Lista 01 de Machine Learning

Pedro Barbosa Bahia

IMPA, Verão 2026

Sumário

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:

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

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

A Figura 1 mostra os dados gerados com heterocedasticidade:

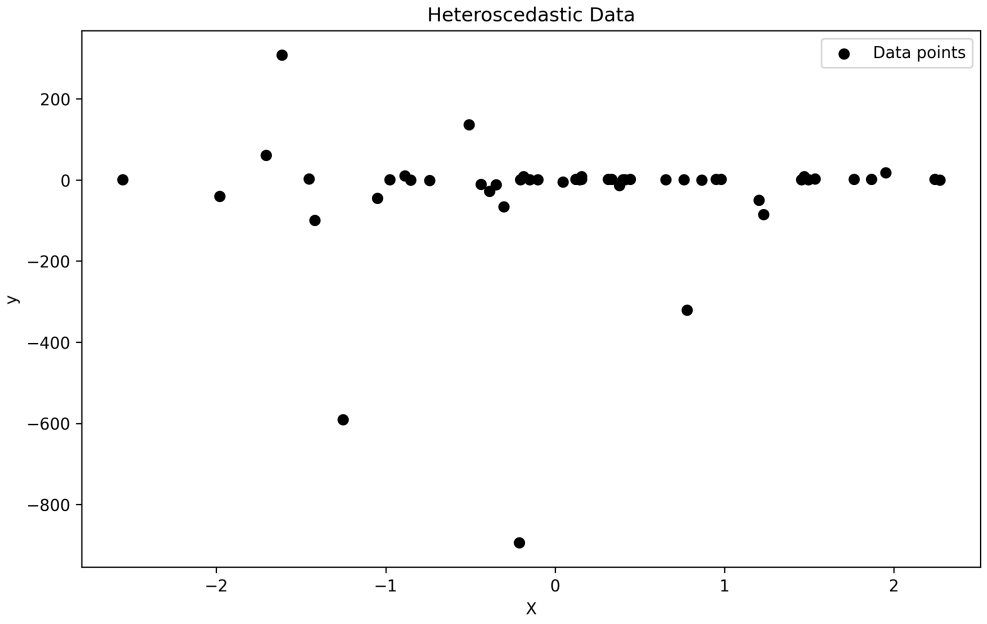

Figura 1: Dados heteroscedásticos gerados

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

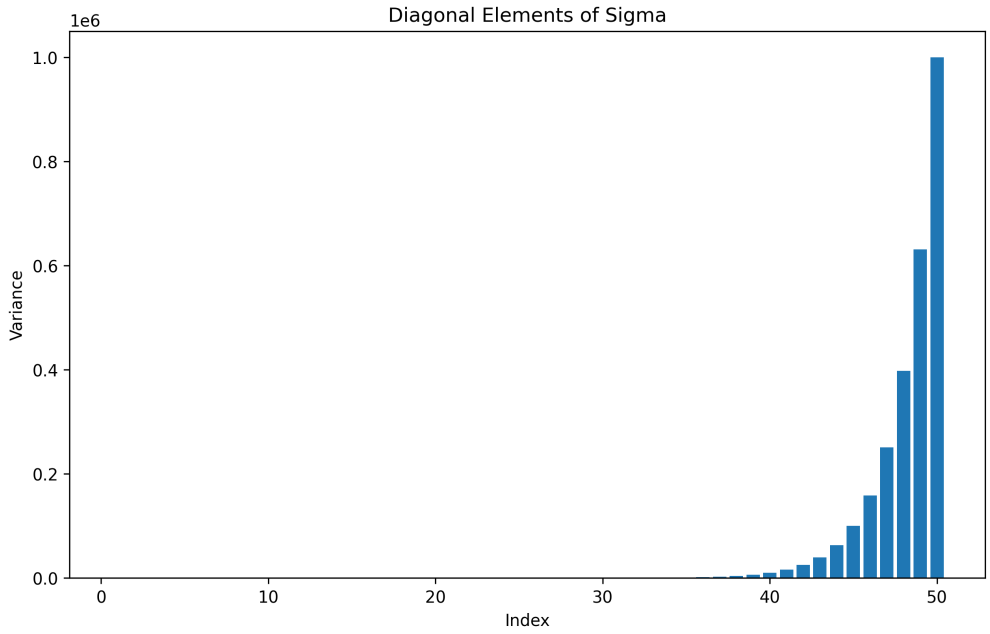

Figura 2: Elementos diagonais da matriz Σ

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

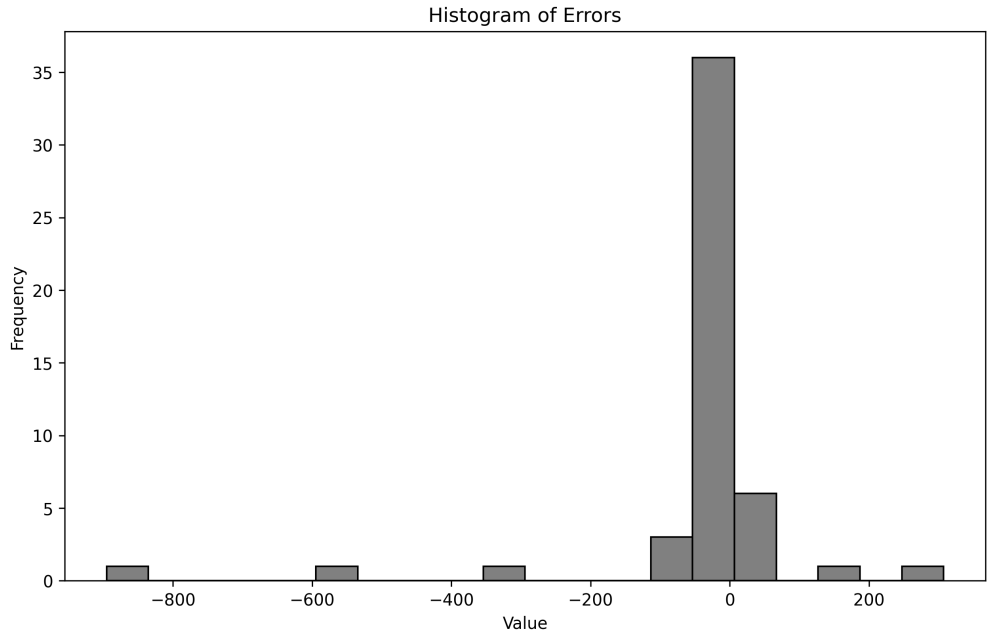

Figura 3: Histograma dos erros

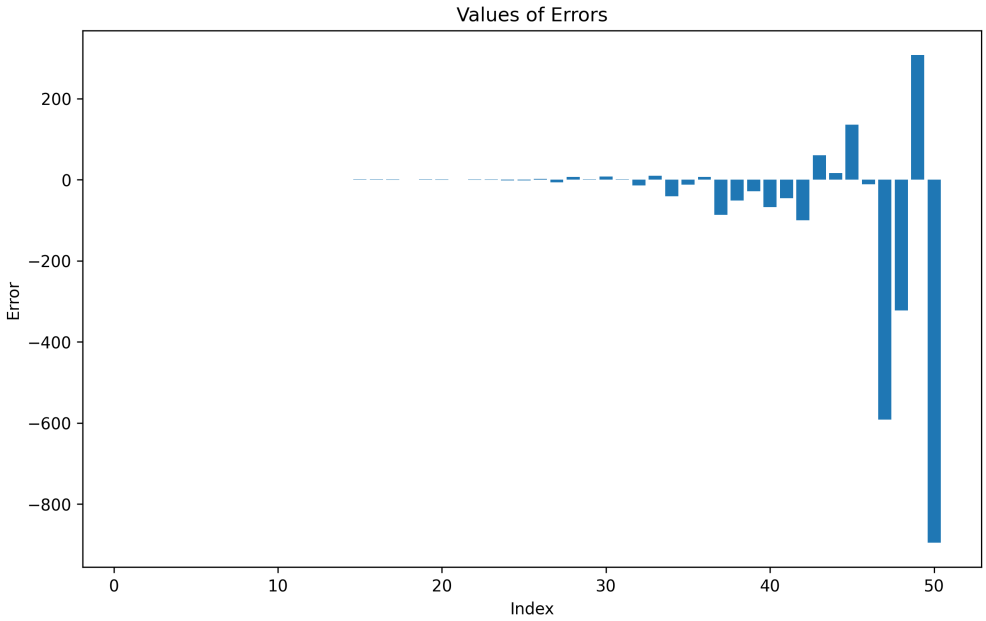

Figura 4: Valores dos erros por índice

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:

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

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

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

print("True Beta:", beta)
print("Ordinary Beta:", beta_hat_ordinary)
print("Beta Sigma:", beta_hat_sigma)
```

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

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

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

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

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

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

    return p_value
```

5.1.4 iv) Estatística Z para o Estimador Generalizado

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

5.1.5 v) P-valores para o Estimador Generalizado

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

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

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

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

    return p_value
```

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

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:

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

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

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

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

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

Geramos dados sintéticos para testar os estimadores:

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

A Figura 5 mostra os dados gerados:

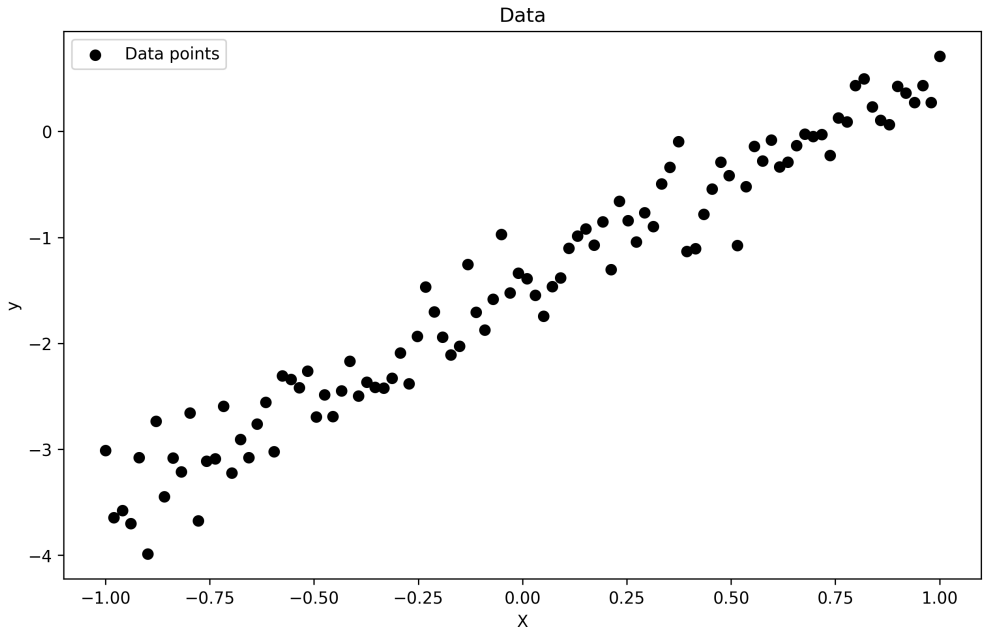

Figura 5: Dados sintéticos para comparação dos estimadores

As Figuras 6 e 7 mostram a análise dos erros:

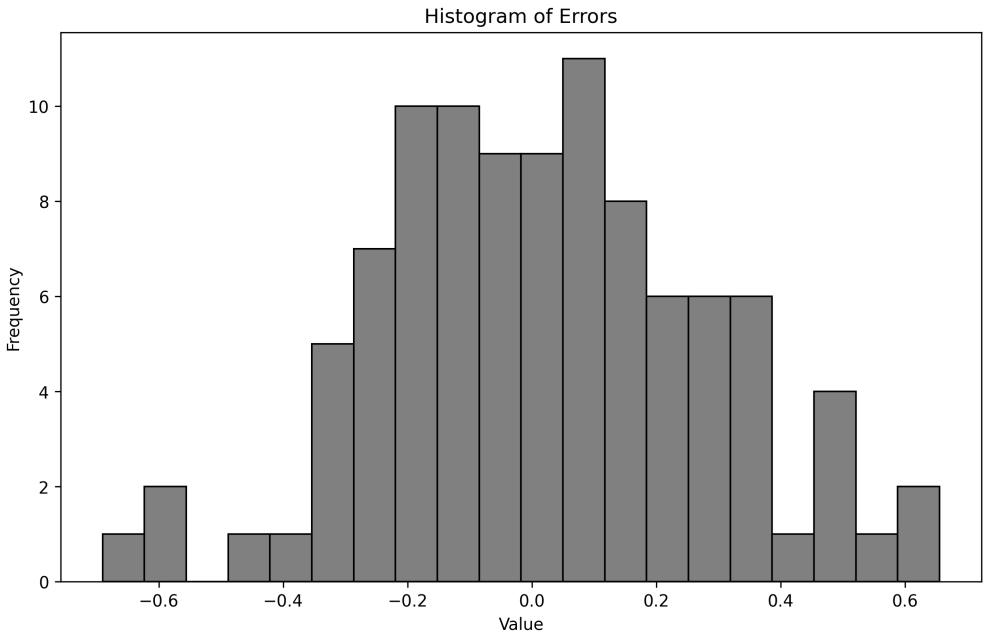

Figura 6: Histograma dos erros

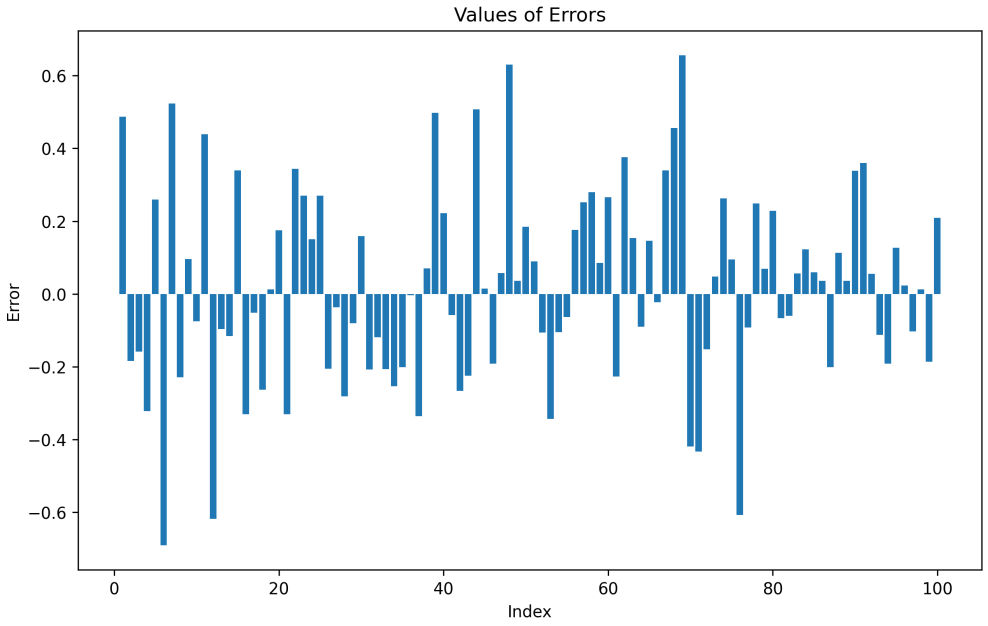

Figura 7: Valores dos erros por índice

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:

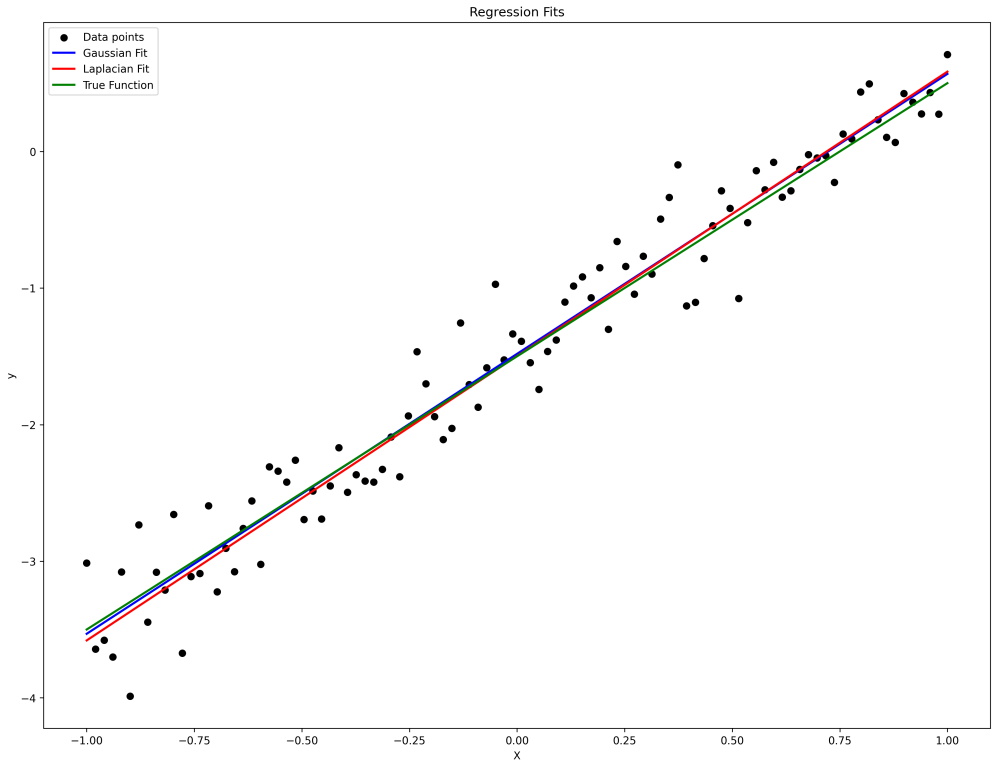

Figura 8: Comparação dos ajustes de regressão sem outliers

6.1.3 iii) Robustez a Outliers

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

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

Resultados com Outlier:

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

Análise de Erro com Outlier (Norma L2):

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

A Figura 9 mostra o impacto do outlier:

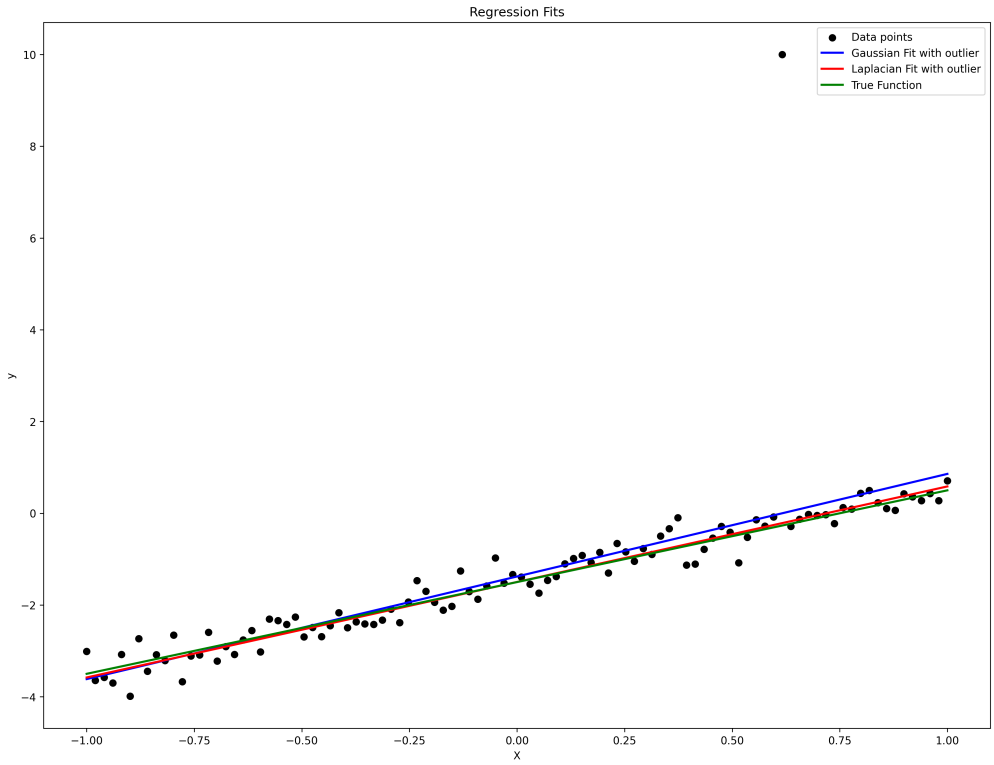

Figura 9: Comparação dos ajustes de regressão com outlier

Conclusões:

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

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

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

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

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

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)

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

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

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

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

Informações do Dataset:

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

8 Exercício 4b

8.1 Treinamento e Avaliação dos Modelos

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

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

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

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

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

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:

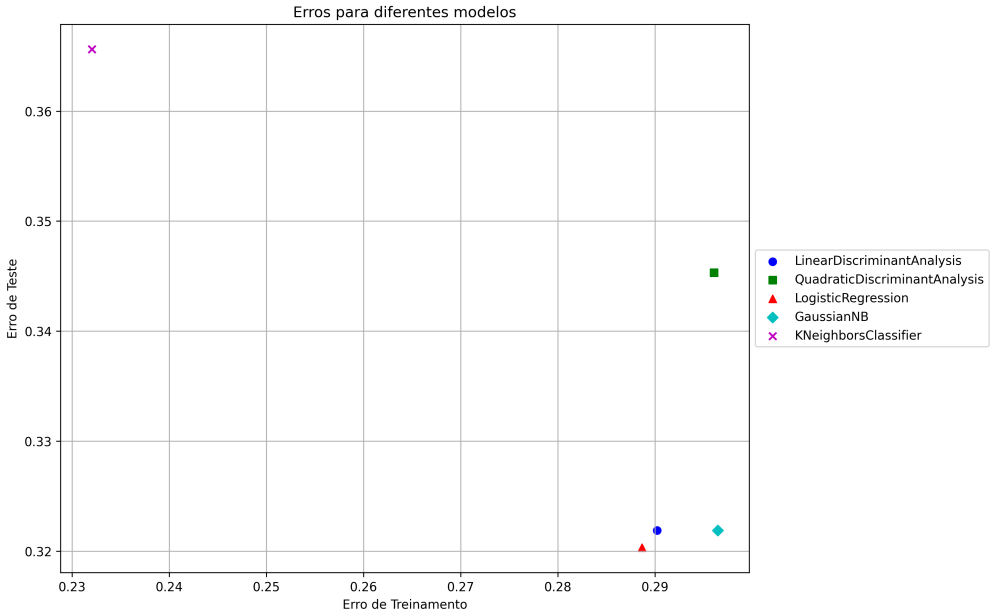

Figura 10: Comparação dos erros de treinamento vs teste para diferentes modelos

```
for model_name in models_to_test:
    model_type_name = model_name.__name__
    in_sample_predictions =
        results_dict[model_type_name]["in_sample_predictions"]
    test_predictions = results_dict[model_type_name]["test_predictions"]

    train_error = np.mean(in_sample_predictions != y_train.values.ravel())
    test_error = np.mean(test_predictions != y_test.values.ravel())

    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:

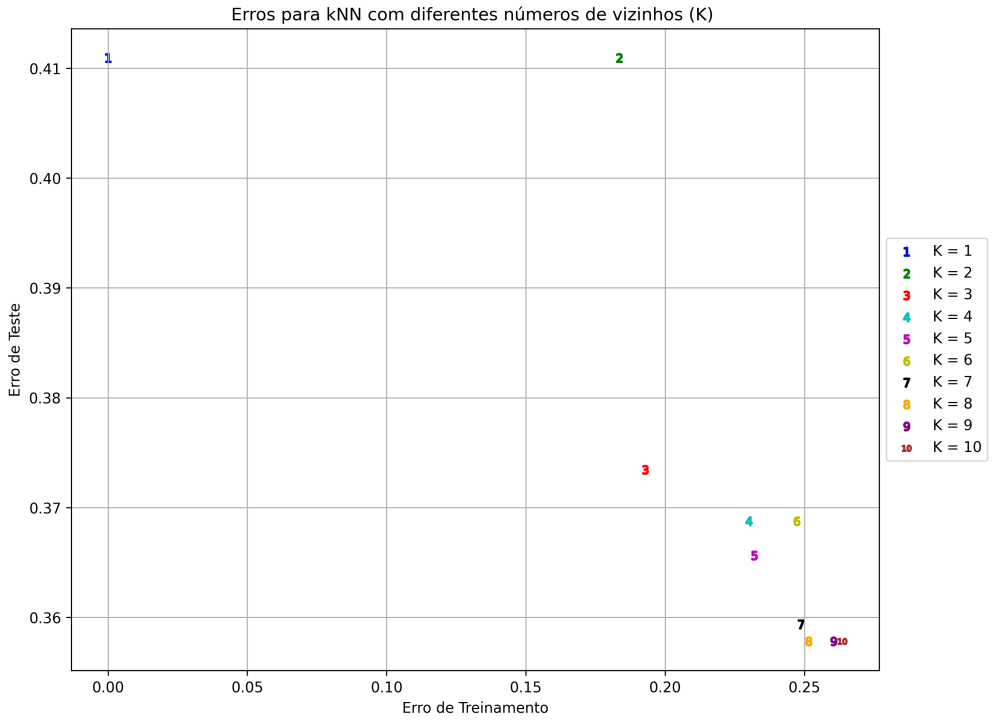

Figura 11: Erros do k-NN para diferentes números de vizinhos (K=1 a K=10)

```
for k in range(1, 11):
    model = kNN(n_neighbors=k)
    model.fit(X_train, y_train.values.ravel())

    in_sample_predictions_k = model.predict(X_train)
    test_predictions_k = model.predict(X_test)

    train_error = np.mean(in_sample_predictions_k !=
        y_train.values.ravel())
    test_error = np.mean(test_predictions_k != y_test.values.ravel())

    plt.scatter(train_error, test_error, label=f"K = {k}",
        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:

- 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

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

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

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

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

9.1.2 Implementação dos Algoritmos

Best Subset Selection:

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

```

```
    # Model with no features
    best_model_k["0"] = {
        "best_model": sm.OLS(Y_train, np.ones((len(Y_train), 1))).fit(),
        "best_r2": 0.0,
        "best_features": []
    }

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

        # Evaluate all combinations of k features
        for feature_combination in itertools.combinations(range(p), k):
            X_train_aux = sm.add_constant(X_train[:, feature_combination])
            model = sm.OLS(Y_train, X_train_aux).fit()

            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination

        best_model_k[str(k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": list(best_features),
        }

    return best_model_k
```

Forward Stepwise Selection:

```
def forward_stepwise_selection(X_train: np.ndarray, Y_train: np.ndarray)
    -> dict:
    """
    Perform forward stepwise selection. Iteratively adds the best feature.
    """
    n, p = X_train.shape
    best_model_k = {}
    remaining_features = list(range(p))
    current_features = []

    # Model with no features
    best_model_k["0"] = {
        "best_model": sm.OLS(Y_train, np.ones((len(Y_train), 1))).fit(),
        "best_r2": 0.0,
        "best_features": []
    }

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

```
        best_model = []

        # Try adding each remaining feature
        for new_feature in remaining_features:
            feature_combination_list = current_features + [new_feature]
            X_train_aux = sm.add_constant(X_train[:,
               feature_combination_list])
            model = sm.OLS(Y_train, X_train_aux).fit()

            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination_list

        # Update feature lists
        remaining_features = list(set(remaining_features) -
           set(best_features))
        current_features = list(set(best_features + current_features))

        best_model_k[str(k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": current_features.copy(),
        }

    return best_model_k
```

Backward Stepwise Selection:

```
def backward_stepwise_selection(X_train: np.ndarray, Y_train: np.ndarray)
   -> dict:
    """
    Perform backward stepwise selection. Iteratively removes the worst
       feature.
    """
    n, p = X_train.shape
    best_model_k = {}
    current_features = list(range(p))

    # Model with all features
    best_model_k[f"{p}"] = {
        "best_model": sm.OLS(Y_train, sm.add_constant(X_train)).fit(),
        "best_features": list(current_features)
    }
    best_model_k[f"{p}"]["best_r2"] =
       best_model_k[f"{p}"]["best_model"].rsquared

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

```

```
        # Try removing each current feature
        for feature_to_remove in current_features:
            feature_combination_list = current_features.copy()
            feature_combination_list.remove(feature_to_remove)
            X_train_aux = sm.add_constant(X_train[:,
                feature_combination_list])
            model = sm.OLS(Y_train, X_train_aux).fit()

            if model.rsquared > best_r2:
                best_r2 = model.rsquared
                best_model = model
                best_features = feature_combination_list

        current_features = best_features

        best_model_k[str(p - k)] = {
            "best_model": best_model,
            "best_r2": best_r2,
            "best_features": current_features.copy(),
        }

    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:

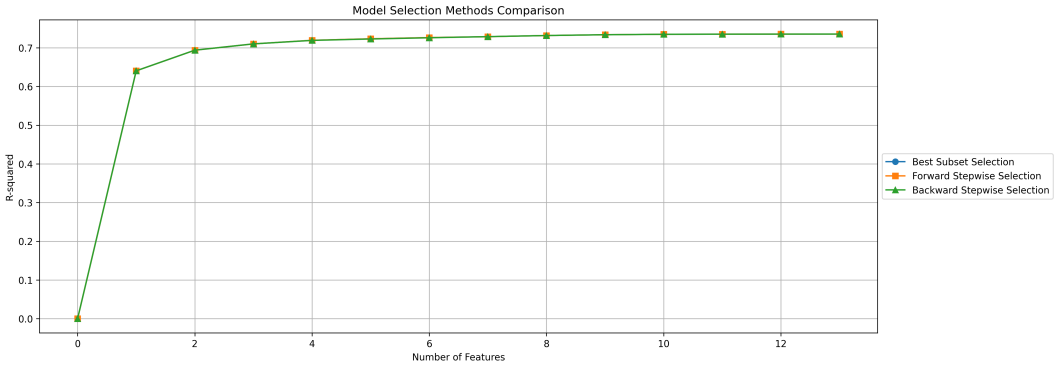

Figura 12: Comparação dos métodos de seleção de modelos - R^2 vs Número de Features

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

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

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

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

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

    mean_cv_error[alpha_0] = np.mean(fold_error)

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

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

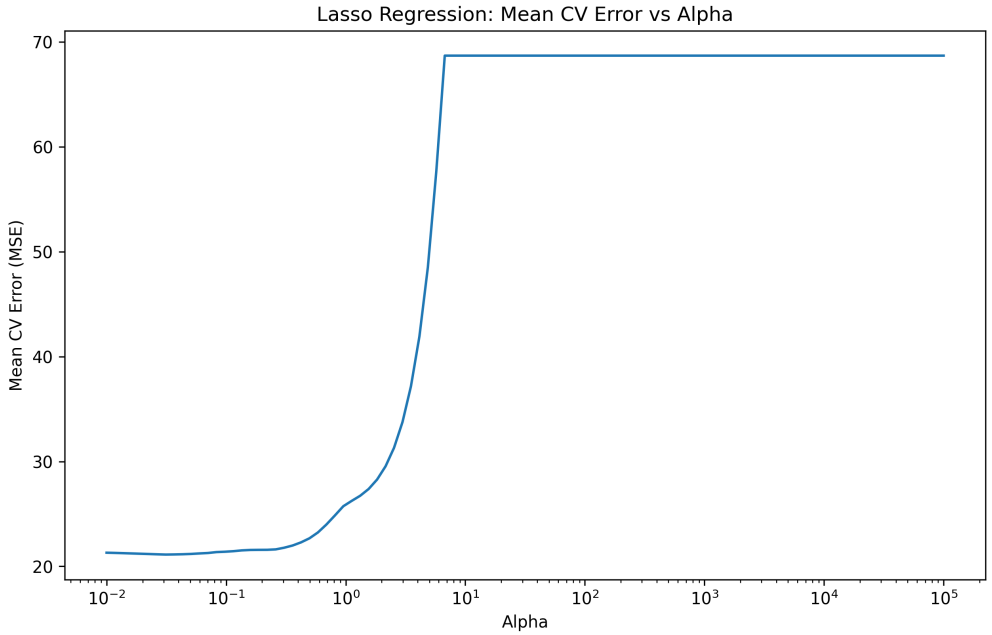

Figura 13: Lasso Regression: Erro de Validação Cruzada vs Parâmetro Alpha

Resultados do Lasso:

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

9.4 Comparação no Conjunto de Teste

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

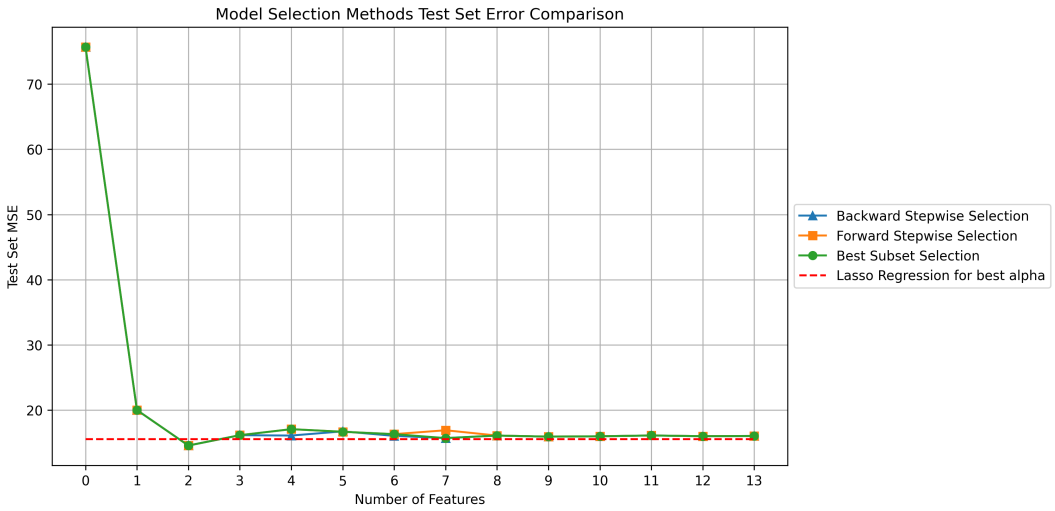

Figura 14: Comparação dos erros no conjunto de teste para diferentes métodos

9.5 Resultados Numéricos

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

9.6 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)
- **R^2**: 0.694
- **MSE no teste**: 14.58

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.