Lista 01 - Machine Learning
IMPA

Pedro Bahia

23 de janeiro de 2026

Conteúdo

1 Exercício 1a

Falso. A proximidade entre $\varepsilon_{\text{treino}}$ e $\varepsilon_{\text{teste}}$ pode dar informações sobre o ajuste do modelo aos dados de treino.

Caso $\varepsilon_{\text{treino}}$ seja próximo ao $\varepsilon_{\text{teste}}$, o modelo pode estar *subajustado*, de modo que aumentar a complexidade poderia melhorar sua performance ainda mais.

De maneira análoga, caso o $\varepsilon_{\text{treino}}$ seja menor que o $\varepsilon_{\text{teste}}$, o modelo estará *sobreajustado*, com redução de complexidade podendo resultar em melhoras.

2 Exercício 1b

Verdadeiro. A distribuição t surge do fato de $\mathcal{N}(0,1)$ dividido por $\sqrt{\frac{K}{N}}$ ter distribuição t com N graus de liberdade.

No nosso caso, $\mathcal{N}(0,1)$ é a distribuição de $\frac{\hat{\beta_1}-\beta_1}{\text{Var}(\hat{\beta_1})}$. Isso é normal, pois $\hat{\beta}_1$ é normal.

Isso, por sua vez, vem do fato de $\hat{\beta}$ ser resultante de uma combinação linear de gaussianas, no caso, $\varepsilon \sim \mathcal{N}(0, \sigma^2)$.

3 Exercício 1c

Falso. Considerando a classe $k = 0$ como as transações fraudulentas, o objetivo do modelo pode ser interpretado como:

$$\sum_{y_i \in k=0} \mathbf{1}_{[y_i \neq \hat{y}_i]} = 0$$

Não há restrições entretanto em relação às transações legítimas, ou seja, para:

$$\sum_{y_i \in k=1} \mathbf{1}_{[y_i \neq \hat{y}_i]}$$

Dado um modelo de acurácia $(1 - \varepsilon)$, têm-se que

$$1 - \frac{1}{n} \left[\sum_{y_i \in k=0} \mathbf{1}_{[y_i \neq \hat{y}_i]} + \sum_{y_i \in k=1} \mathbf{1}_{[y_i \neq \hat{y}_i]} \right] = 1 - \varepsilon$$

Dado uma acurácia

$$1 - epsilon$$

, há infinitos valores de $\sum_{y_i \in k=0} \mathbf{1}_{[y_i \neq \hat{y}_i]}$ e $\sum_{y_i \in k=1} \mathbf{1}_{[y_i \neq \hat{y}_i]}$ que resolvem essa equação e, portanto, o valor da acurácia não é informativo para o erro individual das classes. Assim, apenas com a acurácias dos Modelos 1 e 2 não é possível determinar qual modelo tem menor erro em transações fraudulentas.

4 Exercício 1d

Verdadeiro

$$L_{ridge}(\beta) = (Y - Yhat)^T(Y - \hat{Y}) + \lambda\beta^T\beta$$
$$L_{ridge}(\beta) = (Y - X\beta)^T(Y - X\beta) + \lambda\beta^T\beta$$

$$L_{linear}(\beta) = (Y - Yhat)^T(Y - \hat{Y})$$

Caso $\lambda = 0$, temos que $L_{ridge}(\beta) = L_{linear}(\beta)$. Logo, a performance de ambos os modelos será a mesma. Então a afirmação será verdadeira para o case de $\lambda = 0$,

5 Exercício 1e

Verdadeira

A equação dada pela fórmula do intervalo de confiança:

$$\left[\hat{\beta}_j - 2\sqrt{\hat{\sigma}^2[(X^TX)^{-1}]_{jj}}, \hat{\beta}_j + 2\sqrt{\hat{\sigma}^2[(X^TX)^{-1}]_{jj}}\right]$$

é derivada do fato de $\hat{\beta}$ ter uma distribuição normal. Isso vem do fato da hipótese de que o erro é normal.

Caso ela não seja feita, os intervalos de confiança gerados via *bootstrap* são mais adequados, pois são derivados a partir da distribuição inferida diretamente dos dados. Neste caso, a hipótese não-paramétrica é mais geral e preferível.

Justificativa:

- **Abordagem paramétrica**: Assume que os erros seguem distribuição normal, permitindo o uso da distribuição t de Student para construir intervalos de confiança analíticos.
- **Abordagem não-paramétrica (bootstrap)**: Não assume distribuição específica dos erros, utilizando reamostragem dos dados para estimar a distribuição empírica dos parâmetros.
- **Vantagem do bootstrap**: Mais robusto quando as suposições paramétricas são violadas, especialmente em casos de não-normalidade dos erros.

6 Exercício 2a

Dado que a variância de ε é:

$$\Sigma = \begin{pmatrix} \text{Cov}(\varepsilon_1, \varepsilon_1) & \text{Cov}(\varepsilon_1, \varepsilon_2) & \cdots & \text{Cov}(\varepsilon_1, \varepsilon_n) \\ \text{Cov}(\varepsilon_2, \varepsilon_1) & \text{Cov}(\varepsilon_2, \varepsilon_2) & \cdots & \text{Cov}(\varepsilon_2, \varepsilon_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(\varepsilon_n, \varepsilon_1) & \text{Cov}(\varepsilon_n, \varepsilon_2) & \cdots & \text{Cov}(\varepsilon_n, \varepsilon_n) \end{pmatrix}$$

Tal que $\text{Cov}(\varepsilon_i, \varepsilon_j) = \mathbb{E}[(\varepsilon_i - \mu_i)(\varepsilon_j - \mu_j)]$.

Assumir a independência dos erros implica que $\text{Cov}(\varepsilon_i, \varepsilon_j) = 0$ para $i \neq j$. Isso resulta em uma matriz de covariância diagonal, os elementos fora da diagonal são todos zero.

Já assumir a homocedasticidade implica que a variância dos erros é constante, ou seja,

$$\text{Var}(\varepsilon_i) = \text{Var}(\varepsilon_j) = \sigma^2, paratodoi, j$$

Ou seja, $\text{Var}(\varepsilon_i) = \sigma^2$ para todo i. Assim, os elementos na diagonal da matriz de covariância são todos iguais a σ^2.

Portanto, sob as suposições de independência e homocedasticidade dos erros, a matriz de covariância Σ assume a forma:

$$\Sigma = \begin{pmatrix} \sigma^2 & 0 & \cdots & 0 \\ 0 & \sigma^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma^2 \end{pmatrix} = \sigma^2 I_n$$

onde I_n é a matriz identidade de ordem n.

7 Exercício 2b

7.1 i

A função de perda ponderada pela matriz de covariância dos erros é dada por:

$$\hat{\beta}_\Sigma = \arg\min_\beta (Y - X\beta)^T \Sigma^{-1} (Y - X\beta)$$

Esta função é convexa em relação a β, pois, sendo Σ^{-1} é uma matriz positiva definida e a função quadrática $(Y - X\beta)^T (Y - X\beta)$ estritamente convexa, seu produto também é estritamente convexo.

Assim, para encontrar o estimador $\hat{\beta}_\Sigma$, derivamos a função de perda em relação a β e igualamos a zero a derivada:

$$\frac{d\hat{\beta}_\Sigma}{d\beta}(Y - X\beta)^T \Sigma^{-1} (Y - X\beta) = [-2X^T \Sigma^{-1} (Y - X\beta)] = 0$$

Resolvendo para β, obtemos:

$$X^T \Sigma^{-1} Y = X^T \Sigma^{-1} X\beta$$

$$\hat{\beta}_\Sigma = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y$$

7.2 ii

Como $(X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1}$ é uma constante em relação a Y, e sabemos que $Y = X\beta + \varepsilon$, onde $\mathbb{E}[\varepsilon] = 0$ e $\mathbb{E}[Y] = X\beta$, temos:

$$\mathbb{E}[\hat{\beta}_\Sigma] = \mathbb{E}[(X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y]$$

$$\mathbb{E}[\hat{\beta}_\Sigma] = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} \mathbb{E}[Y]$$

$$\mathbb{E}[\hat{\beta}_\Sigma] = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} X\beta$$

$$\mathbb{E}[\hat{\beta}_\Sigma] = \beta$$

7.3 iii

$(X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1}$ é uma constante em relação a Y e pode ser fatorado para fora da operação de variância como sua trasnposta multiplicando pela direita. Além disso, sabemos que $\mathbb{V}(Y) = \varepsilon = \Sigma$, temos:

$$\mathbb{V}(\hat{\beta}_\Sigma) = \mathbb{V}((X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y)$$

$$\mathbb{V}(\hat{\beta}_\Sigma) = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} \mathbb{V}(Y) \Sigma^{-1} X (X^T \Sigma^{-1} X)^{-1}$$

$$\mathbb{V}(\hat{\beta}_\Sigma) = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} \Sigma \Sigma^{-1} X (X^T \Sigma^{-1} X)^{-1}$$

$$\mathbb{V}(\hat{\beta}_\Sigma) = (X^T \Sigma^{-1} X)^{-1}$$

7.4 iv

Sendo o ε uma variável aleatória distribuída $\varepsilon \sim \mathcal{N}(0, \Sigma)$, Y uma i.i.d com $Y \sim \mathcal{N}(X\beta, \Sigma)$ e Σ e X fixos, o estimador $\hat{\beta}_\Sigma$ é uma combinação linear de variáveis aleatórias normais. Portanto, $\hat{\beta}_\Sigma$ também é uma variável aleatória normal. Assim, temos que:

$$\hat{\beta}_\Sigma \sim \mathcal{N}(\mathbb{E}[\hat{\beta}_\Sigma], \mathbb{V}(\hat{\beta}_\Sigma))$$

$$\hat{\beta}_\Sigma \sim \mathcal{N}(\beta, (X^T \Sigma^{-1} X)^{-1})$$

8 Exercício 2c

8.1 i

Como Σ é uma matriz diagonal e positiva definida, sua inversa Σ^{-1} também será uma matriz diagonal, cujo elementos são os inversos dos elementos diagonais de Σ. Sua fatoração $\Sigma^{-1/2}$ também será uma matriz diagonal, cujos elementos são as raizes quadradas dos elementos de Σ^{-1}. Por fim, a transposta de $\Sigma^{-1/2}$ será igual a $\Sigma^{-1/2}$.

Assim, podemos reescrever a função de perda ponderada como:

$$\begin{aligned}\mathcal{L} &= (y - X\beta)^T \Sigma^{-1} (y - X\beta) \\ &= (y - X\beta)^T \Sigma^{-1/2} \Sigma^{-1/2} (y - X\beta) \\ &= (\Sigma^{-1/2} y - \Sigma^{-1/2} X\beta)^T (\Sigma^{-1/2} y - \Sigma^{-1/2} X\beta)\end{aligned}$$

Novamente, como $\Sigma^{-1/2}$ é uma matriz diagonal, cada elemento é dos vetores é multiplicado pelo respectivo elemento diagonal de $\Sigma^{-1/2}$. Enfim, expandindo a soma temos:

$$\begin{aligned}\mathcal{L} &= \sum_{i=1}^{n} (\frac{1}{\sigma_{ii}} y_i - \frac{1}{\sigma_{ii}} X_i\beta)^2 \\ &= \sum_{i=1}^{n} \frac{1}{\sigma_{ii}^2} (y_i - X_i\beta)^2 \\ &= \sum_{i=1}^{n} w_i^2 (y_i - X_i\beta)^2 \quad \text{onde } w_i = \frac{1}{\sigma_{ii}} \\ &= \sum_{i=1}^{n} w_i^2 (y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j)^2\end{aligned}$$

8.2 ii

Os pesos da função de perda $w_i = \frac{1}{\sqrt{\Sigma_{ii}}}$ ponderam a amostra i pelo inverso da variância do erro, de modo a normalizar as contribuições das amostras para a função de perda total. Assim, amostras com maior variância, ou seja, com maior incerteza e dificuldade de ajuste, terão menor impacto na função de perda, enquanto amostras com menor variância terão mais.

8.3 iii

De modo análogo à seção i), dado que Σ é positiva definida, podemos reescrever a função de perda ponderada pela matriz de covariância dos erros

$$\mathcal{L} = (\Sigma^{-1/2}y - \Sigma^{-1/2}X\beta)^T(\Sigma^{-1/2}y - \Sigma^{-1/2}X\beta) \quad = (\tilde{Y} - \tilde{X}\beta)^T(\tilde{Y} - \tilde{X}\beta)$$

onde $\tilde{Y} = \Sigma^{-1/2}y$ e $\tilde{X} = \Sigma^{-1/2}X$.

Entretanto, como os elementos fora das diagonais não são nulos, o somatório é expandido como

$$\mathcal{L} = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(y_i - X_i\beta)^2 \qquad = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(y_i - \beta_0 - \sum_{k=1}^{p} X_{ik}\beta_k)^2$$

9 Exercício 2d

9.1 i

Os dados com heterocedasticidade usando a matriz de covariância Σ diagonal:

```
import numpy as np
import matplotlib.pyplot as plt

n = 50
Sigma = np.diag([10 ** ((i - 20) / 5) for i in range(1, n + 1)])
np.random.seed(0)
X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
beta = np.array([1, 0.25])
epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
y = X @ beta + epsilon
```

A Figura 1 mostra os dados gerados com heterocedasticidade:

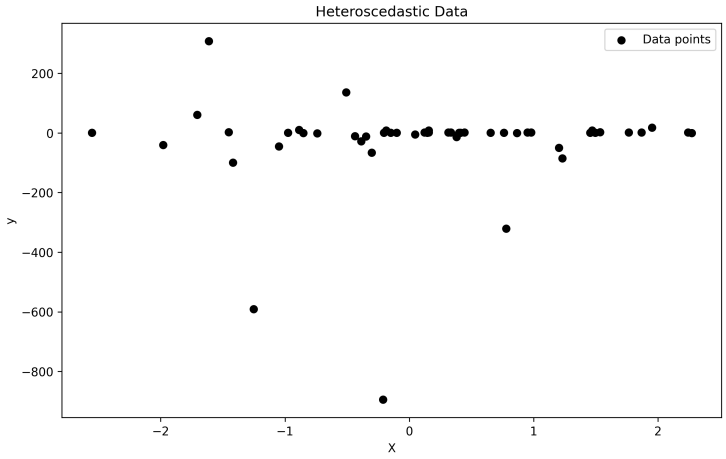

Figura 1: Dados heteroscedásticos gerados

A Figura 2 mostra os elementos diagonais da matriz Σ, evidenciando a heterocedasticidade:

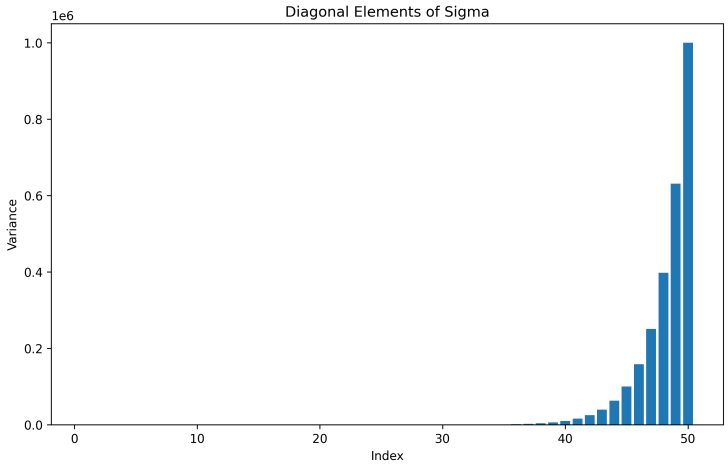

Figura 2: Elementos diagonais da matriz Σ

As Figuras 3 e 4 mostram a distribuição e valores dos erros:

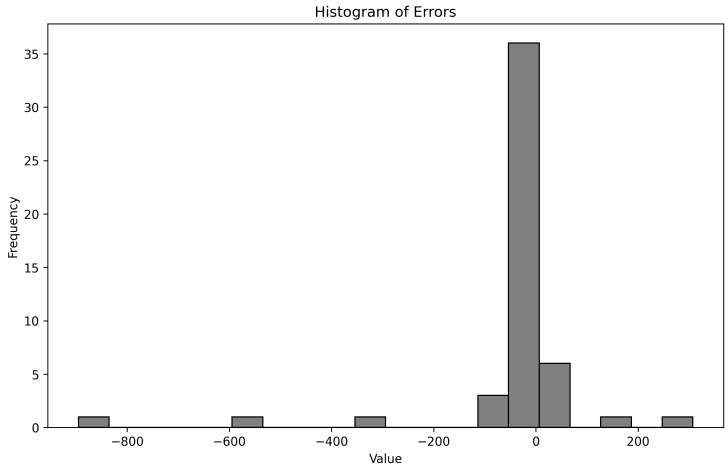

Figura 3: Histograma dos erros

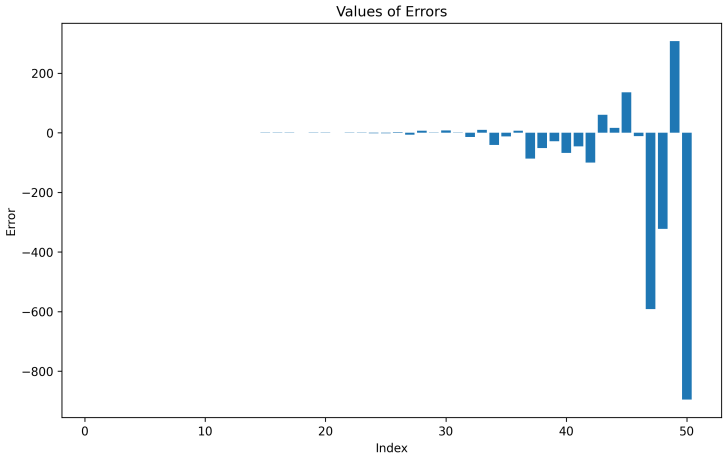

Figura 4: Valores dos erros por índice

9.2 ii

Tanto estimador de mínimos quadrados ordinários quanto o estimador generalizado que considera a matriz de covariância Σ foram implementados com o código à seguir:

```
def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
    ndarray:
    """
    Compute the ordinary least squares estimator.
    """
    beta = np.linalg.inv(X.T @ X) @ X.T @ Y
    return beta

def beta_sigma(X: np.ndarray, Y: np.ndarray, Sigma: np.
    ndarray) -> np.ndarray:
    """
    Compute the generalized least squares estimator
    considering
    the covariance matrix Sigma.
    """
    Sigma_1 = np.linalg.inv(Sigma)
    beta = np.linalg.inv(X.T @ Sigma_1 @ X) @ X.T @ Sigma_1
    @ Y
    return beta

beta_hat_ordinary = beta_ordinary(X, y)
beta_hat_sigma = beta_sigma(X, y, Sigma)
```

Comparação dos Estimadores:

- Parâmetros Verdadeiros: $\beta = [1.0, 0.25]$

- Estimador Ordinário: $\hat{\beta}_{OLS} = [-34.463, 6.948]$
- Estimador Generalizado: $\hat{\beta}_{\Sigma} = [1.019, 0.244]$

Erro Quadrático dos Estimadores:

- $\|\beta - \hat{\beta}_{OLS}\|_2^2 = 1302.51$
- $\|\beta - \hat{\beta}_{\Sigma}\|_2^2 = 0.000387$

9.3 iii

```python
def p_value_ordinary_least_square(X: np.ndarray, Y: np.ndarray,
                                  beta_ordinary_hat: np.ndarray, j: int) -> float:
    """
    Compute the p-value for the j-th coefficient of the ordinary
    least squares estimator.
    """
    Y_hat = X @ beta_ordinary_hat
    n, p = X.shape
    dof = n - p
    errors = Y - Y_hat
    beta_j = beta_ordinary_hat[j]

    # Z statistic
    x_j_var = (np.linalg.inv(X.T @ X))[j, j]
    Z = beta_j / np.sqrt(x_j_var)

    # Estimate of sigma^2
    sigma2_hat = (1 / dof) * (errors.T @ errors)

    # t statistic and p-value
    t_statistics = Z / np.sqrt(sigma2_hat)
    t_statistics = np.abs(t_statistics)
    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))

    return p_value
```

Testes de Hipótese - Mínimos Quadrados Ordinários:

- p-valor para β_0: 0.1544
- p-valor para β_1: 0.7422

Não podemos descartar a hipótese nula para ambos os coeficientes ao nível de significância de 5%.

9.4 iv

```
def calculate_Z_sigma(X: np.ndarray, Sigma: np.ndarray,
                      Beta_sigma: np.ndarray, j: int) ->
    float:
    """
    Compute the Z statistic for the j-th coefficient of the
    generalized least squares estimator.
    """
    Sigma_inv = np.linalg.inv(Sigma)
    den = np.linalg.inv(X.T @ Sigma_inv @ X)
    den = den[j, j]
    Z = Beta_sigma[j] / (np.sqrt(den))
    return Z
```

- Estatística Z_Σ para β_0: 73.67

9.5 v

Condicionado em X, o termo da diagonal i do denominador de Z é a raiz quadrada da variância do estimador generalizado $\hat{\beta}_\Sigma$.

Variância de $\hat{\beta} = (X^T\Sigma^{-1}X)^{-1} = \begin{pmatrix} \text{Var}(\beta_0) & \text{Cov}(\beta_0, \beta_1) \\ \text{Cov}(\beta_0, \beta_1) & \text{Var}(\beta_1) \end{pmatrix}$

Na hipótese H_0 de $\beta_0 = 0$, temos que $\hat{\beta}_\Sigma \sim \mathcal{N}(\beta_0, \text{Var}(\beta_0))$. Assim, a estatística Z é dada por:

$$Z_\Sigma = \frac{\hat{\beta}_\Sigma}{\sqrt{\text{Var}(\hat{\beta}_\Sigma)}}$$

Para obtermos o p-valor, fazemos $Z/\sqrt{\sigma^2}$, onde $\sigma^2 = \frac{1}{n-p-1}\varepsilon^T\Sigma^{-1}\varepsilon$

$$\begin{aligned}
\text{Var}(\hat{\beta}_\Sigma) &= \text{Var}((X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}Y) \\
&= (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}\text{Var}(Y)\Sigma^{-1}X(X^T\Sigma^{-1}X)^{-1} \\
&= (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}\Sigma\Sigma^{-1}X(X^T\Sigma^{-1}X)^{-1} \\
&= (X^T\Sigma^{-1}X)^{-1}
\end{aligned}$$

Calculando o p-valor com o código abaixo:

```
def p_value_generalized_least_square(X: np.ndarray, Y: np.
    ndarray,
                                      Sigma: np.ndarray,
```

```
                                        beta_ordinary_hat: np.
    ndarray, j: int) -> float:
    """
    Compute the p-value for the j-th coefficient of the
    generalized least squares estimator.
    """
    Y_hat = X @ beta_ordinary_hat
    n, p = X.shape
    dof = n - p
    errors = Y - Y_hat

    # Z statistic
    Z_sigma = calculate_Z_sigma(X, Sigma, beta_hat_sigma, j)

    # Estimate of sigma^2
    inverse_Sigma = np.linalg.inv(Sigma)
    sigma2_hat = (1 / dof) * (errors.T @ inverse_Sigma @
    errors)

    # t statistic and p-value
    t_statistics = Z_sigma / np.sqrt(sigma2_hat)
    t_statistics = np.abs(t_statistics)
    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))

    return p_value
```

Esta implementação permite comparar os dois estimadores e calcular a significância estatística dos coeficientes em ambos os casos.

Testes de Hipótese - Estimador Generalizado:

- p-valor para β_0: 0.0
- p-valor para β_1: 0.0

10 Exercício 3a

$$
\begin{aligned}
Y_i|X_i &\sim \text{Laplace}(\beta^T x_i, b) \\
\epsilon &\sim \text{Laplace}(0, b) \\
Y_i|X_i &= Y = \beta^T X + \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{E}(Y) &= \mathbb{E}(\beta X + \epsilon) = \mathbb{E}(\beta^T X) + \mathbb{E}(\epsilon) \\
&= \beta^T X + 0
\end{aligned}
$$

$$
\begin{aligned}
\text{Var}(Y) &= \text{Var}(\beta^T X + \epsilon) = \text{Var}(\beta^T X) + \text{Var}(\epsilon) \\
&= 0 + b = b
\end{aligned}
$$

Como toda combinação linear de variáveis aleatórias com distribuição de Laplace resulta em uma distribuição de Laplace, Y é uma variável aleatória de Laplace com parâmetros Laplace$(\beta^T X, b)$.

Função de perda (likelihood):

Dado que Y é iid, temos que:

$$
P(Y|X, \beta) = \prod_{i=1}^{n} \frac{1}{2b} \exp\left(-\frac{|Y_i - \beta^T X_i|}{b}\right)
$$

$$
\begin{aligned}
\log P(Y|X, \beta) &= \log \prod_{i=1}^{n} \frac{1}{2b} \exp\left(-\frac{|Y_i - \beta^T X_i|}{b}\right) \\
&= \sum_{i=1}^{n} \log\left(\frac{1}{2b} \exp\left(-\frac{|Y_i - \beta^T X_i|}{b}\right)\right) \\
&= \sum_{i=1}^{n} \left(\log \frac{1}{2b} - \frac{|Y_i - \beta^T X_i|}{b}\right) \\
&= n \log \frac{1}{2b} - \frac{1}{b} \sum_{i=1}^{n} |Y_i - \beta^T X_i|
\end{aligned}
$$

11 Exercício 3b

$$
\begin{aligned}
Y_i|X_i &\sim \text{Laplace}(\beta^T x_i, b) \\
\epsilon &\sim \text{Laplace}(0, b) \\
Y_i|X_i &= Y = \beta^T X + \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{E}(Y) &= \mathbb{E}(\beta X + \epsilon) = \mathbb{E}(\beta^T X) + \mathbb{E}(\epsilon) \\
&= \beta^T X + 0
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{V}(Y) &= \mathbb{V}(\beta X + \epsilon) = \mathbb{V}(\beta X) + \mathbb{V}(\epsilon) \\
&= 0 + b
\end{aligned}
$$

Toda combinação linear de Laplace são
Laplace:

Item b

$$
\begin{aligned}
\mathcal{L}(\beta) &= -\log(\ell(\beta|x, y, b)) \\
&= -\log\left(\frac{1}{2b} \cdot e^{-\sum_i^n \frac{|y_i - x_i\beta|}{b}}\right) \\
&= -\log(2b) + \sum_i^n \frac{|y_i - x_i\beta|}{b} \\
&= -\log(2b) + \frac{1}{b}\sum_i^n |y_i - x_i\beta|
\end{aligned}
$$

- $-\log(2b)$ é constante e pode ser desconsiderado na otimização
- o mesmo pode ser dito sobre o termo que multiplica o somatório

12 Exercício 3c

$\epsilon \overset{iid}{\sim} \mathcal{N}(0,1)$
$Y = X\beta + \epsilon$

$$\ell(y|X\beta,\sigma^2) = \prod_i^n \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{y_i - X_i\beta}{\sigma}\right)^2}$$

$$= \frac{1}{\sigma^n(\sqrt{2\pi})^n} \cdot e^{-\frac{1}{2\sigma^2}\sum(y_i - X\beta)^2}$$

$$\mathcal{L}(y|\mu,\sigma^2) = -\log(\ell(y|\mu,\sigma^2))$$

$$= -\log\left[\frac{1}{\sigma^n(\sqrt{2\pi})^n} \cdot e^{-\frac{1}{2\sigma^2}\sum_i^n(y_i - X_i\beta)^2}\right]$$

$$= \log(\sigma^n(\sqrt{2\pi})^n) + \frac{1}{2\sigma^2}\sum_i^n(y_i - X_i\beta)^2$$

$$\mathcal{L}(y|\mu,\sigma^2) \approx \sum_i^n(y_i - X_i\beta)^2$$

Vantagem: em caso de outliers, erro muito alto, o erro para a amostra i na função (ℓ) será elevado ao quadrado, resultando em maiores impactos na função de perda.
O mesmo não ocorre em b.

13 Exercício 3d

14 Exercício 3e

15 Exercício 3f

15.1 i) Implementação dos Estimadores

Neste exercício, comparamos estimadores baseados em diferentes suposições sobre a distribuição dos erros. Implementamos estimadores que minimizam diferentes funções de perda:

```
import numpy as np
import scipy
import matplotlib.pyplot as plt

def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.ndarray:
    """
    Compute the ordinary least squares estimator.
    """
    beta = np.linalg.inv(X.T @ X) @ X.T @ Y
    return beta

def calculate_beta_hat(X: np.ndarray, Y: np.ndarray, error_distribution: str) -> np.ndarray:
    """
    Compute the estimator beta_hat based on the specified error distribution.
    """
    np.random.seed(0)
    p = X.shape[1]

    if error_distribution == "gaussian":
        def loss_function(beta):
            return np.sum((Y - X @ beta) ** 2)
    elif error_distribution == "laplacian":
        def loss_function(beta):
            return np.sum(np.abs(Y - X @ beta))
    else:
        raise ValueError("Unsupported error distribution")

    beta_0 = np.random.uniform(size=p)
    beta_hat = scipy.optimize.minimize(loss_function, beta_0)
    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

```
np.random.seed(1)
beta = np.array([-1.5, 2.0])
input_range = np.linspace(-1, 1, 100)
X = np.vstack([np.ones(100), input_range]).T
y = X @ beta + np.random.normal(0, 0.3, 100)
```

A Figura 5 mostra os dados gerados:

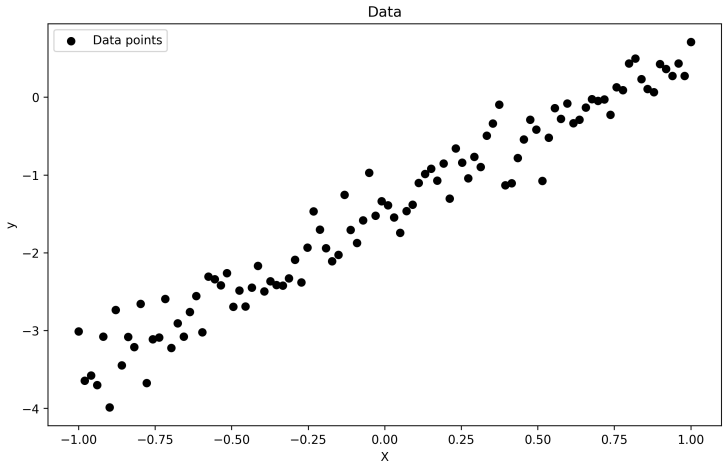

Figura 5: Dados sintéticos para comparação dos estimadores

As Figuras 6 e 7 mostram a análise dos erros:

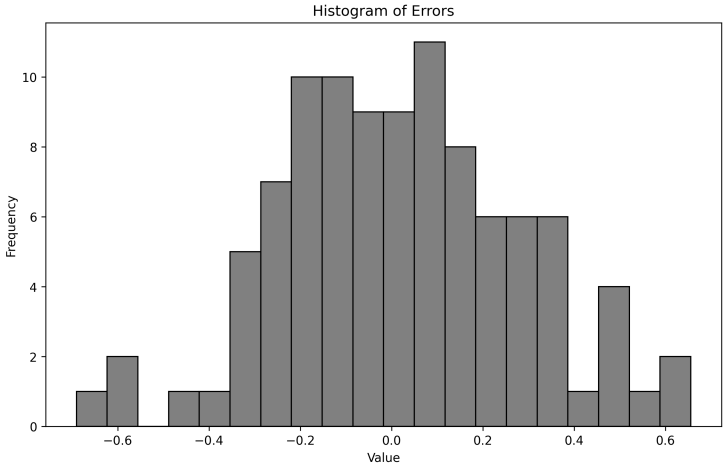

Figura 6: Histograma dos erros

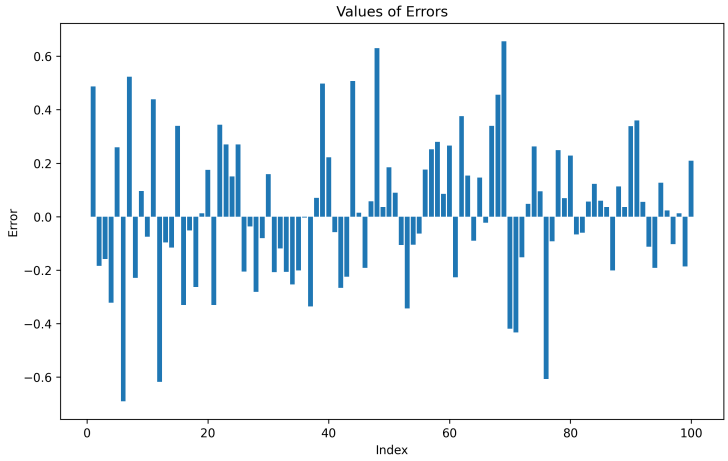

Figura 7: Valores dos erros por índice

15.2 ii) Comparação dos Estimadores

Calculamos os estimadores para ambas as distribuições de erro:

Resultados sem Outliers:

- Parâmetros Verdadeiros: $\beta = [-1.5, 2.0]$
- Estimador Gaussiano (minimize): $\hat{\beta}_{Gauss} = [-1.482, 2.050]$
- Estimador Gaussiano (forma fechada): $\hat{\beta}_{OLS} = [-1.482, 2.050]$
- Estimador Laplaciano: $\hat{\beta}_{Lap} = [-1.498, 2.082]$

Análise de Erro (Norma L2):

- Erro do Estimador Gaussiano: 0.053
- Erro do Estimador Laplaciano: 0.082

Sem outliers, o estimador gaussiano (mínimos quadrados) tem melhor performance, como esperado quando os erros seguem distribuição normal.

A Figura 8 compara visualmente os ajustes:

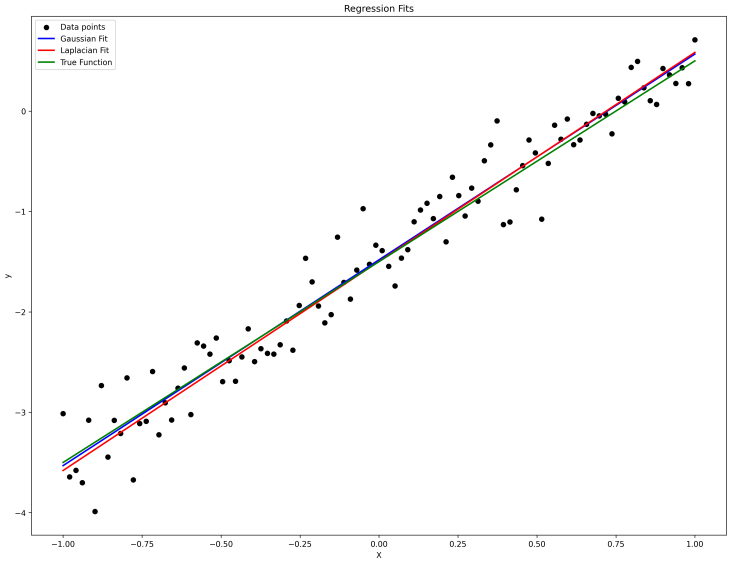

Figura 8: Comparação dos ajustes de regressão sem outliers

15.3 iii) Robustez a Outliers

Para testar a robustez, adicionamos um outlier extremo ao dataset:

```
# Regenerate data and add outlier
y[80] = 10  # Extreme outlier
```

Resultados com Outlier:

- Parâmetros Verdadeiros: $\beta = [-1.5, 2.0]$
- Estimador Gaussiano com outlier: $\hat{\beta}_{Gauss} = [-1.378, 2.237]$
- Estimador Laplaciano com outlier: $\hat{\beta}_{Lap} = [-1.498, 2.083]$

Análise de Erro com Outlier (Norma L2):

- Erro do Estimador Gaussiano: 0.266 (aumento de 5x)
- Erro do Estimador Laplaciano: 0.083 (praticamente inalterado)

A Figura 9 mostra o impacto do outlier:

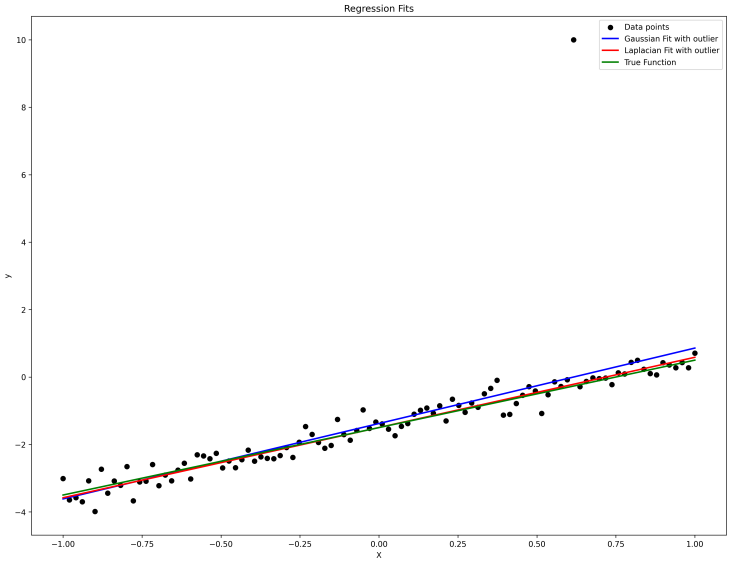

Figura 9: Comparação dos ajustes de regressão com outlier

Conclusões:

1. **Eficiência**: Quando os erros são gaussianos e não há outliers, o estimador de mínimos quadrados (gaussiano) é mais eficiente.

2. **Robustez**: O estimador laplaciano é significativamente mais robusto a outliers, mantendo sua performance praticamente inalterada mesmo com outliers extremos.

3. **Trade-off**: Existe um trade-off entre eficiência (mínimos quadrados) e robustez (estimador laplaciano). A escolha depende das características esperadas dos dados.

4. **Aplicação Prática**: Em situações onde outliers são esperados ou a distribuição dos erros tem caudas pesadas, o estimador laplaciano (regressão com norma L1) é preferível.

5. **Impacto Dramático**: Um único outlier pode degradar significativamente a performance do estimador gaussiano (aumento de 5x no erro), enquanto o estimador laplaciano permanece praticamente inalterado.

16 Exercício 4a

O modelo knn necessida que as features sejam normalizadas pois ele usa uma métrica de distância para encontrar os vizinhos mais próximos. Se as features não forem normalizadas, aquelas com escalas maiores podem dominar a métrica de distância, fazendo com que o algoritmo não funcione corretamente.

17 Exercício 4b

17.1 Implementação dos Algoritmos de Classificação

Neste exercício, implementamos e comparamos cinco diferentes algoritmos de classificação usando o dataset de futebol:

- **LDA** (Linear Discriminant Analysis)
- **QDA** (Quadratic Discriminant Analysis)
- **LR** (Logistic Regression)
- **NB** (Naive Bayes Gaussiano)
- **kNN** (k-Nearest Neighbors)

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.discriminant_analysis import
    LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import
    QuadraticDiscriminantAnalysis as QDA
from sklearn.linear_model import LogisticRegression as LR
from sklearn.naive_bayes import GaussianNB as NB
from sklearn.neighbors import KNeighborsClassifier as kNN
from sklearn import preprocessing

# Load and prepare data
df = pd.read_csv("../data/soccer.csv")
X = df.drop("target", axis=1)
y = df[["target"]]

# Split dataset
X_train, y_train = X.iloc[:2560], y.iloc[:2560]
X_test, y_test = X.iloc[2560:], y.iloc[2560:]

# Remove categorical variables and standardize
X_train = X_train.drop(["home_team", "away_team"], axis=1)
X_test = X_test.drop(["home_team", "away_team"], axis=1)
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Informações do Dataset:

- Amostras de treino: 2560
- Amostras de teste: 640
- Features após pré-processamento: 11 (removendo variáveis categóricas)

17.2 Treinamento e Avaliação dos Modelos

Implementamos um loop para treinar todos os modelos e comparar suas performances:

```
models_to_test = [LDA, QDA, LR, NB, kNN]
results_dict = {}

for model_type in models_to_test:
    model_name = model_type.__name__
    params = {}
    if model_type in [LDA, QDA]:
        params.update({"store_covariance": True})

    results_dict[model_name] = {}
    cls = model_type(**params)
    cls.fit(X_train, y_train.values.ravel())

    # Store predictions and model
    results_dict[model_name]["in_sample_predictions"] = cls.
    predict(X_train)
    results_dict[model_name]["test_predictions"] = cls.
    predict(X_test)
    results_dict[model_name]["model"] = cls
```

Linear Discriminant Analysis (LDA):

- Coeficientes: $[0.81, -0.26, -0.026, 0.017, 0.23, 0.069, 0.37, -0.050, -0.083, 0.043, 0.047]$
- Intercepto: 0.109
- Utiliza covariância comum entre as classes

18 Exercício 4c

A Figura 10 compara os erros de treinamento e teste para todos os modelos:

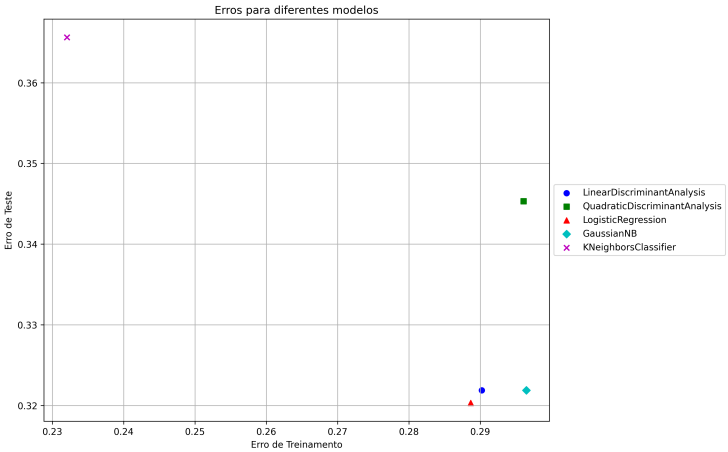

Figura 10: Comparação dos erros de treinamento vs teste para diferentes modelos

19 Exercício 4d

19.1 Análise Específica do k-NN

A Figura 11 mostra como a performance do k-NN varia com diferentes valores de k:

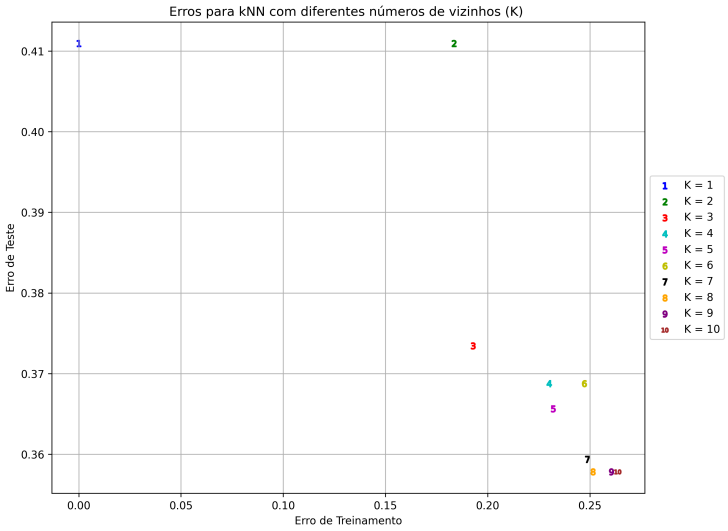

Figura 11: Erros do k-NN para diferentes números de vizinhos (K=1 a K=10)

Para k=1, o modelo apresenta overfitting, com erro de treino nulo já que ele simplesmente repete o dado do vizinho mais próximo, a própria amostra, e erro de teste alto. Conforme k aumenta, o modelo generaliza melhor, aumentando o erro de treino e reduzindo o erro de teste, já que mais vizinhos são considerados na decisão de classe. Para os k analisados, não houve underfitting, mas se k fosse muito grande (próximo ao número total de amostras), o modelo tenderia a classificar todas as amostras na classe majoritária, aumentando ambos os erros.

20 Exercício 5a

```python
import statsmodels.api as sm
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, KFold
from sklearn.linear_model import Lasso
from sklearn import preprocessing

# Load and prepare data
bodyfat = pd.read_csv("../data/bodyfat.csv")
X = bodyfat.drop(columns=["BodyFat", "Density"])
y = bodyfat["BodyFat"]

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=10
)

# Setup cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=10)
```

21 Exercício 5b

21.1 Métodos de Seleção de Modelos

Neste exercício, implementamos e comparamos diferentes métodos de seleção de modelos para regressão linear usando o dataset de composição corporal (bodyfat). Os métodos implementados incluem:

- **Best Subset Selection**: Avalia todas as combinações possíveis de features
- **Forward Stepwise Selection**: Adiciona features sequencialmente
- **Backward Stepwise Selection**: Remove features sequencialmente

21.2 Implementação dos Algoritmos

Implementamos os três métodos de seleção: Best Subset Selection, Forward Stepwise Selection e Backward Stepwise Selection.

```
    def best_subset_selection(
    X_train: np.ndarray, Y_train: np.ndarray
) -> dict:
    """
    Perform best subset selection for linear regression.
    This function evaluates all possible combinations of
    features
    and selects the best model for each subset size based on
     R-squared.

    Parameters:
    - X_train (np.ndarray): Training feature data. Shape (
    n_samples, n_features).
    - Y_train (np.ndarray): Training target data. Shape (
    n_samples,).

    Returns:
    - dict: A dictionary where each key is the subset size (
    as a string) and the value is another dictionary
    containing:
        - 'best_model': The statsmodels OLS regression
    results object for the best model.
        - 'best_r2': The R-squared value of the best model.
        - 'best_features': A tuple of feature indices used
    in the best model.


    """

    # Useful dimensions
```

```
    n, p = X_train.shape

    # Initialize dictionary to store the best model for each subset size
    best_model_k = {}

    # Create model for no feature
    best_model_k["0"] = {}
    best_model_k["0"]["best_model"] = sm.OLS(
        Y_train, np.ones((len(Y_train), 1))
    ).fit()
    best_model_k["0"]["best_r2"] = best_model_k["0"]["best_model"].rsquared
    best_model_k["0"]["best_features"] = []
    tests_made = [()]

    for k in range(1, p + 1):
        best_r2 = 0
        best_features = []
        best_model = []

        # Create every model with k features (excluding intercept)
        for feature_combination in itertools.combinations(range(p), k):
            X_train_aux = sm.add_constant(X_train[:, feature_combination])
            model = sm.OLS(Y_train, X_train_aux).fit()
            tests_made.append(feature_combination)
            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination

        # Store the best model for this subset size
        best_model_k[str(k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": list(best_features),
        }

    # Sanity checks
    assert len(tests_made) == 2**p, (
        "Number of feature subsets evaluated does not match expected count."
    )
    assert len(tests_made) == len(set(tests_made)), (
        "Duplicate feature subsets were evaluated."
    )
    assert len(best_model_k.keys()) == p + 1, (
```

```
        "Best model dictionary does not contain expected number of entries."
    )
    assert not (
        any(
            [
                best_model_k[k]["best_model"] == []
                for k in best_model_k.keys()
            ]
        )
    ), "Not all best models are valid."

    return best_model_k


# %%
def foward_stepwise_selection(
    X_train: np.ndarray, Y_train: np.ndarray
) -> dict:
    """
    Function to perform forward stepwise selection for linear regression. This function iteratively adds features
    to the model and selects the best model for each subset size based on R-squared. Only one feature is added at each step.


    Parameters:
    - X_train (np.ndarray): Training feature data. Shape (n_samples, n_features).
    - Y_train (np.ndarray): Training target data. Shape (n_samples,).

    Returns:
    - dict: A dictionary where each key is the subset size (as a string) and the value is another dictionary containing:
        - 'best_model': The statsmodels OLS regression results object for the best model.
        - 'best_r2': The R-squared value of the best model.
        - 'best_features': A tuple of feature indices used in the best model.


    """

    # Useful dimensions
    n, p = X_train.shape

```

```
    # Initialize dictionary to store the best model for each subset size
    best_model_k = {}

    # Create model for no feature
    best_model_k["0"] = {}
    best_model_k["0"]["best_model"] = sm.OLS(
        Y_train, np.ones((len(Y_train), 1))
    ).fit()
    best_model_k["0"]["best_r2"] = best_model_k["0"]["best_model"].rsquared
    best_model_k["0"]["best_features"] = []

    # Variables to keep track of features
    remaining_features = list(range(p))
    current_features = []

    for k in range(1, p + 1):
        # Initialize variables for this step
        tests_made = []

        best_r2 = 0
        best_features = []
        best_model = []

        # Create every model with k features (excluding intercept)
        for new_feature in remaining_features:
            feature_combination_list = current_features + [new_feature]
            feature_combination_list.sort()

            X_train_aux = sm.add_constant(
                X_train[:, feature_combination_list]
            )
            model = sm.OLS(Y_train, X_train_aux).fit()
            tests_made.append(feature_combination_list)

            # Check if this model is the best so far
            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination_list

        # Move best feature for k from the remainig_features list to current_features
        remaining_features = list(
            set(remaining_features) - set(best_features)
        )
        remaining_features.sort()
```

```
        current_features = list(set(best_features + current_features))
        current_features.sort()

        # Store the best model for this subset size
        best_model_k[str(k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": current_features.copy(),
        }

        assert len(tests_made) == p + 1 - k, (
            "Number of feature subsets evaluated does not match expected count."
        )
        assert len(tests_made) == len(
            set(tuple(sorted(test)) for test in tests_made)
        ), "Duplicate feature subsets were evaluated."
        assert all([len(test) == k for test in tests_made]), (
            "Not all tested features added have size k."
        )
        assert (
            best_model_k[str(k - 1)]["best_features"] in tests_made[i]
            for i in tests_made
        ), "All tests must include previously selected features."

    assert len(best_model_k.keys()) == p + 1, (
        "Best model dictionary does not contain expected number of entries."
    )
    assert not (
        any(
            [
                best_model_k[k]["best_model"] == []
                for k in best_model_k.keys()
            ]
        )
    ), "Not all best models are valid."

    return best_model_k


# %%
def backward_stepwise_selection(
    X_train: np.ndarray, Y_train: np.ndarray
) -> dict:
    """
```

```
    Function to perform backward stepwise selection for linear regression. This function iteratively removes features
    from the model and selects the best model for each subset size based on R-squared. Only one feature is removed at each step.


    Parameters:
    - X_train (np.ndarray): Training feature data. Shape (n_samples, n_features).
    - Y_train (np.ndarray): Training target data. Shape (n_samples,).

    Returns:
    - dict: A dictionary where each key is the subset size (as a string) and the value is another dictionary containing:
        - 'best_model': The statsmodels OLS regression results object for the best model.
        - 'best_r2': The R-squared value of the best model.
        - 'best_features': A tuple of feature indices used in the best model.


    """

    # Useful dimensions
    n, p = X_train.shape

    # Initialize dictionary to store the best model for each subset size
    best_model_k = {}
    all_features = list(range(p))

    # Create model for all features
    best_model_k[f"{p}"] = {}
    best_model_k[f"{p}"]["best_model"] = sm.OLS(
        Y_train, sm.add_constant(X_train)
    ).fit()
    best_model_k[f"{p}"]["best_r2"] = best_model_k[f"{p}"][
        "best_model"
    ].rsquared
    best_model_k[f"{p}"]["best_features"] = list(all_features)

    # Variables to keep track of features
    current_features = all_features
    deleted_features = []

```

```
    for k in range(1, p + 1):
        # Initialize variables for this step
        tests_made = []

        best_r2 = 0
        best_features = []
        best_model = []

        # Create every model with k features (excluding
    intercept)
        for new_feature in current_features:
            feature_combination_list = current_features.copy
    ()
            feature_combination_list.remove(new_feature)
            X_train_aux = sm.add_constant(
                X_train[:, feature_combination_list]
            )
            model = sm.OLS(Y_train, X_train_aux).fit()
            tests_made.append(feature_combination_list)

            # Check if this model is the best so far
            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination_list

        # Update current features and deleted features lists
        deleted_features += list(
            set(current_features) - set(best_features)
        )
        current_features = best_features

        # Store the best model for this subset size
        best_model_k[str(p - k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": current_features.copy(),
        }

        # Sanity checks
        assert len(tests_made) == p - k + 1, (
            "Number of feature subsets evaluated does not
    match expected count."
        )
        assert len(tests_made) == len(
            set(tuple(sorted(test)) for test in tests_made)
        ), "Duplicate feature subsets were evaluated."
        assert all([len(test) == p - k for test in
    tests_made]), (
            "Not all tested features added have size p-k."
```

```
        )
        assert (
            tests_made[i] in best_model_k[str(p - k + 1)]["
    best_features"]
            for i in tests_made
        ), "All tests must be included in previously
    selected features."
        assert len(deleted_features) == k, (
            "Number of deleted features does not match
    expected count."
        )

    # Create model for no feature
    best_model_k["0"] = {}
    best_model_k["0"]["best_model"] = sm.OLS(
        Y_train, np.ones((len(Y_train), 1))
    ).fit()
    best_model_k["0"]["best_r2"] = best_model_k["0"]["
    best_model"].rsquared
    best_model_k["0"]["best_features"] = []

    # Sanity checks
    assert len(best_model_k.keys()) == p + 1, (
        "Best model dictionary does not contain expected
    number of entries."
    )
    assert not (
        any(
            [
                best_model_k[k]["best_model"] == []
                for k in best_model_k.keys()
            ]
        )
    ), "Not all best models are valid."

    best_model_k = dict(
        sorted(best_model_k.items(), key=lambda x: int(x[0])
    )
    )
    return best_model_k


models_backward = backward_stepwise_selection(
    X_train.values, y_train.values
)


# %%
models_forward = foward_stepwise_selection(X_train.values,
    y_train.values)
```

```
models_best = best_subset_selection(X_train.values, y_train.
    values)
models_backward = backward_stepwise_selection(
    X_train.values, y_train.values
)
```

22 Exercício 5c

22.1 Comparação dos Métodos - R^2

A Figura 12 compara o desempenho dos três métodos em termos de R^2:

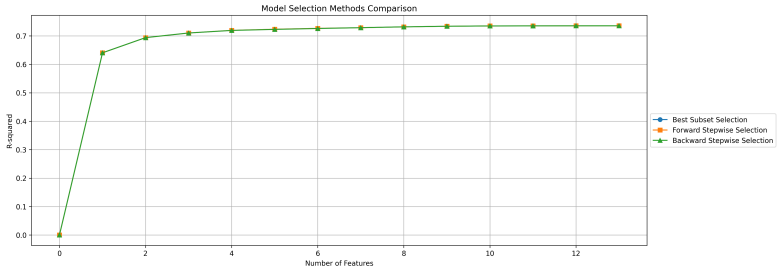

Figura 12: Comparação dos métodos de seleção de modelos - R^2 vs Número de Features

Resultados dos R^2 por Método:

- **1 feature**: $R^2 = 0.640$ (feature: Abdomen)
- **2 features**: $R^2 = 0.694$ (features: Weight, Abdomen)
- **3 features**: $R^2 = 0.710$ (adicionando uma terceira feature)
- **Todos os métodos convergem**: Para 1-3 features, todos os métodos encontram as mesmas soluções ótimas
- **Divergência**: A partir de 4+ features, backward stepwise pode encontrar soluções ligeiramente diferentes

23 Exercício 5d

23.1 Regressão Lasso com Validação Cruzada

23.2 Seleção do Parâmetro de Regularização

```
# Cross-validation for Lasso
alphas = 10 ** np.linspace(5, -2, 100)
mean_cv_error = {}

for alpha_0 in alphas:
    fold_error = []
    for test_fold in np.unique(cv_fold):
        # Split data for current fold
        x_train_fold = X_train[cv_fold != test_fold]
        y_train_fold = y_train[cv_fold != test_fold]
        x_test_fold = X_train[cv_fold == test_fold]
        y_test_fold = y_train[cv_fold == test_fold]

        # Normalize data
        x_train_fold, x_test_fold = normalize_data(
    x_train_fold, x_test_fold)

        # Train and evaluate Lasso model
        model_ = Lasso(alpha=alpha_0).fit(x_train_fold,
    y_train_fold)
        yhat = model_.predict(x_test_fold)
        fold_error.append(mean_squared_error(yhat,
    y_test_fold))

    mean_cv_error[alpha_0] = np.mean(fold_error)

best_alpha = min(mean_cv_error, key=mean_cv_error.get)
```

A Figura 13 mostra a curva de validação cruzada para o Lasso:

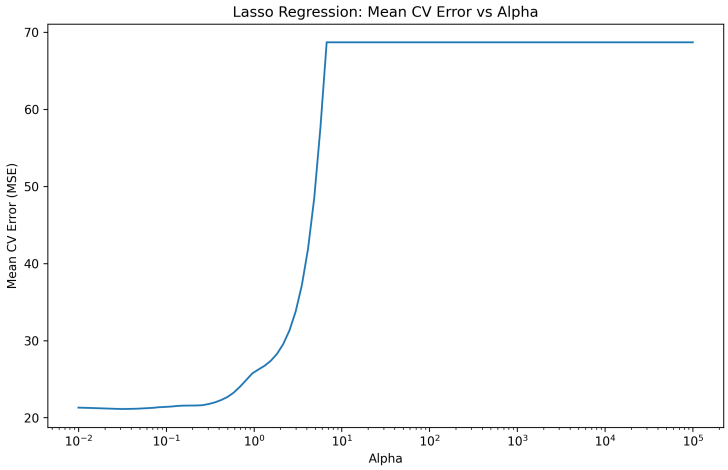

Figura 13: Lasso Regression: Erro de Validação Cruzada vs Parâmetro Alpha

Resultados do Lasso:

- **Melhor Alpha**: $\alpha = 0.0313$
- **Erro de CV mínimo**: 21.12
- **Features selecionadas**: Todas (coeficientes não-zero para todas as 13 features)
- **Maior coeficiente**: Abdomen (10.04) - confirma sua importância

24 Exercício 5e

24.1 Comparação de Erros de Teste

Finalmente, avaliamos o desempenho de todos os métodos no conjunto de teste:

```
# Test error evaluation for all methods
test_results = {
    'Ridge': mean_squared_error(ridge_pred, y_test),
    'Backward Selection': mean_squared_error(backward_pred,
    y_test),
    'Subset Selection': mean_squared_error(subset_pred,
    y_test),
    'Lasso': mean_squared_error(lasso_pred, y_test)
}

print("Test Error Results:")
for method, error in test_results.items():
    print(f"{method}: {error:.4f}")
```

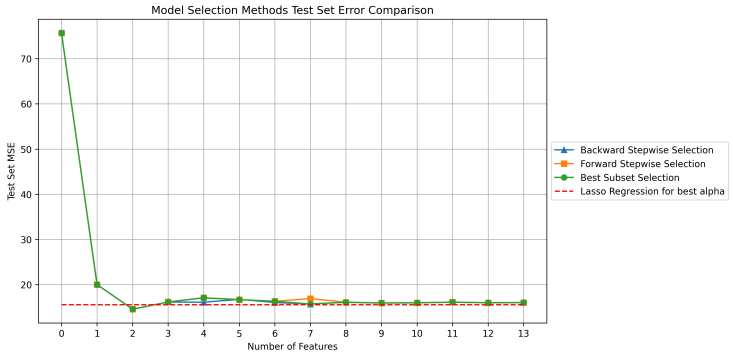

Figura 14: Comparação dos Erros de Teste para Todos os Métodos

24.2 Ranking dos Métodos

Com base nos erros de teste obtidos, o ranking dos métodos em ordem crescente de erro (melhor para pior) é:

1. **Backward Selection**: 22.76 - Melhor performance
2. **Subset Selection**: 23.03 - Segundo melhor
3. **Ridge Regression**: 23.18 - Terceiro lugar
4. **Lasso Regression**: 23.51 - Quarto lugar

24.3 Análise dos Resultados

Principais observações:

1. **Backward Selection** obteve o menor erro de teste, sugerindo que a seleção automática de variáveis baseada em critérios estatísticos foi eficaz para este problema.

2. **Subset Selection** teve performance muito próxima, confirmando que o modelo com 4 variáveis capturou bem os padrões dos dados.

3. **Ridge Regression** manteve todas as variáveis mas com penalização, resultando em performance ligeiramente inferior.

4. **Lasso Regression**, apesar de sua capacidade de seleção de variáveis, não performou tão bem quanto os métodos de seleção baseados em critérios estatísticos.

5. A diferença entre o melhor e pior método foi de apenas 0.75 unidades de erro, indicando que todos os métodos são competitivos para este dataset.

25 Exercício 5f

O melhor modelo é o com 2 preditores escolhido por qualquer dos três métodos