Classificiação de vinhos brancos usando SMOTE, hot-encoding e redimensionamento da qualidade para alta, média e baixa

In [55]:
```
#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 [56]:
```
%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 [57]:
```
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[57]:

	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 [58]:
```
df.isna().sum()
```

Out[58]:
```
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 [59]:
```
print(len(df))
```

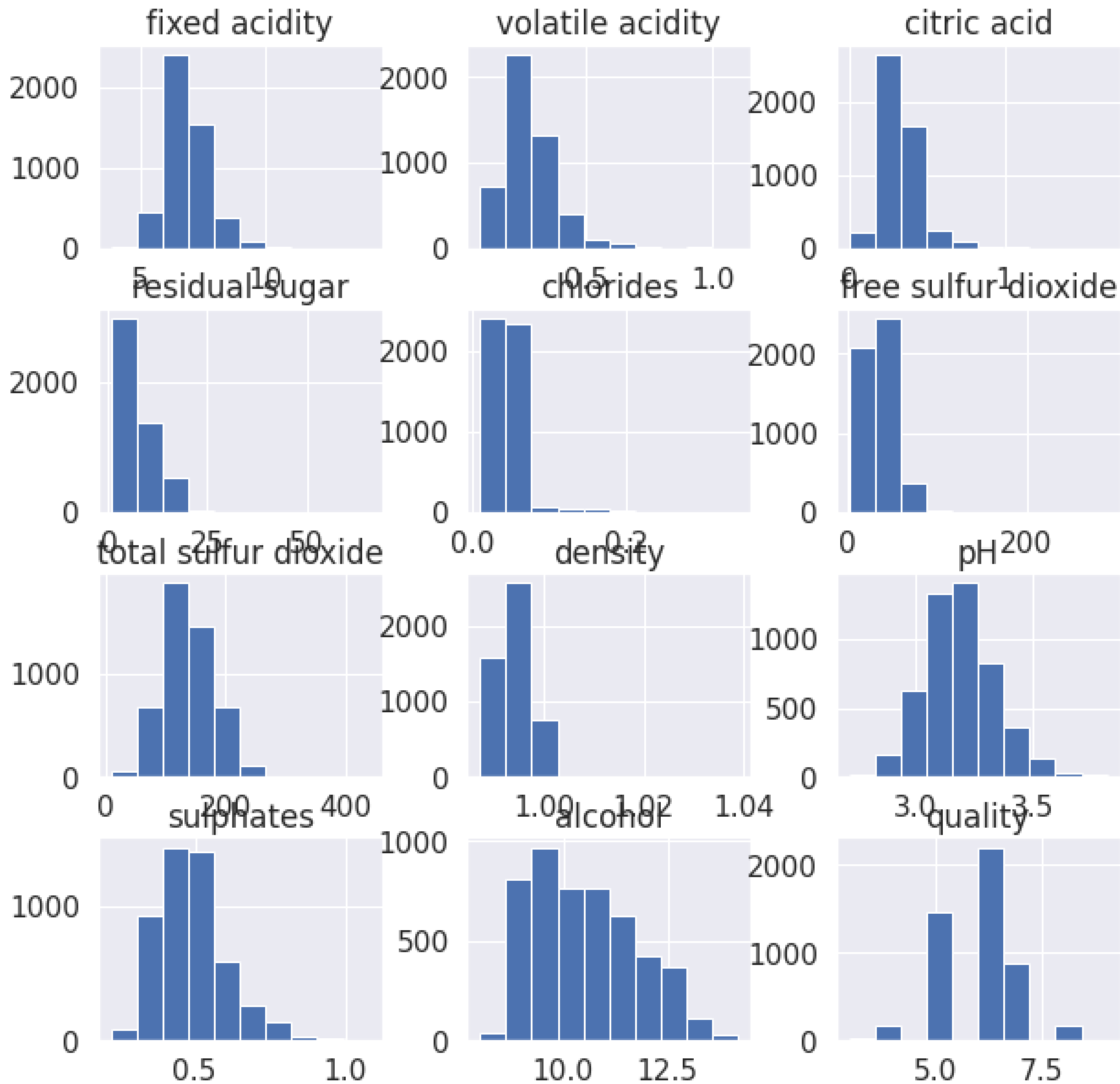
```
4898
```

In [60]:
```
df['quality'].value_counts()
```

Out[60]:
```
6    2198
5    1457
7     880
8     175
4     163
3      20
9       5
Name: quality, dtype: int64
```

In [61]:
```
df.hist(figsize = (10, 10))
```

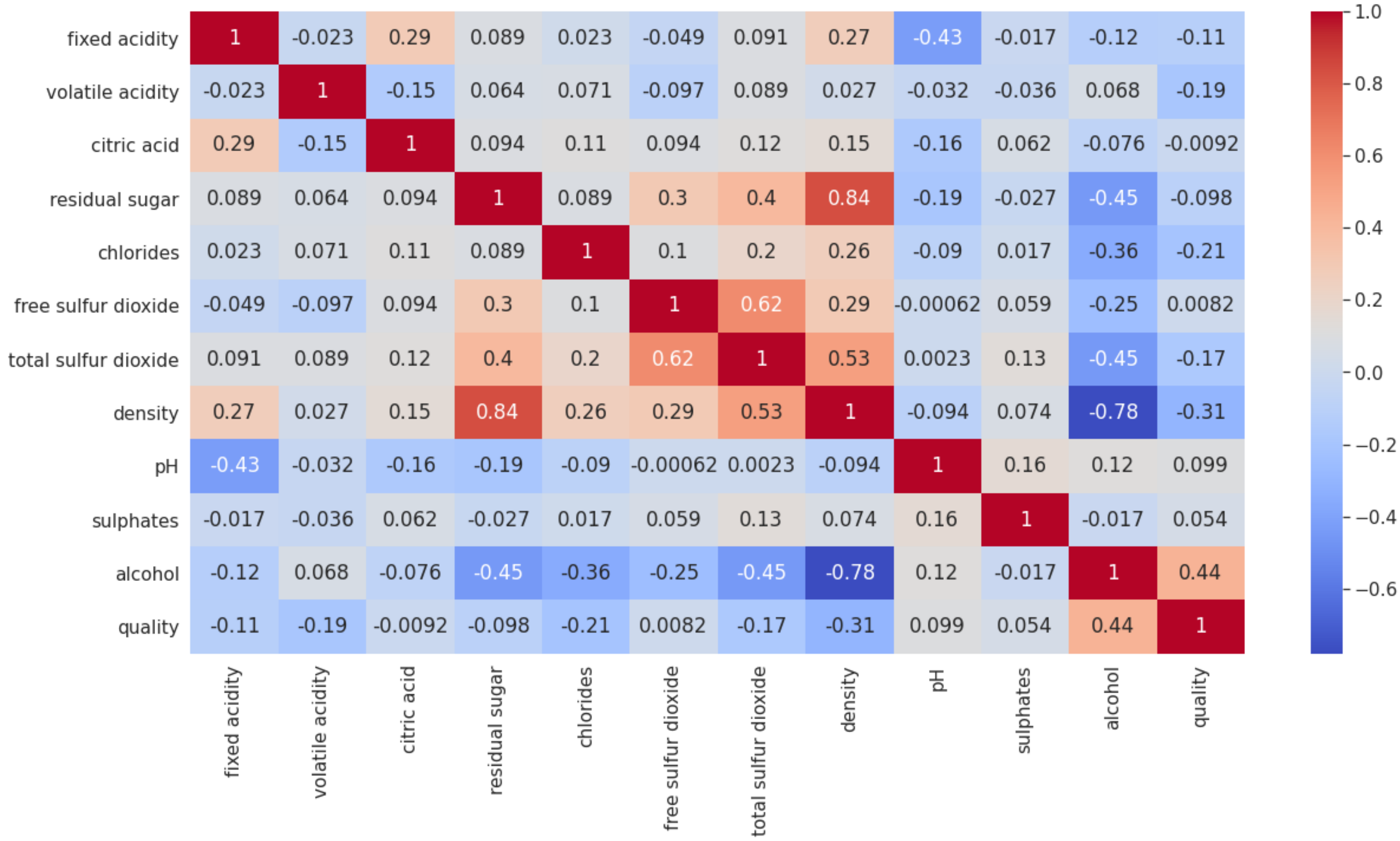
Out[61]:
```
array([[<AxesSubplot:title={'center':'fixed acidity'}>,
        <AxesSubplot:title={'center':'volatile acidity'}>,
        <AxesSubplot:title={'center':'citric acid'}>],
       [<AxesSubplot:title={'center':'residual sugar'}>,
        <AxesSubplot:title={'center':'chlorides'}>,
        <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 [62]:
```
corr=df.corr()
plt.figure(figsize=(20,10))
sb.heatmap(corr,annot=True, cmap='coolwarm')
```

Out[62]:
```
<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 [63]:
```
X=df.drop(columns=['quality'])
y=df['quality']
```

In [64]:
```
y
```

Out[64]:
```
0       6
1       6
2       6
3       6
4       6
       ..
4893    6
4894    5
4895    6
4896    7
4897    6
Name: quality, Length: 4898, dtype: int64
```

In [65]:
```
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 [66]:
```
print(y.dtypes)
print(y.count())
```

```
y.value_counts()
```

Out[66]:

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

In [67]:

```
X
```

Out[67]:

	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	7.095248	0.313141	0.347810	3.340911	0.023076	45.285144	126.809904	0.990093	3.280000	0.406859
15382	8.398567	0.257622	0.412865	7.051574	0.033350	27.587392	130.189115	0.994339	3.233009	0.468252
15383	7.222164	0.251856	0.437062	2.118558	0.031593	29.371150	123.587523	0.990402	3.333351	0.444433
15384	8.525219	0.290692	0.413214	8.530789	0.031781	27.080351	115.033419	0.995310	3.248282	0.494487
15385	8.572814	0.267364	0.460544	8.385819	0.034209	28.790779	121.100477	0.995234	3.244811	0.449456

15386 rows × 11 columns

Pronto, agora temos todas as amostras em quantias iguais.

Hot encoding

Primeiro vamos separar a qualificações que vão de 6 à 9 (no caso desse dataset) em 0 1 e 2. Sendo 0 a qualidade mais baixa, 1 a média e 2 a alta, como vamos configurar para o hot encoding posteriormente.

In [68]:

```
y[y<=4] = 0
y[((y>=5) & (y<=7))] = 1
y[y>=8] = 2
y.value_counts()
```

Out[68]:

```
1    6594
2    4396
0    4396
Name: quality, dtype: int64
```

In [69]:

```
df = pd.concat([X, y.reindex(X.index)], axis=1)
df
```

Out[69]:

	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	7.095248	0.313141	0.347810	3.340911	0.023076	45.285144	126.809904	0.990093	3.280000	0.406859
15382	8.398567	0.257622	0.412865	7.051574	0.033350	27.587392	130.189115	0.994339	3.233009	0.468252
15383	7.222164	0.251856	0.437062	2.118558	0.031593	29.371150	123.587523	0.990402	3.333351	0.444433
15384	8.525219	0.290692	0.413214	8.530789	0.031781	27.080351	115.033419	0.995310	3.248282	0.494487
15385	8.572814	0.267364	0.460544	8.385819	0.034209	28.790779	121.100477	0.995234	3.244811	0.449456

15386 rows × 12 columns

No trecho de código abaixo, vou converter as varíaveis categóricas (0 1 e 2) em uma tabela. Essa tabela tem 3 colunas, onde cada uma corresponde à uma das classificações possíveis. Sempre apenas um dos items dessa coluna vai ser 1 e o resto 0. O nosso modelo ira tentar prever o valor dessas 3 colunas para assim prever a qualidade do vinho.

Substituindo as faixas de qualidade por 0 (baixa), 1 (média) e 2 (alta):

In [70]:

```
one_hot_encoded_data = pd.get_dummies(df, columns = ['quality'])
df = one_hot_encoded_data.rename(columns={'quality_0': 'baixa', 'quality_1': 'media', 'qua
df.head()
```

Out[70]:

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

In [71]:

```
y = df[['baixa','media', 'alta']]
```

Separando o dataset de treinamento e o de predição

In [72]:

```
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, 3)
Formato do dataset de test y:(3078, 3)
```

Construção do modelo

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

In [73]:

```
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(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
model = create_model()
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 10)                120
_________________________________________________________________
dense_4 (Dense)              (None, 60)                660
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 183
=================================================================
Total params: 963
Trainable params: 963
Non-trainable params: 0
_________________________________________________________________
```

In [74]:

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

```
Epoch 1/50
1231/1231 [==============================] - 1s 619us/step - loss: 1.0091 - accuracy: 0.50
93 - val_loss: 0.8258 - val_accuracy: 0.5968
Epoch 2/50
1231/1231 [==============================] - 1s 586us/step - loss: 0.8566 - accuracy: 0.57
69 - val_loss: 0.8015 - val_accuracy: 0.6244
Epoch 3/50
1231/1231 [==============================] - 1s 628us/step - loss: 0.8373 - accuracy: 0.59
58 - val_loss: 0.8040 - val_accuracy: 0.6173
Epoch 4/50
1231/1231 [==============================] - 1s 544us/step - loss: 0.8108 - accuracy: 0.60
92 - val_loss: 0.7496 - val_accuracy: 0.6550
Epoch 5/50
1231/1231 [==============================] - 1s 536us/step - loss: 0.7856 - accuracy: 0.62
56 - val_loss: 0.8489 - val_accuracy: 0.5884
Epoch 6/50
1231/1231 [==============================] - 1s 545us/step - loss: 0.7791 - accuracy: 0.63
45 - val_loss: 0.7965 - val_accuracy: 0.6238
Epoch 7/50
1231/1231 [==============================] - 1s 581us/step - loss: 0.7561 - accuracy: 0.64
84 - val_loss: 0.7756 - val_accuracy: 0.6274
Epoch 8/50
1231/1231 [==============================] - 1s 566us/step - loss: 0.7428 - accuracy: 0.65
71 - val_loss: 0.7102 - val_accuracy: 0.6839
Epoch 9/50
1231/1231 [==============================] - 1s 572us/step - loss: 0.7241 - accuracy: 0.66
```

```
80 - val_loss: 0.6987 - val_accuracy: 0.6940
Epoch 10/50
1231/1231 [==============================] - 1s 568us/step - loss: 0.7029 - accuracy: 0.68
04 - val_loss: 0.6955 - val_accuracy: 0.6992
Epoch 11/50
1231/1231 [==============================] - 1s 551us/step - loss: 0.6884 - accuracy: 0.68
49 - val_loss: 0.6768 - val_accuracy: 0.6946
Epoch 12/50
1231/1231 [==============================] - 1s 545us/step - loss: 0.6798 - accuracy: 0.68
86 - val_loss: 0.6404 - val_accuracy: 0.7212
Epoch 13/50
1231/1231 [==============================] - 1s 574us/step - loss: 0.6698 - accuracy: 0.69
12 - val_loss: 0.7046 - val_accuracy: 0.7031
Epoch 14/50
1231/1231 [==============================] - 1s 562us/step - loss: 0.6578 - accuracy: 0.69
35 - val_loss: 0.6279 - val_accuracy: 0.7248
Epoch 15/50
1231/1231 [==============================] - 1s 572us/step - loss: 0.6553 - accuracy: 0.69
43 - val_loss: 0.7205 - val_accuracy: 0.6540
Epoch 16/50
1231/1231 [==============================] - 1s 557us/step - loss: 0.6474 - accuracy: 0.70
12 - val_loss: 0.6654 - val_accuracy: 0.6904
Epoch 17/50
1231/1231 [==============================] - 1s 586us/step - loss: 0.6423 - accuracy: 0.70
31 - val_loss: 0.6226 - val_accuracy: 0.7186
Epoch 18/50
1231/1231 [==============================] - 1s 559us/step - loss: 0.6339 - accuracy: 0.70
40 - val_loss: 0.6316 - val_accuracy: 0.7034
Epoch 19/50
1231/1231 [==============================] - 1s 545us/step - loss: 0.6270 - accuracy: 0.71
08 - val_loss: 0.6428 - val_accuracy: 0.7151
Epoch 20/50
1231/1231 [==============================] - 1s 546us/step - loss: 0.6254 - accuracy: 0.71
25 - val_loss: 0.6215 - val_accuracy: 0.7190
Epoch 21/50
1231/1231 [==============================] - 1s 559us/step - loss: 0.6189 - accuracy: 0.71
38 - val_loss: 0.6126 - val_accuracy: 0.7417
Epoch 22/50
1231/1231 [==============================] - 1s 587us/step - loss: 0.6180 - accuracy: 0.71
21 - val_loss: 0.6020 - val_accuracy: 0.7349
Epoch 23/50
1231/1231 [==============================] - 1s 526us/step - loss: 0.6167 - accuracy: 0.71
10 - val_loss: 0.6107 - val_accuracy: 0.7316
Epoch 24/50
1231/1231 [==============================] - 1s 537us/step - loss: 0.6089 - accuracy: 0.71
28 - val_loss: 0.5924 - val_accuracy: 0.7326
Epoch 25/50
1231/1231 [==============================] - 1s 579us/step - loss: 0.6021 - accuracy: 0.71
91 - val_loss: 0.6038 - val_accuracy: 0.7401
Epoch 26/50
1231/1231 [==============================] - 1s 598us/step - loss: 0.5994 - accuracy: 0.72
47 - val_loss: 0.5885 - val_accuracy: 0.7310
Epoch 27/50
1231/1231 [==============================] - 1s 568us/step - loss: 0.5932 - accuracy: 0.72
19 - val_loss: 0.5896 - val_accuracy: 0.7238
Epoch 28/50
1231/1231 [==============================] - 1s 561us/step - loss: 0.5839 - accuracy: 0.73
14 - val_loss: 0.5959 - val_accuracy: 0.7355
Epoch 29/50
1231/1231 [==============================] - 1s 551us/step - loss: 0.5848 - accuracy: 0.72
99 - val_loss: 0.5706 - val_accuracy: 0.7404
Epoch 30/50
1231/1231 [==============================] - 1s 535us/step - loss: 0.5821 - accuracy: 0.73
55 - val_loss: 0.5724 - val_accuracy: 0.7394
Epoch 31/50
1231/1231 [==============================] - 1s 564us/step - loss: 0.5752 - accuracy: 0.73
```

```
59 - val_loss: 0.6289 - val_accuracy: 0.7128
Epoch 32/50
1231/1231 [==============================] - 1s 561us/step - loss: 0.5709 - accuracy: 0.74
00 - val_loss: 0.5563 - val_accuracy: 0.7602
Epoch 33/50
1231/1231 [==============================] - 1s 551us/step - loss: 0.5663 - accuracy: 0.74
37 - val_loss: 0.5951 - val_accuracy: 0.7359
Epoch 34/50
1231/1231 [==============================] - 1s 574us/step - loss: 0.5681 - accuracy: 0.73
88 - val_loss: 0.5676 - val_accuracy: 0.7398
Epoch 35/50
1231/1231 [==============================] - 1s 557us/step - loss: 0.5679 - accuracy: 0.73
79 - val_loss: 0.6705 - val_accuracy: 0.6966
Epoch 36/50
1231/1231 [==============================] - 1s 576us/step - loss: 0.5614 - accuracy: 0.74
18 - val_loss: 0.6189 - val_accuracy: 0.7144
Epoch 37/50
1231/1231 [==============================] - 1s 564us/step - loss: 0.5596 - accuracy: 0.74
32 - val_loss: 0.5565 - val_accuracy: 0.7524
Epoch 38/50
1231/1231 [==============================] - 1s 565us/step - loss: 0.5533 - accuracy: 0.74
55 - val_loss: 0.5602 - val_accuracy: 0.7476
Epoch 39/50
1231/1231 [==============================] - 1s 566us/step - loss: 0.5565 - accuracy: 0.74
76 - val_loss: 0.5720 - val_accuracy: 0.7593
Epoch 40/50
1231/1231 [==============================] - 1s 550us/step - loss: 0.5530 - accuracy: 0.74
53 - val_loss: 0.6257 - val_accuracy: 0.7212
Epoch 41/50
1231/1231 [==============================] - 1s 607us/step - loss: 0.5488 - accuracy: 0.74
85 - val_loss: 0.5444 - val_accuracy: 0.7729
Epoch 42/50
1231/1231 [==============================] - 1s 586us/step - loss: 0.5519 - accuracy: 0.74
82 - val_loss: 0.5550 - val_accuracy: 0.7446
Epoch 43/50
1231/1231 [==============================] - 1s 564us/step - loss: 0.5431 - accuracy: 0.75
30 - val_loss: 0.5515 - val_accuracy: 0.7710
Epoch 44/50
1231/1231 [==============================] - 1s 549us/step - loss: 0.5437 - accuracy: 0.75
07 - val_loss: 0.5446 - val_accuracy: 0.7615
Epoch 45/50
1231/1231 [==============================] - 1s 542us/step - loss: 0.5376 - accuracy: 0.75
38 - val_loss: 0.5665 - val_accuracy: 0.7641
Epoch 46/50
1231/1231 [==============================] - 1s 551us/step - loss: 0.5384 - accuracy: 0.75
65 - val_loss: 0.6486 - val_accuracy: 0.6998
Epoch 47/50
1231/1231 [==============================] - 1s 547us/step - loss: 0.5358 - accuracy: 0.75
92 - val_loss: 0.5532 - val_accuracy: 0.7596
Epoch 48/50
1231/1231 [==============================] - 1s 586us/step - loss: 0.5312 - accuracy: 0.76
12 - val_loss: 0.5380 - val_accuracy: 0.7693
Epoch 49/50
1231/1231 [==============================] - 1s 558us/step - loss: 0.5309 - accuracy: 0.75
75 - val_loss: 0.5874 - val_accuracy: 0.7485
Epoch 50/50
1231/1231 [==============================] - 1s 544us/step - loss: 0.5296 - accuracy: 0.76
22 - val_loss: 0.5876 - val_accuracy: 0.7544
```

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 [75]:
```
y_pred = model.predict(X_test)
def max_probs(array):
    parsed_pred = np.empty((0,3))
    for idx, x in enumerate(array):
        idx_max = x.argmax()
        x = np.zeros((3,))
        x[idx_max] = 1
        array[idx] = x

max_probs(y_pred)
```

In [76]:
```
def to_category(array):
    categories = []
    for idx, x in enumerate(array):
        idx_max = x.argmax()
        x = 0
        if idx_max == 0: x = 0
        if idx_max == 1: x = 1
        if idx_max == 2: x = 2
        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)
```

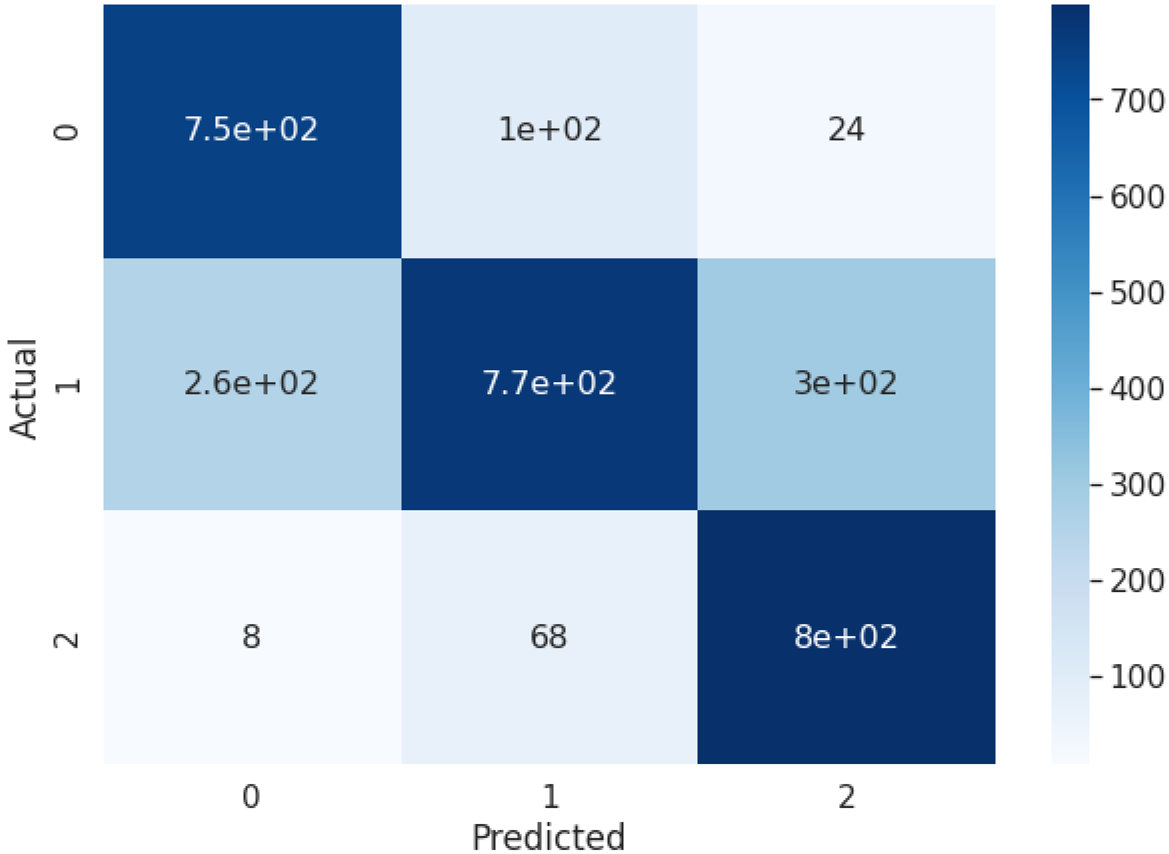
In [77]:
```
len(categorical_y_test)
```

Out[77]:
```
3078
```

Matriz de confusão:

In [78]:
```
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[78]:
```
<AxesSubplot:xlabel='Predicted', ylabel='Actual'>
```



Verificação de acurácia geral

In [79]:
```
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 [80]:
```
accuracy
```

Out[80]:
```
0.7543859649122807
```

Conclusão

Usando a divisão em 3 categorias e SMOTE, o modelo tem uma acurácia mais elevada. Isso acontece porque a classificação nessas 3 categorias apenas que definimios, requer menos exatidão no valor a ser previsto, então facilita para o MLP fazer uma previsão correta.