Resumo IA

Resumo tópicos da Prova

Outubro 2025
20.10.2025

INFORMAÇÕES

Baseado nos materiais de aula
Cópia livre para a circulação

AUTOR

Enzo Serenato
enzoserenato@gmail.com
Ponta Grossa, Paraná

Sumário

1. Métodos de Resolução de Problemas e Busca em Espaço de Estados

1.1. Conceitos Fundamentais

Termo	Definição
Agente	Entidade que toma decisões e executa ações no ambiente
Estado	Configuração específica do problema em um momento
Espaço de Estados	Conjunto de todos os estados possíveis acessíveis
Ação	Movimento que o agente pode fazer para transitar entre estados
Objetivo	Estado final desejado que o agente busca alcançar
Função Sucessor	Define quais novos estados podem ser alcançados
Árvore de Busca	Representação dos caminhos possíveis para resolver o problema
Solução	Sequência de ações do estado inicial ao objetivo

1.2. Busca sem Informação

Algoritmos que exploram o espaço de estados sem conhecimento adicional sobre a proximidade do objetivo.

Algoritmo	Estrutura	Ordem	Melhor caminho	Memória	Aplicação
BFS	Fila (FIFO)	Nível por nível	Se custo uniforme	Alta	Caminho mais curto
DFS	Pilha (LIFO)	Um caminho por vez	Não	Baixa	Jogos
UCS	Fila de prioridade	Menor custo primeiro	Sempre	Alta	Caminho de menor custo

1.2.1. BFS (Breadth-First Search)

Explora todos os nós no mesmo nível antes de avançar para níveis mais profundos.

- **Estratégia**: Explora em largura, nível por nível
- **Estrutura**: Fila (FIFO)
- **Propriedades**: Completo, ótimo para custo uniforme
- **Complexidade**: $O(V + E)$ de espaço

```
from collections import deque
```

```python
def bfs(grafo, inicio, objetivo):
    fila = deque([(inicio, [inicio])])
    visitados = set()

    while fila:
        no, caminho = fila.popleft()
        if no in visitados:
            continue
        visitados.add(no)
        print(f"Visitando {no}")

        if no == objetivo:
            return caminho

        for vizinho in grafo.get(no, []):
            if vizinho not in visitados:
                fila.append((vizinho, caminho + [vizinho]))

    return None
```

1.2.2. DFS (Depth-First Search)

Explora um caminho completamente antes de retornar e tentar outro.

- **Estratégia**: Explora em profundidade primeiro
- **Estrutura**: Pilha (LIFO)
- **Propriedades**: Não é ótimo, pode entrar em loops
- **Complexidade**: $O(V + E)$ tempo, $O(V)$ ou $O(d)$ espaço

```python
def dfs(grafo, inicio, objetivo):
    pilha = [(inicio, [inicio])]
    visitados = set()

    while pilha:
        no, caminho = pilha.pop()
        if no in visitados:
            continue
        visitados.add(no)
        print(f"Visitando {no}")

        if no == objetivo:
            return caminho

        for vizinho in grafo.get(no, []):
            if vizinho not in visitados:
                pilha.append((vizinho, caminho + [vizinho]))

    return None
```

1.2.3. UCS (Uniform Cost Search)

- **Estratégia**: Expande o nó com menor custo acumulado
- **Estrutura**: Fila de prioridade (heap)
- **Propriedades**: Completo e ótimo
- **Aplicação**: Grafos com custos diferentes nas arestas

```python
import heapq
```

```
def ucs(grafo, inicio, objetivo):
    fila = [(0, inicio, [inicio])]
    custo_real = {cidade: float('inf') for cidade in grafo}
    custo_real[inicio] = 0

    while fila:
        custo, no, caminho = heapq.heappop(fila)

        if no == objetivo:
            print(f"Caminho encontrado: {caminho} com custo {custo}")
            return caminho

        for vizinho, custo_vizinho in grafo.get(no, []):
            novo_custo = custo + custo_vizinho
            if novo_custo < custo_real[vizinho]:
                custo_real[vizinho] = novo_custo
                heapq.heappush(fila, (novo_custo, vizinho, caminho +
[vizinho]))

    return None
```

1.3. Busca com Informação (Heurísticas)

Busca informada usa heurísticas — estimativas inteligentes ou regras práticas — para guiar a exploração em direção aos caminhos mais promissores, tornando-a mais eficiente que a busca cega.

Função Heurística h(n): Estimativa do custo para alcançar o objetivo a partir do nó n. Uma boa heurística é rápida de calcular e fornece estimativa razoável.

1.3.1. Busca Gulosa (Greedy Best-First Search)

- **Função**: f(n) = h(n)
- **Estratégia**: Expande o nó que parece estar mais próximo do objetivo, baseado apenas na heurística, ou seja, sempre escolhe o nó com o menor valor de h(n).
- **Vantagens**: Simples e frequentemente rápida
- **Desvantagens**: Não garante solução ótima, pode ficar presa em “mínimos locais”

```
def busca_gulosa(origem, destino):
    caminho = [origem]
    cidade_atual = origem

    while cidade_atual != destino:
        vizinhos = grafo[cidade_atual].keys()

        # Escolhe vizinho com menor heurística
        proxima_cidade = min(vizinhos,
            key=lambda cidade: distancias_em_linha_reta[cidade])

        caminho.append(proxima_cidade)
        cidade_atual = proxima_cidade

    return caminho
```

1.3.2. A* (A-Estrela)

- **Função de avaliação**: f(n) = g(n) + h(n)

    - g(n): custo real do caminho do início até n
    - h(n): estimativa heurística de n até objetivo
- **Estratégia**: Equilibra o custo já incorrido com o custo futuro estimado
- **Vantagens**: Garante solução ótima se heurística for bem escolhida
- **Desvantagens**: Pode ser computacionalmente intensivo
- **Heurística admissível**: Nunca superestima o custo real (h(n) ≤ custo real)

```
import heapq

def a_estrela(origem, destino):
    fila = []
    heapq.heappush(fila, (distancias_em_linha_reta[origem],
                          0, origem, [origem]))

    custo_real = {cidade: float('inf') for cidade in grafo}
    custo_real[origem] = 0

    while fila:
        _, g_atual, cidade_atual, caminho = heapq.heappop(fila)

        if cidade_atual == destino:
            return caminho

        for vizinho, distancia in grafo[cidade_atual].items():
            novo_g = g_atual + distancia

            if novo_g < custo_real[vizinho]:
                custo_real[vizinho] = novo_g
                f_score = novo_g + distancias_em_linha_reta[vizinho]
                novo_caminho = caminho + [vizinho]
                heapq.heappush(fila, (f_score, novo_g, vizinho,
novo_caminho))

    return None
```

2. Satisfação de Restrições (CSP)

CSPs são problemas onde o objetivo não é encontrar um caminho, mas um estado que satisfaça um conjunto de restrições especificadas.

2.1. Estrutura de um CSP

Definido por uma tripla (X, D, C):

- **Variáveis (X)**: Conjunto de variáveis X, Y, Z
- **Domínios (D)**: Conjunto de valores possíveis para cada variável $X \in \{1, 2, 3\}, Y \in \{a, b, c\}$
- **Restrições (C)**: Conjunto de regras $X \neq Y, Y < Z$ que especificam combinações permitidas

2.2. Exemplo: Coloração de Mapas

Variáveis: Regiões do mapa (N, NE, SE, S, CO)

Domínio: Cores disponíveis {vermelho, verde, azul}

Restrições: Regiões adjacentes devem ter cores diferentes (N ≠ NE)

2.3. Algoritmo Backtracking

Algoritmo sistemático para resolver CSPs:

1. Escolher uma variável para atribuir valor
2. Atribuir um valor válido do domínio
3. Verificar se a atribuição é consistente com todas as restrições
   - Se válida: prosseguir para próxima variável
   - Se inválida: backtrack (desfazer e tentar outro valor)
4. Repetir até encontrar solução completa ou esgotar possibilidades

Exemplos clássicos: N-Rainhas, Sudoku, coloração de grafos, agendamento

3. Aprendizado de Máquina

- **IA**: Ciência ampla de criar sistemas inteligentes
- **ML**: Subcampo da IA que usa algoritmos para aprender com dados
- **Deep Learning**: Subcampo do ML que usa redes neurais profundas

3.1. Tipos de Aprendizado

Tipo	Descrição	Dados	Casos de Uso
Supervisionado	Modelo treinado com entradas e saídas esperadas	Rotulados	Previsão de preços, diagnóstico médico
Não Supervisionado	Identifica padrões sem rótulos predefinidos	Não rotulados	Segmentação de clientes, clustering
Por Reforço	Agente aprende interagindo com ambiente	N/A	Robótica, jogos (AlphaGo)

3.2. Fluxo de Trabalho em ML

1. **Coleta de Dados**: Reunir dados relevantes
2. **Pré-processamento**: Limpar, organizar e transformar dados
3. **Divisão de Dados**: Separar em conjuntos de treino e teste
4. **Seleção de Modelo**: Escolher algoritmo apropriado
5. **Treinamento**: Ajustar modelo aos dados de treino
6. **Avaliação**: Medir desempenho nos dados de teste
7. **Predição**: Usar modelo em dados novos

3.3. Manipulação de Dados com Pandas

Pandas é biblioteca central para manipulação de dados em Python.

Operações principais:

- **Valores ausentes**: `df.isnull().sum()` (identificar), `df.dropna()` (remover), `df.fillna()` (preencher)
- **Duplicatas**: `df.duplicated()` (verificar), `df.drop_duplicates()` (remover)
- **Inconsistências**: `str.lower()`, `str.strip()`, `str.replace()` para padronização
- **Transformação**: Criar colunas, filtrar dados

```python
import pandas as pd

# Criar DataFrame
data = {
    'Nome': ['Ana', 'Bruno', 'Carlos', None, 'Ana'],
    'Idade': [23, None, 30, 25, 23],
    'Salário': [4000, 5000, None, 3500, 4000],
    'Cidade': ['São Paulo', 'São Paulo', 'Rio', 'Belo Horizonte', 'São
Paulo']
}
df = pd.DataFrame(data)

# Identificar valores ausentes
print(df.isnull().sum())

# Preencher valores ausentes com média
df['Idade'] = df['Idade'].fillna(df['Idade'].mean())

# Remover duplicatas
df = df.drop_duplicates()

# Padronização
df['Cidade'] = df['Cidade'].str.lower()

# Criar nova coluna
df['Salário Anual'] = df['Salário'] * 12

# Filtrar dados
df_filtrado = df[df['Idade'] > 25]
```

str.strip(): Remove espaços no início e no final da string. str.lower(): Converte todas as letras para minúsculas. str.upper(): Converte todas as letras para maiúsculas. str.title(): Converte a primeira letra de cada palavra para maiúscula. str.replace(old, new): Substitui ocorrências de um valor antigo (old) por um novo (new). str.startswith(prefix): Verifica se a string começa com um prefixo específico. str.endswith(suffix): Verifica se a string termina com um sufixo específico. str.split(delimiter): Divide a string em partes com base em um delimitador. str.normalize('NFKD'): Remove acentos e normaliza caracteres Unicode.

3.4. Estatística Básica e Probabilidade

3.4.1. NumPy e SciPy

NumPy: Essencial para operações numéricas, criar arrays (`np.array`), cálculos estatísticos (`np.mean`, `np.std`)

SciPy: Construído sobre NumPy para computação científica avançada, módulo `scipy.stats` para distribuições

3.4.2. Medidas Estatísticas

Tendência Central: Média, mediana, moda

Dispersão: Variância, desvio padrão, amplitude

3.4.3. Distribuições de Probabilidade

Normal: Modela dados contínuos que se agrupam em torno da média (ex: altura)

Características:

- Modelo de dados contínuos.
- Exemplo: Altura de pessoas.
- Formato de sino: valores próximos à média são mais comuns.
- Gráfico: Largo no meio (média), estreita nas extremidades (valores raros).

Z-score: Quão próximo da média. Fórmula:

$$Z = \frac{x - \mu}{\sigma}$$

onde x: valor observado, μ: média, σ: desvio padrão.

Exemplo: Média (μ) de uma turma em um teste: 70 pontos. Desvio padrão (σ): 10 pontos. Aluno tirou x = 85. Z = (85 - 70) / 10 = 1.5. O Z-score é 1.5, ou seja, o aluno está 1.5 desvios padrão acima da média.

```
import numpy as np
from scipy.stats import norm

media = 0.90
desvio = 0.02

# Probabilidade de precisão < 0.88
prob = norm.cdf(0.88, loc=media, scale=desvio)
print(f"Probabilidade: {prob:.4f}")
```

Binomial: Experimentos com dois resultados possíveis (sucesso/fracasso) em número fixo de tentativas

Características:

- Experimentos com dois resultados possíveis (sucesso ou falha).
- Exemplo: Jogar uma moeda 10 vezes. Resultados: “cara” ou “coroa”.
- Aplicações: Aprovação em provas, jogos, etc.

Fórmula:

$$P(X = k) = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}$$

onde n: número de tentativas, k: número de sucessos, p: probabilidade de sucesso em uma tentativa.

Exemplo: Probabilidade de obter exatamente 3 “caras” em 10 jogadas com p=0.5: P(X=3) = 10 **(0.5)^3** (0.5)^7 = 0.3125 (31.25%).

```
from scipy.stats import binom

# Probabilidade de exatamente 7 acertos em 10 previsões com 80% de chance
prob = binom.pmf(k=7, n=10, p=0.8)
print(f"P(X=7): {prob}")
```

Poisson: Número de eventos raros em intervalo fixo (ex: falhas de sistema por hora)

```
from scipy.stats import poisson

# Chance de exatamente 3 erros, com média 2 por hora
prob = poisson.pmf(k=3, mu=2)
print(f"P(X=3): {prob}")
```

3.4.4. Conceitos Fundamentais

- Probabilidade condicional
- Teorema de Bayes
- Correlação vs. causalidade

3.5. Visualização de Dados

Matplotlib: Biblioteca fundamental com controle completo sobre elementos gráficos

Seaborn: Construído sobre Matplotlib, simplifica criação de gráficos estatísticos

3.5.1. Tipos de Gráficos

Tipo	Uso
Line Plot	Tendências ao longo do tempo
Scatter Plot	Relação entre duas variáveis numéricas
Bar Plot	Comparar valores entre categorias
Histogram	Distribuição de frequência de variável numérica
Box Plot	Resumo de distribuição (mediana, quartis, outliers)
Heatmap	Visualizar relações em matriz (ex: correlação)
Pairplot	Relações entre múltiplas variáveis

3.5.2. Exemplos

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Matplotlib - Linha
x = np.linspace(0, 10, 20)
y = np.sin(x)
plt.plot(x, y, marker='o')
plt.title("Gráfico de Linha")

# Seaborn - Dispersão com cores por categoria
df = pd.DataFrame({
    "Categoria": ["A", "A", "B", "B", "C", "C"],
    "X": [1, 2, 3, 4, 5, 6],
    "Y": [2, 4, 1, 3, 5, 7],
})
sns.scatterplot(x="X", y="Y", hue="Categoria", data=df)

# Heatmap de correlação
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap="coolwarm")

# Pairplot para múltiplas variáveis
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue="species")
```

4. Algoritmos de Aprendizado Supervisionado

4.1. Regressão Linear

Objetivo: Prever valores contínuos (ex: preço, temperatura)

Método: Modela relação linear entre variável independente x e dependente y, ajustando linha que minimiza Mean Squared Error (MSE)

Equação: $y = \beta_0 + \beta_1 x$ com $\beta_0 = \overline{y} - \beta_1 \overline{x}$ (bias) e $\beta_1 = \frac{\sum((x-\overline{x})(y-\overline{y}))}{\sum((x-\overline{x})^2)}$ (coeficiente angular)

Métricas de Avaliação:

- **MSE**: Mean Squared Error (média dos erros ao quadrado)
- **RMSE**: Root Mean Squared Error (raiz do MSE)
- **R^2**: Coeficiente de determinação (qualidade do ajuste)

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

data = {
    'tempo_estudo': [2, 4, 6, 8, 10, 12],
    'frequencia': [50, 60, 70, 80, 90, 100],
    'nota_final': [5, 6, 7, 8, 9, 10]
}
df = pd.DataFrame(data)

X = df[['tempo_estudo', 'frequencia']]
y = df['nota_final']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

modelo = LinearRegression()
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print(f"MSE: {mse}")
print("Coeficientes:", modelo.coef_)
```

4.2. Regressão Logística

Objetivo: Classificação binária (ex: sim/não, doente/saudável)

Método: Usa função sigmoide para transformar saída de equação linear em probabilidade (entre 0 e 1). Parâmetros otimizados com Gradient Descent para maximizar verossimilhança

Função:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

Decisão: Threshold (geralmente 0.5) para classificação final

Métricas de Avaliação:

- **Acurácia**: (VP + VN) / Total
- **Precisão**: VP / (VP + FP)
- **Recall**: VP / (VP + FN)
- **F1-Score**: Média harmônica entre precisão e recall
- **Matriz de Confusão**: Tabela com VP, VN, FP, FN

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

modelo = LogisticRegression()
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

4.3. Árvore de Decisão

Tipo: Aprendizado Supervisionado

Objetivo: Classificação ou regressão através de regras de decisão hierárquicas

Estrutura: Modelo em árvore com nós de decisão e folhas (classes)

Método: Divide dados em subgrupos baseados nos atributos (features) mais informativos em cada nó, buscando criar folhas “puras” (todos os pontos de uma classe)

Vantagem Principal: Equilibra trade-off entre bias (simplificação excessiva) e variance (overfitting)

Critérios de Divisão:

- **Gini**: Mede impureza do nó (0 = nó perfeitamente puro)
- **Entropia**: Mede desordem dos dados

Algoritmo CART: Usado no Scikit-learn, utiliza Gini Index

Hiperparâmetros Importantes:

- **max_depth**: Profundidade máxima da árvore
- **min_samples_split**: Mínimo de amostras para dividir nó
- **criterion**: ‘gini’ ou ‘entropy’

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

data = {
    'Peso': [150, 170, 140, 130, 155, 135, 165, 145],
    'Textura': [1, 0, 1, 0, 1, 0, 0, 1],
    'Fruta': ['maçã', 'laranja', 'maçã', 'laranja', 'maçã', 'laranja',
'laranja', 'maçã']
}
df = pd.DataFrame(data)

X = df[['Peso', 'Textura']]
y = df['Fruta']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

modelo = DecisionTreeClassifier(criterion='gini', max_depth=2,
random_state=42)
modelo.fit(X_train, y_train)

y_pred = modelo.predict(X_test)

print("Acurácia:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure(figsize=(8,6))
plot_tree(modelo, feature_names=X.columns, class_names=modelo.classes_,
filled=True)
plt.show()
```

Métricas de Avaliação:

- Acurácia
- Classification report (precisão, recall, F1)
- Confusion matrix

5. Algoritmos de Aprendizado Não Supervisionado

5.1. K-Means (Clusterização)

Tipo: Aprendizado Não Supervisionado

Objetivo: Agrupar dados em k clusters, de forma que pontos dentro de cada grupo sejam similares entre si e diferentes dos outros grupos

Critério de Semelhança: Distância euclidiana

Algoritmo Iterativo:

1. **Inicializar**: Colocar k centróides aleatoriamente
2. **Atribuir**: Atribuir cada ponto ao centróide mais próximo
3. **Atualizar**: Recalcular centróides como média dos pontos atribuídos
4. **Repetir**: Continuar passos 2-3 até centróides não se moverem significativamente

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

clientes = {
    'cliente_id': [101, 102, 103, 104, 105, 106, 107, 108, 109, 110],
    'tempo_na_loja': [10, 45, 30, 20, 60, 15, 50, 40, 25, 35],
    'valor_medio_compra': [20, 200, 150, 50, 300, 25, 250, 180, 60, 100],
    'visitas_mensais': [1, 5, 4, 2, 6, 1, 5, 4, 2, 3]
}
df = pd.DataFrame(clientes)

X = df[['tempo_na_loja', 'valor_medio_compra', 'visitas_mensais']]

kmeans = KMeans(n_clusters=2, random_state=42)
df['cluster'] = kmeans.fit_predict(X)
```

```
print("Centróides:", kmeans.cluster_centers_)
print(df)

plt.scatter(df['tempo_na_loja'], df['valor_medio_compra'],
            c=df['cluster'], cmap='viridis', s=100)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            c='red', s=200, marker='X')
plt.title("Clusters de Clientes")
plt.show()
```

5.1.1. Determinando o Número Ideal de Clusters

Método do Cotovelo (Elbow Method):

- Calcula inércia (soma das distâncias dos pontos ao centróide) para vários valores de k
- Plota k vs. inércia
- Procura ponto onde inércia para de diminuir significativamente (o “cotovelo”)

Silhouette Score:

- Mede quão bem separado e coeso está cada cluster
- Valores próximos de 1 = melhor separação
- Calcula para diferentes valores de k e escolhe maior score

```
# Método do Cotovelo
inercia = []
for k in range(1, 11):
    modelo = KMeans(n_clusters=k, random_state=42)
    modelo.fit(X)
    inercia.append(modelo.inertia_)

plt.plot(range(1, 11), inercia, marker='o')
plt.xlabel('Número de Clusters (k)')
plt.ylabel('Inércia')
plt.show()

# Silhouette Score
from sklearn.metrics import silhouette_score

silhouette_scores = []
for k in range(2, 10):
    modelo = KMeans(n_clusters=k, random_state=42)
    labels = modelo.fit_predict(X)
    score = silhouette_score(X, labels)
    silhouette_scores.append(score)
    print(f'k={k}, Silhouette = {score:.4f}')

plt.plot(range(2, 10), silhouette_scores)
plt.xlabel('Número de Clusters (k)')
plt.ylabel('Silhouette Score')
plt.show()
```

Métrica de Avaliação:

- **Inércia**: Quanto menor, mais compactos os clusters

6. Pipelines e Otimização de Modelos

6.1. Construção de Pipelines

Pipeline é sequência de transformações seguida de estimador final. Facilita organização do código e evita vazamento de dados (data leakage).

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression

# Exemplo com pré-processamento
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['age', 'balance']),
        ('cat', OneHotEncoder(), ['job', 'marital'])
    ])

pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression())
])

pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

6.2. Ajuste de Hiperparâmetros

Grid Search: Testa todas as combinações possíveis de hiperparâmetros para encontrar a melhor configuração.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'classifier__max_iter': [100, 500, 1000],
    'classifier__C': [0.1, 1, 10]
}

grid_search = GridSearchCV(pipeline, param_grid, cv=5)
grid_search.fit(X_train, y_train)

print(f"Melhores parâmetros: {grid_search.best_params_}")
```

6.3. Balanceamento de Classes

Técnicas para lidar com datasets desbalanceados:

- **Oversampling**: SMOTE (cria exemplos sintéticos da classe minoritária)
- **Undersampling**: Remove exemplos da classe majoritária
- **Class weights**: Ajusta pesos das classes no modelo

```
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression

smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_train, y_train)

model = LogisticRegression(class_weight='balanced', max_iter=1000)
model.fit(X_res, y_res)
```

6.4. Métricas de Avaliação Detalhadas

Métrica	Fórmula	Quando Usar
Acurácia	(VP + VN) / Total	Visão geral do modelo
Precisão	VP / (VP + FP)	Minimizar falsos positivos
Recall	VP / (VP + FN)	Minimizar falsos negativos
F1-Score	2 × (P × R) / (P + R)	Balancear precisão e recall

Legenda:

- VP: Verdadeiros Positivos
- VN: Verdadeiros Negativos
- FP: Falsos Positivos
- FN: Falsos Negativos

7. Validação de Modelos

7.1. Divisão de Dados

Separar dados em treino e teste para avaliar generalização do modelo:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

test_size=0.2: 20% dos dados para teste, 80% para treino **stratify=y**: Mantém distribuição de classes no split

7.2. Validação Cruzada (Cross-Validation)

Técnica mais robusta que divisão simples. Divide dados em k partições (folds), treina k vezes usando k-1 folds para treino e 1 para validação.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(modelo, X, y, cv=5)
print(f"Acurácia média: {scores.mean():.2f} (+/- {scores.std():.2f})")
```

cv=5: 5-fold cross-validation

Vantagem: Usa todos os dados para treino e validação, reduz variância da estimativa