Projeto de Disciplina de Algoritmos de Inteligência Artificial para Classificação [25E1_2]

Aluno: Mateus Teixeira Ramos da Silva

Índice

- Parte 1. Print do módulo
- Parte 2. Download dos dados
- Parte 3. Exploração dos dados
- Parte 4. Modelos de classificação
- Parte 5. Comparação com gráfico de curva ROC
- Parte 6. Inferências
- Parte 7. Exportação do arquivo.pdf

Importar os pacotes

In [ ]:
```
# Importar as bibliotecas
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import warnings

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score, precision_score, recall_score, f1_score, \
                            confusion_matrix, roc_curve, roc_auc_score, auc
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, cross_val_predict, StratifiedKFold
from sklearn.preprocessing import RobustScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.svm import SVC

warnings.filterwarnings('ignore')
```

Parte 1. Print do módulo

Voltar ao início

1.1. Faça o módulo do Kaggle Intro to Machine Learning:

Comprove a finalização do módulo com um print que contenha data e identificação do aluno.

Resposta:

Requisito atendido no arquivo 'print_curso.png', presente no repositório (https://github.com/GitMateusTeixeira/ml_models/tree/main/infnet_classification_pd).

Parte 2. Download dos dados

Voltar ao início

2.1. Faça o download da base - esta é uma base real, apresentada no artigo:

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

Resposta:

O dataset se trata de um conjunto de vinhos portugueses, tintos e brancos, do qual possui diversas características e, ao final, é avaliado por sua qualidade.

O objetivo do presente trabalho é determinar (de forma adaptada) se o modelo é capaz de avaliar os vinhos com precisão, usando as características como base.

In [168…
```
PATH = 'data/'
FILE = 'winequalityN.csv'

wine_df = pd.read_csv(PATH + FILE)
```

In [169…
```
wine_df.columns = wine_df.columns.str.lower().str.replace(' ', '_')
```

2.2. Ela possui uma variável denominada "quality", uma nota de 0 a 10 que denota a qualidade do vinho. Crie uma nova variável, chamada "opinion" que será uma variável categórica igual à 0, quando quality for menor e igual à 5. O valor será 1, caso contrário. Desconsidere a variável quality para o restante da análise.

In [170…
```
# criar uma nova coluna a partir de 'quality' e aplicar as regras estipuladas
wine_df['opinion'] = wine_df['quality'].copy().apply(lambda row: 1 if row <= 5 else 0)
```

In [171…
```
# desconsiderar a coluna 'quality' para o resto da análise
wine_df = wine_df.drop(columns='quality')
```

In [172…
```
wine_df.sample(2)
```

Out[172…

	type	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfur_dioxide	density	ph	sulphates	alcohol
98	white	9.8	0.36	0.46	10.5	NaN	4.0	83.0	0.9956	2.89	0.30	10.1
1121	white	6.0	0.40	0.30	1.6	0.047	30.0	117.0	0.9931	3.17	0.48	10.1

Para as questões 2-5 usaremos apenas os vinhos do tipo "branco".

In [173…
```
# separar os vinhos brancos dos tintos
red = wine_df['type'] == 'red'
white = wine_df['type'] == 'white'
```

Parte 3. Exploração dos dados

Voltar ao início

3.1. Descreva as variáveis presentes na base. Quais são as variáveis? Quais são os tipos de variáveis (discreta, categórica, contínua)? Quais são as médias e desvios padrões?

Resposta:

O data set possui as seguintes colunas:

- **type:** Variável categórica. Define se o vinho é tinto (red) ou branco (white);
- **fixed_acidity:** Variável numérica categórica. Trata da acidez fixa do vinho (ácidos orgânicos);
- **volatile_acidity:** Variável numérica contínua. Trata da acidez volátil, como o ácido acético;
- **citric_acid:** Variável numérica contínua. Mostra a quantidade de ácido cítrico no vinho;
- **residual_sugar:** Variável numérica contínua. Qual o teor de açúcar residual, influencia a doçura;
- **chlorides:** Variável numérica contínua. Mostra a quantidade de cloretos no vinho, pode indicar impurezas;
- **free_sulfur_dioxide:** Variável numérica contínua. Trata da quantidade de dióxido de enxofre livre, que preserva o vinho;
- **total_sulfur_dioxide:** Variável numérica contínua. Mostra o total de dióxido de enxofre (livre + combinado);
- **density:** Variável numérica contínua. Trata da densidade do vinho, correlacionada com álcool e açúcar;
- **ph:** Variável numérica contínua. Exibe a medida de acidez ou alcalinidade do vinho;
- **sulphates:** Variável numérica contínua. Mostra a quantidade de sulfatos, influencia sabor e aroma;
- **alcohol:** Variável numérica contínua. Indica o percentual de álcool no vinho;
- **quality:** Variável numérica contínua. Se trata da avaliação da qualidade do vinho (escala de 0 a 10);
- **opinion:** Variável numérica categórica. Coluna criada para adapatar uma categoria de qualidade (0 para ruim, 1 para bom).

Após uma análise das variáveis, descobriu-se que existem dados nulos em 7 colunas ao todo (considerando brancos e tintos), além de linhas duplicadas que precisam ser tratadas.

In [174…
```
# detalhar a base de dados
def check(df):
    l = []
    colunas = df.columns

    for col in colunas:
        dtypes = df[col].dtypes
        nunique = df[col].nunique()
        sum_null = df[col].isnull().sum()

        # calcular a moda e a frequência da moda
        moda = df[col].mode().iloc[0] if not df[col].mode().empty else 'n/a'
        moda_freq = df[col].value_counts().iloc[0] if not df[col].value_counts().empty else 'n/a'

        if np.issubdtype(dtypes, np.number):
            status = df.describe(include='all').T
            media = status.loc[col, 'mean']
            std = status.loc[col, 'std']
            min_val = status.loc[col, 'min']
            quar1 = status.loc[col, '25%']
            median = df[col].median()
```

```
            quar3 = status.loc[col, '75%']
            max_val = status.loc[col, 'max']

        else:
            # definir 'n/a' de 'não se aplica'
            status = media = std = min_val = quar1 = median = quar3 = max_val = 'n/a'

        l.append([col, dtypes, nunique, sum_null, media, std, min_val, quar1, median, quar3, max_val, moda, moda_freq])

    # criar o DataFrame com as novas colunas para exibição
    df_check = pd.DataFrame(l)
    df_check.columns = ['coluna', 'tipo', 'únicos', 'null_soma', 'media', 'desvio',
                        'minimo', '25%', 'mediana', '75%', 'maximo', 'moda', 'frequência_moda']

    return df_check
```

In [175…

```
# exibir a descrição completa de todas as colunas para os vinhos brancos
check(wine_df[white])
```

Out[175…

	coluna	tipo	únicos	null_soma	media	desvio	minimo	25%	mediana	75%	maximo	moda	frequência_moda
0	type	object	1	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	white	4898
1	fixed_acidity	float64	68	8	6.855532	0.843808	3.8	6.3	6.8	7.3	14.2	6.8	308
2	volatile_acidity	float64	125	7	0.278252	0.100811	0.08	0.21	0.26	0.32	1.1	0.28	263
3	citric_acid	float64	87	2	0.33425	0.120985	0.0	0.27	0.32	0.39	1.66	0.3	307
4	residual_sugar	float64	310	2	6.39325	5.072275	0.6	1.7	5.2	9.9	65.8	1.2	187
5	chlorides	float64	160	2	0.045778	0.02185	0.009	0.036	0.043	0.05	0.346	0.044	201
6	free_sulfur_dioxide	float64	132	0	35.308085	17.007137	2.0	23.0	34.0	46.0	289.0	29.0	160
7	total_sulfur_dioxide	float64	251	0	138.360657	42.498065	9.0	108.0	134.0	167.0	440.0	111.0	69
8	density	float64	890	0	0.994027	0.002991	0.98711	0.991723	0.99374	0.9961	1.03898	0.992	64
9	ph	float64	103	7	3.188203	0.151014	2.72	3.09	3.18	3.28	3.82	3.14	172
10	sulphates	float64	79	2	0.489835	0.114147	0.22	0.41	0.47	0.55	1.08	0.5	248
11	alcohol	float64	103	0	10.514267	1.230621	8.0	9.5	10.4	11.4	14.2	9.4	229
12	opinion	int64	2	0	0.334831	0.471979	0.0	0.0	0.0	1.0	1.0	0	3258

In [176…

```
# exibir a descrição completa de todas as colunas para os vinhos tintos
check(wine_df[red])
```

Out[176…

	coluna	tipo	únicos	null_soma	media	desvio	minimo	25%	mediana	75%	maximo	moda	frequência_moda
0	type	object	1	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	red	1599
1	fixed_acidity	float64	96	2	8.322104	1.740767	4.6	7.1	7.9	9.2	15.9	7.2	67
2	volatile_acidity	float64	143	1	0.527738	0.179085	0.12	0.39	0.52	0.64	1.58	0.6	47
3	citric_acid	float64	80	1	0.271145	0.194744	0.0	0.09	0.26	0.42	1.0	0.0	131
4	residual_sugar	float64	91	0	2.538806	1.409928	0.9	1.9	2.2	2.6	15.5	2.0	156
5	chlorides	float64	153	0	0.087467	0.047065	0.012	0.07	0.079	0.09	0.611	0.08	66
6	free_sulfur_dioxide	float64	60	0	15.874922	10.460157	1.0	7.0	14.0	21.0	72.0	6.0	138
7	total_sulfur_dioxide	float64	144	0	46.467792	32.895324	6.0	22.0	38.0	62.0	289.0	28.0	43
8	density	float64	436	0	0.996747	0.001887	0.99007	0.9956	0.99675	0.997835	1.00369	0.9972	36
9	ph	float64	89	2	3.310864	0.15429	2.74	3.21	3.31	3.4	4.01	3.3	57
10	sulphates	float64	96	2	0.658078	0.169594	0.33	0.55	0.62	0.73	2.0	0.6	69
11	alcohol	float64	65	0	10.422983	1.065668	8.4	9.5	10.2	11.1	14.9	9.5	139
12	opinion	int64	2	0	0.465291	0.49895	0.0	0.0	0.0	1.0	1.0	0	855

Exibindo os dados nulos e duplicados

In [177…

```
import pandas as pd

def shape(white_df, red_df):
    # Função auxiliar para extrair as informações
    def get_info(df):
        return {
            "Total Linhas": df.shape[0],
            "Linhas Únicas": len(df) - df.duplicated().sum(),
            "Linhas Duplicadas": df.duplicated().sum(),
            "Linhas com Nulos": df.isnull().any(axis=1).sum(),
            "": "", # pula uma linha
            "Total Colunas": df.shape[1],
```

```
            "Int64": sum(df.dtypes == "int64"),
            "Float64": sum(df.dtypes == "float64"),
            "Object": sum(df.dtypes == "object")
        }

    # Criar DataFrame com as informações dos vinhos brancos e tintos
    summary_df = pd.DataFrame({
        "White": get_info(white_df),
        "Red": get_info(red_df)
    })

    return summary_df
```

In [178…

```
# analisar os dados dos vinhos brancos e tintos
shape(wine_df[white], wine_df[red])
```

Out[178…

	White	Red
Total Linhas	4898	1599
Linhas Únicas	3970	1359
Linhas Duplicadas	928	240
Linhas com Nulos	28	6
Total Colunas	13	13
Int64	1	1
Float64	11	11
Object	1	1

É possível verificar que o dataset possui diversos dados nulos, além de linhas repetidas, que necessitam de tratamento.

Tratando os dados duplicados

Há, no total, 1168 linhas duplicadas, considerando vinhos tintos e brancos

In [179…

```
# remover duplicadas e transformar 'white' e 'red' em dataframes próprios
white = wine_df[white].drop_duplicates()
red = wine_df[red].drop_duplicates()
```

In [180…

```
# verificar as linhas duplicadas
shape(white, red)
```

Out[180…

	White	Red
Total Linhas	3970	1359
Linhas Únicas	3970	1359
Linhas Duplicadas	0	0
Linhas com Nulos	28	6
Total Colunas	13	13
Int64	1	1
Float64	11	11
Object	1	1

Tratando os dados nulos

Para tratar os dados nulos, primeiro vamos verificar a distribuição das variáveis em cada dataset

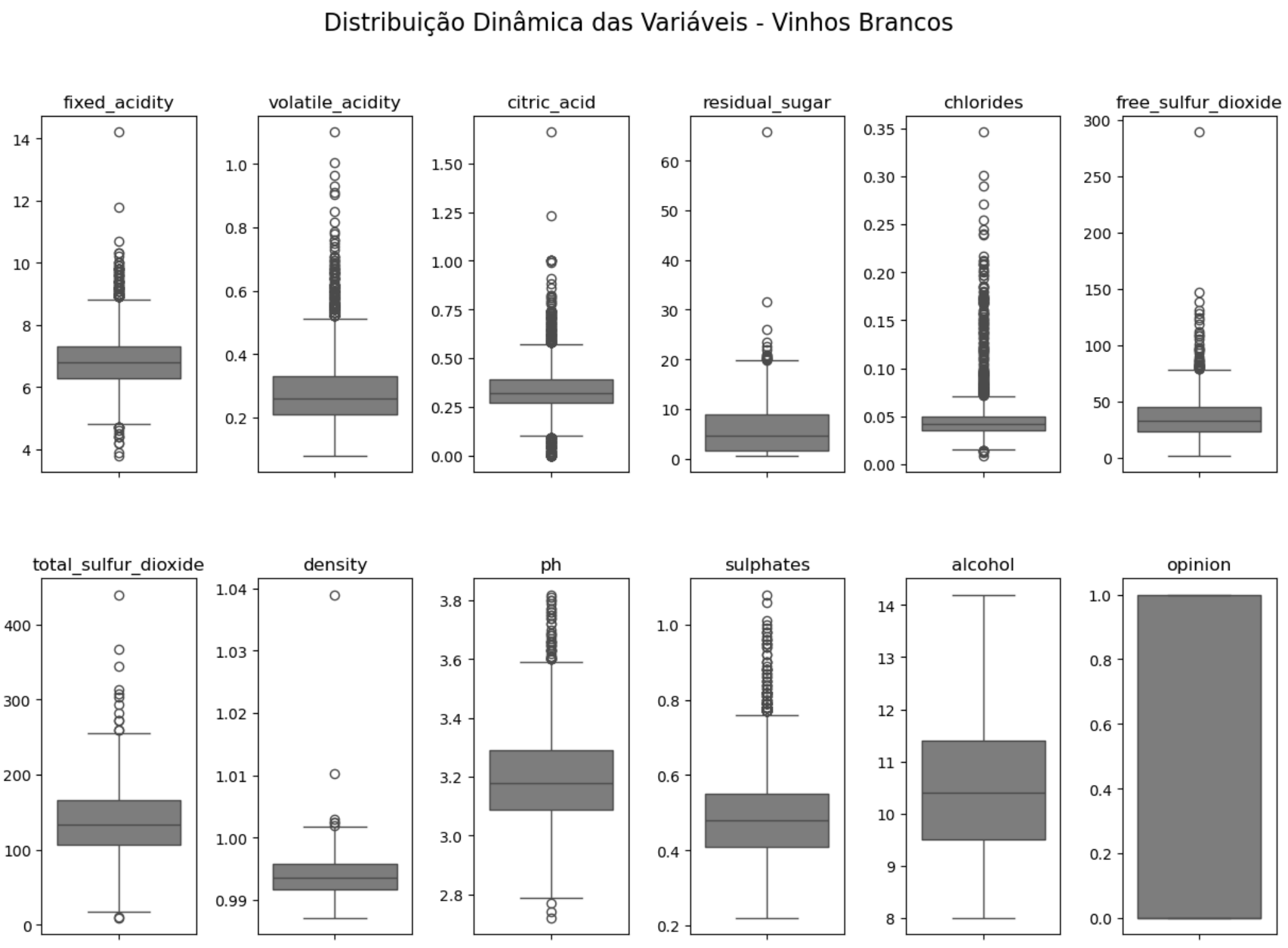
Distribuição dinâmica das variáveis

In [181…

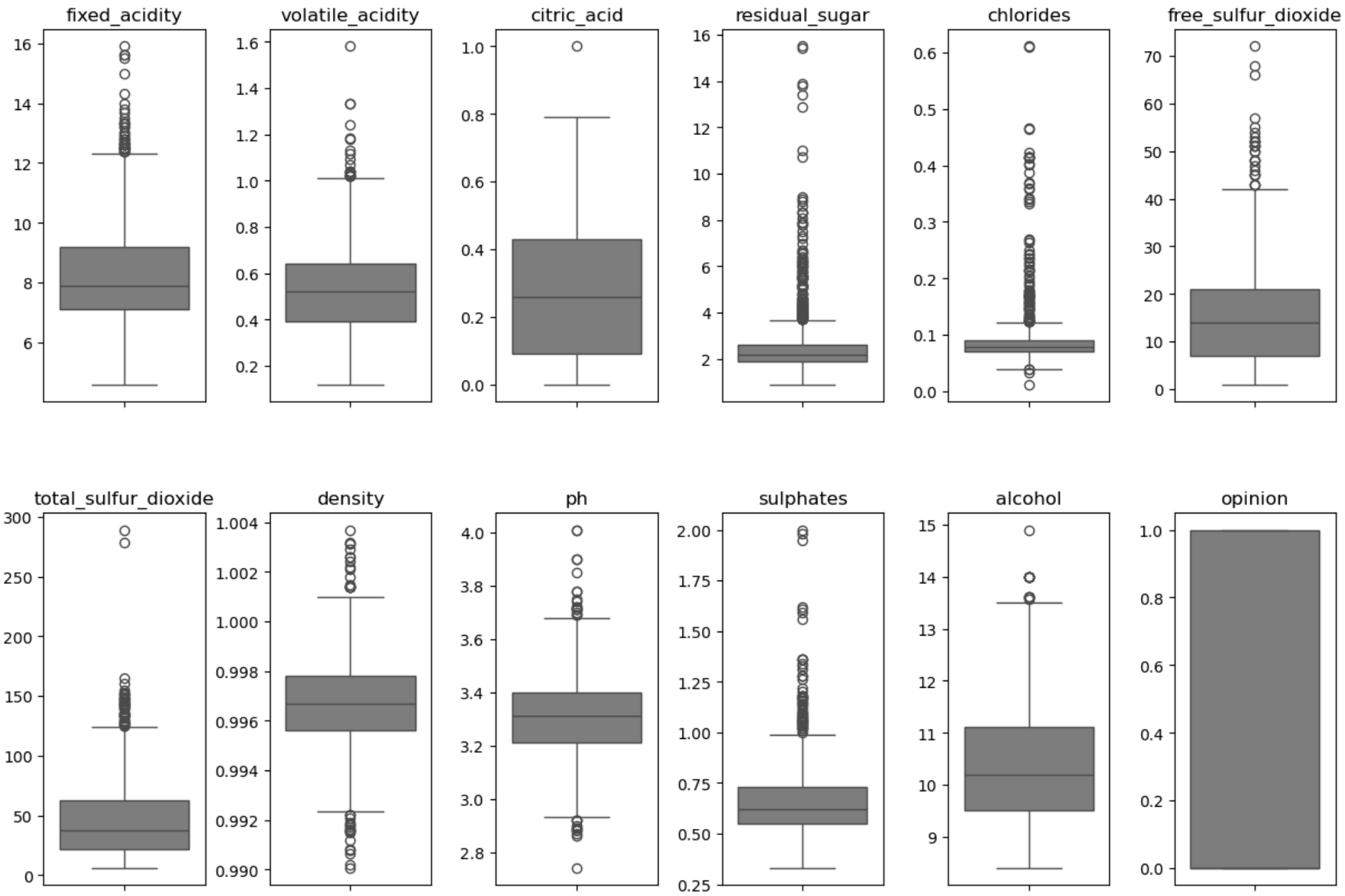
```
# função para exibir os gráficos separadamente para cada tipo de vinho
def plot_boxplots(data, title):
    numeric_columns = data.select_dtypes(include=['float64', 'int64']).columns.tolist()

    # Configurar o layout do gráfico (2 linhas x 6 colunas)
    fig, axes = plt.subplots(nrows=2, ncols=6, figsize=(15, 10))
    fig.suptitle(f"Distribuição Dinâmica das Variáveis - {title}", fontsize=16)

    axes = axes.flatten()

    # Criar os boxplots para cada variável numérica
    for i, column in enumerate(numeric_columns):
        sns.boxplot(data=data, y=column, ax=axes[i], color='gray')
        axes[i].set_title(column, fontsize=12)
        axes[i].set_xlabel('')
        axes[i].set_ylabel('')
```

```
    # Ajustar espaçamento para evitar sobreposição
    plt.subplots_adjust(hspace=0.3, wspace=0.4)

    # Exibir os gráficos
    plt.show()
```

In [182…
```
# Gerar os gráficos para os vinhos brancos
plot_boxplots(white, "Vinhos Brancos")
```



In [183…
```
# Gerar os gráficos para os vinhos tintos
plot_boxplots(red, "Vinhos Tintos")
```



Como se percebe, existem muitos outliers nas colunas de ambos os vinhos, que deverão ser levados em consideração no momento de tratamentos das variáveis

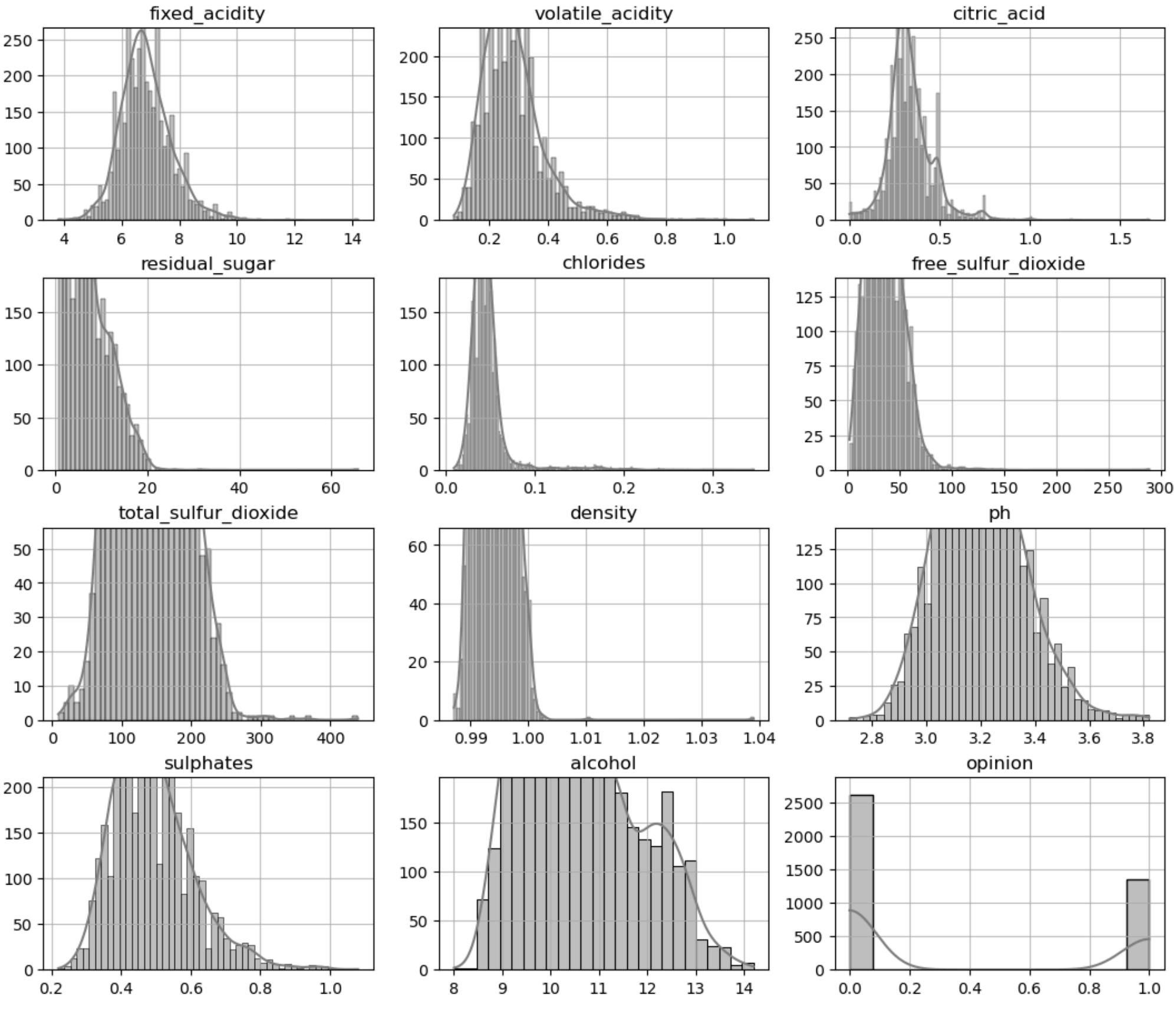
Visão por histogramas

In [184…

```python
# Função para exibir histogramas separadamente para cada tipo de vinho
def plot_histograms(data, title):
    numeric_columns = data.select_dtypes(include=['float64', 'int64']).columns.tolist()

    # Configurar o plano de fundo dos gráficos (4 linhas x 3 colunas)
    fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(13, 11))
    fig.suptitle(f"Análise da Faixa Dinâmica das Variáveis - {title}", fontsize=16)

    axes = axes.flatten()

    # Plotar os histogramas para cada variável numérica
    for i, col in enumerate(numeric_columns):
        sns.histplot(data=data, x=col, color='gray', kde=True, ax=axes[i])

        # Ajustar os limites do eixo Y com base na maior frequência da coluna
        max_y = data[col].value_counts().max()
        axes[i].set_ylim(0, max_y + max_y * 0.1)  # Adiciona 10% para espaço visual

        # Configurar título e labels
        axes[i].set_title(col, fontsize=12)
        axes[i].set_xlabel('')
        axes[i].set_ylabel('')
        axes[i].grid(True, alpha=0.8)

    # Ajustar espaçamento para evitar sobreposição
    plt.subplots_adjust(hspace=0.3, wspace=0.2)

    # Exibir os gráficos
    plt.show()
```

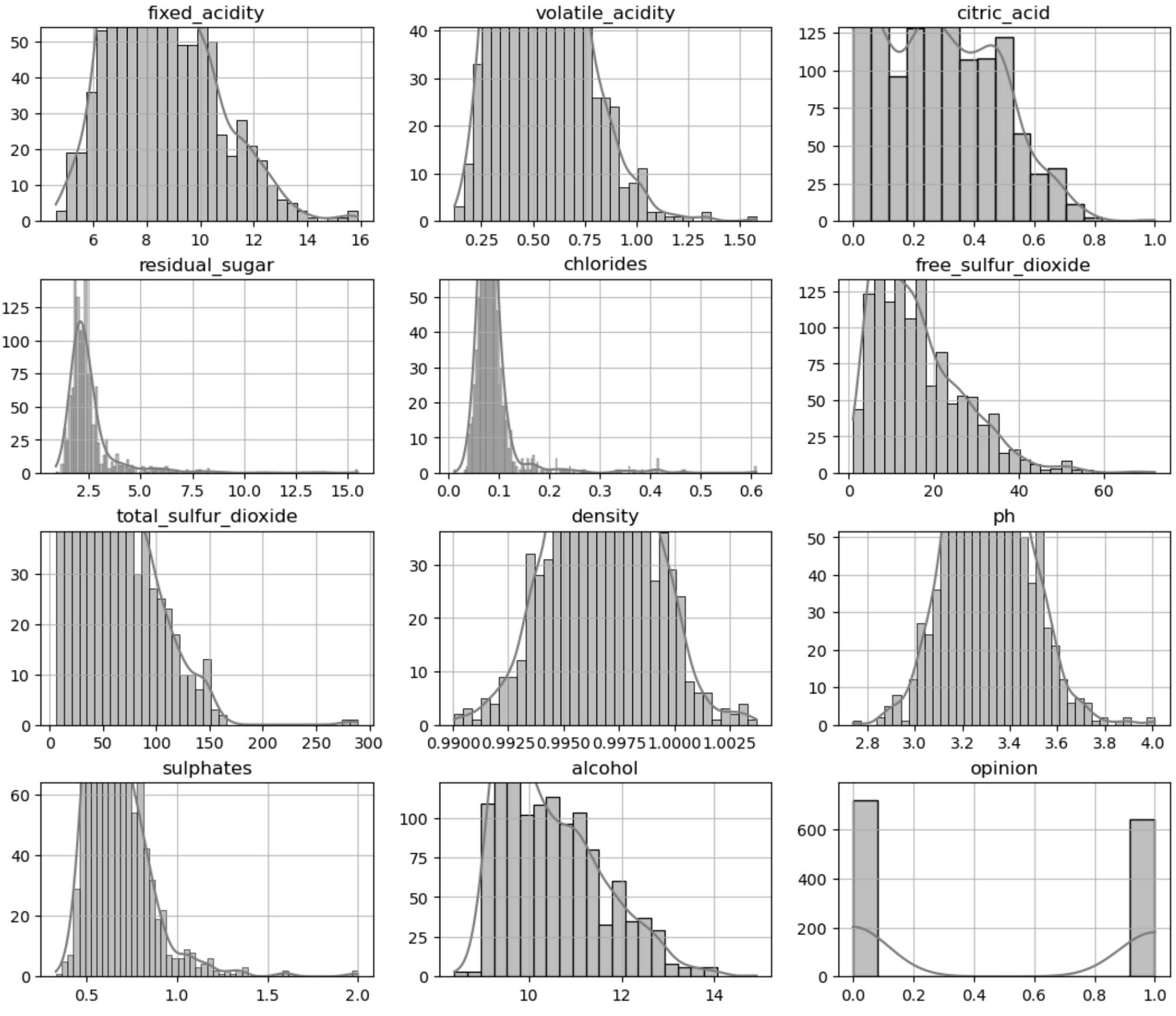
In [185…

```python
# Gerar os histogramas para vinhos brancos
plot_histograms(white, "Vinhos Brancos")
```



In [186…

```
# Gerar os histogramas para vinhos tintos
plot_histograms(red, "Vinhos Tintos")
```



Tratando os dados nulos

Para tratar os dados nulos, optou-se por: (i) selecionar as colunas numéricas, (ii) filtrar as colunas que possuem dados nulos, (iii) aplicar a média, caso a diferença entre média e mediana for pequena (<= tolerância [no caso, 10%] * mediana) (iv) aplicar a mediana caso a diferença seja maior.

In [187…

```python
def fill_missing_values(df, tol=0.1):
    # selecionar as colunas numéricas
    for col in df.select_dtypes(include=["float64", "int64"]).columns:
        mean_val = df[col].mean()
        median_val = df[col].median()

        # filtrar as colunas que possuem dados nulos
        if df[col].isnull().sum() > 0:

            # aplicar a média se 'tol' <= 10%
            if abs(mean_val - median_val) <= (tol * median_val):
                df[col].fillna(mean_val, inplace=True)

            # aplicar a mediana se 'tol' > 10%
            else:
                df[col].fillna(median_val, inplace=True)

    # retornar a função de checkagem das colunas
    return check(df)
```

In [188…

```python
fill_missing_values(white)
```

Out[188…

	coluna	tipo	únicos	null_soma	media	desvio	minimo	25%	mediana	75%	maximo	moda	frequência_moda
0	type	object	1	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	white	3970
1	fixed_acidity	float64	69	0	6.840876	0.865527	3.8	6.3	6.8	7.3	14.2	6.8	242
2	volatile_acidity	float64	126	0	0.280641	0.103486	0.08	0.21	0.26	0.32875	1.1	0.28	213
3	citric_acid	float64	88	0	0.334551	0.122449	0.0	0.27	0.32	0.39	1.66	0.3	239
4	residual_sugar	float64	310	0	5.919874	4.863493	0.6	1.6	4.7	8.9	65.8	1.2	166
5	chlorides	float64	161	0	0.045895	0.023079	0.009	0.036	0.042	0.05	0.346	0.036	166
6	free_sulfur_dioxide	float64	132	0	34.909698	17.218706	2.0	23.0	33.0	45.0	289.0	29.0	126
7	total_sulfur_dioxide	float64	251	0	137.248992	43.133975	9.0	106.0	133.0	166.0	440.0	111.0	51
8	density	float64	890	0	0.993792	0.002905	0.98711	0.99162	0.9935	0.99571	1.03898	0.992	60
9	ph	float64	104	0	3.195309	0.151345	2.72	3.09	3.18	3.29	3.82	3.16	128
10	sulphates	float64	80	0	0.490398	0.113566	0.22	0.41	0.48	0.55	1.08	0.5	191
11	alcohol	float64	103	0	10.588324	1.217302	8.0	9.5	10.4	11.4	14.2	9.5	178
12	opinion	int64	2	0	0.340302	0.47387	0.0	0.0	0.0	1.0	1.0	0	2619

In [189…

```
fill_missing_values(red)
```

Out[189…

	coluna	tipo	únicos	null_soma	media	desvio	minimo	25%	mediana	75%	maximo	moda	frequência_moda
0	type	object	1	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	red	1359
1	fixed_acidity	float64	97	0	8.313486	1.735331	4.6	7.1	7.9	9.2	15.9	7.2	49
2	volatile_acidity	float64	144	0	0.529381	0.182997	0.12	0.39	0.52	0.64	1.58	0.5	37
3	citric_acid	float64	81	0	0.272533	0.195397	0.0	0.095	0.26	0.43	1.0	0.0	117
4	residual_sugar	float64	91	0	2.5234	1.352314	0.9	1.9	2.2	2.6	15.5	2.0	133
5	chlorides	float64	153	0	0.088124	0.049377	0.012	0.07	0.079	0.091	0.611	0.08	50
6	free_sulfur_dioxide	float64	60	0	15.893304	10.44727	1.0	7.0	14.0	21.0	72.0	6.0	121
7	total_sulfur_dioxide	float64	144	0	46.825975	33.408946	6.0	22.0	38.0	63.0	289.0	28.0	35
8	density	float64	436	0	0.996709	0.001869	0.99007	0.9956	0.9967	0.99782	1.00369	0.9968	33
9	ph	float64	90	0	3.309492	0.154807	2.74	3.21	3.31	3.4	4.01	3.3	47
10	sulphates	float64	97	0	0.658622	0.170644	0.33	0.55	0.62	0.73	2.0	0.54	58
11	alcohol	float64	65	0	10.432315	1.082065	8.4	9.5	10.2	11.1	14.9	9.5	111
12	opinion	int64	2	0	0.470935	0.499338	0.0	0.0	0.0	1.0	1.0	0	719

Balanceamento dos dados

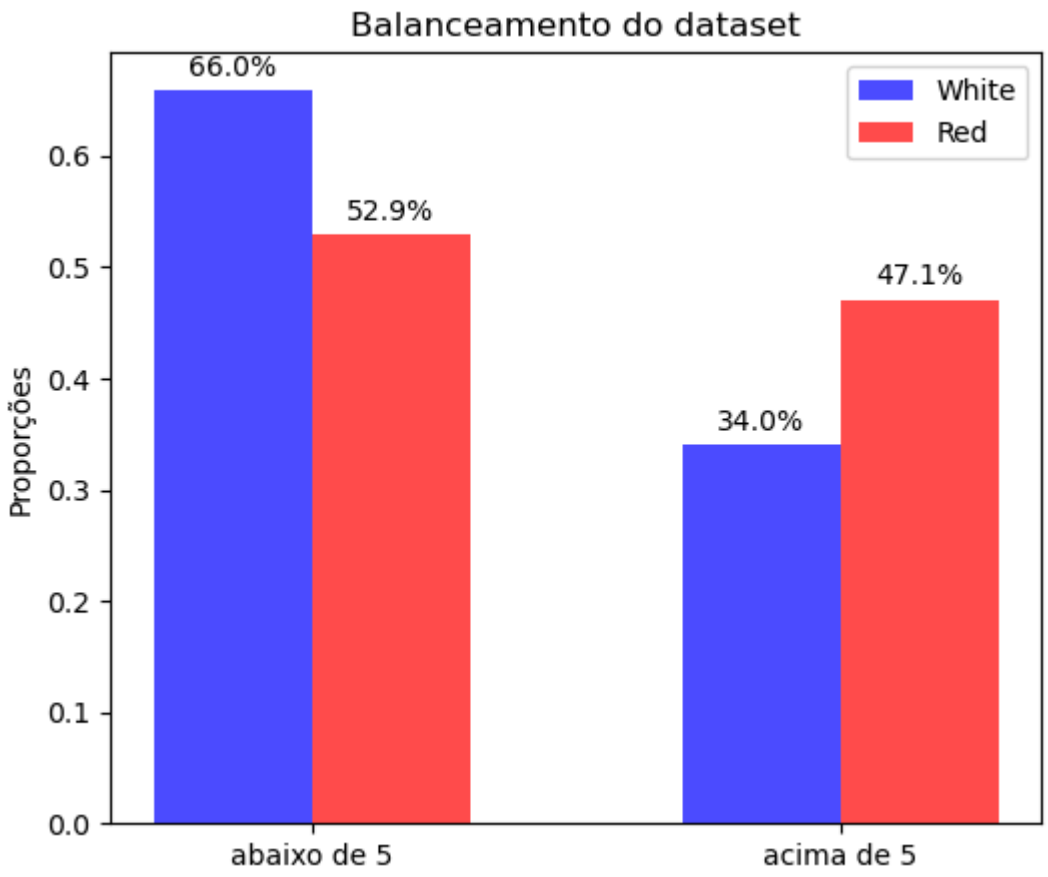
A verificação do balanceamento das varíaveis revela que o dataset possui mais vinhos de qualidade inferiores a '5' (em uma escala de 0 a 10) do que vinhos com qualidade maior que '5', além disso, na qualidade superior a '5', os vinhos tintos possuem vantagem.

Em primeira análise, verifica-se que a base dos vinhos brancos está mais desbalanceada do que os vinhos tintos.

In [598…

```
# verificar o balanceamento dos dados
proportions_white = white['opinion'].value_counts(normalize=True)
proportions_red = red['opinion'].value_counts(normalize=True)

# criar rótulos para o eixo X
labels = ['abaixo de 5', 'acima de 5']
x = np.arange(len(labels))

# definir a largura das barras
width = 0.30

# plotar o gráfico com os dados de ambos os vinhos
fig, ax = plt.subplots(figsize=(6,5))
bars_white = ax.bar(x - width/2, proportions_white.sort_index(), width, label='White', color='blue', alpha=0.7)
bars_red = ax.bar(x + width/2, proportions_red.sort_index(), width, label='Red', color='red', alpha=0.7)

# Adicionar porcentagens nas barras
for bars in [bars_white, bars_red]:
    for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height:.1%}',  # Converte para porcentagem
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 3),  # Deslocamento do texto
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=10, color='black')
```

```
# definir rótulos e título
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.set_ylabel('Proporções')
ax.set_title('Balanceamento do dataset')
ax.legend()

plt.show()
```



Parte 4. Modelos de classificação

Voltar ao início

4.a. Descreva as etapas necessárias para criar um modelo de classificação eficiente.

Resposta:

Par criar um modelo de classificação eficiente, a primeira coisa a ser feita é a análise exploratória e, caso seja necessário, o tratamento dos dados (etapa já realizada acima).

Passado essa fase preliminar, com os dados tratados, precisamos definir qual o problema que queremos resolver (ponto que vai influenciar o tipo de modelo usado e até mesmo em qual tipo de métrica de validação usaremos) e o que queremos prever (definir meu 'target predict' convencionalmente chamado de '*y*').

Com esse conhecimento em mãos, vamos selecionar quais colunas (ou 'features') são necessárias para prever meu '*y*' naquele modelo. Momento em que definimos nossa base de dados a ser trabalhada, convencionalmente chamado de '*X*'. Veja:

In [191…
```
white.columns
```

Out[191…
```
Index(['type', 'fixed_acidity', 'volatile_acidity', 'citric_acid',
       'residual_sugar', 'chlorides', 'free_sulfur_dioxide',
       'total_sulfur_dioxide', 'density', 'ph', 'sulphates', 'alcohol',
       'opinion'],
      dtype='object')
```

In [192…
```
# definir meu X para o dataset white
white_features = ['fixed_acidity', 'volatile_acidity', 'citric_acid', 'residual_sugar', 'chlorides',
                  'free_sulfur_dioxide', 'total_sulfur_dioxide', 'density', 'ph', 'sulphates', 'alcohol']

X_white = white[white_features]
```

In [193…
```
# definir meu 'target predict' ou y
y_white = white['opinion']
```

Com o '*X*' e o '*y*' bem definidos, vamos analisar a necessidade de separar o dataset em treino e teste, de modo a auxiliar no processo de validação (dependendo do problema, essa separação pode não ser necessária), invocando a biblioteca '*train_test_split*' e definindo seus parâmetros (como 'test_size', 'train_size', 'random_state', ...). Caso haja necessidade de separá-los, convencionalmente se chamarão '*train_X*', '*val_X*', '*train_y*', '*val_y*'.

Separação dos sets de treino x teste estratificado

In [194…
```
# separar treino x teste
X_train, X_test, y_train, y_test = train_test_split(
                                                    X_white, y_white,   # define o conjunto de dados e o target predict
                                                    test_size=0.2,      # define o tamanho do conjunto de teste
                                                    random_state=17,    # salva uma seed para reproduções futuras
```

```
                                                shuffle=True,       # determina o embaralhamento do dataset antes da separação
                                                stratify=y_white    # garante a proporção entre as classes
                                                )
```

Vale lembrar que, em datasets grandes (big data) ou em casos de uso de modelos de alta complexidade (deep learing) existe a abordagem da separação dos dados em três sets distintos, quais sejam: treino, teste e validação. Nesse cenário, o set de validação serviria para realizar um 'fine tuning' do modelo sem a necessidade de reitreiná-lo, em razão do seu alto custo.

Em datasets menores, pode ser feito a abordagem da validação cruzada onde os dados são divididos em 'dobras' (folds) e o modelo é testado em cada uma delas, garantindo uma avaliação mais robusta sem precisar de um conjunto de validação separado (o que será aplicado aqui).

Após feito a separação dos sets, caso haja uma disparidade grande na distribuição dos dados, deve-se realizar o correto escalonamento dos dados (seja através da padronização ou da normalização dos dados). No caso em tela, vamos realizar o escalonamento através da Padronização Robusta (ou Robust Scaler). Veja:

Padronização com Robust Scaler

Como se observa, os dados possuem uma alta dispridade de valores (indo de **0.009** na coluna '*chlorides*' até **440.0** na coluna '*total_sulfur_dioxide*'). Isso faz com que seja necessária a realização de uma padronização, de modo a trazer todos os dados para o mesmo patamar.

Para tanto, vamos usar o RobustScaler, em razão de seu desempenho robusto contra outliers. Isso porque, o RobustScaler transforma os dados subtraindo a mediana e dividindo pelo intervalo interquartil (IQR), ou seja, a diferença entre o terceiro quartil (Q3) e o primeiro quartil (Q1).

Dessa forma, a padronização não é influenciada por valores extremos, como aconteceria com métodos tradicionais como MinMaxScaler ou StandardScaler, que são mais sensíveis a outliers.

A normalização é realizada em toda as features que compõe o meu 'X'. Veja:

In [195…
```
# implementar o modelo de normalização
r_scaler = RobustScaler(
                        with_centering=True,            # subtrai a mediana dos dados, centralizando os dados
                        with_scaling=True,              # divide os dados pelo IQR
                        quantile_range=(25.0, 75.0),    # define o intervalo para o cálculo do IQR
                        copy=True                       # cria uma cópia de segurança antes da transformação
                        )
```

In [196…
```
# ajustar (fit) apenas os dados de treino
r_scaler.fit(X_train)
```

Out[196…
```
▾ RobustScaler
RobustScaler()
```

In [197…
```
# transformar os dados de treino e teste
X_train_scaled = r_scaler.transform(X_train)
X_test_scaled = r_scaler.transform(X_test)
```

Vale lembrar que o escalonamento deve ser feito após a separação dos sets para evitar o que chamamos de 'vazamento dos dados' (se o treinamento ocorrer em todo conjunto de dados, você está considerando os dados de treino para calcular a média, a mediana, o IQR, ... dos dados de treino, o que afetará os resultados).

Nesse momento, passamos para a fase de definir qual modelo de classificação melhor se enquadra no problema que estamos enfrentando, definindo seus parâmetros e hiperparâmetros necessários para a predição desejada (como 'n_jobs', 'random_state', 'max_iter', ...).

A partir daí, caso seja aplicada a validação cruzada, o 'ajuste' ('fit') e a 'predição' (predict) já estão inerentes ao StratifiedKFold, onde passamos os parâmetros (número de dobras) e ele treinará os modelos automaticamente.

Caso a validação cruzada não seja aplicada, vamos 'ajustar' (ou 'treinar') o modelo, utilizando o método '*fit*', onde passaremos a dupla: ('*train_X*', '*train_y*') (ou apenas ('*X*','*y*'), caso não haja separação).

- A estrutura seria algo como: "*modelo.fit(train_X, train_y)*" ou "*modelo.fit(X, y)*".

Com o modelo definido e ajustado, é hora de realizar as predições, com o método 'predict' utilizando '*val_X*', caso haja separação de treino x teste. A variável pode ser chamada de '*y_hat*' (de 'chápeu', fazendo alusão aos números preditos em notação científica que possuem o símbolo de ^).

- A estrutura seria algo como: "*y_hat = modelo.predictfit(val_X)*" ou "*y_hat = modelo.predict(X)*".

Com a predição em mãos, passamos para a fase de validação, de modo a verificar e comprovar a eficiência do modelo construído. Para tanto, existem algumas métricas que podemos usar, como a negocial ou a estatística.

As métricas negociais envolvem puramente o conhecimento do negócio que está sendo tratado (exemplo: o nível de glicose acima de 99 mg/dL define uma pessoa diabética), por essa ótica, o modelo será eficaz caso siga corretamente as regras de negócio atrelada ao dataset.

As métricas estatísticas, por seu turno, possuem um escopo mais amplo. Podendo se dividir em métricas de viés e variância e serão utilizadas a depender do problema que queremos prever (exemplo: 'se há indícios de fraude ou não', 'se o avião pode cair ou não', 'se o paciente pode ter câncer ou não', ...). Isso por que, um modelo que possui 99% de precisão (por exemplo), pode não ser nada efetivo se não responder o problema que queremos resolver ou não alcançar a generalização (capacidade de prever cenários futuros com eficiência). Isso pode se dar por vários motivos, como uma base desbalanceada (quando há muito mais de um grupo que de outro), o modelo não for o adequado ou não for passado os parâmetros corretos.

Essa métricas nos auxiliam, inclusive, a entender e melhorar os modelos criados, definindo valores otimizados de parâmetros para cada caso.

Em último lugar, em conjunto com a validação, temos a otimização do modelo ('tuning'), que, embora inicialmente experimental, permite implementar a melhor combinação de parâmetros, se levado em consideração os fatores acima.

Abaixo, vamos treinar alguns modelos com o dataset de vinhos brancos.

Criando uma função para exibir as médias das métricas exigidas em cada modelo

In [ ]:
```
# criar K-Fold estratificado (padrão para todos os modelos)
kf = StratifiedKFold(
                    n_splits=10,        # quantidade de 'dobras' (ou 'folds')
                    shuffle=True,       # se os dados serão embaralhados ou não
                    random_state=17     # salva uma semente para repetição
                    )

# armazenar as métricas de cada modelo
results = {}
```

4.b. Treine um modelo de regressão logística usando um modelo de validação cruzada estratificada com k-folds (k=10) para realizar a classificação. Calcule para a base de teste:

i. a média e desvio da acurácia dos modelos obtidos;

ii. a média e desvio da precisão dos modelos obtidos;

iii. a média e desvio da recall dos modelos obtidos;

iv. a média e desvio do f1-score dos modelos obtidos.

Resposta:

A Regressão Logística se diferencia da Regressão Linear, pois a Regressão Linear se destina a prever variáveis contínuas (preço, altura, ...), gerando uma relação linear (uma 'linha reta'), se plotado em um gráfico. Ao passo em que a Regressão Logística se dispõe a prever dados de forma categórica (sim/não, fraude/não fraude, câncer/não câncer) através de uma relação de probalidade (um número entre 0 e 1), o que gera uma relação não-linear (ou sigmoide).

Há que se ressaltar que a Validação Cruzada, é uma técnica importante para evitar o sobreajuste (overfiting), dividindo os dados em várias 'dobras' (folds) para treinar e testar o modelo múltiplas vezes. O StratifiedKFold garante que a distribuição das classes seja proporcional, o que é crucial em problemas de classificação com classes desbalanceadas, proporcionando uma avaliação mais confiável da performance do modelo.

Sobre o modelo:

LogisticRegression(

- penalty: padrão 'l2'; pode ser ['l1' | 'l2' | 'elasticnet']; define o tipo de penalização aplicada aos coeficientes
- dual: padrão 'False'; pode ser [bool]; só é True se (penalty='l2' & solver='liblinear'); útil para problemas com mais amostras do que features
- tol: padrão '0.0001'; pode ser [float]; define a tolerância para critério de parada do algoritmo de otimização
- C: padrão '1.0'; pode ser [float]; define a inversa da força da regularização (quanto menor, maior a regularização)
- fit_intercept: padrão 'True'; pode ser [bool]; indica se o modelo deve incluir o viés na função de decisão
- intercept_scaling: padrão '1.0'; pode ser [float]; apenas se (solver='liblienar' & fit_intercept=True); ajusta a escala do intercepto para melhorar a estabilidade numérica
- class_weight: padrão 'None'; pode ser [None | dict{} | 'balanced']; define os pesos atribuídos às classes; 'balanced' ajusta automaticamente os pesos inversamente proporcionais às frequências das classes
- random_state: padrão 'None'; pode ser [None | Int | RandomState]; define a semente para geração de números aleatórios, garantindo reprodutibilidade
- solver: padrão 'lbfgs'; pode ser ['lbfgs' | 'newton-cg' | 'liblinear' | 'sag' | 'saga']; define o algoritmo usado para otimização
- max_iter: padrão '1000'; pode ser [int]; define o número máximo de iteracoes até a convergência
- multi_class: padrão 'auto'; pode ser ['auto', 'ovr', 'multinomial']; especifica como lidar com problemas multiclasse ('auto' escolhe automaticamente com base no solver)
- verbose: padrão '0'; pode ser [int]; controla o nível de verbosidade do treinamento (0 = silencioso, valores maiores ativam logs)
- warm_start: padrão 'False'; pode ser [bool]; se True, reutiliza a solução da iteração anterior para inicializar o próximo ajuste
- n_jobs: padrão 'None'; pode ser [None | int]; define o número de CPUs a serem usadas no processamento paralelo (somente alguns solvers suportam)
- l1_ratio: padrão 'None'; pode ser [None | float]; somente aplicável quando penalty='elasticnet', define a mistura entre L1 e L2 na regularização (0 = L2, 1 = L1)

)

4.b.1. Implementando o grid search

In [581…
```
# implementar os hiperparâmetros para o Grid Search
rl_param_grid = {
                 'penalty': ['l2', 'elasticnet', None],
                 'solver': ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'],
                 'multi_class': ['ovr', 'multinomial'],
                 'C': [0.001, 0.01, 0.1, 1, 10, 100],
                 'l1_ratio': [None, 0, 1]
                 }
```

4.b.2. Implementando o modelo

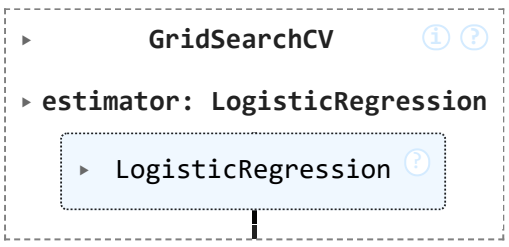
In [583…
```
# implementar o modelo de Regressão Logística (Logistic Regression Model) para o Grid Search
rl_model = LogisticRegression(
                              dual=False,
                              tol=0.0001,
                              fit_intercept=True,
                              intercept_scaling=1.0,
                              class_weight='balanced',
                              random_state=17,
                              max_iter=1000,
                              verbose=0,
                              warm_start=False,
                              n_jobs=None
                              )
```

4.b.3. Implementando a validação cruzada

In [584…
```
# configurar o Grid Search com validação cruzada estratificada
rl_grid_search = GridSearchCV(
                          rl_model,
                          rl_param_grid,
                          cv=kf,
                          scoring='f1_macro',
                          n_jobs=-1,
                          refit=True
                          )
```

In [585…
```
# treinar o modelo, buscando os melhores hiperparâmetros
rl_grid_search.fit(X_train, y_train)
```

Out[585…



4.b.4. Avaliando os melhores resultados para fine tuning

In [495…
```
# analisar os 10 melhores modelos e seus scores
rl_mean_scores = np.nan_to_num(rl_grid_search.cv_results_['mean_test_score'])
rl_params = rl_grid_search.cv_results_['params']

# ordenar os 10 melhores scores
rl_top_10 = rl_mean_scores.argsort()[::-1][:10]

# transformar em dataframe
rl_top_10_df = [
                {**rl_params[idx], "score": rl_mean_scores[idx]} for idx in rl_top_10
            ]

# criar o DataFrame
rl_top_10_df = pd.DataFrame(rl_top_10_df)

print("Top 10 melhores combinações de hiperparâmetros:")
rl_top_10_df
```

```
Top 10 melhores combinações de hiperparâmetros:
```

Out[495…

	C	class_weight	l1_ratio	multi_class	penalty	solver	score
0	100.000	balanced	1.0	multinomial	l2	lbfgs	0.711314
1	100.000	balanced	0.0	multinomial	l2	lbfgs	0.711314
2	100.000	balanced	NaN	multinomial	l2	lbfgs	0.711314
3	0.010	balanced	0.0	ovr	None	lbfgs	0.711223
4	0.100	balanced	0.0	ovr	None	lbfgs	0.711223
5	1.000	balanced	1.0	ovr	None	lbfgs	0.711223
6	0.001	balanced	1.0	ovr	None	lbfgs	0.711223
7	0.001	balanced	0.0	ovr	None	lbfgs	0.711223
8	10.000	balanced	1.0	ovr	None	lbfgs	0.711223
9	0.100	balanced	NaN	ovr	None	lbfgs	0.711223

In [595…

```
# encontrando a melhor combinação de parâmetros
rl_best_params = pd.DataFrame([{**rl_grid_search.best_params_, 'score': rl_grid_search.best_score_}])

rl_best_params
```

Out[595…

	C	l1_ratio	multi_class	penalty	solver	score
0	100	None	multinomial	l2	lbfgs	0.711314

In [496…

```
# extrair o melhor modelo encontrado pelo Grid Search
rl_best_model = rl_grid_search.best_estimator_
```

4.b.5. Imprimindo a média e o desvio padrão dos resultados

In [497…

```
# função para calcular métricas dos modelos obtidos na validação cruzada
def compute_metrics(modelo, X, y, kf):
    metrics = {
        'accuracy': cross_val_score(modelo, X, y, cv=kf, scoring='accuracy'),
        'precision': cross_val_score(modelo, X, y, cv=kf, scoring='precision'),
        'recall': cross_val_score(modelo, X, y, cv=kf, scoring='recall'),
        'f1': cross_val_score(modelo, X, y, cv=kf, scoring='f1')
    }
    return metrics
```

In [498…

```
# calcular as métricas para os melhores modelos
rl_best_metrics_train = compute_metrics(rl_best_model, X_train, y_train, kf)
rl_best_metrics_test = compute_metrics(rl_best_model, X_test, y_test, kf)
```

In [499…

```
# salvar os resultados para treino
metrics_list = ['accuracy', 'precision', 'recall', 'f1']

rl_metrics_train_df = pd.DataFrame({
    'Mínimo': [rl_best_metrics_train[m].min() for m in metrics_list],
    'Média': [rl_best_metrics_train[m].mean() for m in metrics_list],
    'Máximo': [rl_best_metrics_train[m].max() for m in metrics_list],
    'Desvio Padrão': [rl_best_metrics_train[m].std() for m in metrics_list]
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [500…

```
# salvar os resultados para teste
rl_metrics_test_df = pd.DataFrame({
    'Mínimo': [rl_best_metrics_test[m].min() for m in metrics_list],
    'Média': [rl_best_metrics_test[m].mean() for m in metrics_list],
    'Máximo': [rl_best_metrics_test[m].max() for m in metrics_list],
    'Desvio Padrão': [rl_best_metrics_test[m].std() for m in metrics_list]
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [501…

```
# imprimindo os resultados de treino
print('Métricas para Regressão Logística - Treino')
rl_metrics_train_df
```

```
Métricas para Regressão Logística - Treino
```

Out[501…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.687697	0.726067	0.769716	0.023785
Precisão	0.532374	0.576358	0.620690	0.026821
Recall	0.666667	0.736366	0.833333	0.048445
F1-Score	0.599190	0.646281	0.711462	0.033066

In [502…

```
# imprimindo os resultados de teste
print('Métricas para Regressão Logística - Teste')
rl_metrics_test_df
```

Métricas para Regressão Logística - Teste

Out[502…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.645570	0.715443	0.835443	0.052856
Precisão	0.486486	0.567642	0.750000	0.072704
Recall	0.555556	0.733333	0.814815	0.071817
F1-Score	0.526316	0.637666	0.763636	0.061974

4.c. Treine um modelo de árvores de decisão usando um modelo de validação cruzada estratificada com k-folds (k=10) para realizar a classificação. Calcule para a base de teste:

i. a média e desvio da acurácia dos modelos obtidos;

ii. a média e desvio da precisão dos modelos obtidos;

iii. a média e desvio da recall dos modelos obtidos;

iv. a média e desvio do f1-score dos modelos obtidos.

Resposta:

A Árvore de Decisão é um algoritmo de aprendizado supervisionado usado para classificação. Ela divide os dados em subconjuntos com base em perguntas hierárquicas, até chegar a um resultado final. A principal vantagem é sua interpretabilidade, mas pode sofrer de overfitting, especialmente em dados complexos.

Sobre o modelo:

DecisionTreeClassifier(

- 'criterion': padrão 'gini'; pode ser ['gini' | 'entropy' | 'log_loss']; define a função para medir a impureza dos nós
- 'splitter': padrão 'best'; pode ser ['best' | 'random']; escolhe a estratégia de divisão dos nós
- 'max_depth': padrão 'None'; pode ser [None | int]; profundidade máxima da árvore, None significa sem limite
- 'min_samples_split': padrão '2'; pode ser [int | float]; mínimo de amostras necessárias para dividir um nó
- 'min_samples_leaf': padrão '1'; pode ser [int | float]; mínimo de amostras necessárias em cada folha
- 'min_weight_fraction_leaf': padrão '0.0'; pode ser [float]; mínima fração de peso das amostras necessária em um nó folha
- 'max_features': padrão 'None'; pode ser [None | int | float | 'auto' | 'sqrt' | 'log2']; número máximo de features consideradas em cada divisão
- 'random_state': padrão 'None'; pode ser [None | int]; define a semente para reprodutibilidade dos resultados
- 'max_leaf_nodes': padrão 'None'; pode ser [None | int]; número máximo de folhas na árvore, None significa ilimitado
- 'min_impurity_decrease': padrão '0.0'; pode ser [float]; garante que a redução da impureza mínima seja atingida antes da divisão
- 'class_weight': padrão 'None'; pode ser [None | dict | 'balanced']; ajusta pesos das classes para lidar com desbalanceamento
- 'ccp_alpha': padrão '0.0'; pode ser [float]; parâmetro para poda da árvore, valores maiores simplificam a árvore

)

4.c.1. Implementando o grid search

In [ ]:
```
# implementar os hiperparâmetros para o Grid Search
tree_param_grid = {
                    'criterion': ['gini', 'entropy', 'log_loss'],
                    'splitter': ['best', 'random'],
                    'max_depth': range(2, 11),
                    'min_samples_split': [2, 3, 4, 5, 10],
                    'min_samples_leaf': [1, 2, 5],
                    'max_features': [None, 'auto', 'sqrt', 'log2']
                   }
```

4.c.2. Implementando o modelo

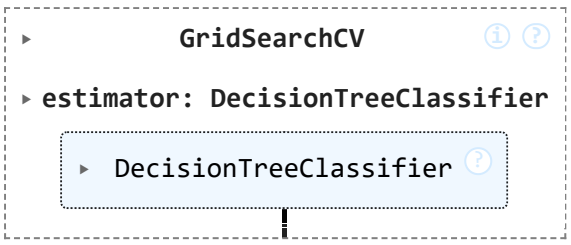
In [504…
```
# implementar o modelo de Árvore de Decisão (Decision Tree Model) para o Grid Search
tree_model = DecisionTreeClassifier(
                                    min_weight_fraction_leaf = 0.0,
                                    random_state = 17,
                                    max_leaf_nodes = None,
                                    min_impurity_decrease = 0.0,
                                    class_weight = None,
                                    ccp_alpha = 0.0
                                    )
```

4.c.3. Implementando a validação cruzada

In [505…
```
# configurar o Grid Search com validação cruzada estratificada
tree_grid_search = GridSearchCV(
                            tree_model,
                            tree_param_grid,
                            cv=kf,
                            scoring='f1_macro',
                            n_jobs=-1
                            )
```

In [506…
```
# treinar o modelo e buscando os melhores hiperparâmetros
tree_grid_search.fit(X_train, y_train)
```

Out[506…



4.c.4. Avaliando os melhores resultados para fine tuning

In [507…
```
# analisar os 10 melhores modelos e seus scores
tree_mean_scores = np.nan_to_num(tree_grid_search.cv_results_['mean_test_score'])
tree_params = tree_grid_search.cv_results_['params']

# ordenar os 10 melhores scores
tree_top_10 = tree_mean_scores.argsort()[::-1][:10]

# transformar em dataframe
tree_top_10_df = [
                {**tree_params[idx], "score": tree_mean_scores[idx]} for idx in tree_top_10
            ]

# criar o DataFrame
tree_top_10_df = pd.DataFrame(tree_top_10_df)

print("Top 10 melhores combinações de hiperparâmetros:")
tree_top_10_df
```

```
Top 10 melhores combinações de hiperparâmetros:
```

Out[507…

	criterion	max_depth	max_features	min_samples_leaf	min_samples_split	splitter	score
0	entropy	5	None	5	5	best	0.703571
1	log_loss	5	None	5	2	best	0.703571
2	log_loss	5	None	5	10	best	0.703571
3	entropy	5	None	5	10	best	0.703571
4	entropy	5	None	5	4	best	0.703571
5	entropy	5	None	5	3	best	0.703571
6	entropy	5	None	5	2	best	0.703571
7	log_loss	5	None	5	5	best	0.703571
8	log_loss	5	None	5	4	best	0.703571
9	log_loss	5	None	5	3	best	0.703571

In [596…
```
# encontrando a melhor combinação de parâmetros
tree_best_params = pd.DataFrame([{**tree_grid_search.best_params_, 'score': tree_grid_search.best_score_}])

tree_best_params
```

Out[596…

	criterion	max_depth	max_features	min_samples_leaf	min_samples_split	splitter	score
0	entropy	5	None	5	2	best	0.703571

In [508…
```
# extrair o melhor modelo encontrado pelo Grid Search
tree_best_model = tree_grid_search.best_estimator_
```

4.c.5. Imprimindo a média e o desvio padrão dos resultados

In [509…
```
# calcular as métricas para os melhores modelos
tree_best_metrics_train = compute_metrics(tree_best_model, X_train, y_train, kf)
tree_best_metrics_test = compute_metrics(tree_best_model, X_test, y_test, kf)
```

In [514…
```
# salvar os resultados para treino
metrics_list = ['accuracy', 'precision', 'recall', 'f1']
```

```
tree_metrics_train_df = pd.DataFrame({
    'Mínimo': [tree_best_metrics_train[m].min() for m in metrics_list],
    'Média': [tree_best_metrics_train[m].mean() for m in metrics_list],
    'Máximo': [tree_best_metrics_train[m].max() for m in metrics_list],
    'Desvio Padrão': [tree_best_metrics_train[m].std() for m in metrics_list]
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [515…

```
# salvar os resultados para teste
tree_metrics_test_df = pd.DataFrame({
    'Mínimo': [tree_best_metrics_test[m].min() for m in metrics_list],
    'Média': [tree_best_metrics_test[m].mean() for m in metrics_list],
    'Máximo': [tree_best_metrics_test[m].max() for m in metrics_list],
    'Desvio Padrão': [tree_best_metrics_test[m].std() for m in metrics_list],
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [516…

```
# imprimindo os resultados de treino
print('Métricas para Árvore de Decisão - Treino')
tree_metrics_train_df
```

```
Métricas para Árvore de Decisão - Treino
```

Out[516…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.712934	0.741501	0.779180	0.019872
Precisão	0.584746	0.637322	0.706522	0.040657
Recall	0.472222	0.566191	0.638889	0.048572
F1-Score	0.528497	0.597957	0.650000	0.032710

In [517…

```
# imprimindo os resultados de teste
print('Métricas para Árvore de Decisão - Teste')
tree_metrics_test_df
```

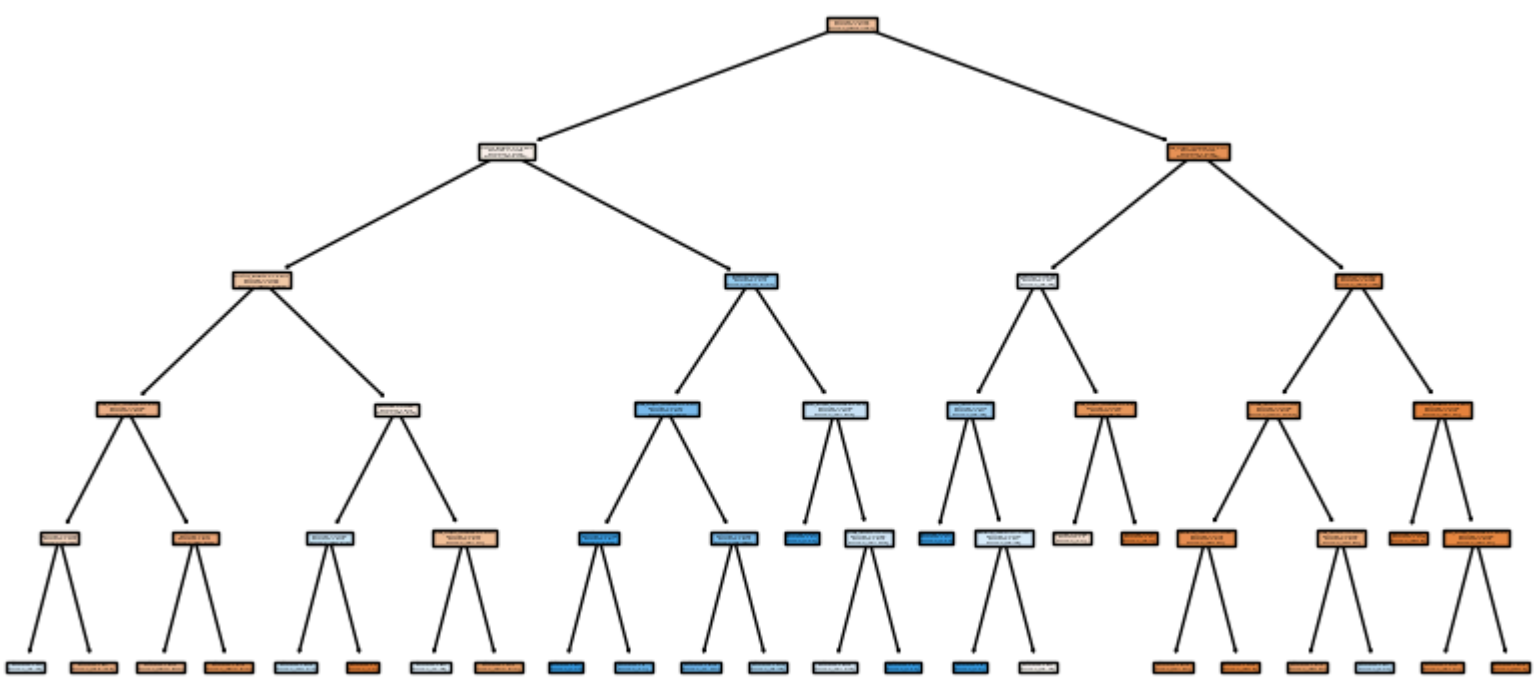
```
Métricas para Árvore de Decisão - Teste
```

Out[517…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.620253	0.686345	0.737500	0.034488
Precisão	0.411765	0.541317	0.615385	0.064056
Recall	0.185185	0.433333	0.703704	0.147266
F1-Score	0.263158	0.471782	0.644068	0.111923

In [523…

```
# visualizando a árvore de decisão
plt.figure(figsize=(10, 5))
plot_tree(tree_best_model, feature_names=X_train.columns, filled=True)
plt.show()
```



In [524…

```
# analisando as dimensoes da arvore
print(f'Profundidade: {tree_best_model.get_depth()}')
print(f'Número de folhas (nós): {tree_best_model.get_n_leaves()}')
```

```
Profundidade: 5
Número de folhas (nós): 27
```

4.d. Treine um modelo de SVM usando um modelo de validação cruzada estratificada com k-folds (k=10) para realizar a classificação. Calcule para a base de teste:

i. a média e desvio da acurácia dos modelos obtidos;

ii. a média e desvio da precisão dos modelos obtidos;

iii. a média e desvio da recall dos modelos obtidos;

iv. a média e desvio do f1-score dos modelos obtidos.

Resposta:

O Support Vector Machine (SVM) é um algoritmo de aprendizado supervisionado usado para classificação, que busca encontrar o hiperplano que separa as classes de dados com a maior margem possível. Ele é eficaz em espaços de alta dimensão e pode lidar com dados não linearmente separáveis usando o truque do 'kernel'. Apesar de ser potente, o SVM pode ser computacionalmente caro em grandes volumes de dados e exige a escolha adequada de parâmetros como o tipo de 'kernel' e o valor de 'C'.

Sobre o modelo:

SVC(

- 'C': padrão '1.0'; pode ser [float]; controla a penalização de erros, quanto maior, menos margem para erro
- 'kernel': padrão 'rbf'; pode ser ['bf' | 'linear' | 'poly' | 'sigmoid' | 'precomputed']; define o tipo de função usada para separar os dados
- 'degree': padrão '3'; pode ser [int]; grau do polinômio, apenas quando kernel='poly'
- 'gamma': padrão 'scale'; pode ser ['scale' | 'auto' | float]; define a influência de um único exemplo de treinamento
- 'coef0': padrão '0.0'; pode ser [float]; termo independente para os kernels=['poly' | 'sigmoid']
- 'shrinking': padrão 'True'; pode ser [bool]; ignora vetores de suporte irrelevantes durante o treinamento, acelerando a convergência do algoritmo
- 'probability': padrão 'False'; pode ser [bool]; ativa a saída de probabilidades, mas deixa o treinamento mais lento
- 'tol': padrão '0.0001'; pode ser [float]; critério de parada do algoritmo
- 'cache_size': padrão '200'; pode ser [float]; tamanho do cache em MB para cálculos de kernel
- 'class_weight': padrão 'None'; pode ser [None | 'balanced' | dict{}]; ajusta pesos das classes para lidar com desbalanceamento
- 'max_iter': padrão '-1'; pode ser [int]; número máximo de iterações, -1 significa sem limite
- 'decision_function_shape': padrão 'ovr'; pode ser ['ovr' | 'ovo']; define a estratégia multi-classe: one-vs-rest (ovr) ou one-vs-one (ovo)
- 'break_ties': padrão 'False'; pode ser [bool]; resolve empates na predição multi-classe quando decision_function_shape='ovr'
- 'random_state': padrão 'None'; pode ser [None | int]; define a semente para reprodutibilidade dos resultados

)

4.d.1. Implementando o grid search

In [550…

```python
# implementar os hiperparâmetros para o Grid Search
svm_param_grid = {
                'C': list(np.arange(1.0, 10.2, 0.1)),
                'degree': [2, 3, 4],  # apenas para o kernel='poly'
                'gamma': ['scale', 'auto'],
                'shrinking': [True, False]
                }
```

4.d.2. Implementando o modelo

In [551…

```python
# configurar o modelo de Máquinas Suportadas por Vetores (Support Vector Machine - SVM)
svm_model = SVC(
                kernel='rbf',
                coef0=0,
                probability=True,
                tol=0.001,
                cache_size=200,
                class_weight=None,
                verbose=False,
                max_iter=1000,
                decision_function_shape='ovr',
                break_ties=False,
                random_state=None
                )
```

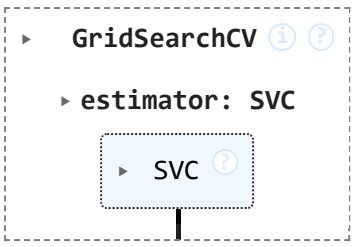
4.d.3. Implementando a validação cruzada

In [527…

```python
# configurar o Grid Search com validação cruzada estratificada
svm_grid_search = GridSearchCV(
                               svm_model,
                               svm_param_grid,
                               cv=kf,
                               scoring='f1_macro',
                               n_jobs=-1
                               )
```

In [528…]
```
# treinar o modelo e buscando os melhores hiperparâmetros
svm_grid_search.fit(X_train, y_train)
```

Out[528…]



4.d.4. Avaliando os melhores resultados para fine tuning

In [529…]
```
# analisar os 10 melhores modelos e seus scores
svm_mean_scores = np.nan_to_num(svm_grid_search.cv_results_['mean_test_score'])
svm_params = svm_grid_search.cv_results_['params']

# ordenar os 10 melhores scores
svm_top_10 = svm_mean_scores.argsort()[::-1][:10]

# transformar em dataframe
svm_top_10_df = [
                {**svm_params[idx], "score": svm_mean_scores[idx]} for idx in svm_top_10
            ]

# criar o DataFrame
svm_top_10_df = pd.DataFrame(svm_top_10_df)

print("Top 10 melhores combinações de hiperparâmetros:")
svm_top_10_df

# 0.618708
```

```
Top 10 melhores combinações de hiperparâmetros:
```

Out[529…]

	C	degree	gamma	kernel	shrinking	score
0	10.1	4	scale	rbf	False	0.622469
1	10.1	2	scale	rbf	False	0.622469
2	10.1	3	scale	rbf	False	0.622469
3	10.1	2	scale	rbf	True	0.622469
4	10.1	4	scale	rbf	True	0.622469
5	10.1	3	scale	rbf	True	0.622469
6	3.1	3	auto	rbf	False	0.611798
7	3.1	2	auto	rbf	True	0.611798
8	3.1	3	auto	rbf	True	0.611798
9	3.1	2	auto	rbf	False	0.611798

In [597…]
```
# encontrando a melhor combinação de parâmetros
svm_best_params = pd.DataFrame([{**svm_grid_search.best_params_, 'score': svm_grid_search.best_score_}])

svm_best_params
```

Out[597…]

	C	degree	gamma	kernel	shrinking	score
0	10.1	2	scale	rbf	True	0.622469

In [534…]
```
# extrair o melhor modelo encontrado pelo Grid Search
svm_best_model = svm_grid_search.best_estimator_
```

4.d.5. Imprimindo a média e o desvio padrão dos resultados

In [535…]
```
# calcular as métricas para os melhores modelos
svm_best_metrics_train = compute_metrics(svm_best_model, X_train, y_train, kf)
svm_best_metrics_test = compute_metrics(svm_best_model, X_test, y_test, kf)
```

In [539…]
```
# salvar os resultados para treino
metrics_list = ['accuracy', 'precision', 'recall', 'f1']

svm_metrics_train_df = pd.DataFrame({
    'Mínimo': [svm_best_metrics_train[m].min() for m in metrics_list],
    'Média': [svm_best_metrics_train[m].mean() for m in metrics_list],
    'Máximo': [svm_best_metrics_train[m].max() for m in metrics_list],
    'Desvio Padrão': [svm_best_metrics_train[m].std() for m in metrics_list]
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [540…]
```
# salvar os resultados para teste
svm_metrics_test_df = pd.DataFrame({
    'Mínimo': [svm_best_metrics_test[m].min() for m in metrics_list],
```

```
    'Média': [svm_best_metrics_test[m].mean() for m in metrics_list],
    'Máximo': [svm_best_metrics_test[m].max() for m in metrics_list],
    'Desvio Padrão': [svm_best_metrics_test[m].std() for m in metrics_list]
}, index=['Acurácia', 'Precisão', 'Recall', 'F1-Score'])
```

In [541…
```
# imprimindo os resultados de treino
print('Métricas para Máquinas Suportadas por Vetores - Treino')
svm_metrics_train_df
```

Métricas para Máquinas Suportadas por Vetores - Treino

Out[541…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.652997	0.691447	0.747634	0.025167
Precisão	0.490741	0.574104	0.637255	0.050131
Recall	0.194444	0.416191	0.620370	0.139595
F1-Score	0.297872	0.465145	0.619048	0.097175

In [542…
```
# imprimindo os resultados de teste
print('Métricas para Máquinas Suportadas por Vetores - Teste')
svm_metrics_test_df
```

Métricas para Máquinas Suportadas por Vetores - Teste

Out[542…

	Mínimo	Média	Máximo	Desvio Padrão
Acurácia	0.625000	0.670079	0.721519	0.030155
Precisão	0.285714	0.601944	1.000000	0.229482
Recall	0.074074	0.129630	0.185185	0.037952
F1-Score	0.117647	0.210858	0.312500	0.061615

Parte 5. Comparação com gráfico de curva ROC

Voltar ao início

5.1. Em relação à questão anterior, qual o modelo deveria ser escolhido para uma eventual operação. Responda essa questão mostrando a comparação de todos os modelos, usando um gráfico mostrando a curva ROC média para cada um dos gráficos e justifique.

Resposta:

A curva ROC avalia modelos de classificação binária, ela representa a relação entre a Taxa de Verdadeiros Positivos (TPR) e a Taxa de Falsos Positivos (FPR). O TPR mede a capacidade do modelo em identificar corretamente os positivos, enquanto o FPR mede a quantidade de negativos classificados erroneamente como positivos. Enquanto o TPR mede a capacidade do modelo de identificar corretamente os positivos, o FPR mostra quantos negativos foram classificados erroneamente como positivos. A Área sob a Curva (AUC) resume essa performance: quanto maior a AUC, melhor a capacidade do modelo de distinguir entre as classes.

No contexto da classificação de vinhos, onde o objetivo é separar vinhos 'bons' (qualidade acima de 5) de 'ruins' (qualidade abaixo de 5), é mais problemático classificar um vinho ruim como bom do que o contrário - momento em que o score da precisãos seria o fator fundamental a ser observado. Porém há que se considerar que a a base de dados é desbalanceada. Desse modo, o F1-score se torna um critério mais adequado para escolher o melhor modelo, pois considera tanto os verdadeiros positivos quanto os falsos positivos e negativos, equilibrando precisão e recall.

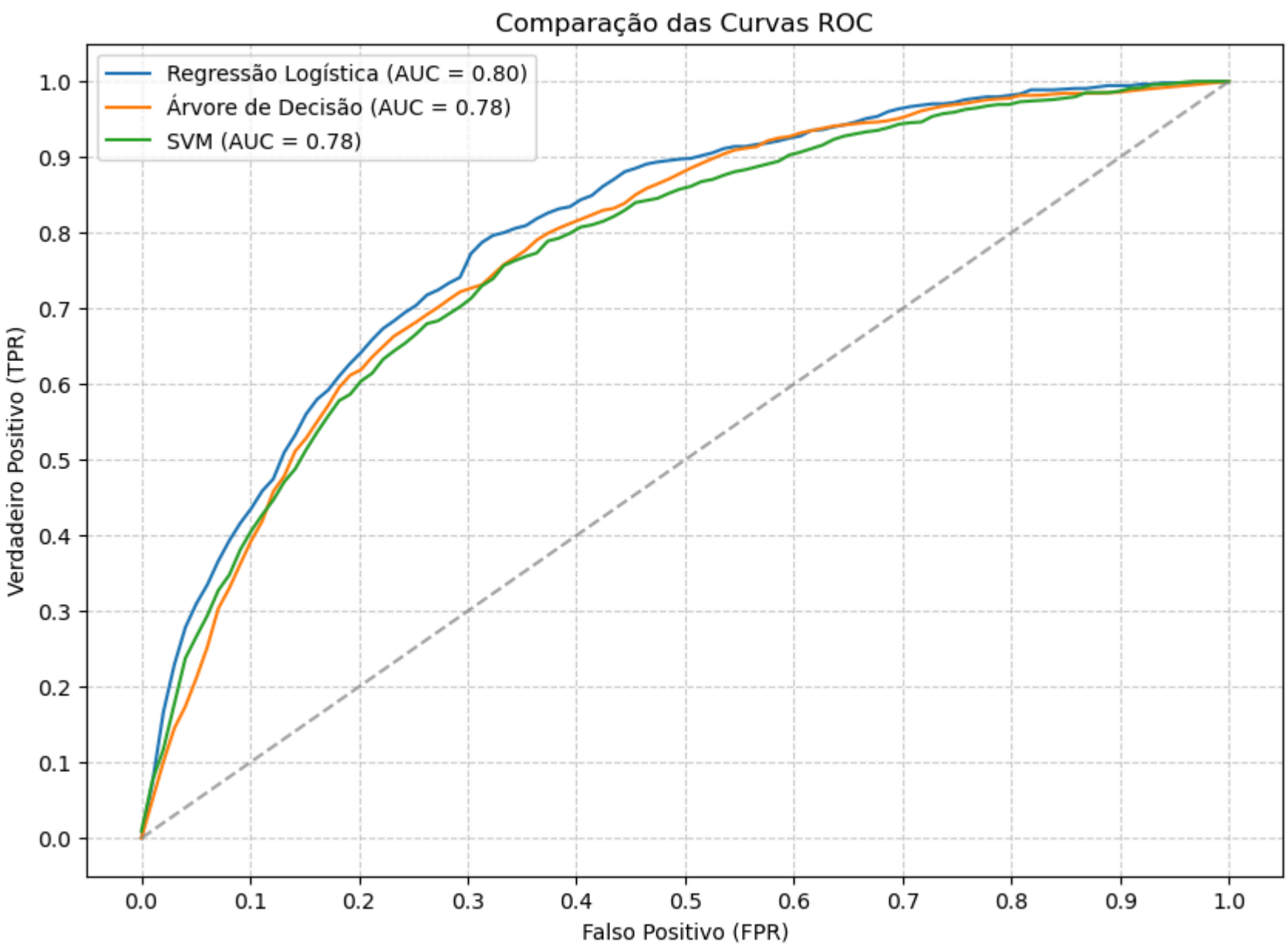
Dentre os modelos analisados, a Regressão Logística apresentou o melhor desempenho em termos de F1-score, sendo 63.77% no treino e 64.63% no teste. Além disso, sua AUC também se destacou como os melhores scores, sendo 80,30% no treino e 80,30% no teste. Esses resultados tornam a Regressão Logística a melhor escolha para uma eventual operação.

Para embasar essa decisão, um gráfico da curva ROC média para cada modelo permite visualizar as diferenças de desempenho e reforça a escolha da Regressão Logística como a opção mais adequada.

In [543…
```
# lista de modelos com os melhores parâmetros
best_models = {
    "Regressão Logística": rl_best_model,
    "Árvore de Decisão": tree_best_model,
    "SVM": svm_best_model
}
```

In [544…
```
# dicionário para armazenar os AUCs de cada modelo
auc_scores = {}

plt.figure(figsize=(10, 7))

# loop para calcular a curva ROC com os modelos treinados
for name, model in best_models.items():
    mean_fpr = np.linspace(0, 1, 100)
    tprs = []
    aucs = []
```

```
    # obter previsões de probabilidade usando validação cruzada
    y_scores = cross_val_predict(model, X_train_scaled, y_train, cv=kf, method='predict_proba')[:, 1]

    # calcular a curva ROC
    fpr, tpr, _ = roc_curve(y_train, y_scores)
    auc_value = auc(fpr, tpr)

    # armazenar o AUC no dicionário
    auc_scores[name] = auc_value

    # interpolanr as curvas ROC para suavização
    tprs.append(np.interp(mean_fpr, fpr, tpr))
    aucs.append(auc_value)

    # média das curvas ROC
    mean_tpr = np.mean(tprs, axis=0)
    mean_auc = np.mean(aucs)

    plt.plot(mean_fpr, mean_tpr, label=f"{name} (AUC = {mean_auc:.2f})")

# linha de referência
plt.grid(True, linestyle="--", alpha=0.6)
plt.xticks(np.arange(0, 1.1, 0.1))
plt.yticks(np.arange(0, 1.1, 0.1))
plt.plot([0, 1], [0, 1], linestyle="--", color="gray", alpha=0.7)
plt.xlabel("Falso Positivo (FPR)")
plt.ylabel("Verdadeiro Positivo (TPR)")
plt.title("Comparação das Curvas ROC")
plt.legend(loc='best')
plt.show()
```



In [545…

```
# criar um dicionário com todas as métricas de treino
test_metrics_dict = {
                'Modelo': ['Regressão Logística', 'Árvore de Decisão', 'SVM'],
                'Acurácia': [rl_metrics_train_df.loc['Acurácia', 'Média'],
                             tree_metrics_train_df.loc['Acurácia', 'Média'],
                             svm_metrics_train_df.loc['Acurácia', 'Média']],
                'Precisão': [rl_metrics_train_df.loc['Precisão', 'Média'],
                             tree_metrics_train_df.loc['Precisão', 'Média'],
                             svm_metrics_train_df.loc['Precisão', 'Média']],
                'Recall': [rl_metrics_train_df.loc['Recall', 'Média'],
                       tree_metrics_train_df.loc['Recall', 'Média'],
                       svm_metrics_train_df.loc['Recall', 'Média']],
                'F1-Score': [rl_metrics_train_df.loc['F1-Score', 'Média'],
                             tree_metrics_train_df.loc['F1-Score', 'Média'],
                             svm_metrics_train_df.loc['F1-Score', 'Média']],
                'AUC': [auc_scores['Regressão Logística'],
                       auc_scores['Árvore de Decisão'],
                       auc_scores['SVM']]
            }
```

In [546…
```
# criar um dicionário com todas as métricas de teste
train_metrics_dict = {
                'Modelo': ['Regressão Logística', 'Árvore de Decisão', 'SVM'],
                'Acurácia': [rl_metrics_test_df.loc['Acurácia', 'Média'],
                             tree_metrics_test_df.loc['Acurácia', 'Média'],
                             svm_metrics_test_df.loc['Acurácia', 'Média']],
                'Precisão': [rl_metrics_test_df.loc['Precisão', 'Média'],
                             tree_metrics_test_df.loc['Precisão', 'Média'],
                             svm_metrics_test_df.loc['Precisão', 'Média']],
                'Recall': [rl_metrics_test_df.loc['Recall', 'Média'],
                       tree_metrics_test_df.loc['Recall', 'Média'],
                       svm_metrics_test_df.loc['Recall', 'Média']],
                'F1-Score': [rl_metrics_test_df.loc['F1-Score', 'Média'],
                             tree_metrics_test_df.loc['F1-Score', 'Média'],
                             svm_metrics_test_df.loc['F1-Score', 'Média']],
                'AUC': [auc_scores['Regressão Logística'],
                       auc_scores['Árvore de Decisão'],
                       auc_scores['SVM']]
            }
```

In [547…
```
# criar os DataFrames
all_train_metrics_df = pd.DataFrame(train_metrics_dict)
all_test_metrics_df = pd.DataFrame(test_metrics_dict)
```

In [548…
```
# exibir o resultado
print("Métricas de treino dos modelos")
all_train_metrics_df
```

```
Métricas de treino dos modelos
```

Out[548…

	Modelo	Acurácia	Precisão	Recall	F1-Score	AUC
0	Regressão Logística	0.715443	0.567642	0.733333	0.637666	0.802965
1	Árvore de Decisão	0.686345	0.541317	0.433333	0.471782	0.782454
2	SVM	0.670079	0.601944	0.129630	0.210858	0.775103

In [549…
```
# exibir o resultado
print("Métricas de teste dos modelos")
all_test_metrics_df
```

```
Métricas de teste dos modelos
```

Out[549…

	Modelo	Acurácia	Precisão	Recall	F1-Score	AUC
0	Regressão Logística	0.726067	0.576358	0.736366	0.646281	0.802965
1	Árvore de Decisão	0.741501	0.637322	0.566191	0.597957	0.782454
2	SVM	0.691447	0.574104	0.416191	0.465145	0.775103

Parte 6. Inferências

Voltar ao início

6.1. Com a escolha do melhor modelo, use os dados de vinho tinto, presentes na base original e faça a inferência (não é para treinar novamente!!!) para saber quantos vinhos são bons ou ruins. Utilize o mesmo critério utilizado com os vinhos brancos, para comparar o desempenho do modelo. Ele funciona da mesma forma para essa nova base? Justifique.

Resposta:

O modelo escolhido para a classificação dos vinhos tintos foi o da Regressão Logística. Ao aplicá-lo, sem realizar um novo treinamento, o modelo classificou 640 vinhos como bons e 719 como ruins. Para comparação, os vinhos brancos foram classificados em 270 bons e 524 ruins. Além disso, o F1-score médio, que foi de 71% para os vinhos brancos, caiu para apenas 33% quando aplicado aos vinhos tintos.

Essa diferença de desempenho pode ser explicada por alguns fatores. Primeiro, a base de dados utilizada no treinamento continha apenas vinhos brancos, e os vinhos tintos possuem características químicas distintas. Como o modelo não foi exposto a essas diferenças durante o treinamento, sua capacidade de generalização foi reduzida. Além disso, o desbalanceamento das bases influencia no resultado. A base de vinhos brancos apresenta uma distribuição de 66% de vinhos classificados como ruins e 34% como bons, enquanto a de vinhos tintos é mais equilibrada, com 52,9% ruins e 47,1% bons.

Como esperado, um modelo treinado exclusivamente com dados de um tipo de vinho não funciona da mesma forma quando testado em outro tipo. Isso reforça a importância de considerar as diferenças entre os dados no treinamento e na validação para garantir um desempenho mais consistente.

In [552…
```
# definir meu X para o dataset red
red_features = ['fixed_acidity', 'volatile_acidity', 'citric_acid', 'residual_sugar', 'chlorides',
                'free_sulfur_dioxide', 'total_sulfur_dioxide', 'density', 'ph', 'sulphates', 'alcohol']

X_red = red[red_features]
```

In [553…
```
# definir meu 'target predict' ou y
y_red = red['opinion']
```

In [561…
```
# transformar os dados de teste com o Scaler de white, já treinado
X_red_scaled = r_scaler.transform(X_red)
```

In [ ]:
```
# fazer a inferência
red_y_hat = rl_best_model.predict(X_red_scaled)
```

In [570…
```
# Exibir o relatório de classificação de vinhos tintos
print("Relatório de classificação para vinhos tintos:")
print(classification_report(y_red, red_y_hat))
```

```
Relatório de classificação para vinhos tintos:
              precision    recall  f1-score   support

           0       0.80      0.01      0.01       719
           1       0.47      1.00      0.64       640

    accuracy                           0.47      1359
   macro avg       0.64      0.50      0.33      1359
weighted avg       0.65      0.47      0.31      1359
```

In [599…
```
# Exibir o relatório de classificação de vinhos brancos, para comparação
print("Relatório de classificação para vinhos brancos:")
white_y_hat = rl_best_model.predict(X_test)
print(classification_report(y_test, white_y_hat))
```

```
Relatório de classificação para vinhos brancos:
              precision    recall  f1-score   support

           0       0.85      0.69      0.77       524
           1       0.56      0.77      0.65       270

    accuracy                           0.72       794
   macro avg       0.71      0.73      0.71       794
weighted avg       0.75      0.72      0.73       794
```

In [ ]:
```
# TODO:
# - aplicar o modelo de árvore de decisão
# - salvar as métricas. Compará-las
# - plotar um gráfico comparando a curva ROC
# - plotar um gráfico de barras da parte de teste de vinhos 'bons' e 'ruins'
```

Parte 7. Exportação do arquivo.pdf

Voltar ao início

7.1. Disponibilize os códigos usados para responder da questão 2-6 em uma conta github e indique o link para o repositório.

Resposta:

O trabalho foi exportado como HTML e convertido em PDF psoteriormente.

Assim que terminar, salve o seu arquivo PDF e poste no Moodle. Utilize o seu nome para nomear o arquivo, identificando também a disciplina no seguinte formato: "nomedoaluno_nomedadisciplina_pd.PDF".

Feito por Mateus Teixeira.