Trabalho AV2: Modelos de Regressão e Classificação

Guilherme de Farias Loureiro, ***Estudante, Unifor***
Tiago Farias Rodrigues, ***Estudante, Unifor***

Departamento de ciência de computação da unifor

Index Terms: Artificial Inteligence, Machine Learning, Ordinary Least Squares, Monte Carlo, Gaussian Classifier

1. Resumo

Nesse relatório acadêmico, foram aplicados algoritmos de aprendizado de máquina - mais especificamente, de regressão e classificação - para analisar os dados relativos a dois problemas e comparar a eficiência desses algoritmos. A priori, no primeiro problema, de regressão, as variantes do algoritmo MQO obtiveram desempenho semelhante, superando por muito o algoritmo de Média dos Valores observáveis. Por fim, no segundo problema, de classificação, o Classificador Guassiano Tradicional enfrentou dificuldades para prever a classe de uma nova amostra, obtendo praticamente o mesmo desempenho - 20% - que uma pessoa teria se apenas escolhesse uma classe aleatória para ela. Em contraste, os outros algoritmos triunfaram, especialmente o Classificador Gausiano Regularizado com hiperparâmetro igual a 0,25, obtendo uma acurácia de 97,48%.

2. Introdução

A Quarta Revolução Industrial promoveu não apenas avanços tecnológicos significativos, mas também uma mudança profunda no paradigma da economia global e nas relações sociais. Atualmente, o acesso ao mundo digital é uma realidade para bilhões de indivíduos, que se conectam por meio de diversos dispositivos eletrônicos, possibilitando a coleta massiva de uma ampla variedade de dados, como informações de busca na internet, dados de consumidores e atividades de usuários em redes sociais, entre outros. Nesse contexto, a necessidade de ferramentas sofisticadas para a análise de grandes volumes de dados torna-se evidente. A relação matemática entre essas informações muitas vezes não é diretamente perceptível, e é justamente nesse aspecto que o campo do aprendizado de máquina se mostra promissor, fornecendo soluções para desafios complexos relacionados à identificação de padrões e à tomada de decisão baseada em dados.

Mediante a importância da área supramencionada, o presente trabalho acadêmico aborda esse tema ao apresentar os resultados da implementação de modelos que realizam predições quantitativas ou qualitativas para analisar os dados de duas situações-problemas. A fim de facilitar a execução de cada trabalho, eles foram divididos em etapas.

2.1. Problema de Regressão

Esse contexto envolve um conjunto de dados contendo a medida de velocidade do vento e a potência gerada pelo aerogerador, no qual o último fator depende do primeiro.

2.2. Problema de Classificação

Esse contexto envolve um conjunto de dados de sinais de eletromiografia captados dos músculos faciais, onde o objetivo é identificar diferentes expressões faciais com base nos padrões desses sinais.

3. Metodologia

No decorrer do desenvolvimento dos códigos de implementação na linguagem de programação *python*, foram utilizadas as seguintes bibliotecas: *numpy* (para a realização de operações matriciais complexas e leitura de dados), *pandas* (para agrupação de dados em *dataframes*), *plotly* e *matplotlib* (para construção de gráficos que possibilitem a visualização dos dados e da eficiência dos algoritmos de predição) e *abc* (para favorecer uma abordagem orientada a objetos).

3.1. Problema de Regressão

As etapas do processo são detalhadas a seguir, buscando assegurar uma abordagem clara e metódica na solução do problema.

1) Visualizar os dados através de um gráfico es espalhamento
2) implementar os algoritmos
    - MQO tradicional
    - MQO regularizado (Tikhonov), hiperâmetros:
        - 0.00
        - 0.25
        - 0.50
        - 0.75
        - 1.00
    - Média dos valores observáveis
3) Calcular medidas de tendência central e de dispersão da soma dos desvios quadráticos obtidos durante a validação por meio de Monte Carlo.

3.2. Problema de Classificação

O problema de classificação consiste em identificar diferentes expressões faciais com base em dados de sinais de eletromiografia capturados por sensores posicionados nos músculos faciais. Foram utilizados dois sensores: o Sensor 1, posicionado no Corrugador do Supercílio, e o Sensor 2, posicionado no Zigomático Maior. Os dados foram coletados com uma taxa de amostragem de 1 kHz, com resolução de 12 bits, e cinco diferentes expressões faciais foram registradas: Neutro, Sorriso, Sobrancelhas Levantadas, Surpreso e Rabugento.

Para resolver esse problema, implementamos alguns modelos de aprendizado supervisionado aprendidos ao longo desta AV2:

1) **Visualização Inicial dos Dados**: Utilizamos um gráfico de dispersão para explorar visualmente a distribuição das classes e verificar a separabilidade entre elas. A (Figura 7) mostra a distribuição das amostras coletadas para cada expressão facial.
2) **Modelos Implementados**:
    - **MQO Tradicional**: Esse modelo, comumente usado para regressão, foi adaptado para fins de classificação, ainda que tenha mostrado limitações no desempenho de separação das classes.

- **Classificador Gaussiano Tradicional**: Modelo probabilístico que assumem uma distribuição gaussiana para cada classe. Este modelo foi implementado dentro do modelo de Friedman, já que seu resultado é igual a lambda zero.
- **Classificador Gaussiano Covariâncias Iguais (LDA)**: Modelo onde a matriz de covariância é compartilhada entre as classes, o que simplifica o cálculo e melhora a eficiência.
- **Classificador Gaussiano com Covariância Agregada**: Este modelo utiliza uma média das covariâncias das classes para formar uma matriz de covariância "agregada", sendo ideal para casos onde as variâncias individuais das classes são próximas, sua implementação também foi no modelo de Friedman quando o lambda era igual a um.
- **Classificador de Bayes Ingênuo**: Assumindo a independência entre as características, o modelo foi configurado para calcular a probabilidade de cada classe com base na média e variância individual de cada característica, utilizando uma matriz de covariância diagonal.
- **Classificador Gaussiano Regularizado (Friedman)**: Este modelo introduz uma regularização nas covariâncias, ajustando o hiperparâmetro λ. Foram testados os valores de $\lambda = \{0, 0.25, 0.5, 0.75, 1\}$ para verificar o impacto da regularização no desempenho do modelo.

3) **Validação e Avaliação**: Foi realizada uma validação cruzada através de simulações de Monte Carlo com 500 rodadas. Em cada rodada, o conjunto de dados teve seus índices embaralhos e foi dividido em 80% para treinamento e 20% para teste. A métrica de avaliação escolhida foi a acurácia, medida como a taxa de acertos entre as previsões e os rótulos reais.

4. Resultados

4.1. Problema de Regressão

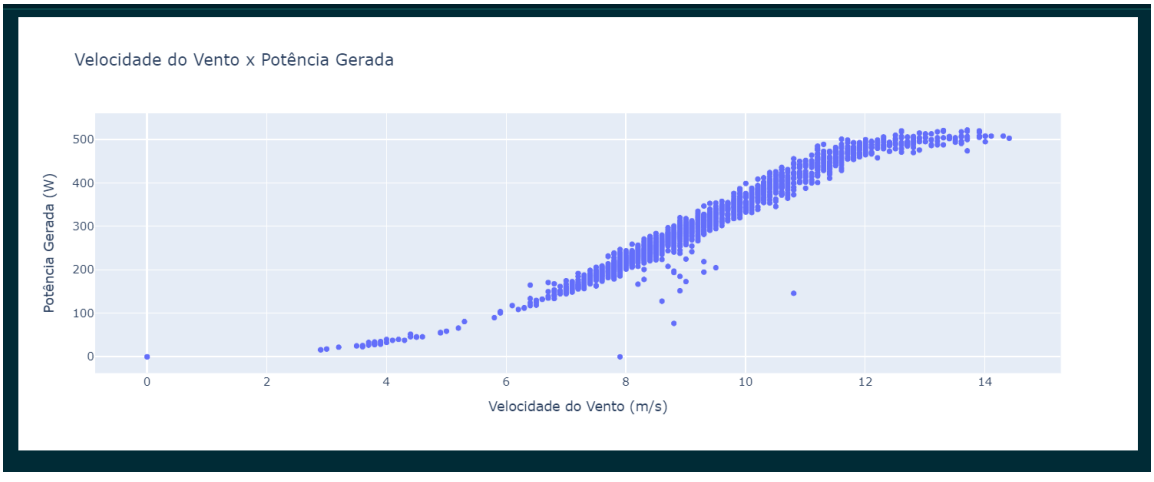

Fig. 1. Visualização inicial dos dados de amostra.

```
intercepto = -217.6903
coeficiente_angular = 56.4439
```

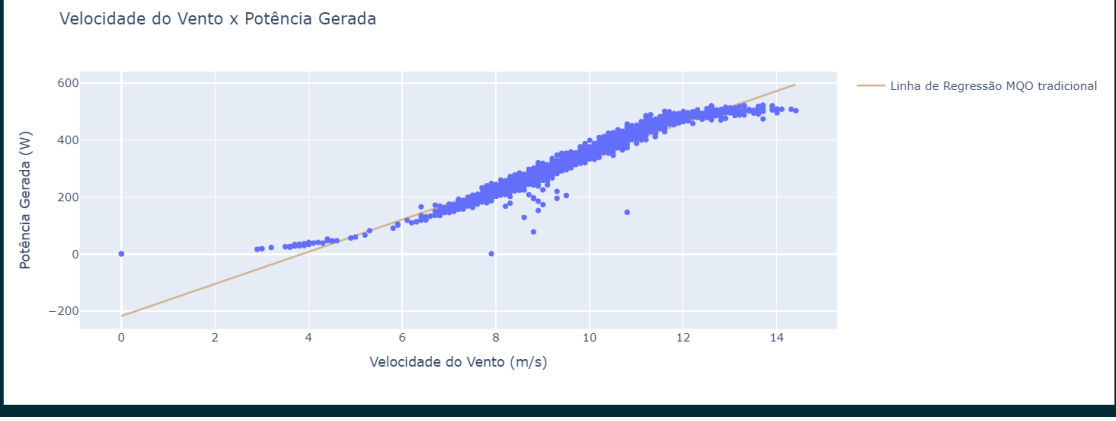

Fig. 2. Visualização dos dados de amostra sendo cortados pela reta regressora obtida a partir do Método dos Mínimos Quadrados Ordinários Tradicional.

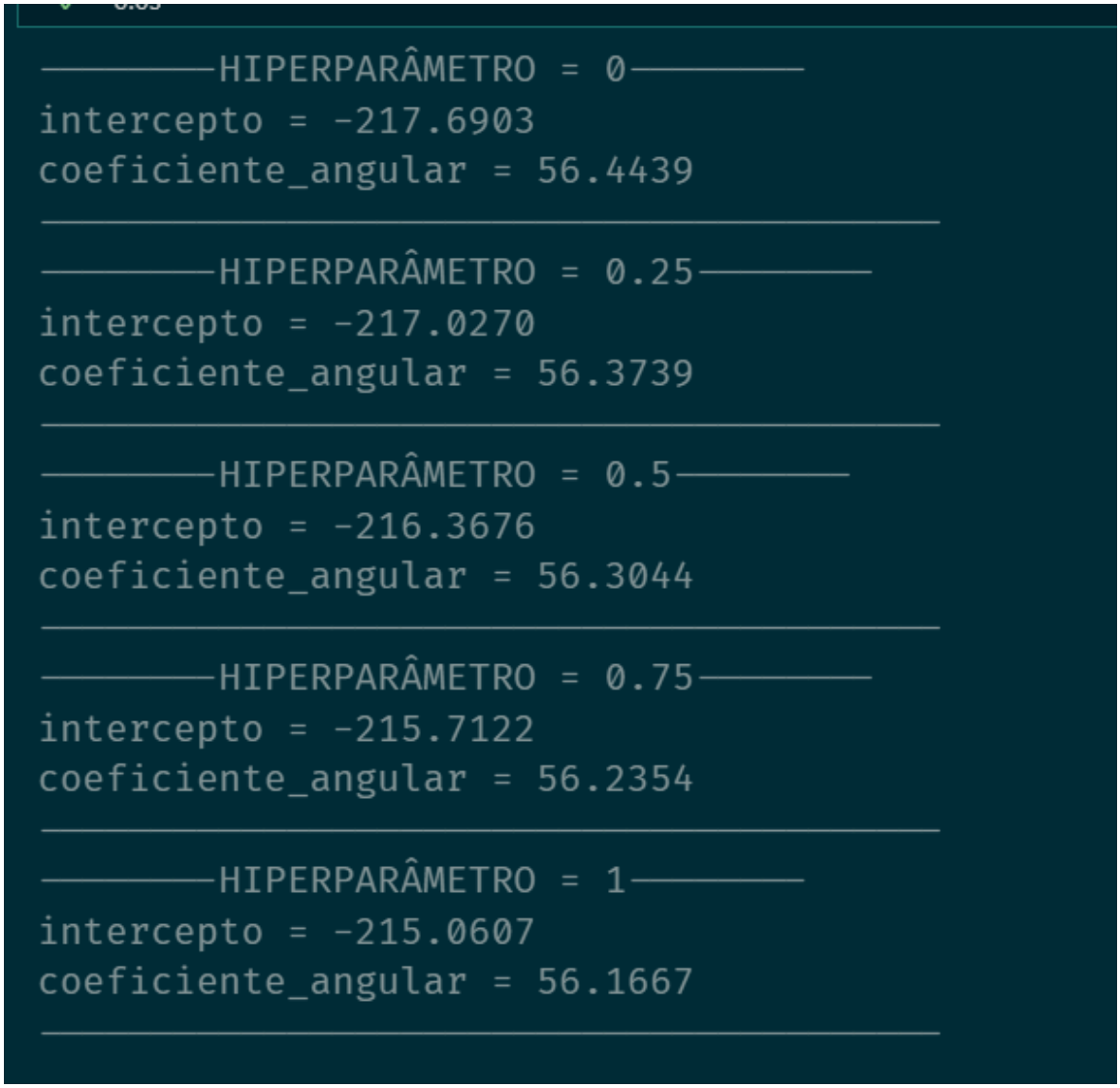

Fig. 3. Visualização dos dados de intercepto e coeficiente angular obtidos aplicando o Método dos Mínimos Quadrados Ordinários Regularizado.

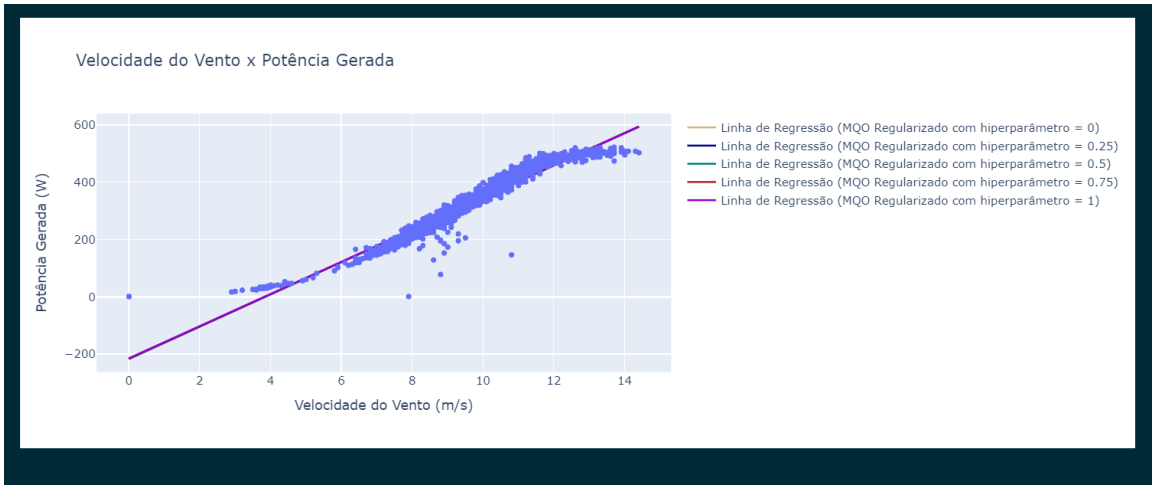

Fig. 4. Visualização dos dados de amostra sendo cortados pela reta regressora obtida a partir do Método dos Mínimos Quadrados Ordinários Regularizado.

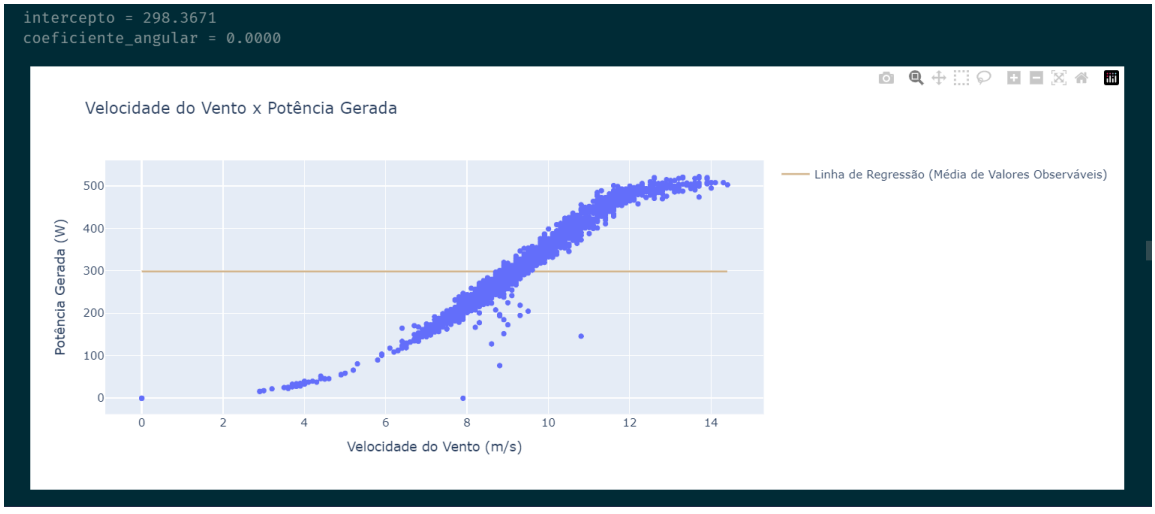

Fig. 5. Visualização dos dados de amostra sendo cortados pela reta regressora obtida a partir da Média dos Valores Observáveis.

Modelos	Média	Desvio-Padrão	Maior Valor	Menor Valor
MQO tradicional	359513.62	80154.48	638958.82	195766.48
MQO regularizado ($\lambda = 0.25$)	359441.97	79573.23	636253.22	196300.45
MQO regularizado ($\lambda = 0.5$)	359394.47	79001.90	633610.22	196849.34
MQO regularizado ($\lambda = 0.75$)	359370.58	78440.32	631028.58	197412.79
MQO regularizado ($\lambda = 1$)	359369.79	77888.31	628507.07	197990.44
Média dos valores observáveis	503093.97	272909.09	5891540.62	4148107.12

TABLE I
RESUMO ESTATÍSTICO DOS DIFERENTES MODELOS DE MQO

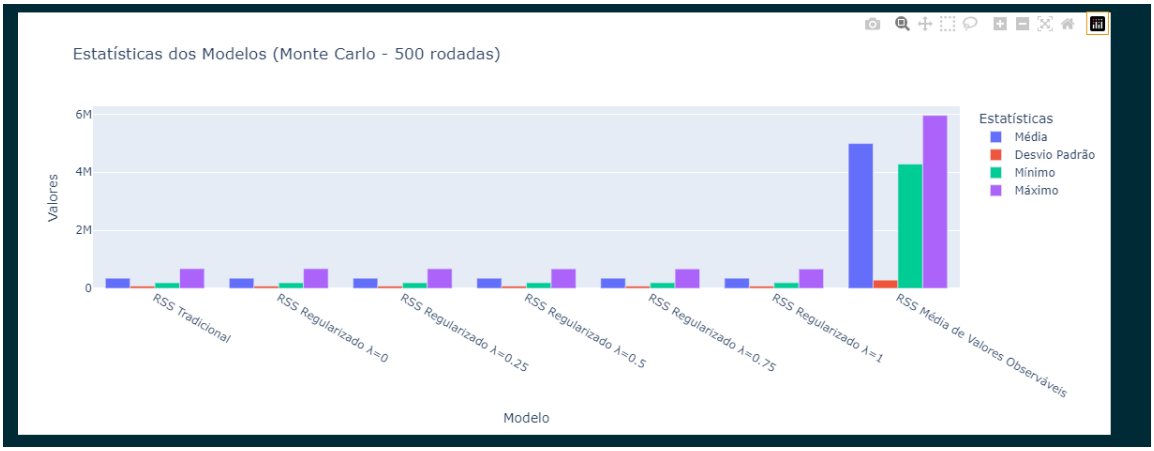

Fig. 6. Representação em forma de gráfico de barras dos dados do resumo estatístico acima

4.2. Problema de Classificação

Primeiro, foi feito o gráfico de dispersão de cada classe.

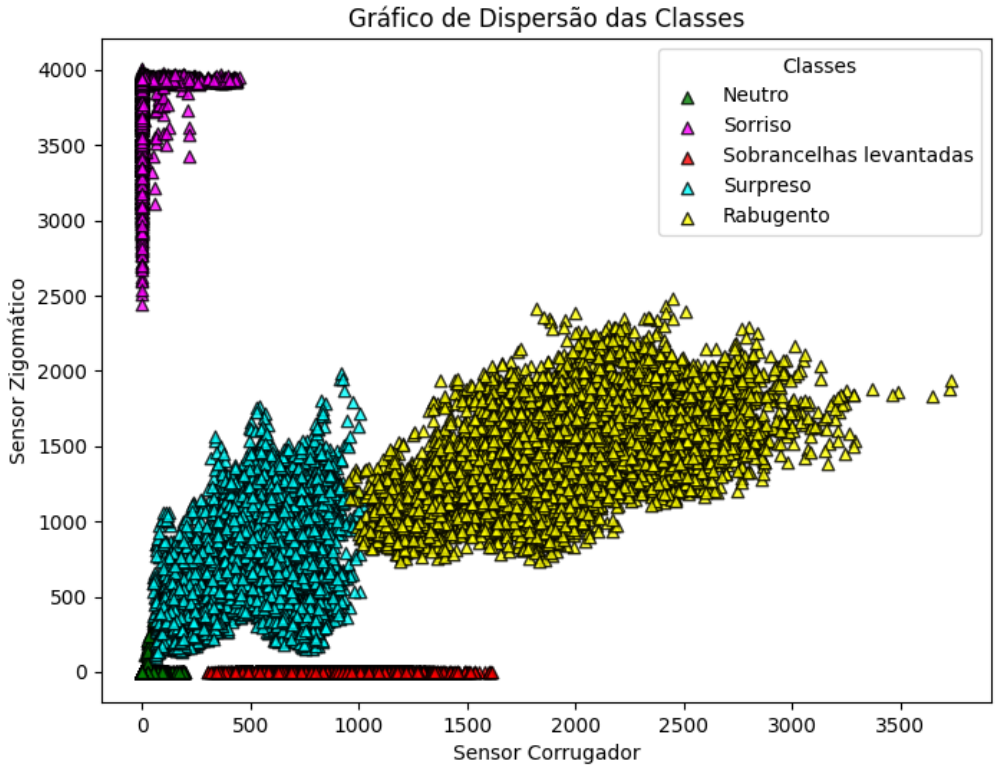

Fig. 7. Gráfico de Dispersão das Classes.

Os resultados dos modelos de classificação foram sintetizados em uma tabela (Tabela II), onde são apresentadas a acurácia média, o desvio padrão, e os valores máximo e mínimo de acurácia

obtidos durante as 500 rodadas de validação.

Modelos	Acurácia Média	Desvio-Padrão	Acurácia Máxima	Acurácia Mínima
MQO Tradicional	0.5459	0.0045	0.5633	0.5340
Gaussiano Tradicional	0.2001	0.0035	0.2105	0.1880
Gaussiano Covariância Iguais	0.9484	0.0020	0.9545	0.9415
Gaussiano Covariância Agregada	0.9626	0.0018	0.9673	0.9573
Bayes Ingênuo	0.9201	0.0028	0.9286	0.9116
Gaussiano Regularizado ($\lambda = 0.25$)	0.9748	0.0015	0.9794	0.9703
Gaussiano Regularizado ($\lambda = 0.5$)	0.9680	0.0017	0.9722	0.9623
Gaussiano Regularizado ($\lambda = 0.75$)	0.9649	0.0018	0.9696	0.9590

TABLE II
RESULTADOS DE ACURÁCIA E DESVIO PADRÃO DOS MODELOS DE CLASSIFICAÇÃO

Os modelos Gaussiano Regularizado, Covariância Agregada e com Covariâncias Iguais (LDA) foram os mais eficazes, apresentando alta acurácia média e baixa variação. O classificador MQOT e o Tradicional não conseguiram performar bem, apresentando baixas acurácias médias e indicando uma inadequação para este conjunto de dados.

5. Conclusão

5.1. Problema de Regressão

Os algoritmos de regressão foram implementados e executados com sucesso, e foi possível observar a eficiência de cada um, tanto em execuções únicas quanto quando submetida a uma validação por meio de Monte carlo.

A priori, é importante ressaltar que o problema requeria uma regressão linear simples, ou seja, a variável dependente é influenciada por somente uma variável independente. Portanto, os gráficos foram bidimensionais. Além disso, a visualização inicial dos dados mostrou que, de forma geral, quanto maior fosse a velocidade do vento, maior seria a potência gerada. Em outras palavras, o gráfico apresentava uma tendência de crescimento. Contudo, é notório que tal crescimento aumenta consideravelmente a partir do ponto em que a velocidade do vento iguala a 6 metros por segundo.

Avançando a discussão para tratar dos algoritmos de regressão, é evidente que a Média dos Valores Observáveis não conseguiu atingir resultados satisfatórios, pois, para obter uma reta passando por pontos o mais próximos possíveis de cada amostra, seria necessário ter um coeficiente angular diferente de zero. Ademais, com relação ao MQO, suas variantes atingiram resultados bastante parecidos, portanto, seria precipitado afirmar que uma se saiu melhor que a outra.

Mediante o exposto, essa atividade me permitiu ampliar meus horizontes ao me proporcionar a oportunidade de desenvolver e colocar em prática meus conhecimentos acerca do tema algoritmos de regressão.

5.2. Problema de Classificação

No contexto do problema de classificação, o Classificador Gaussiano Tradicional baseia-se na suposição de que cada classe segue uma distribuição normal multivariada, calculando a matriz de covariância individual de cada classe para construir uma função discriminante, essa, por sua vez, depende da inversa da matriz de covariância, que permite calcular a ”distância” entre as amostras e as médias das classes.

No entanto, esse método apresenta sérias limitações quando a matriz de covariância de uma ou mais classes é singular ou quase singular.

No caso deste trabalho, o Classificador Gaussiano Tradicional obteve a pior acurácia, com média de 20,01%, o que é praticamente equivalente a uma classificação aleatória. Esse baixo desempenho indica que o modelo enfrentou dificuldades. A necessidade de inverter essas ma-

trizes resultou em previsões instáveis e pouco confiáveis.

Já os outros modelos de classificação implementados neste trabalho permitiram uma análise detalhada da capacidade de cada modelo em distinguir entre diferentes expressões faciais com base nos sinais de eletromiografia. Observou-se que o Classificador Gaussiano Regularizado com $\lambda = 0.25$ obteve o melhor desempenho, alcançando uma acurácia média de 97,48% e mostrando-se estável, com baixa variação.

O Classificador Gaussiano com Covariâncias Iguais (LDA) também apresentou resultados satisfatórios, com uma acurácia média de 94,84%, o que sugere que a suposição de covariâncias compartilhadas entre as classes é razoável para este conjunto de dados. Em contrapartida, modelos como o MQOT e o Tradicional demonstraram baixa eficácia, sugerindo que eles não conseguem lidar bem com as características das classes neste contexto.

Em conclusão, os resultados obtidos indicam que a regularização e a consideração da estrutura das covariâncias das classes são fatores determinantes para melhorar a acurácia em problemas de classificação envolvendo dados de eletromiografia. Este estudo possibilitou uma compreensão mais aprofundada sobre a adequação de diferentes modelos de classificação, bem como sobre o impacto do ajuste de hiperparâmetros na performance dos algoritmos.

6. Códigos

6.1. Problema de Regressão

```
import pandas as pd
import numpy as np
import plotly.express as exp
import plotly.graph_objects as go
from abc import ABC, abstractmethod

pd.set_option('display.float_format', '{:.2f}'.format)

# lendo os dados do arquivo `dados/aerogerador.dat`
dados: pd.DataFrame = (
    pd.read_csv(
        "dados/aerogerador.dat",
        sep=r'\s+|\t', # corresponder a uma ou mais ocorrências de espaços em branc
        engine='python', # a engine c, que é padrão, não suporta lidar com regex.
        header=None, # evita que a primeira linha seja interpretada como cabeçalho
        names=("velocidade do vento", "potência gerada")
    )
)

fig = exp.scatter(
    dados,
    x='velocidade do vento',
    y='potência gerada',
    title="Velocidade do Vento x Potência Gerada",
    labels={
        'velocidade do vento':
            'Velocidade do Vento (m/s)',
        'potência gerada':
            'Potência Gerada (W)'
    }
)
```

```
fig.show()

# função para mostrar o gráfico e restituí-lo
def mostrar_grafico() -> None:
    fig.show()

    # removendo o traço da figura
    fig.data = fig.data[:1]

class MetodoRegressao(ABC):
    global fig

    def __init__(self, x:np.ndarray, y: np.ndarray, *args):
        self.x = x
        self.y = y

    @abstractmethod
    def regredir(
        self
    ) -> tuple[float,float]:
        '''retornar o intercepto e o coeficiente angular, nessa ordem'''
        ...

    @abstractmethod
    def construir_grafico(
        self,
        mostrar: bool = True,
        cor: str = 'tan'
    ) -> None: ...

class MMQO_Tradicional(MetodoRegressao):
    def regredir(self) -> tuple[float, float]:
        # Reformatação para garantir que x e y sejam matrizes colunas
        x = self.x.reshape(-1, 1)
        y = self.y.reshape(-1, 1)
        # -1 é a mesma coisa que len(x) ou len(y) nesses casos

        # Adicionando uma coluna de 1s para o cálculo do intercepto (termo constant
        X: np.ndarray = np.concatenate(
            [np.ones((len(x), 1)), x],
            axis=1
        )

        # Estimação dos coeficientes usando a fórmula do MMQO tradicional (beta = (
        self.B: np.ndarray = np.linalg.pinv(X.T @ X) @ X.T @ y

        # Coeficientes da regressão
        intercepto: float = float(self.B[0][0])
        coeficiente_angular: float = float(self.B[1][0])

        return intercepto, coeficiente_angular
```

```
    def construir_grafico(self, mostrar: bool = True, cor: str = 'tan') -> None:
        # Gerando pontos para a linha de regressão
        x_pred: np.ndarray = np.linspace(np.min(self.x), np.max(self.x), 100)
        X_pred: np.ndarray = np.concatenate([np.ones((len(x_pred), 1)), x_pred.resh
        y_pred: np.ndarray = X_pred @ self.B

        # Adicionando a linha de regressão ao gráfico
        fig.add_trace(
            go.Scatter(
                x=x_pred.flatten(),
                y=y_pred.flatten(),
                mode='lines',
                name='Linha de Regressão MQO tradicional',
                line={'color': cor}
            )
        )

        if mostrar: mostrar_grafico()

mmqo = MMQO_Tradicional(
    x=np.array(dados['velocidade do vento']), # variável INdependente
    y=np.array(dados['potência gerada']) # Variável dependente
)

intercepto, coeficiente_angular = mmqo.regredir()

print(f'{intercepto = :.4f}\n{coeficiente_angular = :.4f}')

del(intercepto) ; del(coeficiente_angular)

mmqo.construir_grafico()

del(mmqo)

class MMQO_Regularizado(MetodoRegressao):
    def __init__(self, x, y, hiperparametro: float) -> None:
        self.hp = hiperparametro
        super().__init__(x, y)

    def regredir(self) -> tuple[float, float]:
        # Reformatação para garantir que x e y sejam matrizes colunas
        x = self.x.reshape(-1, 1)
        y = self.y.reshape(-1, 1)
        # -1 é a mesma coisa que len(x) ou len(y) nesses casos

        # Adicionando uma coluna de 1s para o cálculo do intercepto (termo constant
        X: np.ndarray = np.concatenate(
            [np.ones((len(x), 1)), x],
            axis=1
        )

        # construção da matriz identidade
```

```
        I: np.ndarray = np.eye(X.shape[1])

        # Estimação dos coeficientes usando a fórmula do MMQO regularizado (beta =
        self.B: np.ndarray = np.linalg.pinv((X.T @ X) + hp * I) @ X.T @ y

        # Coeficientes da regressão
        intercepto: float = float(self.B[0][0])
        coeficiente_angular: float = float(self.B[1][0])

        return intercepto, coeficiente_angular

    def construir_grafico(self, mostrar: bool = True, cor: str = 'tan') -> None:
        # Gerando pontos para a linha de regressão
        x_pred: np.ndarray = np.linspace(np.min(self.x), np.max(self.x), 100)
        X_pred: np.ndarray = np.concatenate([np.ones((len(x_pred), 1)), x_pred.resh
        y_pred: np.ndarray = X_pred @ self.B

        # Adicionando a linha de regressão ao gráfico
        fig.add_trace(
            go.Scatter(
                x=x_pred.flatten(),
                y=y_pred.flatten(),
                mode='lines',
                name=f'Linha de Regressão (MQO Regularizado com hiperparâmetro = {s
                line={'color': cor}
            )
        )

        if mostrar: mostrar_grafico()

hiperparametros: tuple[float] = 0, .25, .5, .75, 1
cores: tuple[str] = "tan", "darkblue", "teal", "firebrick", "darkviolet"

for hp, cor in zip(hiperparametros, cores):
    print(f'{'-'*8}HIPERPARÂMETRO = {hp}{'-'*8}')

    mmqo = MMQO_Regularizado(
        x=np.array(dados['velocidade do vento']), # variável INdependente
        y=np.array(dados['potência gerada']), # Variável dependente
        hiperparametro=hp
    )

    intercepto, coeficiente_angular = mmqo.regredir()
    print(f'{intercepto = :.4f}\n{coeficiente_angular = :.4f}')

    mmqo.construir_grafico(mostrar=False, cor=cor)

    del(intercepto) ; del(coeficiente_angular) ; del(mmqo)

    print('-'*40)

del(hp) ; del(cor) ; del(hiperparametros) ; del(cores)
```

```
mostrar_grafico()

class MediaValoresObservaveis(MetodoRegressao):
    def regredir(self) -> tuple[float, float]:
        # intercepto e coeficiente angular
        return float(np.mean(self.y)), 0.0

    def construir_grafico(self, mostrar = True, cor = 'tan') -> None:
        # Gerando pontos para a linha de regressão
        x_pred: np.ndarray = np.linspace(np.min(self.x), np.max(self.x), 100)
        y_pred: np.ndarray = np.mean(self.y) + 0.0 * x_pred

        # Adicionando a linha de regressão ao gráfico
        fig.add_trace(
            go.Scatter(
                x=x_pred.flatten(),
                y=y_pred.flatten(),
                mode='lines',
                name='Linha de Regressão (Média de Valores Observáveis)',
                line={'color': cor}
            )
        )

        if mostrar: mostrar_grafico()

medias = MediaValoresObservaveis(
    x=np.array(dados['velocidade do vento']), # variável INdependente
    y=np.array(dados['potência gerada']), # Variável dependente
)

intercepto, coeficiente_angular = medias.regredir()

print(f'{intercepto = :.4f}\n{coeficiente_angular = :.4f}')

del(intercepto) ; del(coeficiente_angular)

medias.construir_grafico()

del(medias)

# Definindo o número de rodadas da simulação
R: int = 500

# Inicializar listas para armazenar o RSS de cada modelo em cada rodada
rss_mqoTradicional: list[float] = []
rss_mqoRegularizado_0: list[float] = []
rss_mqoRegularizado_025: list[float] = []
rss_mqoRegularizado_05: list[float] = []
rss_mqoRegularizado_075: list[float] = []
rss_mqoRegularizado_1: list[float] = []
rss_media: list[float] = []
```

```
# Definindo hiperparâmetros para o modelo regularizado
hiperparametros: tuple[float] = 0, 0.25, 0.5, 0.75, 1
rss_regularizados: dict[float, list[float]] = {
    0: rss_mqoRegularizado_0,
    0.25: rss_mqoRegularizado_025,
    0.5: rss_mqoRegularizado_05,
    0.75: rss_mqoRegularizado_075,
    1: rss_mqoRegularizado_1
}

# Número da última amostra de treinamento
n80: int = int(len(dados) * 0.8)

def calcular_y_preditor(
    intercepto: float,
    coeficiente_angular: float,
    x_validacao: np.ndarray
) -> np.ndarray:
    return intercepto + coeficiente_angular * x_validacao

def calcular_rss(
    y_validacao: np.ndarray,
    y_preditor: np.ndarray
) -> float:
    return float(np.sum((y_validacao - y_preditor) ** 2))

# Loop de simulação de Monte Carlo
for _ in range(R):
    # Amostra embaralhada de dados para cada iteração
    dados_embaralhados: pd.DataFrame = dados.sample(frac=1, ignore_index=True)

    dados_treinamento: pd.DataFrame = dados_embaralhados.iloc[:n80]
    dados_validacao: pd.DataFrame = dados_embaralhados.iloc[n80:]

    # Separação das variáveis de treinamento e validação
    x_treinamento: np.ndarray = np.array(dados_treinamento['velocidade do vento']).
    y_treinamento: np.ndarray = np.array(dados_treinamento['potência gerada']).resh
    x_validacao: np.ndarray = np.array(dados_validacao['velocidade do vento']).resh
    y_validacao: np.ndarray = np.array(dados_validacao['potência gerada']).reshape(

    # MQO Tradicional
    regressao: MetodoRegressao = MMQO_Tradicional(x_treinamento, y_treinamento)
    intercepto, coeficiente_angular = regressao.regredir()
    y_preditor: np.ndarray = calcular_y_preditor(intercepto, coeficiente_angular, x
    rss: float = calcular_rss(y_validacao, y_preditor)
    rss_mqoTradicional.append(rss)

    # MQO Regularizado para cada valor de hiperparâmetro
    for hp in hiperparametros:
        regressao: MetodoRegressao = MMQO_Regularizado(x_treinamento, y_treinamento
        intercepto, coeficiente_angular = regressao.regredir()
```

```
        y_preditor: np.ndarray = calcular_y_preditor(intercepto, coeficiente_angula
        rss: float = calcular_rss(y_validacao, y_preditor)
        rss_regularizados[hp].append(rss)

    # Média dos Valores Observáveis
    regressao: MetodoRegressao = MediaValoresObservaveis(x_treinamento, y_treinamen
    intercepto, coeficiente_angular = regressao.regredir()
    y_preditor: np.ndarray = calcular_y_preditor(intercepto, coeficiente_angular, x
    rss: float = calcular_rss(y_validacao, y_preditor)
    rss_media.append(rss)

# Resultados finais: listas de RSS para cada modelo em cada rodada
rss_results: dict[str, list[float]] = {
    "RSS Tradicional": rss_mqoTradicional,
    "RSS Regularizado =0": rss_mqoRegularizado_0,
    "RSS Regularizado =0.25": rss_mqoRegularizado_025,
    "RSS Regularizado =0.5": rss_mqoRegularizado_05,
    "RSS Regularizado =0.75": rss_mqoRegularizado_075,
    "RSS Regularizado =1": rss_mqoRegularizado_1,
    "RSS Média de Valores Observáveis": rss_media,
}

# Convertendo para DataFrame para facilitar a visualização
df_rss_results: pd.DataFrame = pd.DataFrame(rss_results)
df_rss_results.describe()

# Usando os dados do describe() para visualização
df_stats = df_rss_results.describe().T  # Transpor para que as estatísticas sejam c

# Criar um gráfico de barras com as estatísticas 'mean', 'std', 'min' e 'max' para
fig = go.Figure()

# Adicionando as estatísticas ao gráfico
fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['mean'],
    name='Média'
))

fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['std'],
    name='Desvio Padrão'
))

fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['min'],
    name='Mínimo'
))

fig.add_trace(go.Bar(
```

```
    x=df_stats.index,
    y=df_stats['max'],
    name='Máximo'
))

# Configurações do layout
fig.update_layout(
    title="Estatísticas dos Modelos (Monte Carlo - 500 rodadas)",
    xaxis_title="Modelo",
    yaxis_title="Valores de RSS",
    barmode='group',  # Para agrupar as barras
    legend_title="Estatísticas"
)

fig.show()

# Definindo o número de rodadas da simulação
R: int = 500

# Inicializar listas para armazenar o RSS de cada modelo em cada rodada
rss_mqoTradicional: list[float] = []
rss_mqoRegularizado_0: list[float] = []
rss_mqoRegularizado_025: list[float] = []
rss_mqoRegularizado_05: list[float] = []
rss_mqoRegularizado_075: list[float] = []
rss_mqoRegularizado_1: list[float] = []
rss_media: list[float] = []

# Definindo hiperparâmetros para o modelo regularizado
hiperparametros: tuple[float] = (0, 0.25, 0.5, 0.75, 1)
rss_regularizados = {
    0: rss_mqoRegularizado_0,
    0.25: rss_mqoRegularizado_025,
    0.5: rss_mqoRegularizado_05,
    0.75: rss_mqoRegularizado_075,
    1: rss_mqoRegularizado_1
}

# Número da última amostra de treinamento
n80: int = int(len(dados) * 0.8)

# Loop de simulação de Monte Carlo
for _ in range(R):
    # Amostra embaralhada de dados para cada iteração
    dados_embaralhados: pd.DataFrame = dados.sample(frac=1, ignore_index=True)

    dados_treinamento: pd.DataFrame = dados_embaralhados.iloc[:n80]
    dados_validacao: pd.DataFrame = dados_embaralhados.iloc[n80:]

    # Separação das variáveis de treinamento e validação
    x_treinamento: np.ndarray = np.array(dados_treinamento['velocidade do vento']).
    y_treinamento: np.ndarray = np.array(dados_treinamento['potência gerada']).resh
```

```
    x_validacao: np.ndarray = np.array(dados_validacao['velocidade do vento']).resh
    y_validacao: np.ndarray = np.array(dados_validacao['potência gerada']).reshape(

    # MQO Tradicional
    regressao = MMQO_Tradicional(x_treinamento, y_treinamento)
    intercepto, coeficiente_angular = regressao.regredir()
    y_pred_trad = intercepto + coeficiente_angular * x_validacao
    rss_mqoTradicional.append(float(np.sum((y_validacao - y_pred_trad) ** 2)))

    # MQO Regularizado para cada valor de hiperparâmetro
    for hp in hiperparametros:
        regressao = MMQO_Regularizado(x_treinamento, y_treinamento, hiperparametro=
        intercepto, coeficiente_angular = regressao.regredir()
        y_preditor = intercepto + coeficiente_angular * x_validacao
        rss_regularizados[hp].append(float(np.sum((y_validacao - y_preditor) ** 2))

    # Média dos Valores Observáveis
    regressao = MediaValoresObservaveis(x_treinamento, y_treinamento)
    intercepto, coeficiente_angular = regressao.regredir()
    y_preditor = intercepto + coeficiente_angular * x_validacao
    rss_media.append(float(np.sum((y_validacao - y_preditor) ** 2)))

# Resultados finais: listas de RSS para cada modelo em cada rodada
rss_results: dict[str, list[float]] = {
    "RSS Tradicional": rss_mqoTradicional,
    "RSS Regularizado =0": rss_mqoRegularizado_0,
    "RSS Regularizado =0.25": rss_mqoRegularizado_025,
    "RSS Regularizado =0.5": rss_mqoRegularizado_05,
    "RSS Regularizado =0.75": rss_mqoRegularizado_075,
    "RSS Regularizado =1": rss_mqoRegularizado_1,
    "RSS Média de Valores Observáveis": rss_media,
}

# Convertendo para DataFrame para facilitar a visualização
df_rss_results: pd.DataFrame = pd.DataFrame(rss_results)
df_rss_results.describe()

# Criando um gráfico de barras para efeito de visualização

# Usando os dados do describe() para visualização
df_stats = df_rss_results.describe().T  # Transpor para que as estatísticas sejam c

# Criar um gráfico de barras com as estatísticas 'mean', 'std', 'min' e 'max' para
fig = go.Figure()

# Adicionando as estatísticas ao gráfico
fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['mean'],
    name='Média'
))
```

```
fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['std'],
    name='Desvio Padrão'
))

fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['min'],
    name='Mínimo'
))

fig.add_trace(go.Bar(
    x=df_stats.index,
    y=df_stats['max'],
    name='Máximo'
))

# Configurações do layout
fig.update_layout(
    title="Estatísticas dos Modelos (Monte Carlo - 500 rodadas)",
    xaxis_title="Modelo",
    yaxis_title="Valores",
    barmode='group',  # Para agrupar as barras
    legend_title="Estatísticas"
)

fig.show()
```

6.2. Problema de Classificação

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Optional

# Função customizada para dividir os dados em treino e teste
def dividir_treino_teste(caracteristicas, rotulos, proporcao_treino=0.8):
    indices = np.arange(caracteristicas.shape[0])
    np.random.shuffle(indices)
    indice_fim_treino = int(proporcao_treino * len(indices))
    caracteristicas_treino = caracteristicas[indices[:indice_fim_treino]]
    caracteristicas_teste = caracteristicas[indices[indice_fim_treino:]]
    rotulos_treino = rotulos[indices[:indice_fim_treino]]
    rotulos_teste = rotulos[indices[indice_fim_treino:]]
    return caracteristicas_treino, caracteristicas_teste, rotulos_treino, rotulos_t

# Carregar o conjunto de dados com numpy
dados = np.loadtxt("EMGsDataset.csv", delimiter=",")

# Organizar os dados em características (X) e rótulos (y)
sensor_corrugador = dados[0, :]  # Sensor 1 - Corrugador do Supercílio
```

```
sensor_zigomatico = dados[1, :]  # Sensor 2 - Zigomático Maior
rotulos = dados[2, :].astype(int)  # Rótulos das classes

# Criar matriz de características (X)
caracteristicas = np.column_stack((sensor_corrugador, sensor_zigomatico)) # N x p

# Definir cores e nomes para as cinco classes
cores_classes = ['green', 'magenta', 'red', 'cyan', 'yellow']
nomes = ['Neutro', 'Sorriso', 'Sobrancelhas levantadas', 'Surpreso', 'Rabugento']
nomes_classes = [f"{classe}" for classe in nomes]

classes_unicas = np.unique(rotulos)

# Plotar o gráfico de dispersão
plt.figure(figsize=(8, 6))
for indice, classe in enumerate(classes_unicas):
    pontos_classe = caracteristicas[rotulos == classe]
    plt.scatter(pontos_classe[:, 0], pontos_classe[:, 1],
                color=cores_classes[indice], label=nomes_classes[indice], alpha=0.8

plt.xlabel("Sensor Corrugador")
plt.ylabel("Sensor Zigomático")
plt.title("Gráfico de Dispersão das Classes")
plt.legend(title="Classes")
plt.show()

# Função para calcular a Distância de Mahalanobis Quadrática - Ajuda a escolher a m
def distancia_mahalanobis(amostra, media_classe, covariancia_inversa):
    diferenca = amostra - media_classe
    return np.dot(np.dot(diferenca.T, covariancia_inversa), diferenca)

# Classe MQOT para classificação
class ClassificadorMQOT:
    def __init__(self, caracteristicas: np.ndarray, rotulos: np.ndarray) -> None:
        self.caracteristicas = caracteristicas
        self.rotulos = rotulos
        self.coeficientes: Optional[np.ndarray] = None
        self.classes = np.unique(rotulos)

    def treinar(self) -> None:
        rotulos_binarios = np.array([self.rotulos == classe for classe in self.clas
        self.coeficientes = np.linalg.pinv(self.caracteristicas.T @ self.caracteris

    def prever(self, novas_caracteristicas: np.ndarray) -> np.ndarray:
        if self.coeficientes is None:
            self.treinar()
        previsoes = novas_caracteristicas @ self.coeficientes
        indice_classes = np.argmax(previsoes, axis=1)
        return self.classes[indice_classes]

# Classificador Naive Bayes Gaussiano
class ClassificadorBayesIngenuoGaussiano:
```

```
    def __init__(self):
        self.classes = None
        self.medias_classes = {}
        self.covariancias_classes = {}
        self.prioris_classes = {}

    def treinar(self, caracteristicas, rotulos):
        self.classes = np.unique(rotulos)
        for classe in self.classes:
            caracteristicas_classe = caracteristicas[rotulos == classe]
            self.medias_classes[classe] = np.mean(caracteristicas_classe, axis=0)
            variancias = np.var(caracteristicas_classe, axis=0)
            self.covariancias_classes[classe] = np.diag(variancias)
            self.prioris_classes[classe] = len(caracteristicas_classe) / len(rotulo

    def prever(self, novas_caracteristicas):
        previsoes = []
        for amostra in novas_caracteristicas:
            probabilidades_classe = []
            for classe in self.classes:
                media_classe = self.medias_classes[classe]
                covariancia_classe = self.covariancias_classes[classe]
                probabilidade = distancia_mahalanobis(amostra, media_classe, np.lin
                probabilidades_classe.append(probabilidade)
            previsoes.append(self.classes[np.argmin(probabilidades_classe)])
        return np.array(previsoes)

# Classificador LDA - Covariâncias Iguais
class ClassificadorLDA:
    def __init__(self):
        self.classes = None
        self.medias_classes = {}
        self.covariancia_geral = None
        self.prioris_classes = {}

    def treinar(self, caracteristicas, rotulos):
        self.classes = np.unique(rotulos)
        self.covariancia_geral = np.cov(caracteristicas, rowvar=False)
        for classe in self.classes:
            caracteristicas_classe = caracteristicas[rotulos == classe]
            self.medias_classes[classe] = np.mean(caracteristicas_classe, axis=0)
            self.prioris_classes[classe] = len(caracteristicas_classe) / len(rotulo

    def prever(self, novas_caracteristicas):
        covariancia_inversa = np.linalg.pinv(self.covariancia_geral)
        previsoes = []
        for amostra in novas_caracteristicas:
            distancias_classe = []
            for classe in self.classes:
                media_classe = self.medias_classes[classe]
                distancia = distancia_mahalanobis(amostra, media_classe, covarianci
                distancias_classe.append(distancia)
```

```
            previsoes.append(self.classes[np.argmin(distancias_classe)])
        return np.array(previsoes)

# Função para calcular a matriz de covariância agregada
def calcular_covariancia_agregada(caracteristicas_treino, rotulos_treino, num_class
    covariancias = []
    for classe in range(1, num_classes + 1):
        dados_classe = caracteristicas_treino[rotulos_treino == classe]
        covariancias.append(np.cov(dados_classe, rowvar=False))
    return np.mean(covariancias, axis=0)

# Função para calcular as covariâncias regularizadas por Friedman
def calcular_covariancias_regularizadas_friedman(caracteristicas_treino, rotulos_tr
    num_classes = len(np.unique(rotulos_treino))
    num_amostras = len(caracteristicas_treino)
    covariancia_agregada = calcular_covariancia_agregada(caracteristicas_treino, ro
    medias_classes = {}
    covariancias_regularizadas = {}
    inversas_covariancias = {}
    determinantes_covariancias = {}

    for classe in range(1, num_classes + 1):
        dados_classe = caracteristicas_treino[rotulos_treino == classe]
        num_amostras_classe = len(dados_classe)
        media_classe = np.mean(dados_classe, axis=0)
        covariancia_classe = np.cov(dados_classe, rowvar=False)

        medias_classes[classe] = media_classe
        covariancias_regularizadas[classe] = {}
        inversas_covariancias[classe] = {}
        determinantes_covariancias[classe] = {}

        for lam in lambdas:
            covariancia_reg = ((1 - lam) * num_amostras_classe * covariancia_classe
            covariancias_regularizadas[classe][lam] = covariancia_reg
            inversas_covariancias[classe][lam] = np.linalg.pinv(covariancia_reg)
            determinantes_covariancias[classe][lam] = np.linalg.det(covariancia_reg

    return medias_classes, covariancias_regularizadas, inversas_covariancias, deter

# Função para prever a classe usando o Classificador Gaussiano com regularização po
def prever_gaussiano_friedman(amostra, medias_classes, inversas_covariancias, deter
    probabilidades = []
    for classe in medias_classes:
        media_classe = medias_classes[classe]
        diferenca = amostra - media_classe
        cov_inversa = inversas_covariancias[classe][lambda_]
        det_cov = determinantes_covariancias[classe][lambda_]

        discriminante = -0.5 * (diferenca @ cov_inversa @ diferenca.T) - 0.5 * np.l
        probabilidades.append(discriminante)
```

```
    return np.argmax(probabilidades) + 1

# Parâmetros para validação Monte Carlo
num_rodadas = 500  # Número de rodadas para validação Monte Carlo - 10 pra teste, 5
lambdas = [0, 0.25, 0.5, 0.75, 1] # Invés de fazer um para o Tradicional
acuracias_lda = []
acuracias_bayes = []
acuracias_mqot = []
acuracias_gaussiano_reg = {lmbd: [] for lmbd in lambdas}

# Loop para Monte Carlo com várias rodadas
for _ in range(num_rodadas):
    caracteristicas_treino, caracteristicas_teste, rotulos_treino, rotulos_teste =

    # Classificador Gaussiano (Covariâncias Iguais)
    lda = ClassificadorLDA()
    lda.treinar(caracteristicas_treino, rotulos_treino)
    previsoes = lda.prever(caracteristicas_teste)
    acuracia = np.mean(rotulos_teste == previsoes)
    acuracias_lda.append(acuracia)

    # Classificador Bayes Ingênuo Gaussiano
    bayes_gaussiano = ClassificadorBayesIngenuoGaussiano()
    bayes_gaussiano.treinar(caracteristicas_treino, rotulos_treino)
    previsoes = bayes_gaussiano.prever(caracteristicas_teste)
    acuracia = np.mean(rotulos_teste == previsoes)
    acuracias_bayes.append(acuracia)

    # Classificador MQOT
    mqot = ClassificadorMQOT(caracteristicas_treino, rotulos_treino)
    mqot.treinar()
    previsoes = mqot.prever(caracteristicas_teste)
    acuracia = np.mean(rotulos_teste == previsoes)
    acuracias_mqot.append(acuracia)

    # Classificador Gaussiano Regularizado Friedman
    medias_classes, covariancias_regularizadas, inversas_covariancias, determinante
    for lam in lambdas:
        previsoes = [prever_gaussiano_friedman(amostra, medias_classes, inversas_co
        acuracia = np.mean(rotulos_teste == previsoes)
        acuracias_gaussiano_reg[lam].append(acuracia)

# Cálculo das estatísticas de acurácia para os classificadores
resultados = {
    'Modelo': [],
    'Acurácia Média': [],
    'Desvio Padrão': [],
    'Acurácia Máxima': [],
    'Acurácia Mínima': []
}

def adicionar_resultados(nome_modelo, acuracias):
```

```
    resultados['Modelo'].append(nome_modelo)
    resultados['Acurácia Média'].append(np.mean(acuracias))
    resultados['Desvio Padrão'].append(np.std(acuracias))
    resultados['Acurácia Máxima'].append(np.max(acuracias))
    resultados['Acurácia Mínima'].append(np.min(acuracias))

# Adicionar resultados de cada classificador
adicionar_resultados('MQOT', acuracias_mqot)
adicionar_resultados('Bayes Ingênuo', acuracias_bayes)
adicionar_resultados('Covariância Iguais', acuracias_lda)
for lam, acuracias in acuracias_gaussiano_reg.items():
    if lam == 1:
        adicionar_resultados('Covariância Agregada', acuracias)
    elif lam == 0:
        adicionar_resultados('Tradicional', acuracias)
    else:
        adicionar_resultados(f'Regularizado (={lam})', acuracias)

# Configuração da tabela para exibir com matplotlib
fig, ax = plt.subplots(figsize=(10, 6))  # Tamanho da tabela
ax.axis('off')  # Remove os eixos do gráfico

# Dados para a tabela
colunas = ['Modelo', 'Acurácia Média', 'Desvio Padrão', 'Acurácia Máxima', 'Acuráci
linhas = [
    [resultados['Modelo'][i],
     f"{resultados['Acurácia Média'][i]:.4f}",
     f"{resultados['Desvio Padrão'][i]:.4f}",
     f"{resultados['Acurácia Máxima'][i]:.4f}",
     f"{resultados['Acurácia Mínima'][i]:.4f}"]
    for i in range(len(resultados['Modelo']))
]

# Criação da tabela
tabela = ax.table(cellText=linhas, colLabels=colunas, cellLoc='center', loc='center
tabela.auto_set_font_size(False)
tabela.set_fontsize(10)
tabela.scale(1.2, 1.2)

plt.title("Resultados de Acurácia dos Modelos", fontsize=14)
plt.show()
```