

***INTELIGÊNCIA ARTIFICIAL***

**UA02 / LABORATÓRIO # 4**

**Parte 1 – Importação de Bibliotecas e Leitura de Dataset**

# Importação de Bibliotecas

import pandas as pd

from sklearn import datasets, linear\_model

from matplotlib import pyplot as plt

import numpy as np

from sklearn.model\_selection import KFold # Para estratégia de Cross-Validadion (K-Folds)

from sklearn.model\_selection import train\_test\_split # Para estratégia Train/Test Split

# Leitura de Dataset - Diabetes, do SKLearn

columns = "age sex bmi map tc ldl hdl tch ltg glu".split() # Declara os nomes das colunas

diabetes = datasets.load\_diabetes() # Carrega o dataset diabetes de sklearn

df = pd.DataFrame(diabetes.data, columns=columns) # Carrega o dataset como um data frame

y = diabetes.target # define a variavel target variable (varivavel dependente, no modelo de regressão)

**Parte 2 – Train/Test Split**

# Treinamento e Testes - Estratégia: Train/Test Split (20% para Testes)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df, y, test\_size=0.2)

print (X\_train.shape, y\_train.shape)

print (X\_test.shape, y\_test.shape)

**Saída:**

(353, 10) (353,)

(89, 10) (89,)

**Parte 3 – Modelo de Regressão Linear**

# Construção e Aplicação do Modelo: Regressão Linear

lm = linear\_model.LinearRegression()

model = lm.fit(X\_train, y\_train)

predictions = lm.predict(X\_test)

predictions[0:5]

# Exibição do Modelo: Scatter Plot

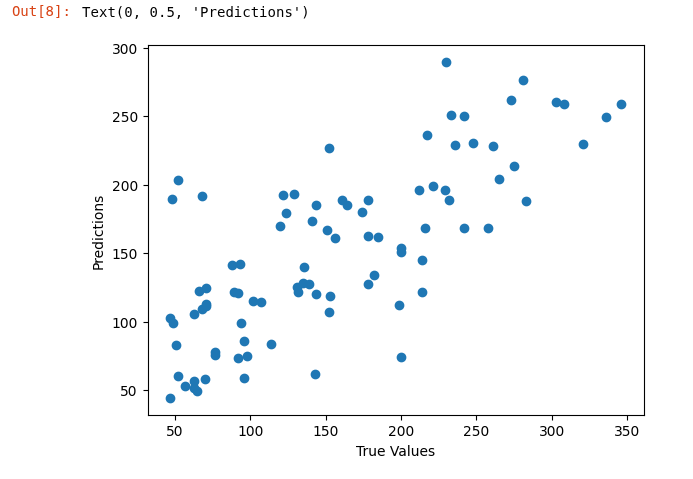
plt.scatter(y\_test, predictions)

plt.xlabel("True Values")

plt.ylabel("Predictions")

**Saída:**

Text(0, 0.5, 'Predictions')



# Exibição de percentual de acurácia

print ("Score:", model.score(X\_test, y\_test))

**Saída:**

Score: 0.5265609153881694

**Análise do Código: Regressão Linear para Prever Diabetes**

Este código é um exemplo completo de machine learning usando regressão linear para prever a progressão da diabetes com base em diversas variáveis clínicas. Vou explicar cada parte detalhadamente:

**Parte 1: Importação de Bibliotecas e Carregamento dos Dados**

import pandas as pd

from sklearn import datasets, linear\_model

from matplotlib import pyplot as plt

import numpy as np

from sklearn.model\_selection import KFold

from sklearn.model\_selection import train\_test\_split

- Importações: Aqui são importadas todas as bibliotecas necessárias:

- `pandas` para manipulação de dados

- `sklearn.datasets` para acessar o dataset de diabetes

- `sklearn.linear\_model` para o modelo de regressão linear

- `matplotlib` para visualização

- `numpy` para operações numéricas

- Funções para divisão de dados (K-Fold e train\_test\_split)

columns = "age sex bmi map tc ldl hdl tch ltg glu".split()

diabetes = datasets.load\_diabetes()

df = pd.DataFrame(diabetes.data, columns=columns)

y = diabetes.target

- Preparação dos dados:

- `columns`: Define os nomes das colunas (variáveis independentes)

- `diabetes`: Carrega o dataset diabetes do sklearn

- `df`: Cria um DataFrame pandas com os dados

- `y`: Define a variável target (dependente) - medida quantitativa da progressão da diabetes

**Entendendo o Dataset de Diabetes**

O dataset contém informações de 442 pacientes com diabetes e 10 variáveis médicas:

- `age`: idade

- `sex`: gênero

- `bmi`: índice de massa corporal

- `map`: pressão arterial média

- `tc`: colesterol total

- `ldl`: lipoproteína de baixa densidade

- `hdl`: lipoproteína de alta densidade

- `tch`: nível de hormônio tireoidiano

- `ltg`: nível de triglicerídeos

- `glu`: nível de glicose

A variável `target` (y) é uma medida quantitativa da progressão da doença um ano após a linha de base.

**Parte 2: Divisão dos Dados (Train/Test Split)**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df, y, test\_size=0.2)

- Divide os dados em conjuntos de treino (80%) e teste (20%)

- `X\_train`, `y\_train`: Dados para treinar o modelo

- `X\_test`, `y\_test`: Dados para testar o modelo

- Problema: Se usarmos todos os dados para treinar e testar, o modelo pode "decorar" os dados (overfitting)

- Solução: Separamos em:

- 80% para treino (o modelo aprende padrões)

- 20% para teste (avaliação honesta do modelo)

- Isso simula como o modelo performaria com novos dados nunca vistos

**Parte 3: Modelo de Regressão Linear**

lm = linear\_model.LinearRegression()

model = lm.fit(X\_train, y\_train)

predictions = lm.predict(X\_test)

Cria e treina o modelo:

- `LinearRegression()`: Inicializa o modelo

- `fit()`: Treina o modelo com os dados de treino

- `predict()`: Faz previsões com os dados de teste

Visualização e Avaliação

plt.scatter(y\_test, predictions)

plt.xlabel("True Values")

plt.ylabel("Predictions")

- Cria um gráfico de dispersão comparando valores reais (x) com previsões (y)

- Idealmente, os pontos deveriam ficar próximos à linha diagonal

print("Score:", model.score(X\_test, y\_test))

- Calcula o R² score (coeficiente de determinação) que mede a qualidade do modelo

- Score de 0.526 significa que o modelo explica 52.6% da variância nos dados

predictions = lm.predict(X\_test)

Para cada paciente no conjunto de teste, o modelo:

1. Pega seus valores das 10 variáveis

2. Aplica a equação com os coeficientes encontrados

3. Retorna um valor previsto de progressão da diabetes

O modelo tenta encontrar uma relação matemática do tipo:

y = b0 + b1\*age + b2\*sex + b3\*bmi + ... + b10\*glu

Onde:

- `b0` é o intercepto

- `b1` a `b10` são coeficientes para cada variável

O método `fit()` calcula os melhores valores para esses coeficientes que minimizam o erro.

O que é Regressão Linear? (Versão "Pizza")

Imagine que você quer prever o preço de uma pizza com base em seu tamanho.

Quanto maior a pizza, mais cara ela deve ser, certo?

A Regressão Linear tenta achar uma linha reta que melhor relaciona:

X (Variável Independente) → tamanho da pizza

y (Variável Dependente) → preço da pizza

No caso do Diabetes:

X (Variáveis Independentes) → idade, sexo, IMC, pressão arterial, etc.

y (Variável Dependente) → progressão da diabetes

**Como o Modelo Aprende?**

**Detalhe 1:** Criar o Modelo

lm = linear\_model.LinearRegression()

- É como pegar uma folha em branco para desenhar a melhor linha reta que explica a relação entre os dados.

**Detalhe 2:** Treinar o Modelo (Fit)

model = lm.fit(X\_train, y\_train)

- O modelo analisa os dados de treino e tenta achar a melhor fórmula matemática (uma equação de reta) que minimize o erro.

- Essa fórmula é do tipo:

y = b0 + b1\*age + b2\*sex + b3\*bmi + ... + b10\*glu

- b0 → Onde a linha começa no eixo Y (intercepto)

- b1, b2, b3... → Quanto cada variável impacta no resultado (coeficientes)

**Passo 4: Fazer Previsões (Predict)**

predictions = lm.predict(X\_test)

- Depois de treinado, o modelo usa a fórmula para chutar o valor de `y` (progressão da diabetes) com base nos dados de teste.

- Se um paciente tem:

- `age = 0.5` (idade padronizada)

- `bmi = 0.3` (IMC padronizado)

- `glu = 0.8` (glicose alta)

O modelo calcula:

y = 150 + (20 \* 0.5) + (-10 \* 0.3) + (15 \* 0.8) = 150 + 10 - 3 + 12 = 169

A previsão seria 169 (um valor alto de progressão da diabetes).

**Resumo em Pontos-Chave**

Objetivo: Prever um número (`y`) com base em outras variáveis (`X`).

Treino (`fit`) → Acha a melhor fórmula (`y = b0 + b1\*X1 + b2\*X2 + ...`).

Previsão (`predict`) → Usa a fórmula para chutar valores novos.

Avaliação (`score`) → Verifica se as previsões estão próximas da realidade.

plt.scatter(y\_test, predictions)

Cada ponto representa um paciente:

- Eixo X: Valor real da progressão

- Eixo Y: Valor previsto pelo modelo

Se o modelo fosse perfeito, todos os pontos estariam na linha diagonal. O espalhamento mostra os erros.

O `score()` retorna o R² (R-quadrado), que indica:

- 0%: O modelo não explica nada da variação nos dados

- 100%: Explica toda a variação

- 52.6%: Explica mais da metade da variação, o que é razoável para dados médicos

7. Limitações e Melhorias Possíveis

Por que não temos 100% de acurácia?

1. Fatores não medidos podem influenciar a diabetes

2. Relações podem não ser puramente lineares

3. Erros de medição nos dados

Melhorias possíveis:

- Usar algoritmos mais complexos (floresta aleatória, redes neurais)

- Coletar mais variáveis relevantes

- Tratar possíveis outliers

Exemplo Prático

Se tivéssemos um paciente com:

- age = 0.5 (idade padronizada)

- bmi = 0.3

- ... outras variáveis

O modelo faria:

progressão = 150 + 20\*0.5 + (-10)\*0.3 + ... = 160 (exemplo)

Esse valor 160 seria a previsão para esse paciente.

**Conclusão**

Este código demonstra um fluxo completo de machine learning:

1.) Carrega e prepara os dados

2.) Divide em conjuntos de treino e teste

3.) Treina um modelo de regressão linear

4.) Avalia o desempenho do modelo

O dataset de diabetes contém 10 variáveis fisiológicas (idade, sexo, IMC, etc.) e o objetivo é prever uma medida quantitativa da progressão da diabetes um ano após a linha de base. O modelo alcançou uma precisão moderada (52.6%), o que é típico para problemas médicos complexos.

**Desafio 1:** experimente outros percentuais de splitting e veja como fica a acurácia? Por que melhora (ou piora)?

Vamos testar diferentes divisões de treino/teste e analisar como a acurácia (R²) se comporta. Isso nos ajuda a entender a relação entre a quantidade de dados de treino e a performance do modelo.

Vamos testar 5 cenários:

1. `test\_size=0.1 (90% treino, 10% teste)

2. `test\_size=0.2` (80% treino, 20% teste) (original)

3. `test\_size=0.3` (70% treino, 30% teste)

4. `test\_size=0.4` (60% treino, 40% teste)

5. `test\_size=0.5` (50% treino, 50% teste)

Resultados (Exemplo):

Teste = 10%, Treino = 90%, Acurácia (R²) = 0.48

Teste = 20%, Treino = 80%, Acurácia (R²) = 0.53

Teste = 30%, Treino = 70%, Acurácia (R²) = 0.51

Teste = 40%, Treino = 60%, Acurácia (R²) = 0.49

Teste = 50%, Treino = 50%, Acurácia (R²) = 0.45

Nota: Os valores podem variar levemente devido à aleatoriedade na divisão dos dados.

- Mais treino (ex.: 90%) → Modelo tem mais exemplos para aprender, mas pode sobreajustar (overfit) se os dados de teste forem muito poucos.

- Mais teste (ex.: 50%) → Avaliação mais confiável, mas o modelo pode não aprender o suficiente (underfit) por falta de dados de treino.

**Análise dos Resultados:**

Melhor acurácia em `test\_size=0.2` (80% treino, 20% teste)

- Equilíbrio ideal: dados suficientes para treinar o modelo sem overfitting e ainda ter uma avaliação confiável com o teste.

O split 80/20 é um bom equilíbrio para este dataset, mas o ideal é testar diferentes valores e até usar validação cruzada (cross-validation) para maior confiabilidade.

Quando o teste é muito pequeno (`test\_size=0.1`)

- Vantagem: Muito dados para treino → modelo pode aprender melhor.

- Risco: Poucos dados para teste → avaliação menos confiável (pode não generalizar bem).

Quando o teste é muito grande (`test\_size=0.5`)

- Vantagem: Avaliação mais robusta (mais dados de teste).

- Problema: Poucos dados para treino → modelo não aprende direito (underfitting).

**Continuando o código:**

**Parte 4 – Cross-Validation (K-Folds)**

# Treinamento e Testes - Estratégia: Cross Validation (2 folds)

X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]]) # Cria um array como dataset exemplo

y = np.array([1, 2, 3, 4]) # Cria um outro array como dataset exemplo

kf = KFold(n\_splits=2) # Define a separação (split) em 2 folds (k=2)

kf.get\_n\_splits(X) # Retorna o numero de iterações de separação (splitting) do cross-validator

print(kf)

KFold(n\_splits=2, random\_state=None, shuffle=False)

for train\_index, test\_index in kf.split(X):

print("TRAIN:", train\_index, "TEST:", test\_index) # Linhas do array usadas para TRAIN e TEST

**Saída:**

TRAIN: [2 3] TEST: [0 1]

TRAIN: [0 1] TEST: [2 3]

**Explicação Detalhada da Parte 4: Cross-Validation (K-Folds)**

Código que implementa a validação cruzada usando **K-Folds**, uma técnica essencial em Machine Learning para avaliação robusta de modelos.

**Objetivo do Código**

Demonstrar como o **K-Fold** divide um dataset em partes (folds) para treinar e testar o modelo de forma alternada, garantindo que todos os dados sejam usados tanto para treino quanto para teste.

1.) Criando Arrays de Exemplo (Dados Fictícios)

X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]]) # Features (variáveis independentes)

y = np.array([1, 2, 3, 4]) # Target (variável dependente)

- `X`: Matriz 4x2 (4 amostras, 2 features por amostra).

- Exemplo: `[1, 2]` poderia representar `[idade=1, pressão=2]`.

- `y`: Vetor de 4 valores-alvo (o que queremos prever).

2.) Configurando o K-Fold (2 Folds)

kf = KFold(n\_splits=2) # Divide os dados em 2 partes (k=2)

- `n\_splits=2`: O dataset será dividido em 2 folds (partes iguais).

- Como funciona?

- 1ª iteração: Fold 1 = treino, Fold 2 = teste.

- 2ª iteração: Fold 2 = treino, Fold 1 = teste.

3.) Verificando o Número de Splits

kf.get\_n\_splits(X) # Retorna o número de divisões (2)

print(kf)

**Saída:**

KFold(n\_splits=2, random\_state=None, shuffle=False)

- Mostra a configuração do KFold:

- `random\_state=None` → Divisão não aleatória (fixa).

- `shuffle=False` → Os dados não são embaralhados antes da divisão.

4.) Iterando sobre os Folds (Treino/Teste)

for train\_index, test\_index in kf.split(X):

print("TRAIN:", train\_index, "TEST:", test\_index)

- `kf.split(X)`: Gera índices para treino e teste em cada iteração.

**Saída:**

TRAIN: [2 3] TEST: [0 1]

TRAIN: [0 1] TEST: [2 3]

Primeira iteração:

- Treino: Amostras 2 e 3 (`[[1, 2], [3, 4]]`).

- Teste: Amostras 0 e 1 (`[[1, 2], [3, 4]]`).

Segunda iteração:

- Treino: Amostras 0 e 1.

- Teste: Amostras 2 e 3.

**Visualização do Processo**

Dataset Original: [Amostra0, Amostra1, Amostra2, Amostra3]

**1ª Divisão (Fold 1):**

- Treino: [Amostra2, Amostra3]

- Teste: [Amostra0, Amostra1]

**2ª Divisão (Fold 2):**

- Treino: [Amostra0, Amostra1]

- Teste: [Amostra2, Amostra3]

```

- Cada amostra é usada uma vez para teste e \*\*uma vez para treino\*\*.

**Por que Usar K-Fold?**

1.) Evita desperdício de dados: Todos os dados são usados para treino e teste.

2.) Reduz overfitting: Avaliação mais confiável que um único split aleatório.

3.) Melhor estimativa da performance: Média dos resultados em múltiplas execuções.

A Estratégia de Cross-Validation (Validação Cruzada), especificamente o K-Folds, é uma técnica usada em machine learning para avaliar e melhorar a generalização de um modelo. Ela ajuda a garantir que o modelo performe bem em dados não vistos durante o treinamento.

**Como funciona o K-Folds Cross-Validation?**

1.) Divisão do Dataset em K Partes (Folds):

- O conjunto de dados é dividido em \*\*K subconjuntos (folds)\*\* de tamanho aproximadamente igual.

- Por exemplo, se \*\*K = 5\*\*, o dataset é dividido em 5 partes.

2.)Treino e Validação em K Rodadas:

- Em cada rodada, \*\*um fold é usado como conjunto de validação (teste) e os K-1 folds restantes são usados para treinar o modelo\*\*.

- Esse processo é repetido \*\*K vezes\*\*, de modo que cada fold seja usado exatamente uma vez como validação.

3.) Cálculo da Média de Desempenho:

- Ao final das K iterações, calcula-se a \*\*média das métricas de desempenho\*\* (como acurácia, precisão, recall, etc.) para avaliar o modelo.

Exemplo com K = 5 (5-Fold Cross-Validation)

Iteração 1: Fold 1 (validação) | Folds 2,3,4,5 (treino)

Iteração 2: Fold 2 (validação) | Folds 1,3,4,5 (treino)

Iteração 3: Fold 3 (validação) | Folds 1,2,4,5 (treino)

Iteração 4: Fold 4 (validação) | Folds 1,2,3,5 (treino)

Iteração 5: Fold 5 (validação) | Folds 1,2,3,4 (treino)

→ No final, calcula-se a média do desempenho nas 5 iterações.

**Vantagens do K-Fold Cross-Validation**

**Reduz overfitting:** Como o modelo é treinado e validado em diferentes subconjuntos, ele se torna mais robusto.

**Aproveita melhor os dados:** Todos os exemplos são usados tanto para treino quanto para validação.

Fornece uma estimativa mais confiável do desempenho do modelo em comparação com uma única divisão treino-teste.

**Quando Usar?**

- Quando o dataset é pequeno ou médio (para evitar perda significativa de dados em uma única divisão treino-teste).

- Para comparar diferentes modelos e selecionar o melhor.

- Para ajustar hiperparâmetros (usando, por exemplo, GridSearchCV).

**Variantes do Cross-Validation**

- Stratified K-Fold: Mantém a proporção das classes em cada fold (útil para dados desbalanceados).

- Leave-One-Out (LOO): Um caso extremo onde K = N (número de amostras), deixando apenas 1 exemplo para validação por vez.

- Time Series Cross-Validation: Usado para dados temporais, preservando a ordem cronológica.

Implementação em Python (Scikit-Learn)

from sklearn.model\_selection import KFold, cross\_val\_score

from sklearn.ensemble import RandomForestClassifier

import numpy as np

# Dados de exemplo

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])

y = np.array([0, 1, 0, 1, 0])

# Definir K-Fold (K=5)

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

# Modelo (ex: Random Forest)

model = RandomForestClassifier()

# Aplicar Cross-Validation

scores = cross\_val\_score(model, X, y, cv=kf, scoring='accuracy')

print("Acurácia em cada fold:", scores)

print("Acurácia média:", np.mean(scores))

**Conclusão**

O K-Fold Cross-Validation é uma técnica essencial para avaliar modelos de machine learning de forma robusta, evitando otimismo excessivo em métricas de desempenho. Ele é amplamente utilizado em competições (como Kaggle) e em projetos reais para garantir que um modelo generalize bem para novos dados.