Classificiação de vinhos brancos usando SMOTE, hot-encoding, mantendo a qualidade com valores de 3 à 9

In [160…

```
#Desabilita logs e mantém apenas logs críticos (para evitar o libcuda ficar me avisando qu
import logging
logger = logging.getLogger()
logger.setLevel(logging.CRITICAL)
```

In [161…

```
%config Completer.use_jedi = False
import pandas as pd
import numpy as np
import seaborn as sb
import matplotlib.pyplot as plt
import scipy as spy
import keras
from sklearn.metrics import accuracy_score, recall_score, confusion_matrix
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense,Dropout,Input
from keras.optimizers import Adam,RMSprop
```

In [162…

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
df = pd.read_csv('datasets/winequality-white.csv', sep = ',')
df.head()
```

Out[162…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6

Olhando abaixo, não temos nenhum valor N/A, então não precisamos tratar isso.

In [163…

```
df.isna().sum()
```

Out[163…

```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
```

```
quality                 0
dtype: int64
```

In [164…

```python
print(len(df))
```

```
4898
```

In [165…

```python
df['quality'].value_counts()
```

Out[165…

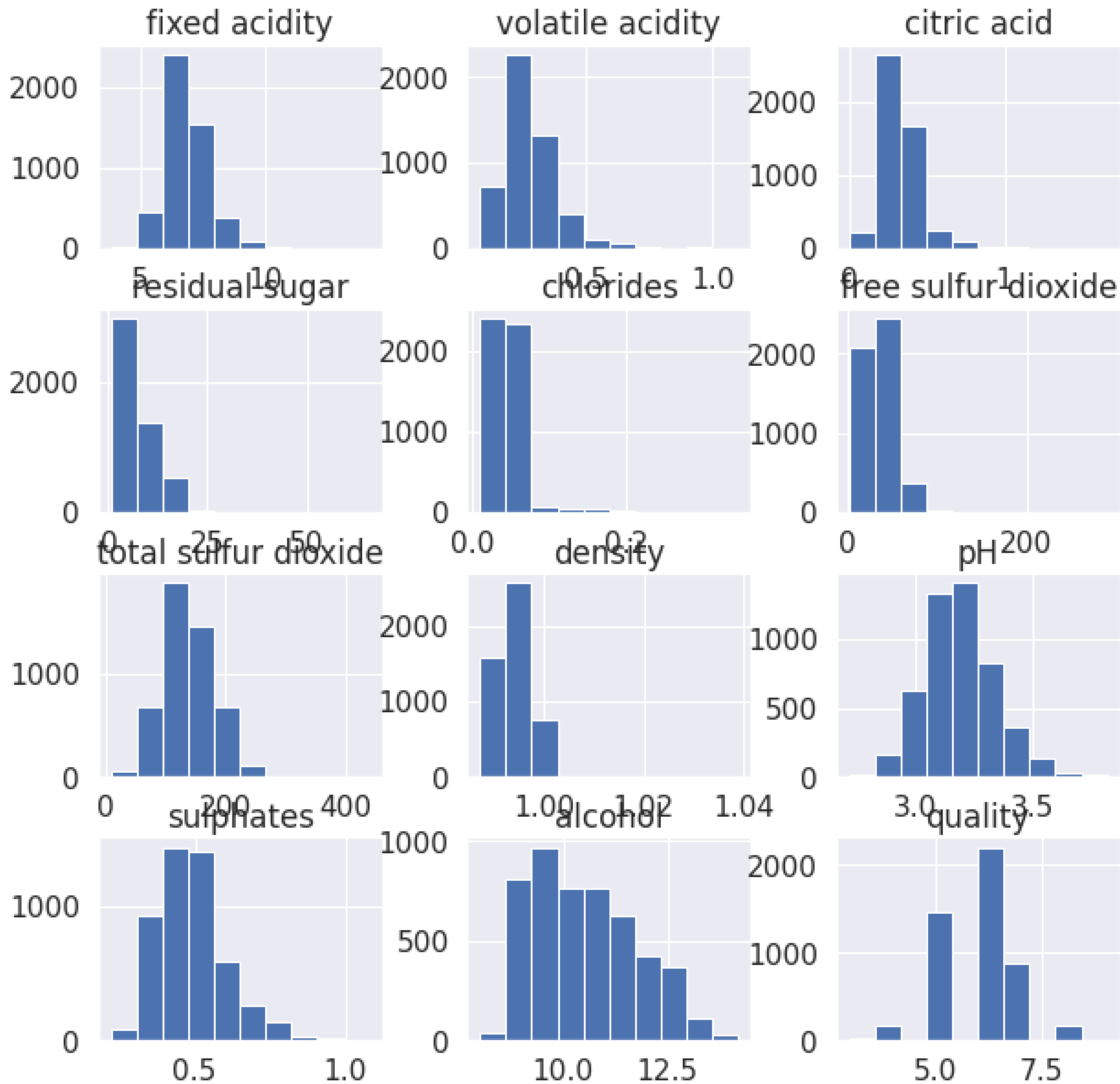
```
6    2198
5    1457
7     880
8     175
4     163
3      20
9       5
Name: quality, dtype: int64
```

In [166…

```python
df.hist(figsize = (10, 10))
```

Out[166…

```
array([[<AxesSubplot:title={'center':'fixed acidity'}>,
        <AxesSubplot:title={'center':'volatile acidity'}>,
        <AxesSubplot:title={'center':'citric acid'}>],
       [<AxesSubplot:title={'center':'residual sugar'}>,
        <AxesSubplot:title={'center':'chlorides'}>,
        <AxesSubplot:title={'center':'free sulfur dioxide'}>],
       [<AxesSubplot:title={'center':'total sulfur dioxide'}>,
        <AxesSubplot:title={'center':'density'}>,
        <AxesSubplot:title={'center':'pH'}>],
       [<AxesSubplot:title={'center':'sulphates'}>,
        <AxesSubplot:title={'center':'alcohol'}>,
        <AxesSubplot:title={'center':'quality'}>]], dtype=object)
```



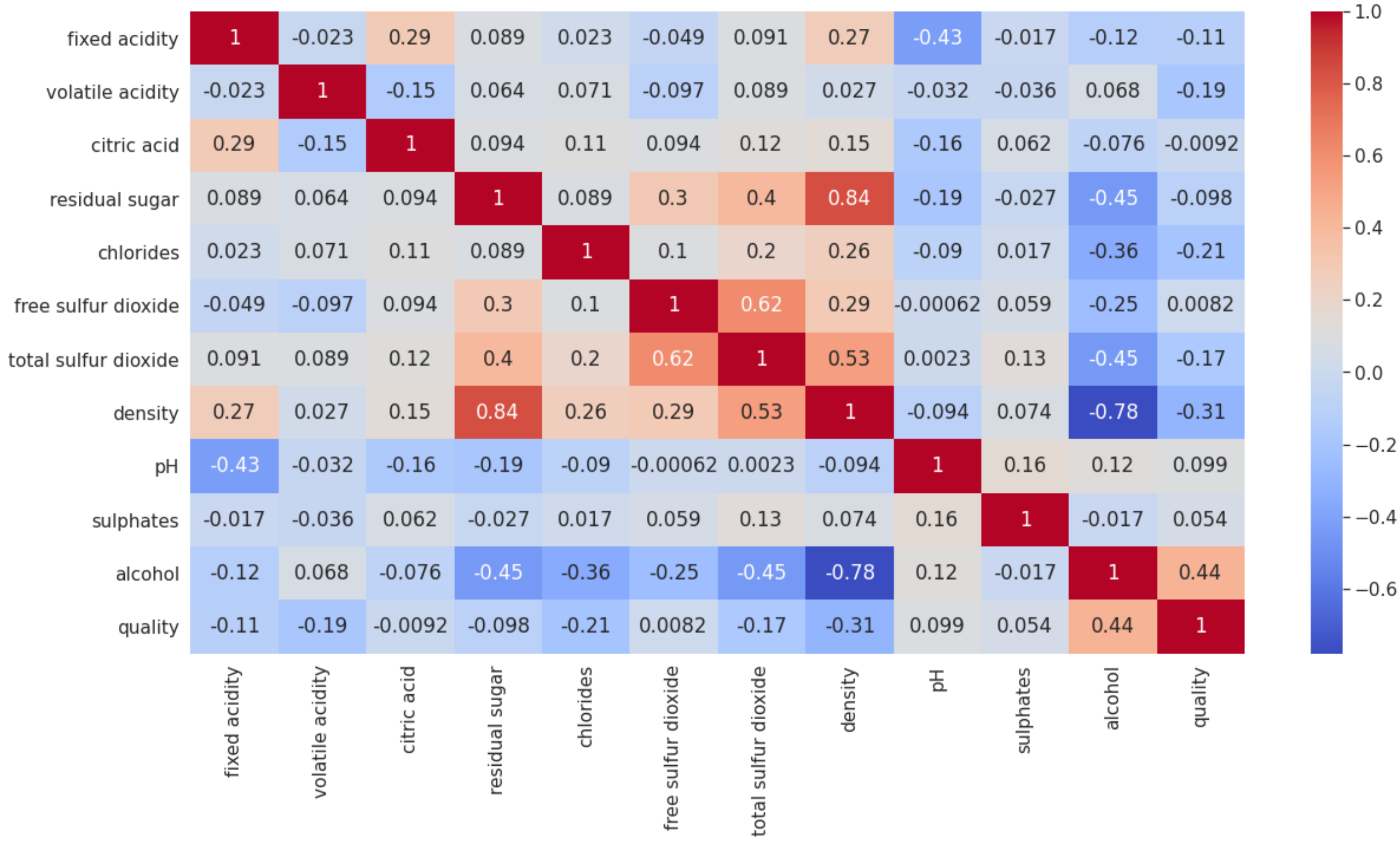
Matriz de correlação:

In [167…

```
corr=df.corr()
plt.figure(figsize=(20,10))
sb.heatmap(corr,annot=True, cmap='coolwarm')
```

Out[167…

```
<AxesSubplot:>
```



SMOTE

Podemos ver acima que temos um desbalanceamento na quantidade de amostras. Temos muitas regulares e poucas ruins e ótimas. Vamos usar uma técnica chamada SMOTE que consiste em fazer o oversampling das amostras minoritária}s, deixando assim o dataset balanceado. Essa técnica foi descrita no artigo.

In [168…

```
X=df.drop(columns=['quality'])
y=df['quality']
```

In [169…

```
from imblearn.over_sampling import SMOTE
oversample = SMOTE(k_neighbors=4)
X, y = oversample.fit_resample(X, y)
```

Vemos abaixo que as classificações agora estão igualmente distribuiídas:

In [170…

```
print(y.dtypes)
print(y.count())
y.value_counts()
```

Out[170…

```
int64
15386
6    2198
5    2198
7    2198
8    2198
4    2198
3    2198
9    2198
Name: quality, dtype: int64
```

In [171…

```
len(X)
```

Out[171… 15386

In [172…
```
len(y)
```

Out[172… 15386

Pronto, agora temos todas as amostras em quantias iguais.

Hot encoding

Ao contrário do outra versão deste notebook, não iremos mudar as classificações para 3 úinicas. Vamos usar todas disponíveis, que são as avaliações de qualidade numa escala de 3 à 9.

No trecho de código abaixo, vou converter as varíaveis categóricas (valores de 3 à 9) em uma tabela. Cada coluna dessa tabela

In [173…
```
df = pd.concat([X, y.reindex(X.index)], axis=1)
hot_df = pd.get_dummies(df, columns = ['quality'])
hot_df
```

Out[173…

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates
0	7.000000	0.270000	0.360000	20.700000	0.045000	45.000000	170.000000	1.001000	3.000000	0.450000
1	6.300000	0.300000	0.340000	1.600000	0.049000	14.000000	132.000000	0.994000	3.300000	0.490000
2	8.100000	0.280000	0.400000	6.900000	0.050000	30.000000	97.000000	0.995100	3.260000	0.440000
3	7.200000	0.230000	0.320000	8.500000	0.058000	47.000000	186.000000	0.995600	3.190000	0.400000
4	7.200000	0.230000	0.320000	8.500000	0.058000	47.000000	186.000000	0.995600	3.190000	0.400000
...	...	...	...	...	...	...	...	...	...	...
15381	6.614834	0.357033	0.295934	1.617801	0.021326	24.207675	85.830698	0.989669	3.408813	0.604363
15382	8.346797	0.256708	0.410125	6.789677	0.033228	27.556939	130.645912	0.994142	3.235445	0.468861
15383	7.174785	0.294052	0.350991	2.990946	0.025144	40.512894	129.991404	0.990212	3.280000	0.425948
15384	7.258153	0.261277	0.347588	1.929077	0.029227	26.468075	129.425354	0.990390	3.303050	0.503050
15385	8.320956	0.266105	0.465581	7.328013	0.033831	29.168567	119.715255	0.994390	3.266219	0.444419

15386 rows × 18 columns

In [174…
```
y = hot_df[['quality_3','quality_4', 'quality_5', 'quality_6', 'quality_7', 'quality_8', '
y
```

Out[174…

	quality_3	quality_4	quality_5	quality_6	quality_7	quality_8	quality_9
0	0	0	0	1	0	0	0
1	0	0	0	1	0	0	0
2	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...
15381	0	0	0	0	0	0	1

	quality_3	quality_4	quality_5	quality_6	quality_7	quality_8	quality_9
15382	0	0	0	0	0	0	1
15383	0	0	0	0	0	0	1
15384	0	0	0	0	0	0	1
15385	0	0	0	0	0	0	1

15386 rows × 7 columns

Separando o dataset de treinamento e o de predição

In [175…

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.20,random_state=21)
print('Formato do dataset de treinamento Xs:{}'.format(X_train.shape))
print('Formato do datasett de teste Xs:{}'.format(X_test.shape))
print('Formato do dataset de treino y:{}'.format(y_train.shape))
print('Formato do dataset de test y:{}'.format(y_test.shape))
```

```
Formato do dataset de treinamento Xs:(12308, 11)
Formato do datasett de teste Xs:(3078, 11)
Formato do dataset de treino y:(12308, 7)
Formato do dataset de test y:(3078, 7)
```

Construção do modelo

O artigo utilizou relu e tanh. Aqui abaixo vamos usar relu.

In [176…

```
dimension = X_train.shape[1]
from keras import backend as K
def create_model():
    model = Sequential()
    model.add(Dense(10, input_dim = dimension,  activation='relu'))
    model.add(Dense(60, input_dim = dimension,  activation='relu'))
    model.add(Dense(7, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
model = create_model()
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_9 (Dense)              (None, 10)                120
_________________________________________________________________
dense_10 (Dense)             (None, 60)                660
_________________________________________________________________
dense_11 (Dense)             (None, 7)                 427
=================================================================
Total params: 1,207
Trainable params: 1,207
Non-trainable params: 0
_________________________________________________________________
```

In [177…

```
history=model.fit(X_train, y_train, validation_data=(X_test, y_test),epochs=50, batch_size
```

```
Epoch 1/50
1231/1231 [==============================] - 1s 602us/step - loss: 1.7783 - accuracy: 0.26
```

```
33 - val_loss: 1.6731 - val_accuracy: 0.3067
Epoch 2/50
1231/1231 [==============================] - 1s 602us/step - loss: 1.5994 - accuracy: 0.35
51 - val_loss: 1.5404 - val_accuracy: 0.3882
Epoch 3/50
1231/1231 [==============================] - 1s 577us/step - loss: 1.5294 - accuracy: 0.38
58 - val_loss: 1.4869 - val_accuracy: 0.4100
Epoch 4/50
1231/1231 [==============================] - 1s 577us/step - loss: 1.4876 - accuracy: 0.40
12 - val_loss: 1.4506 - val_accuracy: 0.4068
Epoch 5/50
1231/1231 [==============================] - 1s 574us/step - loss: 1.4509 - accuracy: 0.41
59 - val_loss: 1.4136 - val_accuracy: 0.4240
Epoch 6/50
1231/1231 [==============================] - 1s 539us/step - loss: 1.4276 - accuracy: 0.42
22 - val_loss: 1.4042 - val_accuracy: 0.4198
Epoch 7/50
1231/1231 [==============================] - 1s 592us/step - loss: 1.4168 - accuracy: 0.42
18 - val_loss: 1.3869 - val_accuracy: 0.4298
Epoch 8/50
1231/1231 [==============================] - 1s 566us/step - loss: 1.4046 - accuracy: 0.42
66 - val_loss: 1.4432 - val_accuracy: 0.4025
Epoch 9/50
1231/1231 [==============================] - 1s 733us/step - loss: 1.3938 - accuracy: 0.42
70 - val_loss: 1.3770 - val_accuracy: 0.4295
Epoch 10/50
1231/1231 [==============================] - 1s 516us/step - loss: 1.3877 - accuracy: 0.43
14 - val_loss: 1.3472 - val_accuracy: 0.4353
Epoch 11/50
1231/1231 [==============================] - 1s 734us/step - loss: 1.3712 - accuracy: 0.43
68 - val_loss: 1.3680 - val_accuracy: 0.4298
Epoch 12/50
1231/1231 [==============================] - 1s 675us/step - loss: 1.3652 - accuracy: 0.43
58 - val_loss: 1.3558 - val_accuracy: 0.4344
Epoch 13/50
1231/1231 [==============================] - 1s 643us/step - loss: 1.3565 - accuracy: 0.43
52 - val_loss: 1.3188 - val_accuracy: 0.4594
Epoch 14/50
1231/1231 [==============================] - 1s 679us/step - loss: 1.3465 - accuracy: 0.43
66 - val_loss: 1.3478 - val_accuracy: 0.4425
Epoch 15/50
1231/1231 [==============================] - 1s 556us/step - loss: 1.3388 - accuracy: 0.44
46 - val_loss: 1.3141 - val_accuracy: 0.4636
Epoch 16/50
1231/1231 [==============================] - 1s 573us/step - loss: 1.3259 - accuracy: 0.44
82 - val_loss: 1.2899 - val_accuracy: 0.4701
Epoch 17/50
1231/1231 [==============================] - 1s 584us/step - loss: 1.3091 - accuracy: 0.45
25 - val_loss: 1.2870 - val_accuracy: 0.4672
Epoch 18/50
1231/1231 [==============================] - 1s 678us/step - loss: 1.2991 - accuracy: 0.46
01 - val_loss: 1.2836 - val_accuracy: 0.4708
Epoch 19/50
1231/1231 [==============================] - 1s 596us/step - loss: 1.2947 - accuracy: 0.46
23 - val_loss: 1.2839 - val_accuracy: 0.4565
Epoch 20/50
1231/1231 [==============================] - 1s 547us/step - loss: 1.2920 - accuracy: 0.46
29 - val_loss: 1.2512 - val_accuracy: 0.4753
Epoch 21/50
1231/1231 [==============================] - 1s 565us/step - loss: 1.2820 - accuracy: 0.46
41 - val_loss: 1.2543 - val_accuracy: 0.4789
Epoch 22/50
1231/1231 [==============================] - 1s 650us/step - loss: 1.2746 - accuracy: 0.46
78 - val_loss: 1.2537 - val_accuracy: 0.4841
Epoch 23/50
1231/1231 [==============================] - 1s 594us/step - loss: 1.2705 - accuracy: 0.47
```

```
07 - val_loss: 1.2557 - val_accuracy: 0.4789
Epoch 24/50
1231/1231 [==============================] - 1s 518us/step - loss: 1.2600 - accuracy: 0.47
60 - val_loss: 1.2455 - val_accuracy: 0.4773
Epoch 25/50
1231/1231 [==============================] - 1s 635us/step - loss: 1.2562 - accuracy: 0.47
64 - val_loss: 1.2761 - val_accuracy: 0.4877
Epoch 26/50
1231/1231 [==============================] - 1s 616us/step - loss: 1.2524 - accuracy: 0.48
00 - val_loss: 1.2843 - val_accuracy: 0.4571
Epoch 27/50
1231/1231 [==============================] - 1s 580us/step - loss: 1.2430 - accuracy: 0.48
07 - val_loss: 1.2455 - val_accuracy: 0.4867
Epoch 28/50
1231/1231 [==============================] - 1s 658us/step - loss: 1.2450 - accuracy: 0.48
52 - val_loss: 1.2162 - val_accuracy: 0.4847
Epoch 29/50
1231/1231 [==============================] - 1s 582us/step - loss: 1.2320 - accuracy: 0.48
87 - val_loss: 1.2111 - val_accuracy: 0.4987
Epoch 30/50
1231/1231 [==============================] - 1s 569us/step - loss: 1.2262 - accuracy: 0.49
42 - val_loss: 1.2057 - val_accuracy: 0.4925
Epoch 31/50
1231/1231 [==============================] - 1s 544us/step - loss: 1.2228 - accuracy: 0.49
35 - val_loss: 1.2128 - val_accuracy: 0.4883
Epoch 32/50
1231/1231 [==============================] - 1s 598us/step - loss: 1.2202 - accuracy: 0.49
71 - val_loss: 1.2482 - val_accuracy: 0.4747
Epoch 33/50
1231/1231 [==============================] - 1s 619us/step - loss: 1.2147 - accuracy: 0.50
13 - val_loss: 1.1964 - val_accuracy: 0.5003
Epoch 34/50
1231/1231 [==============================] - 1s 571us/step - loss: 1.2019 - accuracy: 0.50
61 - val_loss: 1.1813 - val_accuracy: 0.5068
Epoch 35/50
1231/1231 [==============================] - 1s 552us/step - loss: 1.1967 - accuracy: 0.50
63 - val_loss: 1.2278 - val_accuracy: 0.4948
Epoch 36/50
1231/1231 [==============================] - 1s 547us/step - loss: 1.1944 - accuracy: 0.51
01 - val_loss: 1.2417 - val_accuracy: 0.4873
Epoch 37/50
1231/1231 [==============================] - 1s 551us/step - loss: 1.1862 - accuracy: 0.51
15 - val_loss: 1.2220 - val_accuracy: 0.4877
Epoch 38/50
1231/1231 [==============================] - 1s 559us/step - loss: 1.1799 - accuracy: 0.51
83 - val_loss: 1.1794 - val_accuracy: 0.5104
Epoch 39/50
1231/1231 [==============================] - 1s 590us/step - loss: 1.1811 - accuracy: 0.51
55 - val_loss: 1.1787 - val_accuracy: 0.5130
Epoch 40/50
1231/1231 [==============================] - 1s 527us/step - loss: 1.1781 - accuracy: 0.51
18 - val_loss: 1.2199 - val_accuracy: 0.5065
Epoch 41/50
1231/1231 [==============================] - 1s 556us/step - loss: 1.1679 - accuracy: 0.51
95 - val_loss: 1.1691 - val_accuracy: 0.5117
Epoch 42/50
1231/1231 [==============================] - 1s 664us/step - loss: 1.1589 - accuracy: 0.52
19 - val_loss: 1.1545 - val_accuracy: 0.5182
Epoch 43/50
1231/1231 [==============================] - 1s 597us/step - loss: 1.1591 - accuracy: 0.52
22 - val_loss: 1.1661 - val_accuracy: 0.5166
Epoch 44/50
1231/1231 [==============================] - 1s 566us/step - loss: 1.1567 - accuracy: 0.51
73 - val_loss: 1.1520 - val_accuracy: 0.5234
Epoch 45/50
1231/1231 [==============================] - 1s 587us/step - loss: 1.1548 - accuracy: 0.52
```

```
53 - val_loss: 1.1453 - val_accuracy: 0.5162
Epoch 46/50
1231/1231 [==============================] - 1s 581us/step - loss: 1.1524 - accuracy: 0.52
55 - val_loss: 1.1374 - val_accuracy: 0.5351
Epoch 47/50
1231/1231 [==============================] - 1s 584us/step - loss: 1.1441 - accuracy: 0.53
04 - val_loss: 1.1540 - val_accuracy: 0.5224
Epoch 48/50
1231/1231 [==============================] - 1s 552us/step - loss: 1.1402 - accuracy: 0.53
28 - val_loss: 1.1337 - val_accuracy: 0.5374
Epoch 49/50
1231/1231 [==============================] - 1s 566us/step - loss: 1.1394 - accuracy: 0.52
81 - val_loss: 1.1767 - val_accuracy: 0.5104
Epoch 50/50
1231/1231 [==============================] - 1s 588us/step - loss: 1.1406 - accuracy: 0.52
94 - val_loss: 1.1277 - val_accuracy: 0.5286
```

Sobre val_accuracy e accuracy:

Quando ambos crescem na mesma proporção quer dizer que o modelo não causou nem overfitting nem underfitting.

Se o accuracy cresce mais que o val_accuracy, quer dizer que temos overfitting.

Se o accuracy cresce menos que o val_accuracy, quer dizer que temos underfitting.

avaliação do resultado:

In [178…

```
y_pred = model.predict(X_test)
def max_probs(array):
    parsed_pred = np.empty((0,7))
    for idx, x in enumerate(array):
        idx_max = x.argmax()
        x = np.zeros((7,))
        x[idx_max] = 1
        array[idx] = x

max_probs(y_pred)
y_pred
```

Out[178…

```
array([[0., 1., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

In [179…

```
def to_category(array):
    categories = []
    for idx, x in enumerate(array):
        idx_max = x.argmax()
        x = 0
        if idx_max == 0: x = 3
        if idx_max == 1: x = 4
        if idx_max == 2: x = 5
        if idx_max == 3: x = 6
        if idx_max == 4: x = 7
        if idx_max == 5: x = 8
        if idx_max == 6: x = 9
        categories.append(x)
    return categories
```

```
categorical_y_pred = to_category(y_pred)
categorical_y_test = to_category(y_test.to_numpy())
data = confusion_matrix(categorical_y_test, categorical_y_pred)
```

In [180…

```
len(categorical_y_test)
```

Out[180…

```
3078
```

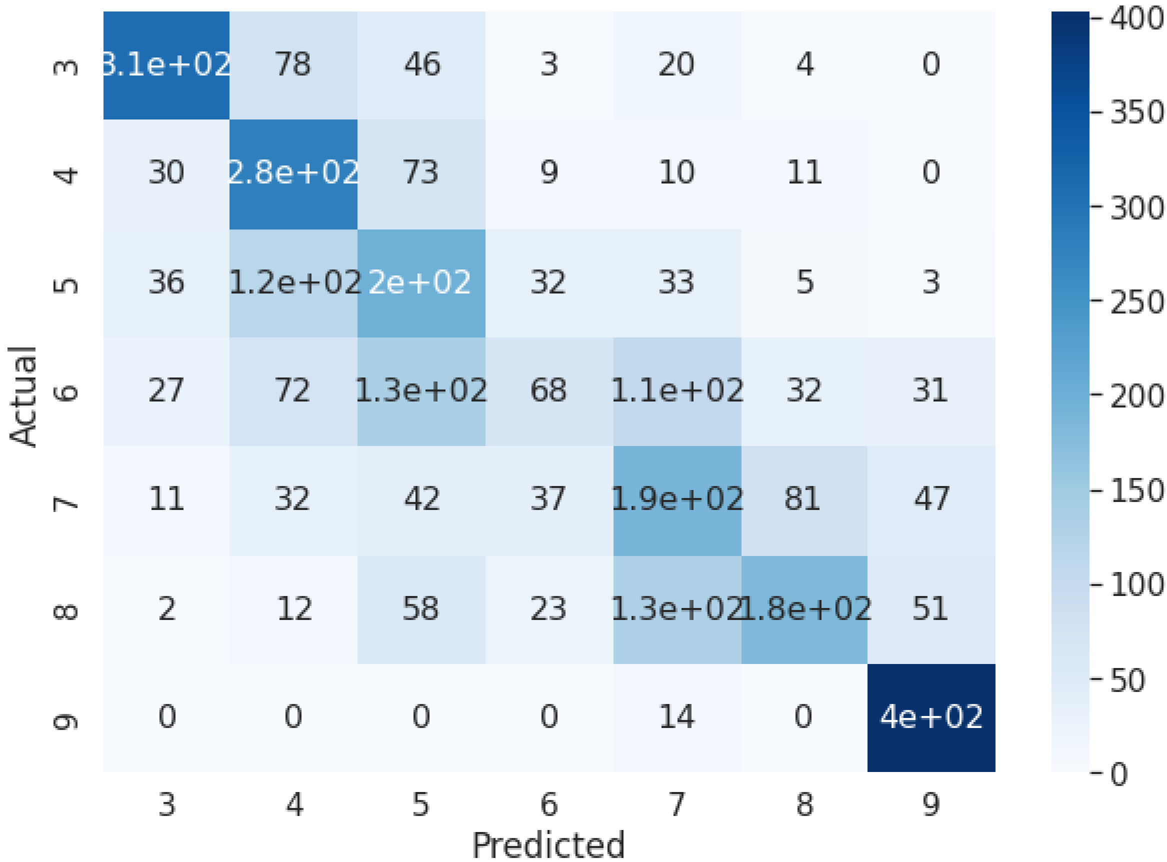
Matriz de confusão:

In [181…

```
df_cm = pd.DataFrame(data, columns=np.unique(categorical_y_test), index = np.unique(catego
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
sb.set(font_scale=1.4)#for label size
sb.heatmap(df_cm, cmap="Blues", annot=True,annot_kws={"size": 16})# font size
```

Out[181…

```
<AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



Verificação de acurácia geral

In [182…

```
correct = 0
total = 0
for i in range(len(categorical_y_test)):
    if(categorical_y_test[i] == categorical_y_pred[i]):
        correct += 1
    total += 1
accuracy = (correct/total)
```

In [183…

```
accuracy
```

Out[183…

```
0.5285899935022742
```

Conclusão

Usando SMOTE porém sem separar as categorias em 3 categorias de qualidade antes de fazer o treinamento, temos uma queda de 10% na capacidade de classificação. Isso era esperado, é mais fácil predizer o vinho quando não é mais necessária tanta exatidão no momento de prever a categoria, como foi feito na versão deste estudo redimensionando as categorias para alta média e baixa.