

List 01 - Machine Learning

IMPA

Pedro Bahia

23 de janeiro de 2026

Conteúdo

1 Exercício 1a	3
2 Exercício 1b	4
3 Exercício 1c	5
4 Exercício 1d	6
5 Exercício 1e	7
6 Exercício 2a	8
7 Exercício 2b	9
7.1 i	9
7.2 ii	9
7.3 iii	9
7.4 iv	10
8 Exercício 2c	11
8.1 i	11
8.2 ii	11
8.3 iii	12
9 Exercício 2d	13
9.1 i	13
9.2 ii	15
9.3 iii	16
9.4 iv	17
9.5 v	17
10 Exercício 3a	19

11 Exercício 3b	20
12 Exercício 3c	21
13 Exercício 3d	22
14 Exercício 3e	23
14.1 i	23
14.2 ii	23
14.3 iii	23
15 Exercício 3f	24
15.1 i) Implementação dos Estimadores	24
15.2 ii) Comparação dos Estimadores	26
15.3 iii) Robustez a Outliers	27
16 Exercício 4a	29
17 Exercício 4b	30
17.1 Implementação dos Algoritmos de Classificação	30
17.2 Treinamento e Avaliação dos Modelos	31
18 Exercício 4c	32
19 Exercício 4d	33
19.1 Análise Específica do k-NN	33
20 Exercício 5a	34
21 Exercício 5b	35
21.1 Métodos de Seleção de Modelos	35
21.2 Implementação dos Algoritmos	35
22 Exercício 5c	44
22.1 Comparação dos Métodos - R^2	44
23 Exercício 5d	45
23.1 Regressão Lasso com Validação Cruzada	45
23.2 Seleção do Parâmetro de Regularização	45
24 Exercício 5e	47
24.1 Comparação de Erros de Teste	47
24.2 Ranking dos Métodos	47
24.3 Análise dos Resultados	48
25 Exercício 5f	49

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 melhorias.

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

$$1 - \text{epsilon}$$

, 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 acurácia dos Modelos 1 e 2 não é possível determinar qual modelo tem menor erro em transações fraudulentas.

4 Exercício 1d

Verdadeiro

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

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

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:

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

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 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,

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

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:

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

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 trasposta multiplicando pela direita. Além disso, sabemos que $\mathbb{V}(Y) = \varepsilon = \Sigma$, temos:

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

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 raízes 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:

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

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

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

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

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

9 Exercício 2d

9.1 i

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

```
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 + 1)])
6 np.random.seed(0)
7 X = np.array([np.ones(n), np.random.normal(0, 1, n)]).T
8 beta = np.array([1, 0.25])
9 epsilon = np.random.multivariate_normal(np.zeros(n), Sigma)
10 y = X @ beta + epsilon
```

A Figura 1 mostra os dados gerados com heterocedasticidade:

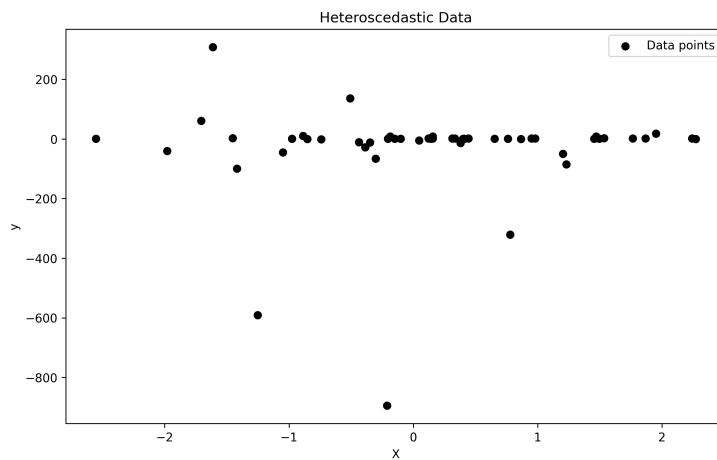


Figura 1: Dados heteroscedásticos gerados

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

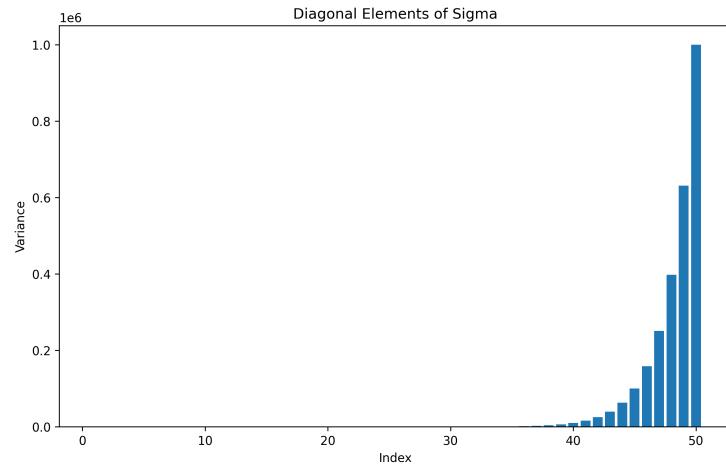


Figura 2: Elementos diagonais da matriz Σ

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

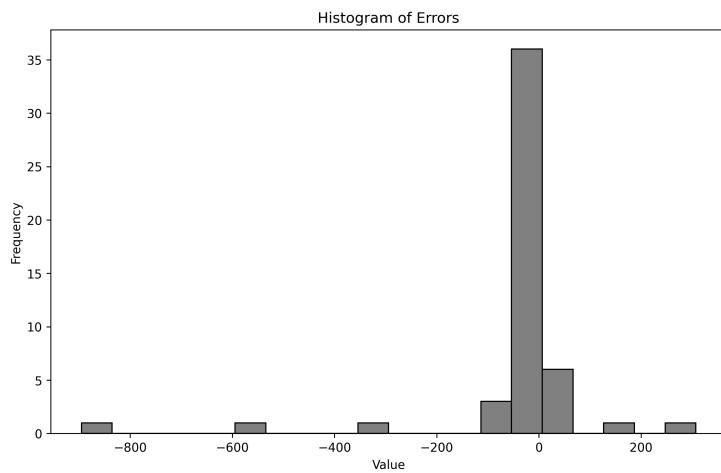


Figura 3: Histograma dos erros

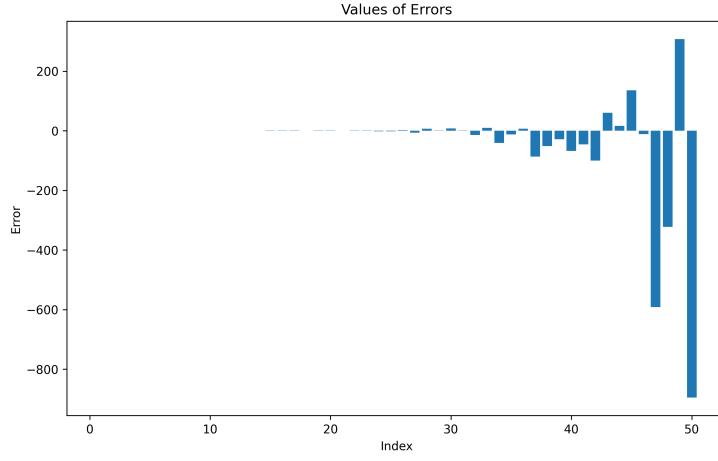


Figura 4: Valores dos erros por índice

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:

```

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

```

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

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

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

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:

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

```

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

$$Y_i|X_i \sim \text{Laplace}(\beta^T x_i, b)$$

$$\epsilon \sim \text{Laplace}(0, b)$$

$$Y_i|X_i = Y = \beta^T X + \epsilon$$

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

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

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 $\text{Laplace}(\beta^T X, b)$.

Função de perda (likelihood):

Dado que Y é iid, temos que:

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

11 Exercício 3b

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

Como $n \log \frac{1}{2b}$ é constante em relação a β , é possível retirar essa parte da função de perda para otimização. O mesmo pode ser dito para o fator $\frac{1}{b}$.

12 Exercício 3c

$$\begin{aligned}\epsilon &\stackrel{iid}{\sim} \mathcal{N}(0, 1) \\ Y &= X\beta + \epsilon\end{aligned}$$

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

$$\begin{aligned}\mathcal{L}(y|\mu, \sigma^2) &= -\log(\ell(y|\mu, \sigma^2)) \\ &= -\log \left[\frac{1}{\sigma^n (\sqrt{2\pi})^n} \cdot e^{-\frac{1}{2\sigma^2} \sum_i^n (y_i - X_i\beta)^2} \right] \\ &= \log(\sigma^n (\sqrt{2\pi})^n) + \frac{1}{2\sigma^2} \sum_i^n (y_i - X_i\beta)^2\end{aligned}$$

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

Vantagem: em caso de outliers, em que o erro do valor estimado muito alto, o termo associado à será elevado ao quadrado, resultando em maiores impactos na função de perda. Isso não acontece com a função de perda da distribuição Laplace, onde o erro é considerado linearmente. Assim, a função de perda baseada na distribuição Laplace é mais robusta a outliers.

13 Exercício 3d

A regra de atualização do gradiente descendente para a função de perda baseada na distribuição Laplace é dada por:

$$\begin{aligned}\beta^{(t+1)} &= \beta^{(t)} - \eta \nabla_{\beta} \mathcal{L}(\beta^{(t)}) \\&= \beta^{(t)} - \eta \left(\sum_{i=1}^n \text{sign}(Y_i - X_i^T \beta^{(t)}) X_i \right) \\&= \beta^{(t)} + \eta \sum_{i=1}^n \text{sign}(Y_i - X_i^T \beta^{(t)}) X_i\end{aligned}$$

14 Exercício 3e

14.1 i

O modelo com maior passo será o Laplaciano pois o gradiente da função gaussiana proporcional ao erro, enquanto o gradiente da função Laplaciana não é. Para ela, O erro define apenas o sinal do gradiente, e não sua magnitude.

14.2 ii

Para um mesmo beta e uma amostra de treino, a direção de ambos os gradientes será a mesma, já que ambos os gradientes dependem apenas do sinal do erro.

Já para novas iterações, a direção pode ser diferente já que ambos os gradientes terão magnitude diferente e resultarão em atualizações diferentes de beta.

14.3 iii

Para o caso com mais de uma amostra, a direção do gradiente gaussiano será dada pela soma ponderada de cada erro, enquanto a direção do gradiente Laplaciano será dada pela soma dos sinais de cada erro. Assim, a direção do gradiente pode divergir entre os dois modelos, dependendo dos erros de cada amostra. Por exemplo, se uma amostra tiver um erro muito alto para uma direção enquanto outras duas tiverem erros baixos na direção oposta, o gradiente gaussiano tenderá a seguir a direção da amostra com maior erro, enquanto o gradiente Laplaciano tenderá a seguir a direção das duas amostras com erros menores.

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.
6     ndarray:
7         """
8             Compute the ordinary least squares estimator.
9         """
10        beta = np.linalg.inv(X.T @ X) @ X.T @ Y
11        return beta
12
13 def calculate_beta_hat(X: np.ndarray, Y: np.ndarray,
14     error_distribution: str) -> np.ndarray:
15     """
16         Compute the estimator beta_hat based on the specified
17         error distribution.
18     """
19     np.random.seed(0)
20     p = X.shape[1]
21
22     if error_distribution == "gaussian":
23         def loss_function(beta):
24             return np.sum((Y - X @ beta) ** 2)
25     elif error_distribution == "laplacian":
26         def loss_function(beta):
27             return np.sum(np.abs(Y - X @ beta))
28     else:
29         raise ValueError("Unsupported error distribution")
30
31     beta_0 = np.random.uniform(size=p)
32     beta_hat = scipy.optimize.minimize(loss_function, beta_0)
33
34     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 5 mostra os dados gerados:

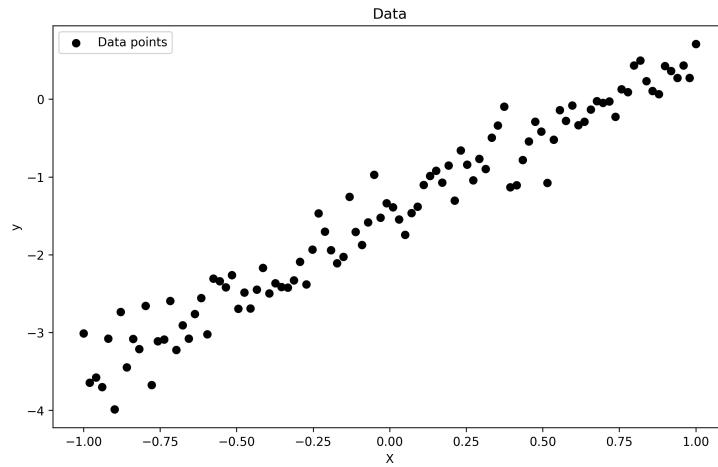


Figura 5: Dados sintéticos para comparação dos estimadores

As Figuras 6 e 7 mostram a análise dos erros:

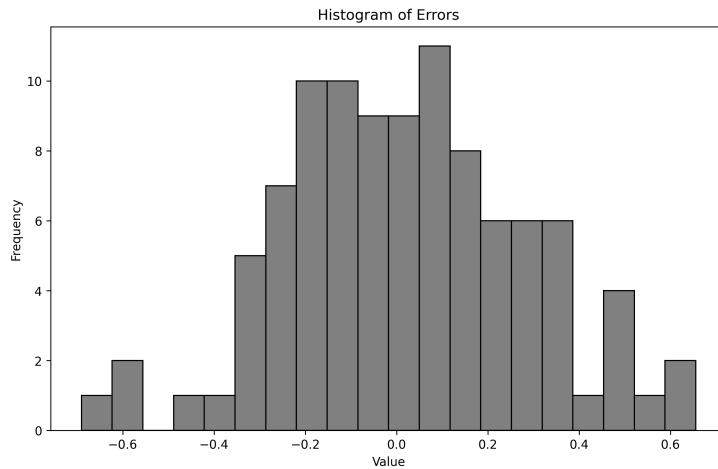


Figura 6: Histograma dos erros

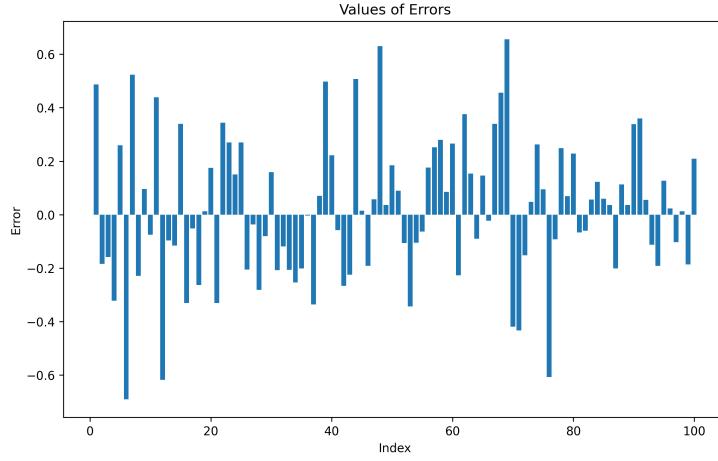


Figura 7: Valores dos erros por índice

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:

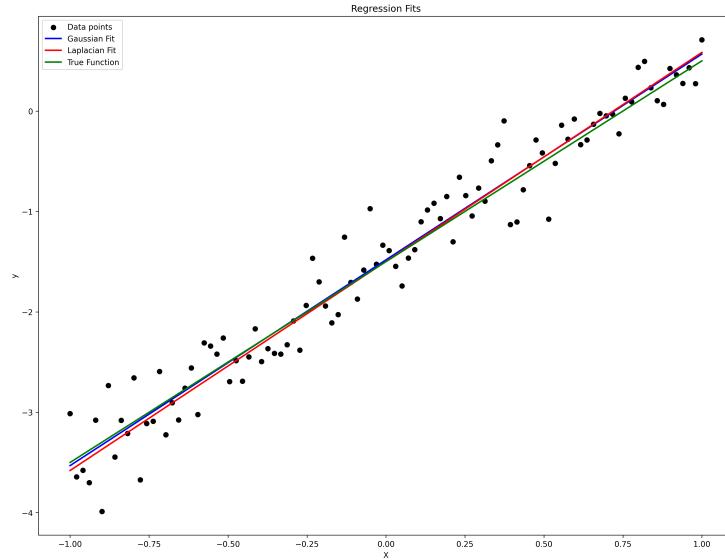


Figura 8: Comparação dos ajustes de regressão sem outliers

15.3 iii) Robustez a Outliers

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

```
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 9 mostra o impacto do outlier:

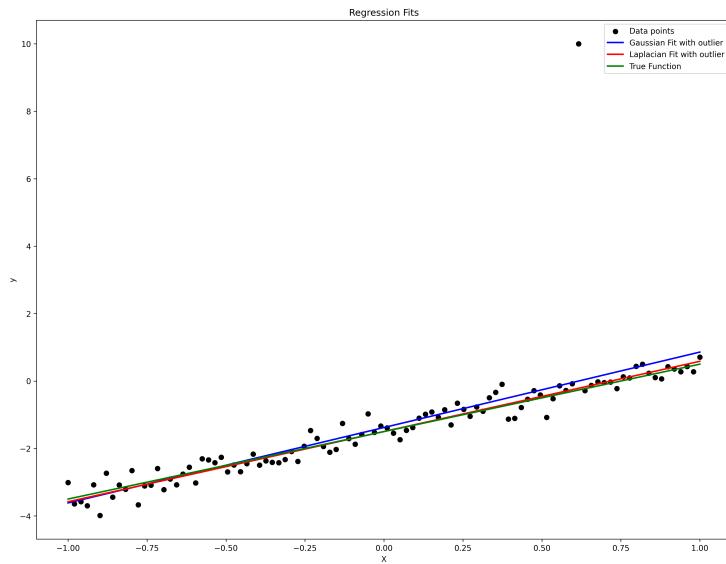


Figura 9: Comparaçāo dos ajustes de regressāo com outlier

Conclusões:

1. **Eficiēcia:** Quando os erros sāo gaussianos e nāo hā outliers, o estimador de mŕimos 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 eficiēcia (mŕimos quadrados) e robustez (estimador laplaciano). A escolha depende das caracterŕicas esperadas dos dados.
4. **Aplicaçāo Prática:** Em situações onde outliers sāo esperados ou a distribuiçāo dos erros tem caudas pesadas, o estimador laplaciano (regressāo 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)

```
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)

17.2 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.
16    predict(X_train)
17    results_dict[model_name]["test_predictions"] = cls.
18    predict(X_test)
19    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:

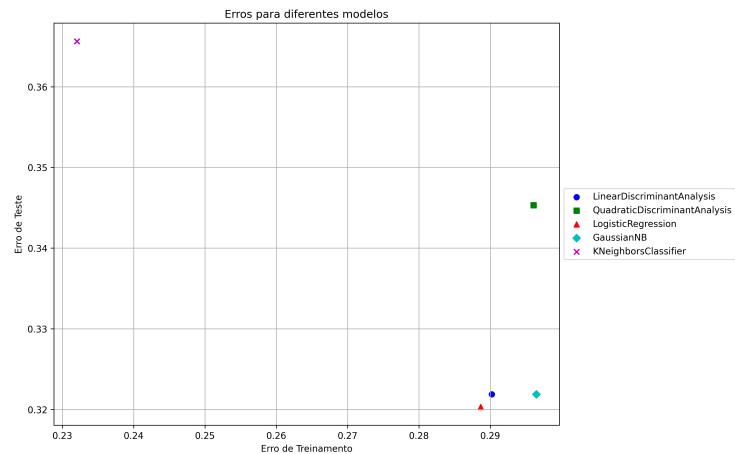


Figura 10: Comparação dos erros de treinamento vs teste para diferentes modelos

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 :

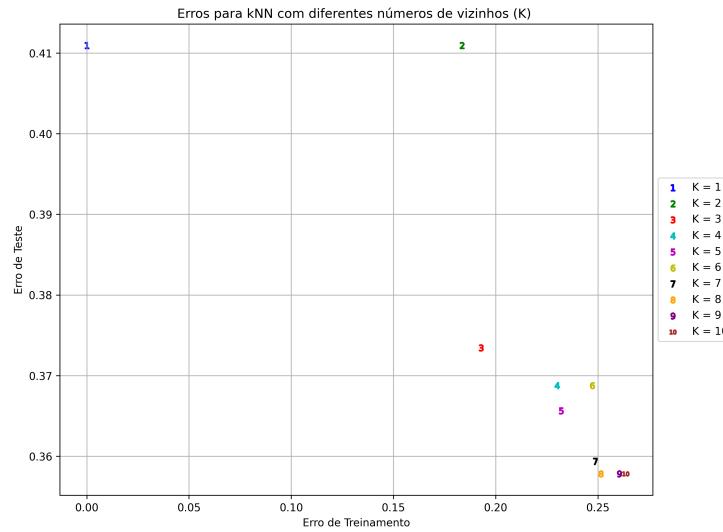


Figura 11: Eros do k-NN para diferentes números de vizinhos ($K=1$ a $K=10$)

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

Dentre os modelos analisados, o Lasso necessita que as variáveis sejam padronizadas para que o modelo funcione corretamente. Isso ocorre porque o Lasso aplica uma penalização baseada na soma dos valores absolutos dos coeficientes, o que pode levar a uma seleção inadequada de variáveis se elas estiverem em escalas diferentes. Variáveis com escalas maiores podem dominar a penalização, resultando em coeficientes distorcidos e uma seleção de variáveis que não reflete a importância real de cada variável no modelo.

Na função de custo, isso se traduz em uma constante proporcional ao valor da variável escalonada, que entraria dentro do somatório da penalização L1, afetando a otimização dos coeficientes.

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.

```
1     def best_subset_selection(
2         X_train: np.ndarray, Y_train: np.ndarray
3     ) -> dict:
4         """
5             Perform best subset selection for linear regression.
6             This function evaluates all possible combinations of
7             features
8             and selects the best model for each subset size based on
9             R-squared.
10
11            Parameters:
12                - X_train (np.ndarray): Training feature data. Shape (
13                    n_samples, n_features).
14                - Y_train (np.ndarray): Training target data. Shape (
15                    n_samples,).
16
17            Returns:
18                - dict: A dictionary where each key is the subset size (
19                    as a string) and the value is another dictionary
20                    containing:
21                        - 'best_model': The statsmodels OLS regression
22                            results object for the best model.
23                        - 'best_r2': The R-squared value of the best model.
24                        - 'best_features': A tuple of feature indices used
25                            in the best model.
26
27
28            """
29
30        # Useful dimensions
```

```

22     n, p = X_train.shape
23
24     # Initialize dictionary to store the best model for each
25     # subset size
26     best_model_k = {}
27
28     # Create model for no feature
29     best_model_k["0"] = {}
30     best_model_k["0"]["best_model"] = sm.OLS(
31         Y_train, np.ones((len(Y_train), 1))
32     ).fit()
33     best_model_k["0"]["best_r2"] = best_model_k["0"]["best_model"].rsquared
34     best_model_k["0"]["best_features"] = []
35     tests_made = [()]
36
37     for k in range(1, p + 1):
38         best_r2 = 0
39         best_features = []
40         best_model = []
41
42         # Create every model with k features (excluding
43         # intercept)
44         for feature_combination in itertools.combinations(
45             range(p), k):
46             X_train_aux = sm.add_constant(X_train[:, feature_combination])
47             model = sm.OLS(Y_train, X_train_aux).fit()
48             tests_made.append(feature_combination)
49             if model.rsquared > best_r2:
50                 best_r2 = model.rsquared
51                 best_model = model
52                 best_features = feature_combination
53
54         # Store the best model for this subset size
55         best_model_k[str(k)] = {
56             "best_model": best_model,
57             "best_r2": best_r2,
58             "best_features": list(best_features),
59         }
60
61     # Sanity checks
62     assert len(tests_made) == 2**p, (
63         "Number of feature subsets evaluated does not match
64         expected count."
65     )
66     assert len(tests_made) == len(set(tests_made)), (
67         "Duplicate feature subsets were evaluated."
68     )
69     assert len(best_model_k.keys()) == p + 1, (

```

```

66     "Best model dictionary does not contain expected
67     number of entries."
68 )
69 assert not (
70     any(
71         [
72             best_model_k[k]["best_model"] == []
73             for k in best_model_k.keys()
74         ]
75     )
76 ), "Not all best models are valid."
77
78
79
80 # %%
81 def forward_stepwise_selection(
82     X_train: np.ndarray, Y_train: np.ndarray
83 ) -> dict:
84     """
85         Function to perform forward stepwise selection for
86         linear regression. This function iteratively adds
87         features
88         to the model and selects the best model for each subset
89         size based on R-squared. Only one feature is added at
90         each step.
91
92
93     Parameters:
94         - X_train (np.ndarray): Training feature data. Shape (
95             n_samples, n_features).
96         - Y_train (np.ndarray): Training target data. Shape (
97             n_samples,).
98
99     Returns:
100         - dict: A dictionary where each key is the subset size (
101             as a string) and the value is another dictionary
102             containing:
103                 - 'best_model': The statsmodels OLS regression
104                     results object for the best model.
105                     - 'best_r2': The R-squared value of the best model.
106                     - 'best_features': A tuple of feature indices used
107                         in the best model.
108
109
110     """
111
112     # Useful dimensions
113     n, p = X_train.shape

```

```

105     # Initialize dictionary to store the best model for each
106     # subset size
107     best_model_k = {}
108
109     # Create model for no feature
110     best_model_k["0"] = {}
111     best_model_k["0"]["best_model"] = sm.OLS(
112         Y_train, np.ones((len(Y_train), 1))
113     ).fit()
114     best_model_k["0"]["best_r2"] = best_model_k["0"]["best_model"].rsquared
115     best_model_k["0"]["best_features"] = []
116
117     # Variables to keep track of features
118     remaining_features = list(range(p))
119     current_features = []
120
121     for k in range(1, p + 1):
122         # Initialize variables for this step
123         tests_made = []
124
125         best_r2 = 0
126         best_features = []
127         best_model = []
128
129         # Create every model with k features (excluding
130         # intercept)
131         for new_feature in remaining_features:
132             feature_combination_list = current_features + [
133                 new_feature]
134             feature_combination_list.sort()
135
136             X_train_aux = sm.add_constant(
137                 X_train[:, feature_combination_list]
138             )
139             model = sm.OLS(Y_train, X_train_aux).fit()
140             tests_made.append(feature_combination_list)
141
142             # Check if this model is the best so far
143             if model.rsquared > best_r2:
144                 best_r2 = model.rsquared
145                 best_model = model
146                 best_features = feature_combination_list
147
148                 # Move best feature for k from the remainig_features
149                 # list to current_features
150                 remaining_features = list(
151                     set(remaining_features) - set(best_features)
152                 )
153                 remaining_features.sort()

```

```

150     current_features = list(set(best_features +
151         current_features))
152         current_features.sort()
153
154     # Store the best model for this subset size
155     best_model_k[str(k)] = {
156         "best_model": best_model,
157         "best_r2": best_r2,
158         "best_features": current_features.copy(),
159     }
160
161     assert len(tests_made) == p + 1 - k, (
162         "Number of feature subsets evaluated does not
163         match expected count."
164     )
165     assert len(tests_made) == len(
166         set(tuple(sorted(test)) for test in tests_made)
167     ), "Duplicate feature subsets were evaluated."
168     assert all([len(test) == k for test in tests_made]),
169     (
170         "Not all tested features added have size k."
171     )
172     assert (
173         best_model_k[str(k - 1)]["best_features"] in
174         tests_made[i]
175         for i in tests_made
176     ), "All tests must include previously selected
177     features."
178
179     assert len(best_model_k.keys()) == p + 1, (
180         "Best model dictionary does not contain expected
181         number of entries."
182     )
183     assert not (
184         any(
185             [
186                 best_model_k[k]["best_model"] == []
187                 for k in best_model_k.keys()
188             ]
189         )
190     ), "Not all best models are valid."
191
192     return best_model_k
193
194
195 # %%
196 def backward_stepwise_selection(
197     X_train: np.ndarray, Y_train: np.ndarray
198 ) -> dict:
199     """

```

```

194     Function to perform backward stepwise selection for
195     linear regression. This function iteratively removes
196     features
197
198     from the model and selects the best model for each
199     subset size based on R-squared. Only one feature is
200     removed at each step.
201
202
203     Parameters:
204     - X_train (np.ndarray): Training feature data. Shape (
205         n_samples, n_features).
206     - Y_train (np.ndarray): Training target data. Shape (
207         n_samples,).
208
209     Returns:
210     - dict: A dictionary where each key is the subset size (
211         as a string) and the value is another dictionary
212         containing:
213             - 'best_model': The statsmodels OLS regression
214                 results object for the best model.
215             - 'best_r2': The R-squared value of the best model.
216             - 'best_features': A tuple of feature indices used
217                 in the best model.
218
219     """
220
221     # Useful dimensions
222     n, p = X_train.shape
223
224     # Initialize dictionary to store the best model for each
225     # subset size
226     best_model_k = {}
227     all_features = list(range(p))
228
229     # Create model for all features
230     best_model_k[f"{p}"] = {}
231     best_model_k[f"{p}"]["best_model"] = sm.OLS(
232         Y_train, sm.add_constant(X_train)
233     ).fit()
234     best_model_k[f"{p}"]["best_r2"] = best_model_k[f"{p}"][
235         "best_model"
236     ].rsquared
237     best_model_k[f"{p}"]["best_features"] = list(
238         all_features)
239
240     # Variables to keep track of features
241     current_features = all_features
242     deleted_features = []

```

```

232     for k in range(1, p + 1):
233         # Initialize variables for this step
234         tests_made = []
235
236         best_r2 = 0
237         best_features = []
238         best_model = []
239
240         # Create every model with k features (excluding
241         # intercept)
242         for new_feature in current_features:
243             feature_combination_list = current_features.copy()
244
245             feature_combination_list.remove(new_feature)
246             X_train_aux = sm.add_constant(
247                 X_train[:, feature_combination_list])
248             model = sm.OLS(Y_train, X_train_aux).fit()
249             tests_made.append(feature_combination_list)
250
251             # Check if this model is the best so far
252             if model.rsquared > best_r2:
253                 best_r2 = model.rsquared
254                 best_model = model
255                 best_features = feature_combination_list
256
257             # Update current features and deleted features lists
258             deleted_features += list(
259                 set(current_features) - set(best_features))
260
261             current_features = best_features
262
263             # Store the best model for this subset size
264             best_model_k[str(p - k)] = {
265                 "best_model": best_model,
266                 "best_r2": best_r2,
267                 "best_features": current_features.copy(),
268             }
269
270             # Sanity checks
271             assert len(tests_made) == p - k + 1, (
272                 "Number of feature subsets evaluated does not
273                 match expected count."
274             )
275             assert len(tests_made) == len(
276                 set(tuple(sorted(test)) for test in tests_made)
277             ), "Duplicate feature subsets were evaluated."
278             assert all([len(test) == p - k for test in
279                 tests_made]), (
280                 "Not all tested features added have size p-k."

```

```

278     )
279     assert (
280         tests_made[i] in best_model_k[str(p - k + 1)]["best_features"]
281         for i in tests_made
282     ), "All tests must be included in previously
283     selected features."
284     assert len(deleted_features) == k, (
285         "Number of deleted features does not match
286     expected count."
287     )
288
289     # Create model for no feature
290     best_model_k["0"] = {}
291     best_model_k["0"]["best_model"] = sm.OLS(
292         Y_train, np.ones((len(Y_train), 1))
293     ).fit()
294     best_model_k["0"]["best_r2"] = best_model_k["0"]["best_model"].rsquared
295     best_model_k["0"]["best_features"] = []
296
297     # Sanity checks
298     assert len(best_model_k.keys()) == p + 1, (
299         "Best model dictionary does not contain expected
300     number of entries."
301     )
302     assert not (
303         any(
304             [
305                 best_model_k[k]["best_model"] == []
306                 for k in best_model_k.keys()
307             ]
308         )
309     ), "Not all best models are valid."
310
311     best_model_k = dict(
312         sorted(best_model_k.items(), key=lambda x: int(x[0]))
313     )
314
315     return best_model_k
316
317
318
319 # %%
320 models_backward = backward_stepwise_selection(
321     X_train.values, y_train.values
322 )

```

```
321 models_best = best_subset_selection(X_train.values, y_train.  
322     values)  
323 models_backward = backward_stepwise_selection(  
324     X_train.values, y_train.values  
325 )
```

22 Exercício 5c

22.1 Comparação dos Métodos - R^2

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

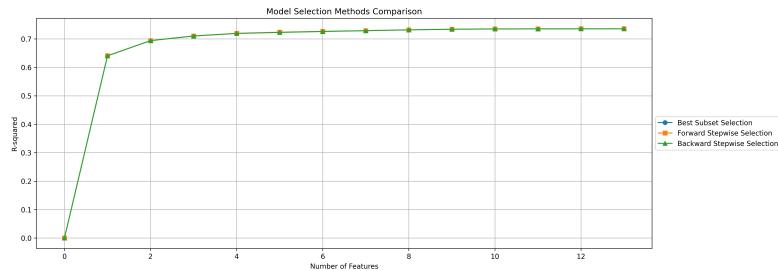


Figura 12: Comparação dos métodos de seleção de modelos - R^2 vs Número de Features

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

```
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 13 mostra a curva de validação cruzada para o Lasso:

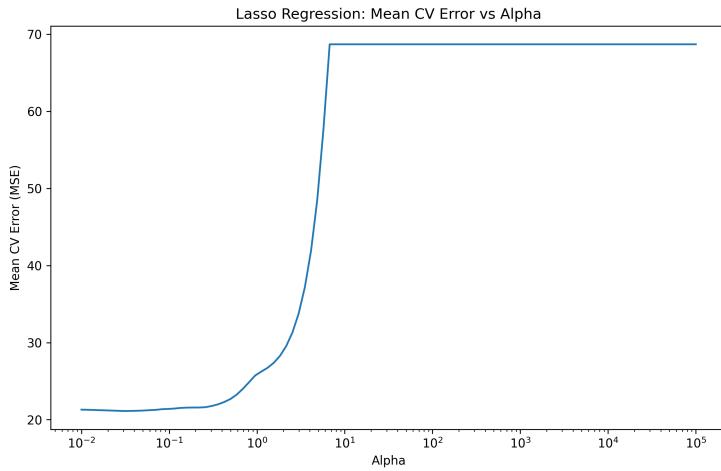


Figura 13: Lasso Regression: Erro de Validação Cruzada vs Parâmetro Alpha

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:

```
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}")
```

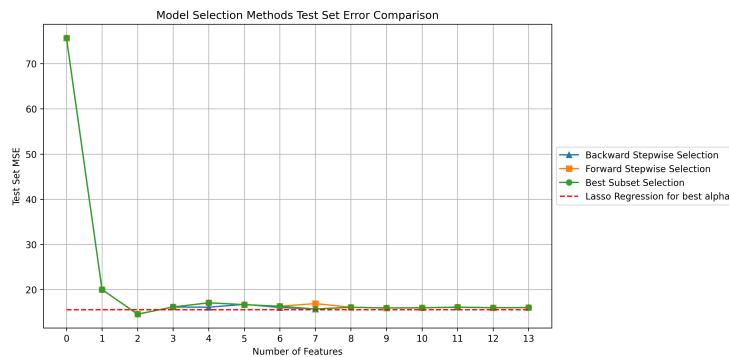


Figura 14: Comparação dos Erros de Teste para Todos os Métodos

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