tcc

June 17, 2023

[1]:
```
# Importando as bibliotecas
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Importando as bibliotecas para a construção dos modelos de Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve

from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier

from imblearn.under_sampling import RandomUnderSampler

from yellowbrick.classifier import ROCAUC

# import warnings filter
from warnings import simplefilter

# ignore all future warnings
simplefilter(action='ignore', category=FutureWarning)

# Configurando o notebook
%matplotlib inline
sns.set(style='white')
```

```
# Carregar os dados dos vinhos tintos
dados_vinhos_tintos = pd.read_csv("winequality-red.csv")
dados_vinhos_tintos["Tipo"] = 1  # Adicionar coluna "Tipo" para identificar o␣
 ↪tipo de vinho 1 = Tinto

# Carregar os dados dos vinhos brancos
dados_vinhos_brancos = pd.read_csv("winequality-white.csv")
dados_vinhos_brancos["Tipo"] = 0  # Adicionar coluna "Tipo" para identificar o␣
 ↪tipo de vinho 0 = Branco

print(dados_vinhos_tintos.head())
print(dados_vinhos_brancos.head())
```

```
  fixed acidity;"volatile acidity";"citric acid";"residual
sugar";"chlorides";"free sulfur dioxide";"total sulfur
dioxide";"density";"pH";"sulphates";"alcohol";"quality"  \
0   7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5
1   7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5
2  7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;…
3  11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58…
4   7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5

   Tipo
0     1
1     1
2     1
3     1
4     1
  fixed acidity;"volatile acidity";"citric acid";"residual
sugar";"chlorides";"free sulfur dioxide";"total sulfur
dioxide";"density";"pH";"sulphates";"alcohol";"quality"  \
0   7;0.27;0.36;20.7;0.045;45;170;1.001;3;0.45;8.8;6
1  6.3;0.3;0.34;1.6;0.049;14;132;0.994;3.3;0.49;9…
2  8.1;0.28;0.4;6.9;0.05;30;97;0.9951;3.26;0.44;1…
3  7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4…
4  7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4…

   Tipo
0     0
1     0
2     0
3     0
4     0
```

[2]:
```
# Junção dos datasets de vinhos tintos e brancos
```

```
dados_vinhos = pd.concat([dados_vinhos_tintos, dados_vinhos_brancos],␣
 ↪ignore_index=True)

# Verificar a quantidade de registros no dataset unificado
print("Quantidade de registros de vinhos:", dados_vinhos.shape[0])
```

```
Quantidade de registros de vinhos: 6497
```

[3]:
```
# Visualizar a estrutora do DataFrame
dados_vinhos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 2 columns):
 #   Column
Non-Null Count  Dtype
---  ------
--------------  -----
 0   fixed acidity;"volatile acidity";"citric acid";"residual
sugar";"chlorides";"free sulfur dioxide";"total sulfur
dioxide";"density";"pH";"sulphates";"alcohol";"quality"  6497 non-null   object
 1   Tipo
6497 non-null   int64
dtypes: int64(1), object(1)
memory usage: 101.6+ KB
```

[4]:
```
# Separar as colunas corretamente usando o ponto e vírgula como separador
vinhos_total = dados_vinhos["fixed acidity;\"volatile acidity\";\"citric acid\";
 ↪\"residual sugar\";\"chlorides\";\"free sulfur dioxide\";\"total sulfur␣
 ↪dioxide\";\"density\";\"pH\";\"sulphates\";\"alcohol\";\"quality\""].str.
 ↪split(";", expand=True)

# Renomear as colunas
vinhos_total.columns = [
    "acidez fixa",
    "acidez volátil",
    "ácido cítrico",
    "açúcar residual",
    "cloretos",
    "dióxido de enxofre livre",
    "dióxido de enxofre total",
    "densidade",
    "pH",
    "sulfatos",
    "álcool",
    "qualidade"
]
```

```
# Adicionar a coluna "Tipo" aos dados separados
vinhos_total["tipo"] = dados_vinhos["Tipo"]
```

[5]:
```
# Exibir as primeiras linhas do DataFrame com as colunas renomeadas
vinhos_total.head()
```

[5]:

	acidez fixa	acidez volátil	ácido cítrico	açúcar residual	cloretos	dióxido de enxofre livre	dióxido de enxofre total	densidade	pH	sulfatos	álcool	qualidade	tipo
0	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5	1
1	7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5	1
2	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5	1
3	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6	1
4	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5	1

[6]:
```
# Dimensões do dataset
print("Dimensões do conjunto de dados:\n{} linhas e {} colunas\n".
  ↪format(vinhos_total.shape[0], vinhos_total.shape[1]))

# Primeiras entradas do dataset
print("Primeiras entradas:")
vinhos_total.head()
```

```
Dimensões do conjunto de dados:
6497 linhas e 13 colunas

Primeiras entradas:
```

[6]:

	acidez fixa	acidez volátil	ácido cítrico	açúcar residual	cloretos	dióxido de enxofre livre	dióxido de enxofre total	densidade	pH	sulfatos	álcool	qualidade	tipo
0	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5	1
1	7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5	1
2	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5	1
3	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6	1
4	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5	1

```
0                       11                        34    0.9978  3.51      0.56
1                       25                        67    0.9968   3.2      0.68
2                       15                        54     0.997  3.26      0.65
3                       17                        60     0.998  3.16      0.58
4                       11                        34    0.9978  3.51      0.56

   álcool qualidade  tipo
0    9.4         5     1
1    9.8         5     1
2    9.8         5     1
3    9.8         6     1
4    9.4         5     1
```

[7]:
```
print("Nome dos atributos:\n{}".format(vinhos_total.columns.values))
print("\nQuantidade de valores ausentes por atributo:\n{}".format(vinhos_total.
  ↪isnull().sum()))
print("\nTipo de cada atributo:\n{}".format(vinhos_total.dtypes))
```

```
Nome dos atributos:
['acidez fixa' 'acidez volátil' 'ácido cítrico' 'açúcar residual'
 'cloretos' 'dióxido de enxofre livre' 'dióxido de enxofre total'
 'densidade' 'pH' 'sulfatos' 'álcool' 'qualidade' 'tipo']

Quantidade de valores ausentes por atributo:
acidez fixa                 0
acidez volátil              0
ácido cítrico               0
açúcar residual             0
cloretos                    0
dióxido de enxofre livre    0
dióxido de enxofre total    0
densidade                   0
pH                          0
sulfatos                    0
álcool                      0
qualidade                   0
tipo                        0
dtype: int64

Tipo de cada atributo:
acidez fixa                 object
acidez volátil              object
ácido cítrico               object
açúcar residual             object
cloretos                    object
dióxido de enxofre livre    object
dióxido de enxofre total    object
densidade                   object
```

```
pH                          object
sulfatos                    object
álcool                      object
qualidade                   object
tipo                         int64
dtype: object
```

[8]:
```
#Verificar presença de nulos
vinhos_total.isnull().any()
```

[8]:
```
acidez fixa                 False
acidez volátil              False
ácido cítrico               False
açúcar residual             False
cloretos                    False
dióxido de enxofre livre    False
dióxido de enxofre total    False
densidade                   False
pH                          False
sulfatos                    False
álcool                      False
qualidade                   False
tipo                        False
dtype: bool
```

[9]:
```
# Converter colunas relevantes para tipos numéricos
colunas_numericas = ['acidez fixa', 'acidez volátil', 'ácido cítrico', 'açúcar␣
 ↪residual', 'cloretos', 'dióxido de enxofre livre', 'dióxido de enxofre␣
 ↪total', 'densidade', 'pH', 'sulfatos', 'álcool', 'qualidade']
vinhos_total[colunas_numericas] = vinhos_total[colunas_numericas].astype(float)

print("\nTipo de cada atributo:\n{}".format(vinhos_total.dtypes))
```

```
Tipo de cada atributo:
acidez fixa                 float64
acidez volátil              float64
ácido cítrico               float64
açúcar residual             float64
cloretos                    float64
dióxido de enxofre livre    float64
dióxido de enxofre total    float64
densidade                   float64
pH                          float64
sulfatos                    float64
álcool                      float64
qualidade                   float64
tipo                          int64
```

```
dtype: object
```

[10]:
```
#Analisar o dataset
#O objetivo é entender se podemos considerar o dataset como um todo ou se␣
 ↪devemos observá-los por tipo de vinho para isso iremos agregar os dados por␣
 ↪tipo de vinho e ver como as variáveis se comportam

agrupado_tipo = vinhos_total.groupby('tipo').std()
print(agrupado_tipo)
```

```
      acidez fixa  acidez volátil  ácido cítrico  açúcar residual  cloretos  \
tipo
0        0.843868        0.100795       0.121020         5.072058  0.021848
1        1.741096        0.179060       0.194801         1.409928  0.047065

      dióxido de enxofre livre  dióxido de enxofre total  densidade        pH  \
tipo
0                    17.007137                 42.498065   0.002991  0.151001
1                    10.460157                 32.895324   0.001887  0.154386

      sulfatos    álcool  qualidade
tipo
0     0.114126  1.230621   0.885639
1     0.169507  1.065668   0.807569
```

[12]:
```
vinhos_total.describe()
```

[12]:
```
        acidez fixa  acidez volátil  ácido cítrico  açúcar residual  \
count  6497.000000     6497.000000    6497.000000      6497.000000
mean      7.215307        0.339666       0.318633         5.443235
std       1.296434        0.164636       0.145318         4.757804
min       3.800000        0.080000       0.000000         0.600000
25%       6.400000        0.230000       0.250000         1.800000
50%       7.000000        0.290000       0.310000         3.000000
75%       7.700000        0.400000       0.390000         8.100000
max      15.900000        1.580000       1.660000        65.800000

          cloretos  dióxido de enxofre livre  dióxido de enxofre total  \
count  6497.000000               6497.000000               6497.000000
mean      0.056034                 30.525319                115.744574
std       0.035034                 17.749400                 56.521855
min       0.009000                  1.000000                  6.000000
25%       0.038000                 17.000000                 77.000000
50%       0.047000                 29.000000                118.000000
75%       0.065000                 41.000000                156.000000
max       0.611000                289.000000                440.000000

         densidade           pH     sulfatos       álcool    qualidade  \
```

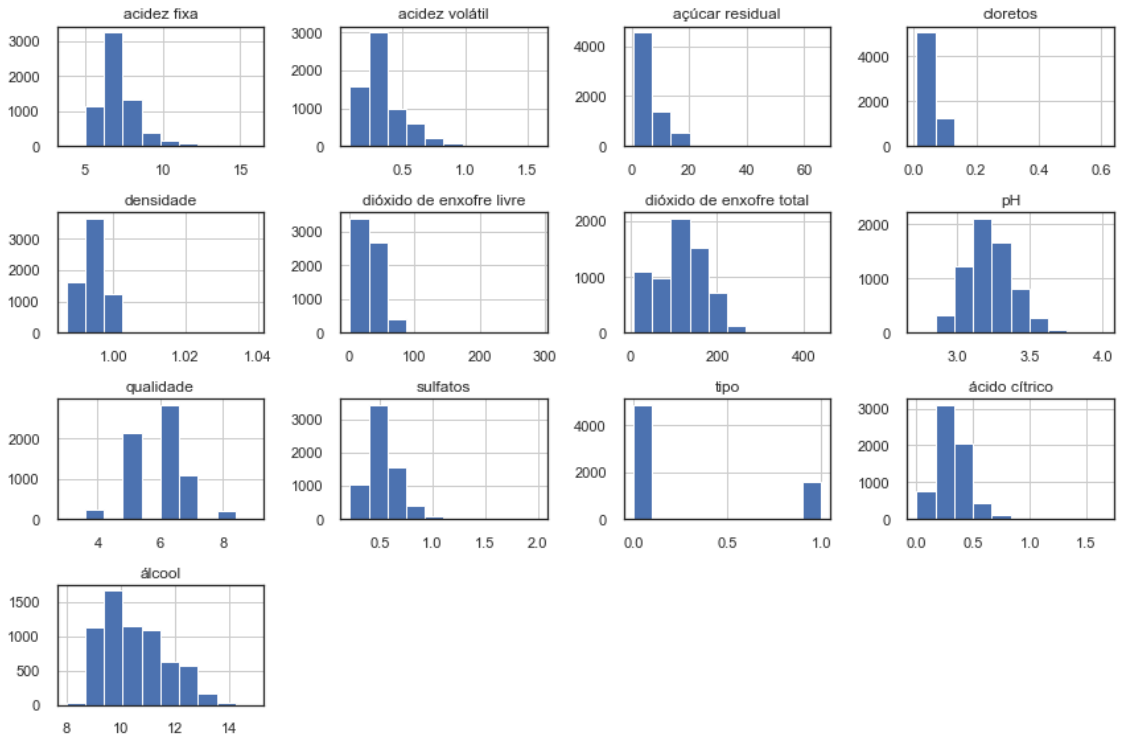
```
count  6497.000000  6497.000000  6497.000000  6497.000000  6497.000000
mean      0.994697     3.218501     0.531268    10.491801     5.818378
std       0.002999     0.160787     0.148806     1.192712     0.873255
min       0.987110     2.720000     0.220000     8.000000     3.000000
25%       0.992340     3.110000     0.430000     9.500000     5.000000
50%       0.994890     3.210000     0.510000    10.300000     6.000000
75%       0.996990     3.320000     0.600000    11.300000     6.000000
max       1.038980     4.010000     2.000000    14.900000     9.000000

              tipo
count  6497.000000
mean      0.246114
std       0.430779
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       1.000000
```

[13]:
```
# Categorizando a variável qualidade e criando a variável qualidade_cat
vinhos_total['qualidade_cat'] = pd.cut(vinhos_total['qualidade'], bins=(2, 6.5,␣
↪8), labels = [0, 1])
```

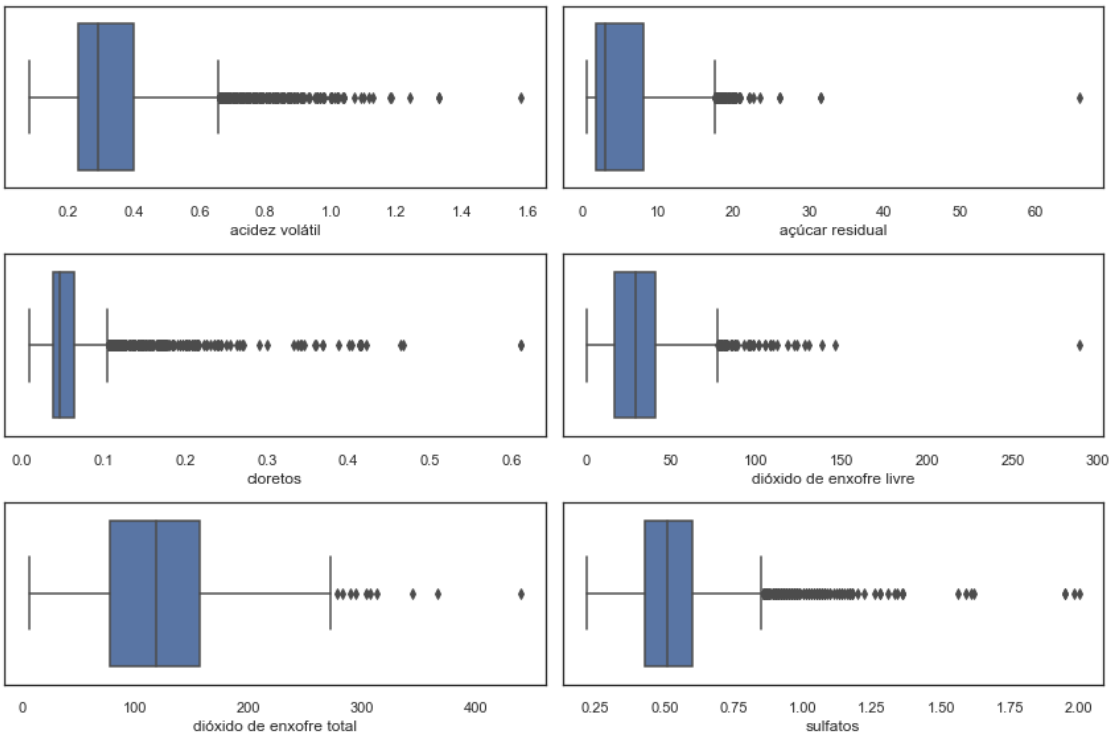
[14]:
```
# Verificando quais as entradas únicas da variável qualidade_cat
vinhos_total['qualidade_cat'] = vinhos_total['qualidade_cat'].astype('category')
vinhos_total['qualidade_cat'].unique()
```

[14]:
```
[0, 1, NaN]
Categories (2, int64): [0 < 1]
```

[15]:
```
# Criar um histograma das colunas numéricas
vinhos_total.hist(bins=10, figsize=(12, 8))
plt.tight_layout()
plt.show()
```



[16]:
```
# Configurando o plot
fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(12, 8))

sns.boxplot(vinhos_total['acidez volátil'], ax=ax[0, 0])
sns.boxplot(vinhos_total['açúcar residual'], ax=ax[0, 1])
sns.boxplot(vinhos_total['cloretos'], ax=ax[1, 0])
sns.boxplot(vinhos_total['dióxido de enxofre livre'], ax=ax[1, 1])
sns.boxplot(vinhos_total['dióxido de enxofre total'], ax=ax[2, 0])
sns.boxplot(vinhos_total['sulfatos'], ax=ax[2, 1])

plt.tight_layout()
```



[17]:
```
# Identificando os outliers para a variável residual sugar
q1_rsugar = vinhos_total['açúcar residual'].quantile(0.25)
q3_rsugar = vinhos_total['açúcar residual'].quantile(0.75)
IQR_rsugar = q3_rsugar - q1_rsugar

print("IQR da variável açúcar residual: {}\n".format(round(IQR_rsugar, 2)))

# Definindo os limites para a variável residual sugar
sup_rsugar = q3_rsugar + 1.5 * IQR_rsugar
inf_rsugar = q1_rsugar - 1.5 * IQR_rsugar

print("Limite superior de açúcar residual: {}".format(round(sup_rsugar, 2)))
print("Limite inferior de açúcar residual: {}".format(round(inf_rsugar, 2)))
```

```
IQR da variável açúcar residual: 6.3

Limite superior de açúcar residual: 17.55
Limite inferior de açúcar residual: -7.65
```

[18]:
```
cut_rsugar = len(vinhos_total[vinhos_total['açúcar residual'] < 0.85]) +␣
 ↪len(vinhos_total[vinhos_total['açúcar residual'] > 3.65])
```

```
print("As entradas da variável açúcar residual fora dos limites representam {}␣
 ↪% do dataset.\n".format(round((cut_rsugar / vinhos_total.shape[0]) * 100,␣
 ↪2)))
```

```
As entradas da variável açúcar residual fora dos limites representam 47.21 % do
dataset.
```

[19]:
```
# Identificando os outliers para a variável cloretos
q1_chlo = vinhos_total['cloretos'].quantile(0.25)
q3_chlo = vinhos_total['cloretos'].quantile(0.75)
IQR_chlo = q3_chlo - q1_chlo

print("IQR da variável cloretos: {}\n".format(round(IQR_chlo, 2)))

# Definindo os limites para a variável cloretos
sup_chlo = q3_chlo + 1.5 * IQR_chlo
inf_chlo = q1_chlo - 1.5 * IQR_chlo

print("Limite superior de cloretos: {}".format(round(sup_chlo, 2)))
print("Limite inferior de cloretos: {}".format(round(inf_chlo, 2)))
```

```
IQR da variável cloretos: 0.03

Limite superior de cloretos: 0.11
Limite inferior de cloretos: -0.0
```

[20]:
```
cut_chlo = len(vinhos_total[vinhos_total['cloretos'] < 0.04]) +␣
 ↪len(vinhos_total[vinhos_total['cloretos'] > 0.12])

print("As entradas da variável cloretos fora dos limites representam {} % do␣
 ↪dataset.\n".format(round((cut_chlo / vinhos_total.shape[0]) * 100, 2)))
```

```
As entradas da variável cloretos fora dos limites representam 32.03 % do
dataset.
```

[21]:
```
# Identificando os outliers para a variável sulfatos
q1_sulp = vinhos_total['sulfatos'].quantile(0.25)
q3_sulp = vinhos_total['sulfatos'].quantile(0.75)
IQR_sulp = q3_sulp - q1_sulp

print("IQR da variável sulfatos: {}\n".format(round(IQR_sulp, 2)))

# Definindo os limites para a variável sulfatos
sup_sulp = q3_sulp + 1.5 * IQR_sulp
inf_sulp = q1_sulp - 1.5 * IQR_sulp
```

```
print("Limite superior de sulfatos: {}".format(round(sup_sulp, 2)))
print("Limite inferior de sulfatos: {}".format(round(inf_sulp, 2)))
```

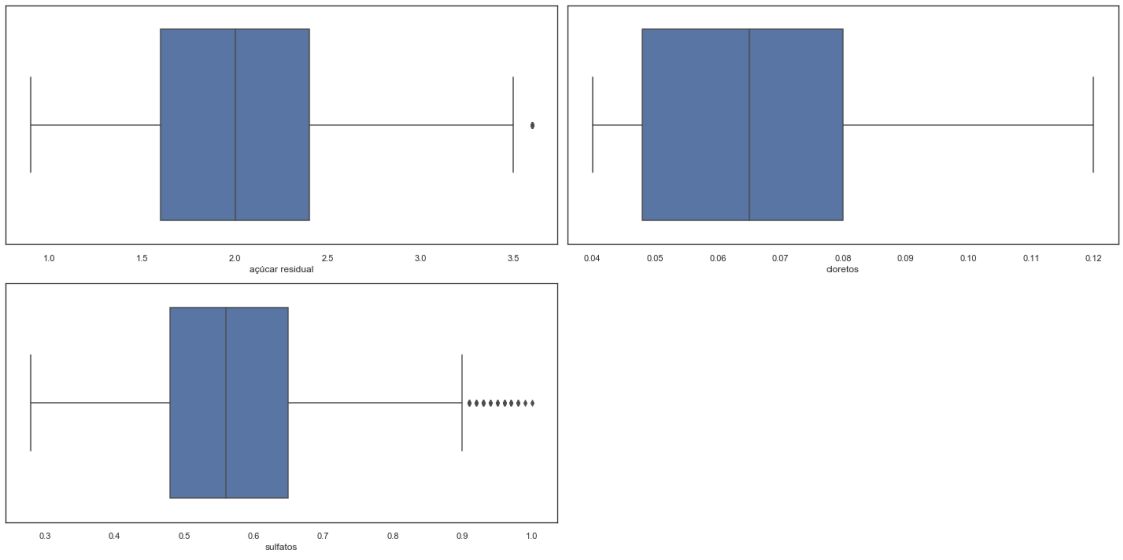
```
IQR da variável sulfatos: 0.17

Limite superior de sulfatos: 0.86
Limite inferior de sulfatos: 0.18
```

[22]:
```
cut_sulp = len(vinhos_total[vinhos_total['sulfatos'] < 0.28]) +␣
 ↪len(vinhos_total[vinhos_total['sulfatos'] > 1.0])

print("As entradas da variável sulfatos fora dos limites representam {} % do␣
 ↪dataset.\n".format(round((cut_sulp / vinhos_total.shape[0]) * 100, 2)))
```

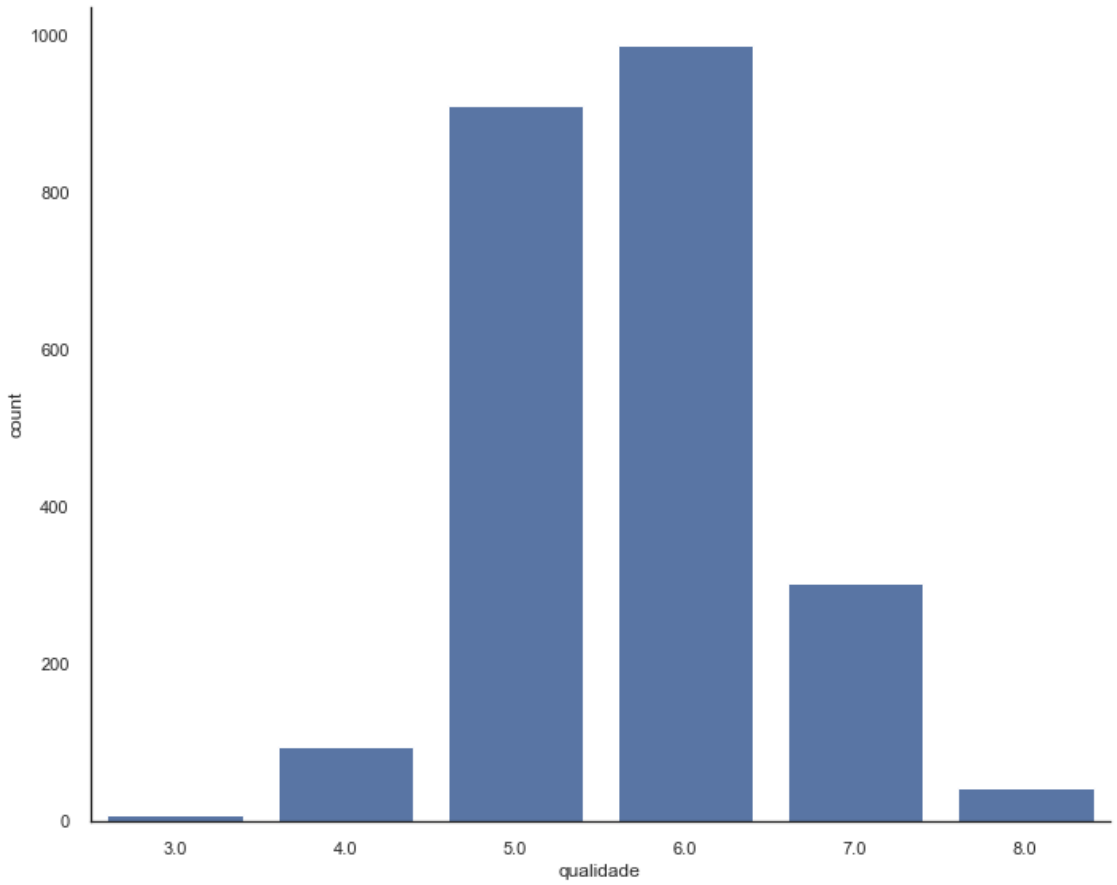
```
As entradas da variável sulfatos fora dos limites representam 1.29 % do dataset.
```

[23]:
```
vinhos_total_clean = vinhos_total.copy()

vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['açúcar␣
 ↪residual'] > 3.65].index, axis=0, inplace=True)
vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['açúcar␣
 ↪residual'] < 0.85].index, axis=0, inplace=True)
vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['cloretos'] > 0.
 ↪12].index, axis=0, inplace=True)
vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['cloretos'] < 0.
 ↪04].index, axis=0, inplace=True)
vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['sulfatos'] > 1.
 ↪0].index, axis=0, inplace=True)
vinhos_total_clean.drop(vinhos_total_clean[vinhos_total_clean['sulfatos'] < 0.
 ↪28].index, axis=0, inplace=True)
```

[24]:
```
# Configurando o plot
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(20, 10))
fig.delaxes(ax[1,1])

sns.boxplot(vinhos_total_clean['açúcar residual'], ax=ax[0, 0])
sns.boxplot(vinhos_total_clean['cloretos'], ax=ax[0, 1])
sns.boxplot(vinhos_total_clean['sulfatos'], ax=ax[1, 0])

plt.tight_layout()
```



[25]:
```
# Construindo o gráfico
fig, ax = plt.subplots(figsize=(10,8))

sns.countplot(vinhos_total_clean['qualidade'], color='b', ax=ax)
sns.despine()

plt.tight_layout()
```



[26]:
```
# Porcentagem de cada classe
print("A classe 0 representa {} % de todas as entradas.".
  ↪format(round((len(vinhos_total_clean[vinhos_total_clean['qualidade_cat'] ==␣
  ↪0]) / vinhos_total_clean.shape[0]) * 100, 2)))
print("A classe 1 representa {} % de todas as entradas.".
  ↪format(round((len(vinhos_total_clean[vinhos_total_clean['qualidade_cat'] ==␣
  ↪1]) / vinhos_total_clean.shape[0]) * 100, 2)))
```

```
A classe 0 representa 85.31 % de todas as entradas.
A classe 1 representa 14.69 % de todas as entradas.
```

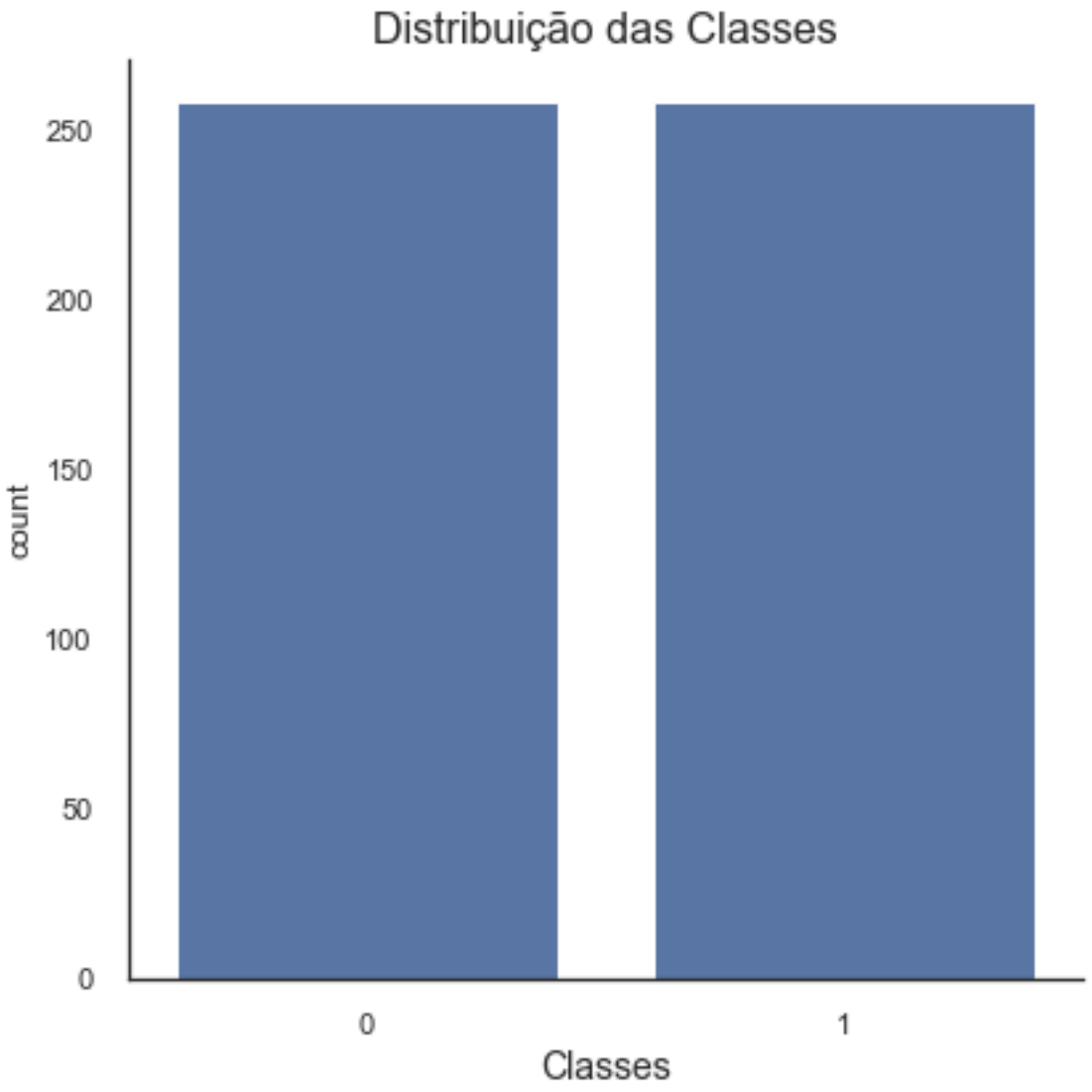
[27]:
```
# Cronstruindo o gráfico de barras
fig, ax = plt.subplots(figsize=(6,6))

sns.countplot(vinhos_total_clean['qualidade_cat'], color='b', ax=ax)
sns.despine()

ax.set_title("Distribuição das Classes", fontsize=16)
ax.set_xlabel("Classes", fontsize=14)
```

```
plt.tight_layout()
```

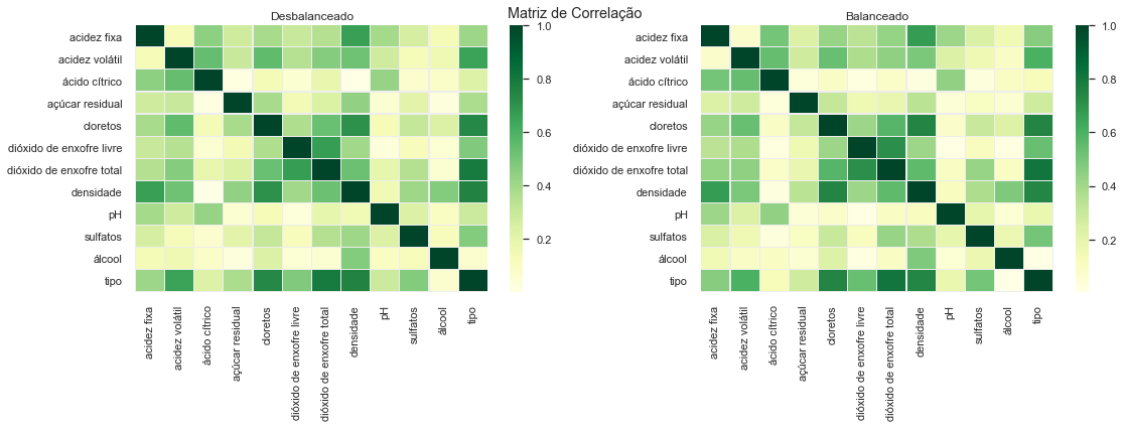
[28]:
```
# Separando os dados entre feature matrix e target vector
X = vinhos_total_clean.drop(['qualidade', 'qualidade_cat'], axis=1)
y = vinhos_total_clean['qualidade_cat'] # Pois usaremos apenas a separação␣
 ↪entre "ruim" ou "bom" (0 ou 1)

# Dividindo os dados entre treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)
```

[29]:
```
# Definindo o modelo para balancear
und = RandomUnderSampler()

X_und, y_und = und.fit_resample(X_train, y_train)

# Verificando o balanceamento dos dados
print(pd.Series(y_und).value_counts(), "\n")

# Plotando a nova Distribuição de Classes
fig, ax = plt.subplots(figsize=(6,6))

sns.countplot(pd.Series(y_und), color='b', ax=ax)

sns.despine()

ax.set_title("Distribuição das Classes", fontsize=16)
ax.set_xlabel("Classes", fontsize=14)

plt.tight_layout()
```

```
1    258
0    258
Name: qualidade_cat, dtype: int64
```



[30]:
```
# Construindo o heatmap
fig, ax = plt.subplots(figsize=(16, 6), nrows=1, ncols=2)
fig.suptitle("Matriz de Correlação")

sns.heatmap(X_train.corr().abs(), cmap='YlGn', linecolor='#eeeeee',␣
 ↪linewidths=0.1, ax=ax[0])
ax[0].set_title("Desbalanceado")

sns.heatmap(X_und.corr().abs(), cmap='YlGn', linecolor='#eeeeee', linewidths=0.
 ↪1, ax=ax[1])
ax[1].set_title("Balanceado")

plt.tight_layout()
```



[31]:
```
dados_base_tinto = vinhos_total[vinhos_total['tipo'] == 0].iloc[:, :15].copy()
dados_base_branco = vinhos_total[vinhos_total['tipo'] != 0].iloc[:, :15].copy()
```

[32]:
```
# Separando os dados entre feature matrix e target vector
dados_base_tinto_clean = vinhos_total_clean.copy()

X = dados_base_tinto_clean.drop(['qualidade', 'qualidade_cat'], axis=1)
y = dados_base_tinto_clean['qualidade_cat'] # Pois usaremos apenas a separação␣
  ↪entre "ruim" ou "bom" (0 ou 1)

# Dividindo os dados entre treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)
```

[33]:
```
# Modelo Random Forest
rf_model = RandomForestClassifier()

# Definindo o melhor parâmetro
parameters = {'n_estimators': range(25, 1000, 25)}

kfold = StratifiedKFold(n_splits=5, shuffle=True)

rf_clf = GridSearchCV(rf_model, parameters, cv=kfold)
rf_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhor parâmetro: {}".format(rf_clf.best_params_))
```

```
Melhor parâmetro: {'n_estimators': 600}
```

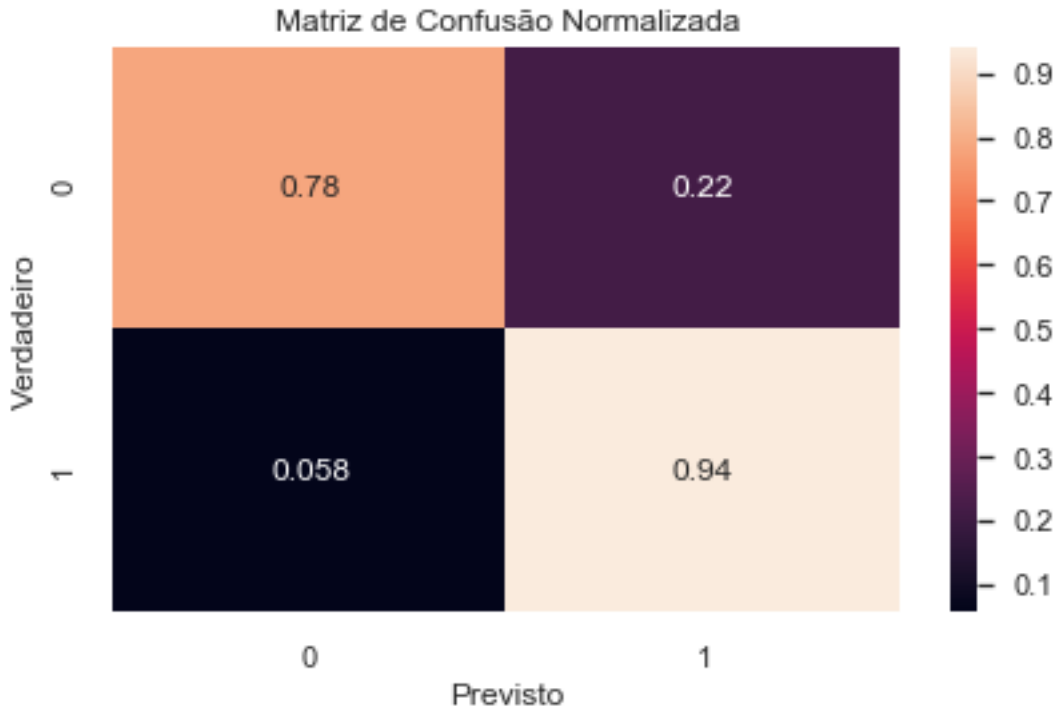
[34]:
```
# Definindo o modelo com n_estimators igual a 375
rf_model = RandomForestClassifier(n_estimators = 375)
```

```
# Fit do modelo
rf_model.fit(X_und, y_und)

# Testando o modelo
y_pred_rf = rf_model.predict(X_test)
y_prob_rf = rf_model.predict_proba(X_test)
```

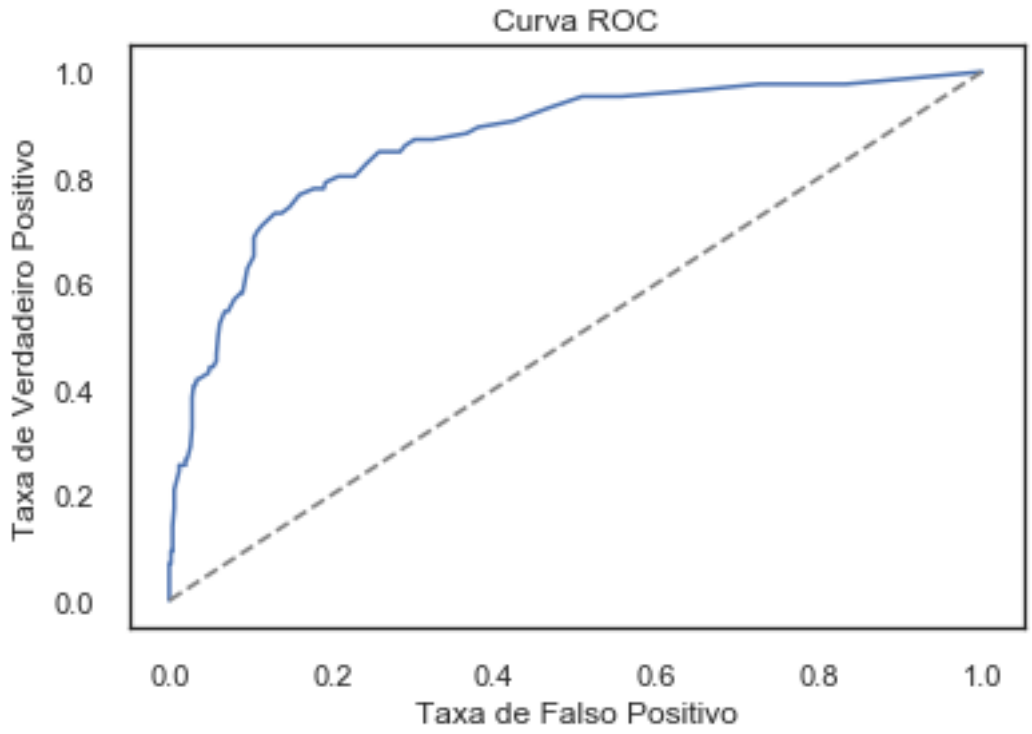
[35]:
```
# Relatório de classificação
print("Relatório de classificação para o Random Forest:\n",␣
 ↪classification_report(y_test, y_pred_rf, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_rf) * 100, 2)))
```

```
Relatório de classificação para o Random Forest:
              precision    recall  f1-score   support

           0     0.9874    0.7840    0.8740       500
           1     0.4286    0.9419    0.5891        86

    accuracy                         0.8072       586
   macro avg     0.7080    0.8629    0.7316       586
weighted avg     0.9054    0.8072    0.8322       586

Área sob a curva (AUC): 86.29%
```

[36]:
```
# Matriz de confusão
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_rf, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[44]:
```
# Converter y_test em valores numéricos
y_test_numeric = y_test.cat.codes

# Treinar o modelo de classificação
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)

# Calcular a probabilidade das classes positivas
y_pred_proba = rf_model.predict_proba(X_test)[:, 1]

# Calcular a curva ROC
fpr, tpr, thresholds = roc_curve(y_test_numeric, y_pred_proba)

# Plotar a curva ROC
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('Taxa de Falso Positivo')
plt.ylabel('Taxa de Verdadeiro Positivo')
plt.title('Curva ROC')
plt.show()
```



[84]:
```
# Modelo XGBoost
xgb_model = XGBClassifier(learning_rate=0.1)

# Definindo os melhores parâmetros
param_gs = {'n_estimators': range(0, 1000, 50)}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'n_estimators': 100}
```

[88]:
```
# Modelo XGBoost
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=100, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {
    'max_depth': range(1, 8, 1),
```

```
    'min_child_weigth': range(1, 5, 1),
    }

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'max_depth': 3, 'min_child_weigth': 1}
```

[89]:
```
# Modelo XGBoost
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=100, max_depth=3,␣
 ↪min_child_weigth=1, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {'gamma': [i/10.0 for i in range(0,5)]}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'gamma': 0.0}
```

[91]:
```
# Modelo XGBoost
xgb_model = XGBClassifier(n_estimators=100, max_depth=3, min_child_weigth=1,␣
 ↪gamma=0.0, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {'learning_rate': [0.001, 0.01, 0.1, 1]}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

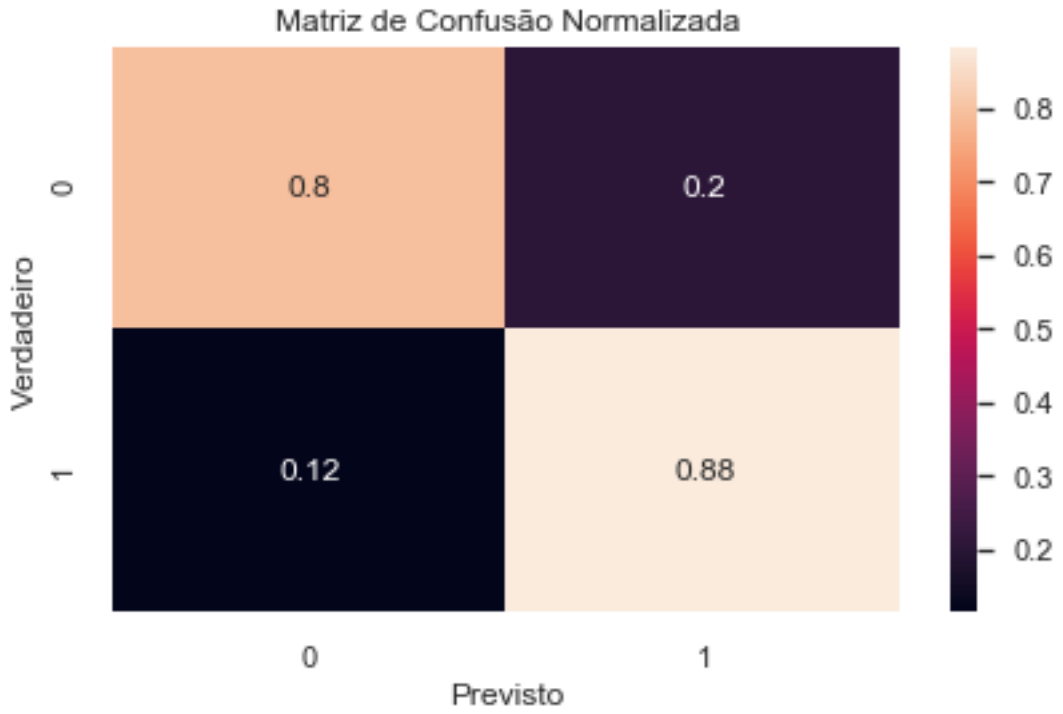
```
Melhores parâmetros: {'learning_rate': 0.1}
```

[92]:
```
# Modelo XGBoost final
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=100, max_depth=3,␣
 ↪min_child_weigth=1, gamma=0.0, verbosity=0)

# Treinando o modelo
xgb_model.fit(X_und, y_und)

# Fazendo previsões
y_pred_xgb = xgb_model.predict(X_test)
y_prob_xgb = xgb_model.predict_proba(X_test)
```

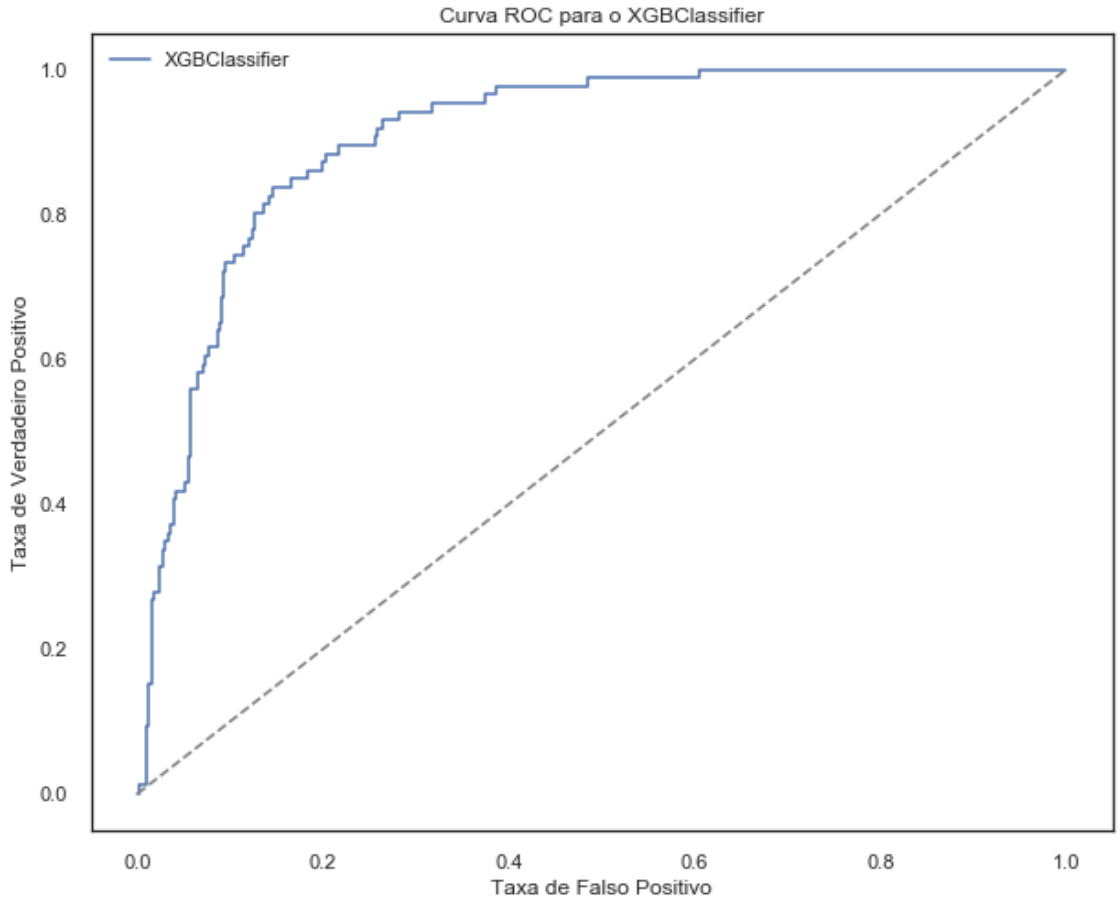
[190]:
```
# Relatório de classificação
print("Relatório de classificação para o XGBClassifier:\n",␣
 ↪classification_report(y_test, y_pred_xgb, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_xgb) * 100, 2)))
```

```
Relatório de classificação para o XGBClassifier:
              precision    recall  f1-score   support

           0     0.9807    0.7120    0.8250       500
           1     0.3543    0.9186    0.5113        86

    accuracy                         0.7423       586
   macro avg     0.6675    0.8153    0.6682       586
weighted avg     0.8888    0.7423    0.7790       586

Área sob a curva (AUC): 81.53%
```

[93]:
```
# Matriz de confusão
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_xgb, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[94]:
```
# Ajustar o modelo aos dados de treinamento
xgb_model.fit(X_und, y_und)

# Converter y_test para valores numéricos
y_test_numeric = y_test.cat.codes

# Calcular as probabilidades de previsão
y_pred_proba = xgb_model.predict_proba(X_test)

# Calcular a pontuação ROC
fpr, tpr, _ = roc_curve(y_test_numeric, y_pred_proba[:, 1])

# Plotar a curva ROC
fig, ax = plt.subplots(figsize=(10,8))
ax.plot(fpr, tpr, label="XGBClassifier")
ax.plot([0, 1], [0, 1], color='gray', linestyle='--')
ax.set_xlabel("Taxa de Falso Positivo")
ax.set_ylabel("Taxa de Verdadeiro Positivo")
ax.set_title("Curva ROC para o XGBClassifier")
ax.legend()
plt.show()
```



[95]:
```
# Regressão logística
rl_model = LogisticRegression()
# Padronizando os dados de treino
scaler = StandardScaler().fit(X_und)
X_und_std = scaler.transform(X_und)

# Definindo o melhor parâmetro
param_rl = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# Identificando os melhor parâmetro
kfold = StratifiedKFold(n_splits=5, shuffle=True)

rl_clf = GridSearchCV(rl_model, param_rl, cv=kfold)
rl_clf.fit(X_und_std, y_und)

# Visualizar o melhor parâmetro
print("Melhor parâmetro: {}".format(rl_clf.best_params_))
```

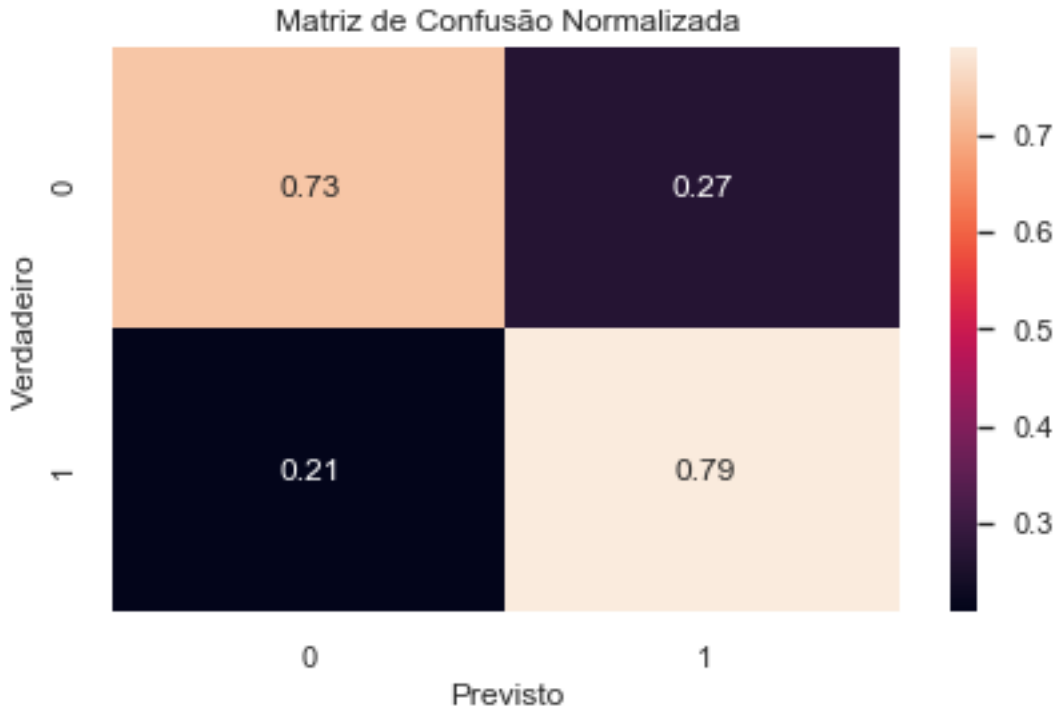
```
Melhor parâmetro: {'C': 1}
```

[96]:
```
# Regressão Logística com pipeline
rl_model = make_pipeline(StandardScaler(), LogisticRegression(C=1))

# Treinando o modelo
rl_model.fit(X_und, y_und)

# Fazendo previsões
y_pred_rl = rl_model.predict(X_test)
y_prob_rl = rl_model.predict_proba(X_test)
```

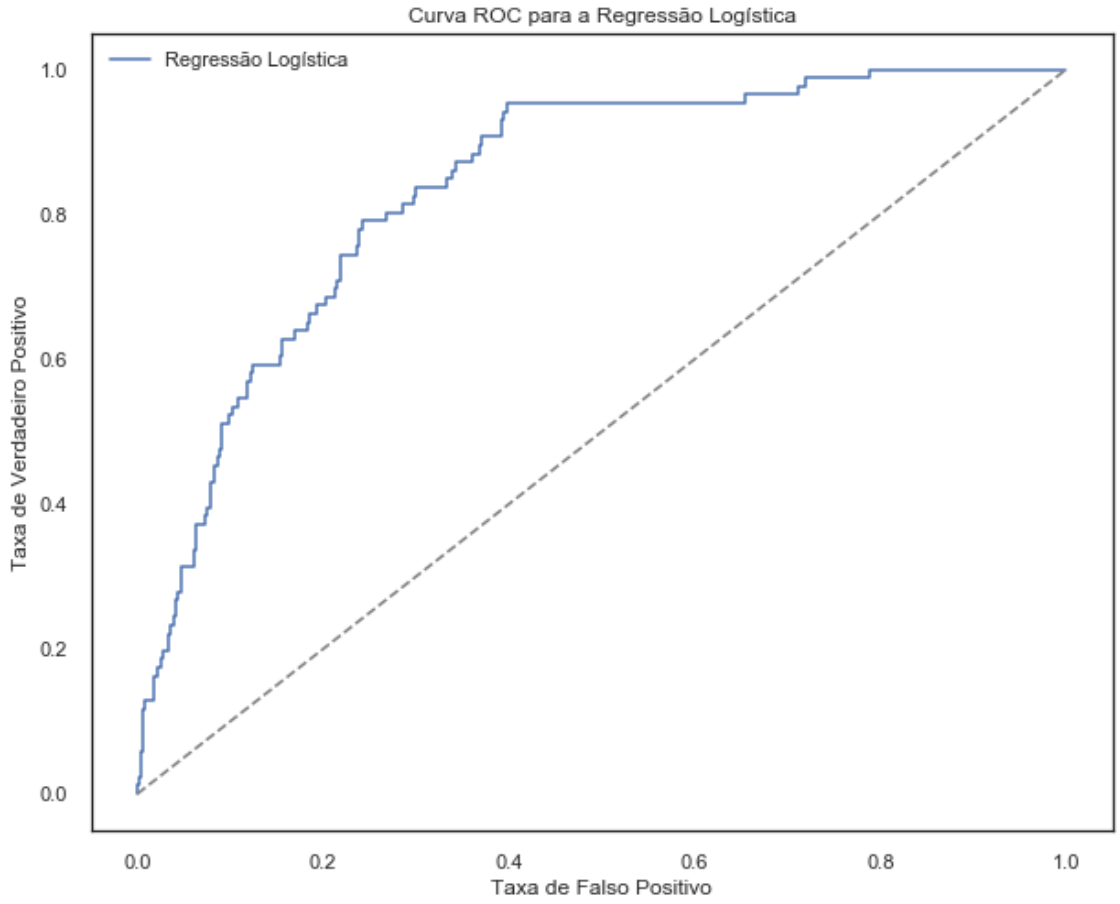
[98]:
```
# Relatório de classificação
print("Relatório de classificação para a Regressão Logística:\n",␣
 ↪classification_report(y_test, y_pred_rl, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_rl) * 100, 2)))
```

```
Relatório de classificação para a Regressão Logística:
              precision    recall  f1-score   support

           0     0.9531    0.7320    0.8281       500
           1     0.3366    0.7907    0.4722        86

    accuracy                         0.7406       586
   macro avg     0.6449    0.7613    0.6501       586
weighted avg     0.8627    0.7406    0.7758       586

Área sob a curva (AUC): 76.13%
```

[99]:
```
# Matriz de confusão
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_rl, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[100]:
```
# Ajustar o modelo aos dados de treinamento
rl_model.fit(X_und, y_und)

# Converter y_test para valores numéricos
y_test_numeric = y_test.cat.codes

# Calcular as probabilidades de previsão
y_pred_proba = rl_model.predict_proba(X_test)

# Calcular a pontuação ROC
fpr, tpr, _ = roc_curve(y_test_numeric, y_pred_proba[:, 1])

# Plotar a curva ROC
fig, ax = plt.subplots(figsize=(10,8))
ax.plot(fpr, tpr, label="Regressão Logística")
ax.plot([0, 1], [0, 1], color='gray', linestyle='--')
ax.set_xlabel("Taxa de Falso Positivo")
ax.set_ylabel("Taxa de Verdadeiro Positivo")
ax.set_title("Curva ROC para a Regressão Logística")
ax.legend()
plt.show()
```



[101]:
```
dados_base_branco_clean = vinhos_total_clean.copy()

X = dados_base_branco_clean.drop(['qualidade', 'qualidade_cat'], axis=1)
y = dados_base_branco_clean['qualidade_cat'] # Pois usaremos apenas a separação␣
 ↪entre "ruim" ou "bom" (0 ou 1)

# Dividindo os dados entre treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)
rf_model = RandomForestClassifier()
# Definindo o melhor parâmetro
parameters = {'n_estimators': range(25, 1000, 25)}

kfold = StratifiedKFold(n_splits=5, shuffle=True)

rf_clf = GridSearchCV(rf_model, parameters, cv=kfold)
rf_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
```

```
print("Melhor parâmetro: {}".format(rf_clf.best_params_))
```

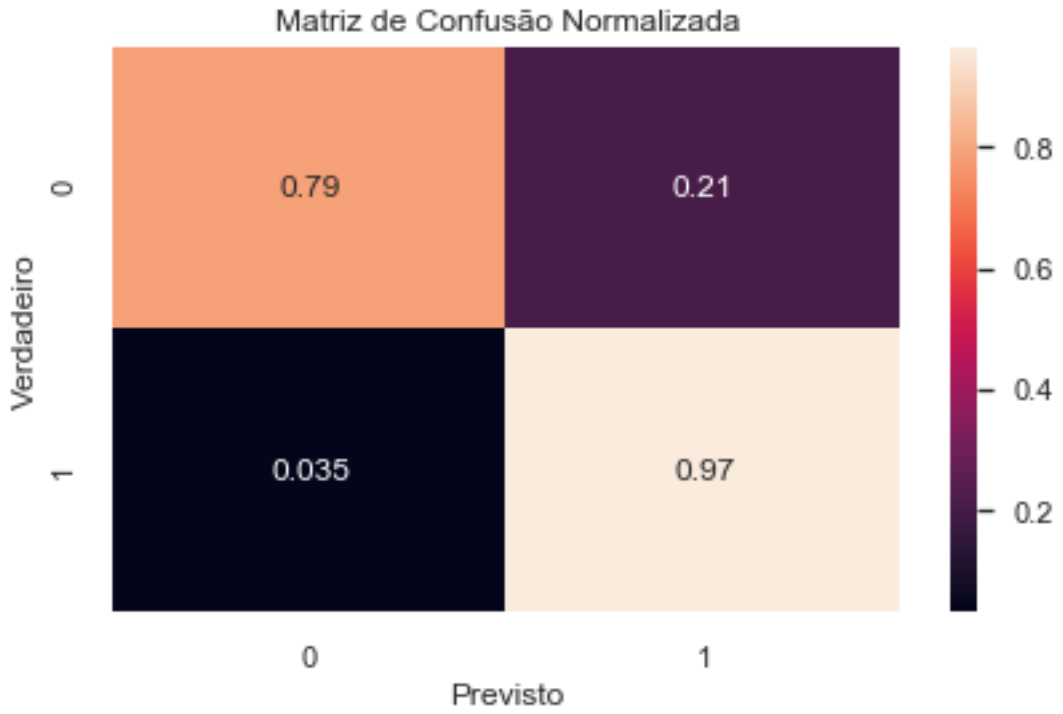
```
Melhor parâmetro: {'n_estimators': 750}
```

[102]:
```
# Definindo o modelo com n_estimators igual a 750
rf_model = RandomForestClassifier(n_estimators = 750)

# Fit do modelo
rf_model.fit(X_und, y_und)

# Testando o modelo
y_pred_rf = rf_model.predict(X_test)
y_prob_rf = rf_model.predict_proba(X_test)
```

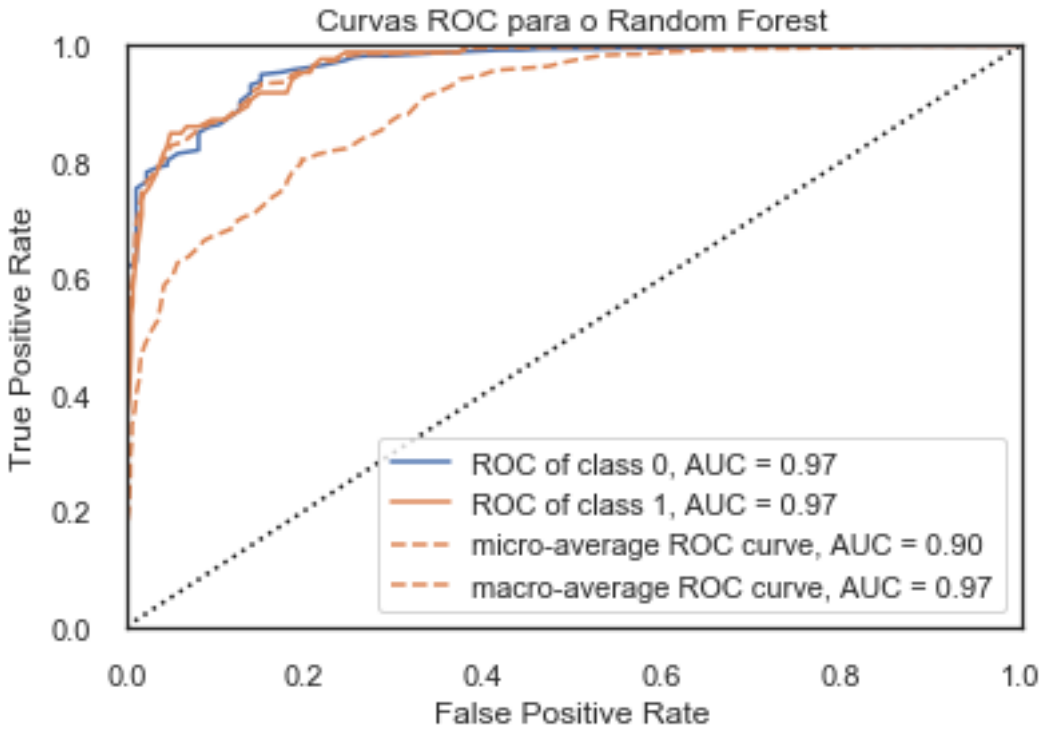
[165]:
```
# Relatório de classificação
print("Relatório de classificação para o Random Forest:\n",␣
 ↪classification_report(y_test, y_pred_rf, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_rf) * 100, 2)))
```

```
Relatório de classificação para o Random Forest:
               precision    recall  f1-score   support

           0     0.9846    0.7680    0.8629       500
           1     0.4082    0.9302    0.5674        86

    accuracy                         0.7918       586
   macro avg     0.6964    0.8491    0.7151       586
weighted avg     0.9000    0.7918    0.8195       586

Área sob a curva (AUC): 84.91%
```

[103]:
```
# Matriz de confusão
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_rf, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[104]:
```
# Converter y_test para valores numéricos
y_test_numeric = y_test.cat.codes

# Calcular a pontuação com os dados de teste
vis_rf.score(X_test, y_test_numeric)

# Mostrar a figura
vis_rf.show()
```



[104]: <matplotlib.axes._subplots.AxesSubplot at 0x244e3458d48>

[105]:
```
xgb_model = XGBClassifier(learning_rate=0.1)

# Definindo os melhores parâmetros
param_gs = {'n_estimators': range(0, 1000, 50)}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'n_estimators': 400}
```

[106]:
```
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=400, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {
    'max_depth': range(1, 8, 1),
    'min_child_weigth': range(1, 5, 1),
```

```
    }

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'max_depth': 3, 'min_child_weigth': 1}
```

[107]:
```
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=400, max_depth=2,␣
 ↪min_child_weigth=1, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {'gamma': [i/10.0 for i in range(0,5)]}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

```
Melhores parâmetros: {'gamma': 0.4}
```

[108]:
```
xgb_model = XGBClassifier(n_estimators=400, max_depth=2, min_child_weigth=1,␣
 ↪gamma=0.4, verbosity=0)

# Definindo os melhores parâmetros
param_gs = {'learning_rate': [0.001, 0.01, 0.1, 1]}

# Identificando os melhores parâmetros
kfold = StratifiedKFold(n_splits=5, shuffle=True)

xgb_clf = GridSearchCV(xgb_model, param_gs, cv=kfold)
xgb_clf.fit(X_und, y_und)

# Visualizar o melhor parâmetro
print("Melhores parâmetros: {}".format(xgb_clf.best_params_))
```

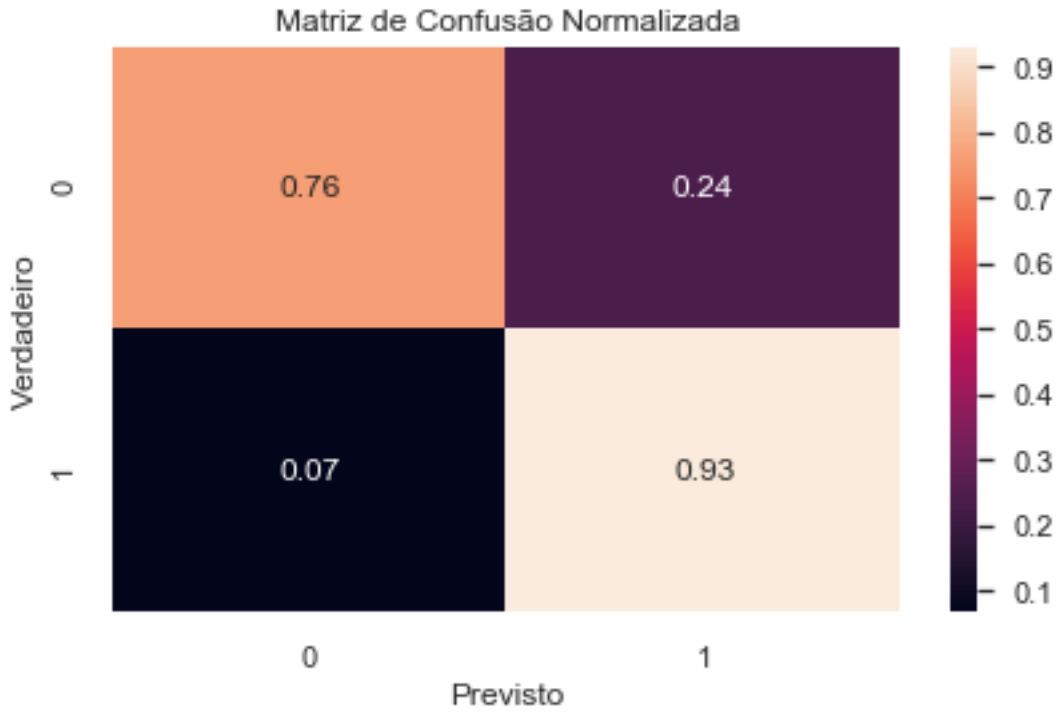
```
Melhores parâmetros: {'learning_rate': 0.1}
```

[109]:
```
xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=400, max_depth=2,␣
 ↪min_child_weigth=1, gamma=0.4, verbosity=0)

# Treinando o modelo
xgb_model.fit(X_und, y_und)

# Fazendo previsões
y_pred_xgb = xgb_model.predict(X_test)
y_prob_xgb = xgb_model.predict_proba(X_test)
```

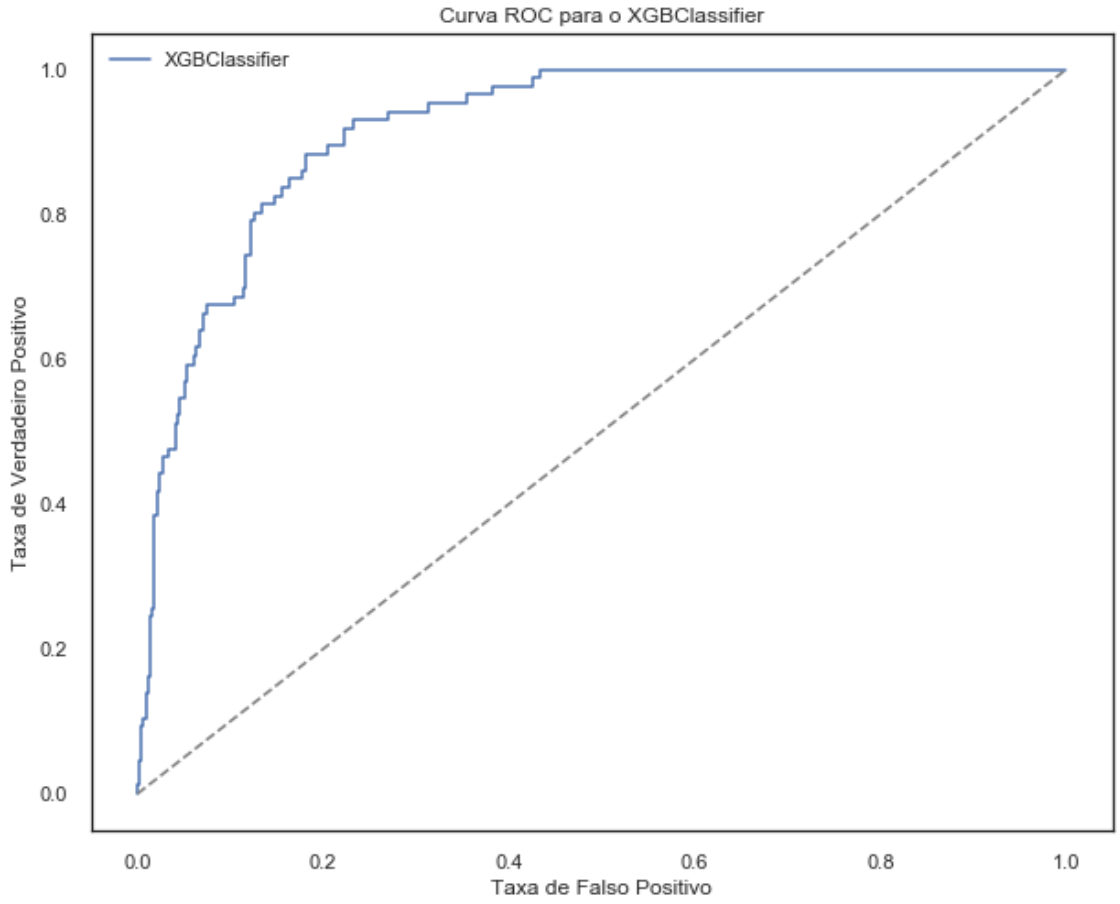
[110]:
```
# Relatório de classificação
print("Relatório de classificação para o XGBClassifier:\n",␣
 ↪classification_report(y_test, y_pred_xgb, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_xgb) * 100, 2)))
```

```
Relatório de classificação para o XGBClassifier:
              precision    recall  f1-score   support

           0     0.9845    0.7620    0.8591       500
           1     0.4020    0.9302    0.5614        86

    accuracy                         0.7867       586
   macro avg     0.6933    0.8461    0.7102       586
weighted avg     0.8990    0.7867    0.8154       586

Área sob a curva (AUC): 84.61%
```

[111]:
```
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_xgb, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[112]:
```
# Ajustar o modelo aos dados de treinamento
xgb_model.fit(X_und, y_und)

# Converter y_test para valores numéricos
y_test_numeric = y_test.cat.codes

# Calcular as probabilidades de previsão
y_pred_proba = xgb_model.predict_proba(X_test)

# Calcular a pontuação ROC
fpr, tpr, _ = roc_curve(y_test_numeric, y_pred_proba[:, 1])

# Plotar a curva ROC
fig, ax = plt.subplots(figsize=(10,8))
ax.plot(fpr, tpr, label="XGBClassifier")
ax.plot([0, 1], [0, 1], color='gray', linestyle='--')
ax.set_xlabel("Taxa de Falso Positivo")
ax.set_ylabel("Taxa de Verdadeiro Positivo")
ax.set_title("Curva ROC para o XGBClassifier")
ax.legend()
plt.show()
```



[113]:
```
rl_model = LogisticRegression()
# Padronizando os dados de treino
scaler = StandardScaler().fit(X_und)
X_und_std = scaler.transform(X_und)

# Definindo o melhor parâmetro
param_rl = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# Identificando os melhor parâmetro
kfold = StratifiedKFold(n_splits=5, shuffle=True)

rl_clf = GridSearchCV(rl_model, param_rl, cv=kfold)
rl_clf.fit(X_und_std, y_und)

# Visualizar o melhor parâmetro
print("Melhor parâmetro: {}".format(rl_clf.best_params_))
```

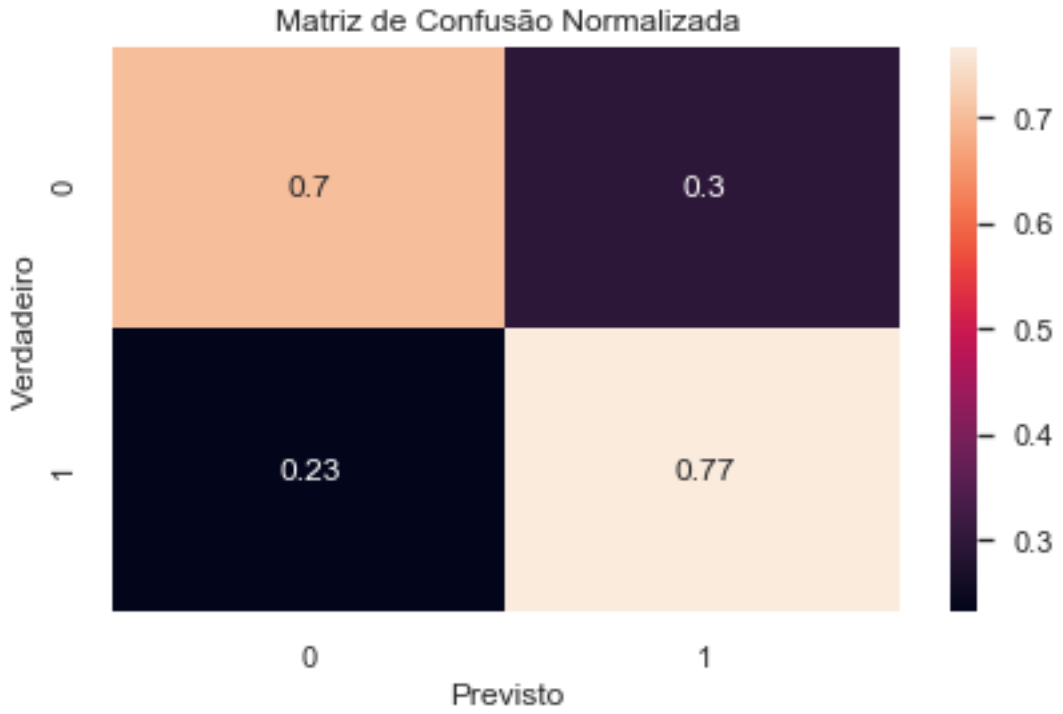
```
Melhor parâmetro: {'C': 10}
```

[114]:
```
rl_model = make_pipeline(StandardScaler(), LogisticRegression(C=10))

# Treinando o modelo
rl_model.fit(X_und, y_und)

# Fazendo previsões
y_pred_rl = rl_model.predict(X_test)
y_prob_rl = rl_model.predict_proba(X_test)
```

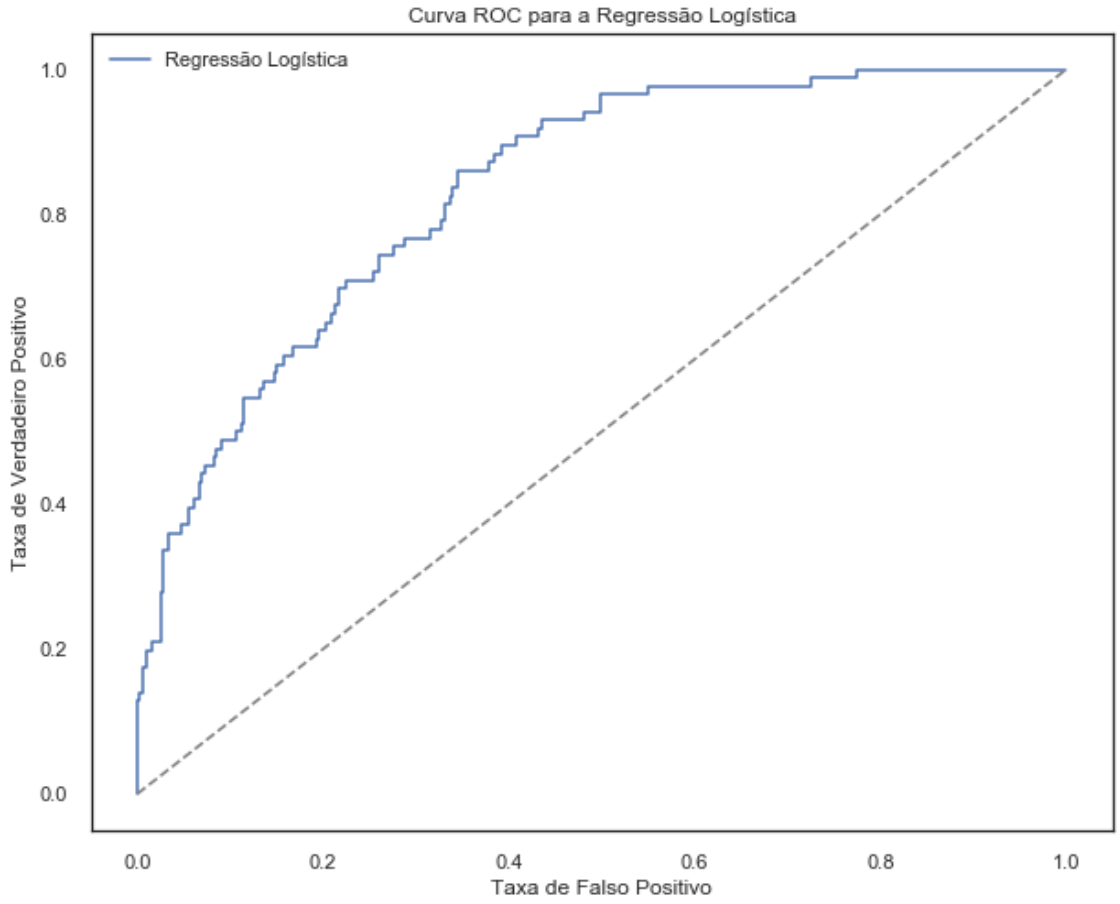
[185]:
```
# Relatório de classificação
print("Relatório de classificação para a Regressão Logística:\n",␣
 ↪classification_report(y_test, y_pred_rl, digits=4))

# Área sob a curva
print("Área sob a curva (AUC):\t{}%".format(round(roc_auc_score(y_test,␣
 ↪y_pred_rl) * 100, 2)))
```

```
Relatório de classificação para a Regressão Logística:
              precision    recall  f1-score   support

           0     0.9643    0.7020    0.8125       500
           1     0.3288    0.8488    0.4740        86

    accuracy                         0.7235       586
   macro avg     0.6466    0.7754    0.6433       586
weighted avg     0.8710    0.7235    0.7628       586

Área sob a curva (AUC): 77.54%
```

[115]:
```
# Matriz de confusão
fig, ax = plt.subplots()

sns.heatmap(confusion_matrix(y_test, y_pred_rl, normalize='true'), annot=True,␣
 ↪ax=ax)

ax.set_title('Matriz de Confusão Normalizada')
ax.set_ylabel('Verdadeiro')
ax.set_xlabel('Previsto')

plt.tight_layout()
```



[116]:
```
# Ajustar o modelo aos dados de treinamento
rl_model.fit(X_und, y_und)

# Converter y_test para valores numéricos
y_test_numeric = y_test.cat.codes

# Calcular as probabilidades de previsão
y_pred_proba = rl_model.predict_proba(X_test)

# Calcular a pontuação ROC
fpr, tpr, _ = roc_curve(y_test_numeric, y_pred_proba[:, 1])

# Plotar a curva ROC
fig, ax = plt.subplots(figsize=(10,8))
ax.plot(fpr, tpr, label="Regressão Logística")
ax.plot([0, 1], [0, 1], color='gray', linestyle='--')
ax.set_xlabel("Taxa de Falso Positivo")
ax.set_ylabel("Taxa de Verdadeiro Positivo")
ax.set_title("Curva ROC para a Regressão Logística")
ax.legend()
plt.show()
```



[129]:
```
# Dados para o primeiro gráfico
tecnicas = ["Random Forest", "XGBClassifiero", "Regressão Logísitca"]
acuracias_tinto = [88.42, 81.53, 76.13]

# Dados para o segundo gráfico
acuracias_branco = [84.91, 82.65, 77.54]

# Configuração das figuras e dos eixos
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Primeiro gráfico
ax1.bar(tecnicas, acuracias_tinto, color='red')
ax1.set_xlabel('Técnica aplicada para vinho Tinto')
ax1.set_ylabel('Acurácia')
ax1.set_title('Gráfico de Barras - Acurácia por Técnica')

# Segundo gráfico
ax2.bar(tecnicas, acuracias_branco, color='green')
```

```
ax2.set_xlabel('Técnica aplicada para vinho Branco')
ax2.set_ylabel('Acurácia')
ax2.set_title('Gráfico de Barras - Acurácia por Técnica')

# Ajusta o espaçamento entre as subplots
plt.tight_layout()

# Exibe os gráficos
plt.show()
```