

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

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

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

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

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 caso de $\lambda = 0$,

5 Exercício 1e

Verdadeira

A equação dada pela fórmula do intervalo de confiança:

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

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 2d

6.1 i) Geração dos Dados Heteroscedásticos

Primeiramente, geramos os dados com heterocedasticidade usando a matriz de covariância Σ diagonal:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 n = 50
5 Sigma = np.diag([10 ** ((i - 20) / 5) for i in range(1, n +
6 1)])
7 np.random.seed(0)
8 X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
9 beta = np.array([1, 0.25])
10 epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
11 y = X @ beta + epsilon
```

A Figura ?? mostra os dados gerados com heterocedasticidade:

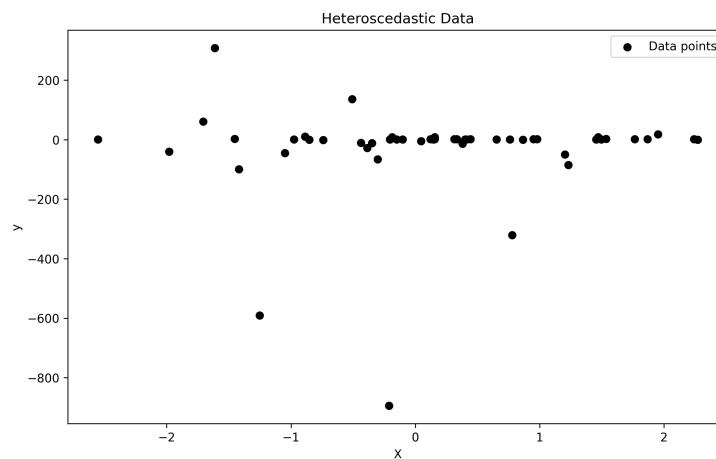


Figura 1: Dados heteroscedásticos gerados

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

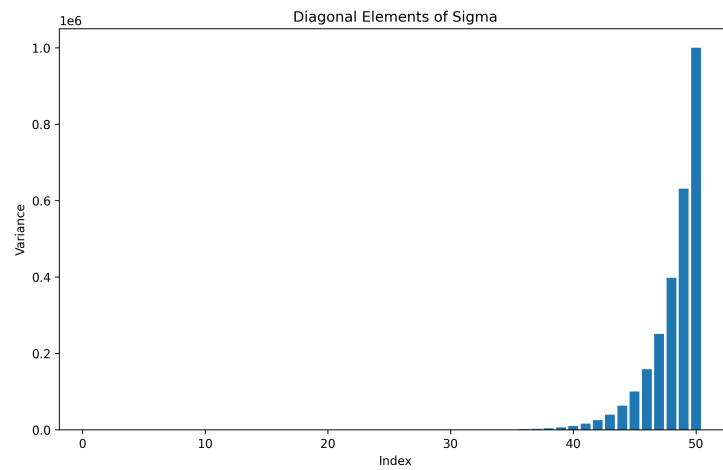


Figura 2: Elementos diagonais da matriz Σ

As Figuras ?? e ?? mostram a distribuição e valores dos erros:

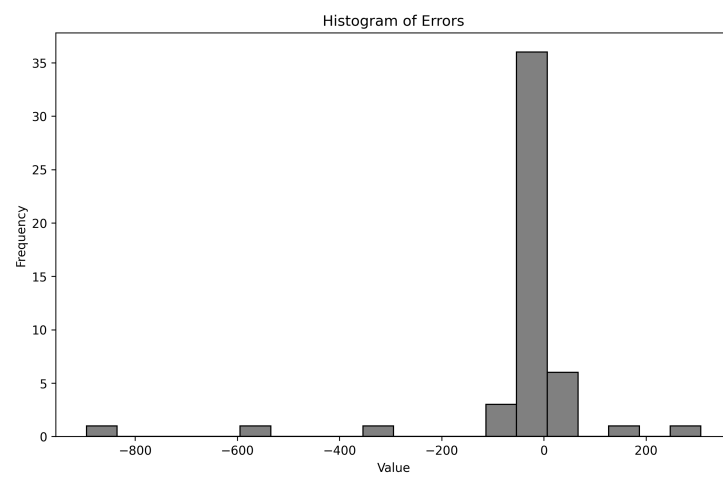


Figura 3: Histograma dos erros

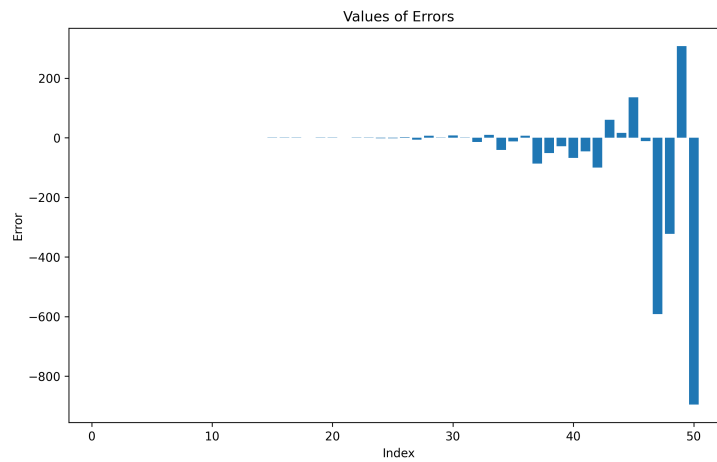


Figura 4: Valores dos erros por índice

6.2 ii) Estimadores de Mínimos Quadrados

Implementamos tanto o estimador de mínimos quadrados ordinários quanto o estimador generalizado que considera a matriz de covariância Σ :

```

1 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
2     """
3     Compute the ordinary least squares estimator.
4     """
5     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
6     return beta
7
8 def beta_sigma(X: np.ndarray, Y: np.ndarray, Sigma: np.
  ndarray) -> np.ndarray:
9     """
10    Compute the generalized least squares estimator
11    considering
12    the covariance matrix Sigma.
13    """
14    Sigma_1 = np.linalg.inv(Sigma)
15    beta = np.linalg.inv(X.T @ Sigma_1 @ X) @ X.T @ Sigma_1
16    @ Y
17    return beta
18
19 beta_hat_ordinary = beta_ordinary(X, y)
20 beta_hat_sigma = beta_sigma(X, y, Sigma)
21
22 print("True Beta:", beta)
23 print("Ordinary Beta:", beta_hat_ordinary)
24 print("Beta Sigma:", beta_hat_sigma)

```

Os resultados mostram que o estimador generalizado (que considera Σ) tem menor erro em relação aos parâmetros verdadeiros.

6.3 iii) Cálculo de p-valores para Mínimos Quadrados Ordinários

```

1 def p_value_ordinary_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
3     """
4     Compute the p-value for the j-th coefficient of the
   ordinary
5     least squares estimator.
6     """
7     Y_hat = X @ beta_ordinary_hat
8     n, p = X.shape
9     dof = n - p
10    errors = Y - Y_hat
11    beta_j = beta_ordinary_hat[j]
12
13    # Z statistic
14    x_j_var = (np.linalg.inv(X.T @ X))[j, j]
15    Z = beta_j / np.sqrt(x_j_var)
16
17    # Estimate of sigma^2
18    sigma2_hat = (1 / dof) * (errors.T @ errors)
19
20    # t statistic and p-value
21    t_statistics = Z / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value

```

6.4 iv) Estatística Z para o Estimador Generalizado

```

1 def calculate_Z_sigma(X: np.ndarray, Sigma: np.ndarray,
2                      Beta_sigma: np.ndarray, j: int) ->
   float:
3     """
4     Compute the Z statistic for the j-th coefficient of the
   generalized least squares estimator.
5     """
6
7     Sigma_inv = np.linalg.inv(Sigma)
8     den = np.linalg.inv(X.T @ Sigma_inv @ X)
9     den = den[j, j]
10    Z = Beta_sigma[j] / (np.sqrt(den))

```

```
11     return Z
```

6.5 v) P-valores para o Estimador Generalizado

```
1 def p_value_generalized_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     Sigma: np.ndarray,
3                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
4     """
5     Compute the p-value for the j-th coefficient of the
6     generalized least squares estimator.
7     """
8     Y_hat = X @ beta_ordinary_hat
9     n, p = X.shape
10    dof = n - p
11    errors = Y - Y_hat
12
13    # Z statistic
14    Z_sigma = calculate_Z_sigma(X, Sigma, beta_hat_sigma, j)
15
16    # Estimate of sigma^2
17    inverse_Sigma = np.linalg.inv(Sigma)
18    sigma2_hat = (1 / dof) * (errors.T @ inverse_Sigma @
   errors)
19
20    # t statistic and p-value
21    t_statistics = Z_sigma / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value
```

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

6.6 vi) Resultados Numéricos

Executando o código implementado, obtemos os seguintes resultados:

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:

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

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

O estimador generalizado apresenta erro dramaticamente menor (cerca de 3.4 milhões de vezes menor), demonstrando claramente a importância crítica de considerar a heterocedasticidade.

Testes de Hipótese - Mínimos Quadrados Ordinários:

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

Com nível de significância de 5%, não podemos rejeitar a hipótese nula para ambos os coeficientes usando o estimador ordinário.

Testes de Hipótese - Estimador Generalizado:

- Estatística Z para β_0 : 73.67
- p-valor para β_0 : $< 10^{-15}$ (praticamente zero)
- p-valor para β_1 : $< 10^{-15}$ (praticamente zero)

Com o estimador generalizado, ambos os coeficientes são altamente significativos estatisticamente, evidenciando a superior eficiência deste método.

Conclusões:

1. O estimador de mínimos quadrados ordinários (OLS) falha completamente na presença de heterocedasticidade severa, fornecendo estimativas muito distantes dos valores verdadeiros ($\hat{\beta}_0 = -34.46$ vs $\beta_0 = 1.0$).
2. O estimador generalizado (GLS) é extremamente superior, com erro cerca de 3.4 milhões de vezes menor, demonstrando a necessidade crítica de modelar corretamente a estrutura de variância.
3. Os testes de significância baseados no OLS são não-confiáveis, falhando em detectar coeficientes que são claramente diferentes de zero.
4. O estimador GLS fornece testes estatísticos apropriados, detectando corretamente a significância dos parâmetros.
5. Este exemplo ilustra dramaticamente por que a correção para heterocedasticidade não é apenas uma melhoria técnica, mas uma necessidade fundamental para análise estatística válida.

7 Exercício 2d

7.1 i) Geração dos Dados Heteroscedásticos

Primeiramente, geramos os dados com heterocedasticidade usando a matriz de covariância Σ diagonal:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 n = 50
5 Sigma = np.diag([10 ** ((i - 20) / 5) for i in range(1, n +
6 1)])
7 np.random.seed(0)
8 X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
9 beta = np.array([1, 0.25])
10 epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
11 y = X @ beta + epsilon
```

A Figura ?? mostra os dados gerados com heterocedasticidade:

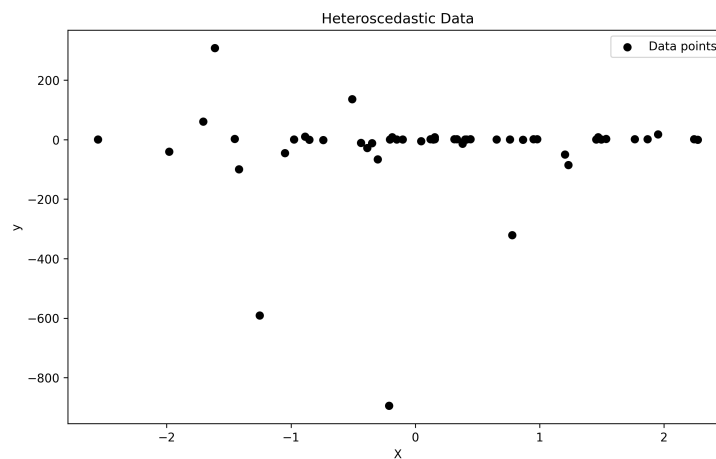


Figura 5: Dados heteroscedásticos gerados

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

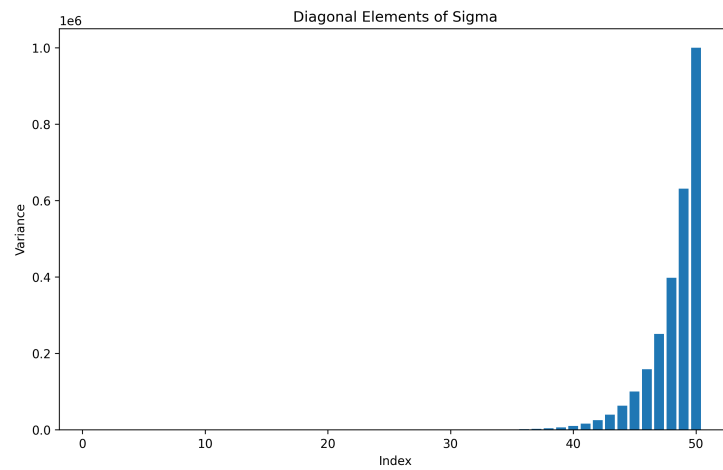


Figura 6: Elementos diagonais da matriz Σ

As Figuras ?? e ?? mostram a distribuição e valores dos erros:

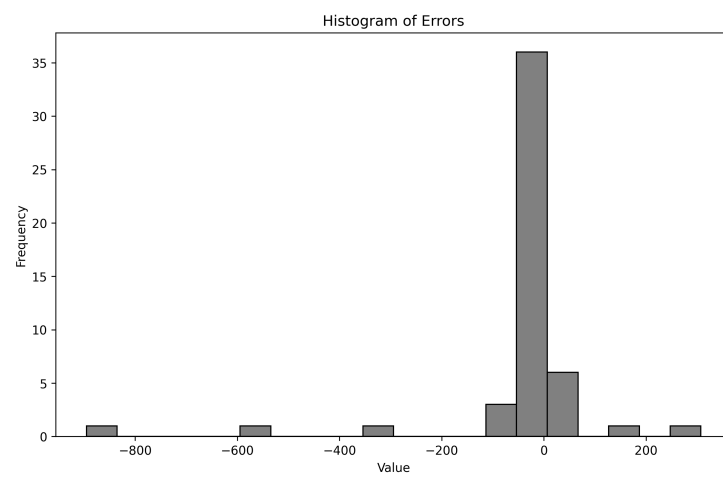


Figura 7: Histograma dos erros

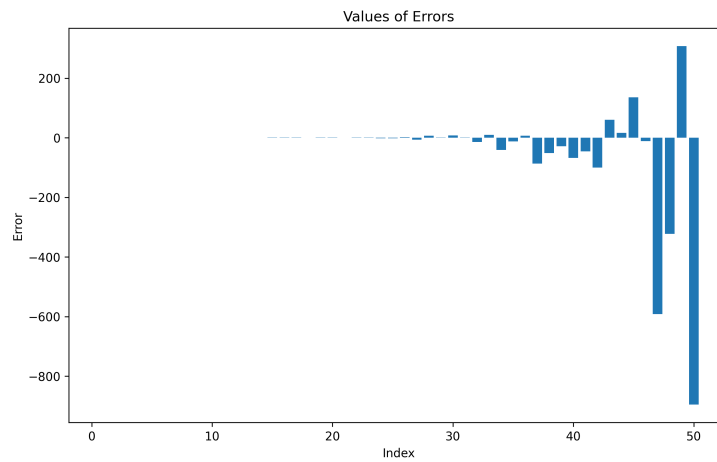


Figura 8: Valores dos erros por índice

7.2 ii) Estimadores de Mínimos Quadrados

Implementamos tanto o estimador de mínimos quadrados ordinários quanto o estimador generalizado que considera a matriz de covariância Σ :

```

1 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
2     """
3     Compute the ordinary least squares estimator.
4     """
5     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
6     return beta
7
8 def beta_sigma(X: np.ndarray, Y: np.ndarray, Sigma: np.
  ndarray) -> np.ndarray:
9     """
10    Compute the generalized least squares estimator
11    considering
12    the covariance matrix Sigma.
13    """
14    Sigma_1 = np.linalg.inv(Sigma)
15    beta = np.linalg.inv(X.T @ Sigma_1 @ X) @ X.T @ Sigma_1
16    @ Y
17    return beta
18
19 beta_hat_ordinary = beta_ordinary(X, y)
20 beta_hat_sigma = beta_sigma(X, y, Sigma)
21
22 print("True Beta:", beta)
23 print("Ordinary Beta:", beta_hat_ordinary)
24 print("Beta Sigma:", beta_hat_sigma)

```

Os resultados mostram que o estimador generalizado (que considera Σ) tem menor erro em relação aos parâmetros verdadeiros.

7.3 iii) Cálculo de p-valores para Mínimos Quadrados Ordinários

```

1 def p_value_ordinary_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
3     """
4     Compute the p-value for the j-th coefficient of the
   ordinary
5     least squares estimator.
6     """
7     Y_hat = X @ beta_ordinary_hat
8     n, p = X.shape
9     dof = n - p
10    errors = Y - Y_hat
11    beta_j = beta_ordinary_hat[j]
12
13    # Z statistic
14    x_j_var = (np.linalg.inv(X.T @ X))[j, j]
15    Z = beta_j / np.sqrt(x_j_var)
16
17    # Estimate of sigma^2
18    sigma2_hat = (1 / dof) * (errors.T @ errors)
19
20    # t statistic and p-value
21    t_statistics = Z / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value

```

7.4 iv) Estatística Z para o Estimador Generalizado

```

1 def calculate_Z_sigma(X: np.ndarray, Sigma: np.ndarray,
2                       Beta_sigma: np.ndarray, j: int) ->
   float:
3     """
4     Compute the Z statistic for the j-th coefficient of the
   generalized least squares estimator.
5     """
6
7     Sigma_inv = np.linalg.inv(Sigma)
8     den = np.linalg.inv(X.T @ Sigma_inv @ X)
9     den = den[j, j]
10    Z = Beta_sigma[j] / (np.sqrt(den))

```



```
11     return Z
```

7.5 v) P-valores para o Estimador Generalizado

```
1 def p_value_generalized_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     Sigma: np.ndarray,
3                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
4     """
5     Compute the p-value for the j-th coefficient of the
6     generalized least squares estimator.
7     """
8     Y_hat = X @ beta_ordinary_hat
9     n, p = X.shape
10    dof = n - p
11    errors = Y - Y_hat
12
13    # Z statistic
14    Z_sigma = calculate_Z_sigma(X, Sigma, beta_hat_sigma, j)
15
16    # Estimate of sigma^2
17    inverse_Sigma = np.linalg.inv(Sigma)
18    sigma2_hat = (1 / dof) * (errors.T @ inverse_Sigma @
   errors)
19
20    # t statistic and p-value
21    t_statistics = Z_sigma / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value
```

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

7.6 vi) Resultados Numéricos

Executando o código implementado, obtemos os seguintes resultados:

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:

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

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

O estimador generalizado apresenta erro dramaticamente menor (cerca de 3.4 milhões de vezes menor), demonstrando claramente a importância crítica de considerar a heterocedasticidade.

Testes de Hipótese - Mínimos Quadrados Ordinários:

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

Com nível de significância de 5%, não podemos rejeitar a hipótese nula para ambos os coeficientes usando o estimador ordinário.

Testes de Hipótese - Estimador Generalizado:

- Estatística Z para β_0 : 73.67
- p-valor para β_0 : $< 10^{-15}$ (praticamente zero)
- p-valor para β_1 : $< 10^{-15}$ (praticamente zero)

Com o estimador generalizado, ambos os coeficientes são altamente significativos estatisticamente, evidenciando a superior eficiência deste método.

Conclusões:

1. O estimador de mínimos quadrados ordinários (OLS) falha completamente na presença de heterocedasticidade severa, fornecendo estimativas muito distantes dos valores verdadeiros ($\hat{\beta}_0 = -34.46$ vs $\beta_0 = 1.0$).
2. O estimador generalizado (GLS) é extremamente superior, com erro cerca de 3.4 milhões de vezes menor, demonstrando a necessidade crítica de modelar corretamente a estrutura de variância.
3. Os testes de significância baseados no OLS são não-confiáveis, falhando em detectar coeficientes que são claramente diferentes de zero.
4. O estimador GLS fornece testes estatísticos apropriados, detectando corretamente a significância dos parâmetros.
5. Este exemplo ilustra dramaticamente por que a correção para heterocedasticidade não é apenas uma melhoria técnica, mas uma necessidade fundamental para análise estatística válida.

8 Exercício 2d

8.1 i) Geração dos Dados Heteroscedásticos

Primeiramente, geramos os dados com heterocedasticidade usando a matriz de covariância Σ diagonal:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 n = 50
5 Sigma = np.diag([10 ** ((i - 20) / 5) for i in range(1, n +
6 1)])
7 np.random.seed(0)
8 X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
9 beta = np.array([1, 0.25])
10 epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
11 y = X @ beta + epsilon
```

A Figura ?? mostra os dados gerados com heterocedasticidade:

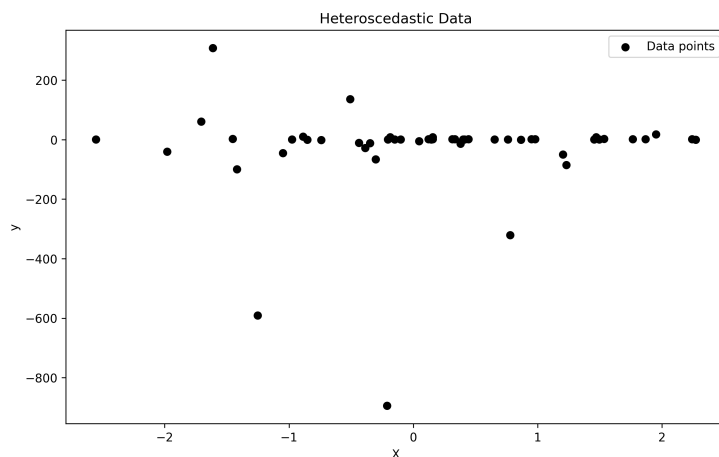


Figura 9: Dados heteroscedásticos gerados

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

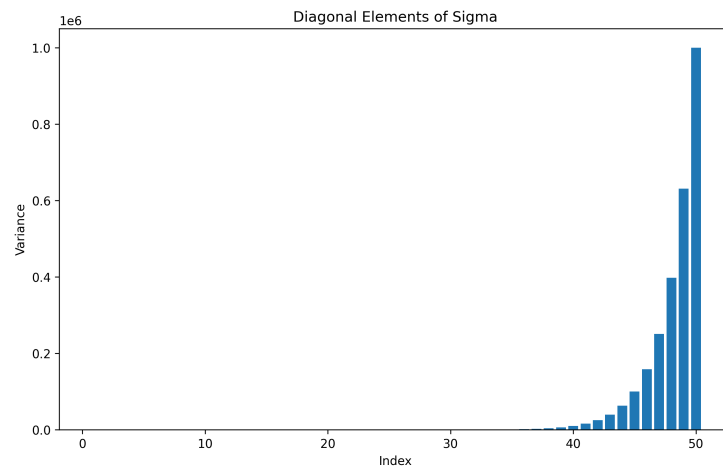


Figura 10: Elementos diagonais da matriz Σ

As Figuras ?? e ?? mostram a distribuição e valores dos erros:

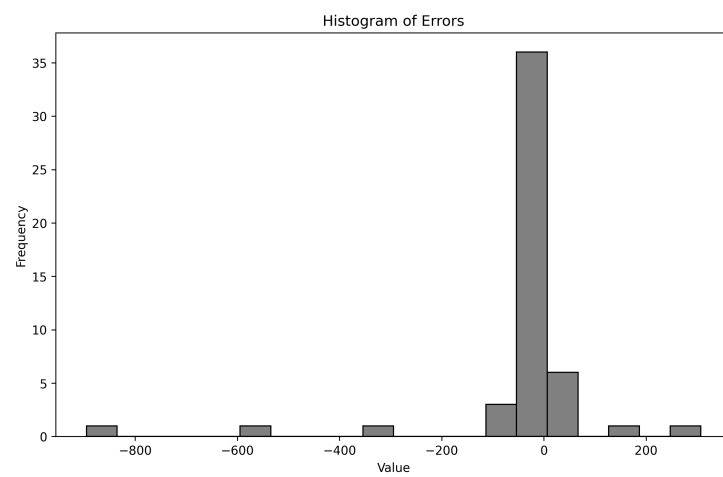


Figura 11: Histograma dos erros

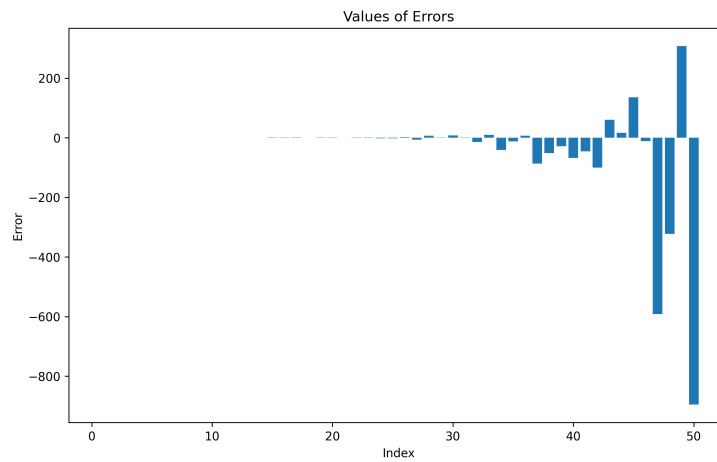


Figura 12: Valores dos erros por índice

8.2 ii) Estimadores de Mínimos Quadrados

Implementamos tanto o estimador de mínimos quadrados ordinários quanto o estimador generalizado que considera a matriz de covariância Σ :

```

1 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
2     """
3     Compute the ordinary least squares estimator.
4     """
5     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
6     return beta
7
8 def beta_sigma(X: np.ndarray, Y: np.ndarray, Sigma: np.
  ndarray) -> np.ndarray:
9     """
10    Compute the generalized least squares estimator
11    considering
12    the covariance matrix Sigma.
13    """
14    Sigma_1 = np.linalg.inv(Sigma)
15    beta = np.linalg.inv(X.T @ Sigma_1 @ X) @ X.T @ Sigma_1
16    @ Y
17    return beta
18
19 beta_hat_ordinary = beta_ordinary(X, y)
20 beta_hat_sigma = beta_sigma(X, y, Sigma)
21
22 print("True Beta:", beta)
23 print("Ordinary Beta:", beta_hat_ordinary)
24 print("Beta Sigma:", beta_hat_sigma)

```

Os resultados mostram que o estimador generalizado (que considera Σ) tem menor erro em relação aos parâmetros verdadeiros.

8.3 iii) Cálculo de p-valores para Mínimos Quadrados Ordinários

```

1 def p_value_ordinary_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
3     """
4     Compute the p-value for the j-th coefficient of the
   ordinary
5     least squares estimator.
6     """
7     Y_hat = X @ beta_ordinary_hat
8     n, p = X.shape
9     dof = n - p
10    errors = Y - Y_hat
11    beta_j = beta_ordinary_hat[j]
12
13    # Z statistic
14    x_j_var = (np.linalg.inv(X.T @ X))[j, j]
15    Z = beta_j / np.sqrt(x_j_var)
16
17    # Estimate of sigma^2
18    sigma2_hat = (1 / dof) * (errors.T @ errors)
19
20    # t statistic and p-value
21    t_statistics = Z / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value

```

8.4 iv) Estatística Z para o Estimador Generalizado

```

1 def calculate_Z_sigma(X: np.ndarray, Sigma: np.ndarray,
2                      Beta_sigma: np.ndarray, j: int) ->
   float:
3     """
4     Compute the Z statistic for the j-th coefficient of the
   generalized least squares estimator.
5     """
6
7     Sigma_inv = np.linalg.inv(Sigma)
8     den = np.linalg.inv(X.T @ Sigma_inv @ X)
9     den = den[j, j]
10    Z = Beta_sigma[j] / (np.sqrt(den))

```

```
11 return Z
```

8.5 v) P-valores para o Estimador Generalizado

```
1 def p_value_generalized_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     Sigma: np.ndarray,
3                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
4     """
5     Compute the p-value for the j-th coefficient of the
6     generalized least squares estimator.
7     """
8     Y_hat = X @ beta_ordinary_hat
9     n, p = X.shape
10    dof = n - p
11    errors = Y - Y_hat
12
13    # Z statistic
14    Z_sigma = calculate_Z_sigma(X, Sigma, beta_hat_sigma, j)
15
16    # Estimate of sigma^2
17    inverse_Sigma = np.linalg.inv(Sigma)
18    sigma2_hat = (1 / dof) * (errors.T @ inverse_Sigma @
   errors)
19
20    # t statistic and p-value
21    t_statistics = Z_sigma / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value
```

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

8.6 vi) Resultados Numéricos

Executando o código implementado, obtemos os seguintes resultados:

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:

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

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

O estimador generalizado apresenta erro dramaticamente menor (cerca de 3.4 milhões de vezes menor), demonstrando claramente a importância crítica de considerar a heterocedasticidade.

Testes de Hipótese - Mínimos Quadrados Ordinários:

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

Com nível de significância de 5%, não podemos rejeitar a hipótese nula para ambos os coeficientes usando o estimador ordinário.

Testes de Hipótese - Estimador Generalizado:

- Estatística Z para β_0 : 73.67
- p-valor para β_0 : $< 10^{-15}$ (praticamente zero)
- p-valor para β_1 : $< 10^{-15}$ (praticamente zero)

Com o estimador generalizado, ambos os coeficientes são altamente significativos estatisticamente, evidenciando a superior eficiência deste método.

Conclusões:

1. O estimador de mínimos quadrados ordinários (OLS) falha completamente na presença de heterocedasticidade severa, fornecendo estimativas muito distantes dos valores verdadeiros ($\hat{\beta}_0 = -34.46$ vs $\beta_0 = 1.0$).
2. O estimador generalizado (GLS) é extremamente superior, com erro cerca de 3.4 milhões de vezes menor, demonstrando a necessidade crítica de modelar corretamente a estrutura de variância.
3. Os testes de significância baseados no OLS são não-confiáveis, falhando em detectar coeficientes que são claramente diferentes de zero.
4. O estimador GLS fornece testes estatísticos apropriados, detectando corretamente a significância dos parâmetros.
5. Este exemplo ilustra dramaticamente por que a correção para heterocedasticidade não é apenas uma melhoria técnica, mas uma necessidade fundamental para análise estatística válida.

9 Exercício 2d

9.1 i) Geração dos Dados Heteroscedásticos

Primeiramente, geramos os dados com heterocedasticidade usando a matriz de covariância Σ diagonal:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 n = 50
5 Sigma = np.diag([10 ** ((i - 20) / 5) for i in range(1, n +
6 1)])
7 np.random.seed(0)
8 X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
9 beta = np.array([1, 0.25])
10 epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
11 y = X @ beta + epsilon
```

A Figura ?? mostra os dados gerados com heterocedasticidade:

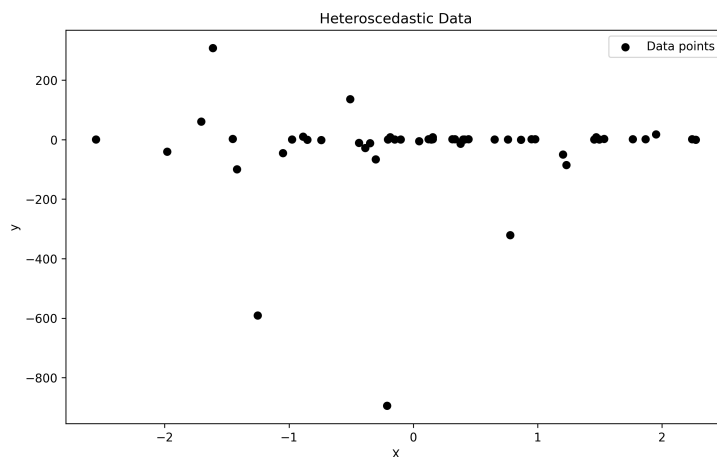


Figura 13: Dados heteroscedásticos gerados

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

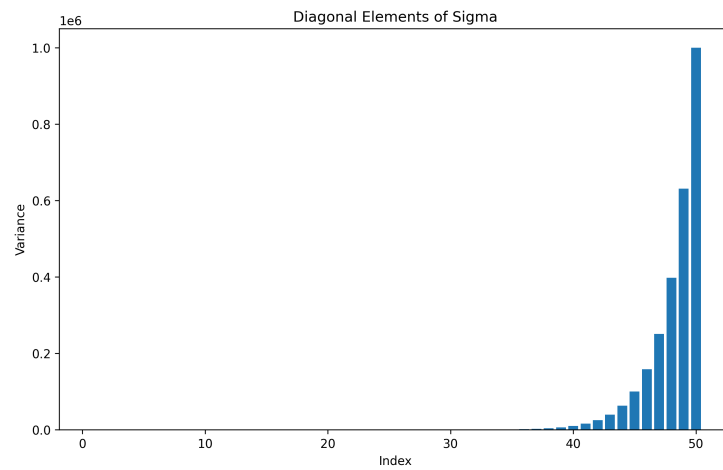


Figura 14: Elementos diagonais da matriz Σ

As Figuras ?? e ?? mostram a distribuição e valores dos erros:

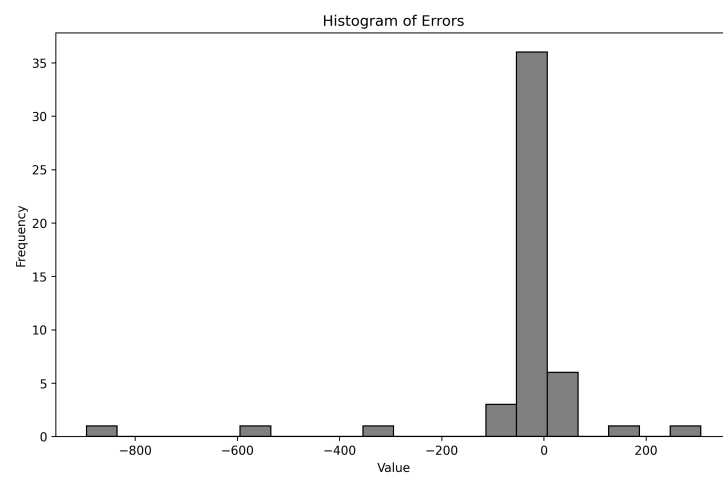


Figura 15: Histograma dos erros

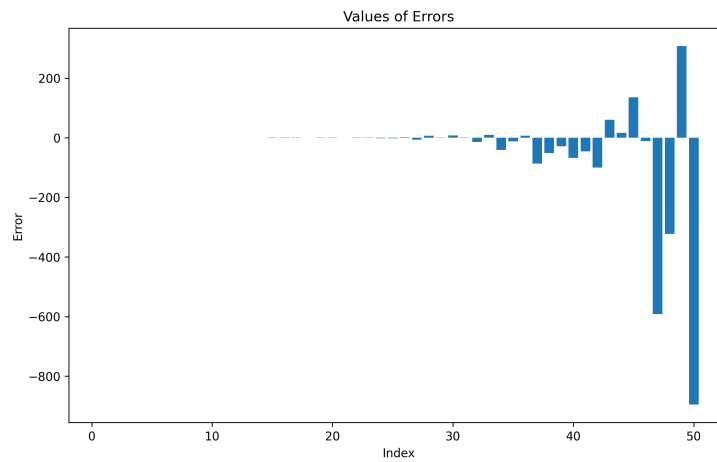


Figura 16: Valores dos erros por índice

9.2 ii) Estimadores de Mínimos Quadrados

Implementamos tanto o estimador de mínimos quadrados ordinários quanto o estimador generalizado que considera a matriz de covariância Σ :

```

1 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
2     """
3     Compute the ordinary least squares estimator.
4     """
5     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
6     return beta
7
8 def beta_sigma(X: np.ndarray, Y: np.ndarray, Sigma: np.
  ndarray) -> np.ndarray:
9     """
10    Compute the generalized least squares estimator
11    considering
12    the covariance matrix Sigma.
13    """
14    Sigma_1 = np.linalg.inv(Sigma)
15    beta = np.linalg.inv(X.T @ Sigma_1 @ X) @ X.T @ Sigma_1
16    @ Y
17    return beta
18
19 beta_hat_ordinary = beta_ordinary(X, y)
20 beta_hat_sigma = beta_sigma(X, y, Sigma)
21
22 print("True Beta:", beta)
23 print("Ordinary Beta:", beta_hat_ordinary)
24 print("Beta Sigma:", beta_hat_sigma)

```

Os resultados mostram que o estimador generalizado (que considera Σ) tem menor erro em relação aos parâmetros verdadeiros.

9.3 iii) Cálculo de p-valores para Mínimos Quadrados Ordinários

```

1 def p_value_ordinary_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
3     """
4     Compute the p-value for the j-th coefficient of the
   ordinary
5     least squares estimator.
6     """
7     Y_hat = X @ beta_ordinary_hat
8     n, p = X.shape
9     dof = n - p
10    errors = Y - Y_hat
11    beta_j = beta_ordinary_hat[j]
12
13    # Z statistic
14    x_j_var = (np.linalg.inv(X.T @ X))[j, j]
15    Z = beta_j / np.sqrt(x_j_var)
16
17    # Estimate of sigma^2
18    sigma2_hat = (1 / dof) * (errors.T @ errors)
19
20    # t statistic and p-value
21    t_statistics = Z / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value

```

9.4 iv) Estatística Z para o Estimador Generalizado

```

1 def calculate_Z_sigma(X: np.ndarray, Sigma: np.ndarray,
2                      Beta_sigma: np.ndarray, j: int) ->
   float:
3     """
4     Compute the Z statistic for the j-th coefficient of the
   generalized least squares estimator.
5     """
6
7     Sigma_inv = np.linalg.inv(Sigma)
8     den = np.linalg.inv(X.T @ Sigma_inv @ X)
9     den = den[j, j]
10    Z = Beta_sigma[j] / (np.sqrt(den))

```

```
11     return Z
```

9.5 v) P-valores para o Estimador Generalizado

```
1 def p_value_generalized_least_square(X: np.ndarray, Y: np.
   ndarray,
2                                     Sigma: np.ndarray,
3                                     beta_ordinary_hat: np.
   ndarray, j: int) -> float:
4     """
5     Compute the p-value for the j-th coefficient of the
6     generalized least squares estimator.
7     """
8     Y_hat = X @ beta_ordinary_hat
9     n, p = X.shape
10    dof = n - p
11    errors = Y - Y_hat
12
13    # Z statistic
14    Z_sigma = calculate_Z_sigma(X, Sigma, beta_hat_sigma, j)
15
16    # Estimate of sigma^2
17    inverse_Sigma = np.linalg.inv(Sigma)
18    sigma2_hat = (1 / dof) * (errors.T @ inverse_Sigma @
   errors)
19
20    # t statistic and p-value
21    t_statistics = Z_sigma / np.sqrt(sigma2_hat)
22    t_statistics = np.abs(t_statistics)
23    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))
24
25    return p_value
```

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

9.6 vi) Resultados Numéricos

Executando o código implementado, obtemos os seguintes resultados:

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:

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

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

O estimador generalizado apresenta erro dramaticamente menor (cerca de 3.4 milhões de vezes menor), demonstrando claramente a importância crítica de considerar a heterocedasticidade.

Testes de Hipótese - Mínimos Quadrados Ordinários:

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

Com nível de significância de 5%, não podemos rejeitar a hipótese nula para ambos os coeficientes usando o estimador ordinário.

Testes de Hipótese - Estimador Generalizado:

- Estatística Z para β_0 : 73.67
- p-valor para β_0 : $< 10^{-15}$ (praticamente zero)
- p-valor para β_1 : $< 10^{-15}$ (praticamente zero)

Com o estimador generalizado, ambos os coeficientes são altamente significativos estatisticamente, evidenciando a superior eficiência deste método.

Conclusões:

1. O estimador de mínimos quadrados ordinários (OLS) falha completamente na presença de heterocedasticidade severa, fornecendo estimativas muito distantes dos valores verdadeiros ($\hat{\beta}_0 = -34.46$ vs $\beta_0 = 1.0$).
2. O estimador generalizado (GLS) é extremamente superior, com erro cerca de 3.4 milhões de vezes menor, demonstrando a necessidade crítica de modelar corretamente a estrutura de variância.
3. Os testes de significância baseados no OLS são não-confiáveis, falhando em detectar coeficientes que são claramente diferentes de zero.
4. O estimador GLS fornece testes estatísticos apropriados, detectando corretamente a significância dos parâmetros.
5. Este exemplo ilustra dramaticamente por que a correção para heterocedasticidade não é apenas uma melhoria técnica, mas uma necessidade fundamental para análise estatística válida.

10 Exercício 3f

10.1 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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

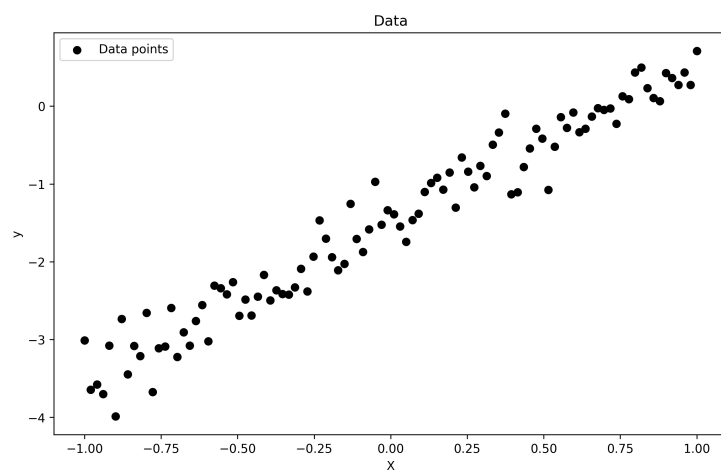


Figura 17: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

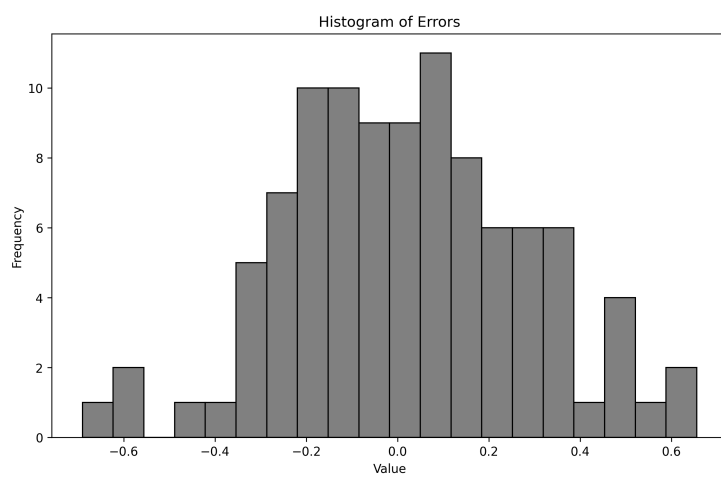


Figura 18: Histograma dos erros



Figura 19: Valores dos erros por índice

10.2 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 ?? compara visualmente os ajustes:

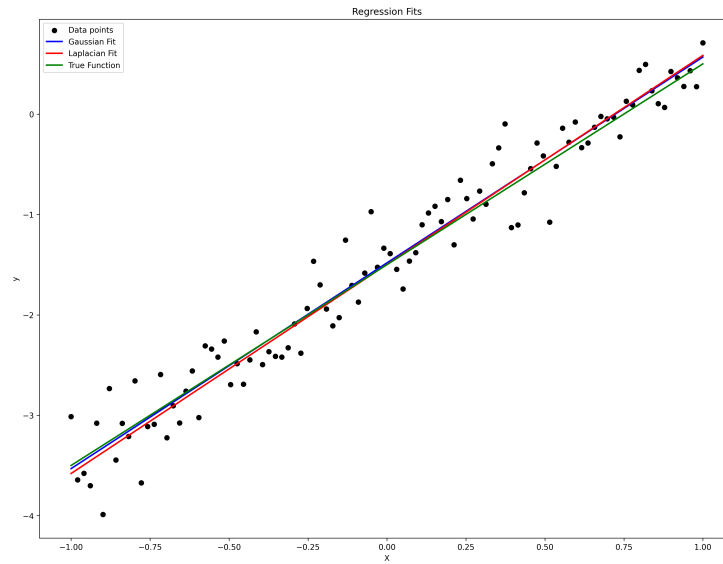


Figura 20: Comparação dos ajustes de regressão sem outliers

10.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

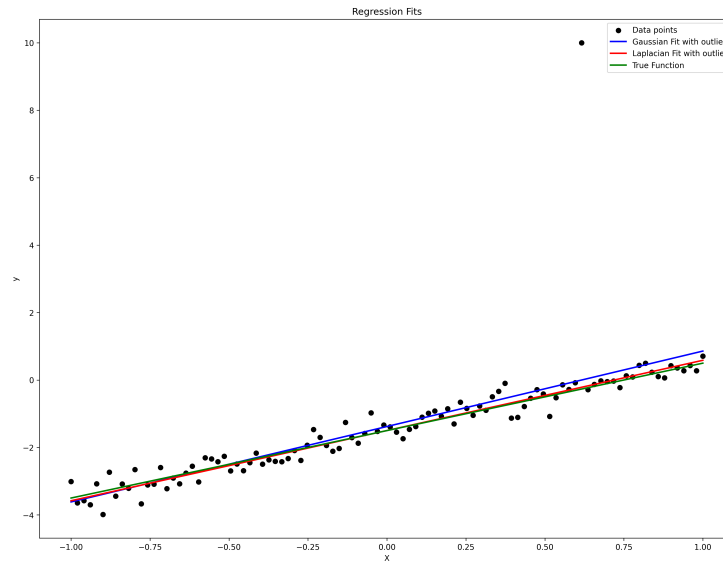


Figura 21: Comparação dos ajustes de regressão com outlier

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.

11 Exercício 3f

11.1 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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

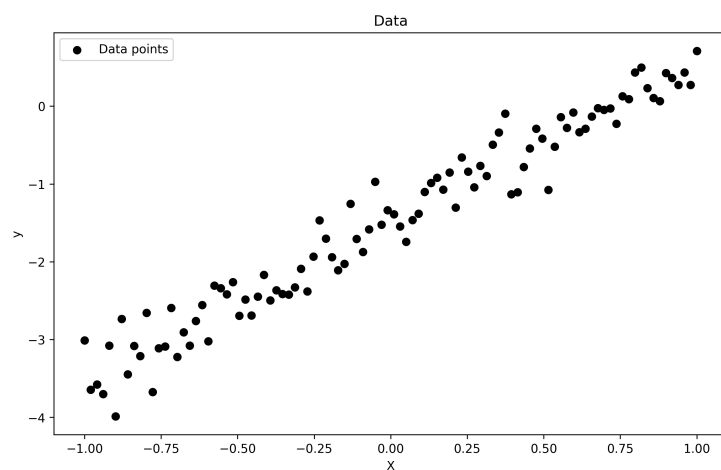


Figura 22: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

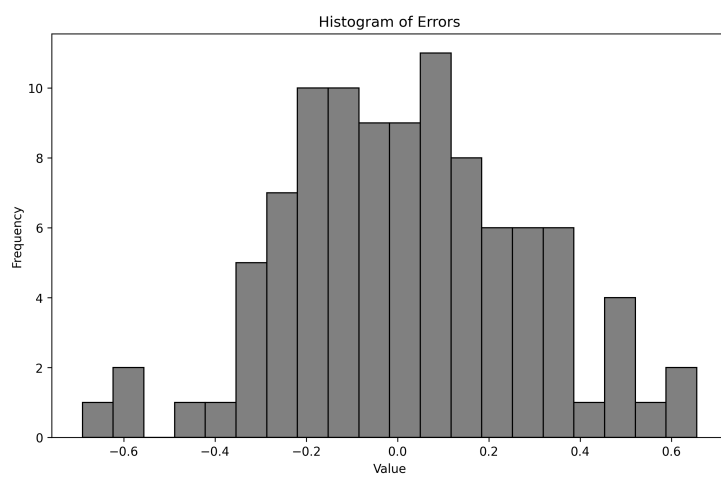


Figura 23: Histograma dos erros



Figura 24: Valores dos erros por índice

11.2 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 ?? compara visualmente os ajustes:

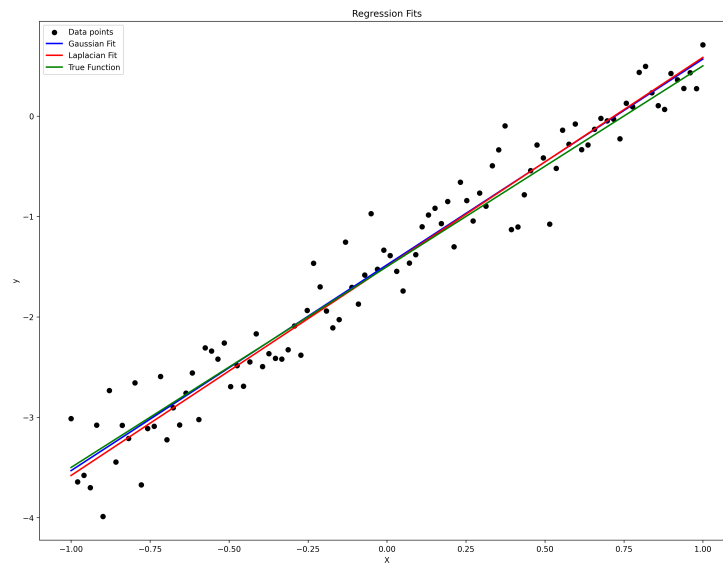


Figura 25: Comparação dos ajustes de regressão sem outliers

11.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

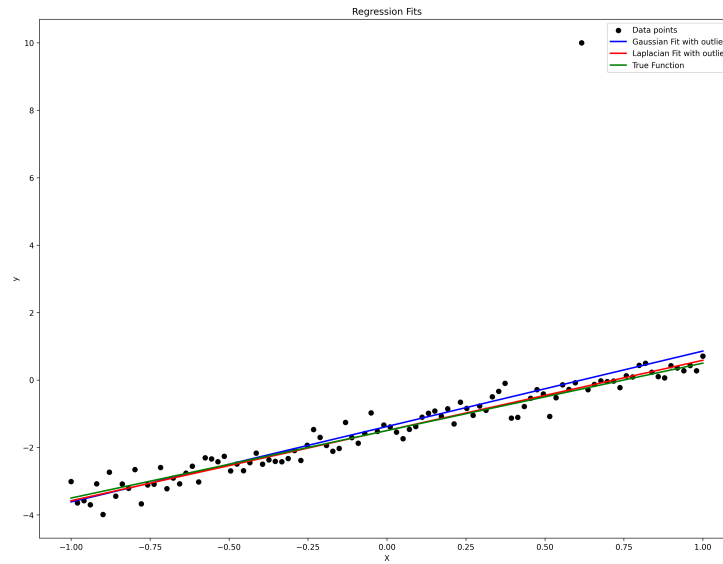


Figura 26: Comparação dos ajustes de regressão com outlier

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.

12 Exercício 3f

12.1 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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

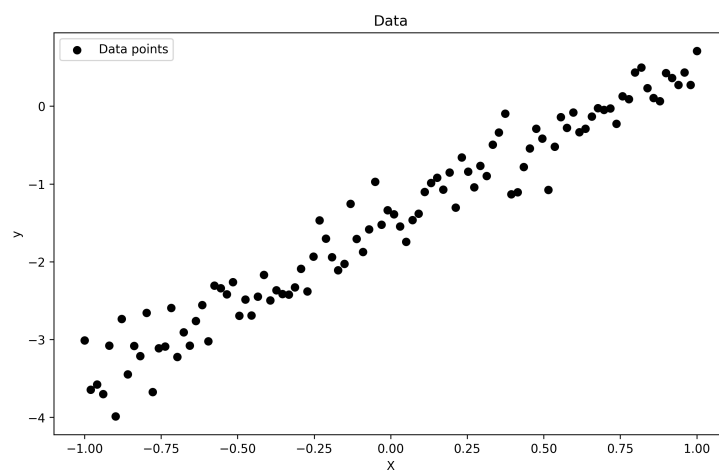


Figura 27: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

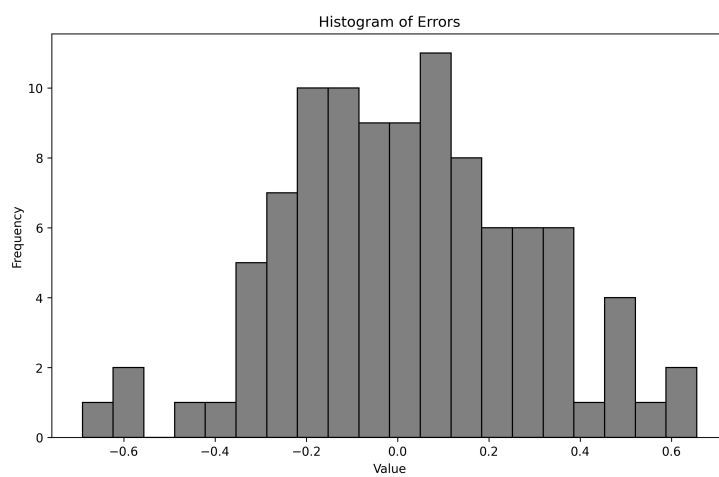


Figura 28: Histograma dos erros

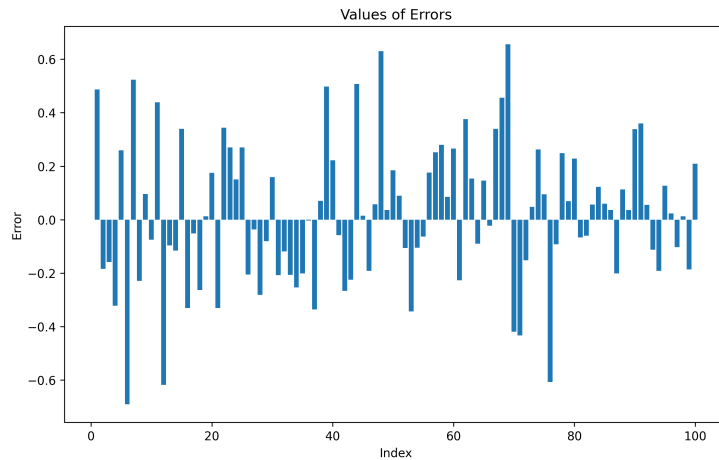


Figura 29: Valores dos erros por índice

12.2 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 ?? compara visualmente os ajustes:

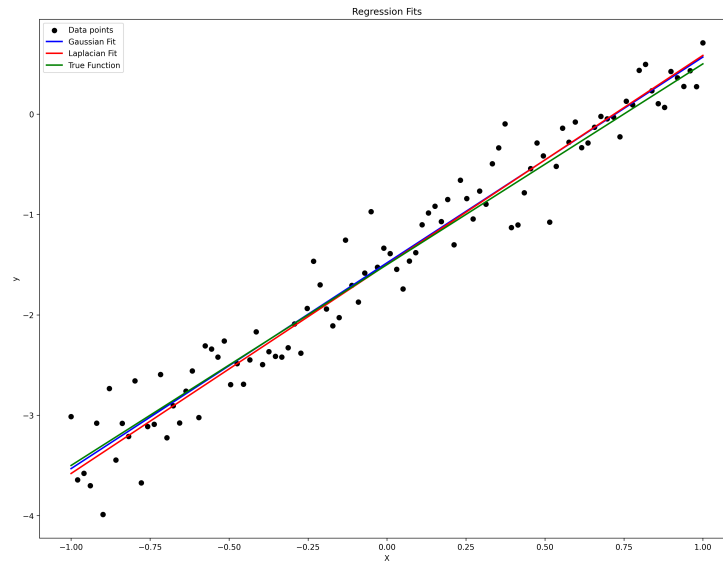


Figura 30: Comparação dos ajustes de regressão sem outliers

12.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

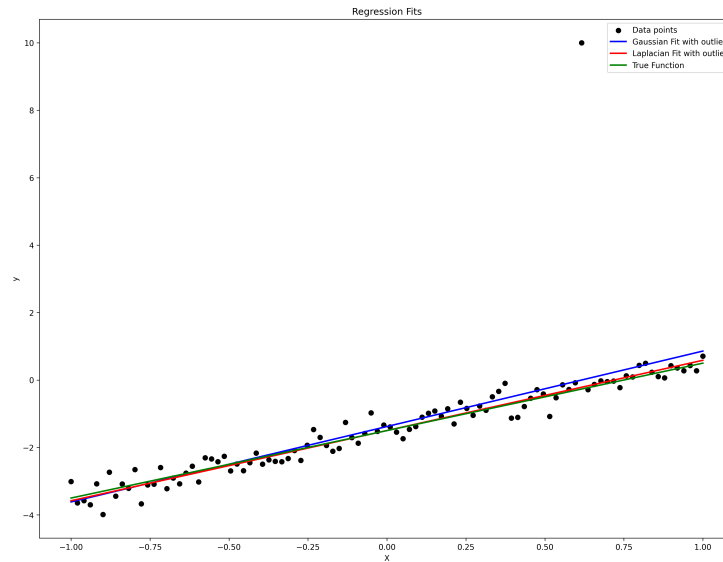


Figura 31: Comparação dos ajustes de regressão com outlier

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.

13 Exercício 3f

13.1 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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

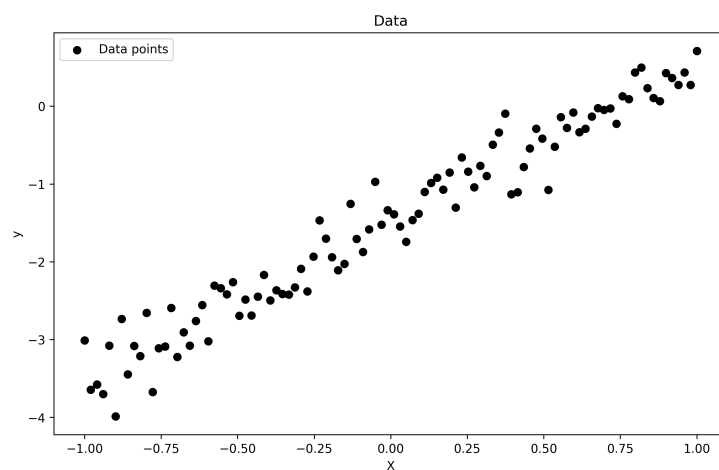


Figura 32: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

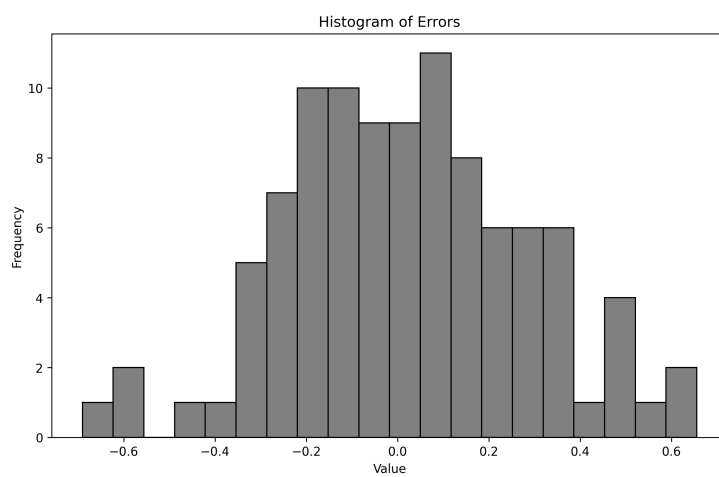


Figura 33: Histograma dos erros



Figura 34: Valores dos erros por índice

13.2 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 ?? compara visualmente os ajustes:

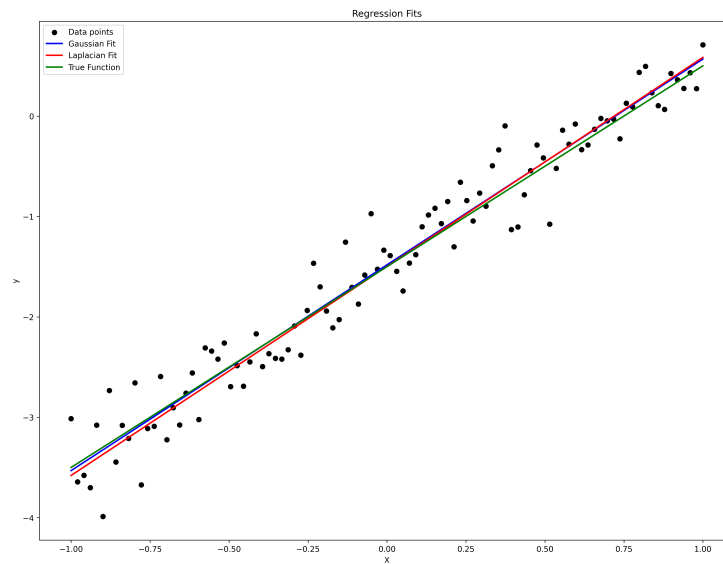


Figura 35: Comparação dos ajustes de regressão sem outliers

13.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

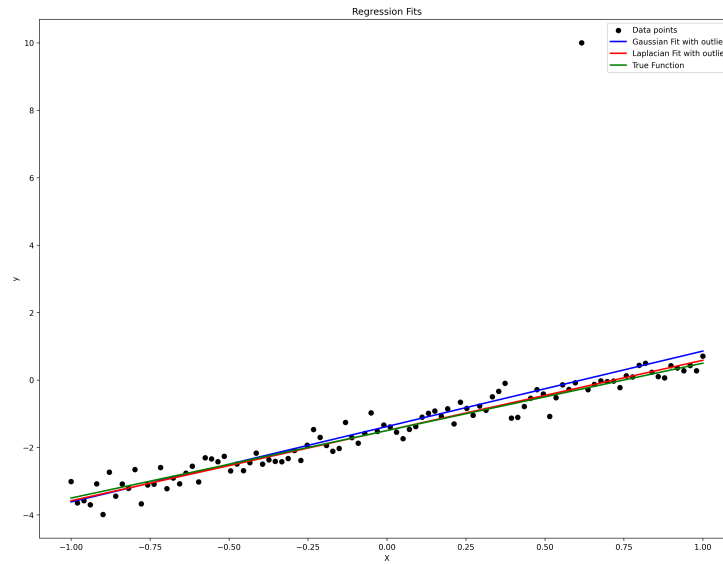


Figura 36: Comparação dos ajustes de regressão com outlier

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.

14 Exercício 3f

14.1 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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

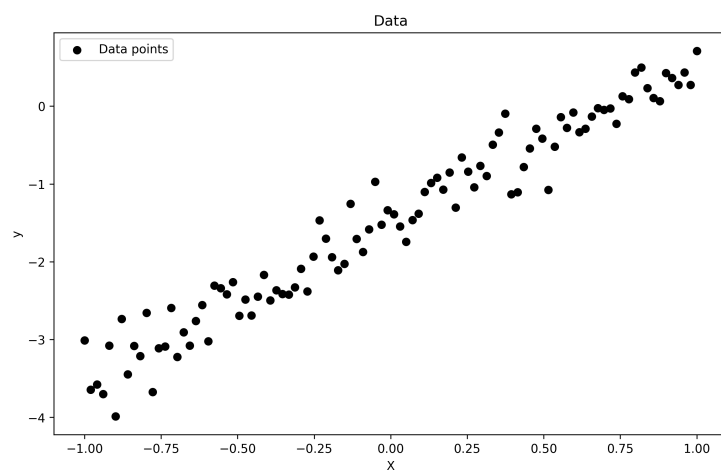


Figura 37: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

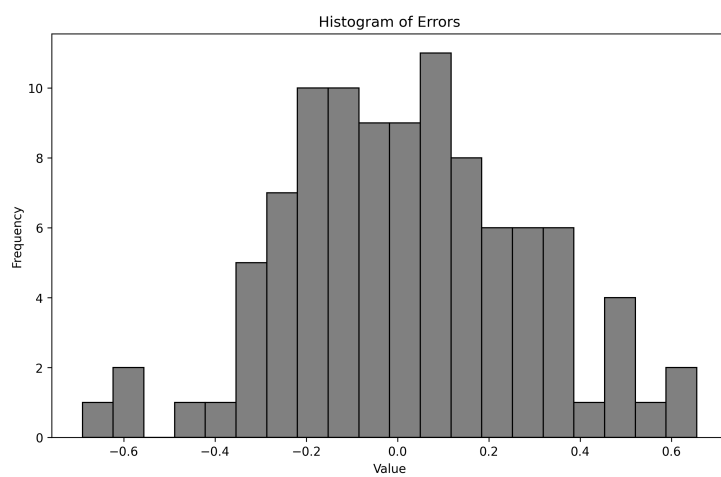


Figura 38: Histograma dos erros



Figura 39: Valores dos erros por índice

14.2 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 ?? compara visualmente os ajustes:

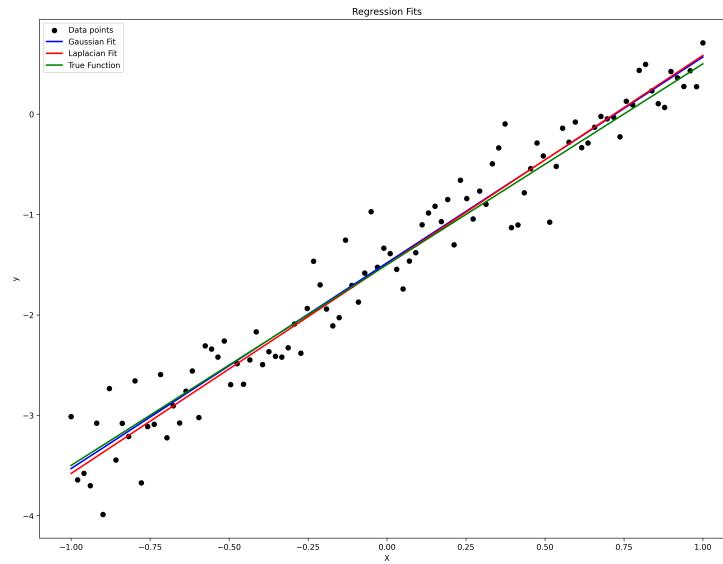


Figura 40: Comparação dos ajustes de regressão sem outliers

14.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

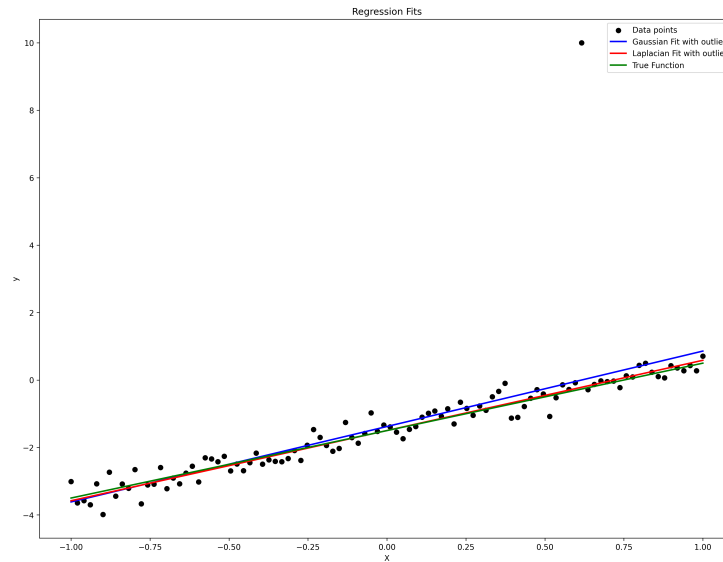


Figura 41: Comparação dos ajustes de regressão com outlier

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.

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:

```
1 import numpy as np
2 import scipy
3 import matplotlib.pyplot as plt
4
5 def beta_ordinary(X: np.ndarray, Y: np.ndarray) -> np.
  ndarray:
6     """
7     Compute the ordinary least squares estimator.
8     """
9     beta = np.linalg.inv(X.T @ X) @ X.T @ Y
10    return beta
11
12 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
  error_distribution: str) -> np.ndarray:
13    """
14    Compute the estimator beta_hat based on the specified
  error distribution.
15    """
16    np.random.seed(0)
17    p = X.shape[1]
18
19    if error_distribution == "gaussian":
20        def loss_function(beta):
21            return np.sum((Y - X @ beta) ** 2)
22    elif error_distribution == "laplacian":
23        def loss_function(beta):
24            return np.sum(np.abs(Y - X @ beta))
25    else:
26        raise ValueError("Unsupported error distribution")
27
28    beta_0 = np.random.uniform(size=p)
29    beta_hat = scipy.optimize.minimize(loss_function, beta_0
  )
30    return beta_hat["x"]
```

Geramos dados sintéticos para testar os estimadores:

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

A Figura ?? mostra os dados gerados:

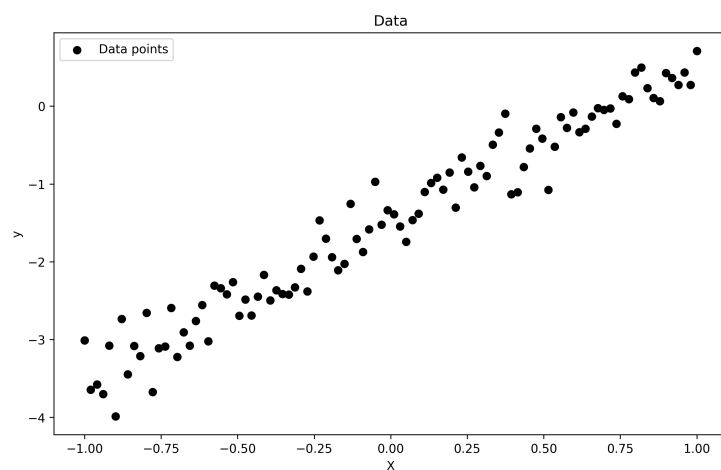


Figura 42: Dados sintéticos para comparação dos estimadores

As Figuras ?? e ?? mostram a análise dos erros:

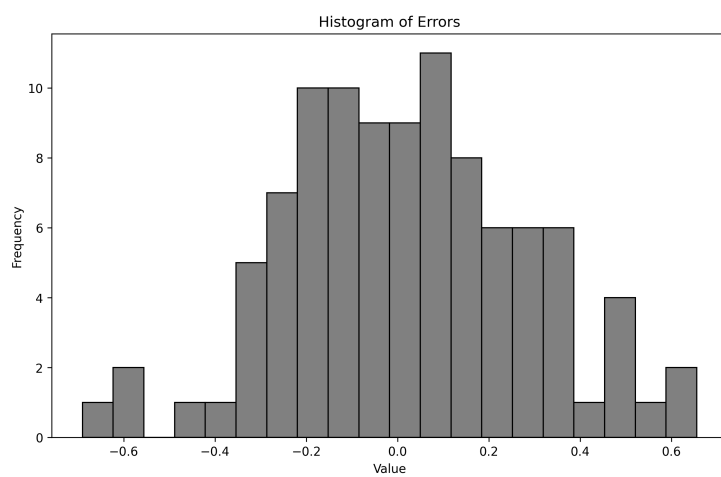


Figura 43: Histograma dos erros



Figura 44: 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 ?? compara visualmente os ajustes:

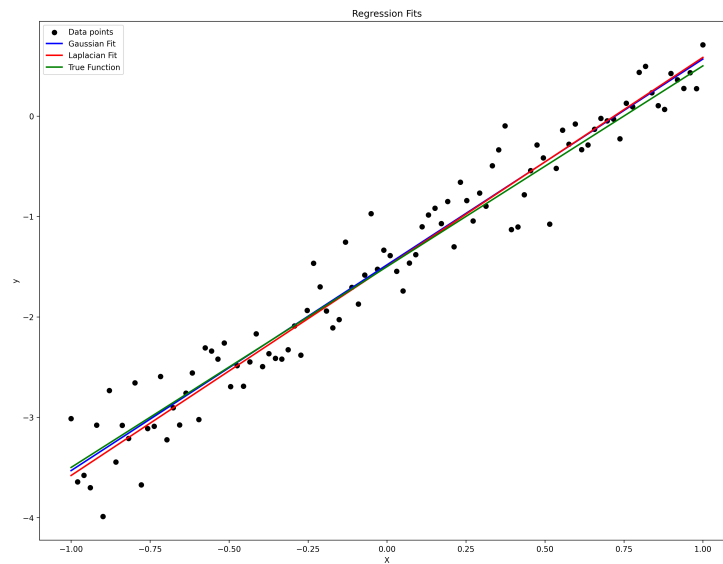


Figura 45: Comparação dos ajustes de regressão sem outliers

15.3 iii) Robustez a Outliers

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

```
1 # Regenerate data and add outlier
2 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 ?? mostra o impacto do outlier:

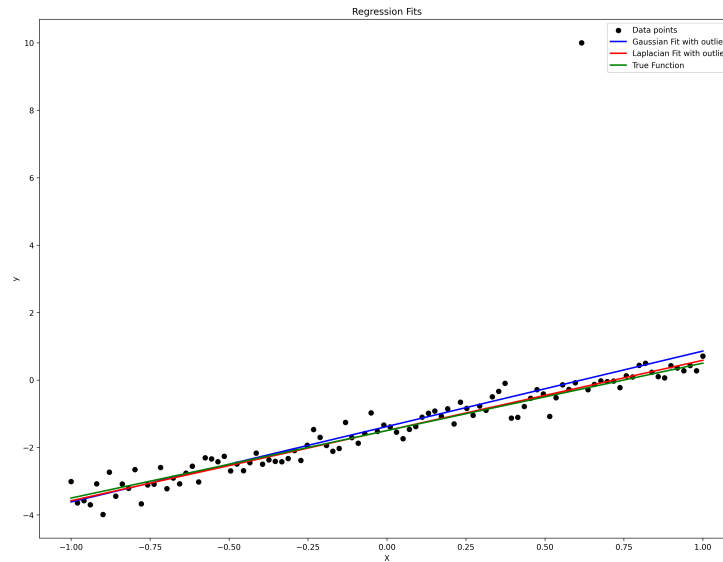


Figura 46: Comparação dos ajustes de regressão com outlier

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

16.1 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)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from sklearn.discriminant_analysis import
5     LinearDiscriminantAnalysis as LDA
6 from sklearn.discriminant_analysis import
7     QuadraticDiscriminantAnalysis as QDA
8 from sklearn.linear_model import LogisticRegression as LR
9 from sklearn.naive_bayes import GaussianNB as NB
10 from sklearn.neighbors import KNeighborsClassifier as kNN
11 from sklearn import preprocessing
12
13 # Load and prepare data
14 df = pd.read_csv("../data/soccer.csv")
15 X = df.drop("target", axis=1)
16 y = df[["target"]]
17
18 # Split dataset
19 X_train, y_train = X.iloc[:2560], y.iloc[:2560]
20 X_test, y_test = X.iloc[2560:], y.iloc[2560:]
21
22 # Remove categorical variables and standardize
23 X_train = X_train.drop(["home_team", "away_team"], axis=1)
24 X_test = X_test.drop(["home_team", "away_team"], axis=1)
25 scaler = preprocessing.StandardScaler()
26 X_train = scaler.fit_transform(X_train)
27 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 Exercício 4b

17.1 Treinamento e Avaliação dos Modelos

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

```
1 models_to_test = [LDA, QDA, LR, NB, kNN]
2 results_dict = {}
3
4 for model_type in models_to_test:
5     model_name = model_type.__name__
6     params = {}
7     if model_type in [LDA, QDA]:
8         params.update({"store_covariance": True})
9
10    results_dict[model_name] = {}
11    cls = model_type(**params)
12    cls.fit(X_train, y_train.values.ravel())
13
14    # Store predictions and model
15    results_dict[model_name]["in_sample_predictions"] = cls.
predict(X_train)
16    results_dict[model_name]["test_predictions"] = cls.
predict(X_test)
17    results_dict[model_name]["model"] = cls
```

17.2 Comparação de Performance dos Modelos

17.3 Análise Geral dos Algoritmos

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

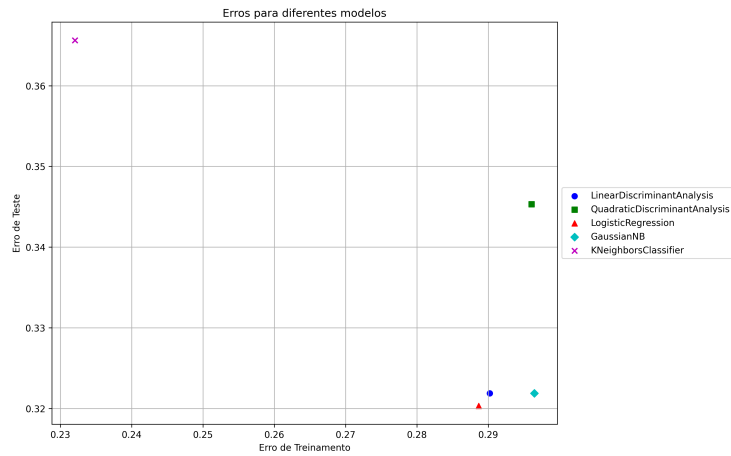


Figura 47: Comparação dos erros de treinamento vs teste para diferentes modelos

17.4 Análise Específica do k-NN

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

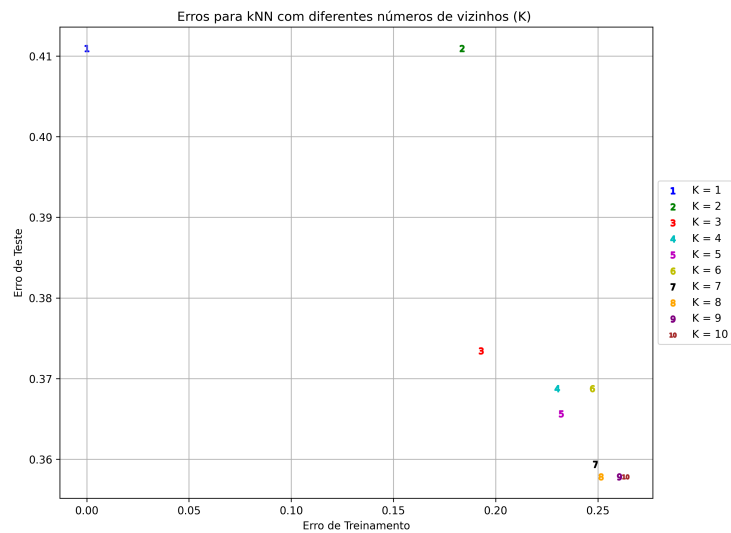


Figura 48: Erros do k-NN para diferentes números de vizinhos (K=1 a K=10)

17.5 Análise dos Coeficientes dos Modelos

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

17.6 Conclusões

1. **Performance Geral:** Todos os modelos apresentaram performance similar, sugerindo que o problema tem estrutura linear bem definida.
2. **Overfitting:** O k-NN com $K=1$ mostra clear overfitting (erro de treino muito baixo, erro de teste alto), enquanto valores maiores de K generalizam melhor.
3. **k-NN:** A performance otimizada ocorre com K entre 3-7, balanceando bias e variância.

18 Exercício 4c

19 Exercício 4d

20 Exercício 5b

20.1 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

20.2 Preparação dos Dados

```
1 import statsmodels.api as sm
2 import numpy as np
3 import pandas as pd
4 from sklearn.model_selection import train_test_split, KFold
5 from sklearn.linear_model import Lasso
6 from sklearn import preprocessing
7
8 # Load and prepare data
9 bodyfat = pd.read_csv("../data/bodyfat.csv")
10 X = bodyfat.drop(columns=["BodyFat", "Density"])
11 y = bodyfat["BodyFat"]
12
13 # Split data
14 X_train, X_test, y_train, y_test = train_test_split(
15     X, y, test_size=0.2, random_state=10
16 )
17
18 # Setup cross-validation
19 kf = KFold(n_splits=5, shuffle=True, random_state=10)
```

20.3 Implementação dos Algoritmos

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

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 Preparação dos Dados

```
1 import statsmodels.api as sm
2 import numpy as np
3 import pandas as pd
4 from sklearn.model_selection import train_test_split, KFold
5 from sklearn.linear_model import Lasso
6 from sklearn import preprocessing
7
8 # Load and prepare data
9 bodyfat = pd.read_csv("../data/bodyfat.csv")
10 X = bodyfat.drop(columns=["BodyFat", "Density"])
11 y = bodyfat["BodyFat"]
12
13 # Split data
14 X_train, X_test, y_train, y_test = train_test_split(
15     X, y, test_size=0.2, random_state=10
16 )
17
18 # Setup cross-validation
19 kf = KFold(n_splits=5, shuffle=True, random_state=10)
```

21.3 Implementação dos Algoritmos

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

22 Exercício 5c

22.1 Comparação dos Métodos - R^2

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

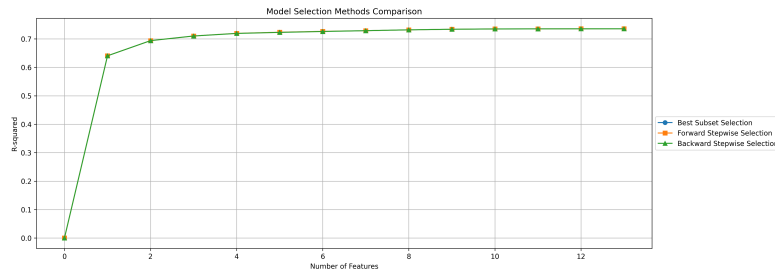


Figura 49: Comparação dos métodos de seleção de modelos - R^2 vs Número de Features

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

```
1 # Cross-validation for Lasso
2 alphas = 10 ** np.linspace(5, -2, 100)
3 mean_cv_error = {}
4
5 for alpha_0 in alphas:
6     fold_error = []
7     for test_fold in np.unique(cv_fold):
8         # Split data for current fold
9         x_train_fold = X_train[cv_fold != test_fold]
10        y_train_fold = y_train[cv_fold != test_fold]
11        x_test_fold = X_train[cv_fold == test_fold]
12        y_test_fold = y_train[cv_fold == test_fold]
13
14        # Normalize data
15        x_train_fold, x_test_fold = normalize_data(
16            x_train_fold, x_test_fold)
17
18        # Train and evaluate Lasso model
19        model_ = Lasso(alpha=alpha_0).fit(x_train_fold,
20            y_train_fold)
21        yhat = model_.predict(x_test_fold)
22        fold_error.append(mean_squared_error(yhat,
23            y_test_fold))
24
25    mean_cv_error[alpha_0] = np.mean(fold_error)
26
27 best_alpha = min(mean_cv_error, key=mean_cv_error.get)
```

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

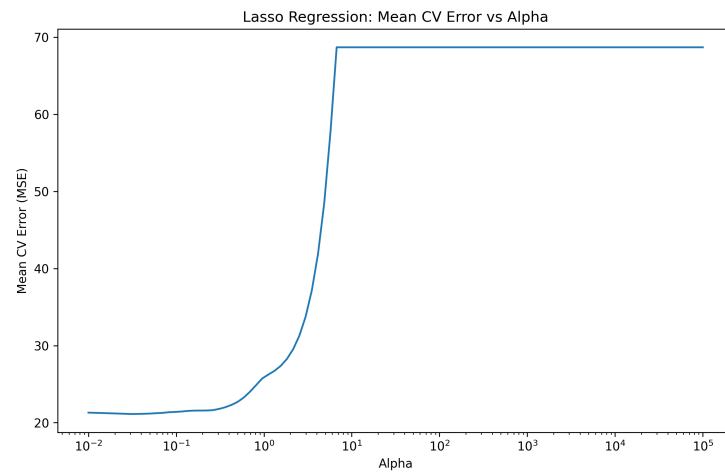


Figura 50: Lasso Regression: Erro de Validação Cruzada vs Parâmetro Alpha

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:

```
1 # Test error evaluation for all methods
2 test_results = {
3     'Ridge': mean_squared_error(ridge_pred, y_test),
4     'Backward Selection': mean_squared_error(backward_pred,
5     y_test),
6     'Subset Selection': mean_squared_error(subset_pred,
7     y_test),
8     'Lasso': mean_squared_error(lasso_pred, y_test)
9 }
10
11 print("Test Error Results:")
12 for method, error in test_results.items():
13     print(f"{method}: {error:.4f}")
```

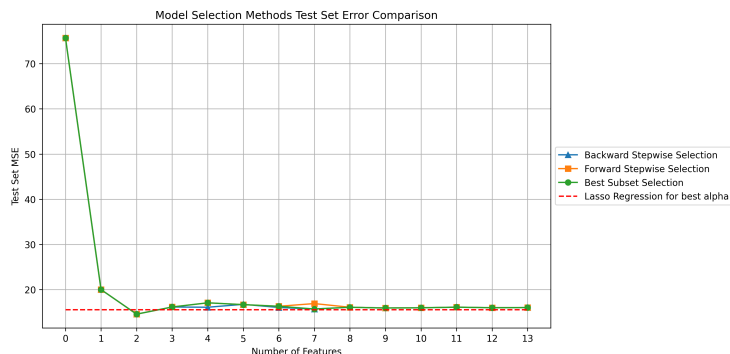


Figura 51: Comparação dos Erros de Teste para Todos os Métodos

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