# project_3

October 8, 2024

**Instituto Tecnológico de Aeronáutica - ITA**

**Divisão de Engenharia Eletrônica - IEE**

**ET-287 - Processamento de sinais usando redes neurais**

**Professora Sarah Negreiros de Carvalho Leite**

**Aluno Felipe Keller Baltor**

# 1 Projeto 3

#### 1.0.1 *1. Baixe a base de dados, disponível em: https://www.kaggle.com/datasets/uciml/faulty-steel-plates?resource=download.*

[1]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])?  y
```

[2]:
```
# Importações básicas

import numpy as np # biblioteca de manipulação vetorial e numérica
import matplotlib.pyplot as plt # biblioteca para traçar gráficos
import pandas as pd # biblioteca de manipulação de dados tabulares
from pathlib import Path # biblioteca para manipulação de "paths"
import urllib3 # biblioteca para download do dataset
from tqdm import tqdm # barra de download
import zipfile
import scipy
import polars as pl
```

[3]:
```
# Checando se estamos no diretório correto

project_dir = Path('.')

assert project_dir.resolve().name == 'project_3'
```

[4]:
```
# Baixando os dados

data_dir = project_dir / 'data'
data_file = data_dir / 'faults.csv'

if not data_file.is_file():
    data_dir.mkdir(exist_ok = True)

    data_compressed = project_dir / 'data.zip'
    if not data_compressed.is_file():
        data_url = "https://storage.googleapis.com/kaggle-data-sets/2363/3972/
↪bundle/archive.zip?
↪X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com%40kaggle-161607.
↪iam.gserviceaccount.
↪com%2F20241002%2Fauto%2Fstorage%2Fgoog4_request&X-Goog-Date=20241002T200249Z&X-Goog-Expires

        http = urllib3.PoolManager()
        CHUNK_SIZE = 2**16

        resp = http.request('GET', data_url, preload_content = False)
        TOTAL_SIZE = int(resp.headers.get('Content-Length'))
        with (
            open(data_compressed, 'wb') as file,
            tqdm(
                total = TOTAL_SIZE,
                desc = f'Downloading {data_compressed.name}',
                unit = 'B',
                unit_scale = True) as bar
            ):
                for chunck in resp.stream(CHUNK_SIZE):
                    size = file.write(chunck)
                    bar.update(size)
        resp.release_conn()

    print("Extracting zip to data folder")
    with zipfile.ZipFile(data_compressed, 'r') as zip_ref:
        zip_ref.extractall(data_dir)
    print("Data is ready!")

else:
    print("Data already exists")
```

```
Data already exists
```

[5]:
```
faults = pl.read_csv(data_file)
```

[6]:
```
faults.columns
```

[6]:
```
['X_Minimum',
 'X_Maximum',
 'Y_Minimum',
 'Y_Maximum',
 'Pixels_Areas',
 'X_Perimeter',
 'Y_Perimeter',
 'Sum_of_Luminosity',
 'Minimum_of_Luminosity',
 'Maximum_of_Luminosity',
 'Length_of_Conveyer',
 'TypeOfSteel_A300',
 'TypeOfSteel_A400',
 'Steel_Plate_Thickness',
 'Edges_Index',
 'Empty_Index',
 'Square_Index',
 'Outside_X_Index',
 'Edges_X_Index',
 'Edges_Y_Index',
 'Outside_Global_Index',
 'LogOfAreas',
 'Log_X_Index',
 'Log_Y_Index',
 'Orientation_Index',
 'Luminosity_Index',
 'SigmoidOfAreas',
 'Pastry',
 'Z_Scratch',
 'K_Scatch',
 'Stains',
 'Dirtiness',
 'Bumps',
 'Other_Faults']
```

Existe um erro de ortografia na coluna *K_Scatch*, que deveria ser *K_Scratch*, vamos corrigir isso:

[7]:
```python
faults = faults.rename({"K_Scatch": "K_Scratch"}, strict = False)

faults
```

[7]:
```
shape: (1_941, 34)


 X_Minimum   X_Maximum   Y_Minimum   Y_Maximum   …   Stains   Dirtiness   Bumps
 Other_Faults
 ---         ---         ---         ---             ---      ---         ---
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| --- | | | | | | | | |
| i64 | i64 | i64 | i64 | | i64 | i64 | i64 | i64 |
| 42 | 50 | 270900 | 270944 | … | 0 | 0 | 0 | 0 |
| 645 | 651 | 2538079 | 2538108 | … | 0 | 0 | 0 | 0 |
| 829 | 835 | 1553913 | 1553931 | … | 0 | 0 | 0 | 0 |
| 853 | 860 | 369370 | 369415 | … | 0 | 0 | 0 | 0 |
| 1289 | 1306 | 498078 | 498335 | … | 0 | 0 | 0 | 0 |
| … | … | … | … | … | … | … | … | … |
| 249 | 277 | 325780 | 325796 | … | 0 | 0 | 0 | 1 |
| 144 | 175 | 340581 | 340598 | … | 0 | 0 | 0 | 1 |
| 145 | 174 | 386779 | 386794 | … | 0 | 0 | 0 | 1 |
| 137 | 170 | 422497 | 422528 | … | 0 | 0 | 0 | 1 |
| 1261 | 1281 | 87951 | 87967 | … | 0 | 0 | 0 | 1 |

Printando os nomes das colunas, temos:

[8]:
```python
for index, column in enumerate(faults.columns):
    print(f"Index: {index + 1}, Column Name: {column}")
```

```
Index: 1, Column Name: X_Minimum
Index: 2, Column Name: X_Maximum
Index: 3, Column Name: Y_Minimum
Index: 4, Column Name: Y_Maximum
Index: 5, Column Name: Pixels_Areas
Index: 6, Column Name: X_Perimeter
Index: 7, Column Name: Y_Perimeter
Index: 8, Column Name: Sum_of_Luminosity
Index: 9, Column Name: Minimum_of_Luminosity
Index: 10, Column Name: Maximum_of_Luminosity
Index: 11, Column Name: Length_of_Conveyer
Index: 12, Column Name: TypeOfSteel_A300
Index: 13, Column Name: TypeOfSteel_A400
```

```
Index: 14, Column Name: Steel_Plate_Thickness
Index: 15, Column Name: Edges_Index
Index: 16, Column Name: Empty_Index
Index: 17, Column Name: Square_Index
Index: 18, Column Name: Outside_X_Index
Index: 19, Column Name: Edges_X_Index
Index: 20, Column Name: Edges_Y_Index
Index: 21, Column Name: Outside_Global_Index
Index: 22, Column Name: LogOfAreas
Index: 23, Column Name: Log_X_Index
Index: 24, Column Name: Log_Y_Index
Index: 25, Column Name: Orientation_Index
Index: 26, Column Name: Luminosity_Index
Index: 27, Column Name: SigmoidOfAreas
Index: 28, Column Name: Pastry
Index: 29, Column Name: Z_Scratch
Index: 30, Column Name: K_Scratch
Index: 31, Column Name: Stains
Index: 32, Column Name: Dirtiness
Index: 33, Column Name: Bumps
Index: 34, Column Name: Other_Faults
```

Dessas, as nossas *features* são as 27 primeiras e os *labels* são as restantes:

[9]:
```
features = faults.columns[:27]
labels = faults.columns[27:]

X = faults[features]
Y = faults[labels]
```

### 1.0.2 *2. Faça uma análise exploratória dos dados.*

[10]:
```
X.describe()
```

[10]:
```
shape: (9, 28)


  statistic   X_Minimum   X_Maximum   Y_Minimum   …  Log_Y_Ind   Orientati
Luminosit   SigmoidO
  ---         ---         ---         ---            ex          on_Index
y_Index     fAreas
  str         f64         f64         f64            ---         ---
---         ---
                                                     f64         f64
f64         f64


  count       1941.0      1941.0      1941.0      …  1941.0      1941.0
```

```
1941.0     1941.0
 null_coun  0.0        0.0        0.0        …  0.0        0.0
0.0        0.0
 t

 mean       571.13601  617.96445  1.6507e6   …  1.403271   0.083288
-0.131305  0.58542
            2          1

 std        520.69067  497.62741  1.7746e6   …  0.454345   0.500868
0.148767   0.339452
            1

 min        0.0        4.0        6712.0     …  0.0        -0.991
-0.9989    0.119
 25%        51.0       192.0      471253.0   …  1.0792     -0.3333
-0.195     0.2482
 50%        435.0      467.0      1.204128e  …  1.3222     0.0952
-0.133     0.5063
                                  6

 75%        1053.0     1072.0     2.183073e  …  1.7324     0.5116
-0.0666    0.9998
                                  6

 max        1705.0     1713.0     1.2987661  …  4.2587     0.9917
0.6421     1.0
                                  e7
```

***i. A base de dados é consistente?*** Sim! Os tipos de cada coluna estão bem definidos e organizados.

***ii. Há dados faltantes?*** Não. Podemos ver dos dataframes gerados acima (`X` e `Y`) que não existem dados faltantes (`null_count = 0` em todas as colunas).

***iii. Há dados não numéricos?*** Não. As características das placas de aço (dataframe `X`) não contém dados não numéricos. Já os defeitos (dataframe `Y`), que são categorias, foram organizados no formato *one-hot encoding*. Além disso, há também duas colunas em `X` codificadas com *one-hot encoding*, `TypeOfSteel_A300`, `TypeOfSteel_A400`.

***iv. A base de dados é balanceada?***

[11]:
```
faults_count = Y.sum()
```

```
faults_count
```

[11]:
```
shape: (1, 7)

  Pastry   Z_Scratch   K_Scratch   Stains   Dirtiness   Bumps   Other_Faults
  ---      ---         ---         ---      ---         ---     ---
  i64      i64         i64         i64      i64         i64     i64

  158      190         391         72       55          402     673
```

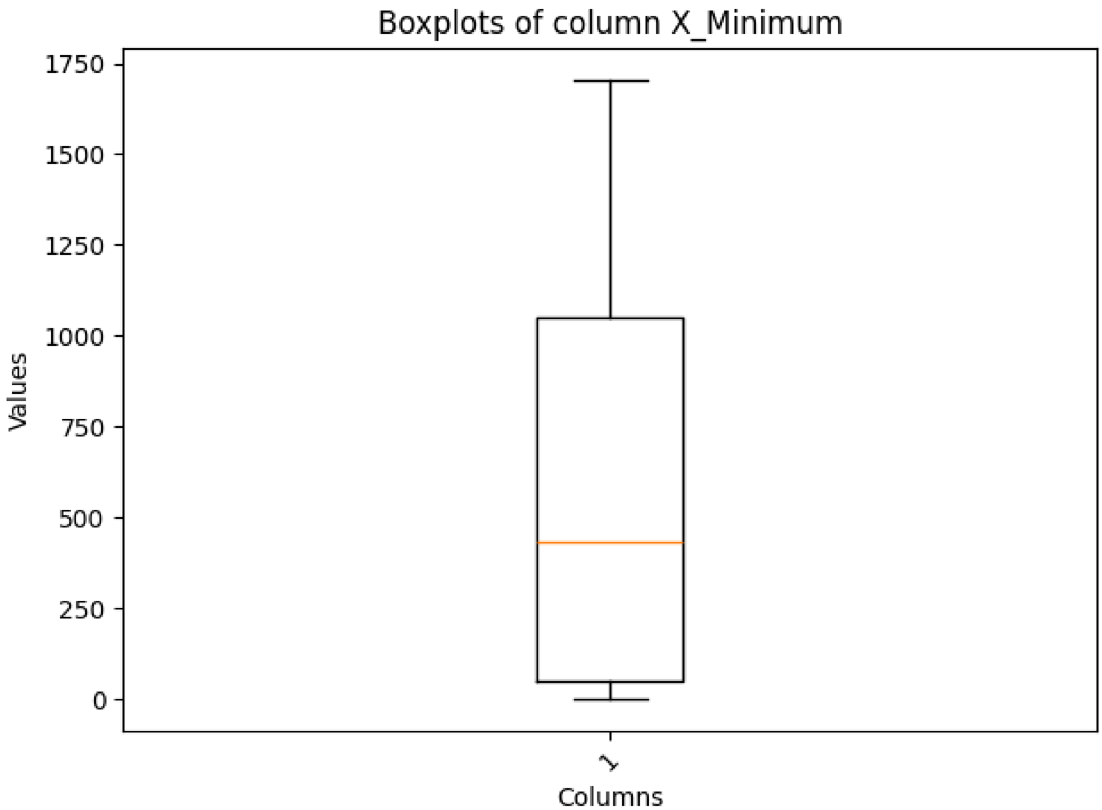
Como se vê acima, a base de dados não é balanceada quanto às classes (tipos de defeitos).

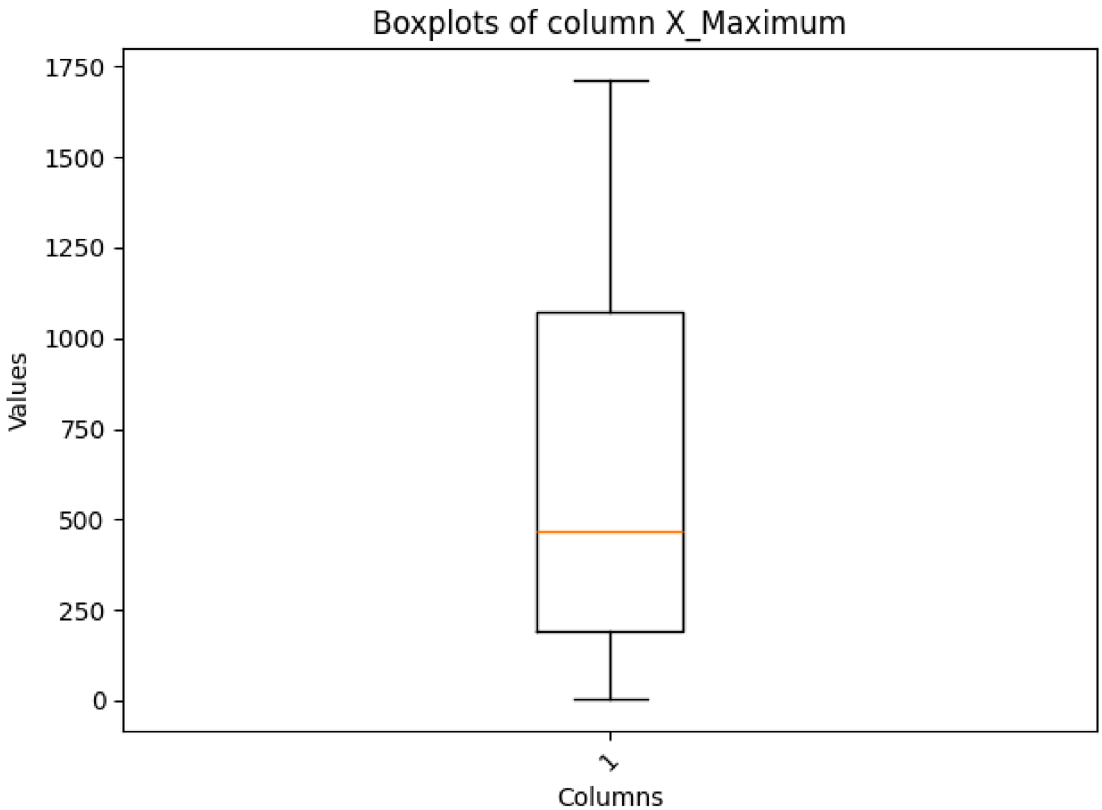
***v. As variáveis assumem valores plausíveis? Como é a distribuição dos dados? Há outliers? Faça boxplots e comente.*** Segundo o output dos métodos `describe` acima, os dados parecem assumir valores plausíveis. Vamos checar isso com boxplots. Removendo as colunas *one-hot encoding* do dataframe `X`, temos:

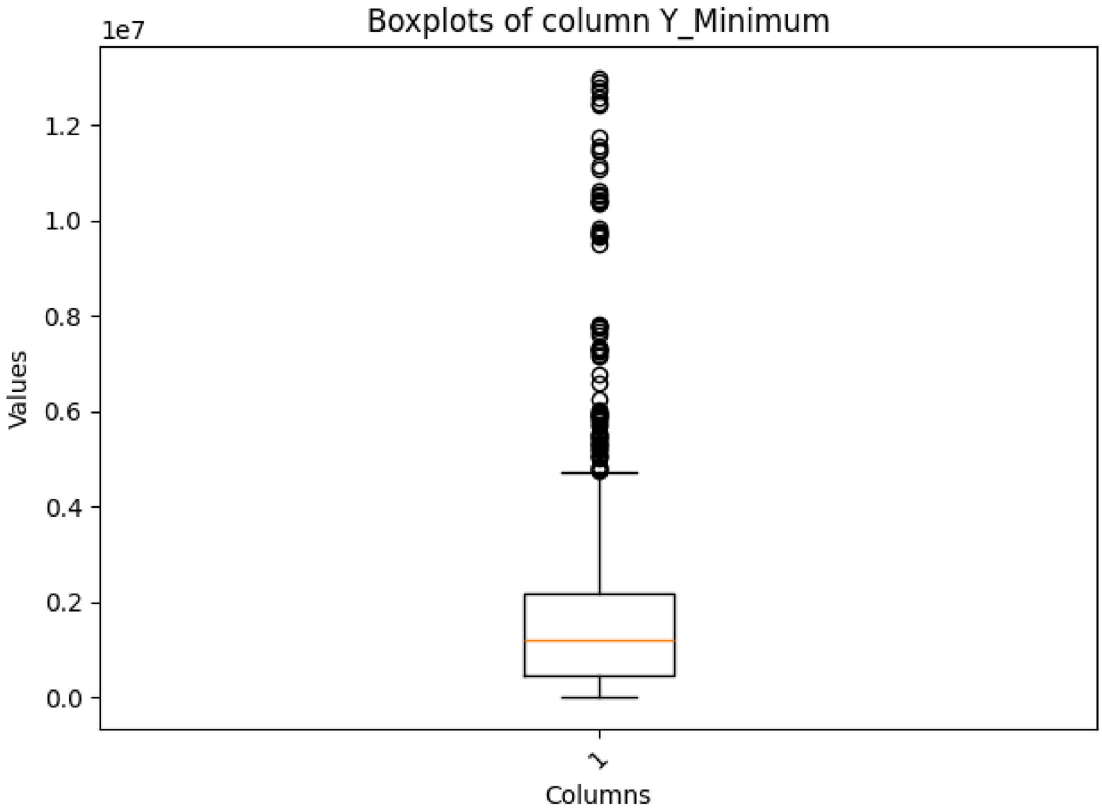
[12]:
```python
X_without_steel_type = X.select(pl.selectors.exclude('TypeOfSteel_A300', 'TypeOfSteel_A400'))
X_for_plotting = np.array([X_without_steel_type[col].to_numpy() for col in X_without_steel_type.columns])

X_for_plotting.shape
```

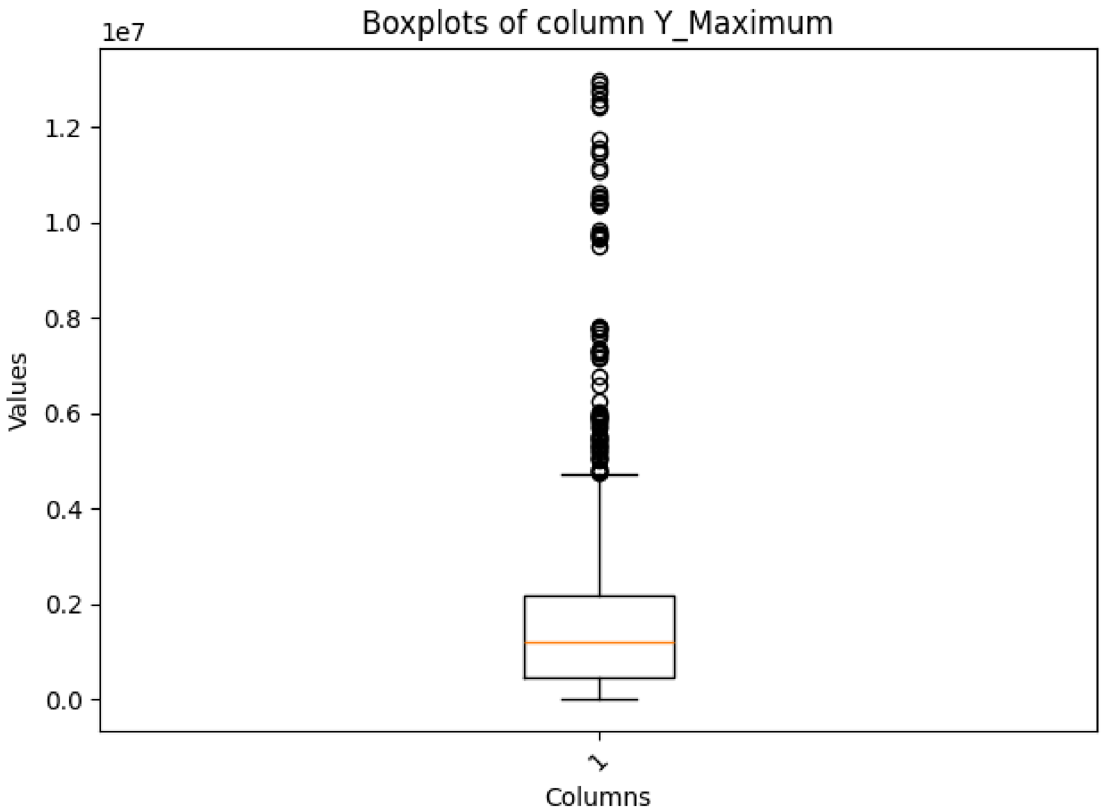
[12]:
```
(25, 1941)
```

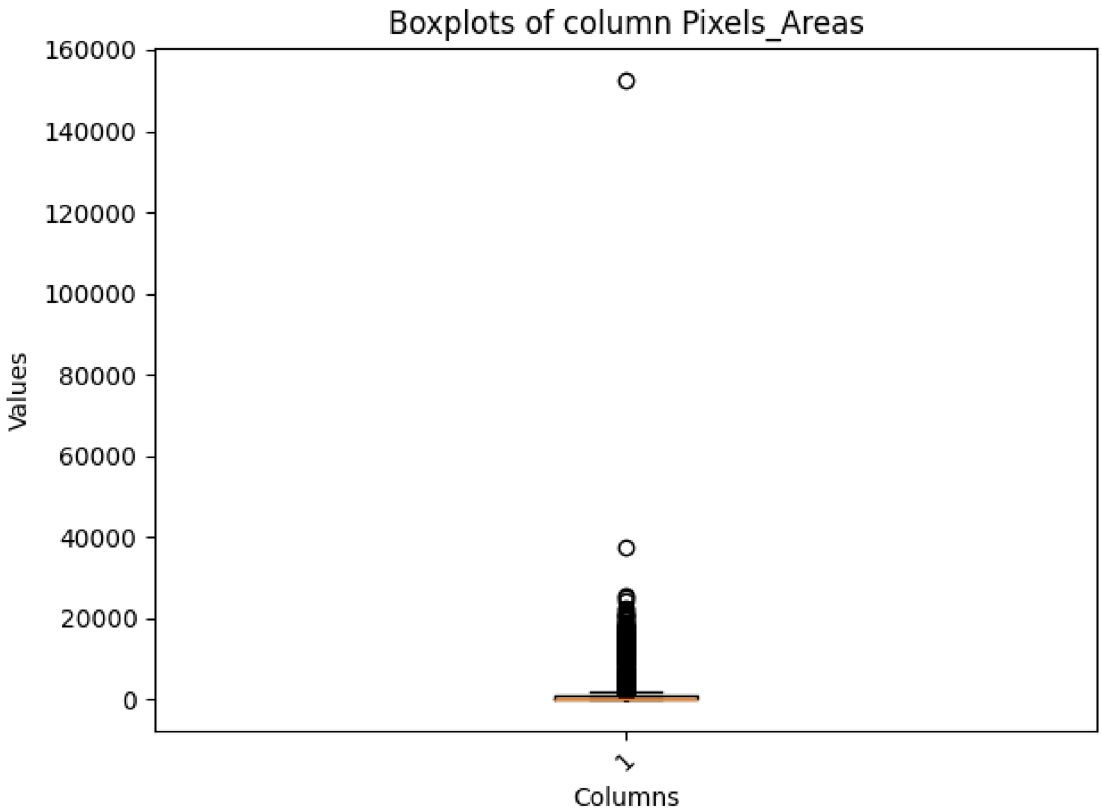
[13]:
```python
for i in range(X_without_steel_type.width):
    plt.boxplot(X_for_plotting[i])
    plt.title(f"Boxplots of column {X_without_steel_type.columns[i]}")
    plt.ylabel("Values")
    plt.xlabel("Columns")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show();
```

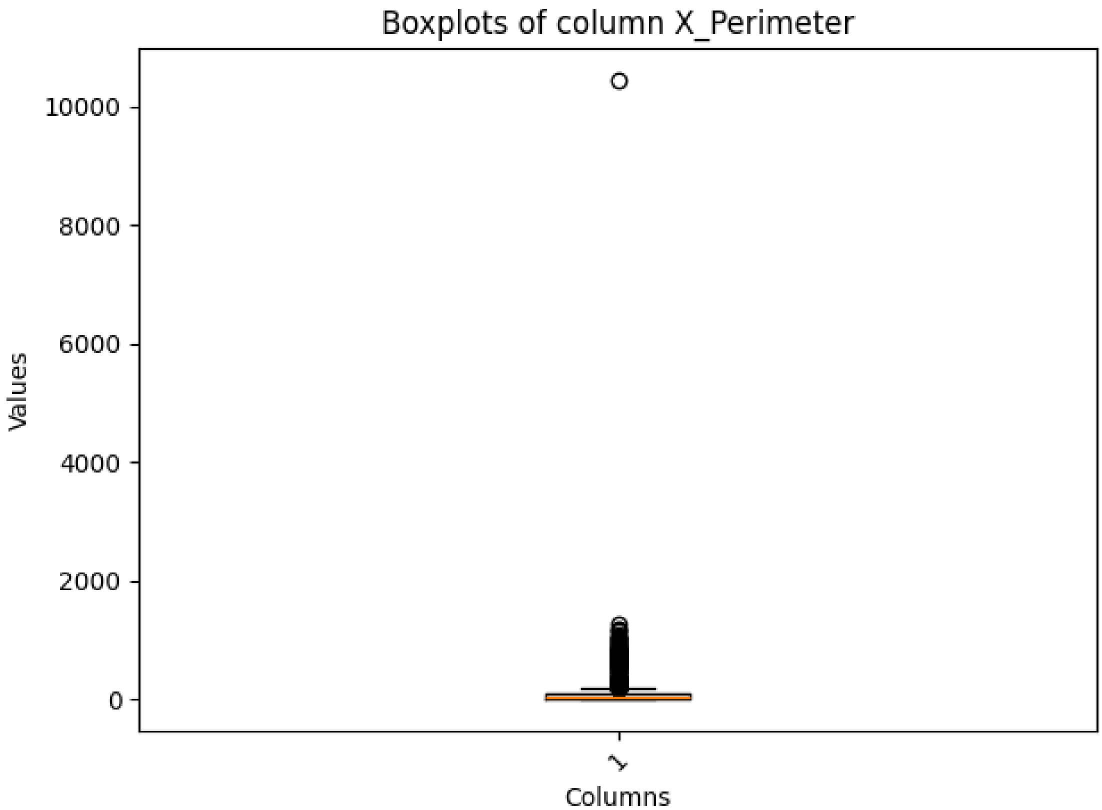

Boxplots of column X_Minimum
Values
Columns
1750
1500
1250
1000
750
500
250
0
1

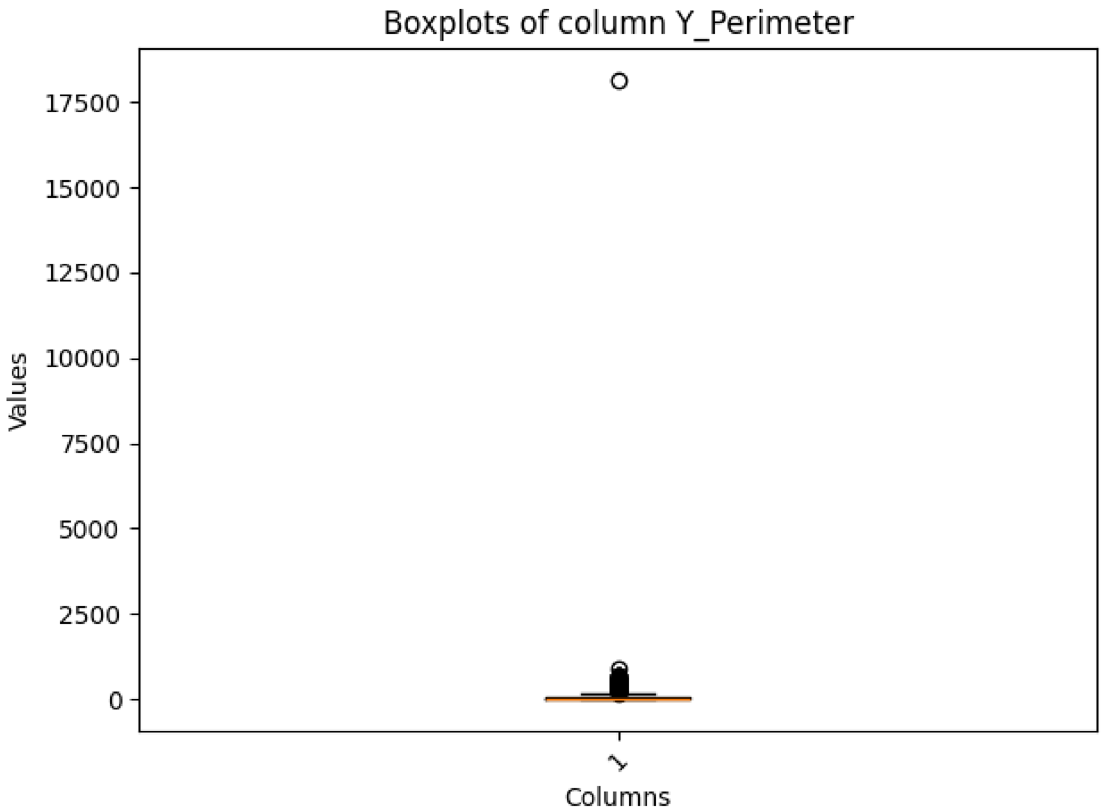

Boxplots of column X_Maximum
1750
1500
1250
1000
750
500
250
0
Values
1
Columns


Boxplots of column Y_Minimum
1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

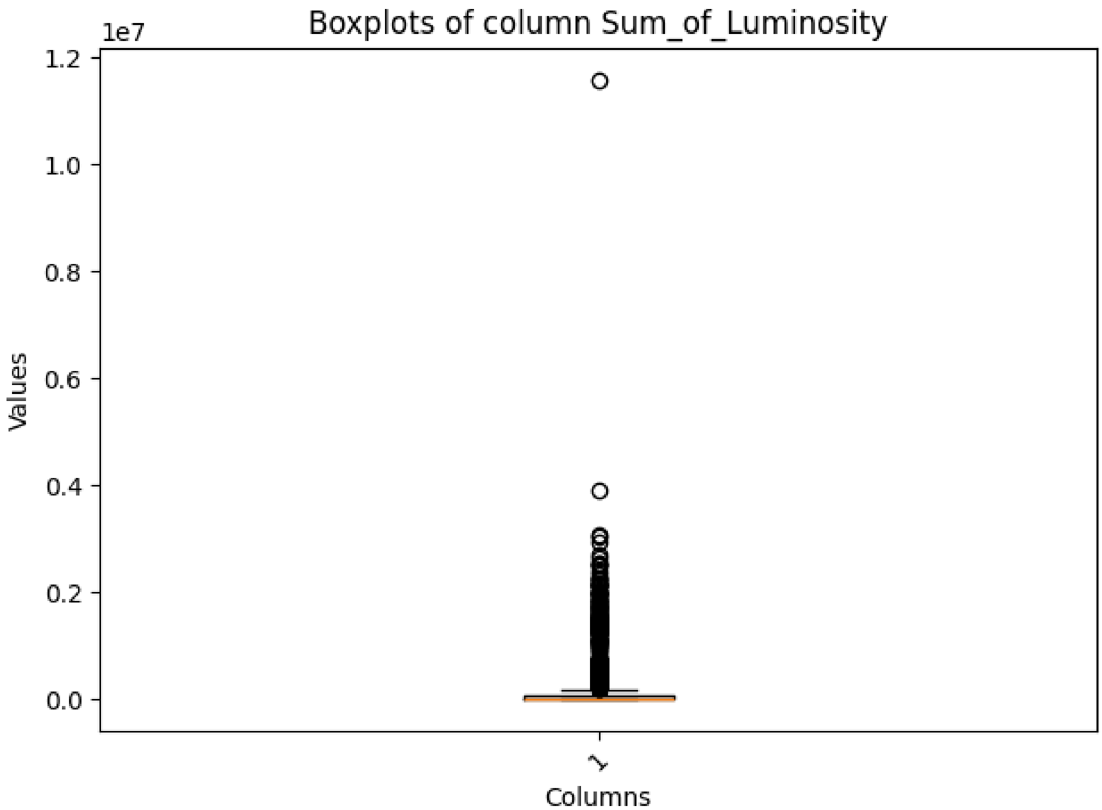

Boxplots of column Y_Maximum
1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

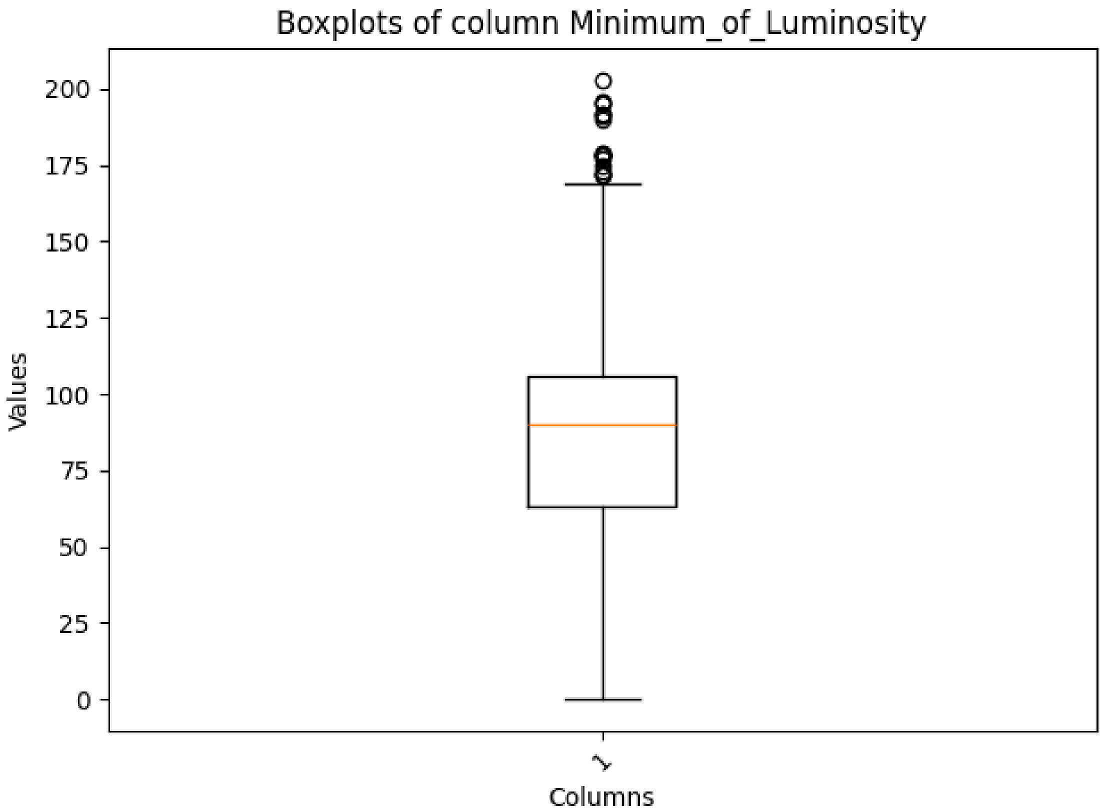

Boxplots of column Pixels_Areas
160000
140000
120000
100000
80000
60000
40000
20000
0
Values
1
Columns

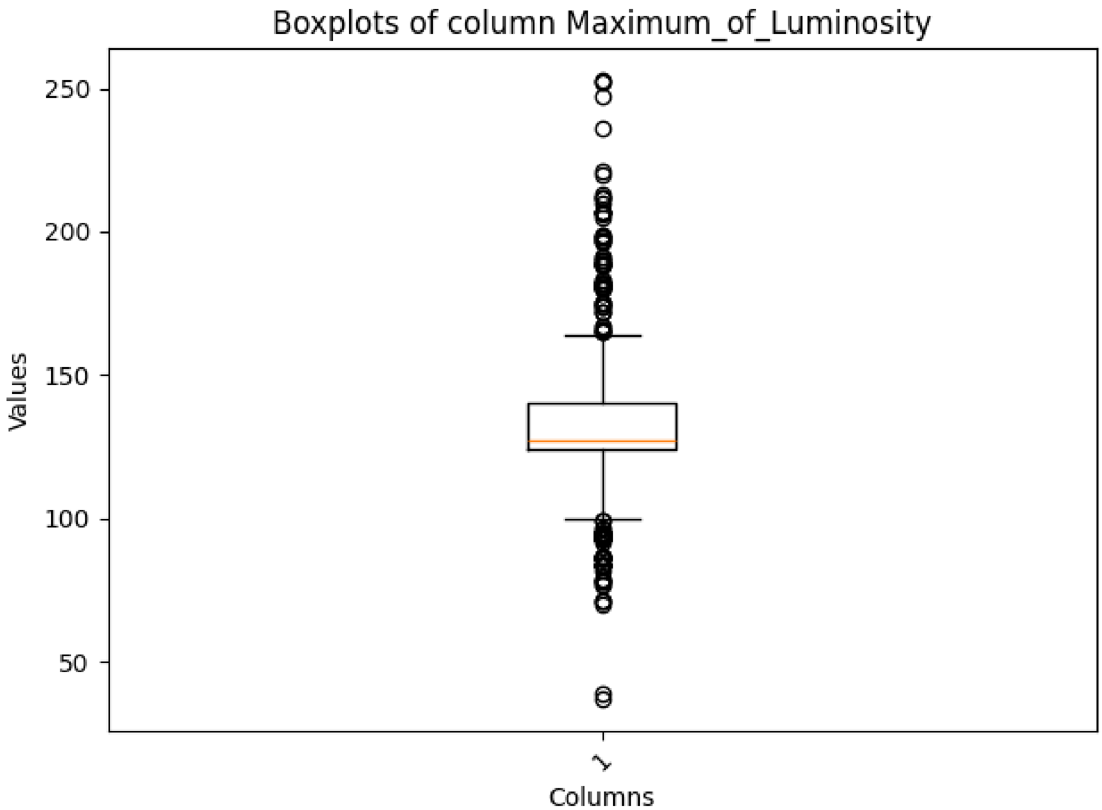

Boxplots of column X_Perimeter
Values
10000
8000
6000
4000
2000
0
1
Columns

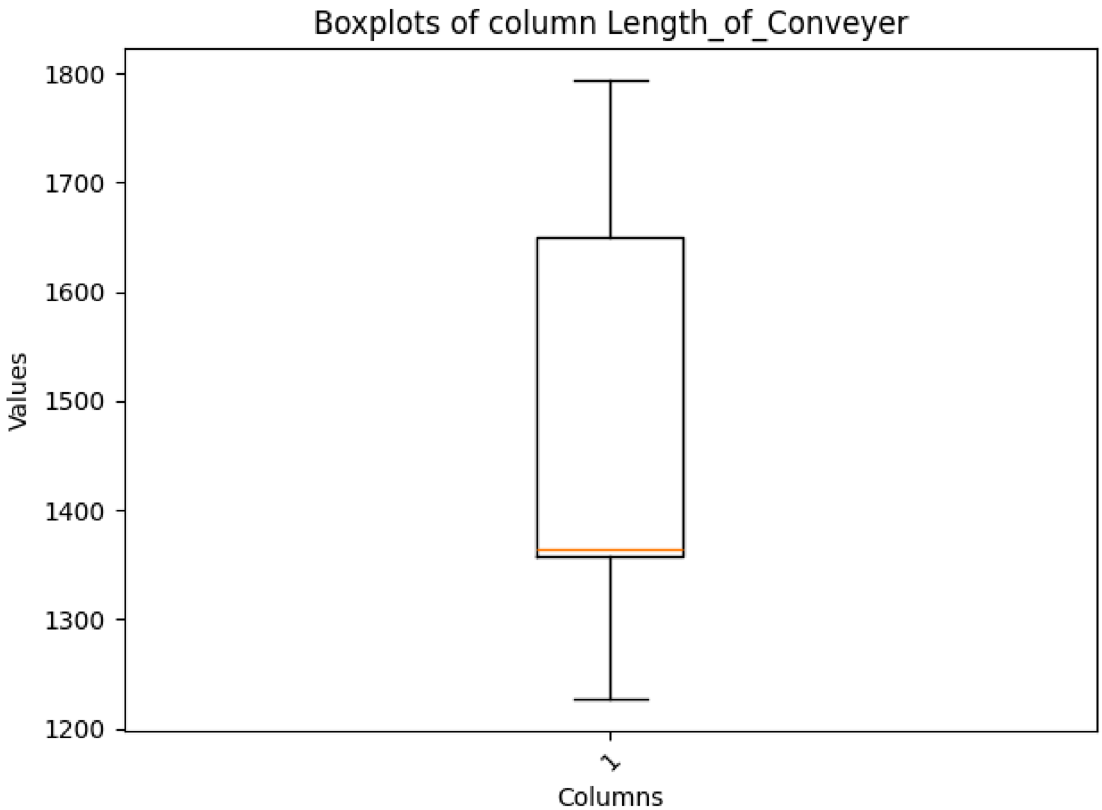


Boxplots of column Y_Perimeter
Values
17500
15000
12500
10000
7500
5000
2500
0
1
Columns

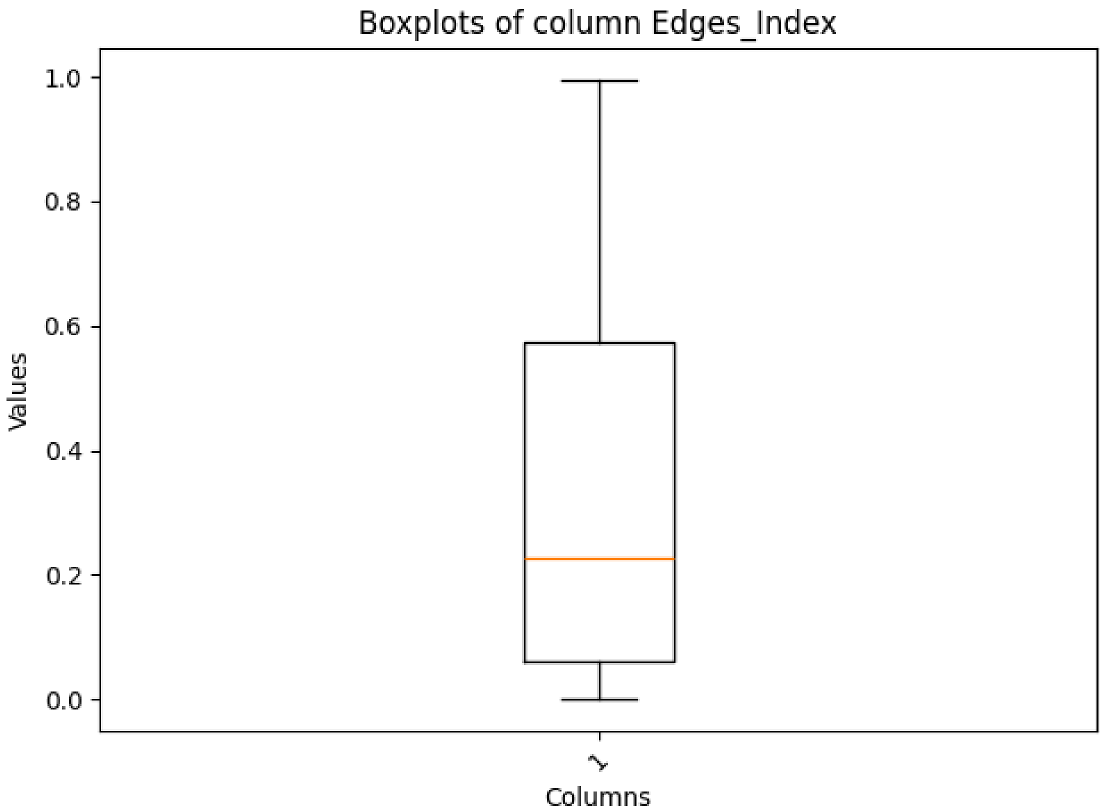

Boxplots of column Sum_of_Luminosity
1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

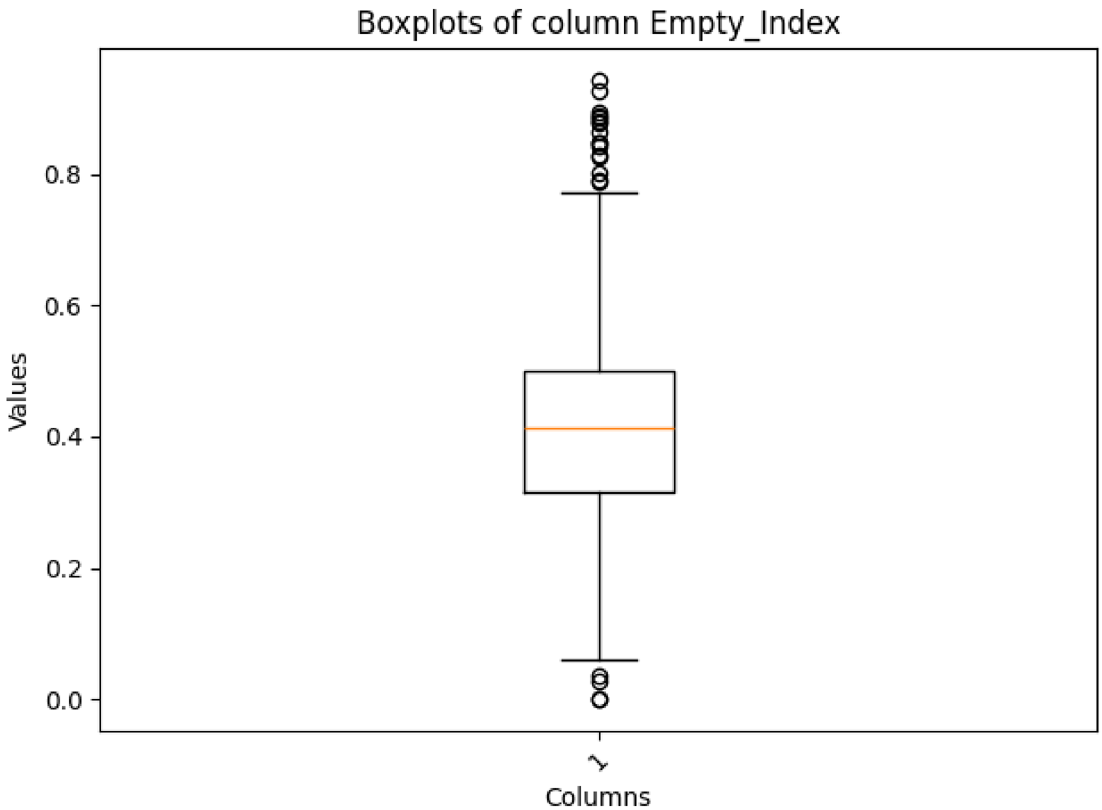

Boxplots of column Minimum_of_Luminosity
Values
200
175
150
125
100
75
50
25
0
1
Columns

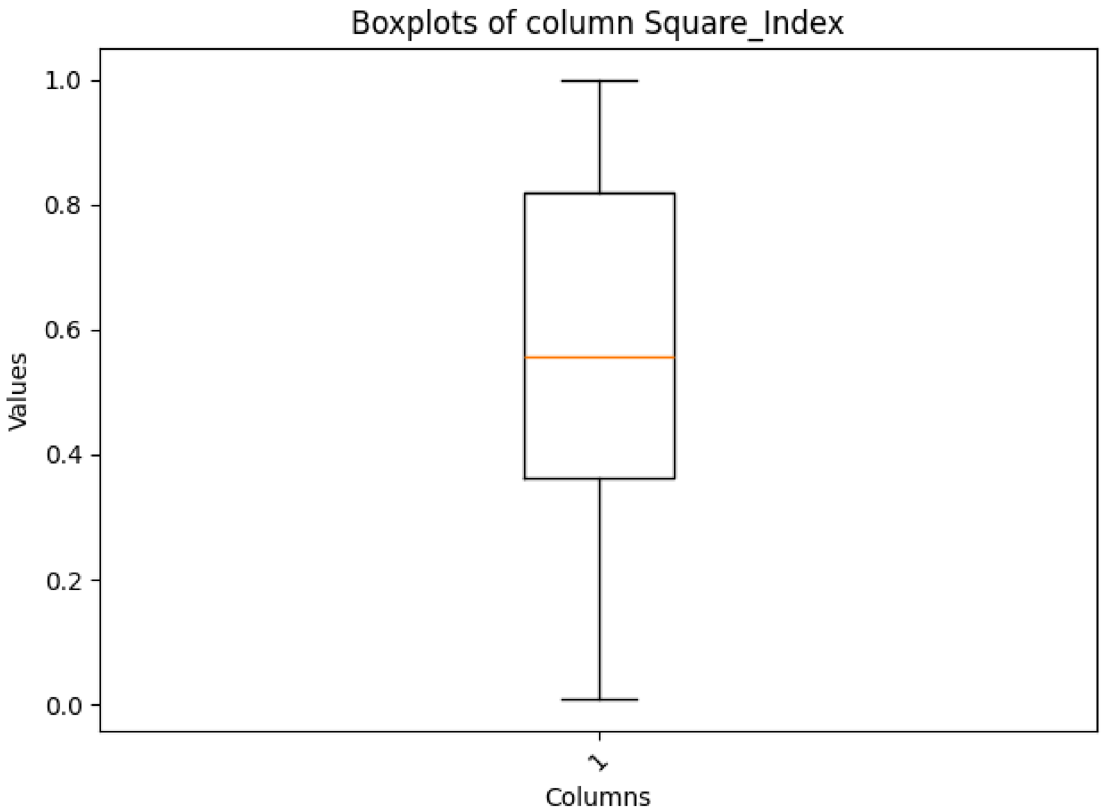

Boxplots of column Maximum_of_Luminosity
250
200
150
100
50
Values
1
Columns

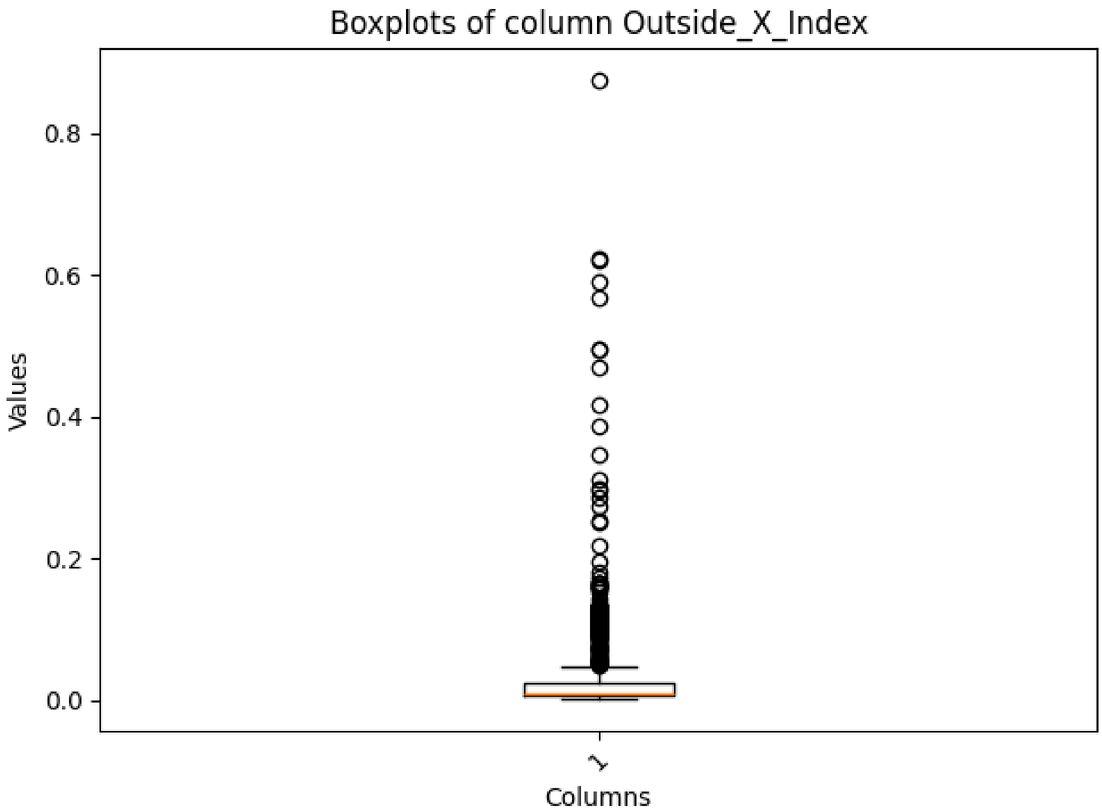

Boxplots of column Length_of_Conveyer
1800
1700
1600
1500
1400
1300
1200
Values
1
Columns

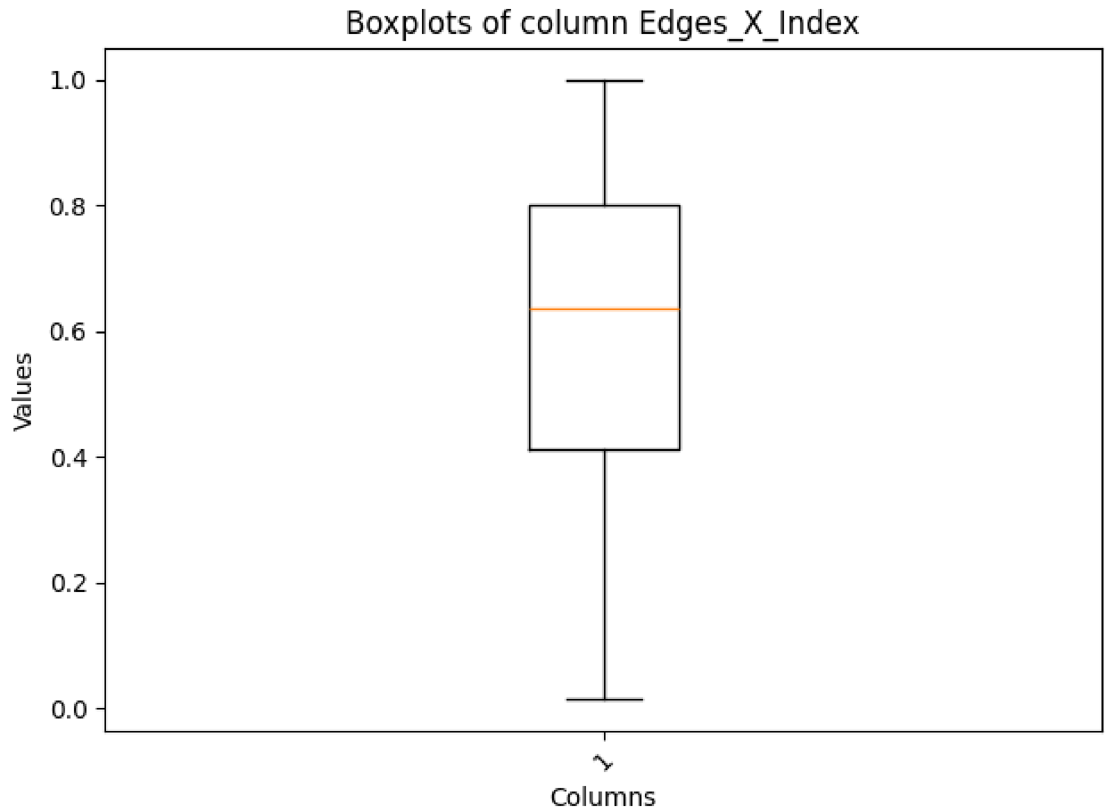

Boxplots of column Steel_Plate_Thickness
300
250
200
150
100
50
Values
1
Columns


Boxplots of column Edges_Index
1.0
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

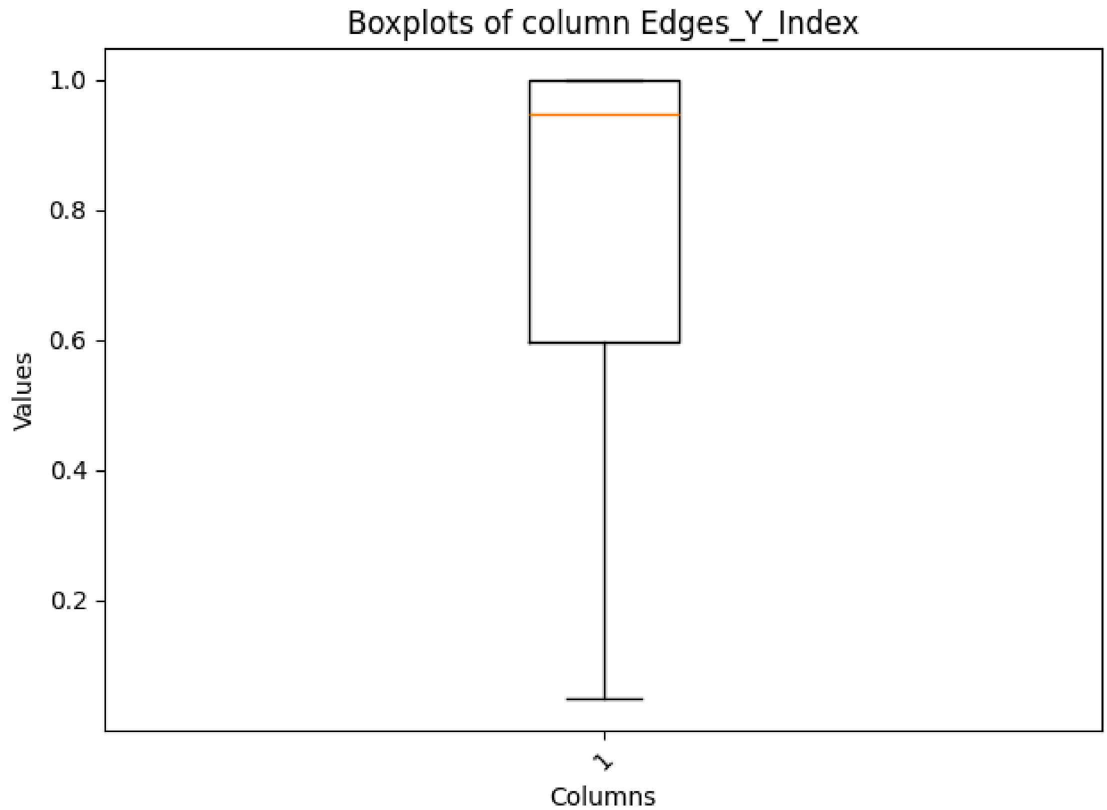

Boxplots of column Empty_Index
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

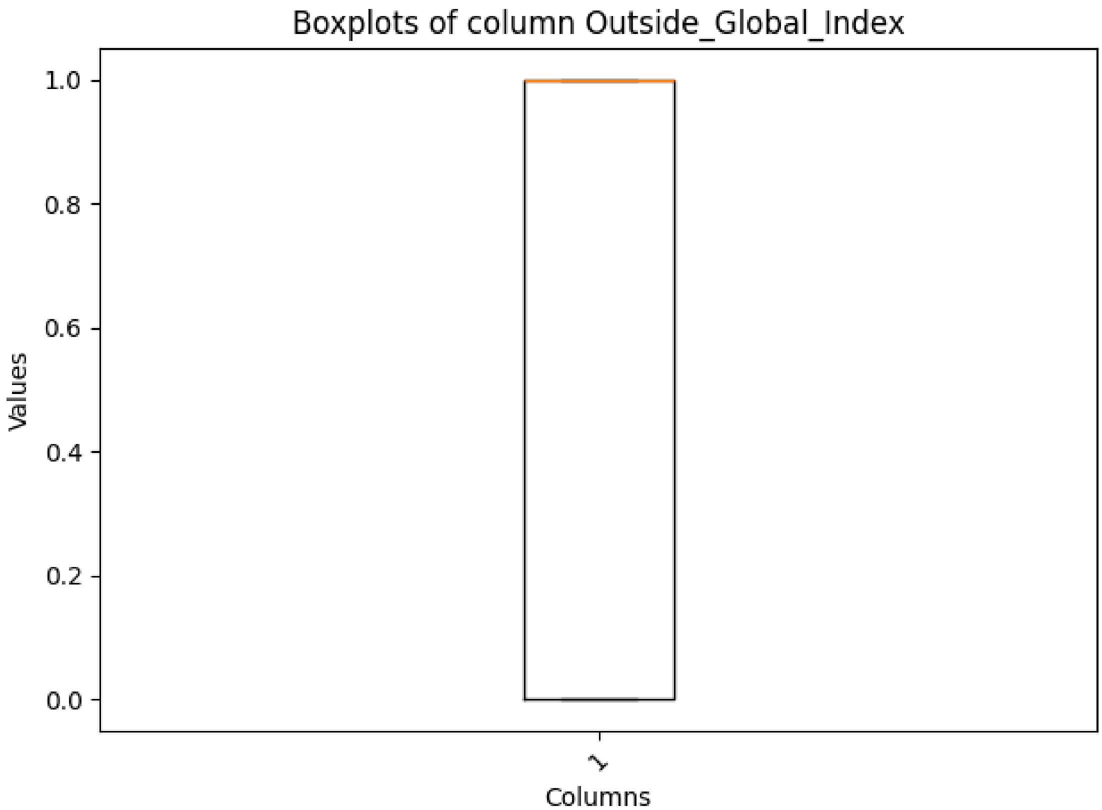

Boxplots of column Square_Index
Values
Columns
1.0
0.8
0.6
0.4
0.2
0.0
1


Boxplots of column Outside_X_Index
Values
0.0
0.2
0.4
0.6
0.8
1
Columns

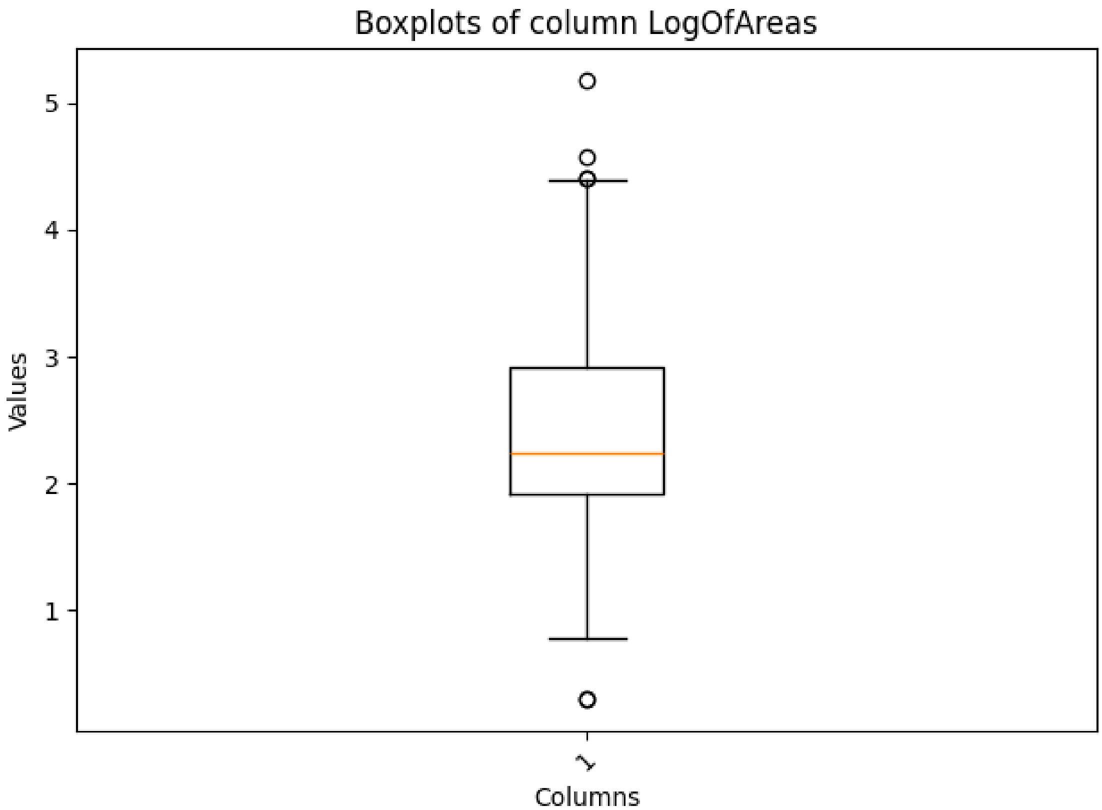


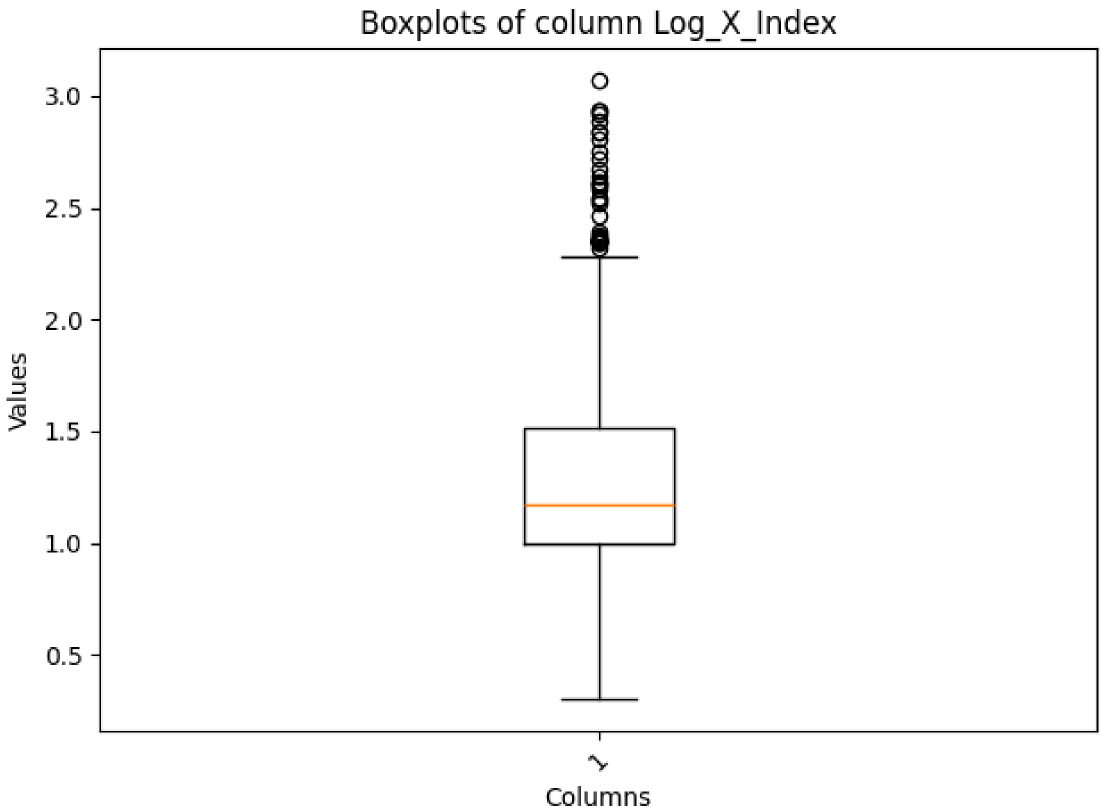
Boxplots of column Edges_X_Index
Values
1.0
0.8
0.6
0.4
0.2
0.0
1
Columns

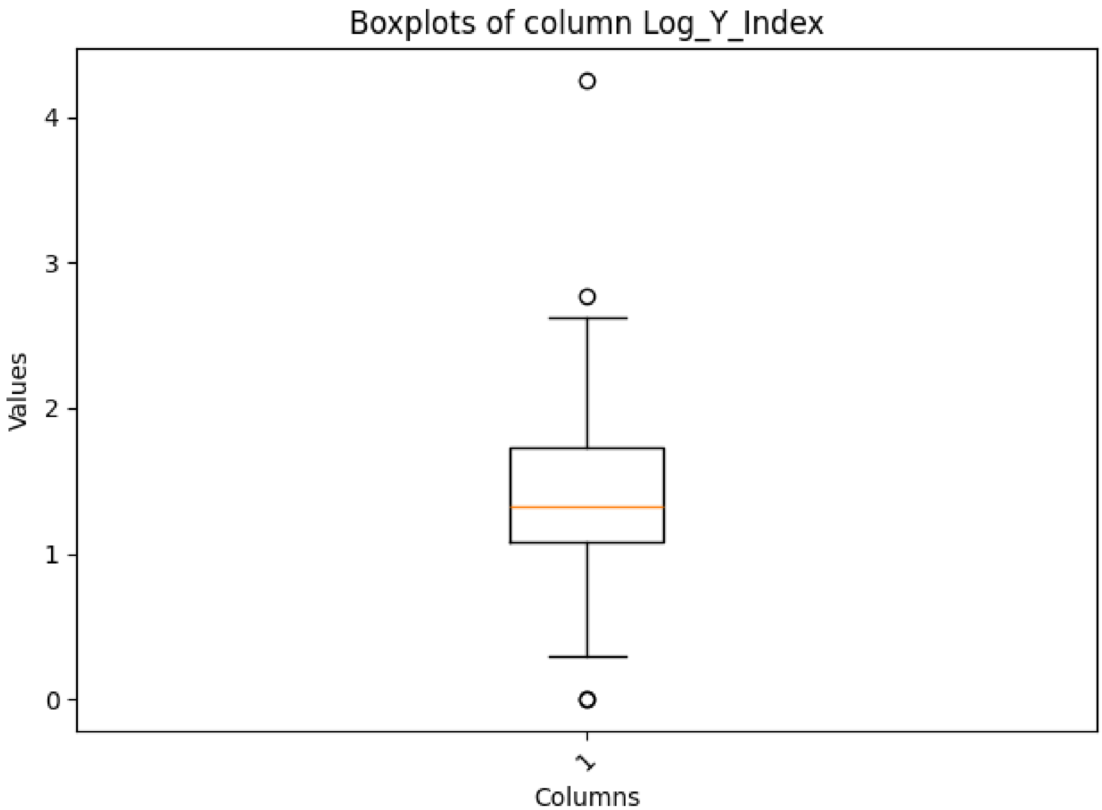

Boxplots of column Edges_Y_Index
1.0
0.8
0.6
0.4
0.2
Values
1
Columns

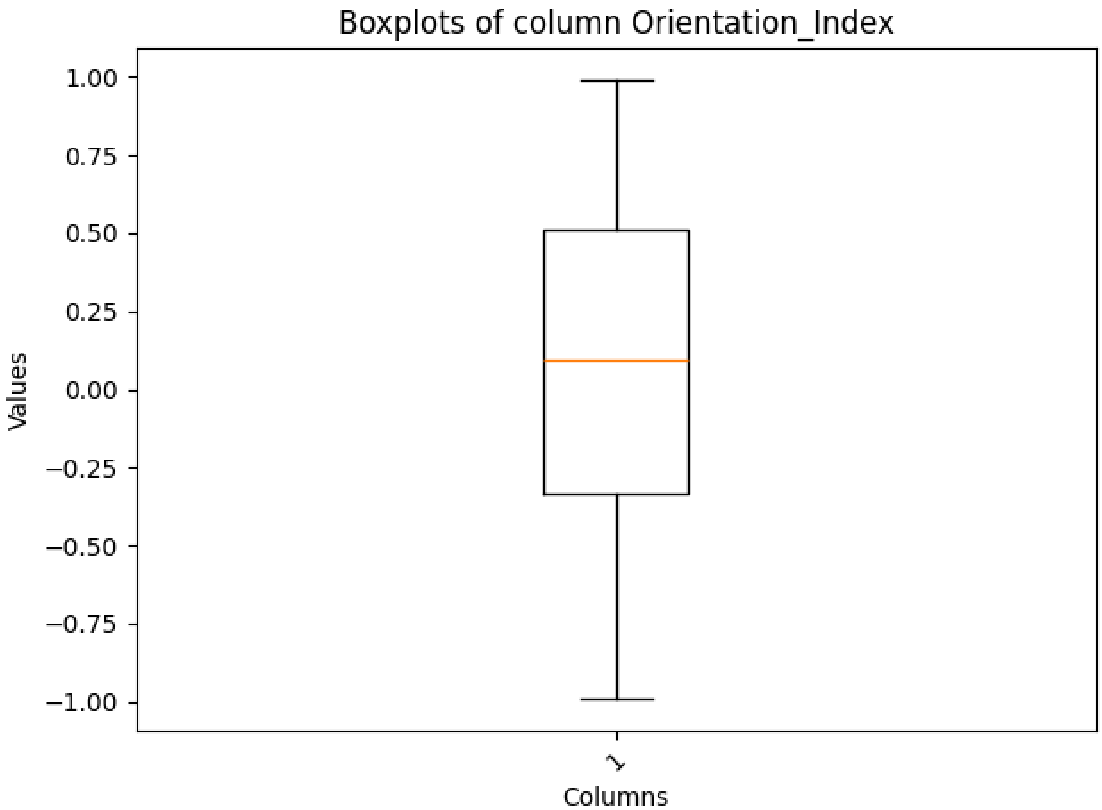


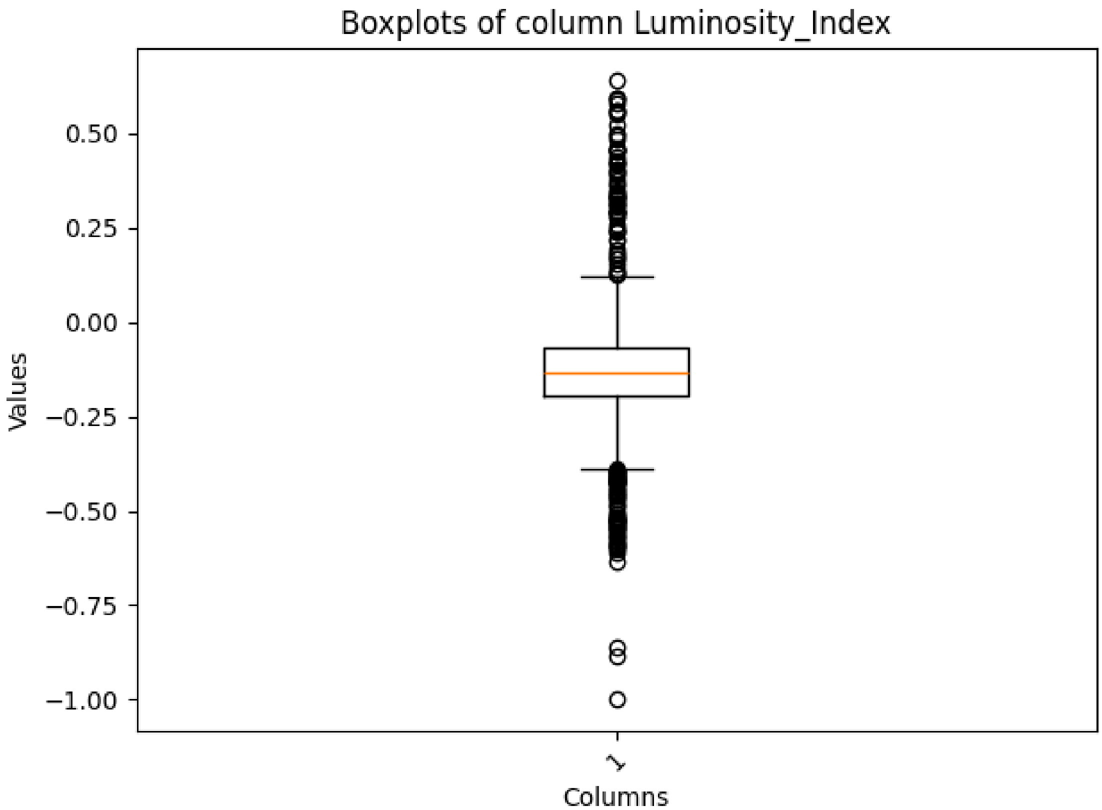
Boxplots of column Outside_Global_Index
1.0
0.8
0.6
0.4
0.2
0.0
Values
1
Columns

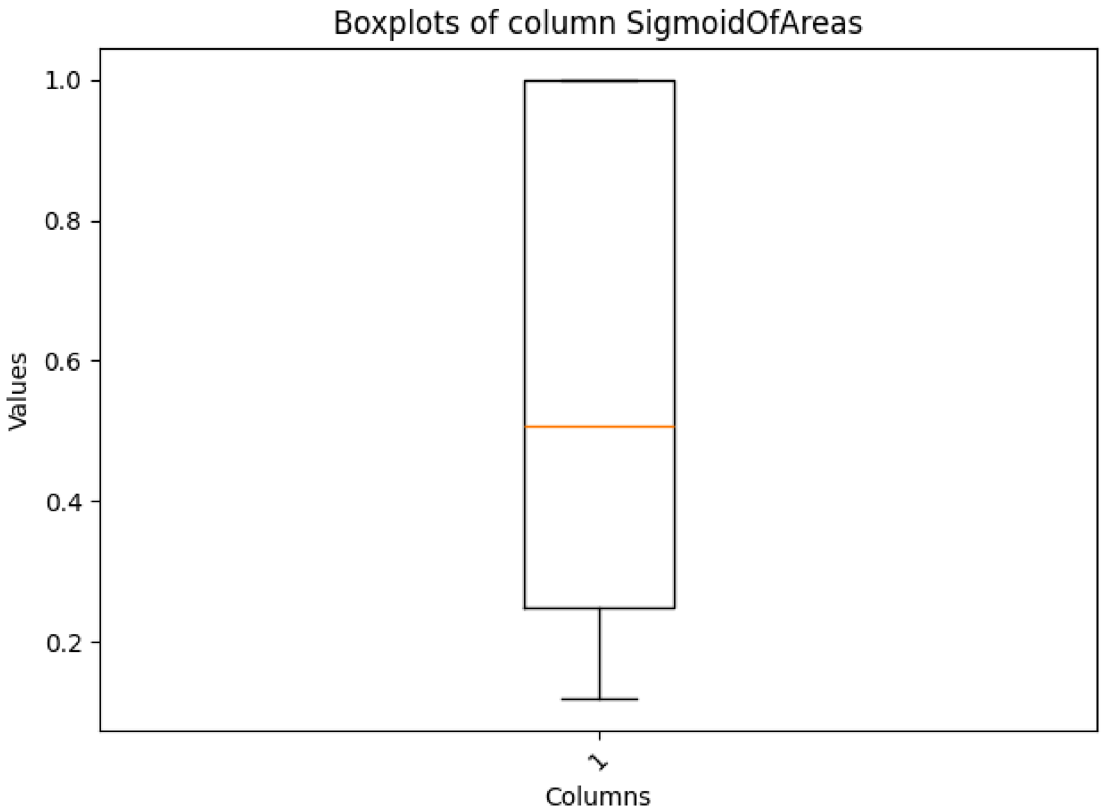

Boxplots of column LogOfAreas
Values
5
4
3
2
1
Columns
1


Boxplots of column Log_X_Index
3.0
2.5
2.0
1.5
1.0
0.5
Values
1
Columns


Boxplots of column Log_Y_Index
4
3
2
1
0
Values
1
Columns


Boxplots of column Orientation_Index
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Values
1
Columns


Boxplots of column Luminosity_Index
Values
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
1
Columns




Parece que existem muitos outliers em várias colunas, vamos quantificá-los:

[14]:
```
def calculate_outliers_percentage(series):
    q1 = series.quantile(0.25)
    q3 = series.quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    # Count outliers
    outliers_count = series.filter((series < lower_bound) | (series > upper_bound)).shape[0]
    total_count = series.shape[0]

    # Calculate percentage of outliers
    percentage = (outliers_count / total_count) * 100
    return round(percentage, 2)
```

[15]:
```
outlier_percentages = {col: calculate_outliers_percentage(X_without_steel_type[col]) for col in X_without_steel_type.columns}
```

```python
outlier_percentages = {'statistic': 'outlier percentage'} | outlier_percentages

# Create a new DataFrame for outlier percentages
outlier_row = pl.DataFrame(outlier_percentages)

# Add the outlier percentages row to the original DataFrame
X_description_with_outliers = pl.concat([X_without_steel_type.describe(),␣
 ↪outlier_row])

X_description_with_outliers
```

[15]: shape: (10, 26)

```
 statistic   X_Minimum   X_Maximum   Y_Minimum   …   Log_Y_Ind   Orientati
Luminosit   SigmoidO
 ---         ---         ---         ---             ex          on_Index
y_Index     fAreas
 str         f64         f64         f64             ---         ---
---         ---
                                                     f64         f64
f64         f64


 count       1941.0      1941.0      1941.0      …   1941.0      1941.0
1941.0      1941.0
 null_coun   0.0         0.0         0.0         …   0.0         0.0
0.0         0.0
 t

 mean        571.13601   617.96445   1.6507e6    …   1.403271    0.083288
-0.131305   0.58542
             2           1

 std         520.69067   497.62741   1.7746e6    …   0.454345    0.500868
0.148767    0.339452
             1

 min         0.0         4.0         6712.0      …   0.0         -0.991
-0.9989     0.119
 25%         51.0        192.0       471253.0    …   1.0792      -0.3333
-0.195      0.2482
 50%         435.0       467.0       1.204128e   …   1.3222      0.0952
-0.133      0.5063
                                     6

 75%         1053.0      1072.0      2.183073e   …   1.7324      0.5116
```

-0.0666 0.9998
6

max 1705.0 1713.0 1.2987661 … 4.2587 0.9917
0.6421 1.0
e7

outlier 0.0 0.0 4.17 … 0.21 0.0
6.9 0.0
percentag

e

[16]:
```python
outliers_row = X_description_with_outliers.row(by_predicate = (pl.
    col('statistic') == "outlier percentage"))

outlier_percentage_threshold = 15
big_outliers_indexes = [
    index for index, value in enumerate(outliers_row)
    if isinstance(value, (int, float)) and value > outlier_percentage_threshold
]
X_description_big_outliers = X_description_with_outliers[:,
    big_outliers_indexes]

X_description_big_outliers
```

[16]: shape: (10, 4)

| Pixels_Areas | X_Perimeter | Sum_of_Luminosity | Outside_X_Index |
|---|---|---|---|
| --- | --- | --- | --- |
| f64 | f64 | f64 | f64 |
| 1941.0 | 1941.0 | 1941.0 | 1941.0 |
| 0.0 | 0.0 | 0.0 | 0.0 |
| 1893.878413 | 111.855229 | 206312.147862 | 0.033361 |
| 5168.45956 | 301.209187 | 512293.587609 | 0.058961 |
| 2.0 | 2.0 | 250.0 | 0.0015 |
| 84.0 | 15.0 | 9522.0 | 0.0066 |
| 174.0 | 26.0 | 19202.0 | 0.0101 |
| 822.0 | 84.0 | 83011.0 | 0.0235 |
| 152655.0 | 10449.0 | 1.1591414e7 | 0.8759 |
| 20.35 | 18.13 | 20.56 | 19.06 |

[17]:
```python
X_description_big_outliers.columns
```

[17]:
```
['Pixels_Areas', 'X_Perimeter', 'Sum_of_Luminosity', 'Outside_X_Index']
```

Como se vê, as características (*features*) acima possuem um alto número de outliers (mais do que 15%).

***vi. Há necessidade de normalizar ou padronizar as variáveis de entrada? Justifique.*** Parece que sim, posto que há uma variabilidade muito grande entre os valores de cada variável (indo da ordem de 10  até 10 ) e sabemos que redes neurais funcionam melhor com dados mais uniformes. Vamos fazer isso posteriormente, antes do treinamento.

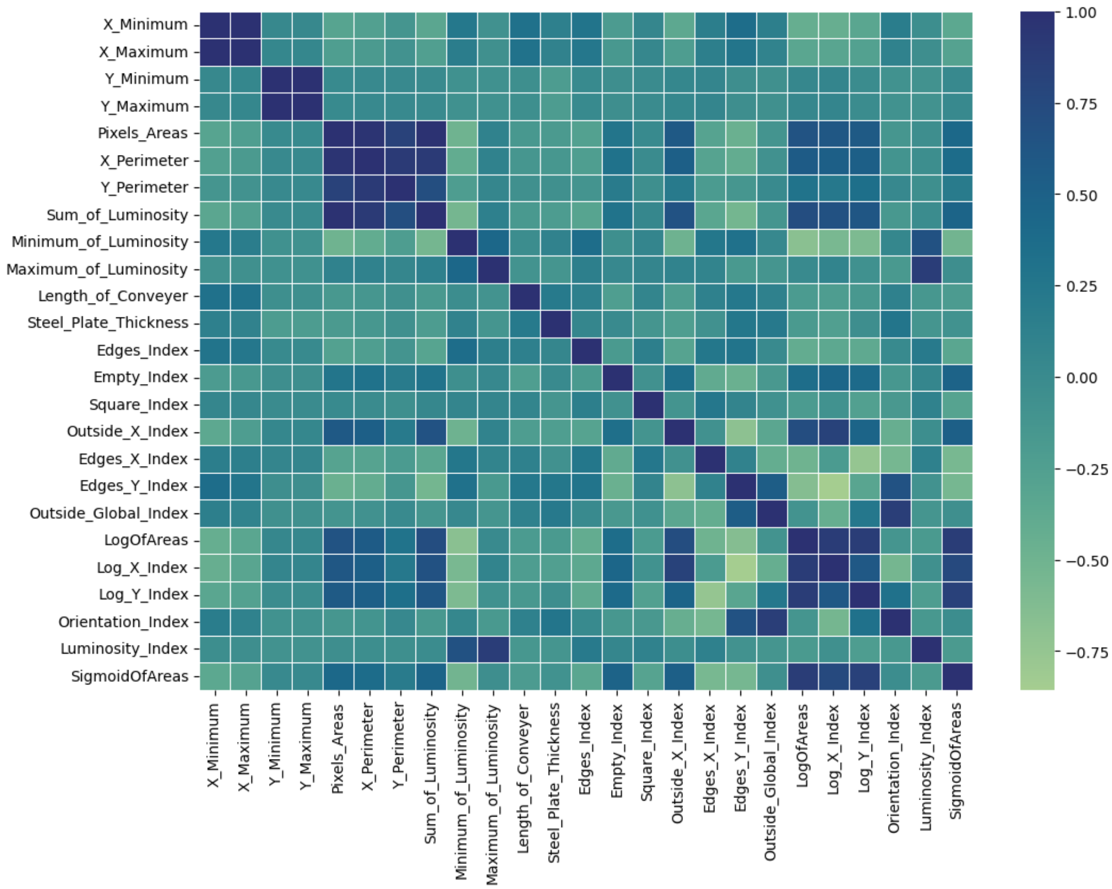
***vii. Analise o heatmap das variáveis e proponha uma estratégia para reduzir as variáveis de entrada sem perda de informação útil para o classificador.*** Heatmap com o matplotlib, excluindo as variáveis (*features*) codificadas com *one-hot encoding*:

[18]:
```python
X_without_steel_type.corr()
```

[18]:
```
shape: (25, 25)


  X_Minimum   X_Maximum   Y_Minimum   Y_Maximum   …   Log_Y_Ind   Orientati
Luminosit   SigmoidO
  ---         ---         ---         ---             ex          on_Index
y_Index     fAreas
  f64         f64         f64         f64             ---         ---
---         ---
                                                      f64         f64
f64         f64


  1.0         0.988314    0.041821    0.041807    …   -0.326851   0.178585
-0.031578   -0.35525


  1
  0.988314    1.0         0.052147    0.052135    …   -0.26599    0.115019
-0.038996   -0.28673


  6
  0.041821    0.052147    1.0         1.0         …   -0.008442   -0.086497
-0.090654   0.025257
  0.041807    0.052135    1.0         1.0         …   -0.008382   -0.08648
-0.090666   0.025284
  -0.307322   -0.225399   0.01767     0.01784     …   0.578342    -0.137604
-0.043449   0.422947
  …           …           …           …           …   …           …           …
  …
```

```
  -0.437944   -0.324012   0.070406    0.070432    …   0.598652    -0.536629
-0.064923   0.757343
  -0.326851   -0.26599    -0.008442   -0.008382   …   1.0         0.316792
-0.21911    0.838188
  0.178585    0.115019    -0.086497   -0.08648    …   0.316792    1.0
-0.153464   -0.02397

  8
  -0.031578   -0.038996   -0.090654   -0.090666   …   -0.21911    -0.153464
1.0         -0.18484
  -0.355251   -0.286736   0.025257    0.025284    …   0.838188    -0.023978
-0.18484    1.0
```

[19]:
```
import seaborn as sns

dim = (11.7, 8.27)
fig, ax = plt.subplots(figsize = dim)

ax = sns.heatmap(
    X_without_steel_type.corr(),
    linewidth=0.5,
    cmap='crest',
    xticklabels=X_without_steel_type.columns, yticklabels=X_without_steel_type.
 ↪columns)
plt.show()
```




Visualmente podemos perceber que existe correlação mais forte entre as variáveis `X_Minimum/X_Maximum`, `Y_Minimum/Y_Maximum`, `Pixels_Areas/X_Perimeter/Y_Perimeter/Sum_of_Luminosity`, `LogOfAreas/Log_X_Index/Log_Y_Index`. Vamos quantificar melhor a magnitude dessas correlações, considerando apenas variáveis cuja correlação é maior do que 95%:

[20]:
```
correlation_threshold = .95
X_corr = X_without_steel_type.corr()
corr_var_indexes = [
    {i, j} for i in range(X_corr.height) for j in range(X_corr.width)
    if abs(X_corr[i, j]) > correlation_threshold
    and i != j
]
corr_var_indexes = list({frozenset(i) for i in corr_var_indexes})

# Display the results
for row, col in corr_var_indexes:
    print(f"Value at ({X_corr.columns[row]}, '{X_corr.columns[col]}') fulfills the condition.")
```

```
Value at (Pixels_Areas, 'X_Perimeter') fulfills the condition.
Value at (X_Minimum, 'X_Maximum') fulfills the condition.
Value at (Y_Minimum, 'Y_Maximum') fulfills the condition.
Value at (Pixels_Areas, 'Sum_of_Luminosity') fulfills the condition.
```

Como se vê acima, os pares de variáveis listados têm uma correlação de mais de 95%. Vamos dropar as variáveis redundantes:

[21]:
```
X = X.drop(['X_Perimeter', 'X_Maximum', 'Y_Maximum', 'Sum_of_Luminosity'],␣
↪strict = False)

X
```

[21]:
```
shape: (1_941, 23)


  X_Minimum   Y_Minimum   Pixels_Ar   Y_Perimet   …   Log_Y_Ind   Orientati
Luminosit   SigmoidO
  ---         ---         eas         er              ex          on_Index
y_Index     fAreas
  i64         i64         ---         ---             ---         ---
---         ---
                          i64         i64             f64         f64
f64         f64



  42          270900      267         44          …   1.6435      0.8182
-0.2913     0.5822
  645         2538079     108         30          …   1.4624      0.7931
-0.1756     0.2984
  829         1553913     71          19          …   1.2553      0.6667
-0.1228     0.215
  853         369370      176         45          …   1.6532      0.8444
-0.1568     0.5212
  1289        498078      2409        260         …   2.4099      0.9338
-0.1992     1.0
  …           …           …           …           …   …           …           …
  …
  249         325780      273         22          …   1.2041      -0.4286
0.0026      0.7254
  144         340581      287         24          …   1.2305      -0.4516
-0.0582     0.8173
  145         386779      292         22          …   1.1761      -0.4828
0.0052      0.7079
  137         422497      419         47          …   1.4914      -0.0606
-0.0171     0.9919
  1261        87951       103         22          …   1.2041      -0.2
-0.1139     0.5296
```

Vamos salvar nossos dados nas variáveis `X_0` e `Y_0` antes de manipulá-los mais:

[22]:
```
X_0 = X
Y_0 = Y
```

### 1.0.3 *3. Particione aleatoriamente 70% das amostras para treinar a rede neural MLP e o restante das amostras para validar o sistema.*

Antes de proceder com a partição do conjunto, vamos aplicar algumas transformações nos dados visando melhorar o resultado da rede neural.

[47]:
```
# Retrieving the data from the saved state in the code cell above

X = X_0
Y = Y_0
```

Repare no entanto que nos dados de entrada existem *features* categóricas que *podem* não ser transformadas corretamente. Vamos salvá-las a parte:

[48]:
```
# Lets remove the categorical features before winsorization
features_categorical = ["TypeOfSteel_A300", "TypeOfSteel_A400"]
X_categorical = X[features_categorical]

X_continuous = X.select(pl.col("*").exclude(features_categorical))
features_continuous = X_continuous.columns
```

Vamos primeiramente remover os valores mais discrepantes através da função `winsorize` (link da documentação), que limita os valores limites nos dados, reduzindo o efeito dos *outliers* (ver *Winsorizing*):

[49]:
```
from scipy.stats import mstats

# Winsorize the data (e.g., cap at 5st and 95th percentiles)
X_winsorized = mstats.winsorize(X_continuous.to_numpy(), limits=[0.05, 0.05],␣
↪axis = 0)
X_winsorized = pl.DataFrame(X_winsorized, schema = features_continuous)
X_continuous = X_winsorized
```

Podemos perceber que há uma diminuição substantiva na variabilidade dos dados após o processo acima. Vamos agora normalizar as variáveis de entrada com auxilio do pacote `sklearn` (link):

[50]:
```
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler

# Initialize the scaler
scaler = MinMaxScaler(feature_range = (-1, 1))
```

```python
# Fit and transform the features
X_scaled = scaler.fit_transform(X_continuous)
X_continuous = pl.DataFrame(X_scaled, schema = features_continuous)
```

Antes de prosseguir com a partição dos dados, vamos tratar do problema de balanceamento das classes. Conforme visto anteriormente, a distribuição é:

[51]:
```python
Y.sum()
```

[51]:
```
shape: (1, 7)

  Pastry   Z_Scratch   K_Scratch   Stains   Dirtiness   Bumps   Other_Faults
  ---      ---         ---         ---      ---         ---     ---
  i64      i64         i64         i64      i64         i64     i64

  158      190         391         72       55          402     673
```

Existem duas possíveis abordagens para resolver isso: *oversampling* e *undersampling*. De maneira ingênua, podemos assumir que a última técnica desprezaria dados, nos fazendo perder informação. Vamos então adotar o *oversampling* com auxílio da biblioteca imbalanced-learn.

[52]:
```python
X = pl.concat([X_continuous, X_categorical], how = "horizontal")
```

Vamos salvar nossos dados antes do balanceamento para posteriormente demostrarmos a importância dessa etapa:

[53]:
```python
X_unbalanced = X
Y_unbalanced = Y
```

[54]:
```python
from imblearn.over_sampling import SMOTE, ADASYN, SMOTENC

# smote_nc = SMOTENC(categorical_features = [21, 22])
# X_over, Y_over = smote_nc.fit_resample(X.to_numpy(), Y.to_numpy())

X_over, Y_over = SMOTE().fit_resample(X.to_numpy(), Y.to_numpy())
```

[55]:
```python
Y = pl.DataFrame(Y_over, schema = Y.columns)
X = pl.DataFrame(X_over, schema = X.columns)

Y.sum()
```

[55]:
```
shape: (1, 7)

  Pastry   Z_Scratch   K_Scratch   Stains   Dirtiness   Bumps   Other_Faults
  ---      ---         ---         ---      ---         ---     ---
  i64      i64         i64         i64      i64         i64     i64
```

```
 673      673         673         673      673         673      673
```

Particionando em 70% a proporção para treino e teste. Repare no argumento `shuffle` como `True` que garante que haverá embaralhamento dos dados, que inicialmente estavam organizados por falha (primeiro todas as falhas do tipo *Pastry*, depois do tipo *Z_Scratch*, etc):

```
[56]: from sklearn.model_selection import train_test_split

      train_fraction = 0.7
      X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = (1 -␣
       ↪train_fraction), shuffle = True)
```

### 1.0.4 *4. Implemente uma rede neural MLP capaz de receber as variáveis de entrada que descrevem a geometria do defeito na placa e indicar a probabilidade de existência de cada possível falha.*

Um modelo simples de MLP que é capaz de receber as variáveis e gerar o rótulo correspondente é:

```
[57]: from tensorflow import keras
      from keras.models import Sequential
      from keras.layers import Flatten
      from keras.layers import Dense
      from keras.optimizers import Adam

      keras.backend.clear_session()

      model = Sequential()
      model.add(Dense(200, activation='relu')),
      model.add(Dense(50, activation='relu')),
      model.add(Dense(10, activation='relu')),
      model.add(Dense(Y_train.width, activation='softmax'))
```

```
[58]: optimizer = Adam()
      model.compile(optimizer=optimizer,
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

```
[59]: history = model.fit(
          X_train.to_numpy(),
          Y_train.to_numpy(),
          epochs = 128,
          batch_size = 32,
          validation_data = (X_test.to_numpy(), Y_test.to_numpy()))
```

```
Epoch 1/128
104/104              1s 3ms/step -
```

```
accuracy: 0.4824 - loss: 1.5507 - val_accuracy: 0.7256 - val_loss: 0.8164
Epoch 2/128
104/104            0s 1ms/step -
accuracy: 0.7736 - loss: 0.7199 - val_accuracy: 0.8154 - val_loss: 0.5819
Epoch 3/128
104/104            0s 2ms/step -
accuracy: 0.8176 - loss: 0.5444 - val_accuracy: 0.8161 - val_loss: 0.5626
Epoch 4/128
104/104            0s 1ms/step -
accuracy: 0.8255 - loss: 0.5044 - val_accuracy: 0.8303 - val_loss: 0.4821
Epoch 5/128
104/104            0s 1ms/step -
accuracy: 0.8535 - loss: 0.4042 - val_accuracy: 0.8423 - val_loss: 0.4407
Epoch 6/128
104/104            0s 1ms/step -
accuracy: 0.8625 - loss: 0.3679 - val_accuracy: 0.8501 - val_loss: 0.4307
Epoch 7/128
104/104            0s 1ms/step -
accuracy: 0.8758 - loss: 0.3555 - val_accuracy: 0.8444 - val_loss: 0.4208
Epoch 8/128
104/104            0s 1ms/step -
accuracy: 0.8774 - loss: 0.3267 - val_accuracy: 0.8557 - val_loss: 0.4004
Epoch 9/128
104/104            0s 1ms/step -
accuracy: 0.8696 - loss: 0.3573 - val_accuracy: 0.8586 - val_loss: 0.4058
Epoch 10/128
104/104            0s 1ms/step -
accuracy: 0.8897 - loss: 0.3108 - val_accuracy: 0.8628 - val_loss: 0.3721
Epoch 11/128
104/104            0s 2ms/step -
accuracy: 0.8960 - loss: 0.2870 - val_accuracy: 0.8593 - val_loss: 0.3834
Epoch 12/128
104/104            0s 2ms/step -
accuracy: 0.8996 - loss: 0.2914 - val_accuracy: 0.8621 - val_loss: 0.3731
Epoch 13/128
104/104            0s 2ms/step -
accuracy: 0.8975 - loss: 0.2782 - val_accuracy: 0.8755 - val_loss: 0.3520
Epoch 14/128
104/104            0s 1ms/step -
accuracy: 0.8995 - loss: 0.2652 - val_accuracy: 0.8762 - val_loss: 0.3556
Epoch 15/128
104/104            0s 2ms/step -
accuracy: 0.9094 - loss: 0.2674 - val_accuracy: 0.8649 - val_loss: 0.3900
Epoch 16/128
104/104            0s 2ms/step -
accuracy: 0.8976 - loss: 0.2734 - val_accuracy: 0.8564 - val_loss: 0.3916
Epoch 17/128
104/104            0s 1ms/step -
```

```
accuracy: 0.8718 - loss: 0.3442 - val_accuracy: 0.8777 - val_loss: 0.3496
Epoch 18/128
104/104            0s 2ms/step -
accuracy: 0.9168 - loss: 0.2285 - val_accuracy: 0.8713 - val_loss: 0.3903
Epoch 19/128
104/104            0s 1ms/step -
accuracy: 0.8950 - loss: 0.3064 - val_accuracy: 0.8833 - val_loss: 0.3404
Epoch 20/128
104/104            0s 2ms/step -
accuracy: 0.9199 - loss: 0.2275 - val_accuracy: 0.8847 - val_loss: 0.3503
Epoch 21/128
104/104            0s 1ms/step -
accuracy: 0.9162 - loss: 0.2169 - val_accuracy: 0.8840 - val_loss: 0.3389
Epoch 22/128
104/104            0s 2ms/step -
accuracy: 0.9324 - loss: 0.2027 - val_accuracy: 0.8911 - val_loss: 0.3320
Epoch 23/128
104/104            0s 2ms/step -
accuracy: 0.9178 - loss: 0.2221 - val_accuracy: 0.8911 - val_loss: 0.3261
Epoch 24/128
104/104            0s 2ms/step -
accuracy: 0.9319 - loss: 0.1912 - val_accuracy: 0.8897 - val_loss: 0.3355
Epoch 25/128
104/104            0s 2ms/step -
accuracy: 0.9281 - loss: 0.1924 - val_accuracy: 0.8868 - val_loss: 0.3421
Epoch 26/128
104/104            0s 2ms/step -
accuracy: 0.9346 - loss: 0.1748 - val_accuracy: 0.8946 - val_loss: 0.3396
Epoch 27/128
104/104            0s 2ms/step -
accuracy: 0.9410 - loss: 0.1691 - val_accuracy: 0.8847 - val_loss: 0.3684
Epoch 28/128
104/104            0s 1ms/step -
accuracy: 0.9278 - loss: 0.1908 - val_accuracy: 0.8996 - val_loss: 0.3211
Epoch 29/128
104/104            0s 1ms/step -
accuracy: 0.9460 - loss: 0.1569 - val_accuracy: 0.8953 - val_loss: 0.3332
Epoch 30/128
104/104            0s 2ms/step -
accuracy: 0.9509 - loss: 0.1549 - val_accuracy: 0.8748 - val_loss: 0.3699
Epoch 31/128
104/104            0s 2ms/step -
accuracy: 0.9339 - loss: 0.1757 - val_accuracy: 0.9038 - val_loss: 0.3262
Epoch 32/128
104/104            0s 1ms/step -
accuracy: 0.9510 - loss: 0.1524 - val_accuracy: 0.8812 - val_loss: 0.3463
Epoch 33/128
104/104            0s 2ms/step -
```

```
accuracy: 0.9428 - loss: 0.1544 - val_accuracy: 0.8982 - val_loss: 0.3402
Epoch 34/128
104/104            0s 2ms/step -
accuracy: 0.9600 - loss: 0.1423 - val_accuracy: 0.8925 - val_loss: 0.3396
Epoch 35/128
104/104            0s 2ms/step -
accuracy: 0.9577 - loss: 0.1305 - val_accuracy: 0.8967 - val_loss: 0.3335
Epoch 36/128
104/104            0s 2ms/step -
accuracy: 0.9522 - loss: 0.1363 - val_accuracy: 0.8960 - val_loss: 0.3321
Epoch 37/128
104/104            0s 2ms/step -
accuracy: 0.9482 - loss: 0.1380 - val_accuracy: 0.8939 - val_loss: 0.3731
Epoch 38/128
104/104            0s 2ms/step -
accuracy: 0.9492 - loss: 0.1426 - val_accuracy: 0.8918 - val_loss: 0.3705
Epoch 39/128
104/104            0s 2ms/step -
accuracy: 0.9431 - loss: 0.1662 - val_accuracy: 0.8678 - val_loss: 0.4032
Epoch 40/128
104/104            0s 2ms/step -
accuracy: 0.9160 - loss: 0.2439 - val_accuracy: 0.8960 - val_loss: 0.3472
Epoch 41/128
104/104            0s 2ms/step -
accuracy: 0.9648 - loss: 0.1113 - val_accuracy: 0.8967 - val_loss: 0.3483
Epoch 42/128
104/104            0s 2ms/step -
accuracy: 0.9574 - loss: 0.1214 - val_accuracy: 0.9010 - val_loss: 0.3517
Epoch 43/128
104/104            0s 1ms/step -
accuracy: 0.9626 - loss: 0.1178 - val_accuracy: 0.9010 - val_loss: 0.3315
Epoch 44/128
104/104            0s 2ms/step -
accuracy: 0.9748 - loss: 0.0951 - val_accuracy: 0.8996 - val_loss: 0.3419
Epoch 45/128
104/104            0s 2ms/step -
accuracy: 0.9610 - loss: 0.1117 - val_accuracy: 0.9045 - val_loss: 0.3437
Epoch 46/128
104/104            0s 2ms/step -
accuracy: 0.9731 - loss: 0.0963 - val_accuracy: 0.8918 - val_loss: 0.3571
Epoch 47/128
104/104            0s 2ms/step -
accuracy: 0.9713 - loss: 0.0937 - val_accuracy: 0.8996 - val_loss: 0.3496
Epoch 48/128
104/104            0s 2ms/step -
accuracy: 0.9730 - loss: 0.0858 - val_accuracy: 0.9010 - val_loss: 0.3824
Epoch 49/128
104/104            0s 2ms/step -
```

```
accuracy: 0.9715 - loss: 0.0907 - val_accuracy: 0.8890 - val_loss: 0.3994
Epoch 50/128
104/104            0s 2ms/step -
accuracy: 0.9578 - loss: 0.1196 - val_accuracy: 0.9017 - val_loss: 0.3429
Epoch 51/128
104/104            0s 2ms/step -
accuracy: 0.9784 - loss: 0.0789 - val_accuracy: 0.8996 - val_loss: 0.3550
Epoch 52/128
104/104            0s 2ms/step -
accuracy: 0.9764 - loss: 0.0784 - val_accuracy: 0.9088 - val_loss: 0.3567
Epoch 53/128
104/104            0s 2ms/step -
accuracy: 0.9791 - loss: 0.0702 - val_accuracy: 0.9095 - val_loss: 0.3535
Epoch 54/128
104/104            0s 2ms/step -
accuracy: 0.9790 - loss: 0.0754 - val_accuracy: 0.9081 - val_loss: 0.3625
Epoch 55/128
104/104            0s 2ms/step -
accuracy: 0.9775 - loss: 0.0763 - val_accuracy: 0.9045 - val_loss: 0.3536
Epoch 56/128
104/104            0s 1ms/step -
accuracy: 0.9749 - loss: 0.0728 - val_accuracy: 0.9024 - val_loss: 0.3780
Epoch 57/128
104/104            0s 1ms/step -
accuracy: 0.9801 - loss: 0.0707 - val_accuracy: 0.8918 - val_loss: 0.3928
Epoch 58/128
104/104            0s 2ms/step -
accuracy: 0.9406 - loss: 0.1726 - val_accuracy: 0.8996 - val_loss: 0.3969
Epoch 59/128
104/104            0s 2ms/step -
accuracy: 0.9794 - loss: 0.0702 - val_accuracy: 0.9010 - val_loss: 0.4126
Epoch 60/128
104/104            0s 2ms/step -
accuracy: 0.9756 - loss: 0.0793 - val_accuracy: 0.8989 - val_loss: 0.4238
Epoch 61/128
104/104            0s 2ms/step -
accuracy: 0.9784 - loss: 0.0629 - val_accuracy: 0.9081 - val_loss: 0.3889
Epoch 62/128
104/104            0s 2ms/step -
accuracy: 0.9797 - loss: 0.0627 - val_accuracy: 0.8960 - val_loss: 0.4228
Epoch 63/128
104/104            0s 1ms/step -
accuracy: 0.9829 - loss: 0.0549 - val_accuracy: 0.8953 - val_loss: 0.4191
Epoch 64/128
104/104            0s 2ms/step -
accuracy: 0.9851 - loss: 0.0541 - val_accuracy: 0.9059 - val_loss: 0.4194
Epoch 65/128
104/104            0s 1ms/step -
```

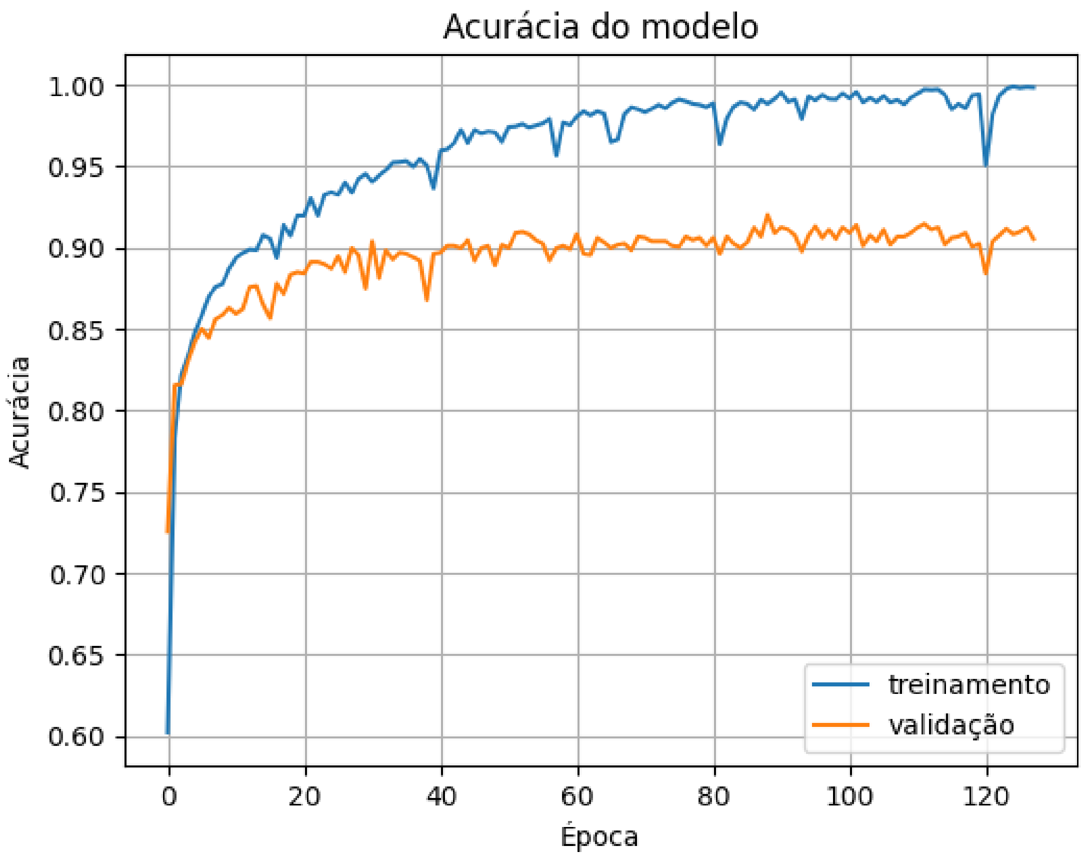
```
accuracy: 0.9865 - loss: 0.0461 - val_accuracy: 0.9031 - val_loss: 0.4011
Epoch 66/128
104/104            0s 1ms/step -
accuracy: 0.9610 - loss: 0.1189 - val_accuracy: 0.8996 - val_loss: 0.4073
Epoch 67/128
104/104            0s 2ms/step -
accuracy: 0.9528 - loss: 0.1543 - val_accuracy: 0.9017 - val_loss: 0.4106
Epoch 68/128
104/104            0s 1ms/step -
accuracy: 0.9806 - loss: 0.0552 - val_accuracy: 0.9024 - val_loss: 0.4038
Epoch 69/128
104/104            0s 1ms/step -
accuracy: 0.9871 - loss: 0.0467 - val_accuracy: 0.8982 - val_loss: 0.4386
Epoch 70/128
104/104            0s 2ms/step -
accuracy: 0.9875 - loss: 0.0448 - val_accuracy: 0.9066 - val_loss: 0.4200
Epoch 71/128
104/104            0s 1ms/step -
accuracy: 0.9814 - loss: 0.0532 - val_accuracy: 0.9059 - val_loss: 0.4079
Epoch 72/128
104/104            0s 2ms/step -
accuracy: 0.9885 - loss: 0.0457 - val_accuracy: 0.9038 - val_loss: 0.4169
Epoch 73/128
104/104            0s 1ms/step -
accuracy: 0.9881 - loss: 0.0437 - val_accuracy: 0.9038 - val_loss: 0.4307
Epoch 74/128
104/104            0s 2ms/step -
accuracy: 0.9879 - loss: 0.0482 - val_accuracy: 0.9038 - val_loss: 0.4282
Epoch 75/128
104/104            0s 2ms/step -
accuracy: 0.9900 - loss: 0.0474 - val_accuracy: 0.9010 - val_loss: 0.4333
Epoch 76/128
104/104            0s 2ms/step -
accuracy: 0.9918 - loss: 0.0404 - val_accuracy: 0.9003 - val_loss: 0.4503
Epoch 77/128
104/104            0s 2ms/step -
accuracy: 0.9905 - loss: 0.0388 - val_accuracy: 0.9066 - val_loss: 0.4555
Epoch 78/128
104/104            0s 2ms/step -
accuracy: 0.9904 - loss: 0.0357 - val_accuracy: 0.9045 - val_loss: 0.4521
Epoch 79/128
104/104            0s 1ms/step -
accuracy: 0.9873 - loss: 0.0470 - val_accuracy: 0.9059 - val_loss: 0.4474
Epoch 80/128
104/104            0s 1ms/step -
accuracy: 0.9850 - loss: 0.0500 - val_accuracy: 0.9010 - val_loss: 0.4742
Epoch 81/128
104/104            0s 2ms/step -
```

```
accuracy: 0.9892 - loss: 0.0343 - val_accuracy: 0.9059 - val_loss: 0.4552
Epoch 82/128
104/104            0s 2ms/step -
accuracy: 0.9575 - loss: 0.1261 - val_accuracy: 0.8960 - val_loss: 0.5016
Epoch 83/128
104/104            0s 2ms/step -
accuracy: 0.9720 - loss: 0.0664 - val_accuracy: 0.9066 - val_loss: 0.4533
Epoch 84/128
104/104            0s 2ms/step -
accuracy: 0.9862 - loss: 0.0430 - val_accuracy: 0.9024 - val_loss: 0.4681
Epoch 85/128
104/104            0s 2ms/step -
accuracy: 0.9892 - loss: 0.0359 - val_accuracy: 0.8996 - val_loss: 0.4928
Epoch 86/128
104/104            0s 1ms/step -
accuracy: 0.9909 - loss: 0.0328 - val_accuracy: 0.9031 - val_loss: 0.4879
Epoch 87/128
104/104            0s 1ms/step -
accuracy: 0.9786 - loss: 0.0561 - val_accuracy: 0.9123 - val_loss: 0.4795
Epoch 88/128
104/104            0s 2ms/step -
accuracy: 0.9896 - loss: 0.0370 - val_accuracy: 0.9066 - val_loss: 0.4756
Epoch 89/128
104/104            0s 2ms/step -
accuracy: 0.9838 - loss: 0.0462 - val_accuracy: 0.9201 - val_loss: 0.4738
Epoch 90/128
104/104            0s 2ms/step -
accuracy: 0.9903 - loss: 0.0320 - val_accuracy: 0.9088 - val_loss: 0.4950
Epoch 91/128
104/104            0s 2ms/step -
accuracy: 0.9937 - loss: 0.0301 - val_accuracy: 0.9123 - val_loss: 0.4662
Epoch 92/128
104/104            0s 2ms/step -
accuracy: 0.9910 - loss: 0.0305 - val_accuracy: 0.9109 - val_loss: 0.4935
Epoch 93/128
104/104            0s 2ms/step -
accuracy: 0.9904 - loss: 0.0268 - val_accuracy: 0.9074 - val_loss: 0.4928
Epoch 94/128
104/104            0s 2ms/step -
accuracy: 0.9728 - loss: 0.0724 - val_accuracy: 0.8975 - val_loss: 0.5042
Epoch 95/128
104/104            0s 2ms/step -
accuracy: 0.9934 - loss: 0.0281 - val_accuracy: 0.9074 - val_loss: 0.4893
Epoch 96/128
104/104            0s 2ms/step -
accuracy: 0.9920 - loss: 0.0290 - val_accuracy: 0.9130 - val_loss: 0.4947
Epoch 97/128
104/104            0s 2ms/step -
```

```
accuracy: 0.9948 - loss: 0.0234 - val_accuracy: 0.9059 - val_loss: 0.5050
Epoch 98/128
104/104            0s 2ms/step -
accuracy: 0.9937 - loss: 0.0270 - val_accuracy: 0.9109 - val_loss: 0.4990
Epoch 99/128
104/104            0s 2ms/step -
accuracy: 0.9891 - loss: 0.0347 - val_accuracy: 0.9052 - val_loss: 0.5158
Epoch 100/128
104/104            0s 2ms/step -
accuracy: 0.9966 - loss: 0.0195 - val_accuracy: 0.9123 - val_loss: 0.5006
Epoch 101/128
104/104            0s 1ms/step -
accuracy: 0.9925 - loss: 0.0245 - val_accuracy: 0.9088 - val_loss: 0.5225
Epoch 102/128
104/104            0s 1ms/step -
accuracy: 0.9973 - loss: 0.0174 - val_accuracy: 0.9137 - val_loss: 0.5143
Epoch 103/128
104/104            0s 2ms/step -
accuracy: 0.9905 - loss: 0.0283 - val_accuracy: 0.9010 - val_loss: 0.5611
Epoch 104/128
104/104            0s 2ms/step -
accuracy: 0.9928 - loss: 0.0253 - val_accuracy: 0.9074 - val_loss: 0.5369
Epoch 105/128
104/104            0s 2ms/step -
accuracy: 0.9926 - loss: 0.0288 - val_accuracy: 0.9038 - val_loss: 0.5575
Epoch 106/128
104/104            0s 2ms/step -
accuracy: 0.9941 - loss: 0.0258 - val_accuracy: 0.9109 - val_loss: 0.5270
Epoch 107/128
104/104            0s 2ms/step -
accuracy: 0.9913 - loss: 0.0334 - val_accuracy: 0.9017 - val_loss: 0.5566
Epoch 108/128
104/104            0s 2ms/step -
accuracy: 0.9923 - loss: 0.0318 - val_accuracy: 0.9066 - val_loss: 0.5662
Epoch 109/128
104/104            0s 2ms/step -
accuracy: 0.9863 - loss: 0.0321 - val_accuracy: 0.9066 - val_loss: 0.5528
Epoch 110/128
104/104            0s 2ms/step -
accuracy: 0.9898 - loss: 0.0325 - val_accuracy: 0.9088 - val_loss: 0.5579
Epoch 111/128
104/104            0s 2ms/step -
accuracy: 0.9956 - loss: 0.0155 - val_accuracy: 0.9123 - val_loss: 0.5588
Epoch 112/128
104/104            0s 2ms/step -
accuracy: 0.9979 - loss: 0.0116 - val_accuracy: 0.9144 - val_loss: 0.5459
Epoch 113/128
104/104            0s 2ms/step -
```

```
accuracy: 0.9964 - loss: 0.0155 - val_accuracy: 0.9109 - val_loss: 0.5466
Epoch 114/128
104/104            0s 2ms/step -
accuracy: 0.9974 - loss: 0.0137 - val_accuracy: 0.9123 - val_loss: 0.5539
Epoch 115/128
104/104            0s 2ms/step -
accuracy: 0.9967 - loss: 0.0158 - val_accuracy: 0.9017 - val_loss: 0.5734
Epoch 116/128
104/104            0s 2ms/step -
accuracy: 0.9845 - loss: 0.0471 - val_accuracy: 0.9059 - val_loss: 0.5916
Epoch 117/128
104/104            0s 2ms/step -
accuracy: 0.9870 - loss: 0.0401 - val_accuracy: 0.9066 - val_loss: 0.6253
Epoch 118/128
104/104            0s 2ms/step -
accuracy: 0.9830 - loss: 0.0431 - val_accuracy: 0.9088 - val_loss: 0.5822
Epoch 119/128
104/104            0s 2ms/step -
accuracy: 0.9936 - loss: 0.0241 - val_accuracy: 0.9003 - val_loss: 0.6008
Epoch 120/128
104/104            0s 2ms/step -
accuracy: 0.9930 - loss: 0.0189 - val_accuracy: 0.9024 - val_loss: 0.6023
Epoch 121/128
104/104            0s 2ms/step -
accuracy: 0.9344 - loss: 0.3151 - val_accuracy: 0.8840 - val_loss: 0.5910
Epoch 122/128
104/104            0s 2ms/step -
accuracy: 0.9806 - loss: 0.0628 - val_accuracy: 0.9038 - val_loss: 0.5987
Epoch 123/128
104/104            0s 2ms/step -
accuracy: 0.9922 - loss: 0.0282 - val_accuracy: 0.9074 - val_loss: 0.5498
Epoch 124/128
104/104            0s 2ms/step -
accuracy: 0.9988 - loss: 0.0145 - val_accuracy: 0.9116 - val_loss: 0.5673
Epoch 125/128
104/104            0s 2ms/step -
accuracy: 0.9993 - loss: 0.0114 - val_accuracy: 0.9081 - val_loss: 0.5661
Epoch 126/128
104/104            0s 2ms/step -
accuracy: 0.9991 - loss: 0.0109 - val_accuracy: 0.9095 - val_loss: 0.5727
Epoch 127/128
104/104            0s 2ms/step -
accuracy: 0.9991 - loss: 0.0102 - val_accuracy: 0.9123 - val_loss: 0.5778
Epoch 128/128
104/104            0s 2ms/step -
accuracy: 0.9989 - loss: 0.0118 - val_accuracy: 0.9052 - val_loss: 0.5981
```

[60]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Acurácia do modelo')
plt.ylabel('Acurácia')
plt.xlabel('Época')
plt.legend(['treinamento', 'validação'], loc='lower right')
plt.grid()
plt.show()
```




### 1.0.5 *5. Apresente o resultado fazendo considerações sobre:*

***i. Como a complexidade do modelo impacta no desempenho?*** Um aumento da complexidade não parece fazer com que o modelo aumente sua acurácia no conjunto de validação, e ainda faz com que o *overfitting* aconteça mais rapidamente:

[61]:
```
model_complex = Sequential()
model_complex.add(Dense(1024, input_dim = X_train.width, activation='relu')),
model_complex.add(Dense(512, activation='relu')),
model_complex.add(Dense(256, activation='relu')),
model_complex.add(Dense(128, activation='relu')),
```

```python
model_complex.add(Dense(64, activation='relu')),
model_complex.add(Dense(32, activation='relu')),
model_complex.add(Dense(16, activation='relu')),
model_complex.add(Dense(Y_train.width, activation='softmax'))

optimizer = Adam()
model_complex.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_complex = model_complex.fit(
    X_train.to_numpy(),
    Y_train.to_numpy(),
    epochs = 32,
    batch_size = 32,
    validation_data = (X_test.to_numpy(), Y_test.to_numpy()))

plt.plot(history_complex.history['accuracy'])
plt.plot(history_complex.history['val_accuracy'])
plt.title('Acurácia do modelo')
plt.ylabel('Acurácia')
plt.xlabel('Época')
plt.legend(['treinamento', 'validação'], loc='lower right')
plt.grid()
plt.show()
```

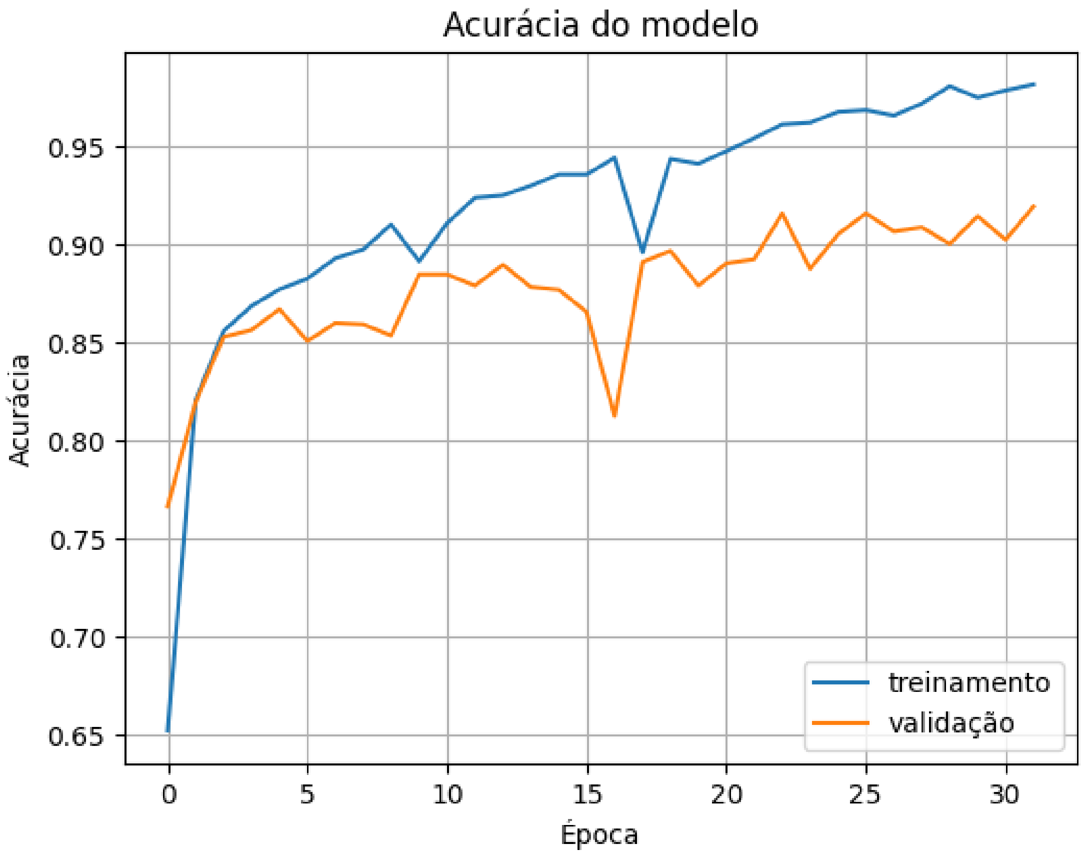
```
Epoch 1/32
```

```
/home/fbaltor/.pyenv/versions/3.11.10/lib/python3.11/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
104/104             2s 9ms/step -
accuracy: 0.5309 - loss: 1.2763 - val_accuracy: 0.7666 - val_loss: 0.6703
Epoch 2/32
104/104             1s 7ms/step -
accuracy: 0.8171 - loss: 0.5328 - val_accuracy: 0.8197 - val_loss: 0.5164
Epoch 3/32
104/104             1s 7ms/step -
accuracy: 0.8532 - loss: 0.4343 - val_accuracy: 0.8529 - val_loss: 0.4043
Epoch 4/32
104/104             1s 8ms/step -
accuracy: 0.8769 - loss: 0.3627 - val_accuracy: 0.8564 - val_loss: 0.4130
Epoch 5/32
104/104             1s 7ms/step -
accuracy: 0.8770 - loss: 0.3398 - val_accuracy: 0.8670 - val_loss: 0.3805
```

```
Epoch 6/32
104/104              1s 7ms/step -
accuracy: 0.8857 - loss: 0.3216 - val_accuracy: 0.8508 - val_loss: 0.4156
Epoch 7/32
104/104              1s 7ms/step -
accuracy: 0.8939 - loss: 0.2795 - val_accuracy: 0.8600 - val_loss: 0.4341
Epoch 8/32
104/104              1s 7ms/step -
accuracy: 0.8904 - loss: 0.2950 - val_accuracy: 0.8593 - val_loss: 0.4334
Epoch 9/32
104/104              1s 7ms/step -
accuracy: 0.9081 - loss: 0.2557 - val_accuracy: 0.8536 - val_loss: 0.4812
Epoch 10/32
104/104              1s 7ms/step -
accuracy: 0.8862 - loss: 0.2901 - val_accuracy: 0.8847 - val_loss: 0.3459
Epoch 11/32
104/104              1s 7ms/step -
accuracy: 0.9111 - loss: 0.2414 - val_accuracy: 0.8847 - val_loss: 0.3800
Epoch 12/32
104/104              1s 7ms/step -
accuracy: 0.9193 - loss: 0.2231 - val_accuracy: 0.8791 - val_loss: 0.4121
Epoch 13/32
104/104              1s 7ms/step -
accuracy: 0.9308 - loss: 0.2015 - val_accuracy: 0.8897 - val_loss: 0.3685
Epoch 14/32
104/104              1s 7ms/step -
accuracy: 0.9307 - loss: 0.1992 - val_accuracy: 0.8784 - val_loss: 0.3673
Epoch 15/32
104/104              1s 7ms/step -
accuracy: 0.9410 - loss: 0.1733 - val_accuracy: 0.8769 - val_loss: 0.4146
Epoch 16/32
104/104              1s 7ms/step -
accuracy: 0.9387 - loss: 0.1602 - val_accuracy: 0.8656 - val_loss: 0.4504
Epoch 17/32
104/104              1s 7ms/step -
accuracy: 0.9369 - loss: 0.1872 - val_accuracy: 0.8126 - val_loss: 0.6420
Epoch 18/32
104/104              1s 7ms/step -
accuracy: 0.8681 - loss: 0.4085 - val_accuracy: 0.8911 - val_loss: 0.3968
Epoch 19/32
104/104              1s 7ms/step -
accuracy: 0.9448 - loss: 0.1647 - val_accuracy: 0.8967 - val_loss: 0.3991
Epoch 20/32
104/104              1s 7ms/step -
accuracy: 0.9479 - loss: 0.1325 - val_accuracy: 0.8791 - val_loss: 0.4716
Epoch 21/32
104/104              1s 7ms/step -
accuracy: 0.9510 - loss: 0.1381 - val_accuracy: 0.8904 - val_loss: 0.3904
```

```
Epoch 22/32
104/104              1s 7ms/step -
accuracy: 0.9576 - loss: 0.1168 - val_accuracy: 0.8925 - val_loss: 0.3745
Epoch 23/32
104/104              1s 8ms/step -
accuracy: 0.9602 - loss: 0.1140 - val_accuracy: 0.9158 - val_loss: 0.3787
Epoch 24/32
104/104              1s 7ms/step -
accuracy: 0.9614 - loss: 0.1082 - val_accuracy: 0.8876 - val_loss: 0.5167
Epoch 25/32
104/104              1s 7ms/step -
accuracy: 0.9614 - loss: 0.1073 - val_accuracy: 0.9052 - val_loss: 0.4000
Epoch 26/32
104/104              1s 7ms/step -
accuracy: 0.9719 - loss: 0.0790 - val_accuracy: 0.9158 - val_loss: 0.4207
Epoch 27/32
104/104              1s 7ms/step -
accuracy: 0.9686 - loss: 0.0889 - val_accuracy: 0.9066 - val_loss: 0.3766
Epoch 28/32
104/104              1s 7ms/step -
accuracy: 0.9716 - loss: 0.0778 - val_accuracy: 0.9088 - val_loss: 0.4342
Epoch 29/32
104/104              1s 7ms/step -
accuracy: 0.9817 - loss: 0.0639 - val_accuracy: 0.9003 - val_loss: 0.4609
Epoch 30/32
104/104              1s 7ms/step -
accuracy: 0.9792 - loss: 0.0620 - val_accuracy: 0.9144 - val_loss: 0.4417
Epoch 31/32
104/104              1s 7ms/step -
accuracy: 0.9841 - loss: 0.0574 - val_accuracy: 0.9024 - val_loss: 0.4987
Epoch 32/32
104/104              1s 7ms/step -
accuracy: 0.9849 - loss: 0.0462 - val_accuracy: 0.9194 - val_loss: 0.4734
```




***ii. Comente sobre a estratégia de ajuste dos hiper parâmetros adotada para ajustar o modelo.***

- O *learning rate* foi deixado em seu valor padrão.
- A complexidade (número de camadas e neurônios de cada camada) foi deixada baixa porque o modelo facilmente apresenta *overfitting*.
- O número de épocas não precisou ser alto (32) dado que rapidamente o treinamento atinge um bom valor de acurácia, ao passo que valores maiores também causam *overfitting*.
- Foram testadas algumas outras funções de ativação (`tanh`, `softmax` em todas as camadas, etc), mas elas não pareceram ser tão determinantes no resultado do treinamento.

***iii. Como a quantidade de amostras de cada classe pode influenciar no desempenho da rede neural?*** A quantidade de amostras de cada classe e seu balanceamento em especial se mostrou extremamente importante para o bom desempenho da rede neural. Abaixo segue o treinamento do modelo com os dados desbalanceados:

[62]:
```
train_fraction = 0.7
X_train, X_test, Y_train, Y_test = train_test_split(X_unbalanced, Y_unbalanced,␣
↪test_size = (1 - train_fraction), shuffle = True)
```

```python
model_unb = Sequential()
model_unb.add(Dense(200, activation='relu')),
model_unb.add(Dense(50, activation='relu')),
model_unb.add(Dense(10, activation='relu')),
model_unb.add(Dense(Y_train.width, activation='softmax'))

optimizer = Adam()
model_unb.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_unb = model_unb.fit(
    X_train.to_numpy(),
    Y_train.to_numpy(),
    epochs = 128,
    batch_size = 32,
    validation_data = (X_test.to_numpy(), Y_test.to_numpy()))

plt.plot(history_unb.history['accuracy'])
plt.plot(history_unb.history['val_accuracy'])
plt.title('Acurácia do modelo')
plt.ylabel('Acurácia')
plt.xlabel('Época')
plt.legend(['treinamento', 'validação'], loc='lower right')
plt.grid()
plt.show()
```

```
Epoch 1/128
43/43            1s 5ms/step -
accuracy: 0.3226 - loss: 1.7882 - val_accuracy: 0.5506 - val_loss: 1.3424
Epoch 2/128
43/43            0s 2ms/step -
accuracy: 0.5232 - loss: 1.3009 - val_accuracy: 0.6261 - val_loss: 1.0726
Epoch 3/128
43/43            0s 2ms/step -
accuracy: 0.6419 - loss: 1.0167 - val_accuracy: 0.6587 - val_loss: 0.9640
Epoch 4/128
43/43            0s 2ms/step -
accuracy: 0.6427 - loss: 0.9565 - val_accuracy: 0.6690 - val_loss: 0.9017
Epoch 5/128
43/43            0s 2ms/step -
accuracy: 0.6946 - loss: 0.8288 - val_accuracy: 0.6844 - val_loss: 0.8661
Epoch 6/128
43/43            0s 2ms/step -
accuracy: 0.6963 - loss: 0.7920 - val_accuracy: 0.6913 - val_loss: 0.8385
Epoch 7/128
43/43            0s 2ms/step -
```

```
accuracy: 0.7041 - loss: 0.7618 - val_accuracy: 0.6913 - val_loss: 0.8207
Epoch 8/128
43/43            0s 2ms/step -
accuracy: 0.7213 - loss: 0.7384 - val_accuracy: 0.7067 - val_loss: 0.7921
Epoch 9/128
43/43            0s 2ms/step -
accuracy: 0.7479 - loss: 0.6574 - val_accuracy: 0.7204 - val_loss: 0.7859
Epoch 10/128
43/43            0s 2ms/step -
accuracy: 0.7536 - loss: 0.6755 - val_accuracy: 0.6913 - val_loss: 0.7842
Epoch 11/128
43/43            0s 2ms/step -
accuracy: 0.7705 - loss: 0.6409 - val_accuracy: 0.7376 - val_loss: 0.7469
Epoch 12/128
43/43            0s 2ms/step -
accuracy: 0.7760 - loss: 0.6132 - val_accuracy: 0.7358 - val_loss: 0.7153
Epoch 13/128
43/43            0s 2ms/step -
accuracy: 0.7876 - loss: 0.5843 - val_accuracy: 0.7256 - val_loss: 0.7448
Epoch 14/128
43/43            0s 2ms/step -
accuracy: 0.7875 - loss: 0.5765 - val_accuracy: 0.7256 - val_loss: 0.7560
Epoch 15/128
43/43            0s 2ms/step -
accuracy: 0.8031 - loss: 0.5477 - val_accuracy: 0.7238 - val_loss: 0.7247
Epoch 16/128
43/43            0s 2ms/step -
accuracy: 0.7936 - loss: 0.5488 - val_accuracy: 0.7427 - val_loss: 0.7031
Epoch 17/128
43/43            0s 2ms/step -
accuracy: 0.7918 - loss: 0.5589 - val_accuracy: 0.7324 - val_loss: 0.6958
Epoch 18/128
43/43            0s 2ms/step -
accuracy: 0.8003 - loss: 0.5130 - val_accuracy: 0.7033 - val_loss: 0.7492
Epoch 19/128
43/43            0s 2ms/step -
accuracy: 0.8337 - loss: 0.4723 - val_accuracy: 0.7479 - val_loss: 0.6996
Epoch 20/128
43/43            0s 3ms/step -
accuracy: 0.8174 - loss: 0.4755 - val_accuracy: 0.7273 - val_loss: 0.6981
Epoch 21/128
43/43            0s 2ms/step -
accuracy: 0.8270 - loss: 0.4859 - val_accuracy: 0.7376 - val_loss: 0.6983
Epoch 22/128
43/43            0s 2ms/step -
accuracy: 0.8240 - loss: 0.4509 - val_accuracy: 0.7496 - val_loss: 0.6958
Epoch 23/128
43/43            0s 2ms/step -
```

```
accuracy: 0.8417 - loss: 0.4484 - val_accuracy: 0.7479 - val_loss: 0.7122
Epoch 24/128
43/43            0s 2ms/step -
accuracy: 0.8457 - loss: 0.4184 - val_accuracy: 0.7290 - val_loss: 0.7119
Epoch 25/128
43/43            0s 2ms/step -
accuracy: 0.8371 - loss: 0.4315 - val_accuracy: 0.7221 - val_loss: 0.7407
Epoch 26/128
43/43            0s 2ms/step -
accuracy: 0.8564 - loss: 0.4162 - val_accuracy: 0.7358 - val_loss: 0.7151
Epoch 27/128
43/43            0s 2ms/step -
accuracy: 0.8447 - loss: 0.4110 - val_accuracy: 0.7427 - val_loss: 0.7057
Epoch 28/128
43/43            0s 2ms/step -
accuracy: 0.8606 - loss: 0.3798 - val_accuracy: 0.7513 - val_loss: 0.7042
Epoch 29/128
43/43            0s 2ms/step -
accuracy: 0.8582 - loss: 0.3933 - val_accuracy: 0.7581 - val_loss: 0.6941
Epoch 30/128
43/43            0s 2ms/step -
accuracy: 0.8645 - loss: 0.3733 - val_accuracy: 0.7513 - val_loss: 0.7069
Epoch 31/128
43/43            0s 2ms/step -
accuracy: 0.8625 - loss: 0.3900 - val_accuracy: 0.7461 - val_loss: 0.7105
Epoch 32/128
43/43            0s 2ms/step -
accuracy: 0.8764 - loss: 0.3440 - val_accuracy: 0.7015 - val_loss: 0.8082
Epoch 33/128
43/43            0s 2ms/step -
accuracy: 0.8480 - loss: 0.4086 - val_accuracy: 0.7341 - val_loss: 0.7143
Epoch 34/128
43/43            0s 2ms/step -
accuracy: 0.8759 - loss: 0.3379 - val_accuracy: 0.7513 - val_loss: 0.7258
Epoch 35/128
43/43            0s 2ms/step -
accuracy: 0.8581 - loss: 0.3672 - val_accuracy: 0.7393 - val_loss: 0.7283
Epoch 36/128
43/43            0s 2ms/step -
accuracy: 0.8804 - loss: 0.3286 - val_accuracy: 0.7410 - val_loss: 0.7216
Epoch 37/128
43/43            0s 2ms/step -
accuracy: 0.8602 - loss: 0.3475 - val_accuracy: 0.7410 - val_loss: 0.7784
Epoch 38/128
43/43            0s 2ms/step -
accuracy: 0.8815 - loss: 0.3340 - val_accuracy: 0.7564 - val_loss: 0.7284
Epoch 39/128
43/43            0s 2ms/step -
```

```
accuracy: 0.8718 - loss: 0.3353 - val_accuracy: 0.7427 - val_loss: 0.7664
Epoch 40/128
43/43            0s 2ms/step -
accuracy: 0.8805 - loss: 0.3079 - val_accuracy: 0.7256 - val_loss: 0.8076
Epoch 41/128
43/43            0s 2ms/step -
accuracy: 0.8810 - loss: 0.3140 - val_accuracy: 0.7410 - val_loss: 0.7567
Epoch 42/128
43/43            0s 2ms/step -
accuracy: 0.9043 - loss: 0.2904 - val_accuracy: 0.7444 - val_loss: 0.7748
Epoch 43/128
43/43            0s 2ms/step -
accuracy: 0.8950 - loss: 0.3011 - val_accuracy: 0.7256 - val_loss: 0.8031
Epoch 44/128
43/43            0s 2ms/step -
accuracy: 0.8770 - loss: 0.3447 - val_accuracy: 0.7479 - val_loss: 0.7744
Epoch 45/128
43/43            0s 2ms/step -
accuracy: 0.8907 - loss: 0.2898 - val_accuracy: 0.7376 - val_loss: 0.8016
Epoch 46/128
43/43            0s 2ms/step -
accuracy: 0.9043 - loss: 0.2797 - val_accuracy: 0.7513 - val_loss: 0.7982
Epoch 47/128
43/43            0s 2ms/step -
accuracy: 0.9076 - loss: 0.2771 - val_accuracy: 0.7444 - val_loss: 0.7923
Epoch 48/128
43/43            0s 2ms/step -
accuracy: 0.9239 - loss: 0.2418 - val_accuracy: 0.7324 - val_loss: 0.8137
Epoch 49/128
43/43            0s 2ms/step -
accuracy: 0.9198 - loss: 0.2405 - val_accuracy: 0.7307 - val_loss: 0.8159
Epoch 50/128
43/43            0s 2ms/step -
accuracy: 0.9244 - loss: 0.2352 - val_accuracy: 0.7290 - val_loss: 0.7956
Epoch 51/128
43/43            0s 2ms/step -
accuracy: 0.9078 - loss: 0.2505 - val_accuracy: 0.7599 - val_loss: 0.8369
Epoch 52/128
43/43            0s 2ms/step -
accuracy: 0.9171 - loss: 0.2580 - val_accuracy: 0.7307 - val_loss: 0.8638
Epoch 53/128
43/43            0s 2ms/step -
accuracy: 0.9068 - loss: 0.2606 - val_accuracy: 0.7410 - val_loss: 0.8481
Epoch 54/128
43/43            0s 2ms/step -
accuracy: 0.9106 - loss: 0.2371 - val_accuracy: 0.7393 - val_loss: 0.8461
Epoch 55/128
43/43            0s 3ms/step -
```

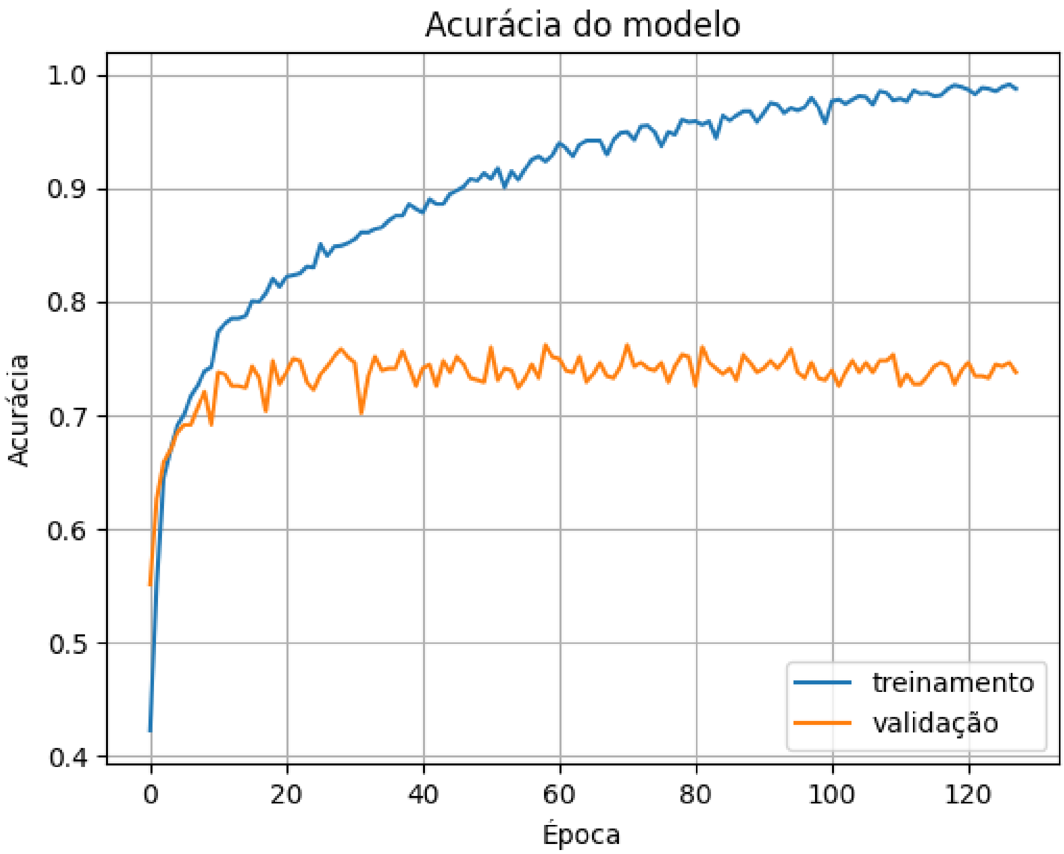
```
accuracy: 0.9257 - loss: 0.2253 - val_accuracy: 0.7238 - val_loss: 0.8672
Epoch 56/128
43/43            0s 2ms/step -
accuracy: 0.9115 - loss: 0.2616 - val_accuracy: 0.7324 - val_loss: 0.8661
Epoch 57/128
43/43            0s 2ms/step -
accuracy: 0.9410 - loss: 0.2011 - val_accuracy: 0.7444 - val_loss: 0.8474
Epoch 58/128
43/43            0s 2ms/step -
accuracy: 0.9233 - loss: 0.2323 - val_accuracy: 0.7324 - val_loss: 0.8842
Epoch 59/128
43/43            0s 2ms/step -
accuracy: 0.9268 - loss: 0.2017 - val_accuracy: 0.7616 - val_loss: 0.8517
Epoch 60/128
43/43            0s 2ms/step -
accuracy: 0.9364 - loss: 0.1942 - val_accuracy: 0.7513 - val_loss: 0.8841
Epoch 61/128
43/43            0s 2ms/step -
accuracy: 0.9366 - loss: 0.1983 - val_accuracy: 0.7496 - val_loss: 0.9224
Epoch 62/128
43/43            0s 2ms/step -
accuracy: 0.9383 - loss: 0.1985 - val_accuracy: 0.7393 - val_loss: 0.9013
Epoch 63/128
43/43            0s 2ms/step -
accuracy: 0.9318 - loss: 0.2033 - val_accuracy: 0.7376 - val_loss: 0.9106
Epoch 64/128
43/43            0s 2ms/step -
accuracy: 0.9434 - loss: 0.1806 - val_accuracy: 0.7513 - val_loss: 0.9144
Epoch 65/128
43/43            0s 2ms/step -
accuracy: 0.9529 - loss: 0.1641 - val_accuracy: 0.7290 - val_loss: 0.9320
Epoch 66/128
43/43            0s 2ms/step -
accuracy: 0.9509 - loss: 0.1689 - val_accuracy: 0.7358 - val_loss: 0.9360
Epoch 67/128
43/43            0s 2ms/step -
accuracy: 0.9509 - loss: 0.1694 - val_accuracy: 0.7461 - val_loss: 0.9404
Epoch 68/128
43/43            0s 2ms/step -
accuracy: 0.9312 - loss: 0.1803 - val_accuracy: 0.7341 - val_loss: 0.9368
Epoch 69/128
43/43            0s 2ms/step -
accuracy: 0.9421 - loss: 0.1722 - val_accuracy: 0.7324 - val_loss: 0.9790
Epoch 70/128
43/43            0s 2ms/step -
accuracy: 0.9537 - loss: 0.1615 - val_accuracy: 0.7427 - val_loss: 0.9615
Epoch 71/128
43/43            0s 2ms/step -
```

```
accuracy: 0.9590 - loss: 0.1398 - val_accuracy: 0.7616 - val_loss: 0.9311
Epoch 72/128
43/43            0s 2ms/step -
accuracy: 0.9480 - loss: 0.1493 - val_accuracy: 0.7427 - val_loss: 0.9914
Epoch 73/128
43/43            0s 2ms/step -
accuracy: 0.9520 - loss: 0.1637 - val_accuracy: 0.7461 - val_loss: 0.9855
Epoch 74/128
43/43            0s 2ms/step -
accuracy: 0.9607 - loss: 0.1377 - val_accuracy: 0.7410 - val_loss: 0.9964
Epoch 75/128
43/43            0s 2ms/step -
accuracy: 0.9596 - loss: 0.1381 - val_accuracy: 0.7393 - val_loss: 1.0230
Epoch 76/128
43/43            0s 2ms/step -
accuracy: 0.9490 - loss: 0.1516 - val_accuracy: 0.7461 - val_loss: 1.0385
Epoch 77/128
43/43            0s 3ms/step -
accuracy: 0.9506 - loss: 0.1492 - val_accuracy: 0.7290 - val_loss: 1.0482
Epoch 78/128
43/43            0s 4ms/step -
accuracy: 0.9447 - loss: 0.1539 - val_accuracy: 0.7427 - val_loss: 1.0549
Epoch 79/128
43/43            0s 3ms/step -
accuracy: 0.9702 - loss: 0.1251 - val_accuracy: 0.7530 - val_loss: 1.0325
Epoch 80/128
43/43            0s 3ms/step -
accuracy: 0.9618 - loss: 0.1307 - val_accuracy: 0.7513 - val_loss: 1.0254
Epoch 81/128
43/43            0s 3ms/step -
accuracy: 0.9576 - loss: 0.1278 - val_accuracy: 0.7256 - val_loss: 1.1121
Epoch 82/128
43/43            0s 3ms/step -
accuracy: 0.9570 - loss: 0.1370 - val_accuracy: 0.7599 - val_loss: 1.0917
Epoch 83/128
43/43            0s 3ms/step -
accuracy: 0.9621 - loss: 0.1151 - val_accuracy: 0.7461 - val_loss: 1.0856
Epoch 84/128
43/43            0s 3ms/step -
accuracy: 0.9512 - loss: 0.1389 - val_accuracy: 0.7410 - val_loss: 1.0946
Epoch 85/128
43/43            0s 3ms/step -
accuracy: 0.9654 - loss: 0.1136 - val_accuracy: 0.7358 - val_loss: 1.1389
Epoch 86/128
43/43            0s 3ms/step -
accuracy: 0.9619 - loss: 0.1141 - val_accuracy: 0.7410 - val_loss: 1.1435
Epoch 87/128
43/43            0s 3ms/step -
```

```
accuracy: 0.9678 - loss: 0.1005 - val_accuracy: 0.7307 - val_loss: 1.1410
Epoch 88/128
43/43            0s 3ms/step -
accuracy: 0.9700 - loss: 0.1087 - val_accuracy: 0.7530 - val_loss: 1.1279
Epoch 89/128
43/43            0s 2ms/step -
accuracy: 0.9735 - loss: 0.0988 - val_accuracy: 0.7461 - val_loss: 1.1329
Epoch 90/128
43/43            0s 2ms/step -
accuracy: 0.9647 - loss: 0.1011 - val_accuracy: 0.7376 - val_loss: 1.1983
Epoch 91/128
43/43            0s 3ms/step -
accuracy: 0.9689 - loss: 0.1002 - val_accuracy: 0.7410 - val_loss: 1.1373
Epoch 92/128
43/43            0s 3ms/step -
accuracy: 0.9742 - loss: 0.0964 - val_accuracy: 0.7479 - val_loss: 1.1706
Epoch 93/128
43/43            0s 4ms/step -
accuracy: 0.9769 - loss: 0.0867 - val_accuracy: 0.7410 - val_loss: 1.1779
Epoch 94/128
43/43            0s 3ms/step -
accuracy: 0.9670 - loss: 0.1152 - val_accuracy: 0.7479 - val_loss: 1.1886
Epoch 95/128
43/43            0s 3ms/step -
accuracy: 0.9737 - loss: 0.0970 - val_accuracy: 0.7581 - val_loss: 1.1832
Epoch 96/128
43/43            0s 3ms/step -
accuracy: 0.9746 - loss: 0.1042 - val_accuracy: 0.7376 - val_loss: 1.2407
Epoch 97/128
43/43            0s 3ms/step -
accuracy: 0.9771 - loss: 0.0922 - val_accuracy: 0.7324 - val_loss: 1.2466
Epoch 98/128
43/43            0s 3ms/step -
accuracy: 0.9816 - loss: 0.0884 - val_accuracy: 0.7461 - val_loss: 1.2517
Epoch 99/128
43/43            0s 3ms/step -
accuracy: 0.9734 - loss: 0.0872 - val_accuracy: 0.7324 - val_loss: 1.3254
Epoch 100/128
43/43            0s 3ms/step -
accuracy: 0.9567 - loss: 0.1140 - val_accuracy: 0.7307 - val_loss: 1.2896
Epoch 101/128
43/43            0s 2ms/step -
accuracy: 0.9772 - loss: 0.0891 - val_accuracy: 0.7393 - val_loss: 1.2625
Epoch 102/128
43/43            0s 2ms/step -
accuracy: 0.9798 - loss: 0.0937 - val_accuracy: 0.7256 - val_loss: 1.3064
Epoch 103/128
43/43            0s 3ms/step -
```

```
accuracy: 0.9780 - loss: 0.0795 - val_accuracy: 0.7376 - val_loss: 1.2794
Epoch 104/128
43/43            0s 2ms/step -
accuracy: 0.9770 - loss: 0.0725 - val_accuracy: 0.7479 - val_loss: 1.2851
Epoch 105/128
43/43            0s 3ms/step -
accuracy: 0.9798 - loss: 0.0774 - val_accuracy: 0.7376 - val_loss: 1.2928
Epoch 106/128
43/43            0s 3ms/step -
accuracy: 0.9845 - loss: 0.0670 - val_accuracy: 0.7461 - val_loss: 1.3410
Epoch 107/128
43/43            0s 5ms/step -
accuracy: 0.9789 - loss: 0.0694 - val_accuracy: 0.7376 - val_loss: 1.3023
Epoch 108/128
43/43            0s 3ms/step -
accuracy: 0.9865 - loss: 0.0644 - val_accuracy: 0.7479 - val_loss: 1.3215
Epoch 109/128
43/43            0s 3ms/step -
accuracy: 0.9856 - loss: 0.0689 - val_accuracy: 0.7479 - val_loss: 1.3904
Epoch 110/128
43/43            0s 3ms/step -
accuracy: 0.9723 - loss: 0.0791 - val_accuracy: 0.7530 - val_loss: 1.3569
Epoch 111/128
43/43            0s 3ms/step -
accuracy: 0.9794 - loss: 0.0719 - val_accuracy: 0.7256 - val_loss: 1.4651
Epoch 112/128
43/43            0s 2ms/step -
accuracy: 0.9773 - loss: 0.0805 - val_accuracy: 0.7358 - val_loss: 1.4272
Epoch 113/128
43/43            0s 3ms/step -
accuracy: 0.9905 - loss: 0.0668 - val_accuracy: 0.7273 - val_loss: 1.4057
Epoch 114/128
43/43            0s 3ms/step -
accuracy: 0.9777 - loss: 0.0673 - val_accuracy: 0.7273 - val_loss: 1.4098
Epoch 115/128
43/43            0s 3ms/step -
accuracy: 0.9892 - loss: 0.0573 - val_accuracy: 0.7341 - val_loss: 1.5103
Epoch 116/128
43/43            0s 3ms/step -
accuracy: 0.9701 - loss: 0.0840 - val_accuracy: 0.7427 - val_loss: 1.4440
Epoch 117/128
43/43            0s 3ms/step -
accuracy: 0.9813 - loss: 0.0741 - val_accuracy: 0.7461 - val_loss: 1.3822
Epoch 118/128
43/43            0s 3ms/step -
accuracy: 0.9887 - loss: 0.0563 - val_accuracy: 0.7427 - val_loss: 1.4268
Epoch 119/128
43/43            0s 4ms/step -
```

```
accuracy: 0.9920 - loss: 0.0549 - val_accuracy: 0.7273 - val_loss: 1.4671
Epoch 120/128
43/43            0s 3ms/step -
accuracy: 0.9904 - loss: 0.0500 - val_accuracy: 0.7393 - val_loss: 1.4386
Epoch 121/128
43/43            0s 3ms/step -
accuracy: 0.9899 - loss: 0.0521 - val_accuracy: 0.7461 - val_loss: 1.4628
Epoch 122/128
43/43            0s 3ms/step -
accuracy: 0.9842 - loss: 0.0579 - val_accuracy: 0.7341 - val_loss: 1.5026
Epoch 123/128
43/43            0s 3ms/step -
accuracy: 0.9896 - loss: 0.0540 - val_accuracy: 0.7341 - val_loss: 1.5182
Epoch 124/128
43/43            0s 3ms/step -
accuracy: 0.9852 - loss: 0.0524 - val_accuracy: 0.7324 - val_loss: 1.5532
Epoch 125/128
43/43            0s 3ms/step -
accuracy: 0.9857 - loss: 0.0575 - val_accuracy: 0.7444 - val_loss: 1.4930
Epoch 126/128
43/43            0s 3ms/step -
accuracy: 0.9899 - loss: 0.0466 - val_accuracy: 0.7427 - val_loss: 1.5143
Epoch 127/128
43/43            0s 3ms/step -
accuracy: 0.9902 - loss: 0.0444 - val_accuracy: 0.7461 - val_loss: 1.4919
Epoch 128/128
43/43            0s 3ms/step -
accuracy: 0.9856 - loss: 0.0496 - val_accuracy: 0.7376 - val_loss: 1.5692
```




A piora do seu desempenho é nítida, com o *overfitting* ainda mais presente e a acurácia estagnando quase 15% abaixo do melhor modelo obtido anteriormente.

***iv. Quais variáveis de entrada são mais relevantes para o problema?*** Vamos empregar uma técnica simples, obtida com ajuda do *Gepeto* e que consiste em calcular uma acurácia básica (*baseline*) com o modelo completo para então randomizar os valores de cada *feature* individualmente e calcular essa acurácia específica:

[72]:
```
from sklearn.metrics import accuracy_score

def permutation_importance(model, X_val, y_val):
    baseline = accuracy_score(y_val, model.predict(X_val).round())
    importances = []

    for col in range(X_val.shape[1]):
        X_temp = X_val.copy()
        np.random.shuffle(X_temp[:, col])  # Shuffle the feature column
        shuffled = accuracy_score(y_val, model.predict(X_temp).round())
        importances.append(baseline - shuffled)  # Calculate importance
```

```
    return importances

importances = permutation_importance(model, X.to_numpy(), Y.to_numpy())
```

```
148/148            0s 601us/step
148/148            0s 575us/step
148/148            0s 596us/step
148/148            0s 635us/step
148/148            0s 626us/step
148/148            0s 704us/step
148/148            0s 628us/step
148/148            0s 625us/step
148/148            0s 675us/step
148/148            0s 670us/step
148/148            0s 681us/step
148/148            0s 650us/step
148/148            0s 606us/step
148/148            0s 622us/step
148/148            0s 601us/step
148/148            0s 628us/step
148/148            0s 579us/step
148/148            0s 640us/step
148/148            0s 611us/step
148/148            0s 606us/step
148/148            0s 643us/step
148/148            0s 590us/step
148/148            0s 586us/step
148/148            0s 579us/step
```

[88]:
```
feature_importance = pl.DataFrame(
    {
        "feature": X.columns,
        "importance": importances
    }
)

feature_importance = feature_importance.sort(by = "importance").reverse()

with pl.Config(tbl_rows = -1):
    print(feature_importance)
```

```
shape: (23, 2)

 feature                  importance
 ---                      ---
 str                      f64

 Length_of_Conveyer       0.184037
```

```
Square_Index             0.165145
Log_Y_Index              0.157504
X_Minimum                0.132668
Edges_X_Index            0.130758
Maximum_of_Luminosity    0.1263
Y_Minimum                0.121842
SigmoidOfAreas           0.118022
Steel_Plate_Thickness    0.114201
Edges_Index              0.097644
Minimum_of_Luminosity    0.09637
TypeOfSteel_A300         0.091063
Log_X_Index              0.08703
Edges_Y_Index            0.081724
Empty_Index              0.077054
Orientation_Index        0.068987
Outside_Global_Index     0.065591
Luminosity_Index         0.063893
TypeOfSteel_A400         0.055827
LogOfAreas               0.050732
Outside_X_Index          0.021651
Y_Perimeter              0.010613
Pixels_Areas             0.009552
```

Percebemos que as variáveis de entrada mais importantes são `Length_of_Conveyer`, `Square_Index` e `Log_Y_Index`. Entretanto, a importância das variáveis parece decrescer gradualmente, de maneira que é difícil dizer que há um conjunto muito mais determinante que os demais.