

Lista 01 de Machine Learning

Pedro Barbosa Bahia

IMPA, Verão 2026

Sumário

1	Exercício 1a	3
2	Exercício 1b	4
3	Exercício 1c	5
4	Exercício 1d	6
5	Exercício 2	7
5.1	Parte d	7
5.1.1	i) Geração dos Dados Heteroscedásticos	7
5.1.2	ii) Estimadores de Mínimos Quadrados	7
5.1.3	iii) Cálculo de p-valores para Mínimos Quadrados Ordinários	9
5.1.4	iv) Estatística Z para o Estimador Generalizado	10
5.1.5	v) P-valores para o Estimador Generalizado	10
5.1.6	vi) Resultados Numéricos	11
6	Exercício 3	13
6.1	Parte f	13
6.1.1	i) Implementação dos Estimadores	13
6.1.2	ii) Comparação dos Estimadores	14
6.1.3	iii) Robustez a Outliers	14
7	Exercício 4a	17
7.1	Implementação dos Algoritmos de Classificação	17
8	Exercício 4b	20
8.1	Treinamento e Avaliação dos Modelos	20
8.2	Comparação de Performance dos Modelos	20
8.2.1	Análise Geral dos Algoritmos	20
8.2.2	Análise Específica do k-NN	20
8.3	Análise dos Coeficientes dos Modelos	21
8.4	Conclusões	22

9	Exercício 5	23
9.1	Métodos de Seleção de Modelos	23
9.1.1	Preparação dos Dados	23
9.1.2	Implementação dos Algoritmos	23
9.2	Comparação dos Métodos - R^2	26
9.3	Regressão Lasso com Validação Cruzada	27
9.3.1	Seleção do Parâmetro de Regularização	27
9.4	Comparação no Conjunto de Teste	27
9.5	Resultados Numéricos	28
9.6	Análise do Modelo de 2 Features	28
9.7	Conclusões	29

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

Falso

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

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

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

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

5 Exercício 2

5.1 Parte d

5.1.1 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 + 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:

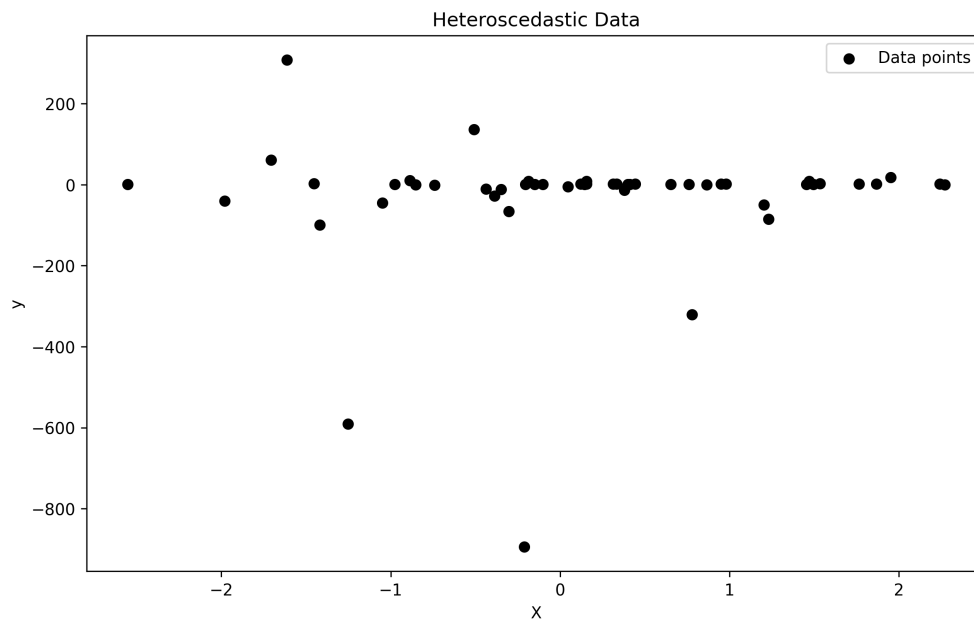


Figura 1: Dados heteroscedásticos gerados

A Figura 2 mostra os elementos diagonais da matriz Σ , evidenciando a heterocedasticidade:
As Figuras 3 e 4 mostram a distribuição e valores dos erros:

5.1.2 ii) Estimadores de Mínimos Quadrados

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

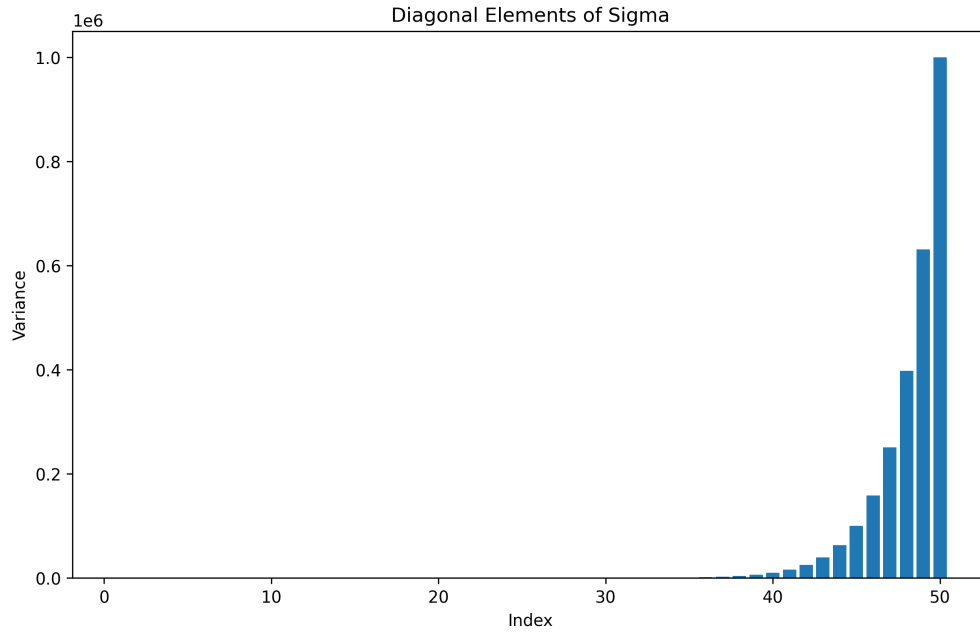


Figura 2: Elementos diagonais da matriz Σ

```

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) ->
9     np.ndarray:
10    """
11    Compute the generalized least squares estimator 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 @ Y
16    return beta
17
18 beta_hat_ordinary = beta_ordinary(X, y)
19 beta_hat_sigma = beta_sigma(X, y, Sigma)
20
21 print("True Beta:", beta)
22 print("Ordinary Beta:", beta_hat_ordinary)
23 print("Beta Sigma:", beta_hat_sigma)

```

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

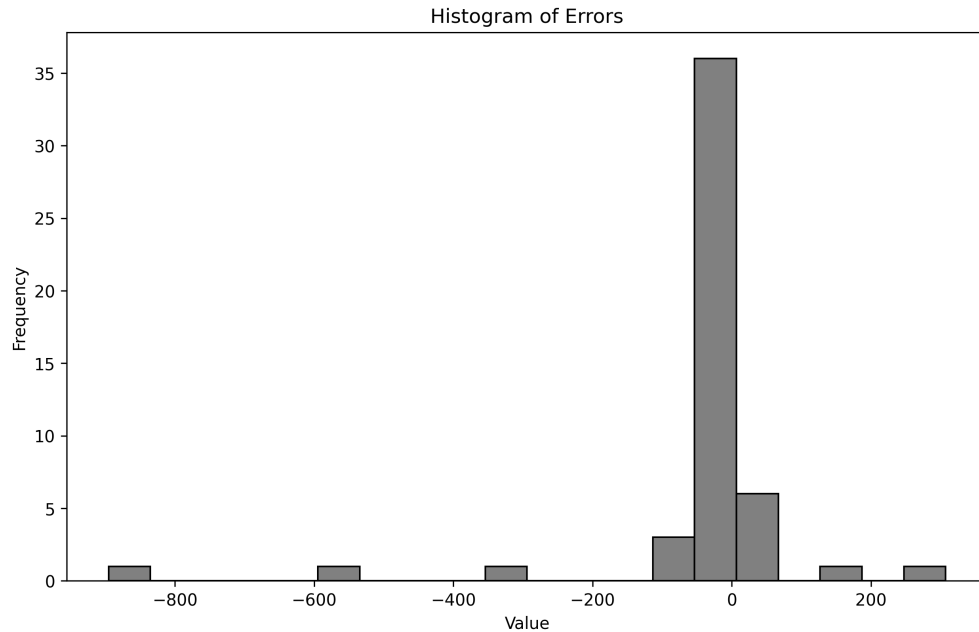


Figura 3: Histograma dos erros

5.1.3 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)
3                                   -> float:
4
5     """
6     Compute the p-value for the j-th coefficient of the ordinary
7     least squares estimator.
8     """
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)
26    p_value = 2 * (1 - scipy.stats.t.cdf(t_statistics, dof))

```

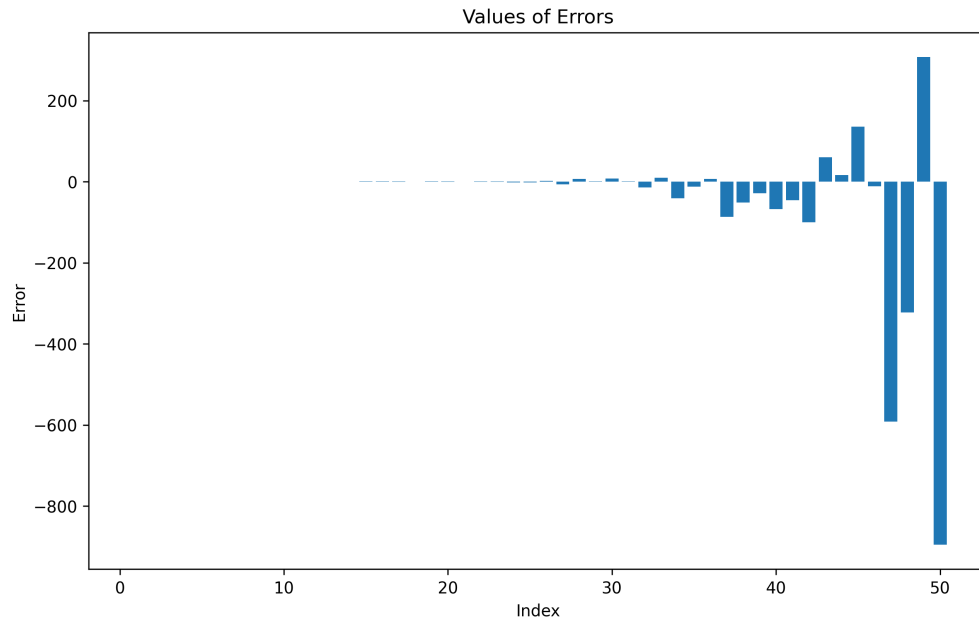


Figura 4: Valores dos erros por índice

```
25 return p_value
```

5.1.4 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
5     generalized least squares estimator.
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
```

5.1.5 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:
4                                           int) -> float:
5     """
6     Compute the p-value for the j-th coefficient of the
7     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.

5.1.6 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.

6 Exercício 3

6.1 Parte f

6.1.1 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:
13    str) -> np.ndarray:
14    """
15    Compute the estimator beta_hat based on the specified error
16    distribution.
17    """
18    np.random.seed(0)
19    p = X.shape[1]
20
21    if error_distribution == "gaussian":
22        def loss_function(beta):
23            return np.sum((Y - X @ beta) ** 2)
24    elif error_distribution == "laplacian":
25        def loss_function(beta):
26            return np.sum(np.abs(Y - X @ beta))
27    else:
28        raise ValueError("Unsupported error distribution")
29
30    beta_0 = np.random.uniform(size=p)
31    beta_hat = scipy.optimize.minimize(loss_function, beta_0)
32    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:

As Figuras 6 e 7 mostram a análise dos erros:

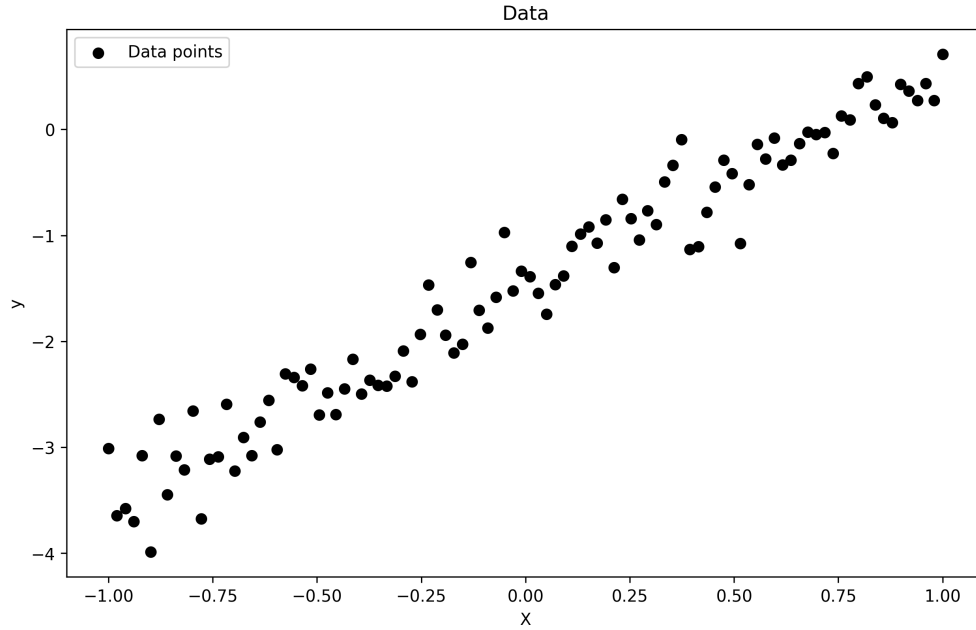


Figura 5: Dados sintéticos para comparação dos estimadores

6.1.2 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:

6.1.3 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
```

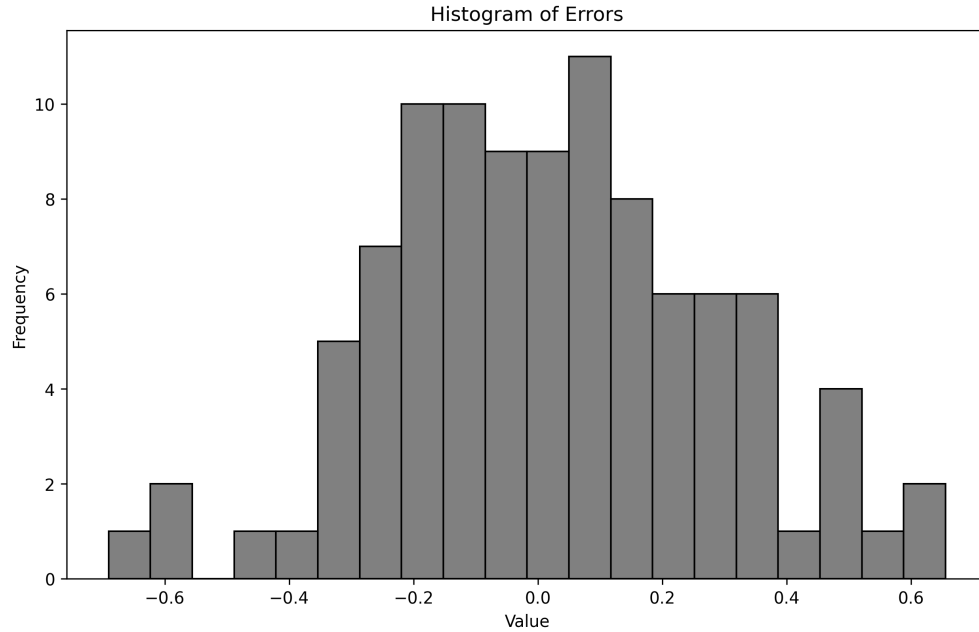


Figura 6: Histograma dos erros

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:

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.

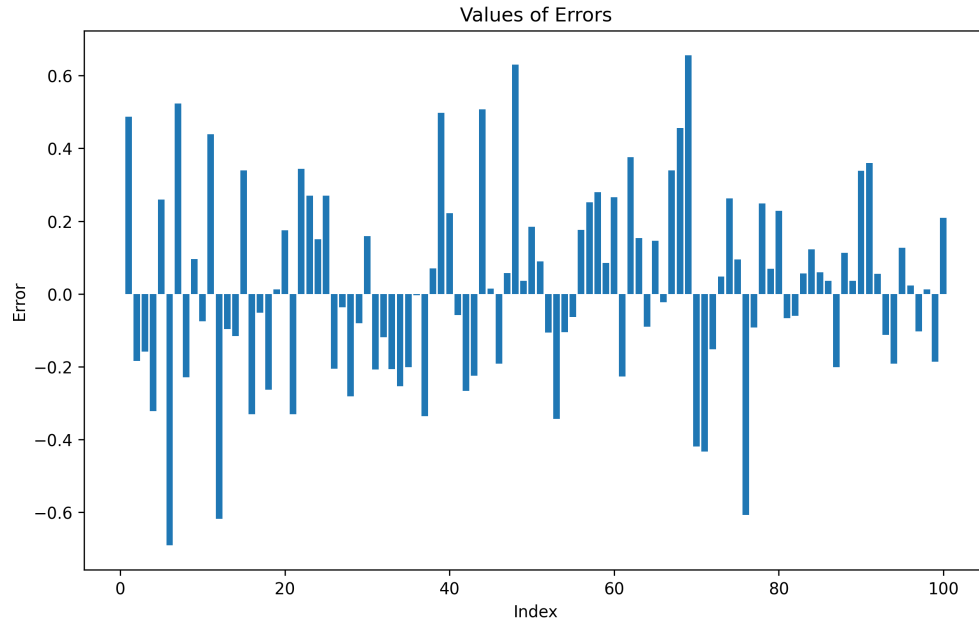


Figura 7: Valores dos erros por índice

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.

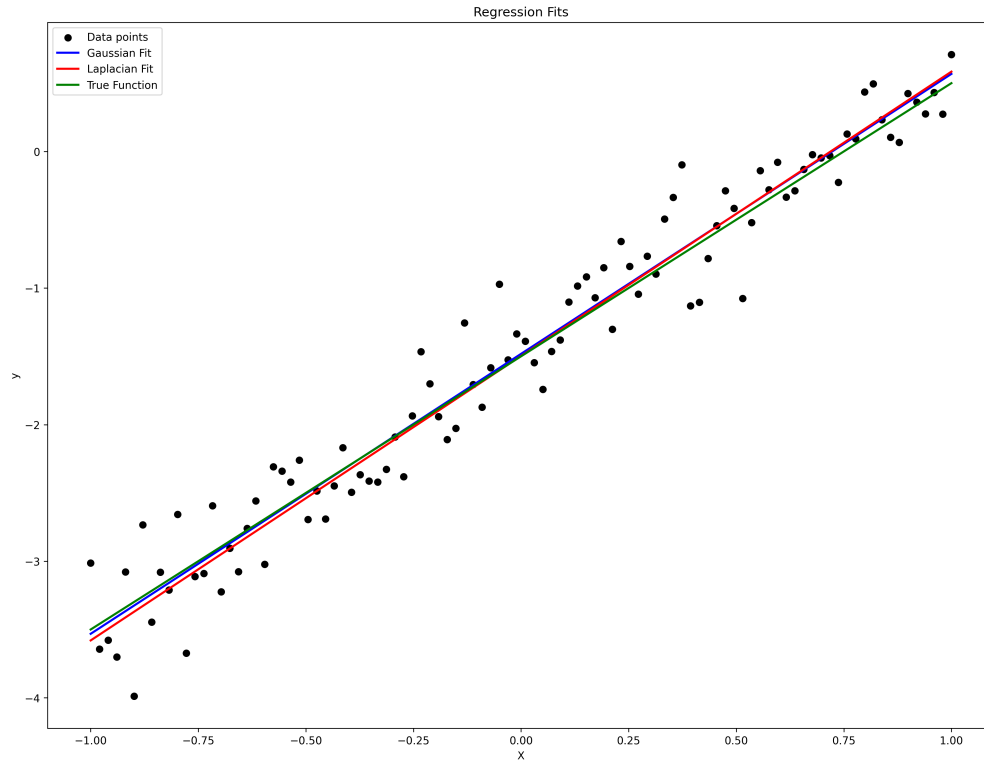


Figura 8: Comparação dos ajustes de regressão sem outliers

7 Exercício 4a

7.1 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 LinearDiscriminantAnalysis as
  LDA
5 from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
  as QDA
```

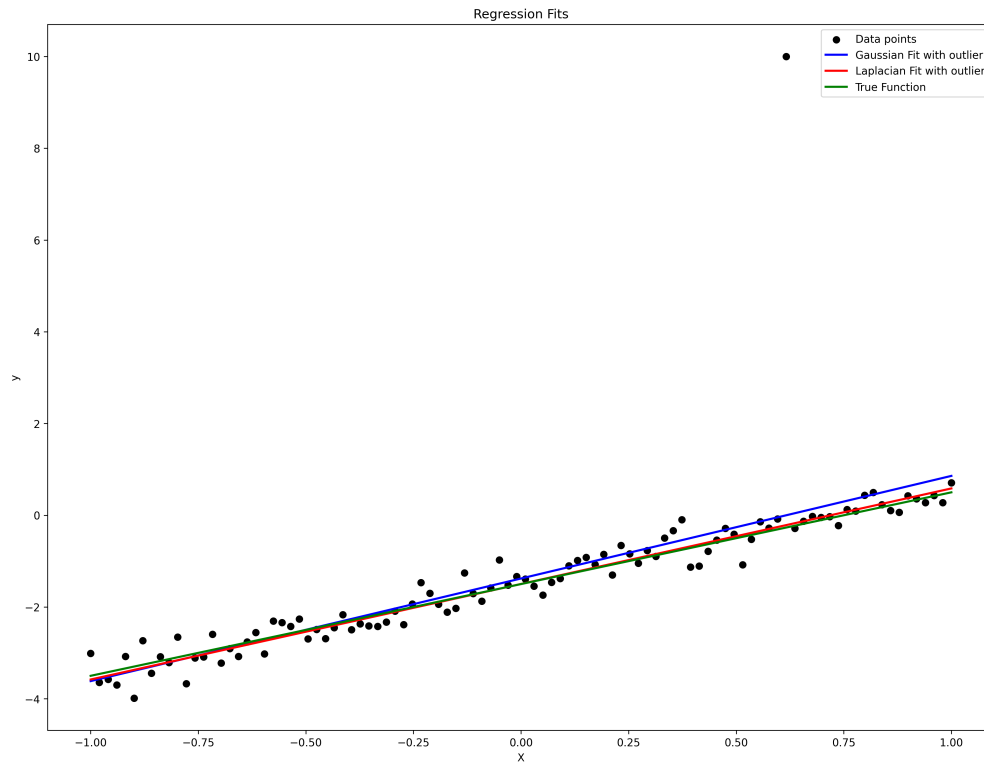


Figura 9: Comparação dos ajustes de regressão com outlier

```

6 from sklearn.linear_model import LogisticRegression as LR
7 from sklearn.naive_bayes import GaussianNB as NB
8 from sklearn.neighbors import KNeighborsClassifier as kNN
9 from sklearn import preprocessing
10
11 # Load and prepare data
12 df = pd.read_csv("../data/soccer.csv")
13 X = df.drop("target", axis=1)
14 y = df[["target"]]
15
16 # Split dataset
17 X_train, y_train = X.iloc[:2560], y.iloc[:2560]
18 X_test, y_test = X.iloc[2560:], y.iloc[2560:]
19
20 # Remove categorical variables and standardize
21 X_train = X_train.drop(["home_team", "away_team"], axis=1)
22 X_test = X_test.drop(["home_team", "away_team"], axis=1)
23 scaler = preprocessing.StandardScaler()
24 X_train = scaler.fit_transform(X_train)
25 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)

8 Exercício 4b

8.1 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
```

8.2 Comparação de Performance dos Modelos

8.2.1 Análise Geral dos Algoritmos

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

```
1 for model_name in models_to_test:
2     model_type_name = model_name.__name__
3     in_sample_predictions =
        results_dict[model_type_name]["in_sample_predictions"]
4     test_predictions = results_dict[model_type_name]["test_predictions"]
5
6     train_error = np.mean(in_sample_predictions != y_train.values.ravel())
7     test_error = np.mean(test_predictions != y_test.values.ravel())
8
9     plt.scatter(train_error, test_error, label=model_type_name)
```

8.2.2 Análise Específica do k-NN

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

```
1 for k in range(1, 11):
2     model = kNN(n_neighbors=k)
3     model.fit(X_train, y_train.values.ravel())
4
5     in_sample_predictions_k = model.predict(X_train)
```

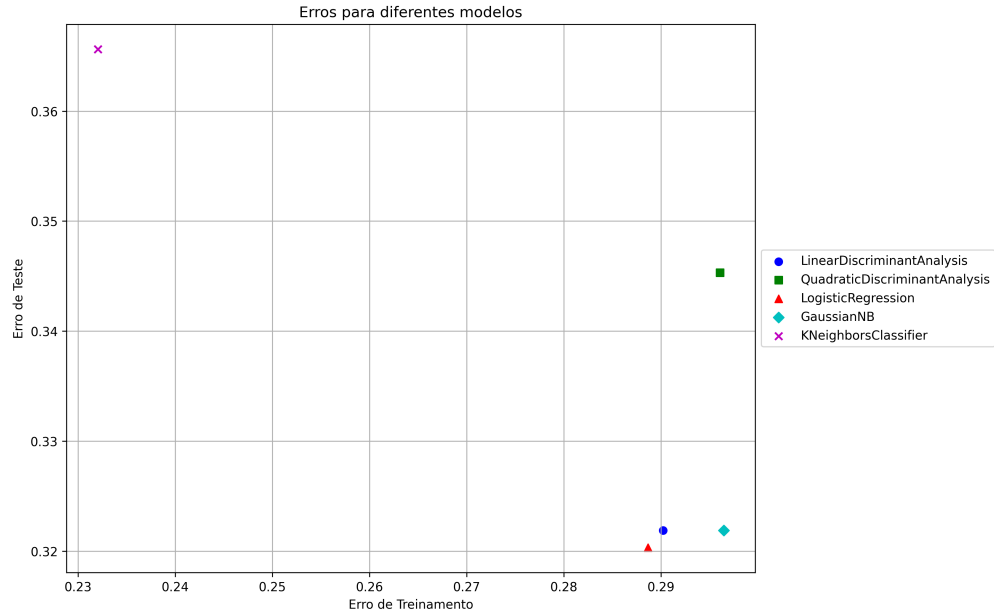


Figura 10: Comparação dos erros de treinamento vs teste para diferentes modelos

```

6 test_predictions_k = model.predict(X_test)
7
8 train_error = np.mean(in_sample_predictions_k !=
9 y_train.values.ravel())
10 test_error = np.mean(test_predictions_k != y_test.values.ravel())
11
12 plt.scatter(train_error, test_error, label=f"K = {k}",
13 marker=f"${k}$")

```

8.3 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

Logistic Regression:

- Coeficientes: $[0.87, -0.29, -0.020, 0.056, 0.26, 0.078, 0.40, -0.053, -0.078, 0.063, 0.047]$
- Intercepto: 0.128
- Coeficientes similares ao LDA, indicando estrutura linear nos dados

Gaussian Naive Bayes:

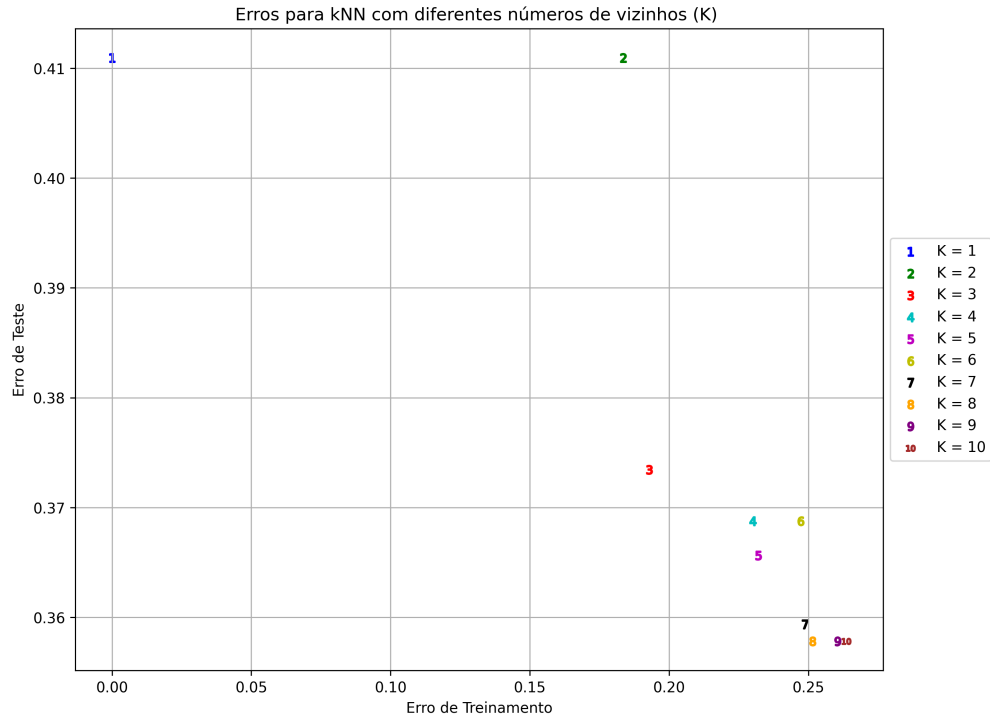


Figura 11: Erros do k-NN para diferentes números de vizinhos (K=1 a K=10)

- Assume independência condicional entre features
- Médias das classes: $[-0.49, 0.28, 0.27, -0.30, -0.31, 0.23, -0.32, -0.18, 0.25, 0.17, 0.031]$ (Classe 0)
- Variâncias por feature calculadas separadamente para cada classe

8.4 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. **Estabilidade:** LDA, QDA e Logistic Regression apresentaram performance consistente entre treino e teste.
4. **Complexidade:** QDA, ao modelar covariâncias separadas por classe, não mostrou melhoria significativa sobre LDA, indicando que a suposição de covariância comum é adequada.
5. **Naive Bayes:** Manteve performance competitiva mesmo com a forte suposição de independência, sugerindo que as correlações entre features não são críticas para este problema.
6. **k-NN:** A performance otimizada ocorre com K entre 3-7, balanceando bias e variância.

9 Exercício 5

9.1 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
- **Lasso Regression:** Regularização L1 com seleção automática de features

9.1.1 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)
```

9.1.2 Implementação dos Algoritmos

Best Subset Selection:

```
1 def best_subset_selection(X_train: np.ndarray, Y_train: np.ndarray) ->
2     dict:
3     """
4     Perform best subset selection for linear regression.
5     Evaluates all possible combinations of features.
6     """
7     n, p = X_train.shape
8     best_model_k = {}
```

```

9      # Model with no features
10     best_model_k["0"] = {
11         "best_model": sm.OLS(Y_train, np.ones((len(Y_train), 1))).fit(),
12         "best_r2": 0.0,
13         "best_features": []
14     }
15
16     for k in range(1, p + 1):
17         best_r2 = 0
18         best_features = []
19         best_model = []
20
21         # Evaluate all combinations of k features
22         for feature_combination in itertools.combinations(range(p), k):
23             X_train_aux = sm.add_constant(X_train[:, feature_combination])
24             model = sm.OLS(Y_train, X_train_aux).fit()
25
26             if model.rsquared > best_r2:
27                 best_r2 = model.rsquared
28                 best_model = model
29                 best_features = feature_combination
30
31         best_model_k[str(k)] = {
32             "best_model": best_model,
33             "best_r2": best_r2,
34             "best_features": list(best_features),
35         }
36
37     return best_model_k

```

Forward Stepwise Selection:

```

1 def forward_stepwise_selection(X_train: np.ndarray, Y_train: np.ndarray)
2     -> dict:
3     """
4     Perform forward stepwise selection. Iteratively adds the best feature.
5     """
6     n, p = X_train.shape
7     best_model_k = {}
8     remaining_features = list(range(p))
9     current_features = []
10
11     # Model with no features
12     best_model_k["0"] = {
13         "best_model": sm.OLS(Y_train, np.ones((len(Y_train), 1))).fit(),
14         "best_r2": 0.0,
15         "best_features": []
16     }
17
18     for k in range(1, p + 1):
19         best_r2 = 0
20         best_features = []

```



```

20     best_model = []
21
22     # Try adding each remaining feature
23     for new_feature in remaining_features:
24         feature_combination_list = current_features + [new_feature]
25         X_train_aux = sm.add_constant(X_train[:,
26             feature_combination_list])
27         model = sm.OLS(Y_train, X_train_aux).fit()
28
29         if model.rsquared > best_r2:
30             best_r2 = model.rsquared
31             best_model = model
32             best_features = feature_combination_list
33
34     # Update feature lists
35     remaining_features = list(set(remaining_features) -
36         set(best_features))
37     current_features = list(set(best_features + current_features))
38
39     best_model_k[str(k)] = {
40         "best_model": best_model,
41         "best_r2": best_r2,
42         "best_features": current_features.copy(),
43     }
44
45     return best_model_k

```

Backward Stepwise Selection:

```

1 def backward_stepwise_selection(X_train: np.ndarray, Y_train: np.ndarray)
2     -> dict:
3     """
4     Perform backward stepwise selection. Iteratively removes the worst
5     feature.
6     """
7     n, p = X_train.shape
8     best_model_k = {}
9     current_features = list(range(p))
10
11     # Model with all features
12     best_model_k[f"{p}"] = {
13         "best_model": sm.OLS(Y_train, sm.add_constant(X_train)).fit(),
14         "best_features": list(current_features)
15     }
16     best_model_k[f"{p}"]["best_r2"] =
17         best_model_k[f"{p}"]["best_model"].rsquared
18
19     for k in range(1, p + 1):
20         best_r2 = 0
21         best_features = []
22         best_model = []

```

```

21     # Try removing each current feature
22     for feature_to_remove in current_features:
23         feature_combination_list = current_features.copy()
24         feature_combination_list.remove(feature_to_remove)
25         X_train_aux = sm.add_constant(X_train[:,
26                                     feature_combination_list])
27         model = sm.OLS(Y_train, X_train_aux).fit()
28
29         if model.rsquared > best_r2:
30             best_r2 = model.rsquared
31             best_model = model
32             best_features = feature_combination_list
33
34     current_features = best_features
35
36     best_model_k[str(p - k)] = {
37         "best_model": best_model,
38         "best_r2": best_r2,
39         "best_features": current_features.copy(),
40     }
41     return best_model_k

```

9.2 Comparação dos Métodos - R^2

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

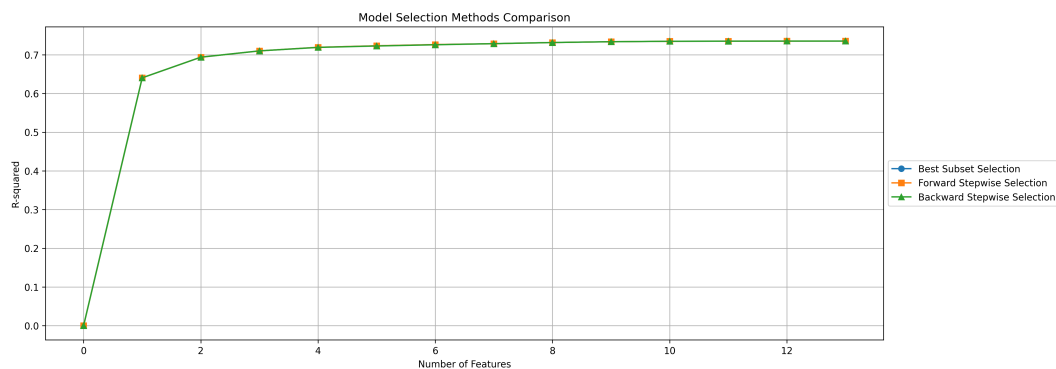


Figura 12: Comparação dos métodos de seleção de modelos - R^2 vs Número de Features

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

9.3 Regressão Lasso com Validação Cruzada

9.3.1 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(x_train_fold,
16                                                    x_test_fold)
17
18        # Train and evaluate Lasso model
19        model_ = Lasso(alpha=alpha_0).fit(x_train_fold, y_train_fold)
20        yhat = model_.predict(x_test_fold)
21        fold_error.append(mean_squared_error(yhat, y_test_fold))
22
23    mean_cv_error[alpha_0] = np.mean(fold_error)
24 best_alpha = min(mean_cv_error, key=mean_cv_error.get)

```

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

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

9.4 Comparação no Conjunto de Teste

A Figura 14 compara o erro no conjunto de teste para todos os métodos:

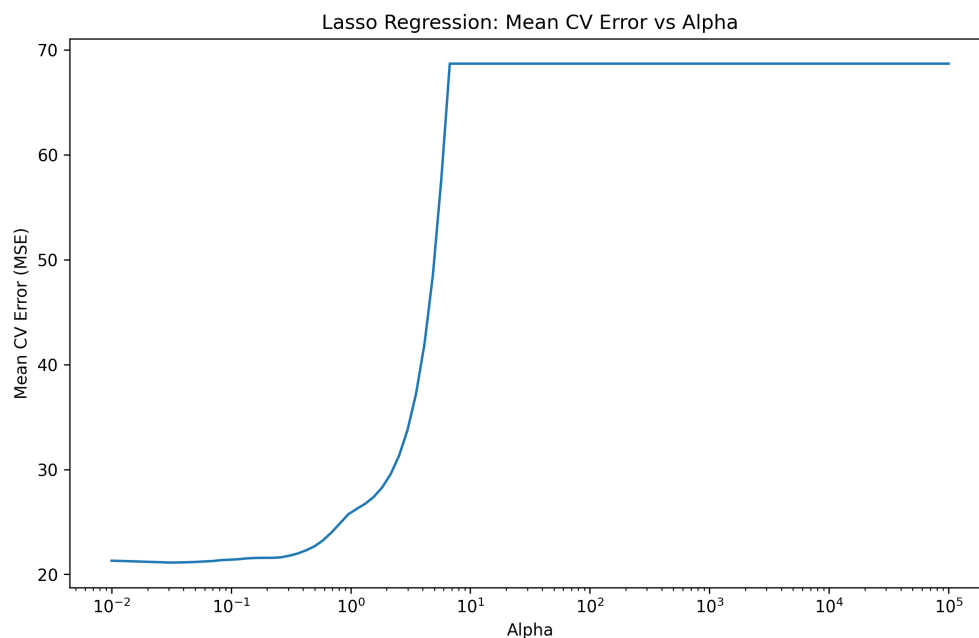


Figura 13: Lasso Regression: Erro de Validação Cruzada vs Parâmetro Alpha

Features	Best Subset MSE	Forward Stepwise MSE	Backward Stepwise MSE
1	20.01	20.01	20.01
2	14.58	14.58	14.58
3	16.16	16.16	16.16
6	16.32	16.32	16.06
7	15.72	16.92	15.72
9	15.94	15.94	15.94
Lasso Regression MSE: 15.56			

Tabela 1: Erros de teste (MSE) para diferentes números de features

9.5 Resultados Numéricos

9.6 Análise do Modelo de 2 Features

O modelo com 2 features (Weight e Abdomen) mostra excelente performance:

Modelo: $\text{BodyFat} = -45.63 - 0.14 \times \text{Weight} + 0.98 \times \text{Abdomen}$

- **Intercepto:** -45.63
- **Coef. Weight:** -0.14 (negativo - mais peso, menos gordura corporal relativa)
- **Coef. Abdomen:** 0.98 (positivo - maior circunferência abdominal, maior gordura corporal)
- **R²:** 0.694
- **MSE no teste:** 14.58

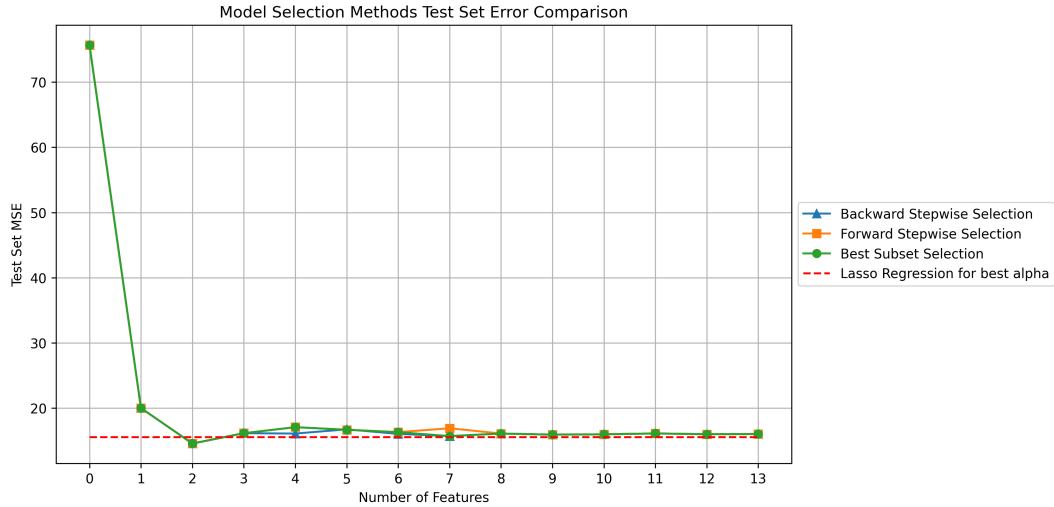


Figura 14: Comparação dos erros no conjunto de teste para diferentes métodos

9.7 Conclusões

1. **Convergência dos Métodos:** Para números pequenos de features (1-3), todos os métodos stepwise encontram as mesmas soluções ótimas que o best subset selection.
2. **Feature Mais Importante:** Abdomen (circunferência abdominal) é consistentemente a feature mais importante, explicando 64% da variância sozinha.
3. **Modelo Parcimonioso:** O modelo de 2 features (Weight + Abdomen) oferece excelente trade-off entre simplicidade e performance (MSE = 14.58).
4. **Lasso Performance:** O Lasso alcança o melhor resultado no teste (MSE = 15.56), mas usa todas as features. O trade-off é menor interpretabilidade.
5. **Overfitting:** Modelos com mais de 3 features mostram sinais de overfitting, com R^2 crescente no treino mas MSE crescente no teste.
6. **Eficiência Computacional:** Forward/backward stepwise são muito mais eficientes que best subset ($O(p^2)$ vs $O(2^p)$), com performance quase idêntica para este problema.
7. **Regularização:** O Lasso fornece uma abordagem automática para seleção de features com excelente performance de generalização.