

# 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
1.1. Conceitos Fundamentais .....	1
1.2. Busca sem Informação .....	1
1.2.1. BFS (Breadth-First Search) .....	1
1.2.2. DFS (Depth-First Search) .....	2
1.2.3. UCS (Uniform Cost Search) .....	2
1.3. Busca com Informação (Heurísticas) .....	3
1.3.1. Busca Gulosa (Greedy Best-First Search) .....	3
1.3.2. A* (A-Estrela) .....	3
2. Satisfação de Restrições (CSP) .....	4
2.1. Estrutura de um CSP .....	4
2.2. Exemplo: Coloração de Mapas .....	4
2.3. Algoritmo Backtracking .....	4
3. Aprendizado de Máquina .....	5
3.1. Tipos de Aprendizado .....	5
3.2. Fluxo de Trabalho em ML .....	5
3.3. Manipulação de Dados com Pandas .....	5
3.4. Estatística Básica e Probabilidade .....	6
3.4.1. NumPy e SciPy .....	6
3.4.2. Medidas Estatísticas .....	6
3.4.3. Distribuições de Probabilidade .....	6
3.4.4. Conceitos Fundamentais .....	6
3.5. Visualização de Dados .....	7
3.5.1. Tipos de Gráficos .....	7
3.5.2. Exemplos .....	7
4. Algoritmos de Aprendizado Supervisionado .....	7
4.1. Regressão Linear .....	7
4.2. Regressão Logística .....	8
5. Algoritmos de Aprendizado Não Supervisionado .....	8
5.1. K-Means (Clusterização) .....	8
5.1.1. Determinando o Número Ideal de Clusters .....	9
5.2. Árvore de Decisão .....	9
6. Pipelines e Otimização de Modelos .....	10
6.1. Construção de Pipelines .....	10
6.2. Ajuste de Hiperparâmetros .....	10
6.3. Balanceamento de Classes .....	11
6.4. Métricas de Avaliação Detalhadas .....	11
7. Validação de Modelos .....	11
7.1. Divisão de Dados .....	11
7.2. Validação Cruzada (Cross-Validation) .....	11

# 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

```
def bfs(grafo, start, end):
    queue = deque([(start, [start])])
    visitado = set()

    while queue:
        node, path = queue.popleft()
        if node in visitado:
            continue
        visitado.add(node)

        if node == end:
            return path

        for vizinho in grafo.neighbors(node):
            if vizinho not in visitado:
                queue.append((vizinho, path + [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

```
def dfs(grafo, start, end):
    stack = [(start, [start])]
    visitado = set()

    while stack:
        node, path = stack.pop()
        if node in visitado:
            continue
        visitado.add(node)

        if node == end:
            return path

        for vizinho in grafo.neighbors(node):
            if vizinho not in visitado:
                stack.append((vizinho, path + [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
- **Apliação:** Grafos com custos diferentes nas arestas

```
def ucs(grafo, start, end):
    priority_queue = [(0, start, [start])]
    visitado = set()
```

```

while priority_queue:
    cost, node, path = heapq.heappop(priority_queue)
    if node in visitado:
        continue
    visitado.add(node)

    if node == end:
        return path, cost

    for vizinho in grafo.neighbors(node):
        if vizinho not in visitado:
            edge_weight = grafo[node][vizinho].get('weight', 1)
            heapq.heappush(priority_queue,
                           (cost + edge_weight, vizinho, path + [vizinho]))

return None, float('inf')

```

### 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 ( $h(n) \leq \text{custo real}$ )

```
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 ( $N \neq NE$ )

### 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

```
import pandas as pd
```

```
# Criar DataFrame
df = pd.DataFrame({
    'Nome': ['Ana', 'Bruno', 'Carlos'],
    'Idade': [23, 30, 25],
    'Salário': [4000, 5000, 3500]
})
```

```
# Identificar valores ausentes
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)

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

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

#### 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

# Matplotlib - Linha
plt.plot(x, y, marker='o')
plt.title("Gráfico de Linha")

# Seaborn - Dispersão com cores por categoria
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
sns.pairplot(df, 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  $\beta_0 = \bar{y} - \beta_1 \bar{x}$  (bias) e  $\beta_1 = \frac{\sum((x-\bar{x})(y-\bar{y}))}{\sum((x-\bar{x})^2)}$  (coeficiente angular)

**Métricas de Avaliação:**

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

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

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

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

## 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))
```

## 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

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=42)
df['cluster'] = kmeans.fit_predict(X)

# Centróides
print(kmeans.cluster_centers_)
```

### 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

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

#### Métrica de Avaliação:

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

### 5.2. 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'

```
from sklearn.tree import DecisionTreeClassifier, plot_tree

modelo = DecisionTreeClassifier(criterion='gini', max_depth=3,
                                random_state=42)
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

# Visualizar árvore
plt.figure(figsize=(12,8))
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

## 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
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('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 = {
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10]
```

```
}
```

```
grid_search = GridSearchCV(DecisionTreeClassifier(), 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

### 6.4. Métricas de Avaliação Detalhadas

Métrica	Fórmula	Quando Usar
Acurácia	$(VP + VN) / \text{Total}$	Visão geral do modelo
Precisão	$VP / (VP + FP)$	Minimizar falsos positivos
Recall	$VP / (VP + FN)$	Minimizar falsos negativos
F1-Score	$2 \times (P \times R) / (P + R)$	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
)
```

**test\_size=0.2:** 20% dos dados para teste, 80% para treino

### 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