ao mesmo tempo, um modelo preditivo e uma linguagem para explicar o problema.
Com esse repertório, o leitor está pronto para consolidar o estudo com os exercícios finais do livro.

Erros comuns

- interpretar importância de atributos como causalidade;
- apresentar regras sem informar suporte ou contexto;
- confundir explicação local com regra global do domínio;
- ignorar que folhas com poucas amostras exigem cautela.

Perguntas de revisão

1. Como transformar um caminho da árvore em regra textual?
2. O que uma folha informa além da classe final?
3. Por que importância de atributo não deve ser lida como causalidade?
4. Como comunicar regras da árvore para público não técnico?

Capítulo 12

Árvore de Decisão versus Random Forest e Ensembles

Uma árvore de decisão isolada é excelente para aprender, explicar e produzir regras claras. No entanto, em muitos problemas reais, modelos de conjunto baseados em árvores conseguem desempenho preditivo superior. Este capítulo compara a árvore isolada com Random Forest e outras estratégias de ensemble.

12.1 Objetivos do capítulo

Ao final da leitura, o leitor deve ser capaz de:

- explicar por que uma árvore isolada é instável;
- entender a intuição do Random Forest;
- reconhecer quando priorizar interpretabilidade ou desempenho;
- enxergar a árvore isolada como base conceitual para modelos mais fortes.

12.2 Por que uma única árvore pode ser insuficiente

Uma árvore individual tem várias qualidades, mas também sofre com problemas conhecidos:

- alta variância;
- sensibilidade a pequenas mudanças nos dados;
- tendência a overfitting quando cresce demais;
- desempenho por vezes inferior ao de modelos combinados.

Isso acontece porque toda a estrutura depende de escolhas locais em cada divisão. Se a raiz muda,

todo o restante pode mudar junto.

12.3 A intuição do ensemble

Em vez de depender de uma única árvore, um ensemble combina várias árvores. A ideia geral é que o conjunto seja mais robusto do que qualquer árvore individual.

Duas famílias muito importantes são:

- bagging, do qual o Random Forest é o exemplo mais famoso;
- boosting, usado em técnicas como Gradient Boosting, XGBoost, LightGBM e CatBoost.

12.4 Random Forest

O Random Forest constrói várias árvores sobre amostras reamostradas dos dados e combina suas previsões.

12.4.1 Ideia central

- cada árvore vê uma variação do conjunto de treino;
- em cada divisão, apenas um subconjunto aleatório de atributos é considerado;
- as previsões finais são agregadas por votação, no caso de classificação, ou por média, no caso de regressão.

Esse processo reduz a variância e torna o modelo mais estável.

12.5 O que o Random Forest ganha

- maior robustez;
- melhor generalização em muitos problemas;
- menor sensibilidade a pequenas perturbações nos dados;
- boa capacidade de modelar interações complexas.

12.6 O que o Random Forest perde

- menor transparência em comparação com uma única árvore;
- dificuldade maior para comunicar uma regra simples única;
- custo computacional mais alto;
- interpretação menos direta para público não técnico.

12.7 Comparação prática de intuição

12.7.1 Árvore isolada

Pense nela como um especialista que toma decisão seguindo uma única cadeia clara de regras.

12.7.2 Random Forest

Pense nele como um comitê de especialistas. Cada um olha para os dados de forma um pouco diferente. A decisão final vem da agregação dessas opiniões.

12.8 Quando a árvore isolada é a melhor escolha

Uma única árvore costuma ser muito adequada quando:

- interpretabilidade é requisito central;
- o modelo precisa virar regra de negócio ou política interna;
- o objetivo inclui ensino ou explicação do problema;
- deseja-se um baseline rápido e claro;
- o domínio exige auditoria e justificativa direta da previsão.

12.9 Quando o Random Forest tende a ser melhor

O Random Forest costuma ser uma boa escolha quando:

- o foco principal é desempenho preditivo;
- a instabilidade da árvore isolada atrapalha;
- há volume razoável de dados;
- a relação entre variáveis e alvo é mais complexa;
- aceita-se trocar parte da interpretabilidade por robustez.

12.10 E o boosting?

No boosting, as árvores são construídas sequencialmente. Cada nova árvore tenta corrigir erros das anteriores. Em muitos contextos, isso produz desempenho ainda maior do que bagging.

Por outro lado, o boosting tende a exigir mais cuidado com ajuste e costuma ser ainda menos transparente do que uma única árvore.

12.11 Exemplo simples em Python

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

tree_model = DecisionTreeClassifier(max_depth=3, random_state=42)
forest_model = RandomForestClassifier(n_estimators=200, random_state=42)

tree_model.fit(X_train, y_train)
forest_model.fit(X_train, y_train)

tree_pred = tree_model.predict(X_test)
forest_pred = forest_model.predict(X_test)

print("Árvore | acc:", accuracy_score(y_test, tree_pred), "| f1_macro:", f1_score(y_t
print("Forest | acc:", accuracy_score(y_test, forest_pred), "| f1_macro:", f1_score(y
```

```
Árvore | acc: 0.9777777777777777 | f1_macro: 0.9777530589543938
Forest | acc: 0.9111111111111111 | f1_macro: 0.9107142857142857
```

Esse exemplo não prova que o Random Forest sempre vencerá, mas ajuda a mostrar como a comparação pode ser feita na prática.

12.12 A árvore isolada continua importante

Mesmo quando o modelo final escolhido é um ensemble, a árvore individual continua sendo essencial para o aprendizado conceitual. Entender bem uma árvore ajuda a compreender:

- pureza e impureza;
- profundidade e folhas;

- divisões binárias;
- critérios de separação;
- importância de atributos;
- efeitos de poda e controle de complexidade.

Resumo

- A árvore isolada privilegia clareza e interpretabilidade.
- O Random Forest privilegia robustez e desempenho médio.
- Ensembles reduzem a fragilidade estrutural de uma única árvore.
- A escolha entre um e outro depende do equilíbrio entre explicabilidade e performance.

Erros comuns

- tratar o Random Forest como substituto automático em qualquer problema;
- abandonar a árvore isolada sem aprender seus fundamentos;
- comparar modelos sem considerar a interpretabilidade como critério;
- escolher ensemble mais complexo sem necessidade prática clara.

Perguntas de revisão

1. Por que uma árvore isolada tende a ter alta variância?
2. Como o Random Forest reduz instabilidade?
3. Em que contexto uma única árvore pode ser preferível mesmo com score menor?
4. O que o estudo da árvore isolada ensina sobre ensembles?

Capítulo 13

Glossário Técnico

Este glossário reúne termos centrais usados ao longo do livro. A ideia é oferecer uma referência rápida para revisão.

- Acurácia: Proporção de previsões corretas sobre o total de exemplos avaliados.
- Atributo: Variável de entrada usada para descrever cada exemplo.
- Bagging: Estratégia de ensemble que combina modelos treinados em diferentes reamostragens dos dados para reduzir variância.
- CART: Sigla para `Classification and Regression Trees`, família de algoritmos de árvore com grande relevância prática.
- Classe majoritária: Classe mais frequente em um conjunto de exemplos ou em uma folha.
- Classificação: Tarefa de prever categorias discretas.
- Critério de divisão: Medida usada para decidir qual atributo ou corte produz a melhor separação em um nó.
- Entropia: Medida de incerteza ou impureza derivada da teoria da informação.
- Ensemble: Modelo formado pela combinação de vários modelos base, como várias árvores.
- Folha: Nó terminal da árvore onde a previsão final é produzida.
- Gain ratio: Refinamento do ganho de informação usado para reduzir o viés por atributos com muitos valores.
- Ganho de informação: Redução esperada da entropia após uma divisão.
- Generalização: Capacidade do modelo de funcionar bem em dados não vistos durante o treino.

- Gradient Boosting: Família de ensembles sequenciais em que novas árvores corrigem erros das anteriores.
- Grid Search: Procedimento sistemático de teste de combinações de hiperparâmetros.
- Gini: Medida de impureza muito usada em árvores de classificação.
- Hiperparâmetro: Configuração definida antes do treino, como profundidade máxima ou número mínimo de amostras por folha.
- ID3: Algoritmo clássico de indução de árvores guiado por ganho de informação.
- Impureza: Grau de mistura entre classes dentro de um nó.
- Instabilidade estrutural: Sensibilidade da árvore a pequenas mudanças no conjunto de treino.
- Interpretabilidade: Facilidade com que humanos conseguem entender o funcionamento e as previsões do modelo.
- Nó interno: Nó que realiza um teste e envia os exemplos para ramos descendentes.
- Nó raiz: Primeiro nó da árvore, onde a decisão começa.
- Overfitting: Situação em que o modelo se ajusta demais ao treino e perde generalização.
- Poda: Processo de simplificação da árvore para reduzir complexidade e melhorar generalização.
- Pós-poda: Redução da árvore após o treino.
- Pre-poda: Restrições aplicadas durante o crescimento da árvore.
- Precision: Entre as previsões positivas, proporção das que realmente são positivas.
- Pureza: Estado em que um nó contém exemplos de uma única classe ou de distribuição muito concentrada.
- Random Forest: Ensemble de várias árvores treinadas com bagging e seleção aleatória de atributos.
- Recall: Entre os exemplos realmente positivos, proporção dos que foram encontrados pelo modelo.
- Regressão: Tarefa de prever valores numéricos contínuos.
- Regra de decisão: Descrição textual de um caminho da raiz até uma folha.
- Splitter: Mecanismo que define como a biblioteca procura o melhor corte em um nó.

- Teste: Conjunto de dados reservado para avaliação final do modelo.
- Treino: Conjunto de dados usado para ajustar os parâmetros do modelo.
- Underfitting: Situação em que o modelo é simples demais para capturar os padrões do problema.
- Validação cruzada: Método de avaliação em que os dados são divididos em vários subconjuntos para treino e validação repetidos.
- Variância do modelo: Sensibilidade do modelo a mudanças no conjunto de treino.

Capítulo 14

Sobre os autores

Giseldo da Silva Neo

GISELDO DA SILVA NEO é Professor de Informática no Instituto Federal de Alagoas (IFAL) e desenvolve pesquisas na área de IA. Doutorado em Ciência da Computação na Universidade Federal de Campina Grande (UFCG). Possui Mestrado em Modelagem Computacional do Conhecimento (UFAL) e Mestrado em Contabilidade (FUCAPE). Possui MBA em Gestão e Estratégia Empresarial , Especialização em Arquitetura e Engenharia de Software, MBA em Gestão de Projetos. Graduação em Análise e Desenvolvimento de Sistemas e Graduação em Processos Gerenciais e possui nível Técnico em Informática (ETFSE - Escola Técnica Federal de Sergipe).

Alana Viana Borges da Silva Neo

ALANA VIANA BORGES DA SILVA NEO é Professora de Informática no Instituto Federal do Mato Grosso do Sul (IFMS) e desenvolve pesquisas na área de Informática na Educação. Doutoranda em Ciência da Computação na Universidade Federal de Campina Grande (UFCG), Mestra em Modelagem Computacional do Conhecimento na Universidade Federal de Alagoas (UFAL), Especialista em Estratégias Didáticas para a Educação Básica com Uso de TIC na Universidade Federal de Alagoas (UFAL), Especialista em Desenvolvimento de Software, Especialista em Segurança da Informação, Graduada em Análise e Desenvolvimento de Sistemas e Bacharel em Sistemas de Informação pela Universidade Estácio de Sá (ESTÁCIO) e Licenciatura em Computação pelo Claretiano Centro Universitário.

Contato

Caso deseje entrar em contato com os autores para reportar algum erro, crítica ou sugestão, envie e-mail para giseldo@gmail.com ou acesse o site https://giseldo.github.io/

Aviso Legal

As informações fornecidas neste trabalho são apenas para fins educacionais e informativos. Embora todos os esforços tenham sido feitos para garantir a precisão e a integridade, o autor e o editor não fazem declarações ou garantias de qualquer tipo, expressas ou implícitas, em relação à precisão, confiabilidade ou completude do conteúdo.

O autor e o editor não poderão ser responsabilizados por quaisquer danos ou perdas decorrentes do uso deste material. As opiniões expressas são de responsabilidade exclusiva do autor e não refletem necessariamente as opiniões de qualquer instituição ou organização com a qual o autor possa estar vinculado.

Uso da IA Generativa

Algumas partes textuais e algumas imagens foram criadas ou alteradas com várias das IAs Generativas disponíveis no momento da escrita. Porém, todo o texto foi revisado pelos autores e revisores.