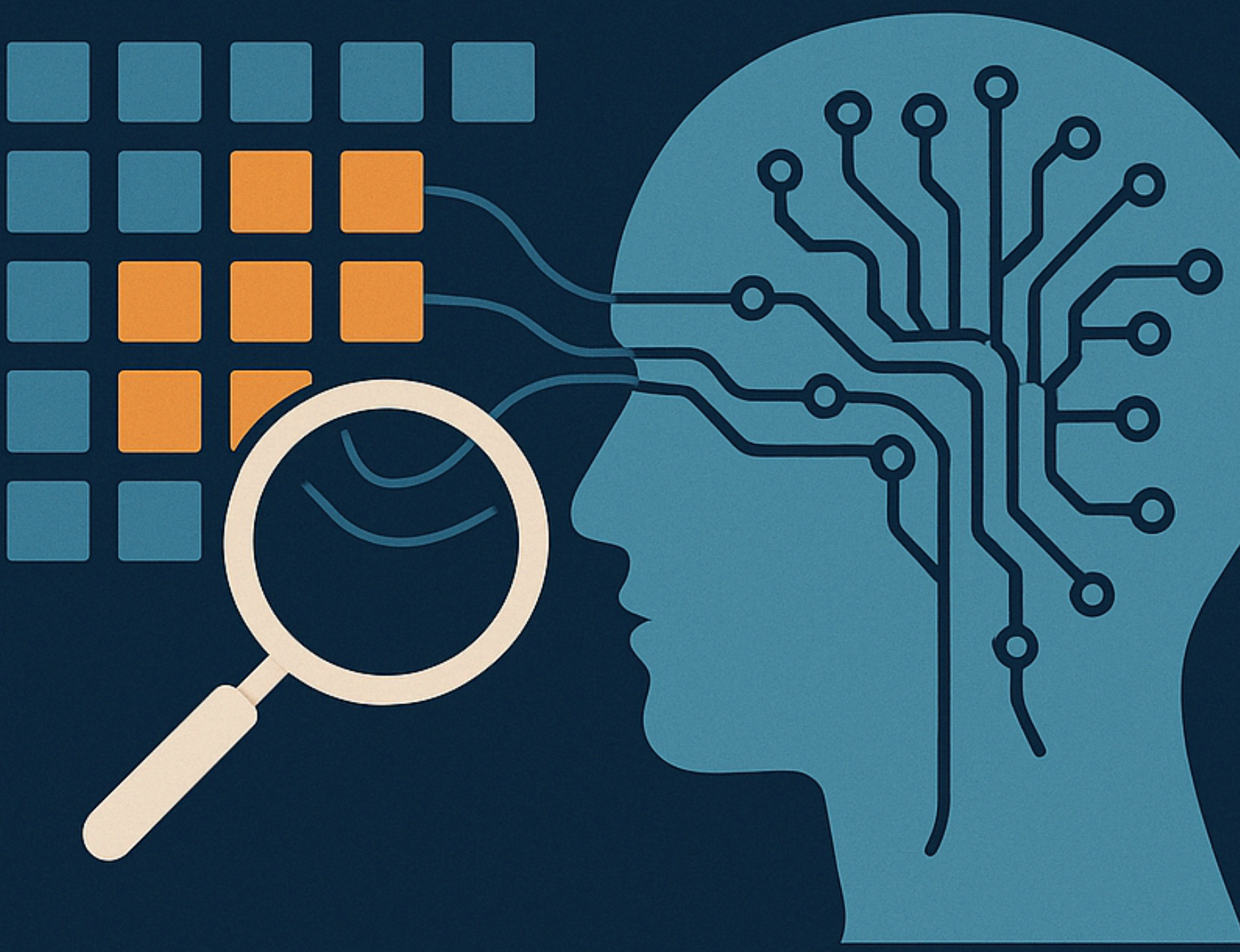
A ARTE DE SELECIONAR DADOS

AMOSTRAGEM PARA UMA IA MAIS INTELIGENTE



Introdução

Por que devemos "testar" nossos modelos?

Imagine que você criou um modelo de Inteligência Artificial para prever se um cliente vai ou não cancelar uma assinatura. Você treina esse modelo com os dados de 1.000 clientes e, ao final, ele acerta 99% das previsões nesses mesmos dados. Sucesso? Talvez não.

O verdadeiro teste de um modelo de IA não é o seu desempenho com os dados que ele já "decorou" durante o treino, mas sim sua capacidade de prever resultados para dados novos, que ele nunca viu antes.

É aqui que entram as técnicas de amostragem. Elas são estratégias para dividir nosso conjunto de dados de forma inteligente, permitindo simular esse cenário de "dados novos". Usando essas técnicas, conseguimos ter uma ideia muito mais realista e confiável de como nosso modelo se sairá no mundo real, evitando surpresas desagradáveis.

Neste guia, vamos explorar três das técnicas mais importantes e utilizadas no dia a dia de um cientista de dados: **Holdout, Validação Cruzada e Bootstrap.**

Holdout

A Divisão Simples

A técnica **Holdout** é a forma mais simples e direta de validar um modelo. A ideia é literalmente "segurar" (hold out) uma parte dos seus dados para usar como um teste final.

Como funciona?

1. Pegamos nosso conjunto de dados completo.
2. Dividimos ele em duas partes: uma maior para treino (geralmente 70% ou 80%) e uma menor para teste (os 30% ou 20% restantes).
3. Treinamos o modelo usando apenas o conjunto de treino.
4. Avaliamos o desempenho do modelo no conjunto de teste, que ele nunca viu. A performance nesse conjunto é a nossa estimativa de como ele funcionará no mundo real.

Vantagem: É muito rápido e simples de implementar. **Desvantagem**: O resultado pode variar muito dependendo de como a divisão dos dados foi feita. Se, por azar, o conjunto de teste contiver apenas exemplos "fáceis" ou "difíceis", nossa avaliação do modelo não será confiável.

```python
# Importando as bibliotecas necessárias
import numpy as np
from sklearn.model_selection import train_test_split

# Imaginando que temos 100 amostras de dados (X) e seus respectivos resultados (y)
X = np.arange(100).reshape(50, 2) # 50 amostras, 2 features cada
y = np.arange(50) # 50 resultados

# Dividindo os dados: 70% para treino, 30% para teste
# O 'random_state=42' garante que a divisão seja sempre a mesma, para podermos reproduzir os resultados
X_treino, X_teste, y_treino, y_teste = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Verificando o tamanho dos conjuntos criados
print(f"Tamanho do conjunto de treino: {X_treino.shape[0]} amostras")
print(f"Tamanho do conjunto de teste: {X_teste.shape[0]} amostras")

# Agora, você usaria X_treino e y_treino para treinar seu modelo
# E depois, usaria X_teste e y_teste para avaliá-lo
```

Validação Cruzada

Testando de Várias Maneiras

A Validação Cruzada é uma evolução do Holdout. Ela resolve a principal desvantagem do método anterior, que é a dependência de uma única divisão dos dados.

Como funciona?

A forma mais comum é a K-Fold Cross-Validation.

1. Dividimos nosso conjunto de dados em K partes iguais (ou "folds"). Um valor comum para K é 5 ou 10.
2. Fazemos um rodízio: treinamos o modelo K vezes.
3. Em cada rodada, usamos uma das K partes como teste e as outras K-1 partes como treino.
4. Ao final das K rodadas, teremos K resultados de performance. A métrica final é a média desses resultados, o que nos dá uma estimativa muito mais estável e confiável do desempenho do modelo.

Vantagem: Reduz a sorte (ou azar) da divisão dos dados, fornecendo uma métrica de performance muito mais robusta; **Desvantagem**: Exige mais processamento, pois o modelo é treinado K vezes.

```python
# Importando as bibliotecas necessárias
import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from sklearn.neighbors import KNeighborsClassifier # Um modelo de exemplo

# Imaginando os mesmos dados de antes
X = np.arange(100).reshape(50, 2)
y = np.arange(50) % 2 # Resultados binários (0 ou 1) para um problema de classificação

# 1. Criando um modelo de exemplo (classificador K-Vizinhos Mais Próximos)
modelo = KNeighborsClassifier(n_neighbors=3)

# 2. Configurando a Validação Cruzada (K-Fold) com K=5
# 'shuffle=True' embaralha os dados antes de dividir, o que é uma boa prática
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# 3. Executando a validação cruzada
# A função 'cross_val_score' faz todo o trabalho de treinar e avaliar 5 vezes
# 'scoring='accuracy'' define que queremos medir a acurácia
scores = cross_val_score(modelo, X, y, cv=kf, scoring='accuracy')
```

Bootstrap

Amostragem com Reposição

O Bootstrap é uma técnica poderosa usada não apenas para validar modelos, mas também para entender a incerteza e a variabilidade das nossas estimativas.

Como funciona?

A forma mais comum é a K-Fold Cross-Validation.

1. A partir do nosso conjunto de dados original (com N amostras), criamos um novo conjunto de dados (o "dataset bootstrap") do mesmo tamanho N.
2. Fazemos isso sorteando amostras do conjunto original com reposição. Isso significa que, após uma amostra ser escolhida, ela é "devolvida" ao conjunto original e pode ser escolhida novamente.
3. O resultado é um novo dataset onde algumas amostras originais aparecem várias vezes e outras não aparecem nenhuma vez.
4. Repetimos esse processo muitas vezes (ex: 1.000 vezes), criando 1.000 datasets bootstrap diferentes.
5. Treinamos nosso modelo em cada um desses datasets e guardamos os resultados. A distribuição desses resultados nos dá uma ideia da estabilidade do nosso modelo.

As amostras que não foram escolhidas para um determinado dataset bootstrap formam um conjunto chamado Out-of-Bag (OOB). Esse conjunto OOB funciona como um conjunto de teste natural para o modelo treinado naquele dataset bootstrap específico.

Vantagem: Excelente para datasets pequenos e para entender a estabilidade e o "intervalo de confiança" de uma métrica do modelo. É a base de algoritmos poderosos como o Random Forest. **Desvantagem**: Pode ser computacionalmente intensivo e a teoria por trás é um pouco menos intuitiva que as anteriores.

```python
# Importando as bibliotecas necessárias
import numpy as np
from sklearn.utils import resample # Função para reamostragem (bootstrap)
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

# Imaginando os mesmos dados de antes
X = np.arange(100).reshape(50, 2)
y = np.arange(50) % 2

# Número de vezes que vamos repetir o processo de bootstrap
n_iteracoes = 1000
acuracias = []

# Loop de Bootstrap
for i in range(n_iteracoes):
    # 1. Criar uma amostra de treino com reposição (bootstrap)
    X_treino, y_treino = resample(X, y, random_state=i)

    # Criar um modelo e treiná-lo com a amostra bootstrap
    modelo = KNeighborsClassifier(n_neighbors=3)
    modelo.fit(X_treino, y_treino)

    # 2. Avaliar o modelo nos dados originais (ou em um conjunto de teste separado)
    # Para simplicidade, vamos avaliar no conjunto completo 'X'
    predicoes = modelo.predict(X)
    score = accuracy_score(y, predicoes)
    acuracias.append(score)

# Analisando os resultados
acuracias = np.array(acuracias)
print(f"Após {n_iteracoes} iterações de Bootstrap:")
print(f"Acurácia Média: {acuracias.mean():.2f}")

# Calculando um "intervalo de confiança" de 95%
# Isso nos dá uma ideia da faixa onde a verdadeira acurácia provavelmente está
alpha = 0.95
p_inferior = ((1.0 - alpha) / 2.0) * 100
p_superior = (alpha + ((1.0 - alpha) / 2.0)) * 100
intervalo_confianca = np.percentile(acuracias, [p_inferior, p_superior])

print(f"Intervalo de Confiança de 95%: {intervalo_confianca.round(2)}")
```