P12_Faux_Billets

September 20, 2025

DÉTECTION DE FAUX BILLETS - DONNÉES ET MODÈLES

1 OBJECTIFS DE CE NOTEBOOK

- Récupérer les données de l'étude ;
- Renseigner les valeurs manquantes grâce à une régression linéaire ;
- Tester plusieurs modèles d'apprentissage et les évaluer.

Partie 1 - Importation des bibliothèques Python et chargement des fichiers

[1]:
```
#Importation de la librairie Pandas
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sb
import statsmodels.api as sm
import statsmodels.stats.diagnostic as ssd
from matplotlib import colormaps as cm

# Fonction pour régression linéaire
from sklearn.linear_model import LinearRegression

# Bibliothèque de tests statistiques
import scipy.stats as st

# Pour création des parties d'une liste
from itertools import chain, combinations
```

[2]:
```
# Normaliser les données
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Modèles d'apprentissage KMeans
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Modèle de régression logistique
```

```python
from sklearn.linear_model import LogisticRegression

# Bibliothèque pour modèle KNN
import sklearn.neighbors as sk_n

# Bibliothèque pour modèle forêt aléatoire
import sklearn.ensemble as sk_e

# Fonctions pour échantillonnage et métriques
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Enregistrer le modèle retenu
import joblib
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, ClassifierMixin
```

[3]:
```python
#Importation du fichier billets.csv
df_ini = pd.read_csv("billets.csv", sep=';')
```

Partie 2 - Analyse exploratoire

2.1 - Aperçu global

[4]:
```python
#Afficher les dimensions du dataset
print("Le tableau comporte {} observation(s) ou article(s)".format(df_ini.
 ↪shape[0]))
print("Le tableau comporte {} colonne(s)".format(df_ini.shape[1]))
```

```
Le tableau comporte 1500 observation(s) ou article(s)
Le tableau comporte 7 colonne(s)
```

[5]:
```python
#La nature des données dans chacune des colonnes
display(df_ini.dtypes)
#Le nombre de valeurs présentes dans chacune des colonnes
for c in list(df_ini):
    print("\nColonne", c, "- Nombre de valeurs NaN :", ((df_ini[c]).isna()).
 ↪sum())
    print("Colonne", c, "- Nombre de valeurs non-vides :", df_ini.shape[0] -␣
 ↪((df_ini[c]).isna()).sum())
```

```
is_genuine         bool
diagonal        float64
height_left     float64
height_right    float64
margin_low      float64
margin_up       float64
```

```
length          float64
dtype: object

Colonne is_genuine - Nombre de valeurs NaN : 0
Colonne is_genuine - Nombre de valeurs non-vides : 1500

Colonne diagonal - Nombre de valeurs NaN : 0
Colonne diagonal - Nombre de valeurs non-vides : 1500

Colonne height_left - Nombre de valeurs NaN : 0
Colonne height_left - Nombre de valeurs non-vides : 1500

Colonne height_right - Nombre de valeurs NaN : 0
Colonne height_right - Nombre de valeurs non-vides : 1500

Colonne margin_low - Nombre de valeurs NaN : 37
Colonne margin_low - Nombre de valeurs non-vides : 1463

Colonne margin_up - Nombre de valeurs NaN : 0
Colonne margin_up - Nombre de valeurs non-vides : 1500

Colonne length - Nombre de valeurs NaN : 0
Colonne length - Nombre de valeurs non-vides : 1500
```

Il y a 37 valeurs manquantes pour la variable *margin_low*, correspondant à la marge inférieure.

[6]:
```python
display(df_ini.groupby("is_genuine").count())
```

is_genuine	diagonal	height_left	height_right	margin_low	margin_up	length
False	500	500	500	492	500	500
True	1000	1000	1000	971	1000	1000

Notre échantillon comprend 1500 billets, 1000 sont vrais, 500 sont faux. Concernant les valeur smanquantes, il y en a dans les 2 groupes : 8 parmi les faux billets, 29 parmi les vrais billets. Pour la suite, on qualifiera de "positif" un faux billet et de "négatif" un vrai billet.

[7]:
```python
display(df_ini.describe())
```

	diagonal	height_left	height_right	margin_low	margin_up	\
count	1500.000000	1500.000000	1500.000000	1463.000000	1500.000000	
mean	171.958440	104.029533	103.920307	4.485967	3.151473	
std	0.305195	0.299462	0.325627	0.663813	0.231813	
min	171.040000	103.140000	102.820000	2.980000	2.270000	
25%	171.750000	103.820000	103.710000	4.015000	2.990000	
50%	171.960000	104.040000	103.920000	4.310000	3.140000	
75%	172.170000	104.230000	104.150000	4.870000	3.310000	
max	173.010000	104.880000	104.950000	6.900000	3.910000	

```
        length
count  1500.00000
mean    112.67850
std       0.87273
min     109.49000
25%     112.03000
50%     112.96000
75%     113.34000
max     114.44000
```

On observe que l'écart-type des distributions des mesures de la marge inférieure (pour lesquelles il y a des valeurs manquantes) est près de 3 fois supérieure à celle de la marge supérieure. Cela ne peut pas s'expliquer par le fait qu'il y ait des valeurs manquantes (2,5% de la taille de l'échantillon). Cela est sûrement dû au procédé de fabrication des billets. Il faut également envisager que les distributions peuvent être différentes selon que le billet est vrai ou faux.

[8]:
```
df_ini0 = df_ini.loc[(df_ini["is_genuine"]==False)]
display(df_ini0.describe())
```

```
         diagonal  height_left  height_right  margin_low   margin_up  \
count  500.000000   500.000000    500.000000  492.000000  500.000000
mean   171.901160   104.190340    104.143620    5.215935    3.350160
std      0.306861     0.223758      0.270878    0.553531    0.180498
min    171.040000   103.510000    103.430000    3.820000    2.920000
25%    171.690000   104.040000    103.950000    4.840000    3.220000
50%    171.910000   104.180000    104.160000    5.190000    3.350000
75%    172.092500   104.332500    104.320000    5.592500    3.472500
max    173.010000   104.880000    104.950000    6.900000    3.910000

         length
count  500.000000
mean   111.630640
std      0.615543
min    109.490000
25%    111.200000
50%    111.630000
75%    112.030000
max    113.850000
```

[9]:
```
df_ini1 = df_ini.loc[(df_ini["is_genuine"]==True)]
display(df_ini1.describe())
```

```
          diagonal  height_left  height_right  margin_low   margin_up  \
count  1000.000000  1000.000000    1000.00000  971.000000  1000.00000
mean    171.987080   103.949130     103.80865    4.116097     3.05213
std       0.300441     0.300231       0.29157    0.319124     0.18634
min     171.040000   103.140000     102.82000    2.980000     2.27000
25%     171.790000   103.740000     103.61000    3.905000     2.93000
50%     171.990000   103.950000     103.81000    4.110000     3.05000
```

```
75%      172.200000   104.140000     104.00000     4.340000     3.18000
max      172.920000   104.860000     104.95000     5.040000     3.74000

            length
count  1000.000000
mean    113.202430
std       0.359552
min     111.760000
25%     112.950000
50%     113.205000
75%     113.460000
max     114.440000
```

On constate des différences numériquement significatives entre les vrais billets et les faux-billets. Mais il faut utiliser des tests statistiques pour vérifier si des différences sont statistiquement significatives.

[10]:
```
# Calcul des z-scores
scaler_ini = StandardScaler(with_std=True)
scaler_ini.fit(df_ini.iloc[:,1:])
```

[10]:
```
StandardScaler()
```

[11]:
```
display(df_ini.groupby("is_genuine").mean().round(3))
display(df_ini.groupby("is_genuine").std(ddof=1).round(3))
```

```
            diagonal  height_left  height_right  margin_low  margin_up  \
is_genuine
False        171.901      104.190       104.144       5.216      3.350
True         171.987      103.949       103.809       4.116      3.052

             length
is_genuine
False       111.631
True        113.202
```

```
            diagonal  height_left  height_right  margin_low  margin_up  length
is_genuine
False          0.307        0.224         0.271       0.554      0.180   0.616
True           0.300        0.300         0.292       0.319      0.186   0.360
```

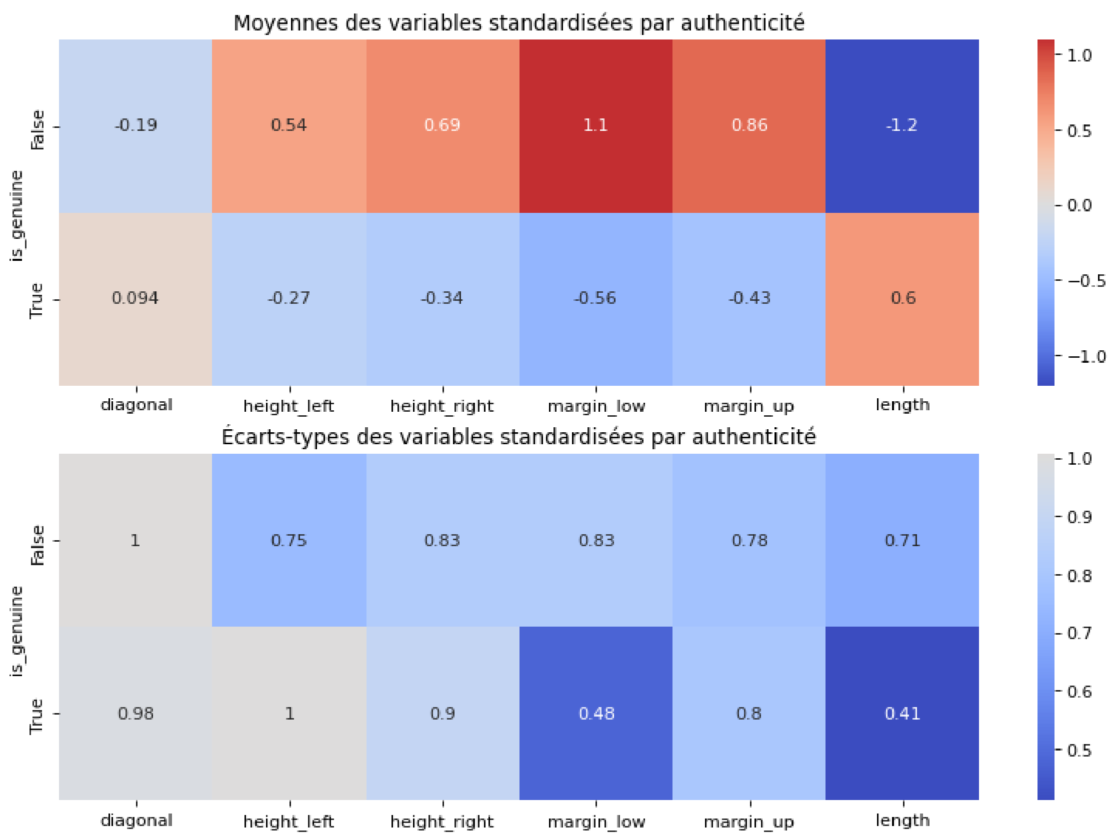
[12]:
```
df_ini_scaled = scaler_ini.transform(df_ini.iloc[:,1:])
# On récupère les index et les noms de colonne
df_ini_scaled = pd.DataFrame(df_ini_scaled, index=df_ini.index,␣
 ↪columns=list(df_ini.columns)[1:])
df_ini_scaled = pd.merge(df_ini["is_genuine"], df_ini_scaled, how='left',␣
 ↪left_index=True, right_index=True)
display(df_ini_scaled.groupby("is_genuine").mean().round(3))
display(df_ini_scaled.groupby("is_genuine").std(ddof=1).round(3))
```

is_genuine	diagonal	height_left	height_right	margin_low	margin_up	length
False	-0.188	0.537	0.686	1.100	0.857	-1.201
True	0.094	-0.269	-0.343	-0.557	-0.429	0.601

is_genuine	diagonal	height_left	height_right	margin_low	margin_up	length
False	1.006	0.747	0.832	0.834	0.779	0.706
True	0.985	1.003	0.896	0.481	0.804	0.412

[13]:

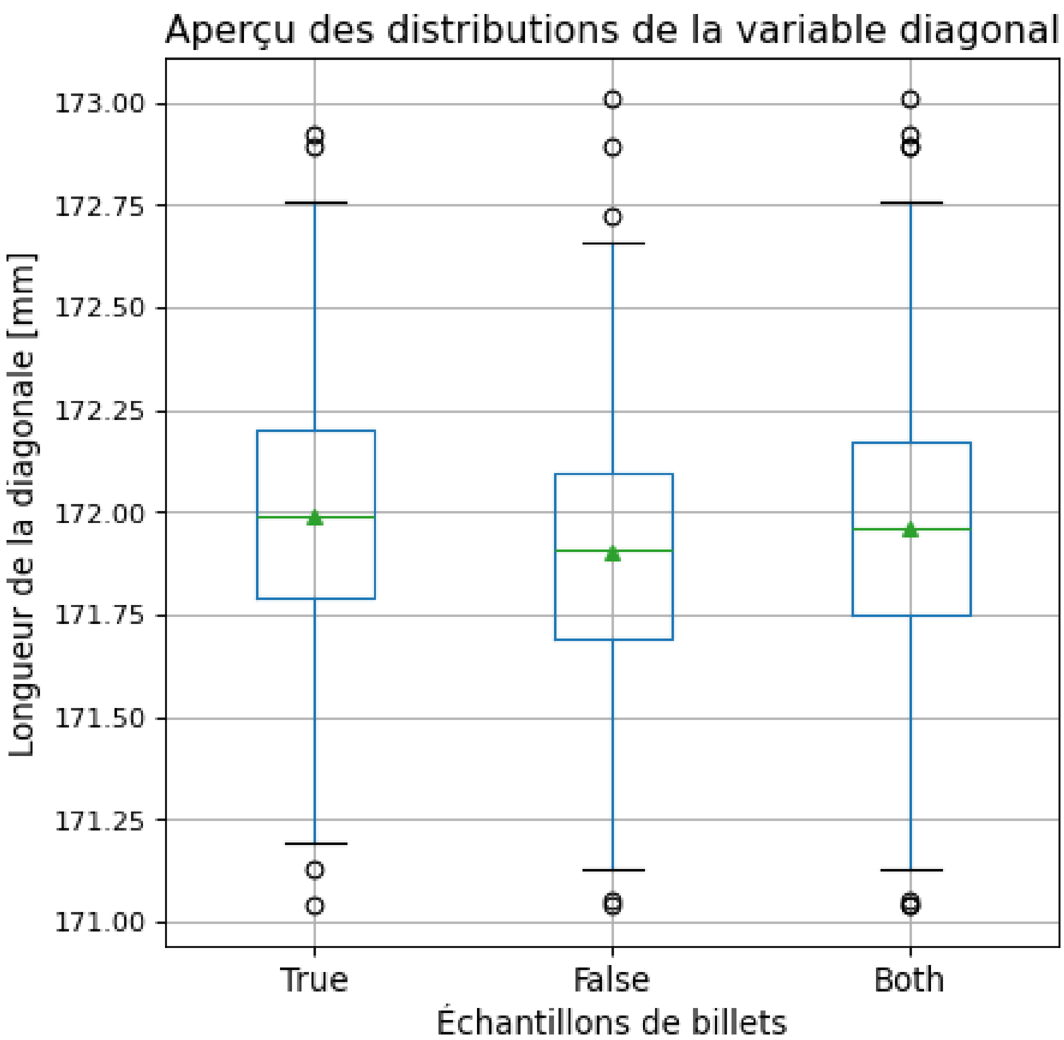
```python
fig = plt.figure(figsize=(12,8), dpi=80)
plt.subplot(2,1,1)
sb.heatmap(df_ini_scaled.groupby("is_genuine").mean().round(3), annot=True, center=0, cmap='coolwarm')
plt.title("Moyennes des variables standardisées par authenticité")

plt.subplot(2,1,2)
sb.heatmap(df_ini_scaled.groupby("is_genuine").std(ddof=1).round(3), annot=True, center=1, cmap='coolwarm')
plt.title("Écarts-types des variables standardisées par authenticité")
plt.savefig("Carac_authenticités.png")
plt.show()
```



2.2 - Analyses univariées

2.2.1 - Variable diagonal

[14]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["diagonal"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["diagonal"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["diagonal"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Longueur de la diagonale [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable diagonal", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



[15]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪proviennent de la même distribution
res = st.ks_2samp(df_ini1["diagonal"], df_ini0["diagonal"],␣
↪alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")

# Hypothèse nulle : les vrais billets ont une diagonale plus grande ou égale à␣
↪celle des faux billets
res = st.ks_2samp(df_ini1["diagonal"], df_ini0["diagonal"],␣
↪alternative='greater')
```

```
# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

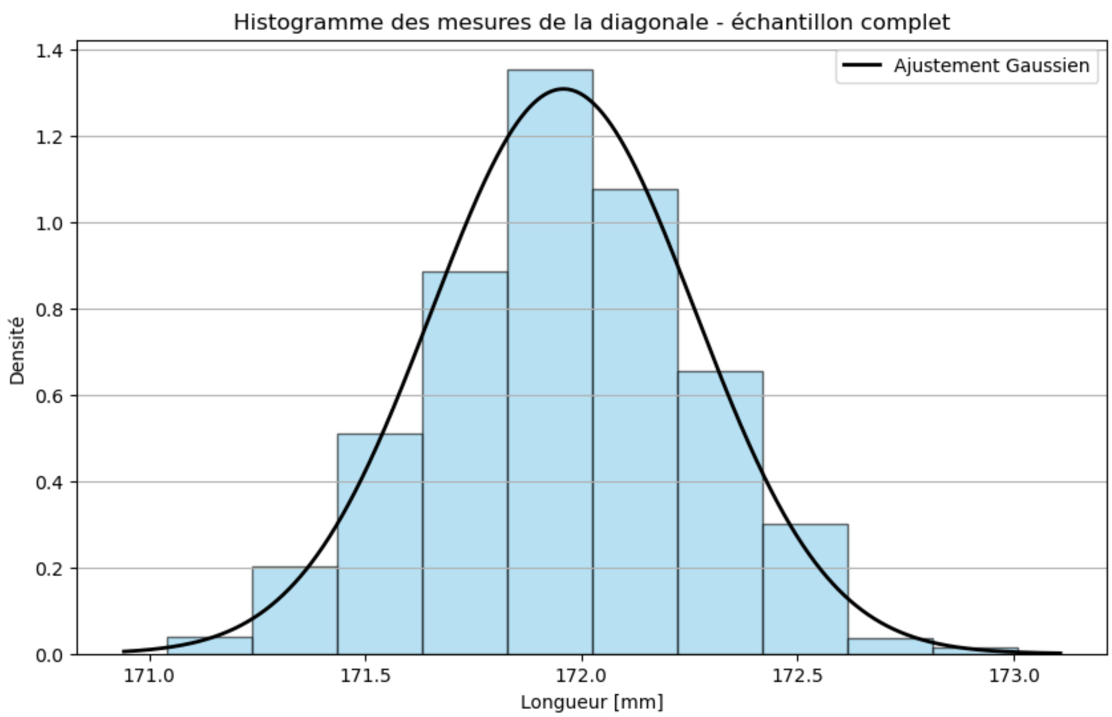
```
Statistique signée du test KS : -0.124
Valeur p : 6.739631403936162e-05

Statistique signée du test KS : 0.002
Valeur p : 0.9955654174830928


C:\Users\nicol\anaconda3\Lib\site-packages\scipy\stats\_axis_nan_policy.py:531:
RuntimeWarning: ks_2samp: Exact calculation unsuccessful. Switching to
method=asymp.
  res = hypotest_fun_out(*samples, **kwds)
```

Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de diagonale des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les diagonales des faux billets sont statistiquement plus basses que celles des vrais billets.

[16]:
```
data = df_ini["diagonal"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
```

```
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la diagonale - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 171.96
Écart type : 0.31
Z_min : -3.01
Z_max : 3.45
```

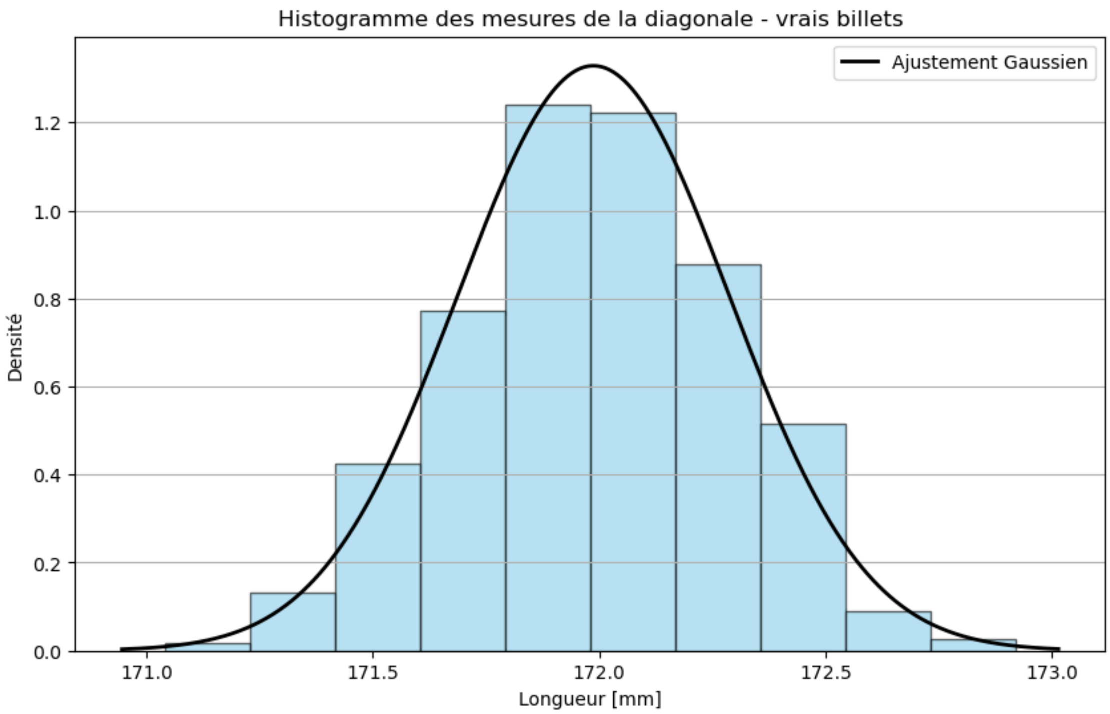



```
Statistique du test de Shapiro-Wilk : 0.9986864738301697
```

```
Valeur p : 0.324085127878592
```

[17]:
```
data = df_ini1["diagonal"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la diagonale - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 171.99
Écart type : 0.3
Z_min : -3.15
Z_max : 3.11
```

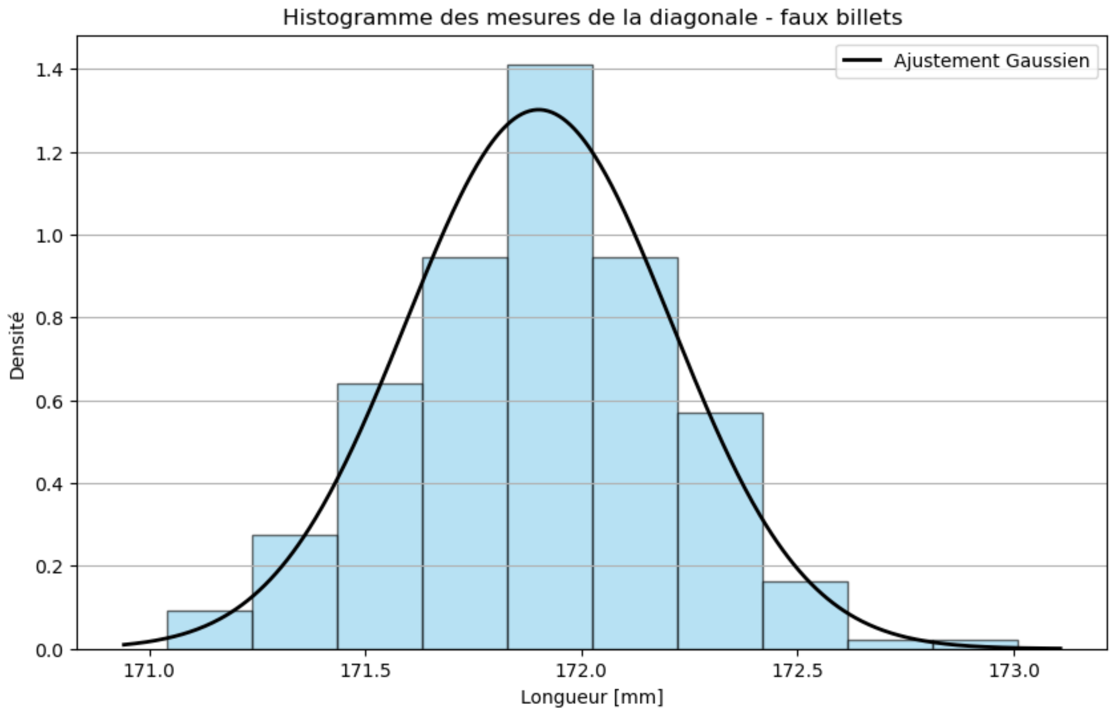



```
Statistique du test de Shapiro-Wilk : 0.9980672908636019
Valeur p : 0.31196640875121606
```

[18]:
```
data = df_ini0["diagonal"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
```

```python
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la diagonale - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 171.9
Écart type : 0.31
Z_min : -2.81
Z_max : 3.61
```



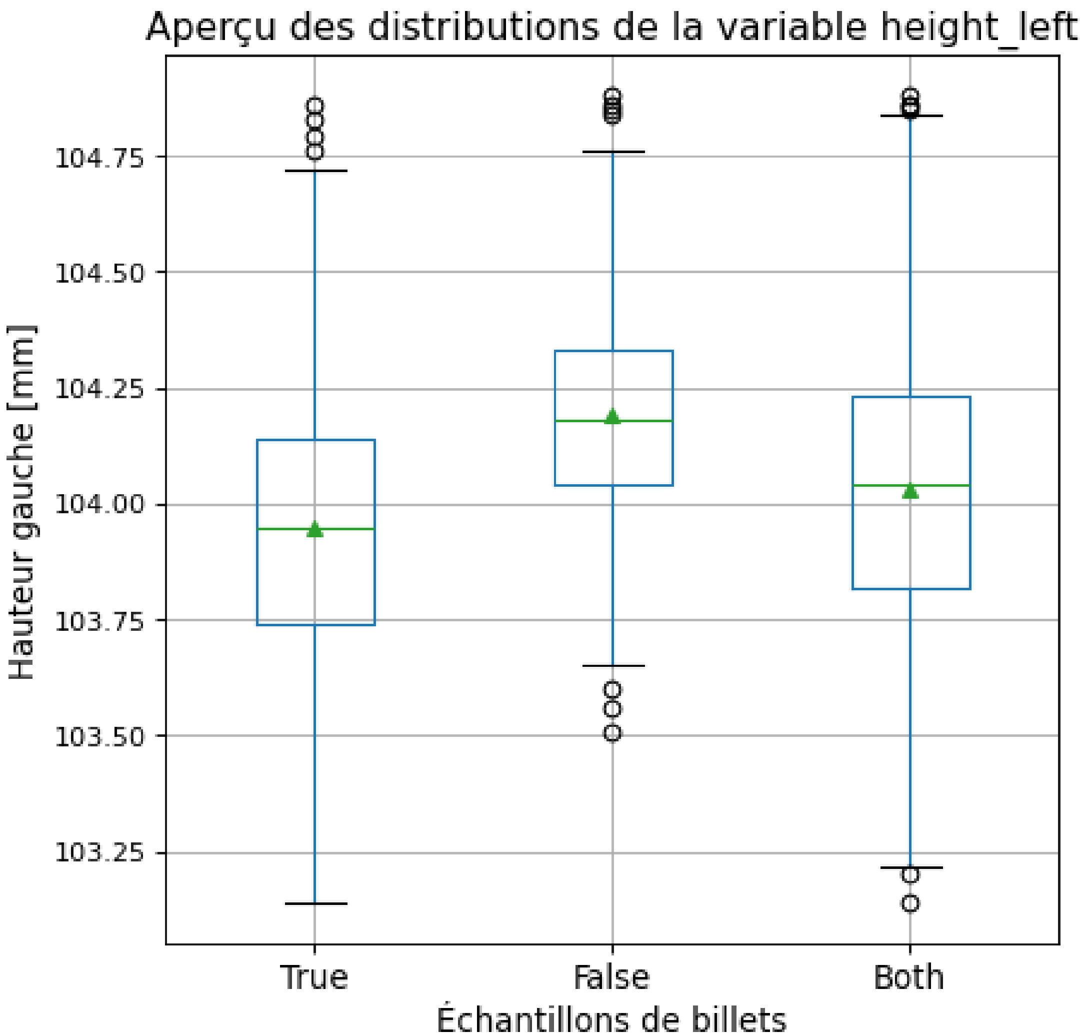

```
Statistique du test de Shapiro-Wilk : 0.9974348748668599
Valeur p : 0.6385168317013452
```

Que l'on prenne l'échantillon complet ou les 2 sous-échantillons, on peut accepter l'hypothèse que les distributions des mesures de la diagonale suivent une loi normale.

2.2.2 - Variable height_left

[19]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["height_left"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["height_left"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["height_left"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Hauteur gauche [mm]", fontsize=12)
```

```
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable height_left", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



[20]:

```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪proviennent de la même distribution
res = st.ks_2samp(df_ini1["height_left"], df_ini0["height_left"],␣
↪alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

```
# Hypothèse nulle : les vrais billets ont une hauteur à gauche plus petite ou␣
 ↪égale à celle faux billets
res = st.ks_2samp(df_ini1["height_left"], df_ini0["height_left"],␣
 ↪alternative='less')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

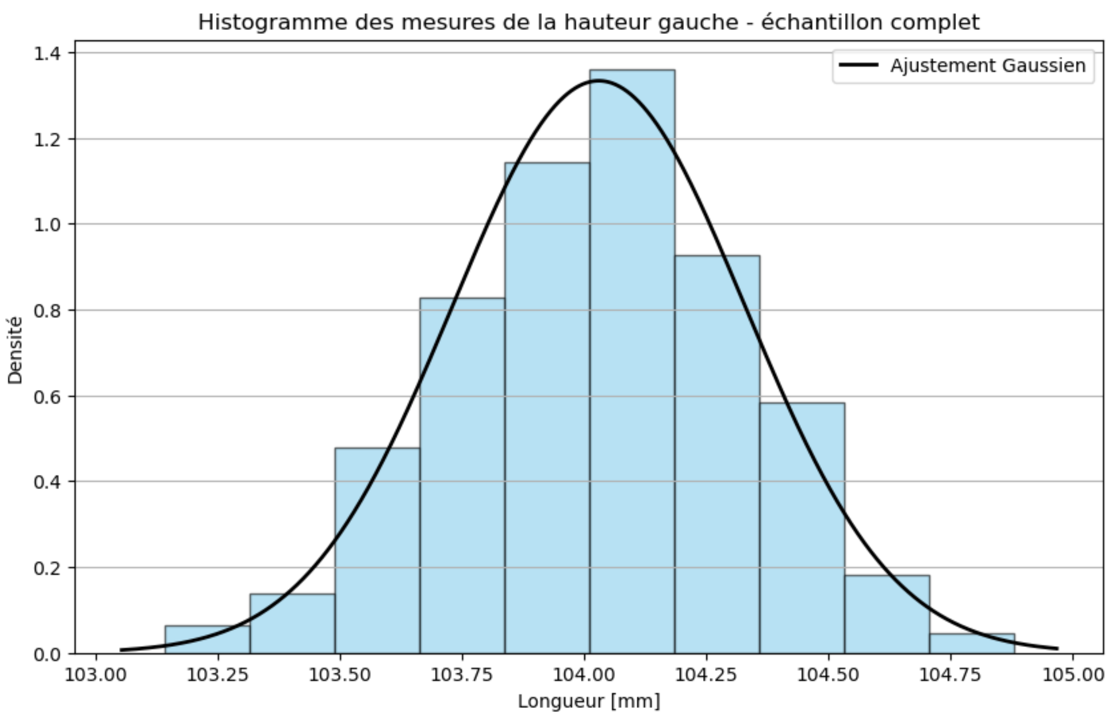
```
Statistique signée du test KS : 0.386
Valeur p : 9.389234875502185e-45

Statistique signée du test KS : -0.0
Valeur p : 1.0
```

Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de hauteur à gauche des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les hauteurs à gauche des faux billets sont statistiquement plus élevées que celles des vrais billets.

[21]:
```
data = df_ini["height_left"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
```

```python
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur gauche - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 104.03
Écart type : 0.3
Z_min : -2.97
Z_max : 2.84
```

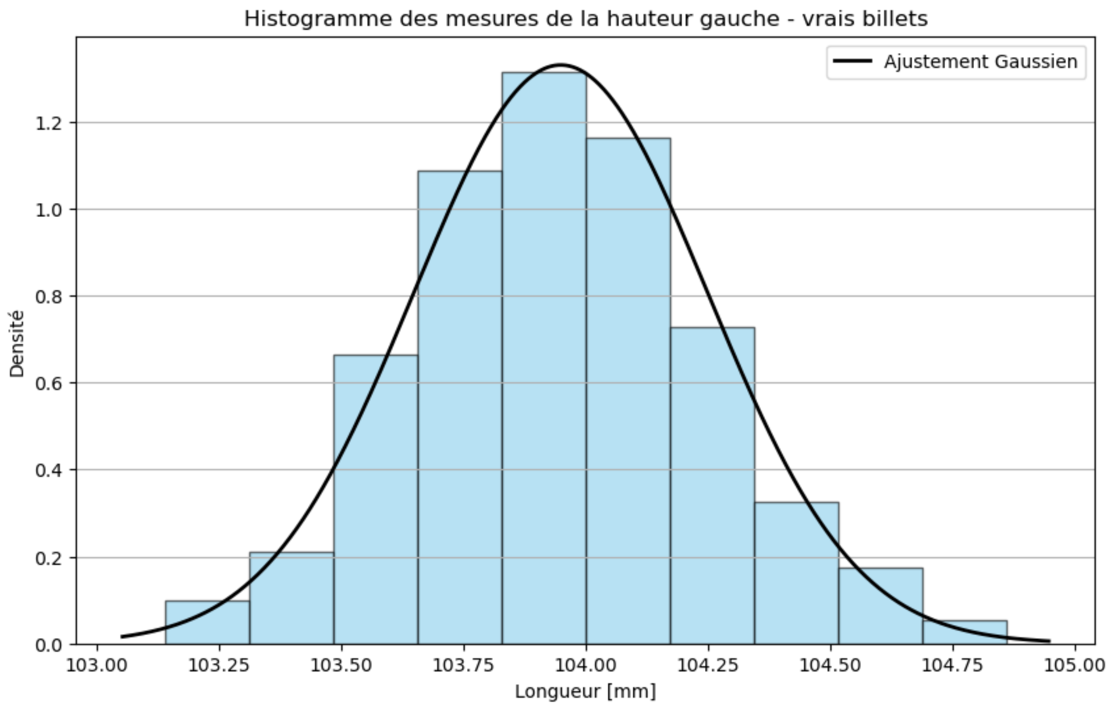



```
Statistique du test de Shapiro-Wilk : 0.9979048516498801
```

```
Valeur p : 0.051416122734409625
```

```
[22]: data = df_ini1["height_left"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur gauche - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 103.95
Écart type : 0.3
Z_min : -2.7
Z_max : 3.03
```



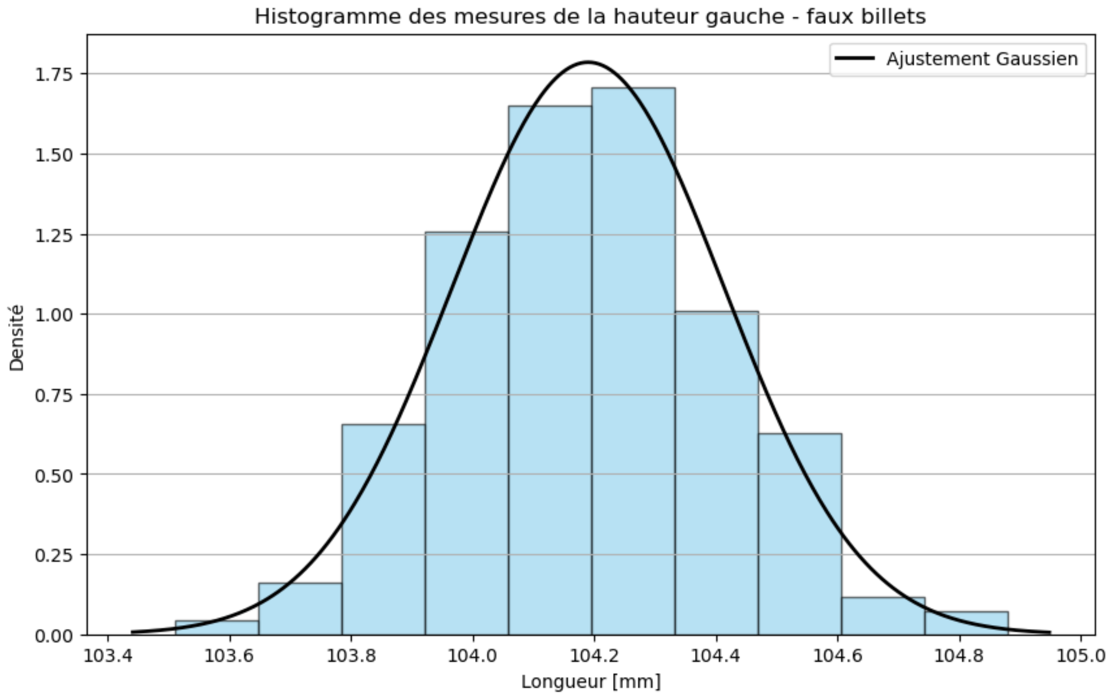

```
Statistique du test de Shapiro-Wilk : 0.9965844064548799
Valeur p : 0.02868852105934944
```

[23]:

```
data = df_ini0["height_left"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
```

```python
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur gauche - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 104.19
Écart type : 0.22
Z_min : -3.04
Z_max : 3.08
```



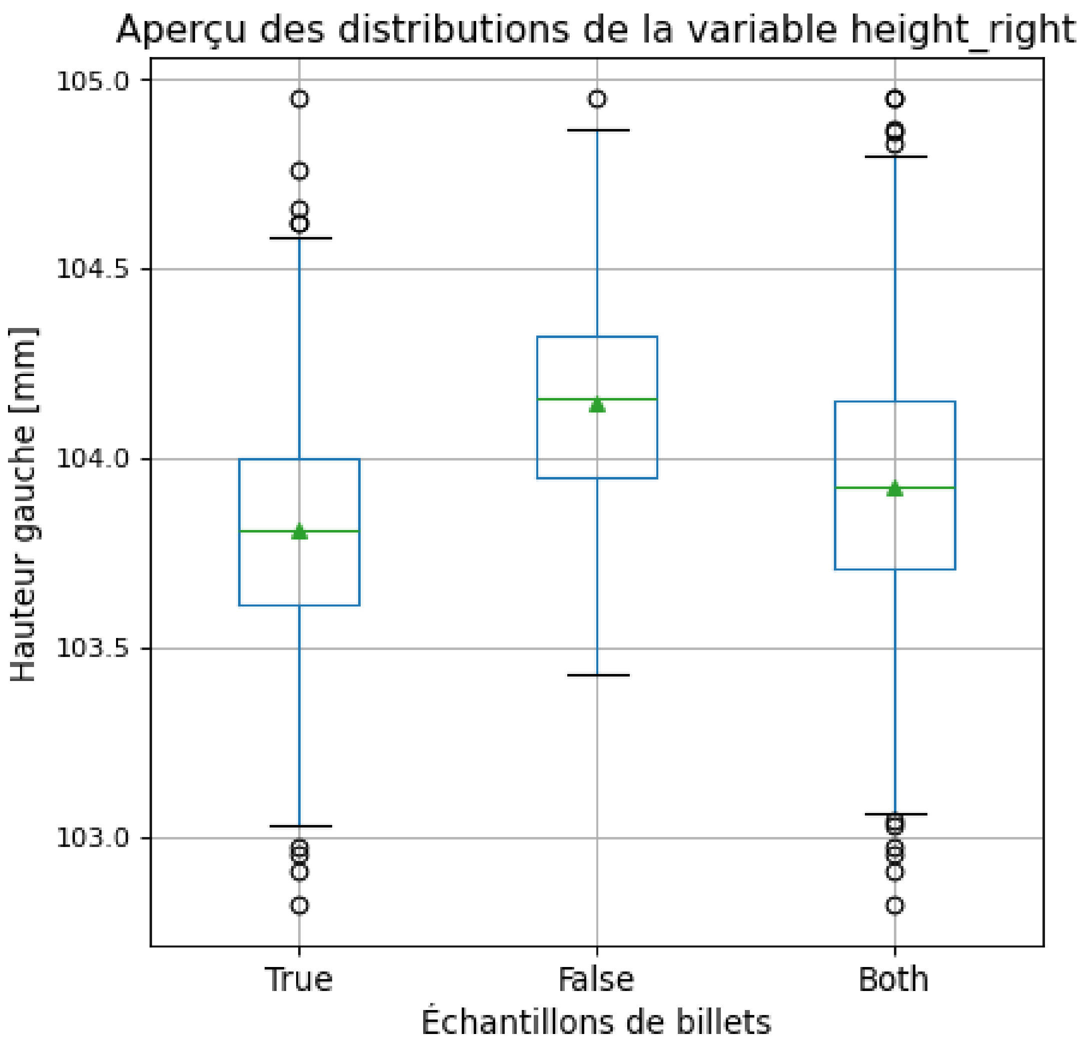

```
Statistique du test de Shapiro-Wilk : 0.9978760420739532
Valeur p : 0.7905021013822957
```

La distribution des mesures de la hauteur à gauche des faux billets est très proche d'une loi normale, alors qu'il semble plus raisonnable de rejeter l'hypothèse de normalité pour la distribution de celle des varsi billets. Si on regroupe les 2 sous-échantillons, on peut tout juste accepter que la distribution suit une loi normale.

2.2.3 - Variable height_right

[24]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["height_right"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["height_right"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["height_right"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
```

```
plt.ylabel("Hauteur gauche [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable height_right", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



[25]:

```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪proviennent de la même distribution
res = st.ks_2samp(df_ini1["height_right"], df_ini0["height_right"],␣
↪alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
```

```python
print(f"Valeur p : {res.pvalue}\n")

# Hypothèse nulle : les vrais billets ont une hauteur à gauche plus petite ou␣
↪égale à celle faux billets
res = st.ks_2samp(df_ini1["height_right"], df_ini0["height_right"],␣
↪alternative='less')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

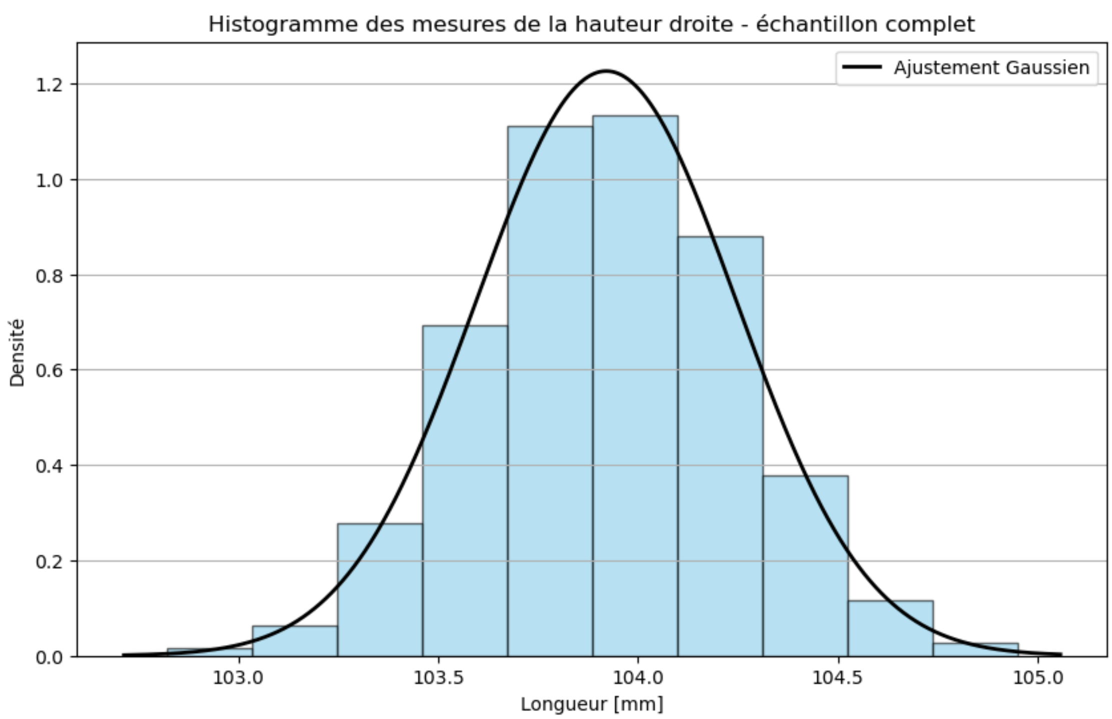
```
Statistique signée du test KS : 0.451
Valeur p : 1.4001528757840663e-61

Statistique signée du test KS : -0.0
Valeur p : 1.0
```

Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de hauteur à droite des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les hauteurs à droite des faux billets sont statistiquement plus élevées que celles des vrais billets.

[26]:
```python
data = df_ini["height_right"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)
```

```python
# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur droite - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 103.92
Écart type : 0.33
Z_min : -3.38
Z_max : 3.16
```

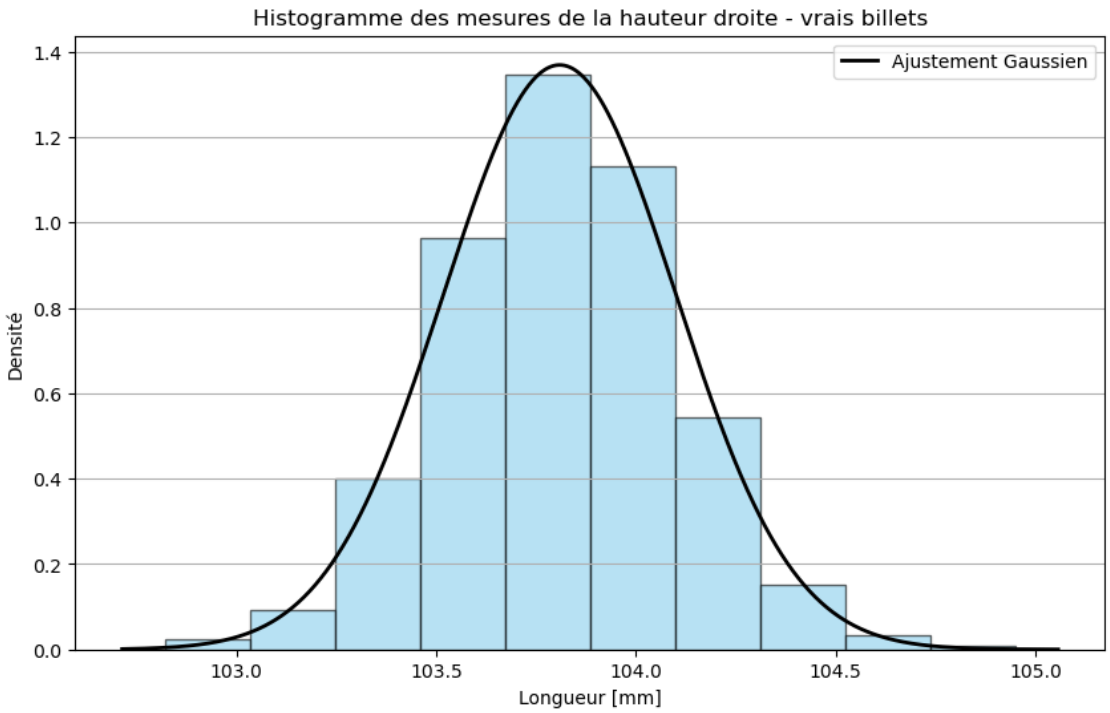


```
Statistique du test de Shapiro-Wilk : 0.999515456099867
Valeur p : 0.9799777902342844
```

[27]:
```python
data = df_ini1["height_right"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur droite - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
```

```
print(f"Valeur p : {p_value}")
```

```
Moyenne : 103.81
Écart type : 0.29
Z_min : -3.39
Z_max : 3.91
```

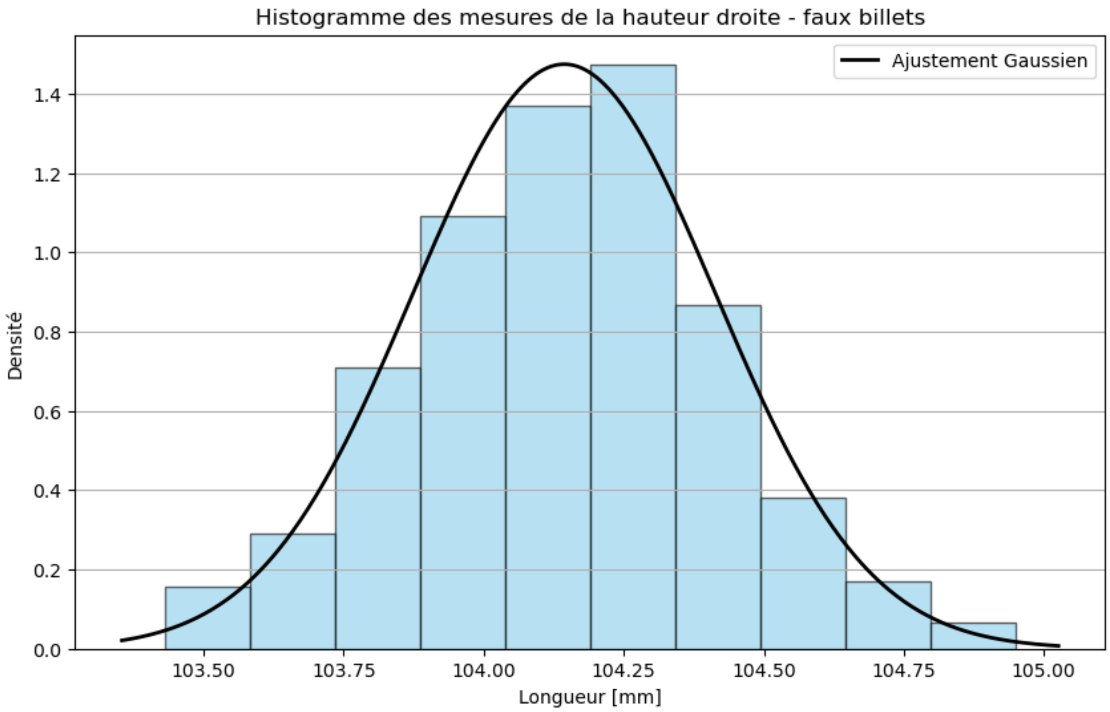



```
Statistique du test de Shapiro-Wilk : 0.9985481811752999
Valeur p : 0.5863761785467336
```

[28]:
```
data = df_ini0["height_right"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))
```

```python
# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la hauteur droite - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 104.14
Écart type : 0.27
Z_min : -2.63
Z_max : 2.98
```



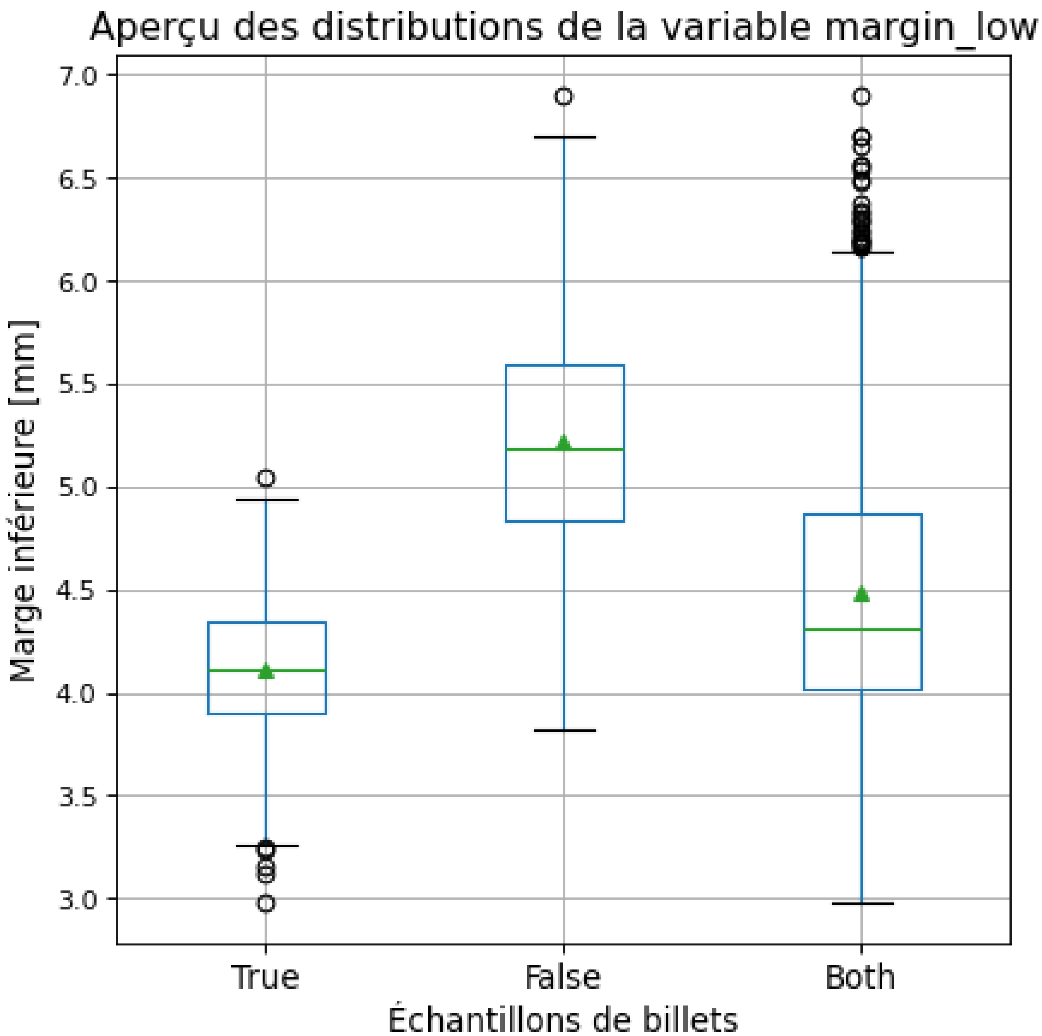

```
Statistique du test de Shapiro-Wilk : 0.9979898161488796
Valeur p : 0.8265215974202331
```

Que l'on prenne l'échantillon complet ou les 2 sous-échantillons, on peut accepter l'hypothèse que les distributions des mesures de la hauteur à droite suivent une loi normale.

2.2.4 - Variable margin_low

[29]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["margin_low"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["margin_low"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["margin_low"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Marge inférieure [mm]", fontsize=12)
```

```
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable margin_low", fontsize=14)
plt.savefig("Boxplot_margin_low.png")
plt.show()
```



[30]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪proviennent de la même distribution
res = st.ks_2samp(df_ini1.loc[(df_ini1["margin_low"].
↪isna())==False]["margin_low"],\
                  df_ini0.loc[(df_ini0["margin_low"].
↪isna())==False]["margin_low"],\
                  alternative='two-sided')
```

```
# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")

# Hypothèse nulle : les vrais billets ont une marge inférieure plus petite ou␣
↪égale à celle faux billets
res = st.ks_2samp(df_ini1.loc[(df_ini1["margin_low"].
↪isna())==False]["margin_low"],\
                  df_ini0.loc[(df_ini0["margin_low"].
↪isna())==False]["margin_low"],\
                  alternative='less')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

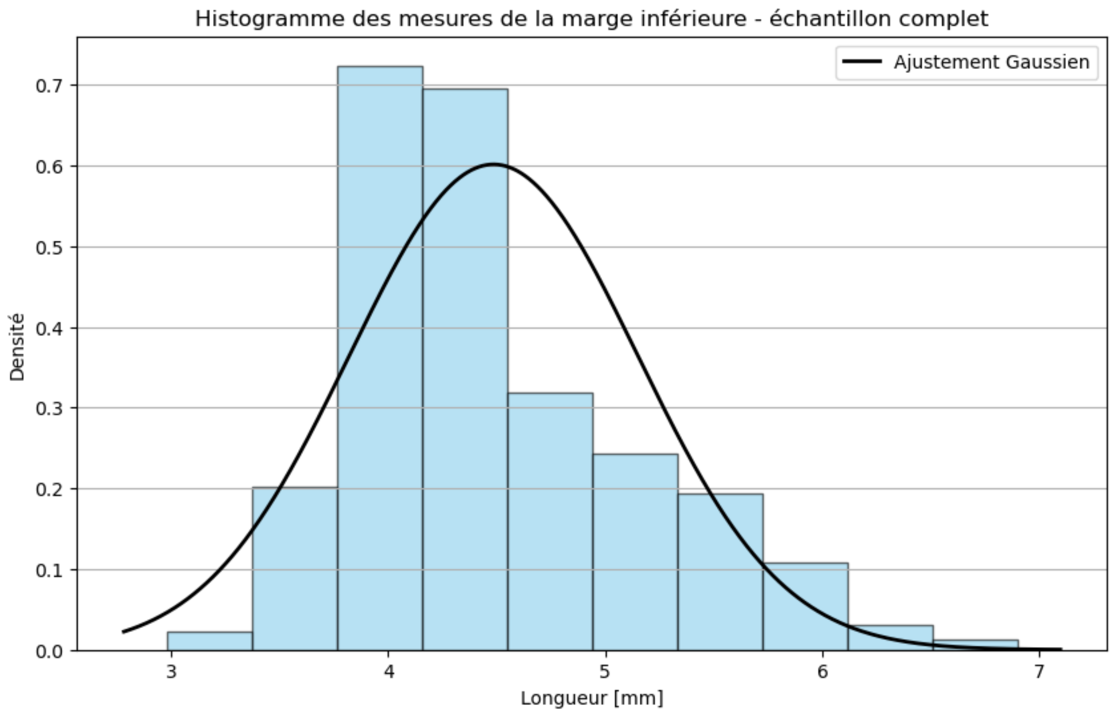
```
Statistique signée du test KS : 0.8111346947661032
Valeur p : 7.653539205087972e-218

Statistique signée du test KS : -0.0
Valeur p : 1.0
```

Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de marge inférieure des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les marges inférieures des faux billets sont statistiquement plus élevées que celles des vrais billets.

[31]:
```
data = df_ini["margin_low"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - échantillon␣
↪complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 4.49
Écart type : 0.66
Z_min : -2.27
Z_max : 3.64
```

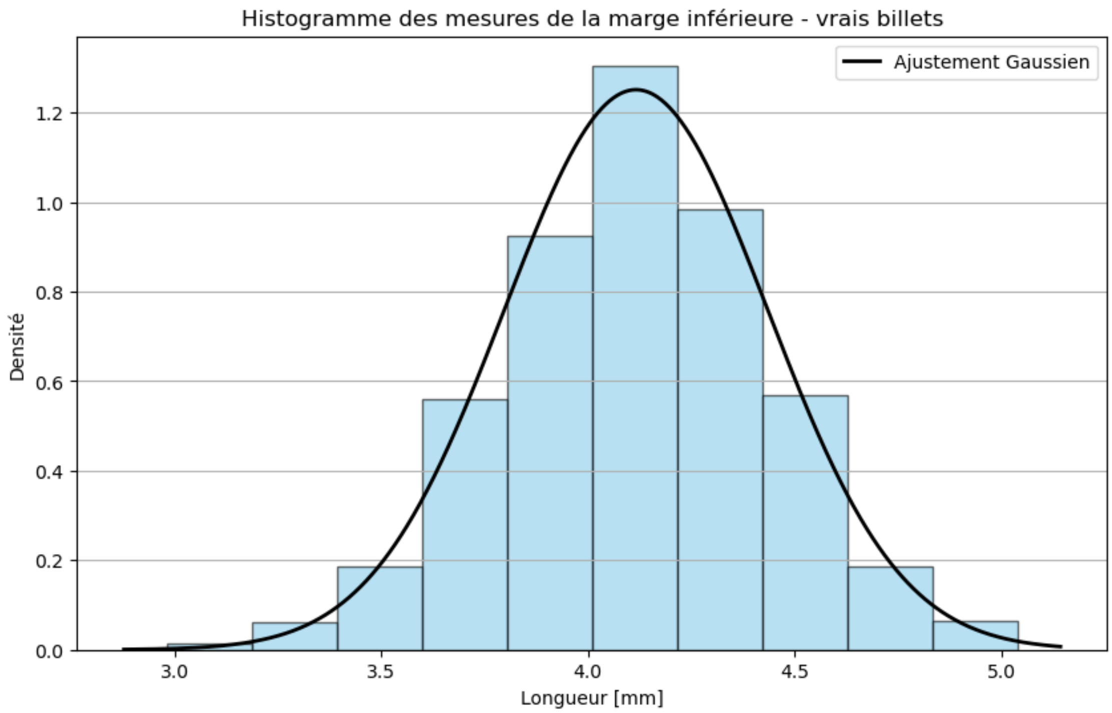



```
Statistique du test de Shapiro-Wilk : 0.9377844940452797
Valeur p : 2.82931851362892e-24
```

[32]:
```python
data = df_ini1["margin_low"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 4.12
Écart type : 0.32
Z_min : -3.56
Z_max : 2.9
```



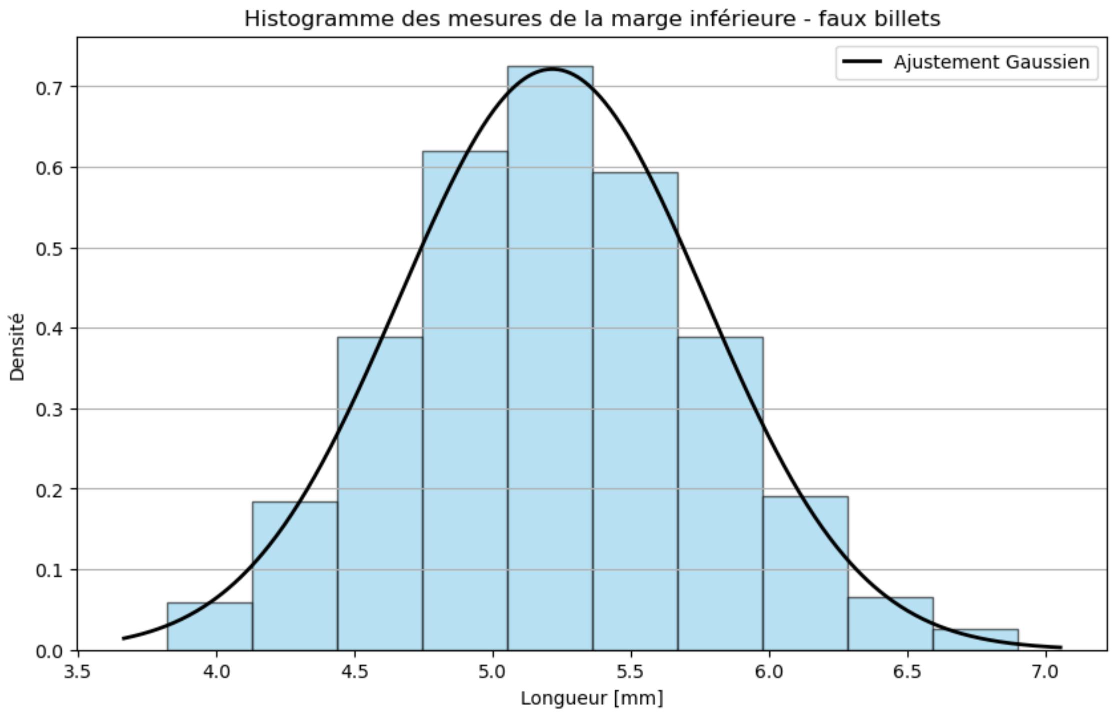

```
Statistique du test de Shapiro-Wilk : 0.9983712677951931
Valeur p : 0.49994166442573323
```

[33]:

```python
data = df_ini0["margin_low"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 5.22
Écart type : 0.55
Z_min : -2.52
Z_max : 3.04
```



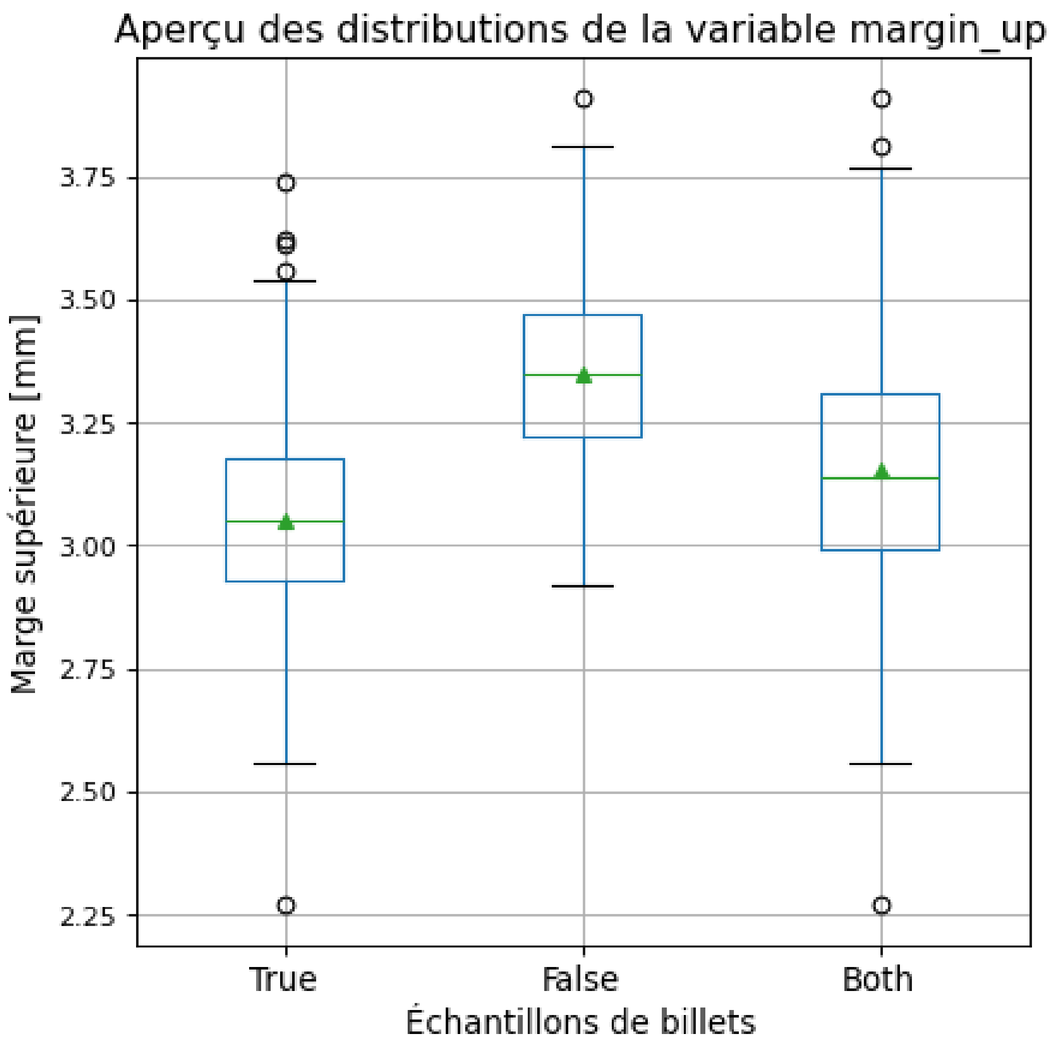

```
Statistique du test de Shapiro-Wilk : 0.9971536757202469
Valeur p : 0.5553809108515791
```

Les distributions des marges inférieures des 2 sous-échantillons sont chacune très proches d'une loi normale. En revanche, si on rassemble les 2 sous-échantillons, la distribution des marges inférieures qui en résulte devient très éloignée d'une loi normale.

2.2.5 - Variable margin_up

[34]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["margin_up"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["margin_up"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["margin_up"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
```

```python
plt.ylabel("Marge supérieure [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable margin_up", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



[35]:
```python
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪proviennent de la même distribution
res = st.ks_2samp(df_ini1["margin_up"], df_ini0["margin_up"],␣
↪alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
```

```
print(f"Valeur p : {res.pvalue}\n")

# Hypothèse nulle : les vrais billets ont une marge_supérieure plus petite ou␣
↪égale à celle faux billets
res = st.ks_2samp(df_ini1["margin_up"], df_ini0["margin_up"],␣
↪alternative='less')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

```
Statistique signée du test KS : 0.583
Valeur p : 1.3450552820927147e-105

Statistique signée du test KS : -0.0
Valeur p : 1.0
```

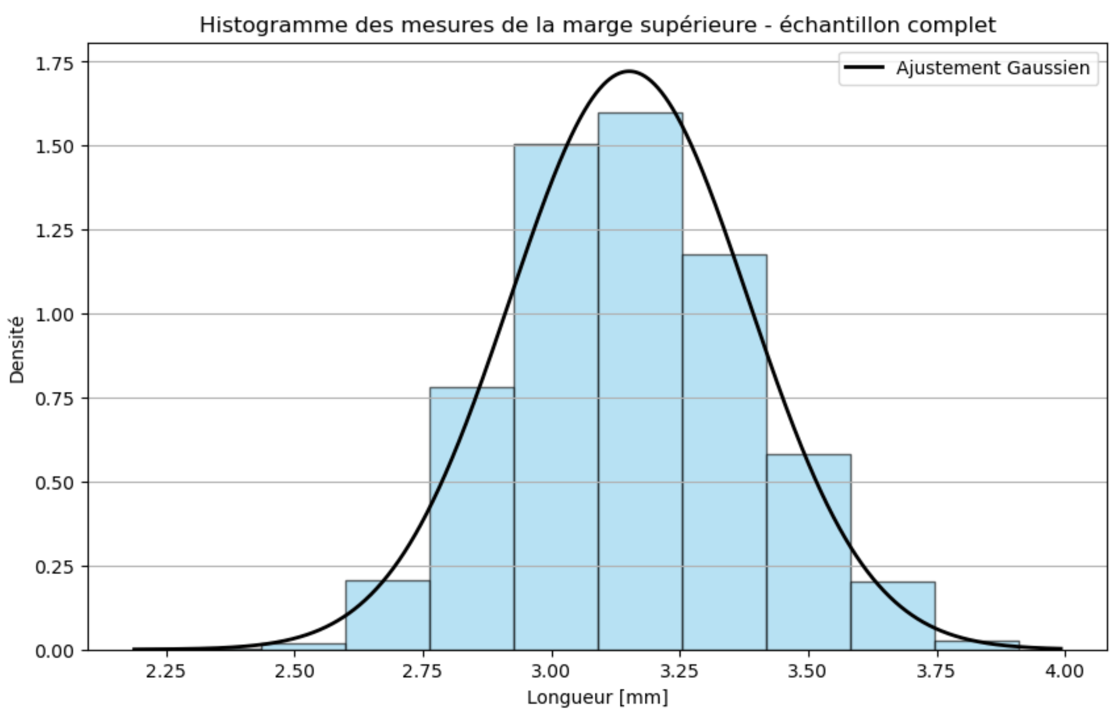
Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de marge supérieure des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les marges supérieures des faux billets sont statistiquement plus élevées que celles des vrais billets.

[36]:

```
data = df_ini["margin_up"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)
```

```
# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge supérieure - échantillon␣
 ↪complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 3.15
Écart type : 0.23
Z_min : -3.8
Z_max : 3.27
```

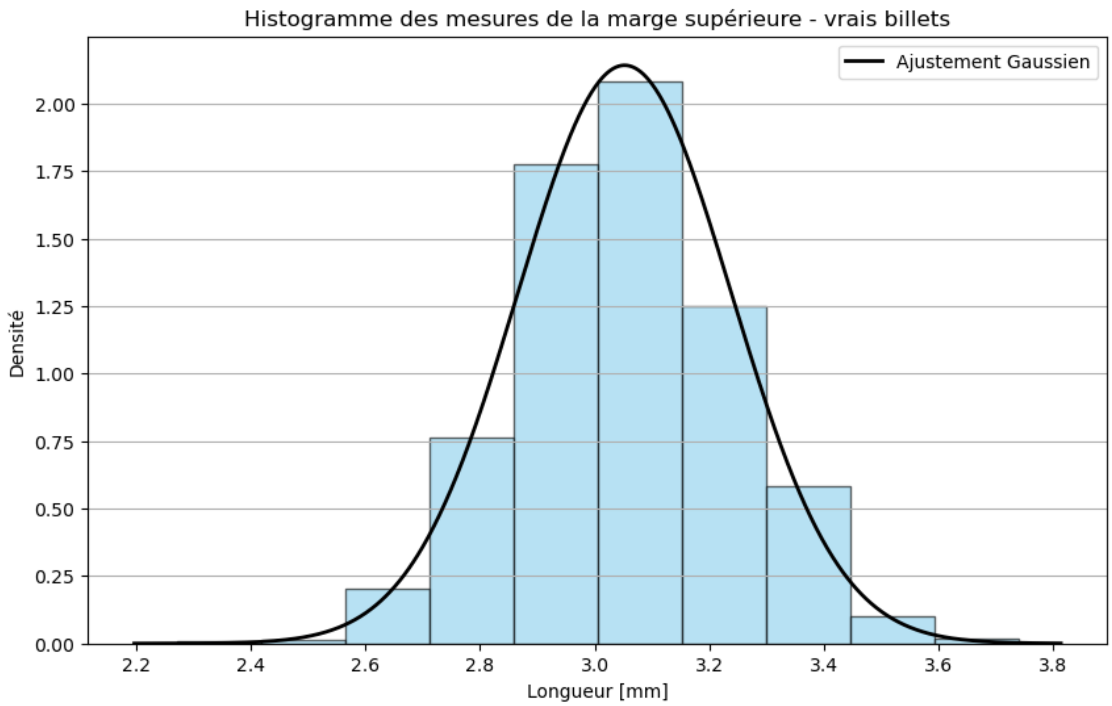


```
Statistique du test de Shapiro-Wilk : 0.9961647384349551
Valeur p : 0.0008086307957689068
```

[37]:
```
data = df_ini1["margin_up"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge supérieure - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
```

```python
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 3.05
Écart type : 0.19
Z_min : -4.2
Z_max : 3.69
```



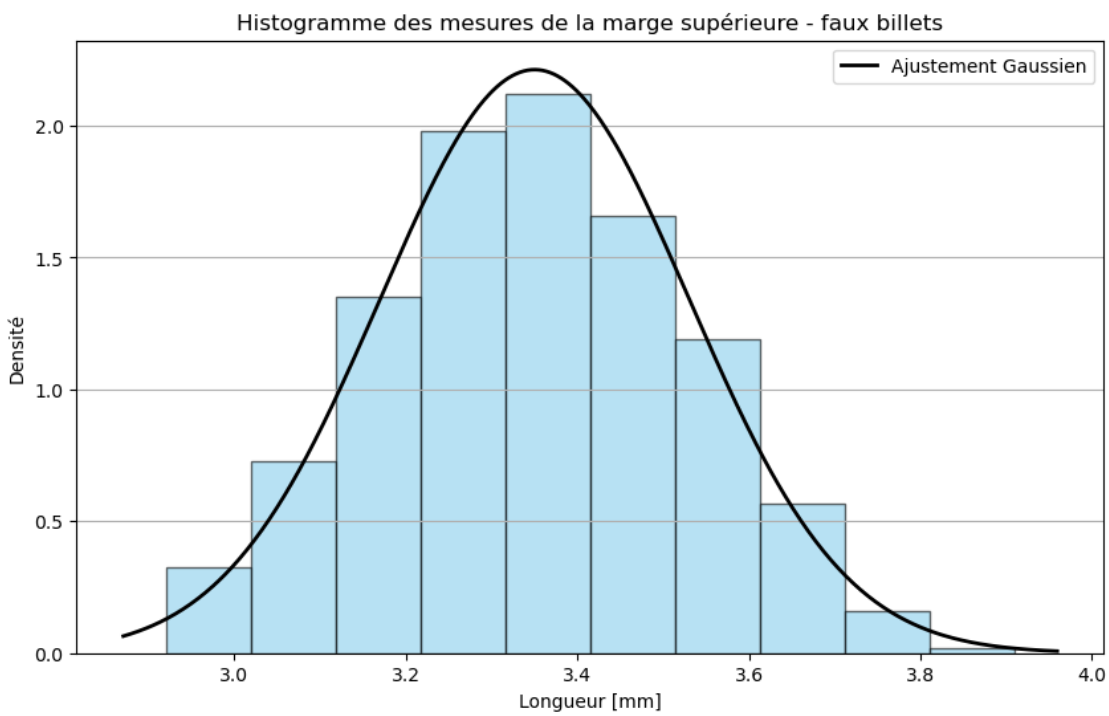

```
Statistique du test de Shapiro-Wilk : 0.9981576358117178
Valeur p : 0.3550467932958449
```

[38]:

```python
data = df_ini0["margin_up"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))
```

```python
# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge supérieure - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 3.35
Écart type : 0.18
Z_min : -2.38
Z_max : 3.1
```



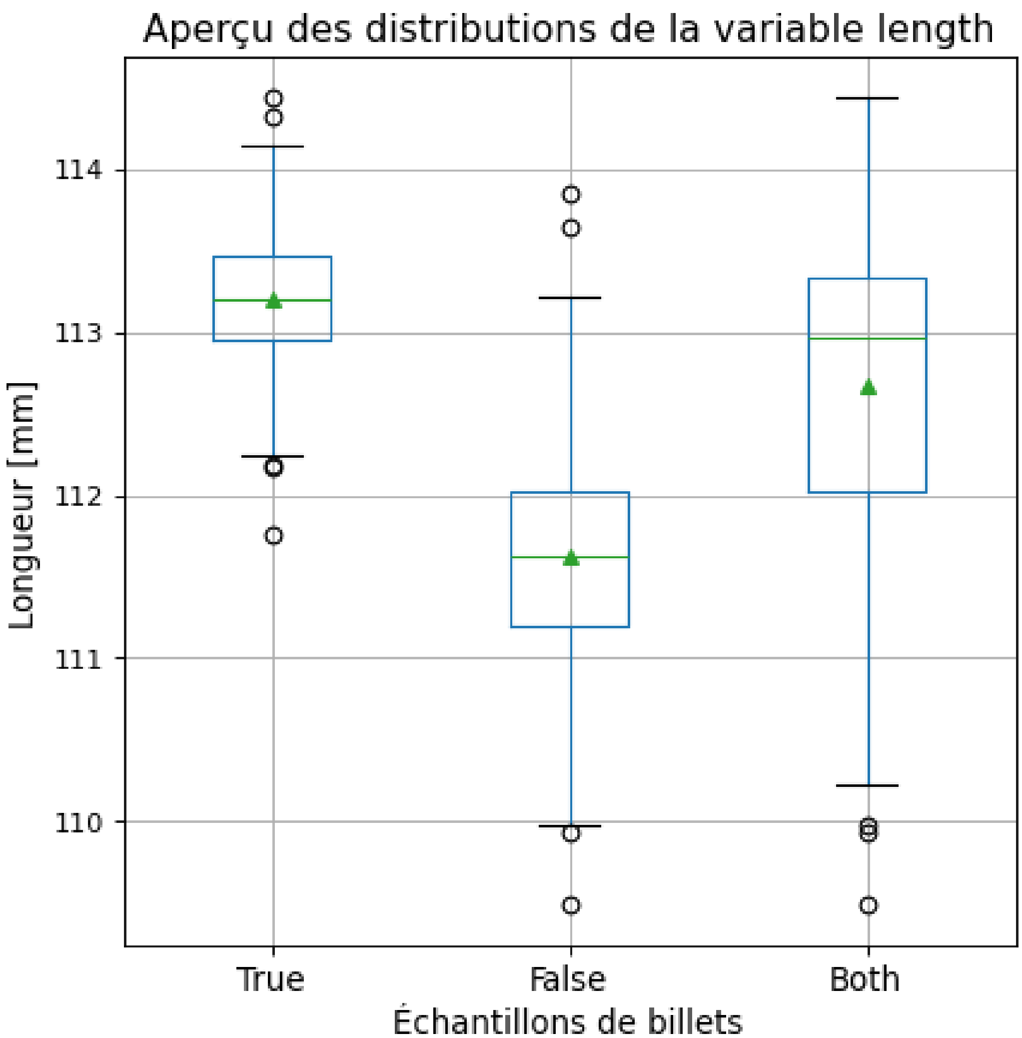

```
Statistique du test de Shapiro-Wilk : 0.99573455798398
Valeur p : 0.19271100431314336
```

Même commentaire que pour les distributions des marges inférieures.

2.2.6 - Variable length

[39]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["length"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["length"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["length"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Longueur [mm]", fontsize=12)
plt.xlim([0.5,3.5])
```

```
plt.title("Aperçu des distributions de la variable length", fontsize=14)
plt.savefig("Boxplot_length.png")
plt.show()
```



[40]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪provienment de la même distribution
res = st.ks_2samp(df_ini1["length"], df_ini0["length"], alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

```
# Hypothèse nulle : les vrais billets ont une longueur plus grande ou égale à␣
↪celle faux billets
res = st.ks_2samp(df_ini1["length"], df_ini0["length"], alternative='greater')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

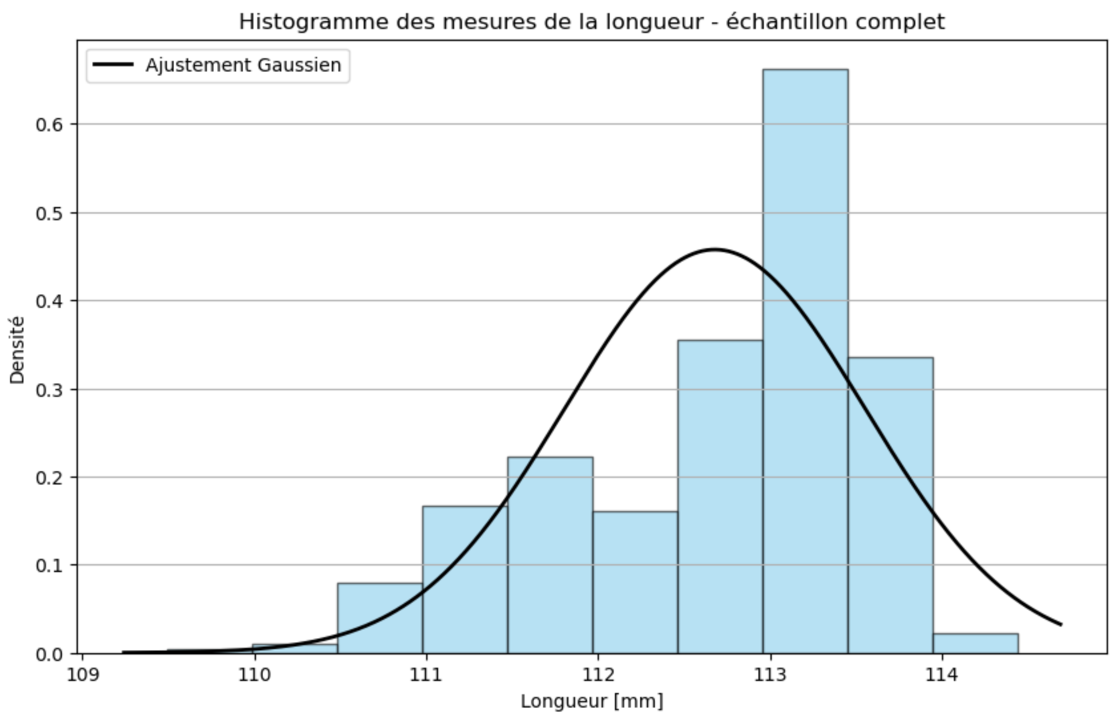
```
Statistique signée du test KS : -0.906
Valeur p : 2.3429706128869517e-295

Statistique signée du test KS : 0.0
Valeur p : 1.0
```

Grâce à ces tests, on peut rejeter l'hypothèse que les mesures de longueur des vrais billets et des faux billets proviennent de la même distribution, et on peut même affirmer que les longueurs des faux billets sont statistiquement plus faibles que celles des vrais billets.

[41]:
```
data = df_ini["length"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')
```

```python
# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la longueur - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 112.68
Écart type : 0.87
Z_min : -3.65
Z_max : 2.02
```

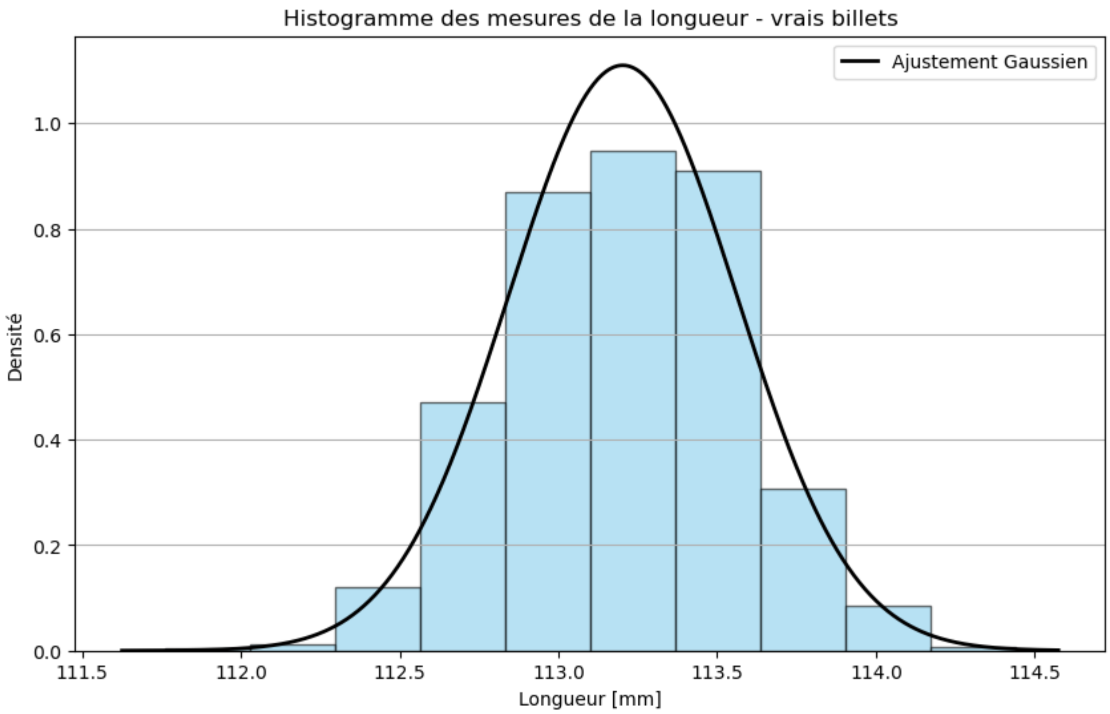



```
Statistique du test de Shapiro-Wilk : 0.9175992755377754
Valeur p : 7.859411012882467e-28
```

```
[42]: data = df_ini1["length"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la longueur - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 113.2
Écart type : 0.36
```

```
Z_min : -4.01
Z_max : 3.44
```



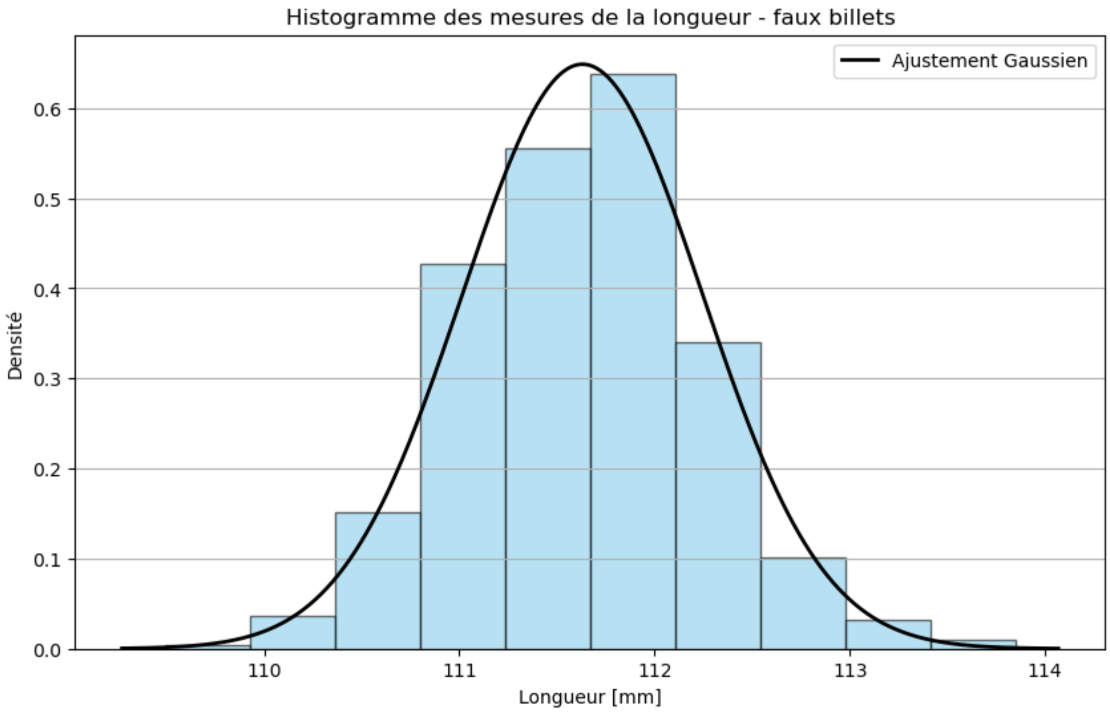

```
Statistique du test de Shapiro-Wilk : 0.9980461673650071
Valeur p : 0.3025071247259741
```

[43]:

```
data = df_ini0["length"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
```

```python
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la longueur - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 111.63
Écart type : 0.62
Z_min : -3.48
Z_max : 3.61
```



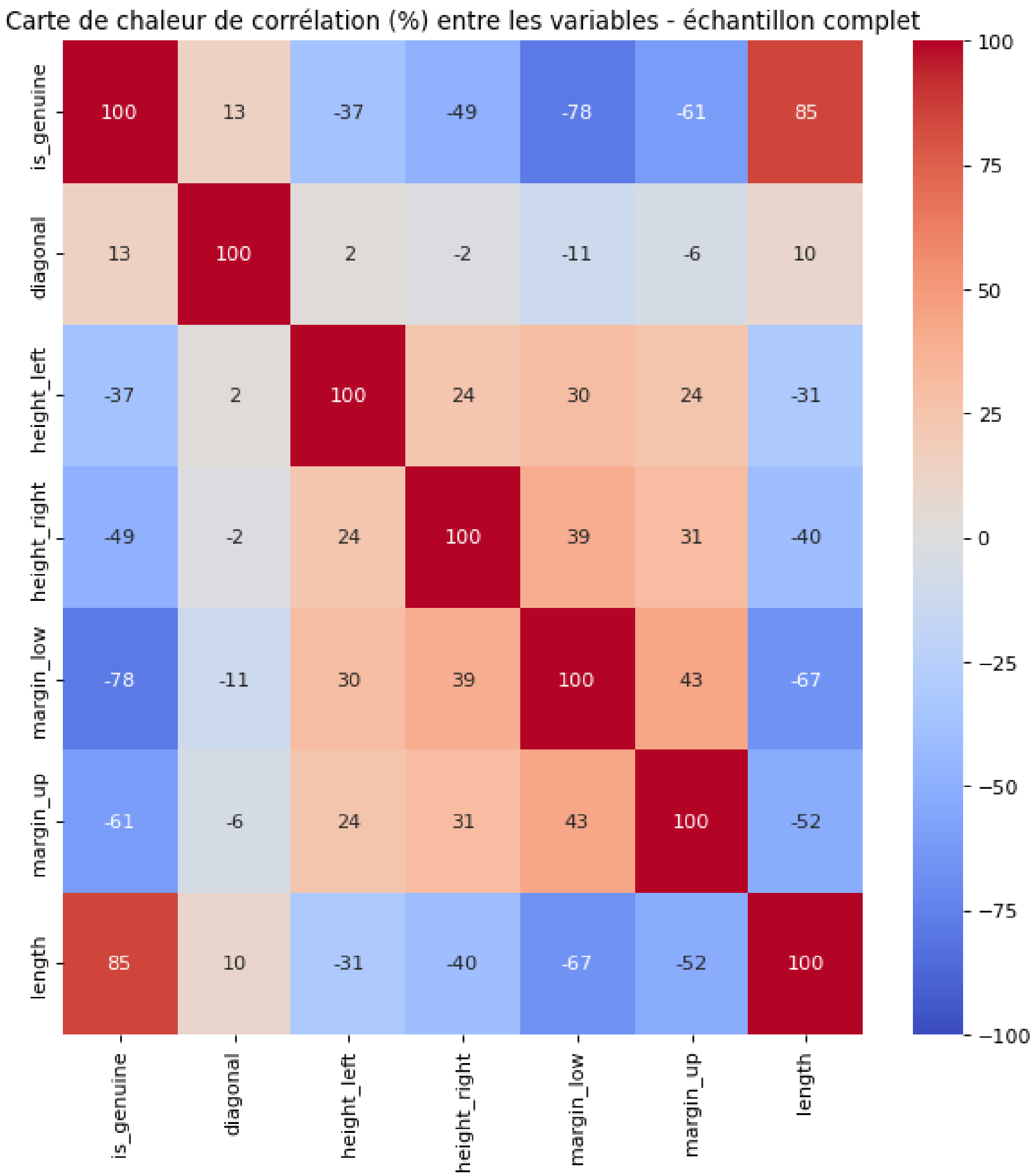

```
Statistique du test de Shapiro-Wilk : 0.9971100757318663
Valeur p : 0.5270688740369139
```

Même commentaire que pour les distributions des marges inférieures.

2.3 - Analyses bivariées

Il ressort des analyses précédentes qu'il y a des différences statistiquement significatives entre les vrais billets et les faux billets. Dans la pratique, cela signifie donc une dépendance des autres variables par rapport à la variable *is_genuine*, et que nous avons tout intérêt à traiter séparément le sous-échantillon des vrais billets du sous-échantillon des faux billets, en particulier si on veut déterminer les valeurs manquantes de mesures de marge inférieure.

[44]:
```
fig = plt.figure(figsize=(9,9), dpi=80)

sb.heatmap(100*df_ini.dropna().corr(method='pearson'),\
           vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
#plt.xticks(rotation=45)
plt.title("Carte de chaleur de corrélation (%) entre les variables -␣
 ↪échantillon complet")
plt.savefig("Heatmap_corr_both.png")
plt.show()
```

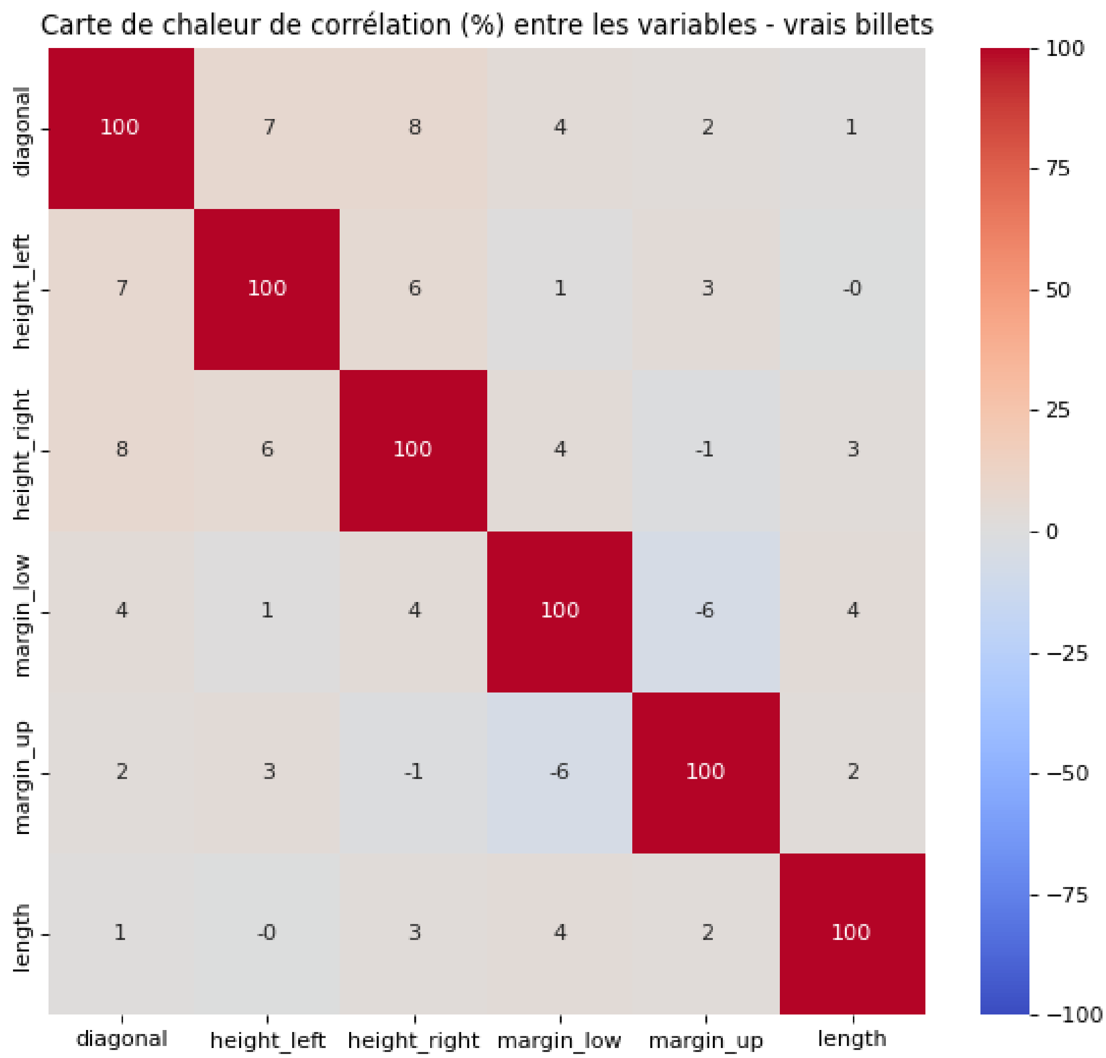


Cette carte de chaleur confirme que la variable *is_genuine* biaise l'analyse des données : il faut scinder l'échantillon complet des données.

2.3.1 - Vrais billets

[45]:
```
fig = plt.figure(figsize=(9,8), dpi=80)

sb.heatmap(100*df_ini1[list(df_ini1.columns)[1:]].dropna().
 ↪corr(method='pearson'),\
            vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
```

```
#plt.xticks(rotation=45)
plt.title("Carte de chaleur de corrélation (%) entre les variables - vrais␣
 ↪billets")
plt.savefig("Heatmap_corr_true.png")
plt.show()
```

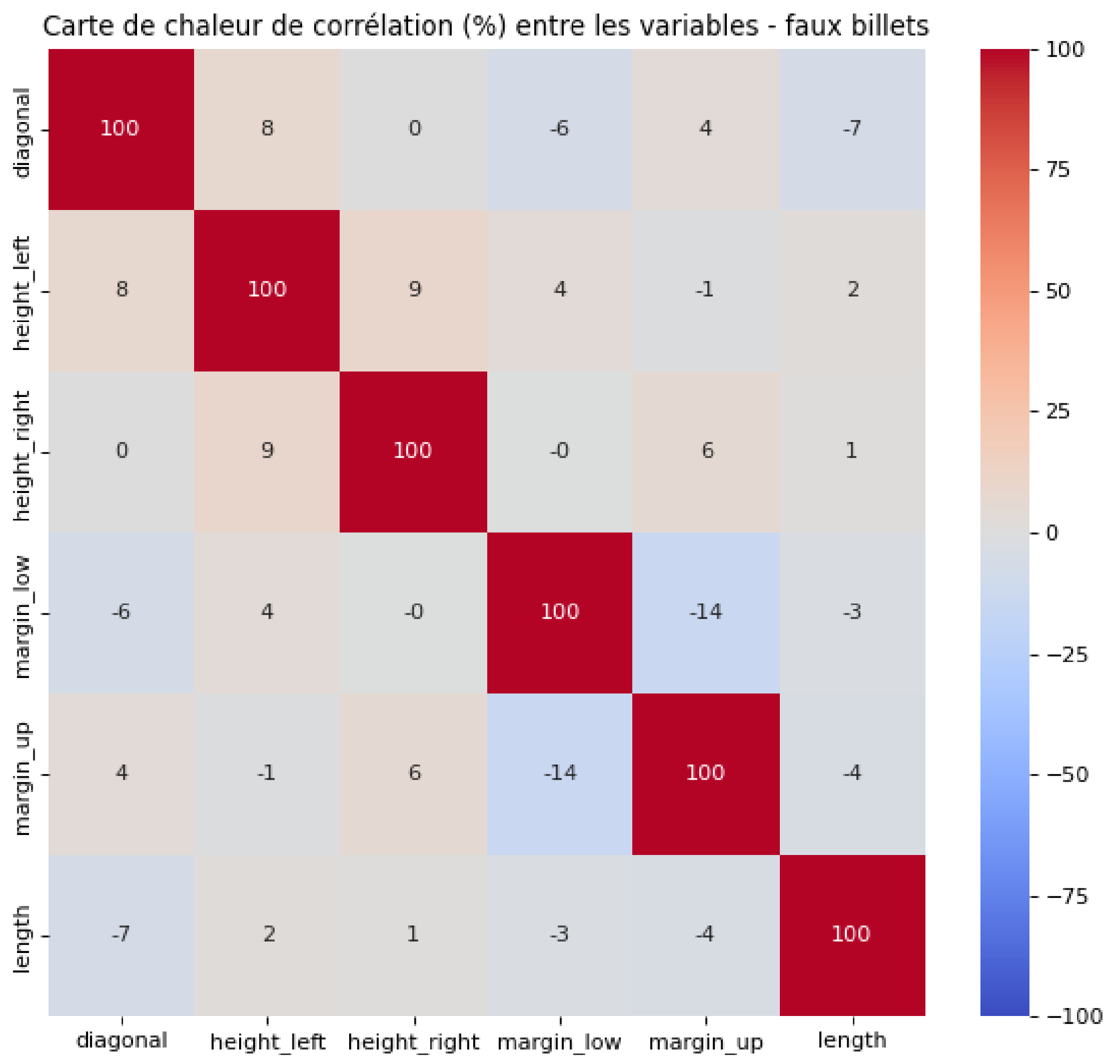


Parmi les vrais billets (et sans tenir compte des billets pour lesquels on a des données manquantes), on constate que la corrélation linéaire (Pearson) entre les vriables est très faible. On peut raisonnablement affirmer qu'au sein de ce sous-échantillon, les variables sont indépendantes entre elles.

2.3.2 - Faux billets

[46]:
```
fig = plt.figure(figsize=(9,8), dpi=80)
```

```
sb.heatmap(100*df_ini0[list(df_ini0.columns)[1:]].dropna().
 ↪corr(method='pearson'),\
          vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
#plt.xticks(rotation=45)
plt.title("Carte de chaleur de corrélation (%) entre les variables - faux␣
 ↪billets")
plt.savefig("Heatmap_corr_false.png")
plt.show()
```



Parmi les vrais billets (et sans tenir compte des billets pour lesquels on a des données manquantes), on constate que la corrélation linéaire (Pearson) entre les vriables est très faible. On peut raisonnablement affirmer qu'au sein de ce sous-échantillon, les variables sont indépendantes entre elles.

2.4 - Analyse en Composantes Principales (ACP)

```
[47]: df_acp = df_ini.dropna()
# Centrage et réduction des distributions
scaler_acp = StandardScaler(with_std=True)
scaler_acp.fit(df_acp)

scaled_acp = scaler_acp.transform(df_acp)
scaled_acp = pd.DataFrame(scaled_acp, columns=list(df_ini.columns))

idx = ["mean", "std"]
display(scaled_acp.describe().round(3).loc[idx, :])
```

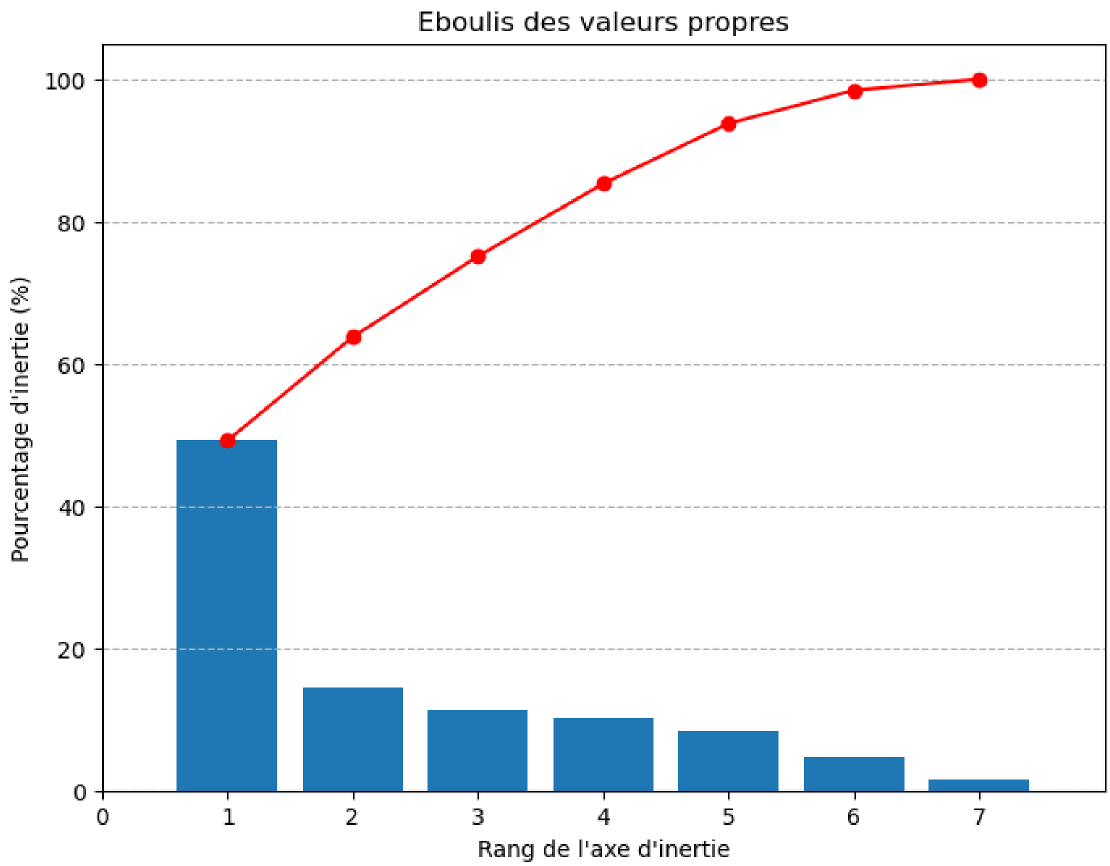
```
      is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
mean        -0.0       0.0          0.0           0.0         0.0       -0.0
std          1.0       1.0          1.0           1.0         1.0        1.0

      length
mean    -0.0
std      1.0
```

```
[48]: n = df_acp.shape[1]
pca = PCA(n_components=n)
pca.fit(scaled_acp)
```

```
[48]: PCA(n_components=7)
```

```
[49]: var = pca.explained_variance_ratio_
cumsum_var = np.cumsum(var)
display(cumsum_var)
d_list = range(1, n+1)
list(d_list)

#Éboulis des valeurs propres
fig, ax = plt.subplots(figsize=(8, 6))
plt.bar(d_list, 100*var)
plt.plot(d_list, 100*cumsum_var,c="red",marker='o')
plt.xlabel("Rang de l'axe d'inertie")
plt.ylabel("Pourcentage d'inertie (%)")
plt.title("Eboulis des valeurs propres")
plt.xticks(np.linspace(0,n,n+1))
plt.grid(axis='y', linestyle='--')
plt.xlim([0,8])
plt.savefig("Eboulis_VP.png")
plt.show(block=False)
```

```
array([0.4916298 , 0.63745968, 0.75106972, 0.85313901, 0.93768911,
       0.9843458 , 1.        ])
```

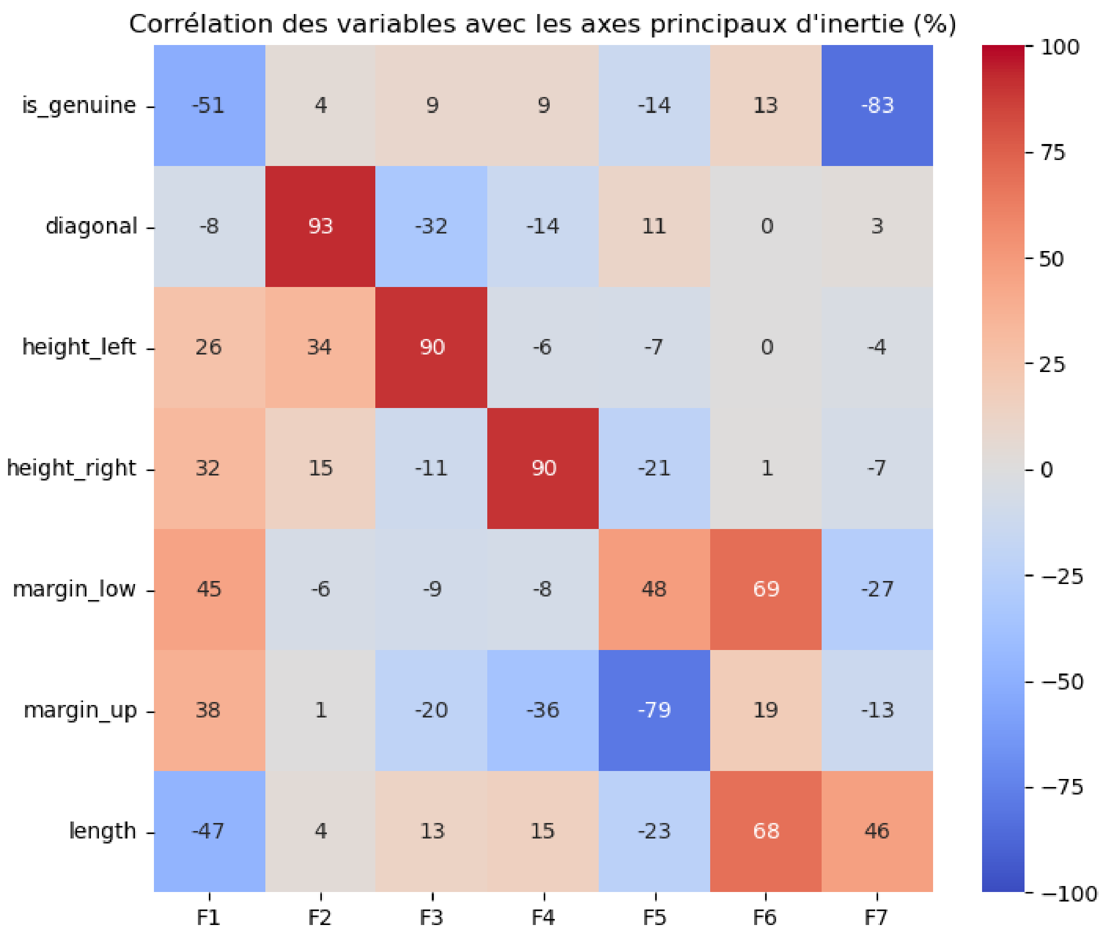


[50]:
```
pcs = pca.components_
df_pcs = pd.DataFrame(pcs)
df_pcs.columns = df_ini.columns
df_pcs.index = [f"F{i}" for i in d_list]
display(df_pcs.round(2))
```

```
    is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
F1       -0.51     -0.08         0.26          0.32        0.45       0.38
F2        0.04      0.93         0.34          0.15       -0.06       0.01
F3        0.09     -0.32         0.90         -0.11       -0.09      -0.20
F4        0.09     -0.14        -0.06          0.90       -0.08      -0.36
F5       -0.14      0.11        -0.07         -0.21        0.48      -0.79
F6        0.13      0.00         0.00          0.01        0.69       0.19
F7       -0.83      0.03        -0.04         -0.07       -0.27      -0.13

    length
F1   -0.47
F2    0.04
F3    0.13
F4    0.15
```

```
F5   -0.23
F6    0.68
F7    0.46
```

[51]:
```
fig, ax = plt.subplots(figsize=(8, 7))
sb.heatmap(100*df_pcs.T, vmin=-100, vmax=100, annot=True, cmap="coolwarm",␣
 ↪fmt=".0f")
plt.title("Corrélation des variables avec les axes principaux d'inertie (%)")
plt.savefig("Correlation_inertia_axes.png")
```



[52]:
```
######

def correlation_graph(pca,
                      x_y,
                      features,
                      fig_name) :
    """Affiche le graphe des correlations
```

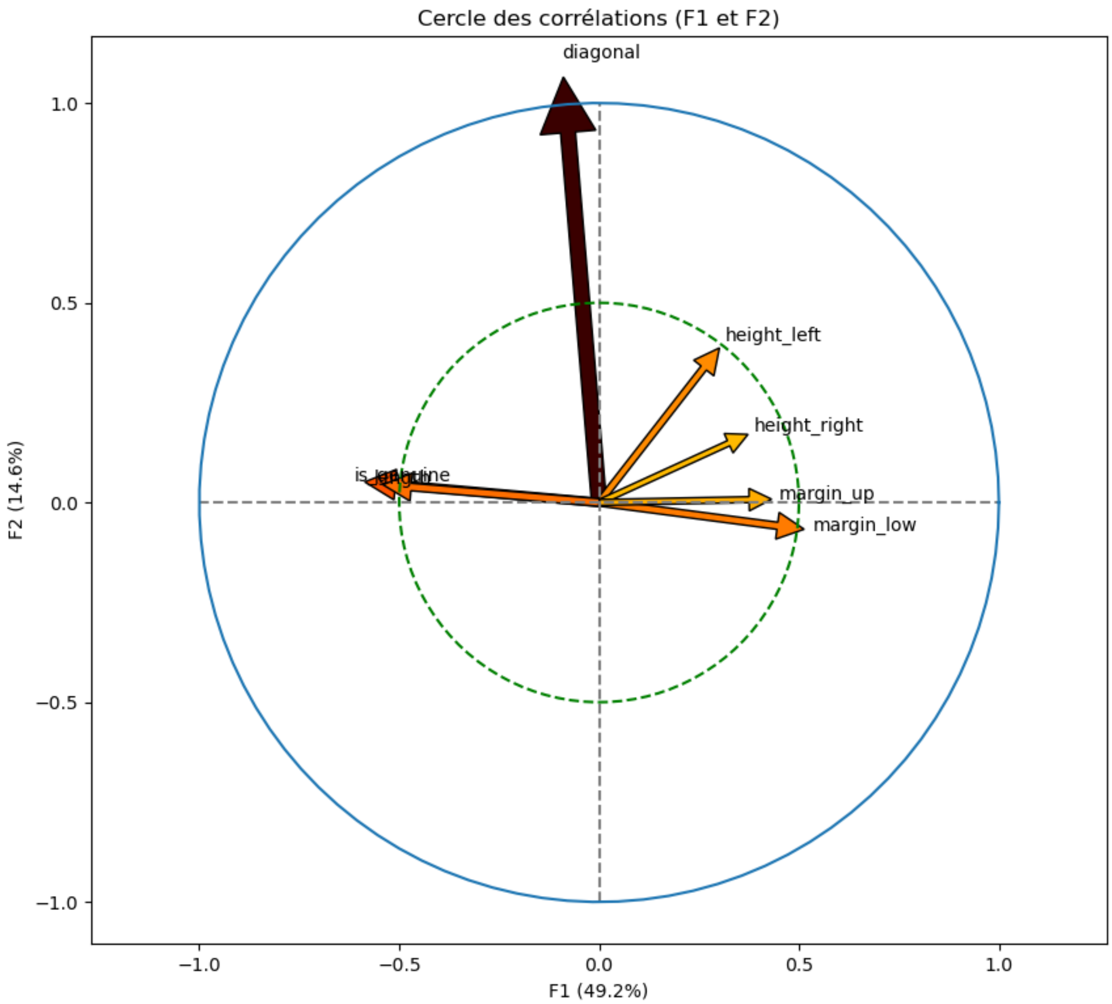
```
    Positional arguments :
    -----------------------------------
    pca : sklearn.decomposition.PCA : notre objet PCA qui a été fit
    x_y : list ou tuple : le couple x,y des plans à afficher, exemple [0,1]␣
↪pour F1, F2
    features : list ou tuple : la liste des features (ie des dimensions) à␣
↪représenter
    fig_name : nom du fichier image pour le graphique généré
    """

    # Extrait x et y
    x,y=x_y

    # Taille de l'image (en inches)
    fig, ax = plt.subplots(figsize=(10, 9))

    # Pour chaque composante :
    for i in range(0, pca.components_.shape[1]):

        v_norm = np.sqrt((pca.components_[x, i])**2 + (pca.components_[y,␣
↪i])**2)
        print("Norme vecteur",features[i],"dans cercle des corrélations :
↪",round(v_norm, 3))

        # Les flèches
        ax.arrow(0,0,
                pca.components_[x, i],
                pca.components_[y, i],
                head_width=0.15*v_norm,
                head_length=0.15*v_norm,
                width=0.04*v_norm,
                color=cm["hot"](1-v_norm), ec='black')

        # Les labels
        plt.text(pca.components_[x, i] + 0.2*pca.components_[x, i],
                pca.components_[y, i] + 0.2*pca.components_[y, i],
                features[i])

    # Affichage des lignes horizontales et verticales
    plt.plot([-1, 1], [0, 0], color='grey', ls='--')
    plt.plot([0, 0], [-1, 1], color='grey', ls='--')

    # Nom des axes, avec le pourcentage d'inertie expliqué
    plt.xlabel('F{} ({}%)'.format(x+1, round(100*pca.
↪explained_variance_ratio_[x],1)))
    plt.ylabel('F{} ({}%)'.format(y+1, round(100*pca.
↪explained_variance_ratio_[y],1)))
```

```python
    # J'ai copié collé le code sans le lire
    plt.title("Cercle des corrélations (F{} et F{})".format(x+1, y+1))

    # Le cercle
    an = np.linspace(0, 2 * np.pi, 100)
    plt.plot(np.cos(an), np.sin(an))
    plt.plot(0.5*np.cos(an), 0.5*np.sin(an), linestyle='--', color='green')

    # Axes et display
    plt.axis('equal')

    # Enregistrement figure
    plt.savefig(fig_name)

    plt.show(block=False)

######
```

[53]:

```python
correlation_graph(pca, x_y=(0,1), features=list(df_pcs.columns),␣
 ↪fig_name="Factoriel_1st.png")
```

```
Norme vecteur is_genuine dans cercle des corrélations : 0.512
Norme vecteur diagonal dans cercle des corrélations : 0.929
Norme vecteur height_left dans cercle des corrélations : 0.426
Norme vecteur height_right dans cercle des corrélations : 0.356
Norme vecteur margin_low dans cercle des corrélations : 0.45
Norme vecteur margin_up dans cercle des corrélations : 0.375
Norme vecteur length dans cercle des corrélations : 0.472
```



Partie 3 - Détermination des valeurs manquantes par régression linéaire

3.1 - Échantillon complet

3.1.1 - Exploration des modèles

[54]:
```
df = df_ini.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles non-vides de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
↪len(L_var)+1))))

score_max = 0
best_set = set()
```

```
for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]
    # la variable cible, la marge inférieure
    y = df["margin_low"]
    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)
```

```
Meilleur score R² obtenu : 0.4773366973063957
pour le jeu de variables ('diagonal', 'height_left', 'height_right',
'margin_up', 'length')
```

[55]:

```
# Meilleur modèle avec tous les prédicteurs possibles
df = df_ini.dropna()

L_var =␣
 ↪['is_genuine','diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles non-vides de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
 ↪len(L_var)+1))))

score_max6 = 0
best_set6 = set()
best_coef6 = []

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
```

```
    # les variables prédictives
    X = df[pl]
    # la variable cible, la marge inférieure
    y = df["margin_low"]
    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max6 :
        score_max6 = reg.score(X, y)
        best_set6 = ps
        best_coef6 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max6}")
print("pour le jeu de variables", best_set6)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef6)

# Valeur à l'origine
f0 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set6)].mean(),␣
 ↪best_coef6)]))
print(f"Valeur à l'origine : {f0}")
```

```
Meilleur score R² obtenu : 0.6168760755671029
pour le jeu de variables ('is_genuine', 'diagonal', 'height_left',
'height_right', 'margin_up', 'length')
Coefficients de la régression : [-1.14059676 -0.0130159   0.02829069  0.02674982
-0.21284432 -0.00388299]
Valeur à l'origine : 2.8668228920543664
```

Ce résultat est déconcertant : cela signifie que la régression linéaire est plus efficace si on tient compte de la variable *is_genuine* alors que celle-ci biaise grandement l'échantillon complet.

[56]:
```
# Meilleur modèle à 1 prédicteur
df = df_ini.dropna()

L_var =␣
 ↪['is_genuine','diagonal','height_left','height_right','margin_up','length']
```

```python
# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1, 2))))

score_max = 0
best_set = set()
best_coef = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps
        best_coef = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef)
print(f"Valeur à l'origine : {(y.mean()-sum([i*j for (i, j) in zip(df[list(best_set)].mean(), best_coef)]))}")
```

```
Meilleur score R² obtenu : 0.6131393378084237
pour le jeu de variables ('is_genuine',)
Coefficients de la régression : [-1.09983815]
Valeur à l'origine : 5.215934959349594
```

[57]:
```python
# Meilleur modèle à 2 prédicteurs
df = df_ini.dropna()

L_var =␣
↪['is_genuine','diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(2,␣
↪3))))

score_max2 = 0
best_set2 = set()
best_coef2 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max2 :
        score_max2 = reg.score(X, y)
        best_set2 = ps
        best_coef2 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max2}")
print("pour le jeu de variables", best_set2)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef2)

# Valeur à l'origine
```

```
b0 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set2)].mean(),␣
 ↪best_coef2)]))
print(f"Valeur à l'origine : {b0}")
```

```
Meilleur score R² obtenu : 0.616565858990371
pour le jeu de variables ('is_genuine', 'margin_up')
Coefficients de la régression : [-1.16319991 -0.21194039]
Valeur à l'origine : 5.926254037548741
```

[58]:

```
# Meilleur modèle à 3 prédicteurs
df = df_ini.dropna()

L_var =␣
 ↪['is_genuine','diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 3 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(3,␣
 ↪4))))

score_max = 0
best_set3 = set()
best_coef3 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set3 = ps
        best_coef3 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
```

```
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set3)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef3)
print(f"Valeur à l'origine : {(y.mean()-sum([i*j for (i, j) in␣
 ↪zip(df[list(best_set3)].mean(), best_coef3)]))}")
```

```
Meilleur score R² obtenu : 0.6167129526058592
pour le jeu de variables ('is_genuine', 'height_left', 'margin_up')
Coefficients de la régression : [-1.15661812  0.02897617 -0.21288181]
Valeur à l'origine : 2.9104242563189544
```

3.1.2 - Comparaison de modèles avec le test F

[59]:
```
df = df.astype({"is_genuine": int})
display(df.groupby("is_genuine").count())
```

is_genuine	diagonal	height_left	height_right	margin_low	margin_up	length
0	492	492	492	492	492	492
1	971	971	971	971	971	971

[60]:
```
# Comparaison modèles {valeur constante} vs. {is_genuine}
# Hypothèse nulle H0 : le coefficient associé au prédicteur is_genuine égale 0
X1 = sm.add_constant([df["margin_low"].mean()]*len(df))   # Modèle 1
X2 = sm.add_constant(df[["is_genuine"]])                  # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(2315.553531505102, 1.3620340580582373e-303, 1.0)
```

On a une p-valeur bien inférieure à 0.05, ce qui nous invite à rejeter H0 : prendre en compte *is_genuine* a un impact statistique siginficatif (H1).

[61]:
```
# Comparaison modèles {is_genuine} vs. {is_genuine, margin_up}
# Hypothèse H2 : le coefficient associé au prédicteur margin_up égale 0
X1 = sm.add_constant(df[["is_genuine"]])               # Modèle 1
X2 = sm.add_constant(df[["is_genuine", "margin_up"]])  # Modèle 2
y = df["margin_low"]
```

```
# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(13.047145234564434, 0.0003140384884724655, 1.0)
```

On a une p-valeur inférieure à 0.05, ce qui nous invite à rejeter H2 : prendre en compte *margin_up* a un impact statistique siginficatif (H3).

[62]:
```
# Comparaison modèles {is_genuine, margin_up} vs. {is_genuine, margin_up,␣
↪height_left}
# Hypothèse H4 : le coefficient associé au prédicteur height_left égale 0
X1 = sm.add_constant(df[["is_genuine", "margin_up"]])                # Modèle 1
X2 = sm.add_constant(df[["is_genuine", "margin_up", "height_left"]])  # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(0.5599186992003954, 0.45441360111375095, 1.0)
```

On a une p-valeur supérieure à 0.05, ce qui nous invite à ne pas rejeter H4 : prendre en compte *height_left* ou tout autre variable (en plus de *is_genuine* et *margin_up*) n'a pas d'impact statistique siginficatif.

[63]:
```
# Comparaison modèles {is_genuine, margin_up} vs. {is_genuine, margin_up,␣
↪height_left}
# Hypothèse : les coefficients associés aux prédicteurs autres que is_genuine␣
↪et margin_up égalent 0
X1 = sm.add_constant(df[["is_genuine", "margin_up"]])                # Modèle 1
X2 = sm.
↪add_constant(df[['is_genuine','diagonal','height_left','height_right','margin_up','length']
↪ # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
```

```
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(0.29473187845830445, 0.881501454669415, 4.0)
```

On observe que l'essentiel de la variance de la variable *margin_low* s'explique par la variable *is_genuine* - et dans une moindre mesure par *margin_up* - ce que montraient déjà nos cartes de chaleur des sous-échantillons. intégrer les autres variables dans un modèle de régression linéaire sur l'échantillon complet n'aurait pas d'impact statistique significatif.

On a donc 2 modèles de régression linéaire : un modèle compact avec 2 prédicteurs, et un modèle exhaustif avec 6 prédicteurs.

[64]:
```
print("Prédicteurs modèle compact :", best_set2)
print("Coefficients modèle compact :", best_coef2)
print("Valeur à l'origine :",b0)
print("Score R² :", score_max2)
```

```
Prédicteurs modèle compact : ('is_genuine', 'margin_up')
Coefficients modèle compact : [-1.16319991 -0.21194039]
Valeur à l'origine : 5.926254037548741
Score R² : 0.616565858990371
```

[65]:
```
df_ini["pred2"] = round(df_ini["is_genuine"]*best_coef2[0] +␣
 ↪df_ini["margin_up"]*best_coef2[1] + b0, 2)
```

[66]:
```
print("Prédicteurs modèle exhaustif :", best_set6)
print("Coefficients modèle exhaustif :", best_coef6)
print("Valeur à l'origine :",f0)
print("Score R² :", score_max6)
```

```
Prédicteurs modèle exhaustif : ('is_genuine', 'diagonal', 'height_left',
'height_right', 'margin_up', 'length')
Coefficients modèle exhaustif : [-1.14059676 -0.0130159   0.02829069  0.02674982
-0.21284432 -0.00388299]
Valeur à l'origine : 2.8668228920543664
Score R² : 0.6168760755671029
```

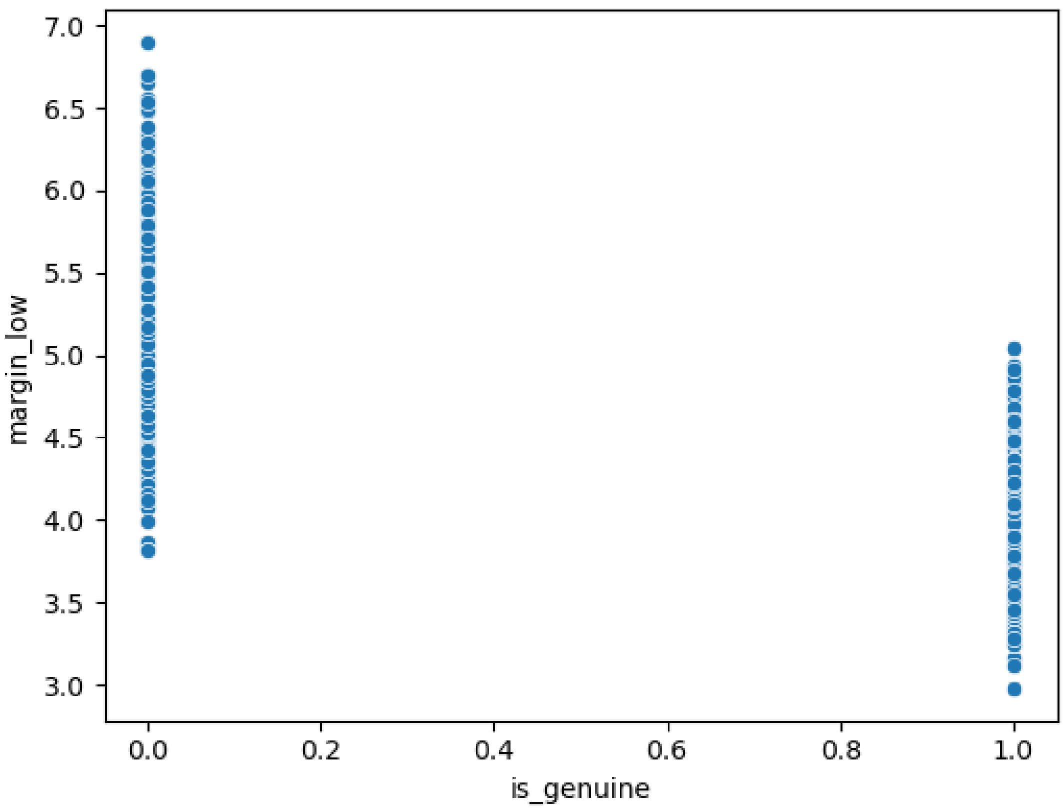
[67]:
```
df_ini["pred6"] = round(df_ini["is_genuine"]*best_coef6[0] +␣
 ↪df_ini["diagonal"]*best_coef6[1] +\
                        df_ini["height_left"]*best_coef6[2] +␣
 ↪df_ini["height_right"]*best_coef6[3] +\
                        df_ini["margin_up"]*best_coef6[4] +␣
 ↪df_ini["length"]*best_coef6[5] + f0, 2)
```

3.1.3 - Vérification des hypothèses de régression

Linéarité de la variable prédite vis-à-vis des prédicteurs

[68]:
```
# Vis-à-vis de is_genuine
sb.scatterplot(data=df_ini.dropna(), x="is_genuine", y="margin_low")
```

[68]:
```
<Axes: xlabel='is_genuine', ylabel='margin_low'>
```



Linéarité pas particulièrement visible vu que *is_genuine* est à valeurs discrètes.

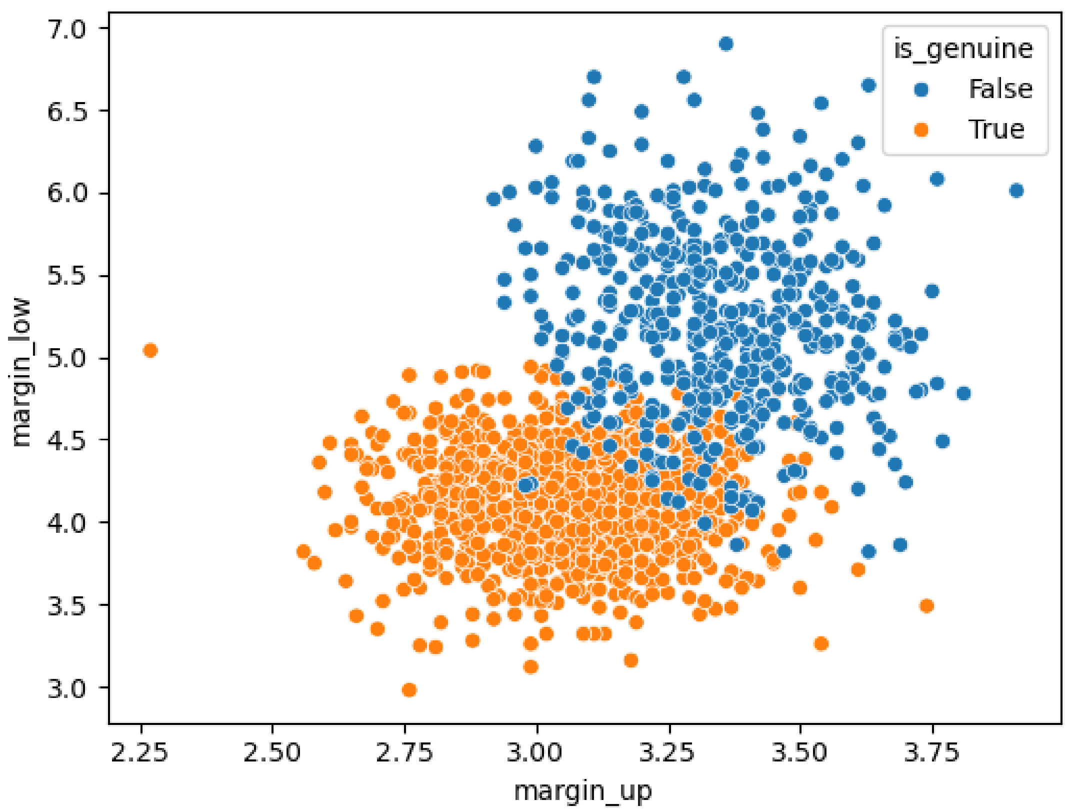
[69]:
```
# Hypothèse nulle H0 : les variables ne sont pas linéairement corrélées
# Calculer le coefficient de corrélation de Pearson et la valeur p
pearson_corr, pearson_p_value = st.pearsonr(x=df_ini.dropna()["is_genuine"],␣
↪y=df_ini.dropna()["margin_low"])

print(f"Coefficient de corrélation de Pearson : {pearson_corr}")
print(f"Valeur p : {pearson_p_value }")

# Interprétation des résultats
alpha = 0.05
if pearson_p_value > alpha:
    print("\nLes variables ne sont pas corrélées (on ne rejette pas H0)")
else:
```

```
    print("\nLes variables sont corrélées (on rejette H0)")
```

```
Coefficient de corrélation de Pearson : -0.7830321435346212
Valeur p : 1.3620340580586928e-303

Les variables sont corrélées (on rejette H0)
```

[70]:
```
# Vis-à-vis de margin_up
sb.scatterplot(data=df_ini.dropna(), x="margin_up", y="margin_low",␣
 ↪hue="is_genuine")
```

[70]: <Axes: xlabel='margin_up', ylabel='margin_low'>



On discerne une tendance linéaire globale.

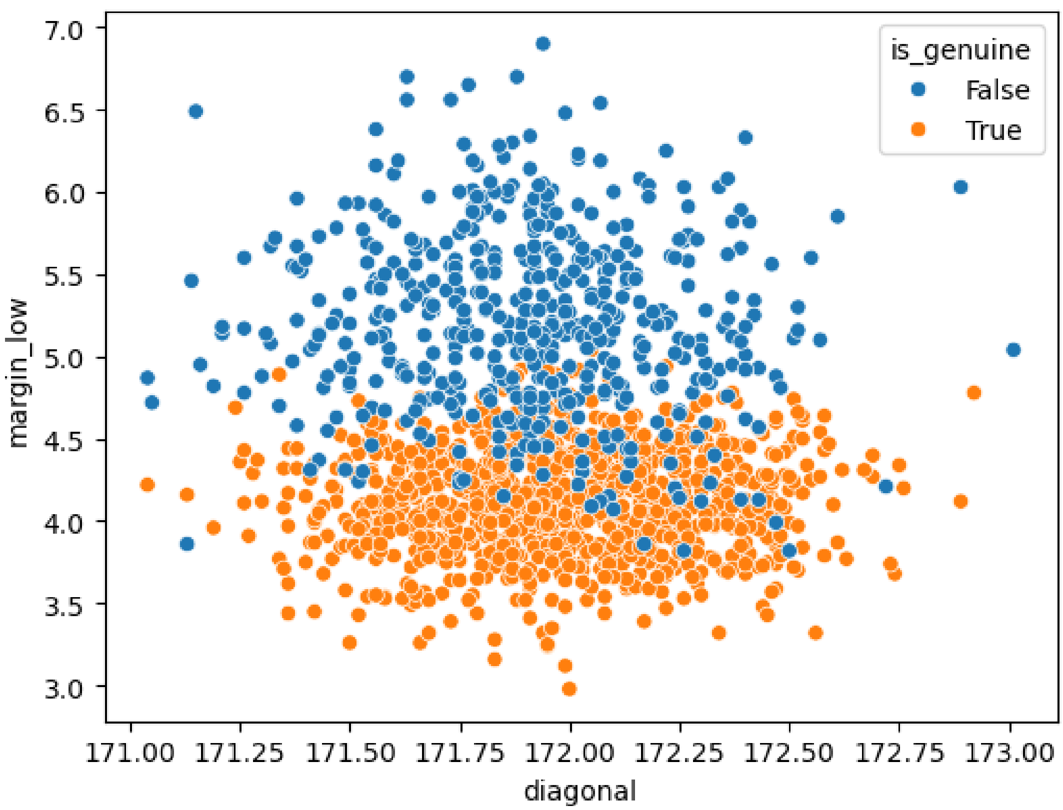
[71]:
```
# Hypothèse nulle H0 : les variables ne sont pas linéairement corrélées
# Calculer le coefficient de corrélation de Pearson et la valeur p
pearson_corr, pearson_p_value = st.pearsonr(x=df_ini.dropna()["margin_up"],␣
 ↪y=df_ini.dropna()["margin_low"])

print(f"Coefficient de corrélation de Pearson : {pearson_corr}")
print(f"Valeur p : {pearson_p_value }")
```

```
# Interprétation des résultats
alpha = 0.05
if pearson_p_value > alpha:
    print("\nLes variables ne sont pas corrélées (on ne rejette pas H0)")
else:
    print("\nLes variables sont corrélées (on rejette H0)")
```

```
Coefficient de corrélation de Pearson : 0.4316060733203143
Valeur p : 1.92160401830609e-67

Les variables sont corrélées (on rejette H0)
```

[72]:
```
# Vis-à-vis de diagonal
sb.scatterplot(data=df_ini.dropna(), x="diagonal", y="margin_low",␣
 ↪hue="is_genuine")
```

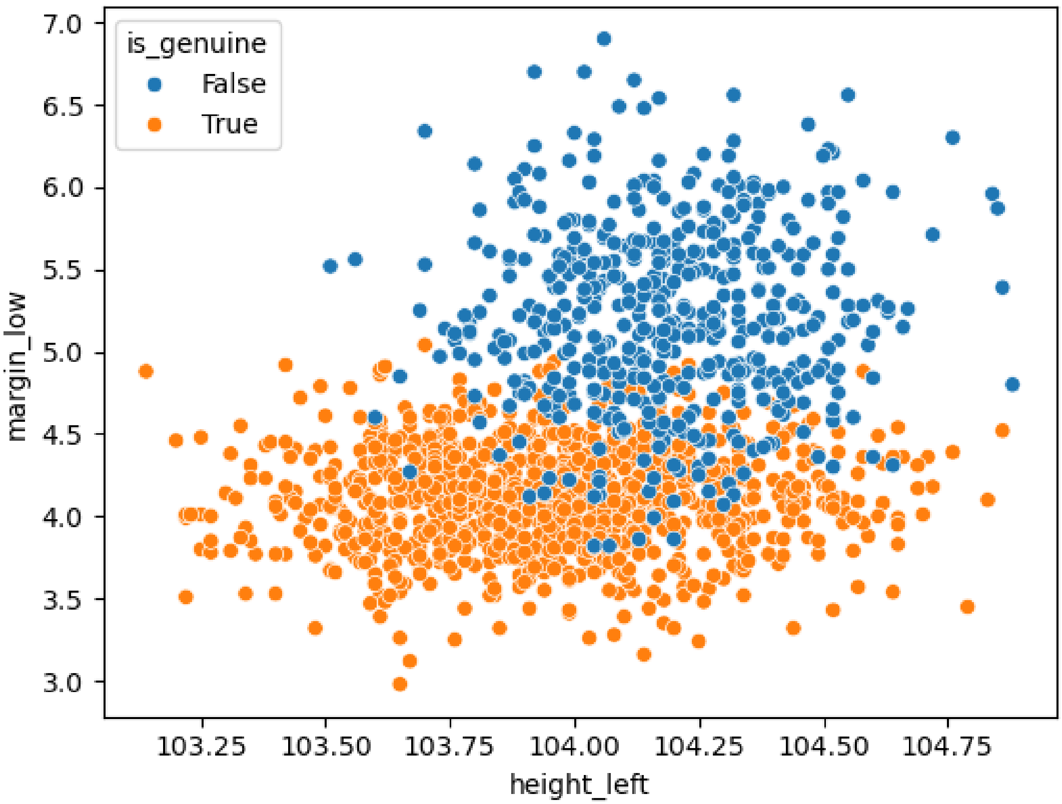
[72]: `<Axes: xlabel='diagonal', ylabel='margin_low'>`



Aucune tendance ne se dégage à la vue de ce nuage de points, si ce n'est des amas de points autour de *margin_low* = 4 et de *diagonal* = 172.

[73]:
```
# Vis-à-vis de height_left
sb.scatterplot(data=df_ini.dropna(), x="height_left", y="margin_low",
  hue="is_genuine")
```

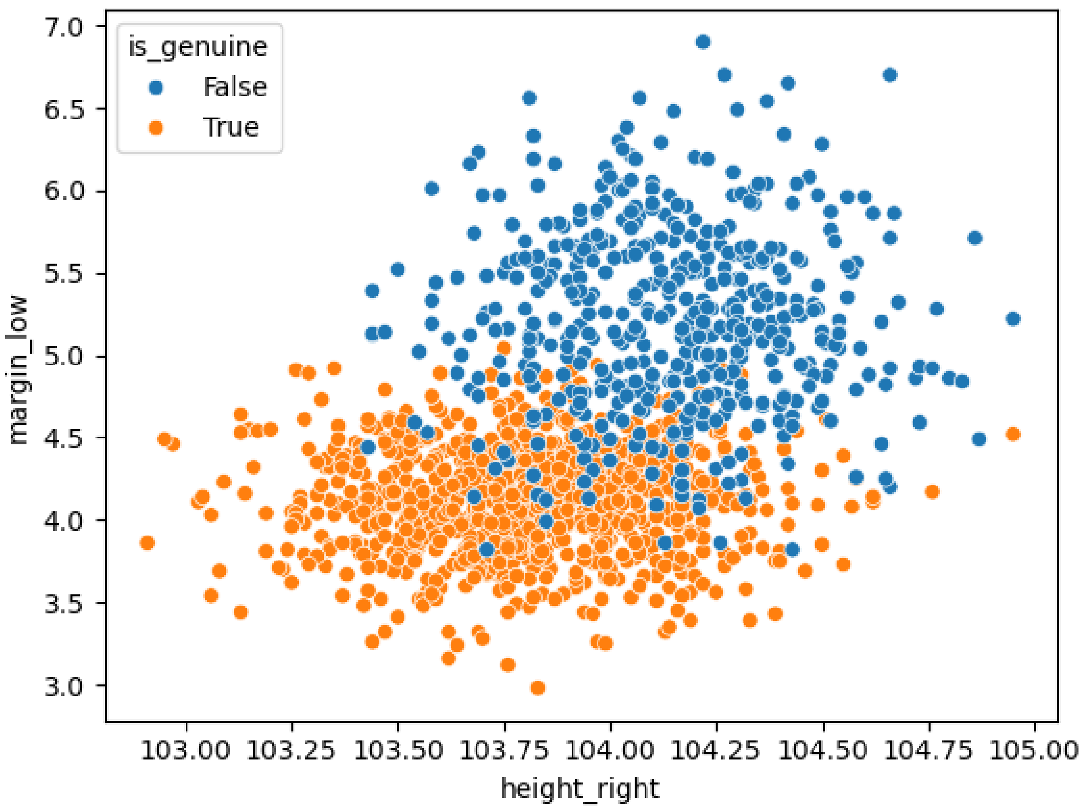
[73]: <Axes: xlabel='height_left', ylabel='margin_low'>



On discerne une tendance linéaire globale.

[74]:
```
# Vis-à-vis de height_right
sb.scatterplot(data=df_ini.dropna(), x="height_right", y="margin_low",
  hue="is_genuine")
```

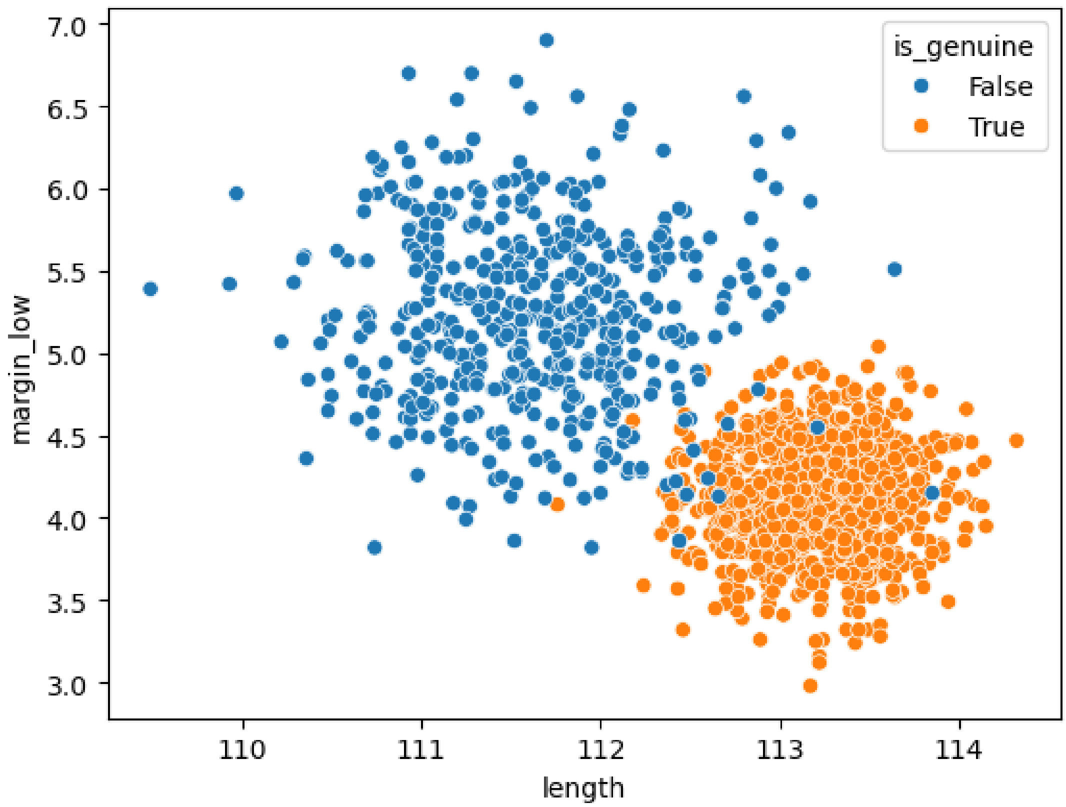
[74]: <Axes: xlabel='height_right', ylabel='margin_low'>



On discerne une tendance linéaire globale.

[75]:
```
# Vis-à-vis de length
sb.scatterplot(data=df_ini.dropna(), x="length", y="margin_low",␣
↪hue="is_genuine")
```

[75]: <Axes: xlabel='length', ylabel='margin_low'>



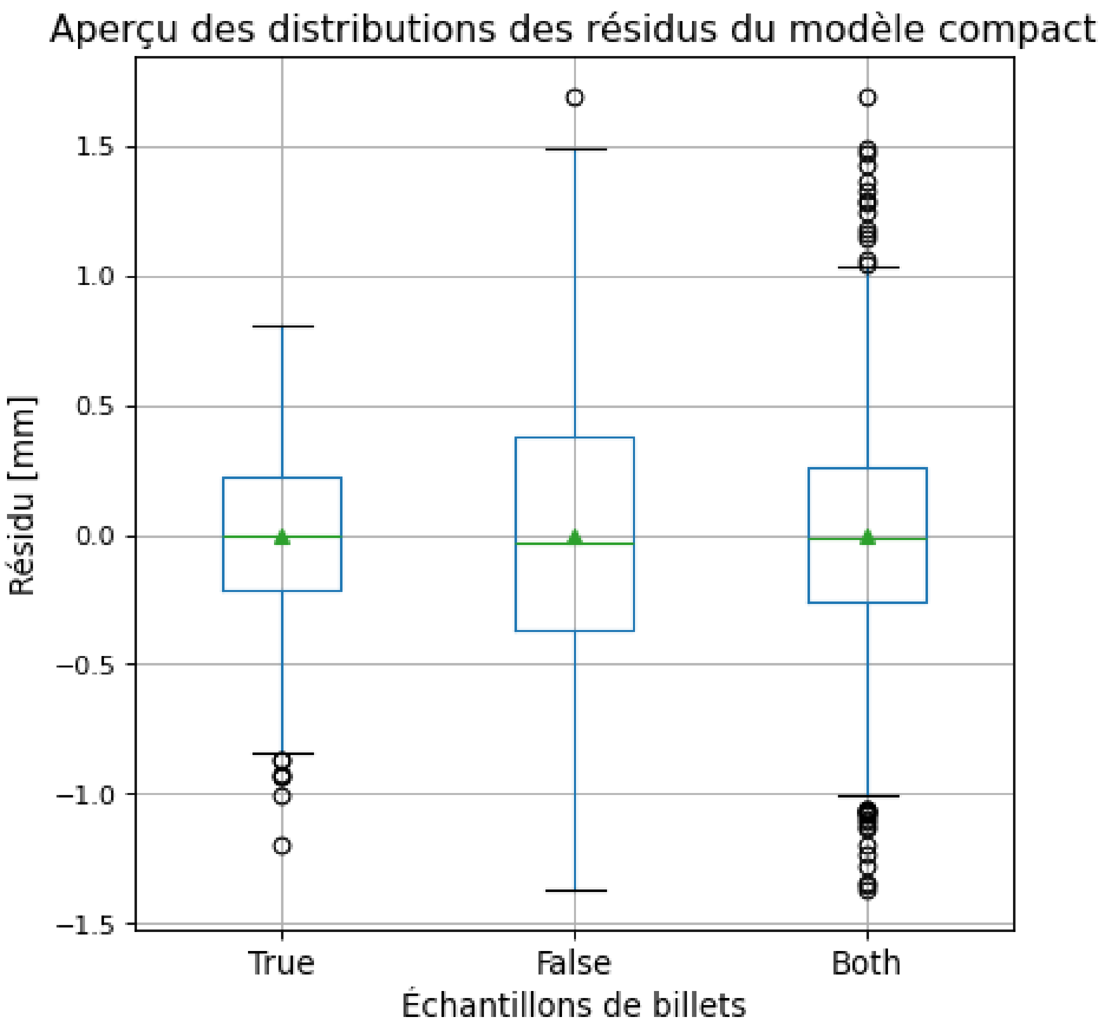
On discerne une tendance linéaire, mais qui se manifeste par 2 amas de points.

Analyse des résidus

[76]:
```
df_ini["res2"] = df_ini["margin_low"] - df_ini["pred2"]
df_ini["res6"] = df_ini["margin_low"] - df_ini["pred6"]

df_ini1 = df_ini.loc[df_ini["is_genuine"]==True].dropna()
df_ini0 = df_ini.loc[df_ini["is_genuine"]==False].dropna()
```

[77]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["res2"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["res2"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["res2"],\
               positions=[3], widths=[0.4],
```

```
                showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Résidu [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions des résidus du modèle compact", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```

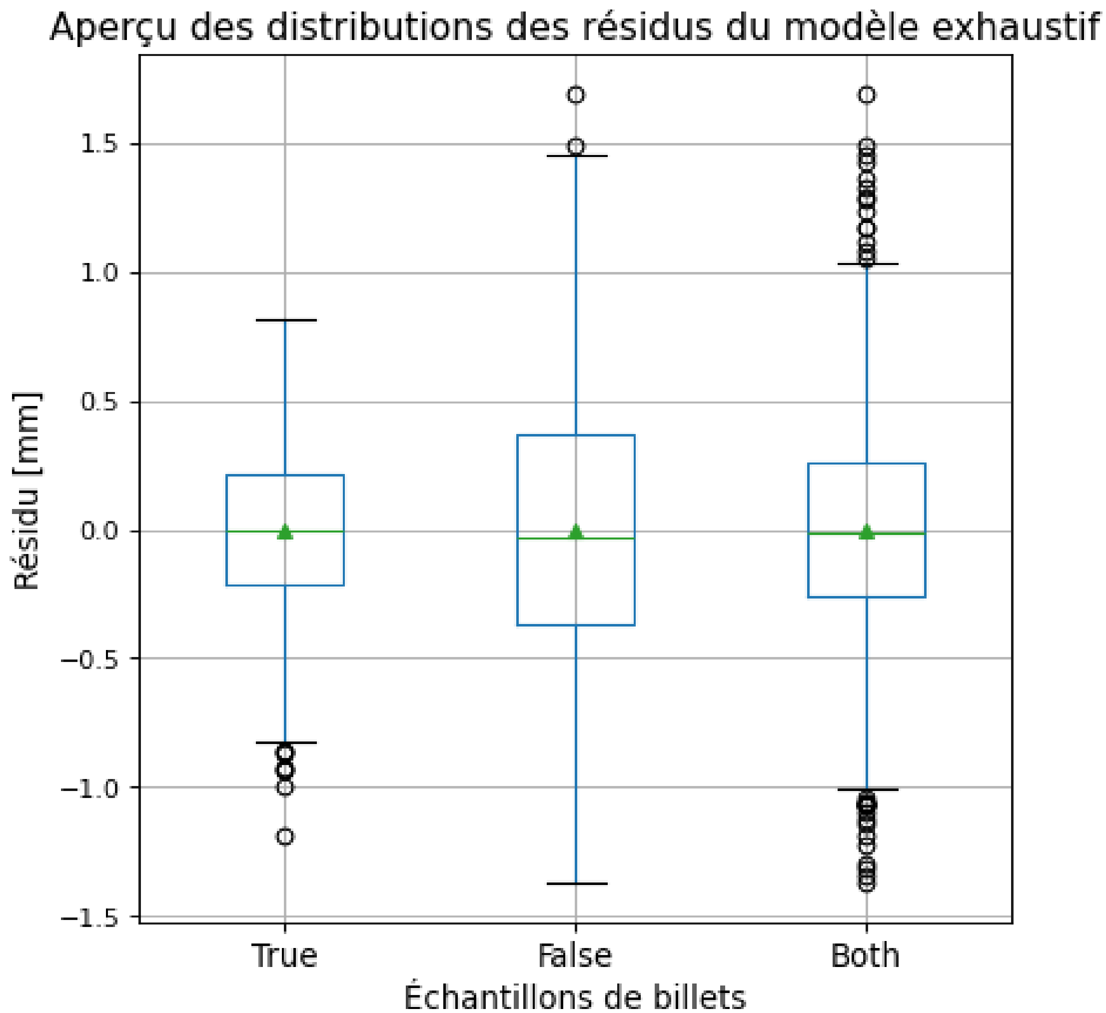


[78]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
 ↪proviennent de la même distribution
res = st.ks_2samp(df_ini1["res2"], df_ini0["res2"], alternative='two-sided')
```

```python
# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

```
Statistique signée du test KS : -0.1386928235914697
Valeur p : 6.010470913951297e-06
```

[79]:
```python
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["res6"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["res6"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["res6"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Résidu [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions des résidus du modèle exhaustif",
 ↪fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



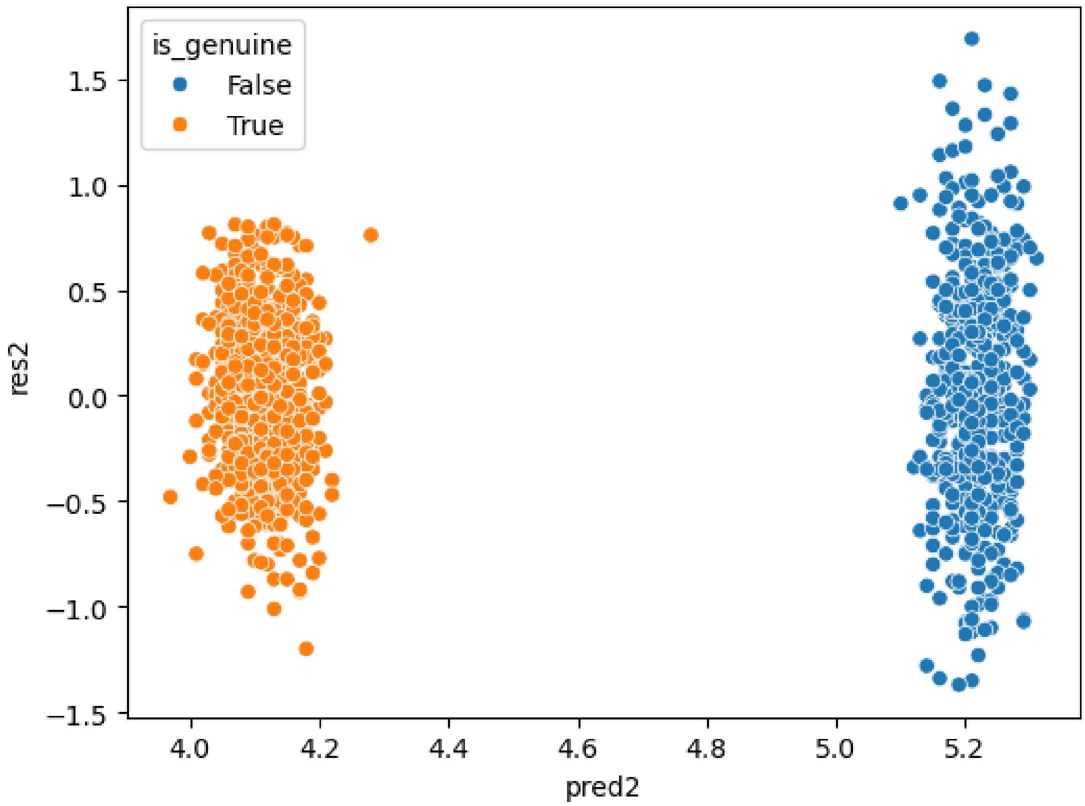
[80]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets␣
↪provienennt de la même distribution
res = st.ks_2samp(df_ini1["res6"], df_ini0["res6"], alternative='two-sided')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

```
Statistique signée du test KS : -0.14665963343464536
Valeur p : 1.3395428833168855e-06
```

La variance des résidus pour les faux billets est manifestement plus grande que pour les vrais billets.

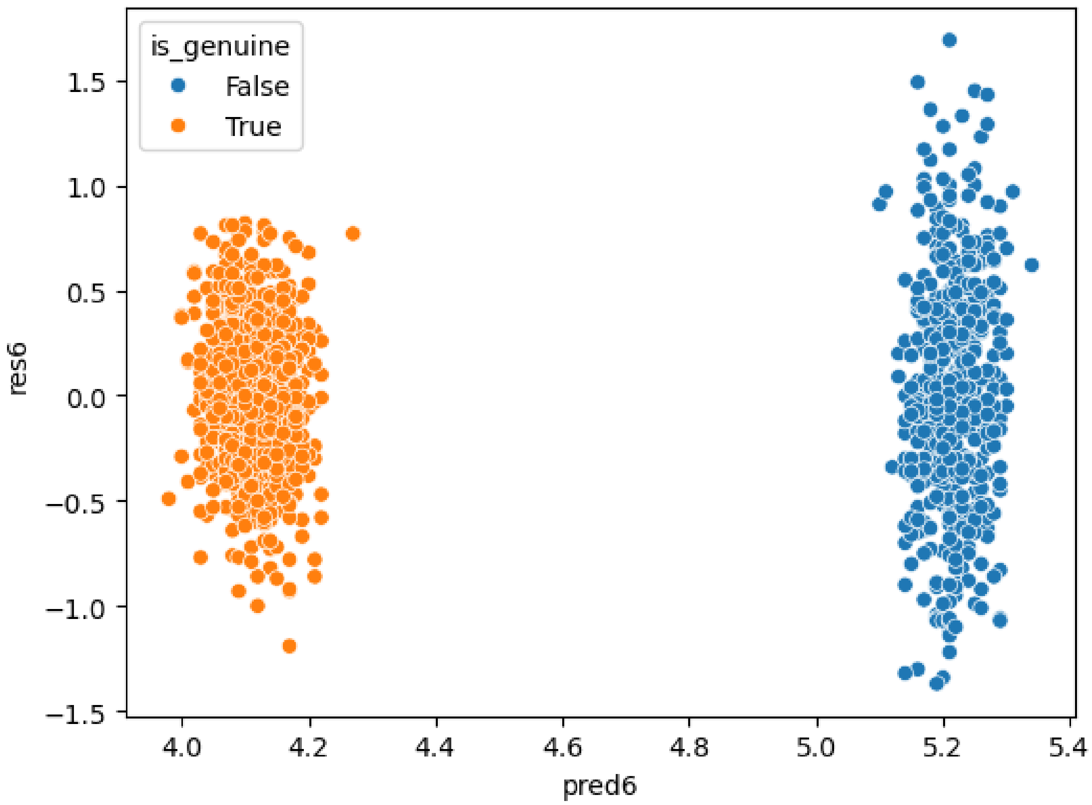
[81]:
```
sb.scatterplot(data=df_ini, x='pred2', y='res2', hue='is_genuine')
```

[81]: <Axes: xlabel='pred2', ylabel='res2'>

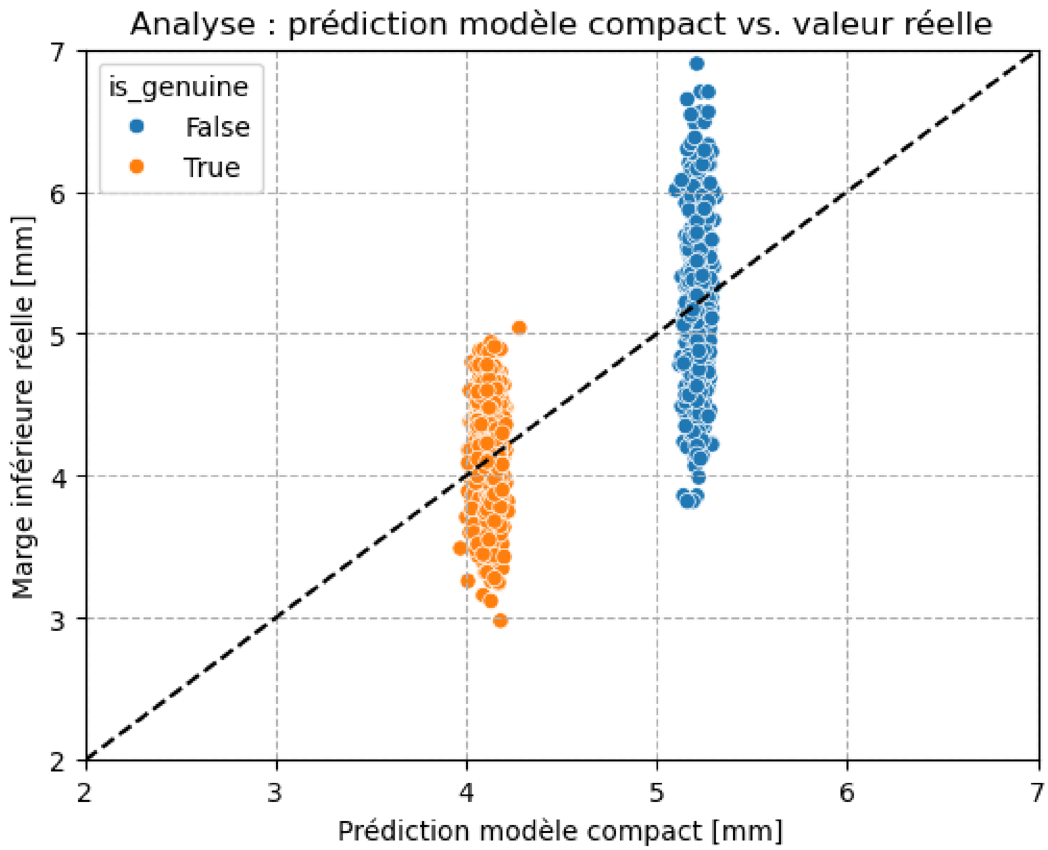


[82]:
```
sb.scatterplot(data=df_ini, x='pred6', y='res6', hue='is_genuine')
```

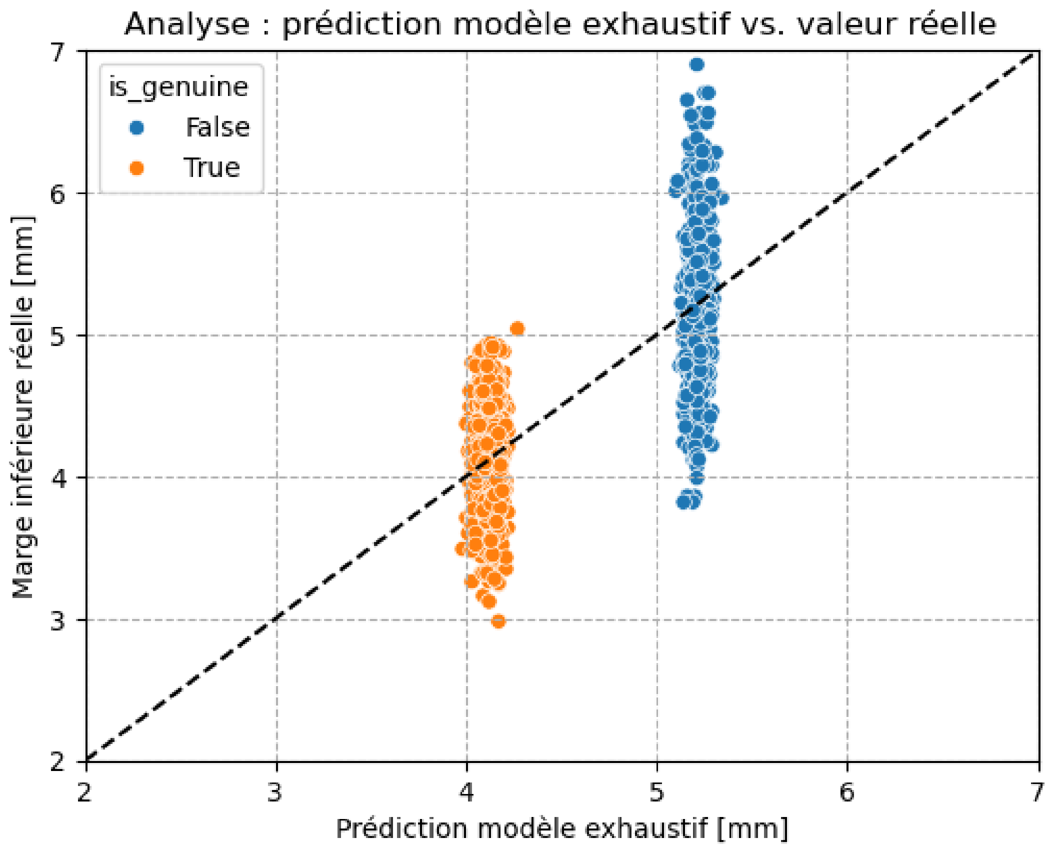
[82]: <Axes: xlabel='pred6', ylabel='res6'>



[83]:
```
sb.scatterplot(data=df_ini, x='pred2', y='margin_low', hue='is_genuine')
plt.plot(np.linspace(2,7,21),np.linspace(2,7,21), color='black',␣
 ↪linestyle='dashed')
plt.title("Analyse : prédiction modèle compact vs. valeur réelle")
plt.xlim([2,7])
plt.ylim([2,7])
plt.grid(visible=True, axis='both', linestyle='--')

plt.xlabel("Prédiction modèle compact [mm]")
plt.ylabel("Marge inférieure réelle [mm]")

#plt.savefig("pred2_vs_real.png")

plt.show()
```



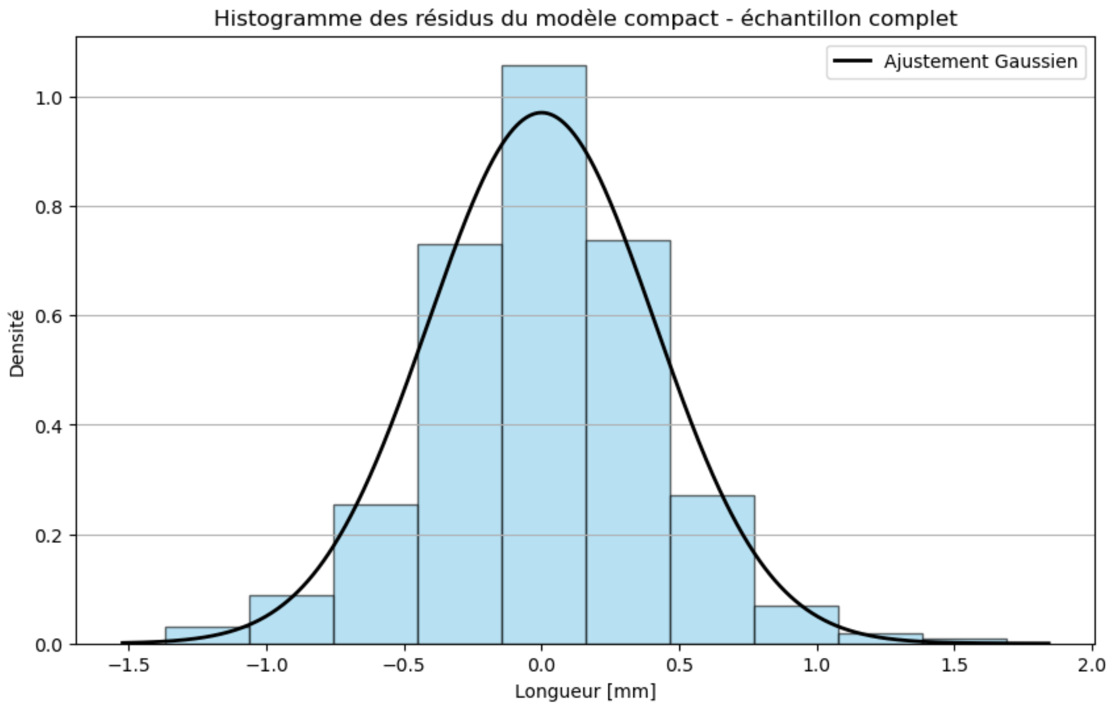
[84]:
```
sb.scatterplot(data=df_ini, x='pred6', y='margin_low', hue='is_genuine')
plt.plot(np.linspace(2,7,21),np.linspace(2,7,21), color='black',␣
↪linestyle='dashed')
plt.title("Analyse : prédiction modèle exhaustif vs. valeur réelle")
plt.xlim([2,7])
plt.ylim([2,7])
plt.grid(visible=True, axis='both', linestyle='--')

plt.xlabel("Prédiction modèle exhaustif [mm]")
plt.ylabel("Marge inférieure réelle [mm]")

#plt.savefig("pred6_vs_real.png")

plt.show()
```



[85]:
```
data = df_ini["res2"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)
```

```python
# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle compact - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.41
Z_min : -3.33
Z_max : 4.11
```

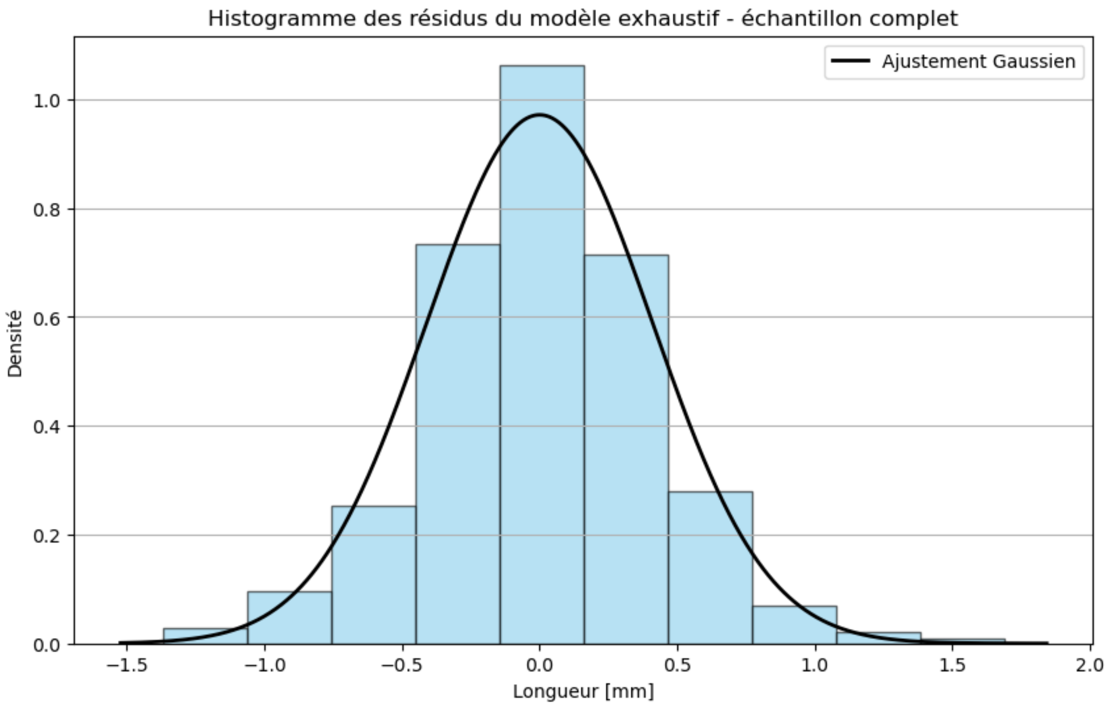



```
Statistique du test de Shapiro-Wilk : 0.9935910574411868
Valeur p : 5.842441926872187e-06
```

[86]:
```
data = df_ini["res6"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle exhaustif - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.41
Z_min : -3.33
Z_max : 4.11
```



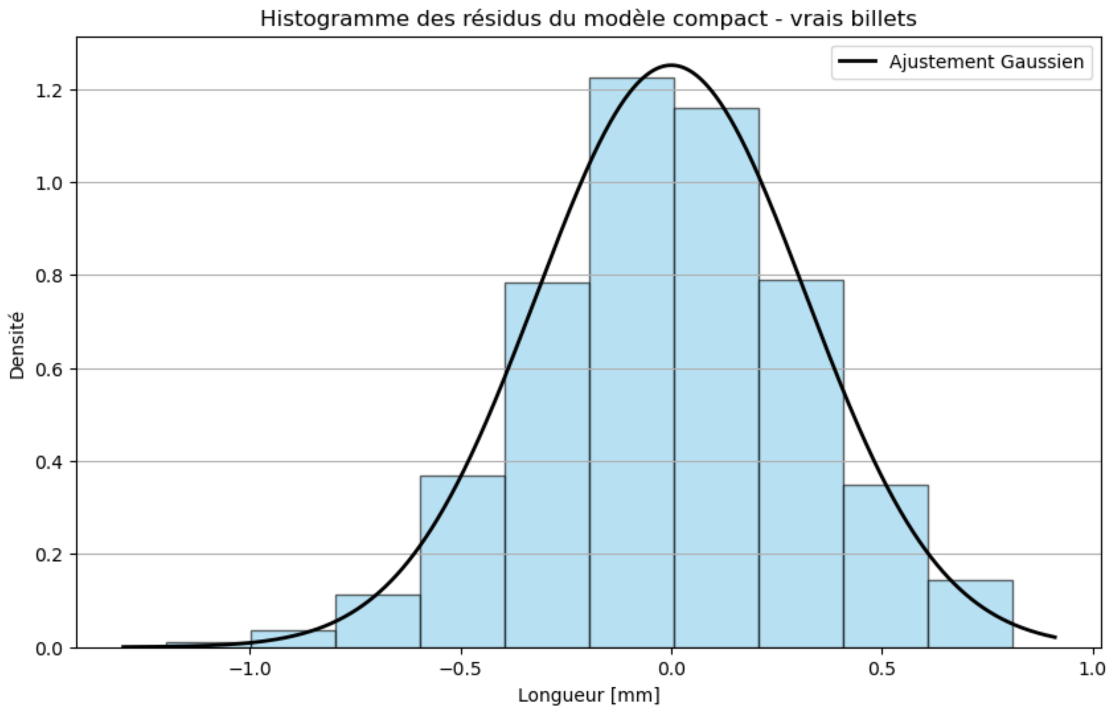

```
Statistique du test de Shapiro-Wilk : 0.993672625376859
Valeur p : 6.770233997782799e-06
```

[87]:

```
data = df_ini1["res2"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle compact - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.32
Z_min : -3.76
Z_max : 2.54
```



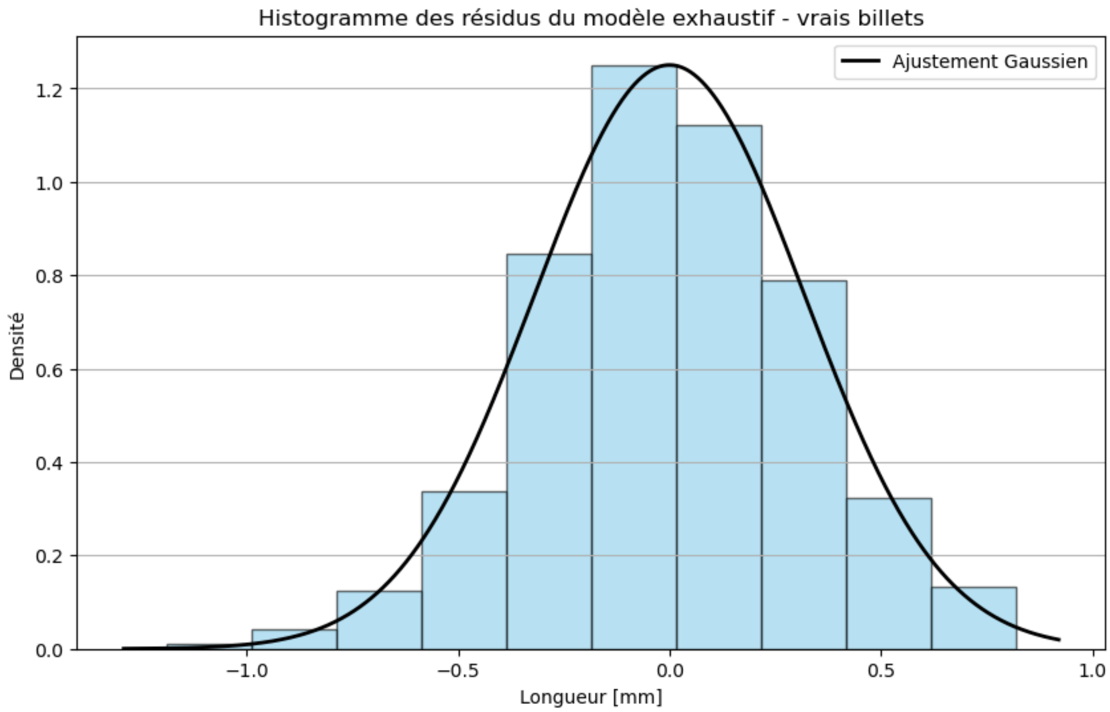

```
Statistique du test de Shapiro-Wilk : 0.9975232872879142
Valeur p : 0.14990535354034434
```

[88]:

```
data = df_ini1["res6"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle exhaustif - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 0.0
Écart type : 0.32
Z_min : -3.73
Z_max : 2.57
```

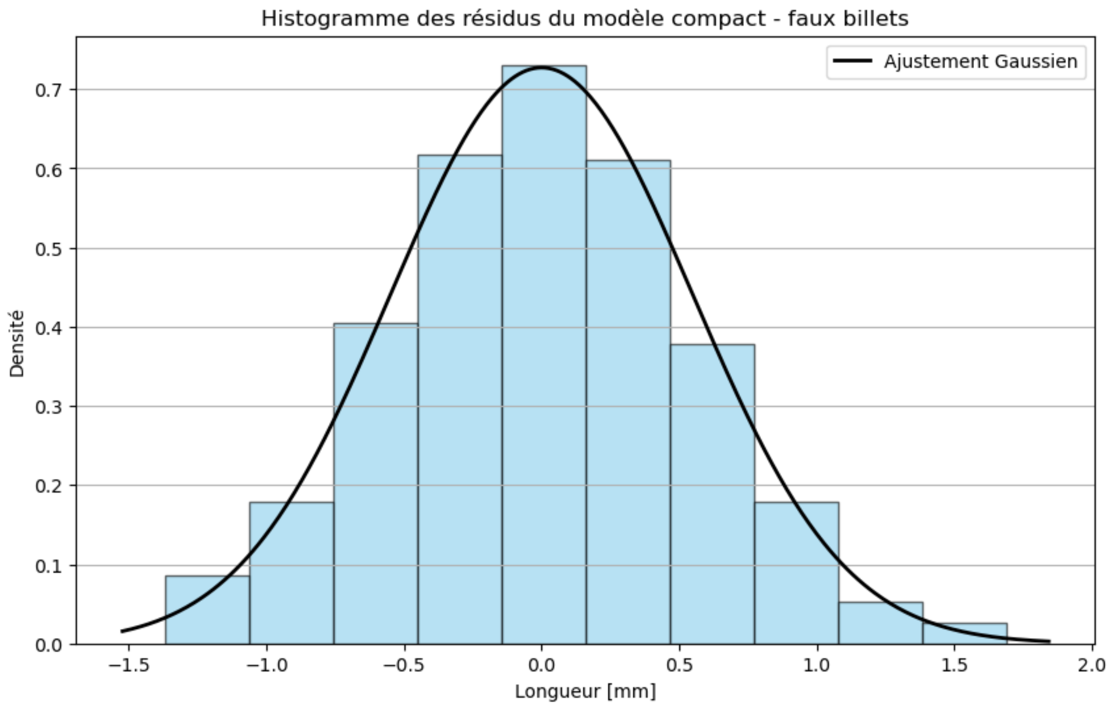



```
Statistique du test de Shapiro-Wilk : 0.9975625577606951
Valeur p : 0.1593078583782332
```

[89]:
```
data = df_ini0["res2"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle compact - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 0.0
Écart type : 0.55
Z_min : -2.49
Z_max : 3.08
```



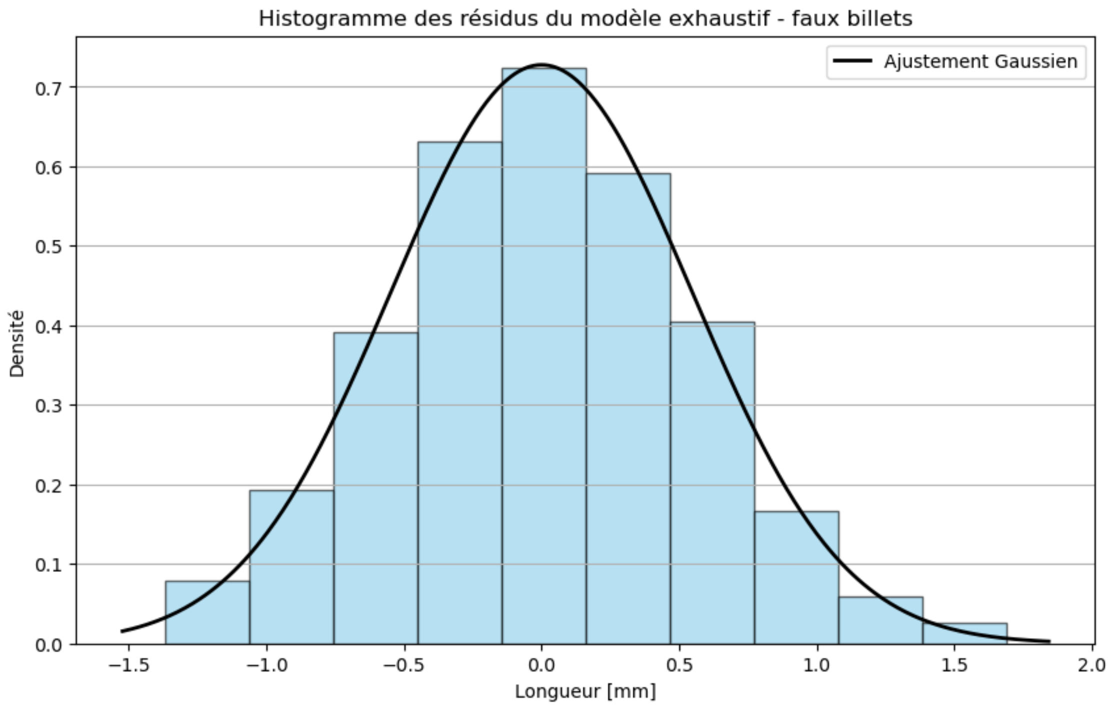

```
Statistique du test de Shapiro-Wilk : 0.9971934979279905
Valeur p : 0.5687185716304073
```

[90]:

```
data = df_ini0["res6"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle exhaustif - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.55
Z_min : -2.5
Z_max : 3.08
```




```
Statistique du test de Shapiro-Wilk : 0.9970256224668824
Valeur p : 0.5133862989397624
```

Il est assez clair pour les 2 modèles de régression linéaire que l'hypothèse de normalité de la distribution des résidus de l'échantillon complet peut être rejetée, alors que l'on peut accepter l'hypothèse de normalité pour les distributions des sous-échantillons vrais billets d'une part et faux billets d'autre part. Il y a également une différence de l'écart-type (racine de la variance) significative selon les échantillons considérés : 0.41 pour l'échantillon complet, 0.32 pour le sous-échantillon des vrais billets, 0.55 pour le sous-échantillon des faux billets.

[91]:
```
# Test d'homoscédasticité de Breusch-Pagan
# Hypothèse nulle H0 : la variance des résidus ne dépend d'aucun des␣
↪prédicteurs (homoscédasticité)
df = df_ini.dropna()[['margin_low','is_genuine','margin_up']]
df['margin_low'] = 1

# Test de Breusch-Pagan
bp_test = ssd.het_breuschpagan(df_ini.dropna()["res2"], df)

# ['LM Statistic', 'LM-Test p-value', 'F-Statistic', 'F-Test p-value']
print(bp_test)
```

```
(163.5570169844264, 3.048210982997846e-36, 91.88292519118576,
2.5966808079875e-38)
```

Les p-valeurs sont inférieures à 0.05, H0 peut être rejetée : les résidus présentent une hétéroscédasticité.

Les valeurs des résidus sont importantes, et le fait de réaliser une régression sur l'échantillon complet donne un poids trop important à la variable *is_genuine*, ce qui semble biaiser les résultats de la régression. On aurait tout intérêt à scinder l'échantillon en deux sous-échantillons, les vrais billets d'une part et les faux billets d'autre part, et à réaliser une régression linéaire sur chacun de ces deux sous-échantillons.

Mesures agrégées de l'erreur

[92]:
```
df_ana = df_ini[["is_genuine","margin_low","pred2","pred6","res2","res6"]].
    ↪dropna().copy()
display(df_ana)
```

```
      is_genuine  margin_low  pred2  pred6  res2  res6
0           True        4.52   4.15   4.21  0.37  0.31
1           True        3.77   4.13   4.12 -0.36 -0.35
2           True        4.40   4.14   4.14  0.26  0.26
3           True        3.62   4.13   4.13 -0.51 -0.51
4           True        4.04   4.03   4.03  0.01  0.01
…         …         …     …     …    …    …
1495       False        4.42   5.27   5.28 -0.85 -0.86
1496       False        5.27   5.21   5.23  0.06  0.04
1497       False        5.51   5.21   5.21  0.30  0.30
1498       False        5.17   5.19   5.19 -0.02 -0.02
1499       False        4.63   5.21   5.21 -0.58 -0.58

[1463 rows x 6 columns]
```

[93]:
```
# Valeur absolue de l'erreur
df_ana["VAE2"] = df_ana["res2"].abs()
df_ana["VAE6"] = df_ana["res6"].abs()

print(f"Somme des erreurs absolues modèle compact : {df_ana["VAE2"].sum()}")
print(f"Somme des erreurs absolues modèle exhaustif : {df_ana["VAE6"].sum()}\n")

print(f"Moyenne des erreurs absolues (MAE) modèle compact : {df_ana["VAE2"].
    ↪mean()}")
print(f"Moyenne des erreurs absolues (MAE) modèle exhaustif : {df_ana["VAE6"].
    ↪mean()}\n")
```

```
Somme des erreurs absolues modèle compact : 462.13
Somme des erreurs absolues modèle exhaustif : 462.05

Moyenne des erreurs absolues (MAE) modèle compact : 0.31587833219412165
Moyenne des erreurs absolues (MAE) modèle exhaustif : 0.31582365003417634
```

[94]:

```
# Carré de l'erreur
df_ana["CE2"] = df_ana["res2"] ** 2
df_ana["CE6"] = df_ana["res6"] ** 2

print(f"Somme des carrés des erreurs modèle compact : {df_ana["CE2"].sum()}")
print(f"Somme des carrés des erreurs modèle exhaustif : {df_ana["CE6"].
 ↪sum()}\n")

print(f"Moyenne des carrés des erreurs (MSE) modèle compact : {df_ana["CE2"].
 ↪mean()}")
print(f"Moyenne des carrés des erreurs (MSE) modèle exhaustif : {df_ana["CE6"].
 ↪mean()}\n")

print(f"Racine de la moyenne des carrés des erreurs (RMSE) modèle compact : {np.
 ↪sqrt(df_ana["CE2"].mean())}")
print(f"Racine de la moyenne des carrés des erreurs (RMSE) modèle exhaustif :␣
 ↪{np.sqrt(df_ana["CE6"].mean())}\n")
```

```
Somme des carrés des erreurs modèle compact : 246.83370000000002
Somme des carrés des erreurs modèle exhaustif : 246.7823

Moyenne des carrés des erreurs (MSE) modèle compact : 0.16871749829118252
Moyenne des carrés des erreurs (MSE) modèle exhaustif : 0.16868236500341763

Racine de la moyenne des carrés des erreurs (RMSE) modèle compact :
0.41075235640368823
Racine de la moyenne des carrés des erreurs (RMSE) modèle exhaustif :
0.41070958718225414
```

[95]:

```
# Erreur relative absolue
df_ana["ERA2"] = (df_ana["res2"]/df_ana["margin_low"]).abs()
df_ana["ERA6"] = (df_ana["res6"]/df_ana["margin_low"]).abs()

print(f"Somme des erreurs relatives absolues modèle compact : {df_ana["ERA2"].
 ↪sum()}")
print(f"Somme des erreurs relatives absolues modèle exhaustif : {df_ana["ERA6"].
 ↪sum()}\n")

print(f"Moyenne des erreurs relatives absolues (MAPE) modèle compact :␣
 ↪{df_ana["ERA2"].mean()}")
print(f"Moyenne des erreurs relatives absolues (MAPE) modèle exhaustif :␣
 ↪{df_ana["ERA6"].mean()}\n")
```

```
Somme des erreurs relatives absolues modèle compact : 102.82285827294459
Somme des erreurs relatives absolues modèle exhaustif : 102.80073643880748

Moyenne des erreurs relatives absolues (MAPE) modèle compact :
```

```
0.0702821997764488
Moyenne des erreurs relatives absolues (MAPE) modèle exhaustif :
0.07026707890554168
```

[96]:

```python
# Carré de l'erreur relative
df_ana["CER2"] = (df_ana["res2"]/df_ana["margin_low"]) ** 2
df_ana["CER6"] = (df_ana["res6"]/df_ana["margin_low"]) ** 2

print(f"Somme des carrés de l'erreur relative modèle compact : {df_ana["CER2"].
    ↪sum()}")
print(f"Somme des carrés de l'erreur relative modèle exhaustif :␣
    ↪{df_ana["CER6"].sum()}\n")

print(f"Moyenne des carrés de l'erreur relative modèle compact :␣
    ↪{df_ana["CER2"].mean()}")
print(f"Moyenne des carrés de l'erreur relative modèle exhaustif :␣
    ↪{df_ana["CER6"].mean()}\n")

print(f"Racine de la moyenne des carrés de l'erreur relative modèle compact :␣
    ↪{np.sqrt(df_ana["CER2"].mean())}")
print(f"Racine de la moyenne des carrés de l'erreur relative modèle exhaustif :␣
    ↪{np.sqrt(df_ana["CER6"].mean())}\n")
```

```
Somme des carrés de l'erreur relative modèle compact : 12.122603227273004
Somme des carrés de l'erreur relative modèle exhaustif : 12.119669269125389

Moyenne des carrés de l'erreur relative modèle compact : 0.008286126607842108
Moyenne des carrés de l'erreur relative modèle exhaustif : 0.008284121168233348

Racine de la moyenne des carrés de l'erreur relative modèle compact :
0.09102816381671174
Racine de la moyenne des carrés de l'erreur relative modèle exhaustif :
0.09101714766039061
```

On commet en moyenne une erreur relative de 7% sur la prédiction de la marge inférieure avec notre modèle de régression linéaire basée sur l'échantillon complet.

3.2 - Échantillon scindé

3.2.1 - Vrais billets

On a a priori aucune idée des variables dont pourrait dépendre *margin_low*.

[97]:

```python
df = df_ini1.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles non-vides de L_var
```

```
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
↪len(L_var)+1))))

score_max = 0
best_set = set()

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]
    # la variable cible, la marge inférieure
    y = df["margin_low"]
    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps

    # Coefficients de la régression linéaire (même ordre que celui des␣
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)
```

```
Meilleur score R² obtenu : 0.007935320186607586
pour le jeu de variables ('diagonal', 'height_left', 'height_right',
'margin_up', 'length')
```

[98]:

```
# Meilleur modèle à 1 prédicteur
df = df_ini1.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
↪2))))

score_max = 0
best_set = set()
```

```
best_coef11 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps
        best_coef11 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef11)
# Valeur à l'origine
a1 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set)].mean(),␣
 ↪best_coef11)]))
print(f"Valeur à l'origine : {a1}")
```

```
Meilleur score R² obtenu : 0.0036584918999577054
pour le jeu de variables ('margin_up',)
Coefficients de la régression : [-0.10409817]
Valeur à l'origine : 4.433861037537617
```

[99]:
```
# Meilleur modèle à 2 prédicteurs
df = df_ini1.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']
```

```python
# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(2,
↪3))))

score_max2 = 0
best_set2 = set()
best_coef2 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max2 :
        score_max2 = reg.score(X, y)
        best_set2 = ps
        best_coef2 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max2}")
print("pour le jeu de variables", best_set2)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef2)

# Valeur à l'origine
b0 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set2)].mean(),
↪best_coef2)]))
print(f"Valeur à l'origine : {b0}")
```

```
Meilleur score R² obtenu : 0.005276342123526678
```

```
pour le jeu de variables ('margin_up', 'length')
Coefficients de la régression : [-0.10580653  0.03605465]
Valeur à l'origine : 0.35757959058453004
```

[100]:
```python
# Meilleur modèle à 3 prédicteurs
df = df_ini1.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 3 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(3,␣
 ↪4))))

score_max = 0
best_set3 = set()
best_coef3 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set3 = ps
        best_coef3 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set3)

# Coefficients de la régression linéaire (même ordre que celui des variables)
```

```
print("Coefficients de la régression :",best_coef3)
print(f"Valeur à l'origine : {(y.mean()-sum([i*j for (i, j) in␣
 ↪zip(df[list(best_set3)].mean(), best_coef3)]))}")
```

```
Meilleur score R² obtenu : 0.006847278642744377
pour le jeu de variables ('diagonal', 'margin_up', 'length')
Coefficients de la régression : [ 0.04198048 -0.10749164  0.03571841]
Valeur à l'origine : -6.8193717624693955
```

[101]:
```
# Comparaison modèles {valeur constante} vs. {margin_up}
# Hypothèse H0 : le coefficient associé au prédicteur margin_up égale 0
X1 = sm.add_constant([df["margin_low"].mean()]*len(df))   # Modèle 1
X2 = sm.add_constant(df[["margin_up"]])                   # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(3.558095916147422, 0.05955450258873047, 1.0)
```

On a une p-valeur supérieure à 0.05, ce qui nous invite à ne pas rejeter H0 : utiliser *margin_up* comme prédicteur n'a pas d'impact statistique significatif.

[102]:
```
# Comparaison modèles {valeur constante} vs. {tous les prédicteurs possibles}
# Hypothèse H2 : les coefficients associés à chacun des prédicteurs égalent 0
X1 = sm.add_constant(df[["margin_up"]])              # Modèle 1
X2 = sm.add_constant(df[['margin_up','length']])    # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(1.5743860156677267, 0.20987405845858115, 1.0)
```

[103]:
```
# Comparaison modèles {valeur constante} vs. {tous les prédicteurs possibles}
# Hypothèse H2 : les coefficients associés à chacun des prédicteurs égalent 0
```

```
X1 = sm.add_constant([df["margin_low"].mean()]*len(df))
 ↪           # Modèle 1
X2 = sm.
 ↪add_constant(df[['diagonal','height_left','height_right','margin_up','length']])
 ↪  # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(1.5437670821053122, 0.17355882021936708, 5.0)
```

On a une p-valeur supérieure à 0.05, ce qui nous invite à ne pas rejeter H2 : aucun des prédicteurs possibles n'a d'impact statistique significatif.

[104]:
```
# Comparaison modèles {valeur constante} vs. {tous les prédicteurs possibles}
# Hypothèse H2 : les coefficients associés à chacun des autres prédicteurs
 ↪égalent 0
X1 = sm.add_constant(df[["margin_up"]])
 ↪           # Modèle 1
X2 = sm.
 ↪add_constant(df[['diagonal','height_left','height_right','margin_up','length']])
 ↪  # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(1.0400378575602376, 0.38536904900058544, 4.0)
```

3.2.2 - Faux billets

[105]:
```
df = df_ini0.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles non-vides de L_var
```

```
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
↪len(L_var)+1))))

score_max = 0
best_set = set()

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]
    # la variable cible, la marge inférieure
    y = df["margin_low"]
    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps

    # Coefficients de la régression linéaire (même ordre que celui des␣
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)
```

```
Meilleur score R² obtenu : 0.02720424077602579
pour le jeu de variables ('diagonal', 'height_left', 'height_right',
'margin_up', 'length')
```

[106]:

```
# Meilleur modèle à 1 prédicteur
df = df_ini0.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(1,␣
↪2))))

score_max = 0
best_set = set()
```

```
best_coef10 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set = ps
        best_coef10 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
 ↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef)

#Valeur à l'origine
a0 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set)].mean(),␣
 ↪best_coef10)]))
print(f"Valeur à l'origine : {a0}")
```

```
Meilleur score R² obtenu : 0.020293853326611044
pour le jeu de variables ('margin_up',)
Coefficients de la régression : [-1.09983815]
Valeur à l'origine : 6.689535130139839
```

[107]:
```
# Meilleur modèle à 2 prédicteurs
df = df_ini0.dropna()
```

```python
L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 2 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(2,␣
↪3))))

score_max2 = 0
best_set2 = set()
best_coef2 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max2 :
        score_max2 = reg.score(X, y)
        best_set2 = ps
        best_coef2 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max2}")
print("pour le jeu de variables", best_set2)

# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef2)

# Valeur à l'origine
b0 = (y.mean()-sum([i*j for (i, j) in zip(df[list(best_set2)].mean(),␣
↪best_coef2)]))
print(f"Valeur à l'origine : {b0}")
```

```
Meilleur score R² obtenu : 0.02382243687639851
pour le jeu de variables ('diagonal', 'margin_up')
Coefficients de la régression : [-0.10771867 -0.43262252]
Valeur à l'origine : 25.182861864282735
```

[108]:

```python
# Meilleur modèle à 3 prédicteurs
df = df_ini0.dropna()

L_var = ['diagonal','height_left','height_right','margin_up','length']

# On construit la liste des sous-ensembles à 3 éléments de L_var
PL_var = list(set(chain.from_iterable(combinations(L_var, r) for r in range(3,␣
↪4))))

score_max = 0
best_set3 = set()
best_coef3 = []

# la variable cible, la marge inférieure
y = df["margin_low"]

for ps in PL_var :
    pl=list(ps)
    #print("Variables :", pl)
    # les variables prédictives
    X = df[pl]

    # on choisit un modèle de régression linéaire
    reg = LinearRegression()

    # on entraîne ce modèle sur les données avec la méthode fit
    reg.fit(X, y)
    # score R2 de la régression
    #print(f"Score R² : {reg.score(X, y)}")
    if reg.score(X, y) > score_max :
        score_max = reg.score(X, y)
        best_set3 = ps
        best_coef3 = reg.coef_

    # Coefficients de la régression linéaire (même ordre que celui des␣
↪variables)
    #print("Coefficients de la régression :",reg.coef_)
    #print("\n")

print(f"Meilleur score R² obtenu : {score_max}")
print("pour le jeu de variables", best_set3)
```

```
# Coefficients de la régression linéaire (même ordre que celui des variables)
print("Coefficients de la régression :",best_coef3)
print(f"Valeur à l'origine : {(y.mean()-sum([i*j for (i, j) in␣
 ↪zip(df[list(best_set3)].mean(), best_coef3)]))}")
```

```
Meilleur score R² obtenu : 0.02547029439491999
pour le jeu de variables ('diagonal', 'height_left', 'margin_up')
Coefficients de la régression : [-0.11339518  0.10043162 -0.43055363]
Valeur à l'origine : 15.687903194179817
```

[109]:
```
# Comparaison modèles {valeur constante} vs. {margin_up}
# Hypothèse H0 : le coefficient associé au prédicteur margin_up égale 0
X1 = sm.add_constant([df["margin_low"].mean()]*len(df))    # Modèle 1
X2 = sm.add_constant(df[["margin_up"]])                    # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(10.14997013523339, 0.0015350629258352997, 1.0)
```

On a une p-valeur inférieure à 0.05, ce qui nous invite à rejeter H0 : utiliser *margin_up* comme prédicteur a un impact statistique significatif (H1).

[110]:
```
# Comparaison modèles {margin_up} vs. {margin_up, diagonal}
# Hypothèse H2 : le coefficient associé au prédicteur diagonal égale 0
X1 = sm.add_constant(df[["margin_up"]])                 # Modèle 1
X2 = sm.add_constant(df[["margin_up","diagonal"]])      # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(1.767585551059742, 0.18430104742180334, 1.0)
```

On a une p-valeur supérieure à 0.05, ce qui nous invite à ne pas rejeter H2 : utiliser *diagonal* comme prédicteur n'a pas d'impact statistique significatif (H3).

[111]:
```
# Comparaison modèles {margin_up} vs. {tous les prédicteurs possibles}
# Hypothèse H2 : les coefficients associés à chaucn des prédicteurs autres que␣
↪margin_up égalent 0
X1 = sm.add_constant(df[["margin_up"]])                                        ␣
↪              # Modèle 1
X2 = sm.
↪add_constant(df[['diagonal','height_left','height_right','margin_up','length']])␣
↪    # Modèle 2
y = df["margin_low"]

# Estimation des modèles
model1 = sm.OLS(y, X1).fit()
model2 = sm.OLS(y, X2).fit()

# Test F de comparaison (likelihood ratio test)
f_test = model2.compare_f_test(model1)

print(f_test)  # (F, p-value, diff_degrés_de_liberté)
```

```
(0.863091833144553, 0.48594736826575846, 4.0)
```

3.2.3 - Recoupement des modèles

[112]:
```
display(df_ini)
```

```
      is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
0           True    171.81       104.86        104.95        4.52       2.89
1           True    171.46       103.36        103.66        3.77       2.99
2           True    172.69       104.48        103.50        4.40       2.94
3           True    171.36       103.91        103.94        3.62       3.01
4           True    171.73       104.28        103.46        4.04       3.48
…            …        …           …             …           …          …
1495       False    171.75       104.38        104.17        4.42       3.09
1496       False    172.19       104.63        104.44        5.27       3.37
1497       False    171.80       104.01        104.12        5.51       3.36
1498       False    172.06       104.28        104.06        5.17       3.46
1499       False    171.47       104.15        103.82        4.63       3.37

      length  pred2  pred6  res2  res6
0     112.83   4.15   4.21  0.37  0.31
1     113.09   4.13   4.12 -0.36 -0.35
2     113.16   4.14   4.14  0.26  0.26
3     113.51   4.13   4.13 -0.51 -0.51
4     112.54   4.03   4.03  0.01  0.01
…       …     …      …     …     …
1495  111.28   5.27   5.28 -0.85 -0.86
1496  110.97   5.21   5.23  0.06  0.04
1497  111.95   5.21   5.21  0.30  0.30
1498  112.25   5.19   5.19 -0.02 -0.02
```

```
1499  112.07   5.21   5.21 -0.58 -0.58

[1500 rows x 11 columns]
```

[113]:
```
# Initialisation de la colonne
df_ini["preds1"] = df_ini["margin_low"]
```

[114]:
```
# Calcul des prédictions
df_ini.loc[df_ini["is_genuine"]==True,"preds1"] =␣
 ↪round(best_coef11[0]*df_ini["margin_up"] + a1, 2)
df_ini.loc[df_ini["is_genuine"]==False,"preds1"] =␣
 ↪round(best_coef10[0]*df_ini["margin_up"] + a0, 2)
# Calcul des résidus
df_ini["ress1"] = df_ini["margin_low"] - df_ini["preds1"]
```

[115]:
```
display(df_ini)
```

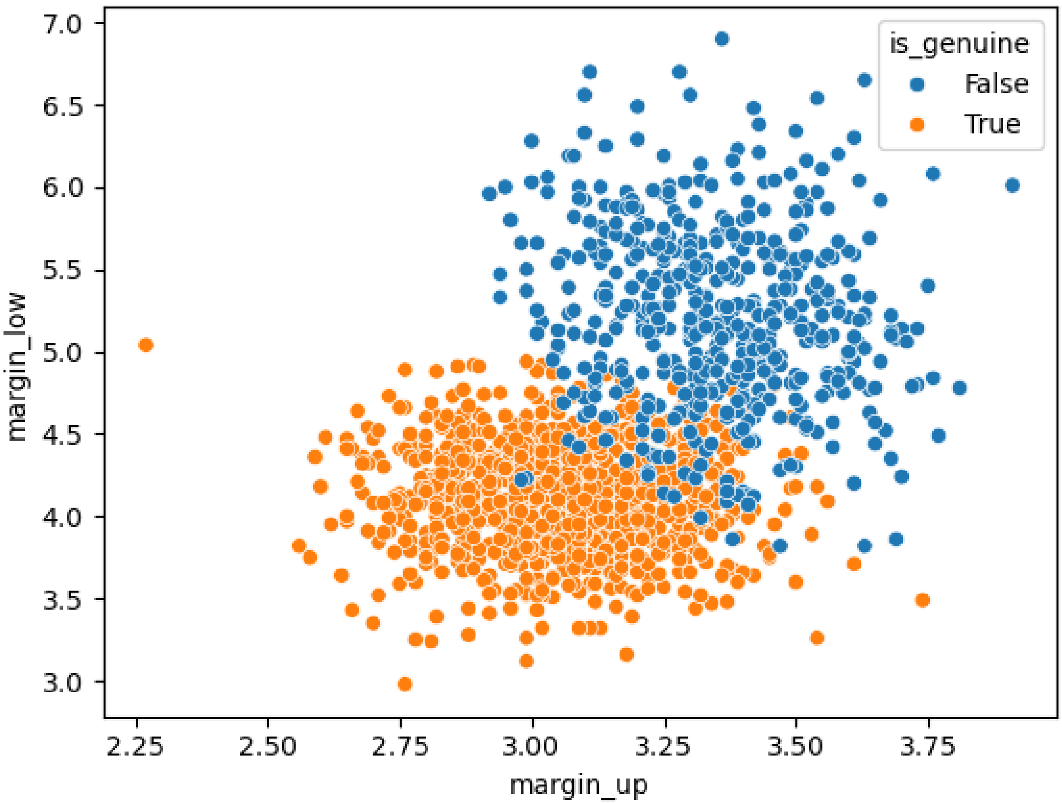
```
      is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
0           True    171.81       104.86        104.95        4.52       2.89
1           True    171.46       103.36        103.66        3.77       2.99
2           True    172.69       104.48        103.50        4.40       2.94
3           True    171.36       103.91        103.94        3.62       3.01
4           True    171.73       104.28        103.46        4.04       3.48
…              …         …           …             …           …          …
1495       False    171.75       104.38        104.17        4.42       3.09
1496       False    172.19       104.63        104.44        5.27       3.37
1497       False    171.80       104.01        104.12        5.51       3.36
1498       False    172.06       104.28        104.06        5.17       3.46
1499       False    171.47       104.15        103.82        4.63       3.37

      length  pred2  pred6  res2  res6  preds1  ress1
0     112.83   4.15   4.21  0.37  0.31    4.13   0.39
1     113.09   4.13   4.12 -0.36 -0.35    4.12  -0.35
2     113.16   4.14   4.14  0.26  0.26    4.13   0.27
3     113.51   4.13   4.13 -0.51 -0.51    4.12  -0.50
4     112.54   4.03   4.03  0.01  0.01    4.07  -0.03
…        …      …      …     …     …       …      …
1495  111.28   5.27   5.28 -0.85 -0.86    5.33  -0.91
1496  110.97   5.21   5.23  0.06  0.04    5.21   0.06
1497  111.95   5.21   5.21  0.30  0.30    5.21   0.30
1498  112.25   5.19   5.19 -0.02 -0.02    5.17   0.00
1499  112.07   5.21   5.21 -0.58 -0.58    5.21  -0.58

[1500 rows x 13 columns]
```

[116]:
```
df_ini1 = df_ini.dropna().loc[df_ini["is_genuine"]==True]
df_ini0 = df_ini.dropna().loc[df_ini["is_genuine"]==False]
```

Linéarité de la variable prédite vis-à-vis des prédicteurs

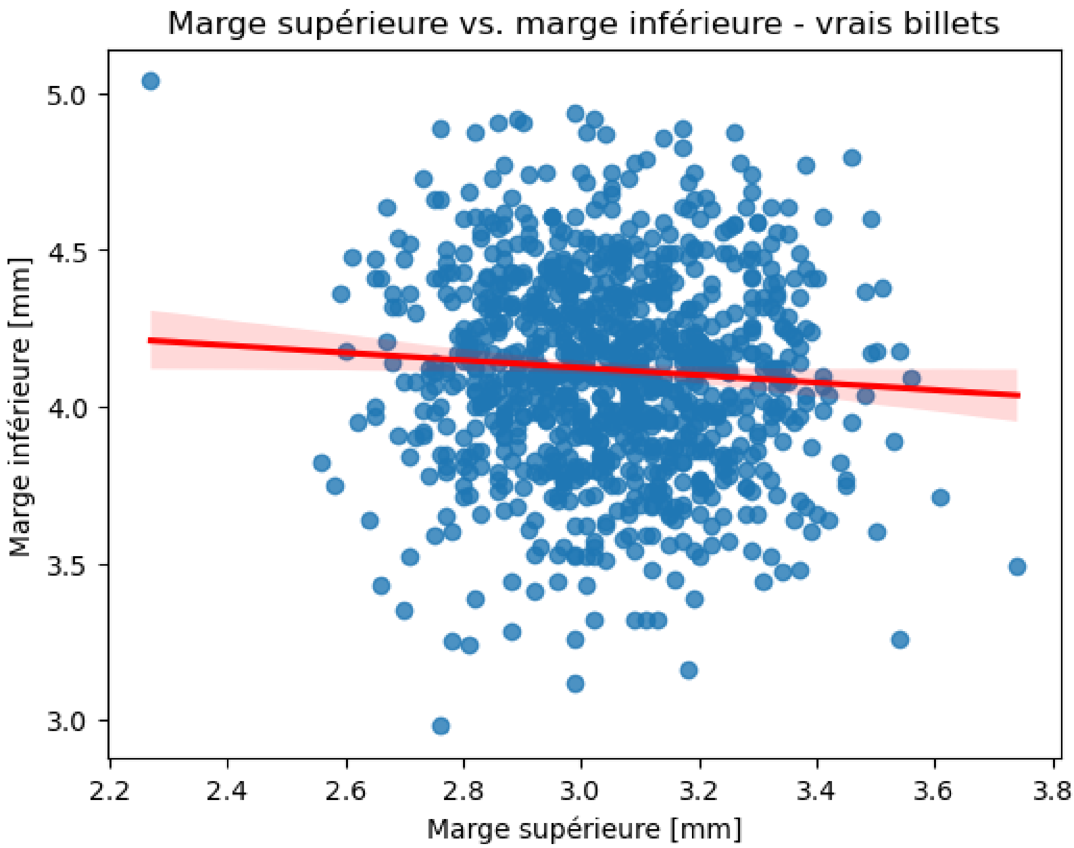
[117]:
```python
# Vis-à-vis de margin_up
sb.scatterplot(data=df_ini.dropna(), x="margin_up", y="margin_low",␣
↪hue="is_genuine")
```

[117]: `<Axes: xlabel='margin_up', ylabel='margin_low'>`



[118]:
```python
# Linéarité vis-à-vis de margin_up - vrais billets

# Visualisation de la tendance
sb.regplot(data=df_ini1, x="margin_up", y="margin_low", robust=True,␣
↪line_kws={'color': "r"})
plt.xlabel("Marge supérieure [mm]")
plt.ylabel("Marge inférieure [mm]")
plt.title("Marge supérieure vs. marge inférieure - vrais billets")
#plt.savefig(".png")
plt.show()

# Hypothèse nulle H0 : les variables ne sont pas linéairement corrélées
# Calculer le coefficient de corrélation de Pearson et la valeur p
pearson_corr, pearson_p_value = st.pearsonr(x=df_ini1["margin_up"],␣
↪y=df_ini1["margin_low"])
```

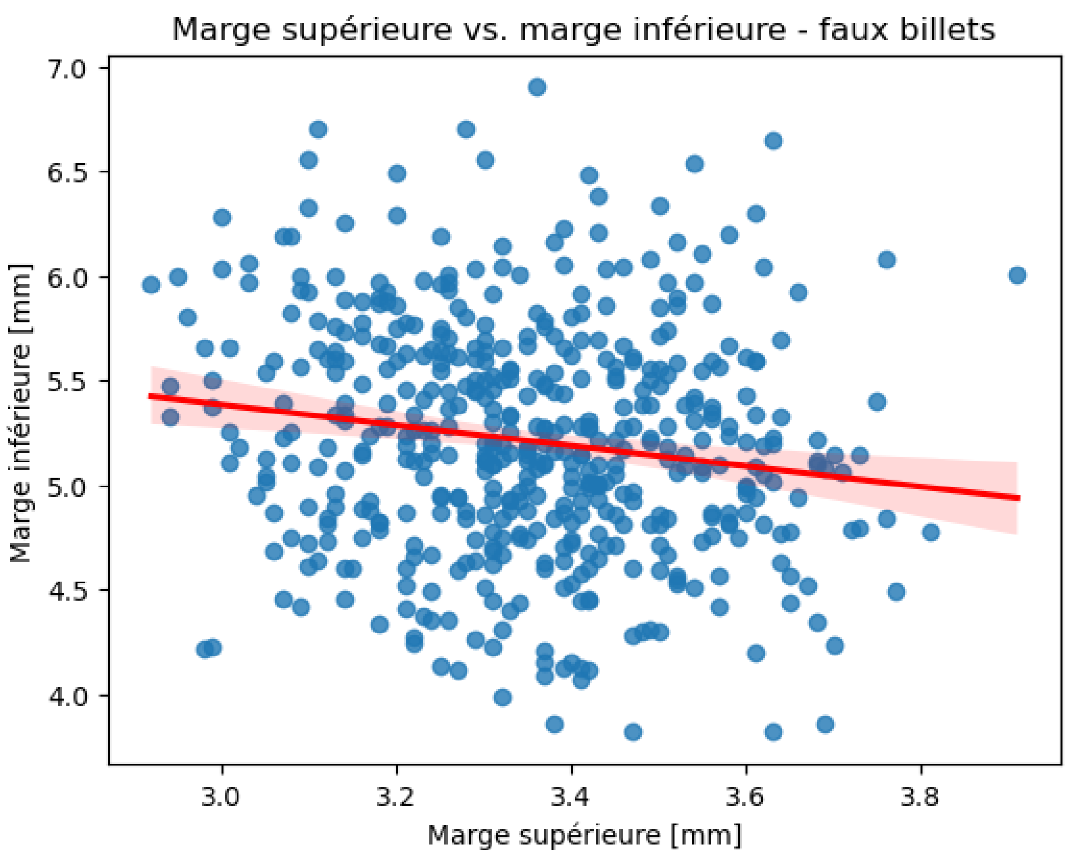
```python
print(f"Coefficient de corrélation de Pearson : {pearson_corr}")
print(f"Valeur p : {pearson_p_value }")

# Interprétation des résultats
alpha = 0.05
if pearson_p_value > alpha:
    print("\nLes variables ne sont pas corrélées (on ne rejette pas H0)")
else:
    print("\nLes variables sont corrélées (on rejette H0)")
```




```
Coefficient de corrélation de Pearson : -0.060485468502424033
Valeur p : 0.05955450258870119

Les variables ne sont pas corrélées (on ne rejette pas H0)
```

[119]:
```python
# Linéarité vis-à-vis de margin_up - faux billets

# Visualisation de la tendance
```

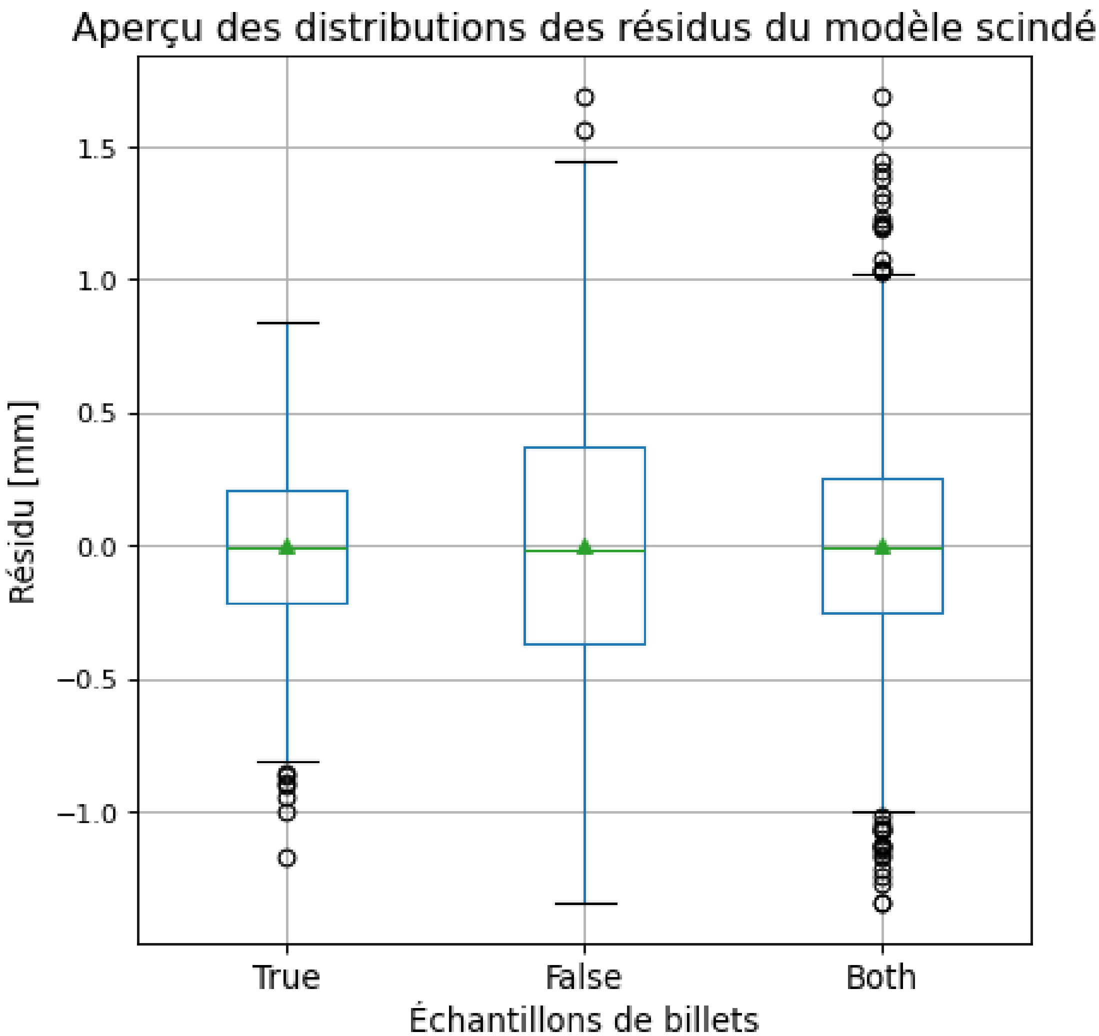
```python
sb.regplot(data=df_ini0, x="margin_up", y="margin_low", robust=True,␣
 ↪line_kws={'color': "r"})
plt.xlabel("Marge supérieure [mm]")
plt.ylabel("Marge inférieure [mm]")
plt.title("Marge supérieure vs. marge inférieure - faux billets")
#plt.savefig(".png")
plt.show()

# Hypothèse nulle H0 : les variables ne sont pas linéairement corrélées
# Calculer le coefficient de corrélation de Pearson et la valeur p
pearson_corr, pearson_p_value = st.pearsonr(x=df_ini0["margin_up"],␣
 ↪y=df_ini0["margin_low"])

print(f"Coefficient de corrélation de Pearson : {pearson_corr}")
print(f"Valeur p : {pearson_p_value }")

# Interprétation des résultats
alpha = 0.05
if pearson_p_value > alpha:
    print("\nLes variables ne sont pas corrélées (on ne rejette pas H0)")
else:
    print("\nLes variables sont corrélées (on rejette H0)")
```



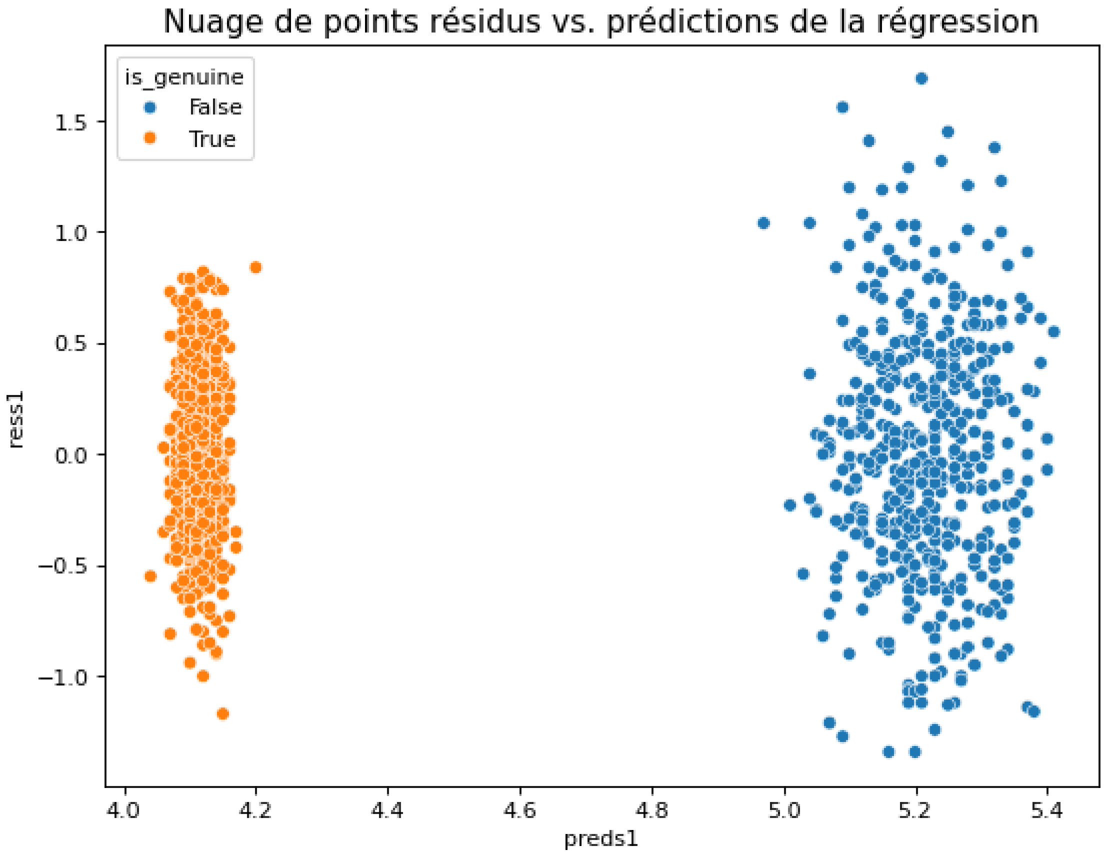
```
Coefficient de corrélation de Pearson : -0.14245649625977433
Valeur p : 0.0015350629258354448

Les variables sont corrélées (on rejette H0)
```

Normalité des résidus

[120]:
```python
fig = plt.figure(figsize=(6,6), dpi=80)

df_ini1.boxplot(column=["ress1"],\
                positions=[1], widths=[0.4],
                showmeans=True)

df_ini0.boxplot(column=["ress1"],\
                positions=[2], widths=[0.4],
                showmeans=True)

df_ini.boxplot(column=["ress1"],\
               positions=[3], widths=[0.4],
               showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Résidu [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions des résidus du modèle scindé", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```

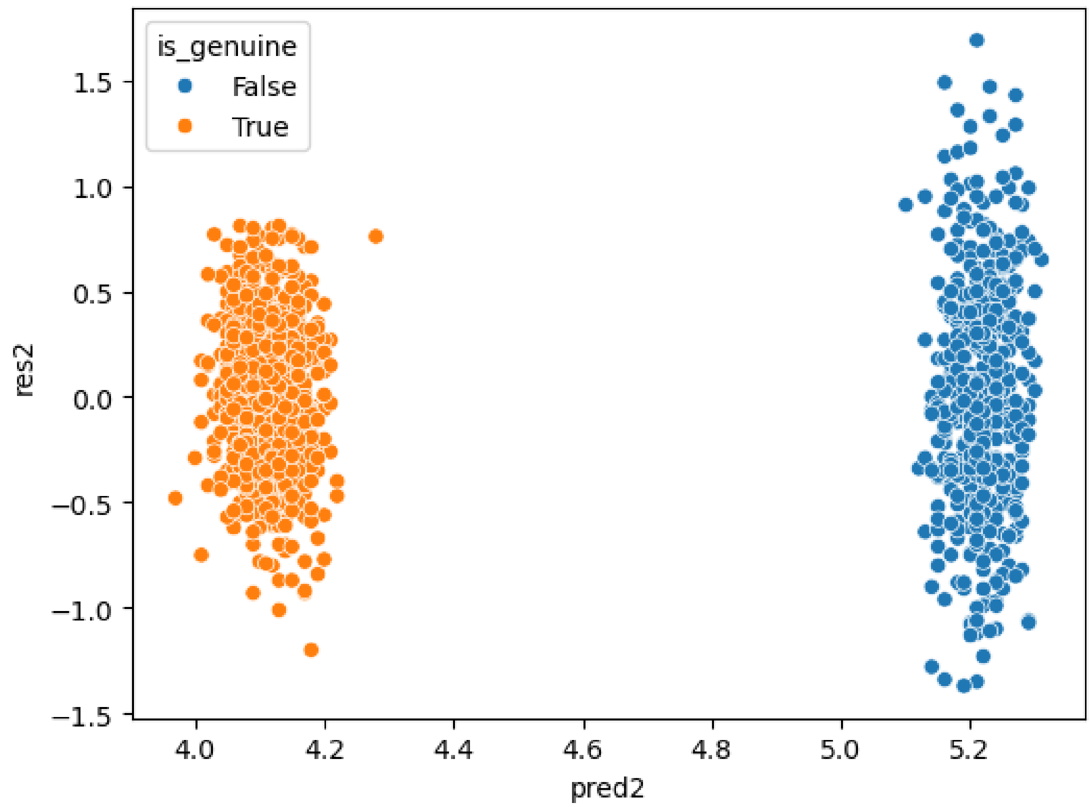


[121]:
```
fig = plt.figure(figsize=(8,6), dpi=80)
sb.scatterplot(data=df_ini, x='preds1', y='ress1', hue='is_genuine')
plt.title("Nuage de points résidus vs. prédictions de la régression",␣
  ↪fontsize=14)
plt.savefig("res_pred_S1.png")
plt.show()
```



[122]:
```
# Pour rappel
sb.scatterplot(data=df_ini, x='pred2', y='res2', hue='is_genuine')
```

[122]: <Axes: xlabel='pred2', ylabel='res2'>

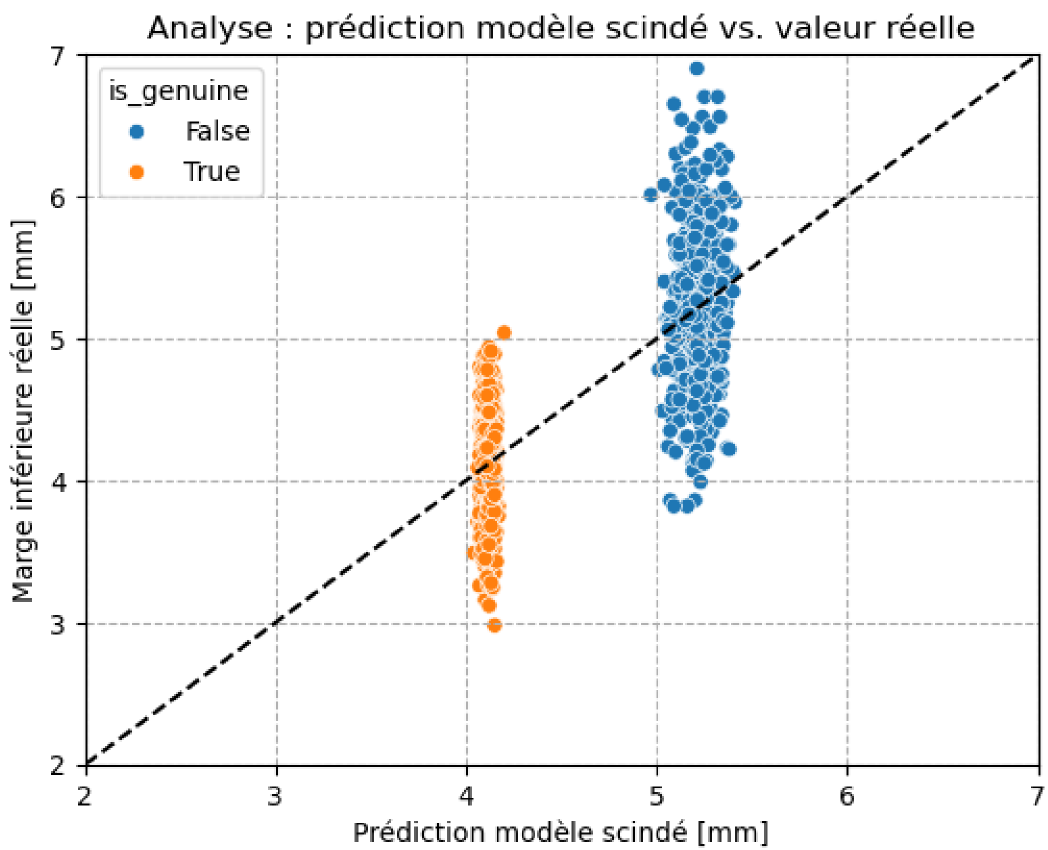


En considérant l'ensemble de l'échantillon comme données d'entraînement de la régression, on a eu tendance à avoir une variabilité des prédictions à peu près comparables entre les sous-échantillons des vrais billets et des faux billets, alors qu'en considérant les sous-échantillons séparément (chacun d'exu ayant sa propre régression, mais avec les mêmes prédicteurs de régression), on constate qu'on a une plus grande variabilité des prédictions parmi les faux billets, à peu près 3 fois plus grande que pour les vrais billets.

[123]:
```
print(f"Amplitude prédictions pred2 vrais billets : {round(df_ini1["pred2"].
 ↪max() - df_ini1["pred2"].min(),2)}")
print(f"Amplitude prédictions preds1 vrais billets : {round(df_ini1["preds1"].
 ↪max() - df_ini1["preds1"].min(),2)}")

print(f"Amplitude prédictions pred2 faux billets : {round(df_ini0["pred2"].
 ↪max() - df_ini0["pred2"].min(),2)}")
print(f"Amplitude prédictions preds1 faux billets : {round(df_ini0["preds1"].
 ↪max() - df_ini0["preds1"].min(),2)}")
```

```
Amplitude prédictions pred2 vrais billets : 0.31
Amplitude prédictions preds1 vrais billets : 0.16
Amplitude prédictions pred2 faux billets : 0.21
Amplitude prédictions preds1 faux billets : 0.44
```

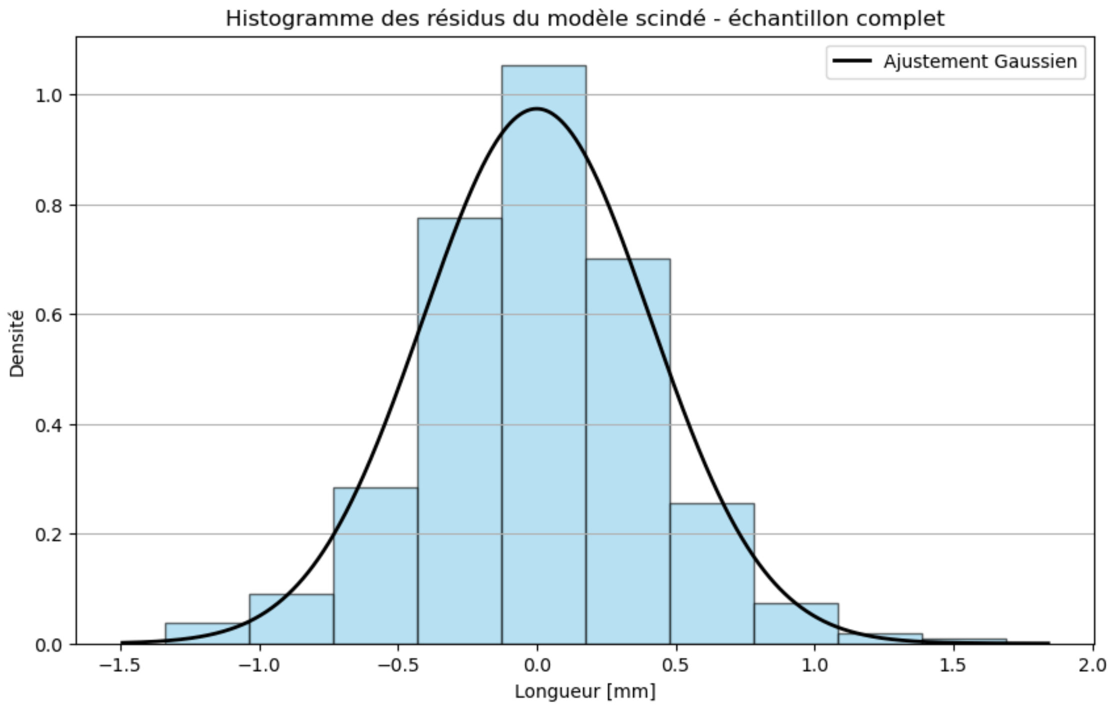
[124]:
```
sb.scatterplot(data=df_ini, x='preds1', y='margin_low', hue='is_genuine')
plt.plot(np.linspace(2,7,21),np.linspace(2,7,21), color='black',␣
 ↪linestyle='dashed')
plt.title("Analyse : prédiction modèle scindé vs. valeur réelle")
plt.xlim([2,7])
plt.ylim([2,7])
plt.grid(visible=True, axis='both', linestyle='--')

plt.xlabel("Prédiction modèle scindé [mm]")
plt.ylabel("Marge inférieure réelle [mm]")

#plt.savefig("pred2_vs_real.png")

plt.show()
```



[125]:
```
data = df_ini["ress1"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
```

```
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle scindé - échantillon complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.41
Z_min : -3.27
Z_max : 4.12
```

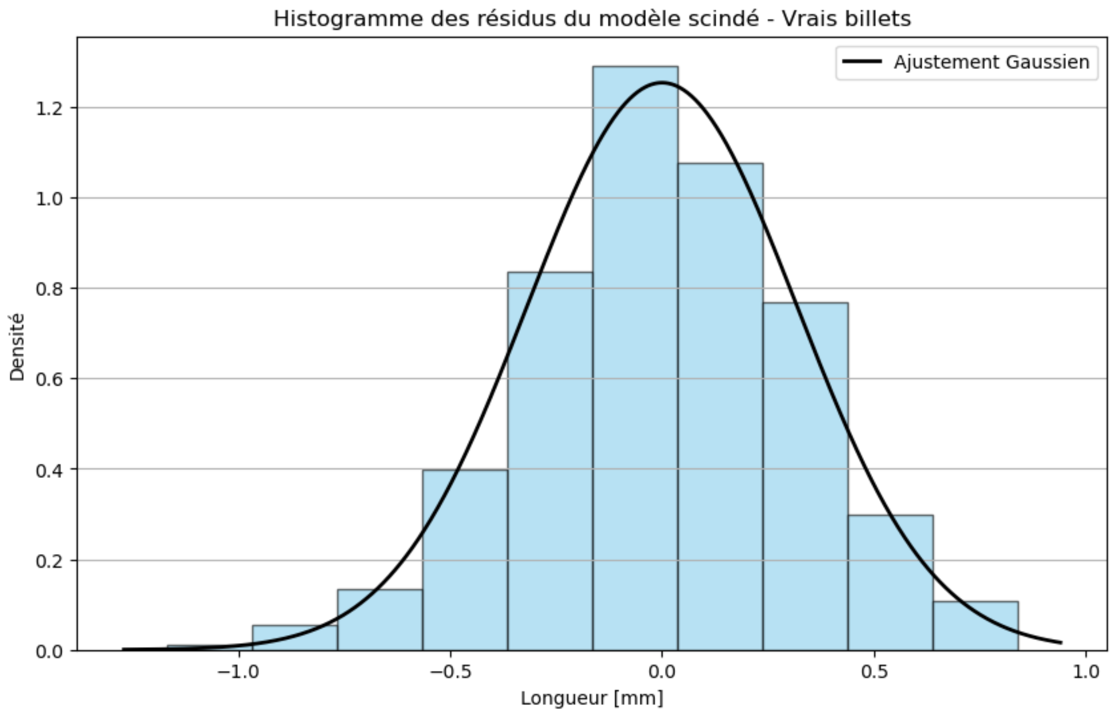



```
Statistique du test de Shapiro-Wilk : 0.9931255288850037
Valeur p : 2.5616590557522963e-06
```

[126]:
```
data = df_ini1["ress1"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
```

```python
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle scindé - Vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.32
Z_min : -3.67
Z_max : 2.64
```



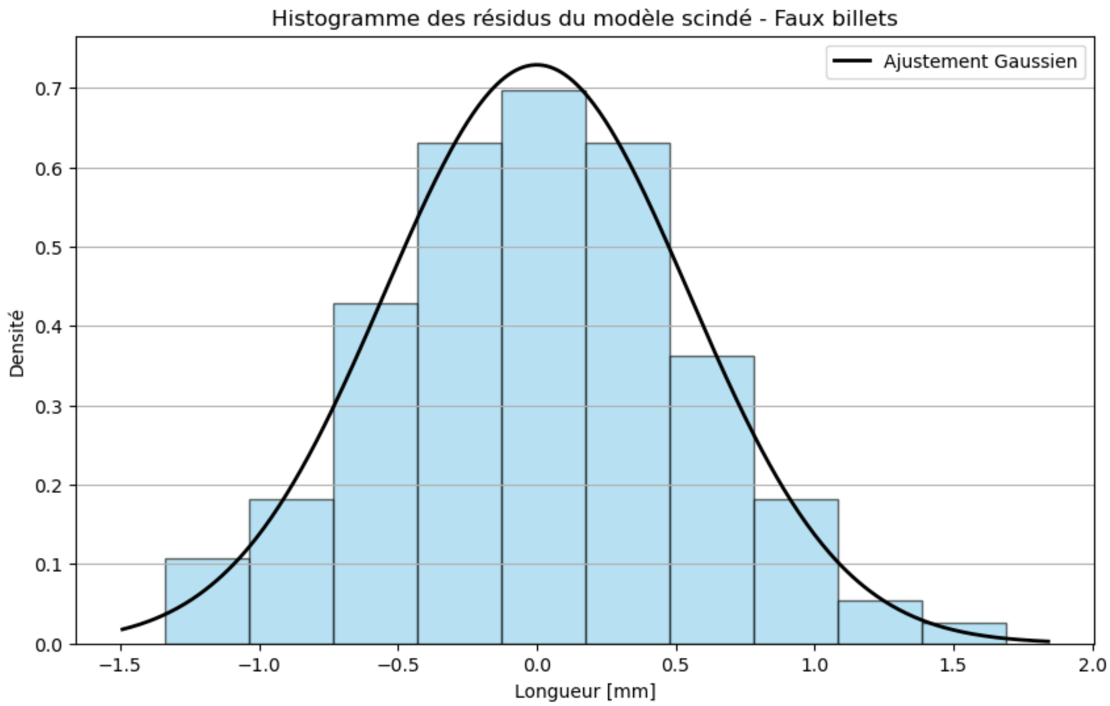

```
Statistique du test de Shapiro-Wilk : 0.9979584277647956
Valeur p : 0.2881008592236531
```

Pris séparément, la distribution des résidus des vrais billets suit une loi normale.

[127]:
```
data = df_ini0["ress1"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)
```

```python
# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des résidus du modèle scindé - Faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : -0.0
Écart type : 0.55
Z_min : -2.45
Z_max : 3.09
```




```
Statistique du test de Shapiro-Wilk : 0.9971026315041422
Valeur p : 0.5384669549866221
```

Pris séparément, la distribution des résidus des faux billets suit une loi normale.

Homoscédasticité des résidus

[128]:
```python
# Test d'homoscédasticité de Breusch-Pagan sur vrais billets
# Hypothèse nulle H0 : la variance des résidus ne dépend d'aucun des␣
↪prédicteurs (homoscédasticité)
df = df_ini1[['margin_low','margin_up']].copy()
df['margin_low'] = 1

# Test de Breusch-Pagan
bp_test = ssd.het_breuschpagan(df_ini1["res2"], df)

# ['LM Statistic', 'LM-Test p-value', 'F-Statistic', 'F-Test p-value']
print(bp_test)
```

```
(0.8916663369795798, 0.3450261455730035, 0.8906476220761546,
0.34553710850446584)
```

[129]:
```python
# Test d'homoscédasticité de Breusch-Pagan sur vrais billets
# Hypothèse nulle H0 : la variance des résidus ne dépend d'aucun des␣
↪prédicteurs (homoscédasticité)
df = df_ini0[['margin_low','margin_up']].copy()
```

```
df['margin_low'] = 1

# Test de Breusch-Pagan
bp_test = ssd.het_breuschpagan(df_ini0["res2"], df)

# ['LM Statistic', 'LM-Test p-value', 'F-Statistic', 'F-Test p-value']
print(bp_test)
```

```
(0.022841422629641173, 0.8798701776787922, 0.022749627484S0942,
0.8801720518484054)
```

Pris séparément, on a bien homoscédasticité des résidus pour le sous-échantillon des vrais billets d'une part, et pour le sous-échantillon des faux billets d'autre part.

Mesures agrégées de l'erreur

[130]:
```
df_ana = df_ini[["is_genuine","margin_low","pred2","res2","preds1","ress1"]].
    ↪dropna().copy()
display(df_ana)
```

```
      is_genuine  margin_low  pred2  res2  preds1  ress1
0           True        4.52   4.15  0.37    4.13   0.39
1           True        3.77   4.13 -0.36    4.12  -0.35
2           True        4.40   4.14  0.26    4.13   0.27
3           True        3.62   4.13 -0.51    4.12  -0.50
4           True        4.04   4.03  0.01    4.07  -0.03
…          …          …    …    …     …    …
1495       False        4.42   5.27 -0.85    5.33  -0.91
1496       False        5.27   5.21  0.06    5.21   0.06
1497       False        5.51   5.21  0.30    5.21   0.30
1498       False        5.17   5.19 -0.02    5.17   0.00
1499       False        4.63   5.21 -0.58    5.21  -0.58

[1463 rows x 6 columns]
```

[131]:
```
# Valeur absolue de l'erreur
df_ana["VAE2"] = df_ana["res2"].abs()
df_ana["VAE1"] = df_ana["ress1"].abs()

print(f"Somme des erreurs absolues modèle compact : {df_ana["VAE2"].sum()}")
print(f"Somme des erreurs absolues modèle scindé : {df_ana["VAE1"].sum()}\n")

print(f"Moyenne des erreurs absolues (MAE) modèle compact : {df_ana["VAE2"].
    ↪mean()}")
print(f"Moyenne des erreurs absolues (MAE) modèle scindé : {df_ana["VAE1"].
    ↪mean()}\n")
```

```
Somme des erreurs absolues modèle compact : 462.13
Somme des erreurs absolues modèle scindé : 460.38
```

```
Moyenne des erreurs absolues (MAE) modèle compact : 0.31587833219412165
Moyenne des erreurs absolues (MAE) modèle scindé : 0.31468215994531784
```

[132]:
```python
# Carré de l'erreur
df_ana["CE2"] = df_ana["res2"] ** 2
df_ana["CE1"] = df_ana["ress1"] ** 2

print(f"Somme des carrés des erreurs modèle compact : {df_ana["CE2"].sum()}")
print(f"Somme des carrés des erreurs modèle scindé : {df_ana["CE1"].sum()}\n")

print(f"Moyenne des carrés des erreurs (MSE) modèle compact : {df_ana["CE2"].
↪mean()}")
print(f"Moyenne des carrés des erreurs (MSE) modèle scindé : {df_ana["CE1"].
↪mean()}\n")

print(f"Racine de la moyenne des carrés des erreurs (RMSE) modèle compact : {np.
↪sqrt(df_ana["CE2"].mean())}")
print(f"Racine de la moyenne des carrés des erreurs (RMSE) modèle scindé : {np.
↪sqrt(df_ana["CE1"].mean())}\n")
```

```
Somme des carrés des erreurs modèle compact : 246.83370000000002
Somme des carrés des erreurs modèle scindé : 245.68620000000004

Moyenne des carrés des erreurs (MSE) modèle compact : 0.16871749829118252
Moyenne des carrés des erreurs (MSE) modèle scindé : 0.16793315105946688

Racine de la moyenne des carrés des erreurs (RMSE) modèle compact :
0.41075235640368823
Racine de la moyenne des carrés des erreurs (RMSE) modèle scindé :
0.40979647516720646
```

[133]:
```python
# Erreur relative absolue
df_ana["ERA2"] = (df_ana["res2"]/df_ana["margin_low"]).abs()
df_ana["ERA1"] = (df_ana["ress1"]/df_ana["margin_low"]).abs()

print(f"Somme des erreurs relatives absolues modèle compact : {df_ana["ERA2"].
↪sum()}")
print(f"Somme des erreurs relatives absolues modèle scindé : {df_ana["ERA1"].
↪sum()}\n")

print(f"Moyenne des erreurs relatives absolues (MAPE) modèle compact :␣
↪{df_ana["ERA2"].mean()}")
print(f"Moyenne des erreurs relatives absolues (MAPE) modèle scindé :␣
↪{df_ana["ERA1"].mean()}\n")
```

```
Somme des erreurs relatives absolues modèle compact : 102.82285827294459
```

```
Somme des erreurs relatives absolues modèle scindé : 102.4558633024153

Moyenne des erreurs relatives absolues (MAPE) modèle compact :
0.0702821997764488
Moyenne des erreurs relatives absolues (MAPE) modèle scindé :
0.07003134880547868
```

[134]:

```
# Carré de l'erreur relative
df_ana["CER2"] = (df_ana["res2"]/df_ana["margin_low"]) ** 2
df_ana["CER1"] = (df_ana["ress1"]/df_ana["margin_low"]) ** 2

print(f"Somme des carrés de l'erreur relative modèle compact : {df_ana["CER2"].
  ↪sum()}")
print(f"Somme des carrés de l'erreur relative modèle exhaustif :␣
  ↪{df_ana["CER1"].sum()}\n")

print(f"Moyenne des carrés de l'erreur relative modèle compact :␣
  ↪{df_ana["CER2"].mean()}")
print(f"Moyenne des carrés de l'erreur relative modèle exhaustif :␣
  ↪{df_ana["CER1"].mean()}\n")

print(f"Racine de la moyenne des carrés de l'erreur relative modèle compact :␣
  ↪{np.sqrt(df_ana["CER2"].mean())}")
print(f"Racine de la moyenne des carrés de l'erreur relative modèle exhaustif :␣
  ↪{np.sqrt(df_ana["CER1"].mean())}\n")
```

```
Somme des carrés de l'erreur relative modèle compact : 12.122603227273004
Somme des carrés de l'erreur relative modèle exhaustif : 12.061179863869516

Moyenne des carrés de l'erreur relative modèle compact : 0.008286126607842108
Moyenne des carrés de l'erreur relative modèle exhaustif : 0.008244142080566997

Racine de la moyenne des carrés de l'erreur relative modèle compact :
0.09102816381671174
Racine de la moyenne des carrés de l'erreur relative modèle exhaustif :
0.09079725811150355
```

3.3 - Affectation des valeurs

[135]:

```
df_tmp = df_ini.loc[df_ini["margin_low"].
  ↪isna(),["is_genuine","margin_low","preds1"]]
display(df_tmp)
print(len(df_tmp))
```

	is_genuine	margin_low	preds1
72	True	NaN	4.10
99	True	NaN	4.11

```
151          True          NaN    4.12
197          True          NaN    4.06
241          True          NaN    4.12
251          True          NaN    4.13
284          True          NaN    4.10
334          True          NaN    4.12
410          True          NaN    4.11
413          True          NaN    4.10
445          True          NaN    4.12
481          True          NaN    4.15
505          True          NaN    4.12
611          True          NaN    4.09
654          True          NaN    4.15
675          True          NaN    4.17
710          True          NaN    4.11
739          True          NaN    4.11
742          True          NaN    4.10
780          True          NaN    4.11
798          True          NaN    4.12
844          True          NaN    4.12
845          True          NaN    4.13
871          True          NaN    4.11
895          True          NaN    4.12
919          True          NaN    4.17
945          True          NaN    4.12
946          True          NaN    4.09
981          True          NaN    4.12
1076        False          NaN    5.28
1121        False          NaN    5.30
1176        False          NaN    5.36
1303        False          NaN    5.40
1315        False          NaN    5.19
1347        False          NaN    5.10
1435        False          NaN    5.12
1438        False          NaN    5.26
```

```
37
```

On utilise la prédiction *preds1* pour imputer les valeurs manquantes, car même si elle est à peine plus performante que *pred2*, elle vérifie mieux les critères de validité de la régression linéaire et surtout elle s'inscrit mieux dans notre démarche qui met clairement en avant que les vrais billets et les faux billets doivent être traités séparément.

[136]:
```
df_work = (df_ini.iloc[:,:7]).copy()
df_work["margin_low"] = df_work["margin_low"].fillna(df_ini["preds1"])
display(df_work)
```

```
      is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
0           True    171.81       104.86        104.95        4.52       2.89
```

1	True	171.46	103.36	103.66	3.77	2.99
2	True	172.69	104.48	103.50	4.40	2.94
3	True	171.36	103.91	103.94	3.62	3.01
4	True	171.73	104.28	103.46	4.04	3.48
…	…	…	…	…	…	…
1495	False	171.75	104.38	104.17	4.42	3.09
1496	False	172.19	104.63	104.44	5.27	3.37
1497	False	171.80	104.01	104.12	5.51	3.36
1498	False	172.06	104.28	104.06	5.17	3.46
1499	False	171.47	104.15	103.82	4.63	3.37

	length
0	112.83
1	113.09
2	113.16
3	113.51
4	112.54
…	…
1495	111.28
1496	110.97
1497	111.95
1498	112.25
1499	112.07

[1500 rows x 7 columns]

[137]:
```
display(df_tmp.merge(df_work["margin_low"], how='left', left_index=True,␣
↪right_index=True))
```

	is_genuine	margin_low_x	preds1	margin_low_y
72	True	NaN	4.10	4.10
99	True	NaN	4.11	4.11
151	True	NaN	4.12	4.12
197	True	NaN	4.06	4.06
241	True	NaN	4.12	4.12
251	True	NaN	4.13	4.13
284	True	NaN	4.10	4.10
334	True	NaN	4.12	4.12
410	True	NaN	4.11	4.11
413	True	NaN	4.10	4.10
445	True	NaN	4.12	4.12
481	True	NaN	4.15	4.15
505	True	NaN	4.12	4.12
611	True	NaN	4.09	4.09
654	True	NaN	4.15	4.15
675	True	NaN	4.17	4.17
710	True	NaN	4.11	4.11
739	True	NaN	4.11	4.11

```
742          True             NaN     4.10          4.10
780          True             NaN     4.11          4.11
798          True             NaN     4.12          4.12
844          True             NaN     4.12          4.12
845          True             NaN     4.13          4.13
871          True             NaN     4.11          4.11
895          True             NaN     4.12          4.12
919          True             NaN     4.17          4.17
945          True             NaN     4.12          4.12
946          True             NaN     4.09          4.09
981          True             NaN     4.12          4.12
1076        False             NaN     5.28          5.28
1121        False             NaN     5.30          5.30
1176        False             NaN     5.36          5.36
1303        False             NaN     5.40          5.40
1315        False             NaN     5.19          5.19
1347        False             NaN     5.10          5.10
1435        False             NaN     5.12          5.12
1438        False             NaN     5.26          5.26
```

[138]:

```
#Le nombre de valeurs présentes dans chacune des colonnes
for c in list(df_work):
    print("\nColonne", c, "- Nombre de valeurs NaN :", ((df_work[c]).isna()).
 ↪sum())
    print("Colonne", c, "- Nombre de valeurs non-vides :", df_work.shape[0] -␣
 ↪((df_work[c]).isna()).sum())
```

```
Colonne is_genuine - Nombre de valeurs NaN : 0
Colonne is_genuine - Nombre de valeurs non-vides : 1500

Colonne diagonal - Nombre de valeurs NaN : 0
Colonne diagonal - Nombre de valeurs non-vides : 1500

Colonne height_left - Nombre de valeurs NaN : 0
Colonne height_left - Nombre de valeurs non-vides : 1500

Colonne height_right - Nombre de valeurs NaN : 0
Colonne height_right - Nombre de valeurs non-vides : 1500

Colonne margin_low - Nombre de valeurs NaN : 0
Colonne margin_low - Nombre de valeurs non-vides : 1500

Colonne margin_up - Nombre de valeurs NaN : 0
Colonne margin_up - Nombre de valeurs non-vides : 1500

Colonne length - Nombre de valeurs NaN : 0
Colonne length - Nombre de valeurs non-vides : 1500
```

L'affectation des valeurs manquantes s'est bien passée.

[139]:
```
df_work.to_csv("billets_filled.csv")
```

3.4 - Impacts sur les indicateurs statistiques de l'échantillon complet

3.4.1 - Indicateurs globaux

[140]:
```
display(df_work["margin_low"].describe())
```

```
count    1500.000000
mean        4.482920
std         0.659986
min         2.980000
25%         4.030000
50%         4.310000
75%         4.870000
max         6.900000
Name: margin_low, dtype: float64
```

[141]:
```
display(df_ini["margin_low"].describe())
```

```
count    1463.000000
mean        4.485967
std         0.663813
min         2.980000
25%         4.015000
50%         4.310000
75%         4.870000
max         6.900000
Name: margin_low, dtype: float64
```

L'écart-type de la variable *margin_low* a légèrement augmenté : il semble qu'on ait plutôt ajouté des valeurs basses, la valeur du premier quartile a un peu diminué.

[142]:
```
display(df_work.loc[df_work["is_genuine"]==True,"margin_low"].describe())
```

```
count    1000.000000
mean        4.116130
std         0.314481
min         2.980000
25%         3.910000
50%         4.110000
75%         4.330000
max         5.040000
Name: margin_low, dtype: float64
```

[143]:
```
display(df_ini.loc[df_ini["is_genuine"]==True,"margin_low"].describe())
```

```
count    971.000000
mean       4.116097
```

```
std        0.319124
min        2.980000
25%        3.905000
50%        4.110000
75%        4.340000
max        5.040000
Name: margin_low, dtype: float64
```

[144]:
```
display(df_work.loc[df_work["is_genuine"]==False,"margin_low"].describe())
```

```
count    500.000000
mean       5.216500
std        0.549242
min        3.820000
25%        4.840000
50%        5.190000
75%        5.590000
max        6.900000
Name: margin_low, dtype: float64
```

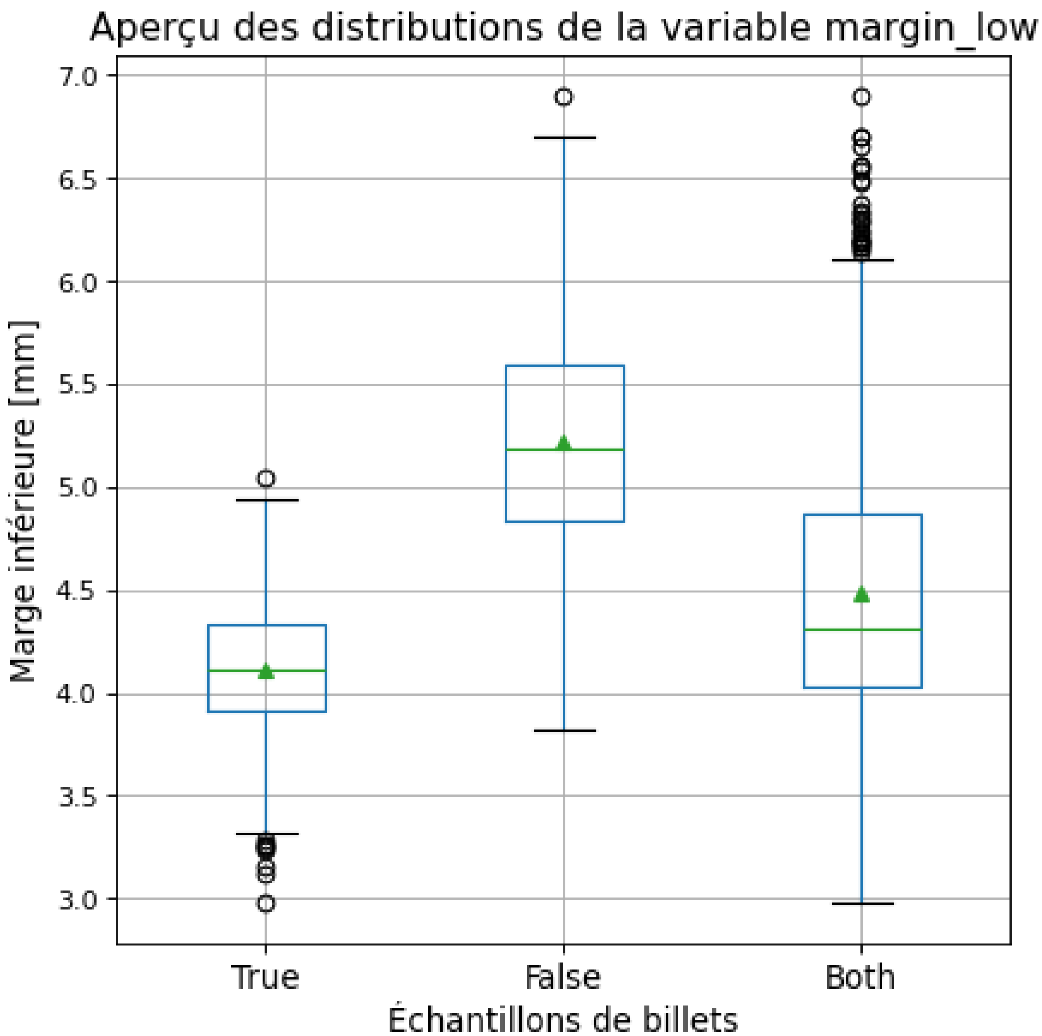
[145]:
```
display(df_ini.loc[df_ini["is_genuine"]==False,"margin_low"].describe())
```

```
count    492.000000
mean       5.215935
std        0.553531
min        3.820000
25%        4.840000
50%        5.190000
75%        5.592500
max        6.900000
Name: margin_low, dtype: float64
```

3.4.2 - Nouvelle distribution de margin_low

[146]:
```
fig = plt.figure(figsize=(6,6), dpi=80)

df_work.loc[df_work["is_genuine"]==True].boxplot(column=["margin_low"],\
                                                 positions=[1], widths=[0.4],
                                                 showmeans=True)

df_work.loc[df_work["is_genuine"]==False].boxplot(column=["margin_low"],\
                                                  positions=[2], widths=[0.4],
                                                  showmeans=True)


df_work.boxplot(column=["margin_low"],\
                positions=[3], widths=[0.4],
                showmeans=True)

plt.xticks(ticks=np.arange(1,4), labels=["True","False","Both"], fontsize=12)
```

```
plt.xlabel("Échantillons de billets", fontsize=12)
plt.ylabel("Marge inférieure [mm]", fontsize=12)
plt.xlim([0.5,3.5])
plt.title("Aperçu des distributions de la variable margin_low", fontsize=14)
#plt.savefig("Boxplot_diagonal.png")
plt.show()
```



[147]:
```
# Tests de Kolmogorov-Smirnov
# Hypothèse nulle : les échantillons de vrais billets et de faux billets
 ↪provienent de la même distribution
res = st.ks_2samp(df_work.loc[df_work["is_genuine"]==True]["margin_low"],\
                  df_work.loc[df_work["is_genuine"]==False]["margin_low"],\
                  alternative='two-sided')
```

```
# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")

# Hypothèse nulle : les vrais billets ont une marge inférieure plus petite ou␣
↪égale à celle faux billets
res = st.ks_2samp(df_work.loc[df_work["is_genuine"]==True]["margin_low"],\
                  df_work.loc[df_work["is_genuine"]==False]["margin_low"],\
                  alternative='less')

# Afficher les résultats
print(f"Statistique signée du test KS : {res.statistic_sign * res.statistic}")
print(f"Valeur p : {res.pvalue}\n")
```

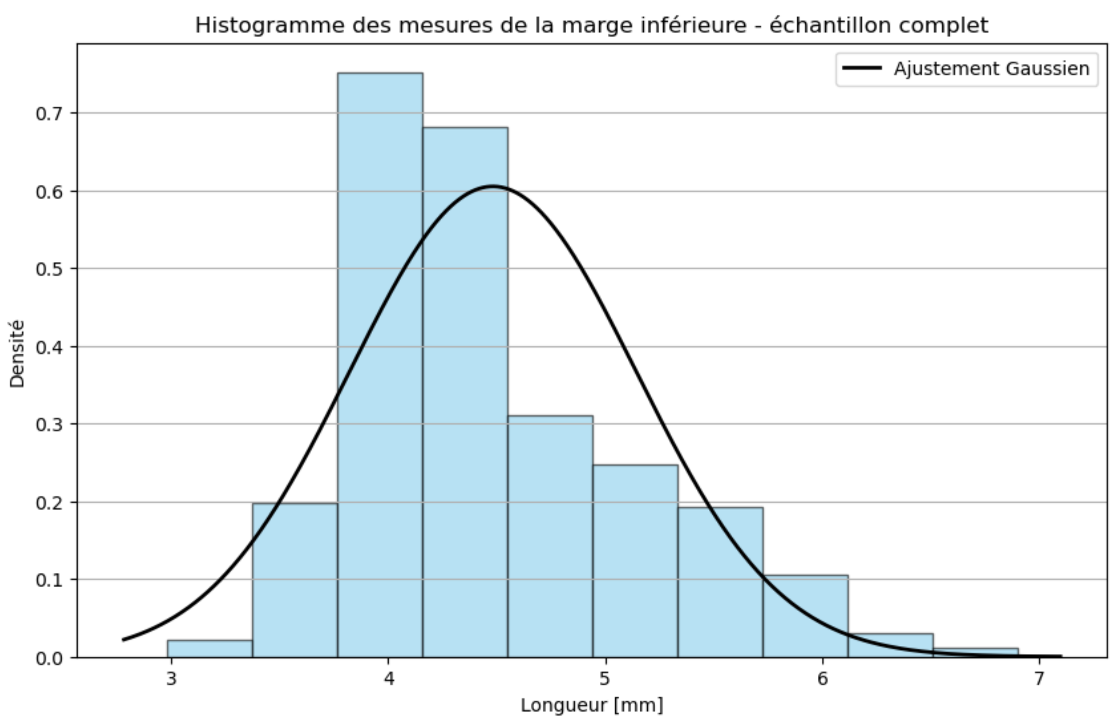
```
Statistique signée du test KS : 0.815
Valeur p : 7.8108485432309e-225

Statistique signée du test KS : -0.0
Valeur p : 1.0
```

[148]:

```
data = df_work["margin_low"].sort_values().dropna()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)
```

```
# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - échantillon␣
↪complet")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 4.48
Écart type : 0.66
Z_min : -2.28
Z_max : 3.66
```

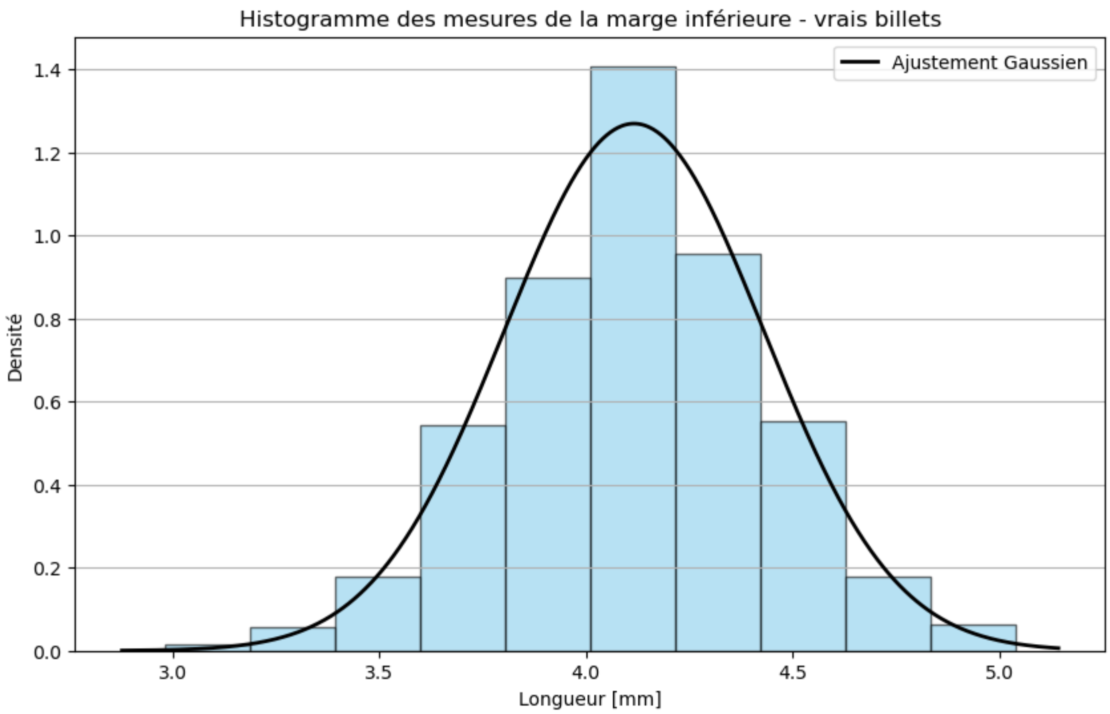


```
Statistique du test de Shapiro-Wilk : 0.9355406803502633
Valeur p : 5.689955466119613e-25
```

[149]:
```python
data = df_work.loc[df_work["is_genuine"]==True,"margin_low"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)    # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))

# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
 ↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - vrais billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
```

```python
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

```
Moyenne : 4.12
Écart type : 0.31
Z_min : -3.61
Z_max : 2.94
```

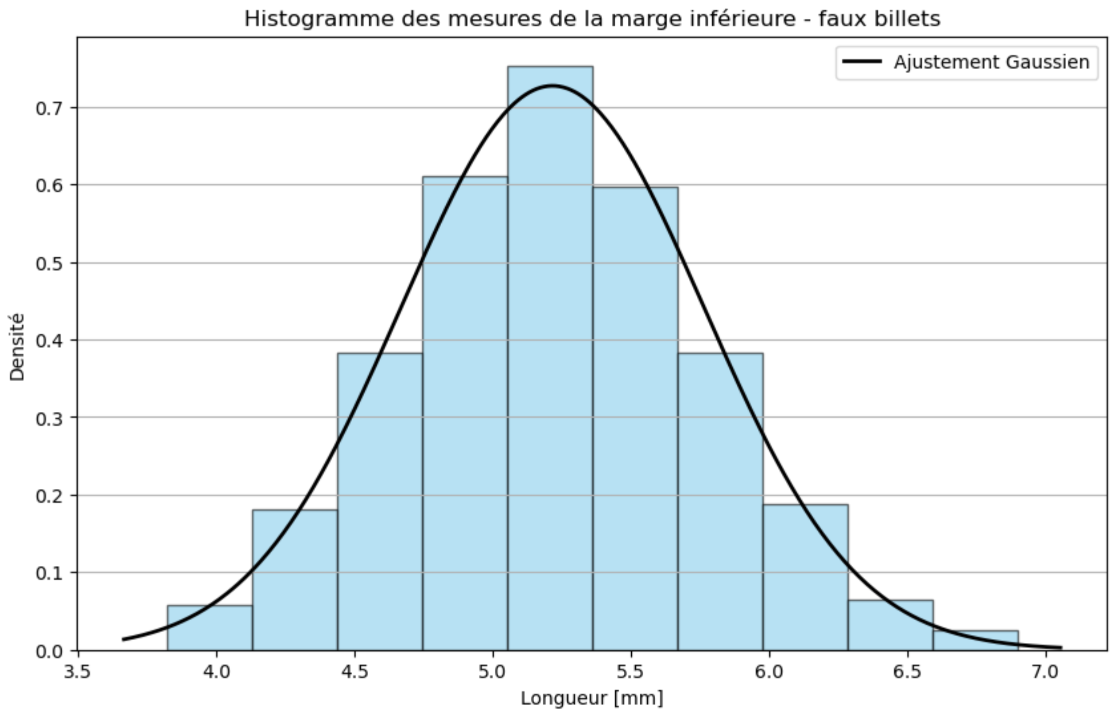



```
Statistique du test de Shapiro-Wilk : 0.9977378776029429
Valeur p : 0.18942293795663795
```

[150]:
```python
data = df_work.loc[df_work["is_genuine"]==False,"margin_low"].sort_values()

# Estimer la moyenne et l'écart type
mn = data.mean()
sd = data.std(ddof=1)     # Estimateur sans biais

# Afficher les valeurs calculées
print(f"Moyenne : {round(mn,2)}")
print(f"Écart type : {round(sd,2)}")
print(f"Z_min : {round((data.min()-mn)/sd,2)}")
print(f"Z_max : {round((data.max()-mn)/sd,2)}")

# Tracer l'histogramme avec une courbe gaussienne
plt.figure(figsize=(10, 6))
```

```python
# Histogramme
#plt.hist(data, bins=np.linspace(15,100,18), density=True, color='skyblue',␣
↪edgecolor='black', alpha=0.6)
plt.hist(data, density=True, color='skyblue', edgecolor='black', alpha=0.6)

# Ajustement gaussien
mu, std = st.norm.fit(data)
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, data.count())
p = st.norm.pdf(x, mu, std)

# Tracer la courbe gaussienne
plt.plot(x, p, 'k', linewidth=2, label='Ajustement Gaussien')

# Ajouter les titres et légendes en français
plt.title("Histogramme des mesures de la marge inférieure - faux billets")
plt.xlabel('Longueur [mm]')
plt.ylabel('Densité')
plt.legend()
plt.grid(axis='y')

# Afficher le graphique
plt.show()

# Test de Shapiro-Wilk
stat, p_value = st.shapiro(data)

# Afficher les résultats
print(f"Statistique du test de Shapiro-Wilk : {stat}")
print(f"Valeur p : {p_value}")
```

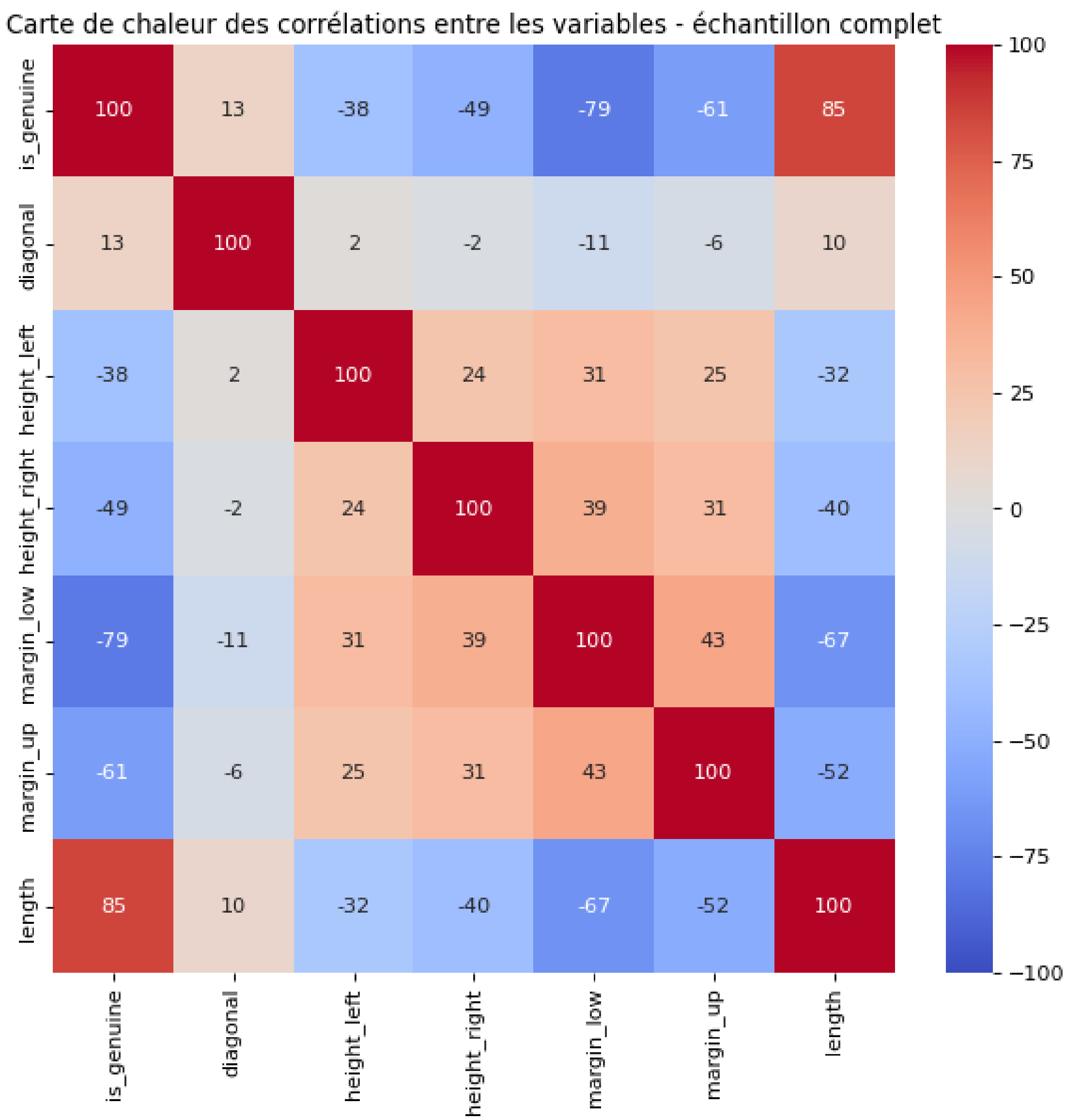
```
Moyenne : 5.22
Écart type : 0.55
Z_min : -2.54
Z_max : 3.07
```



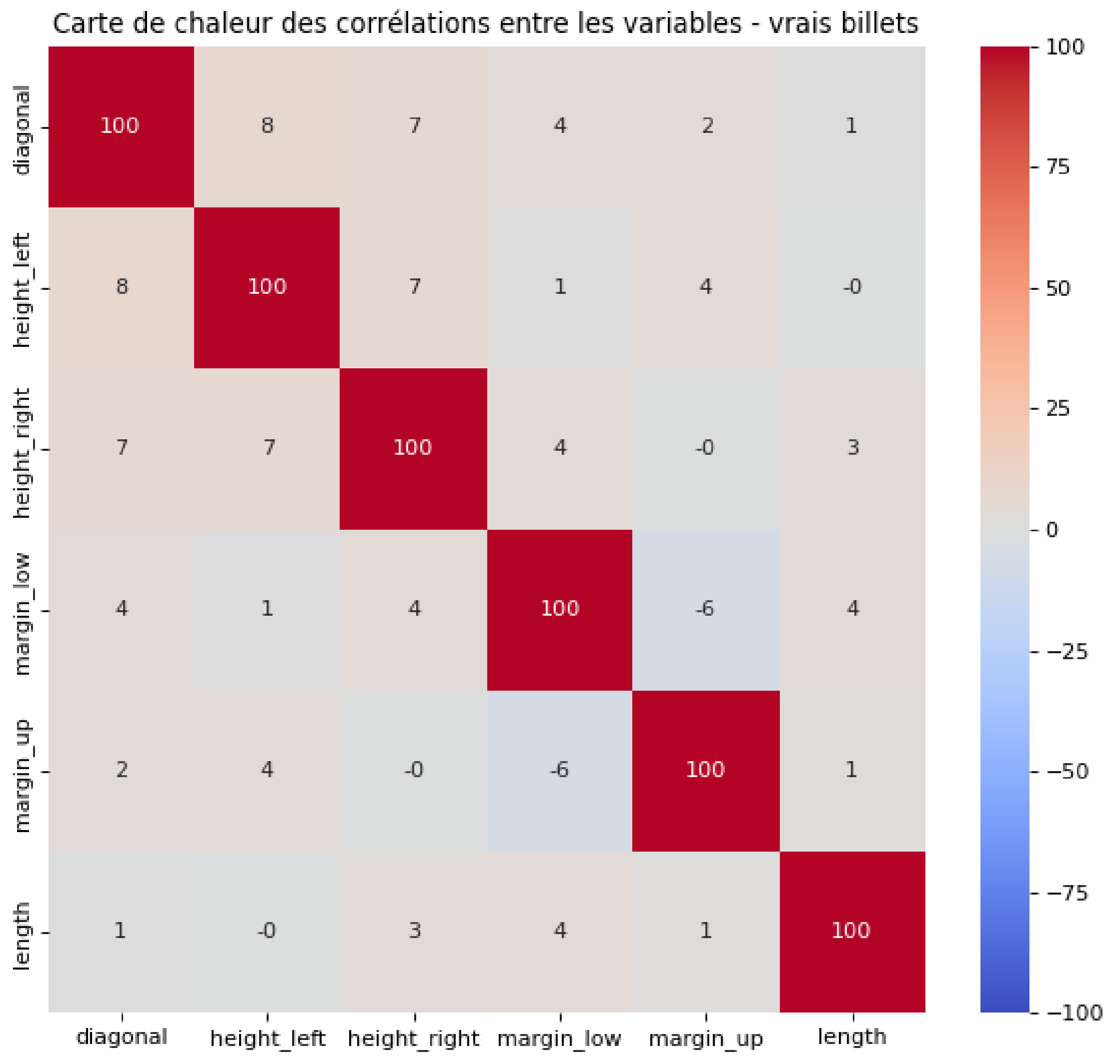

```
Statistique du test de Shapiro-Wilk : 0.9973795670951656
Valeur p : 0.6191380307485279
```

3.4.3 - Analyse bivariées

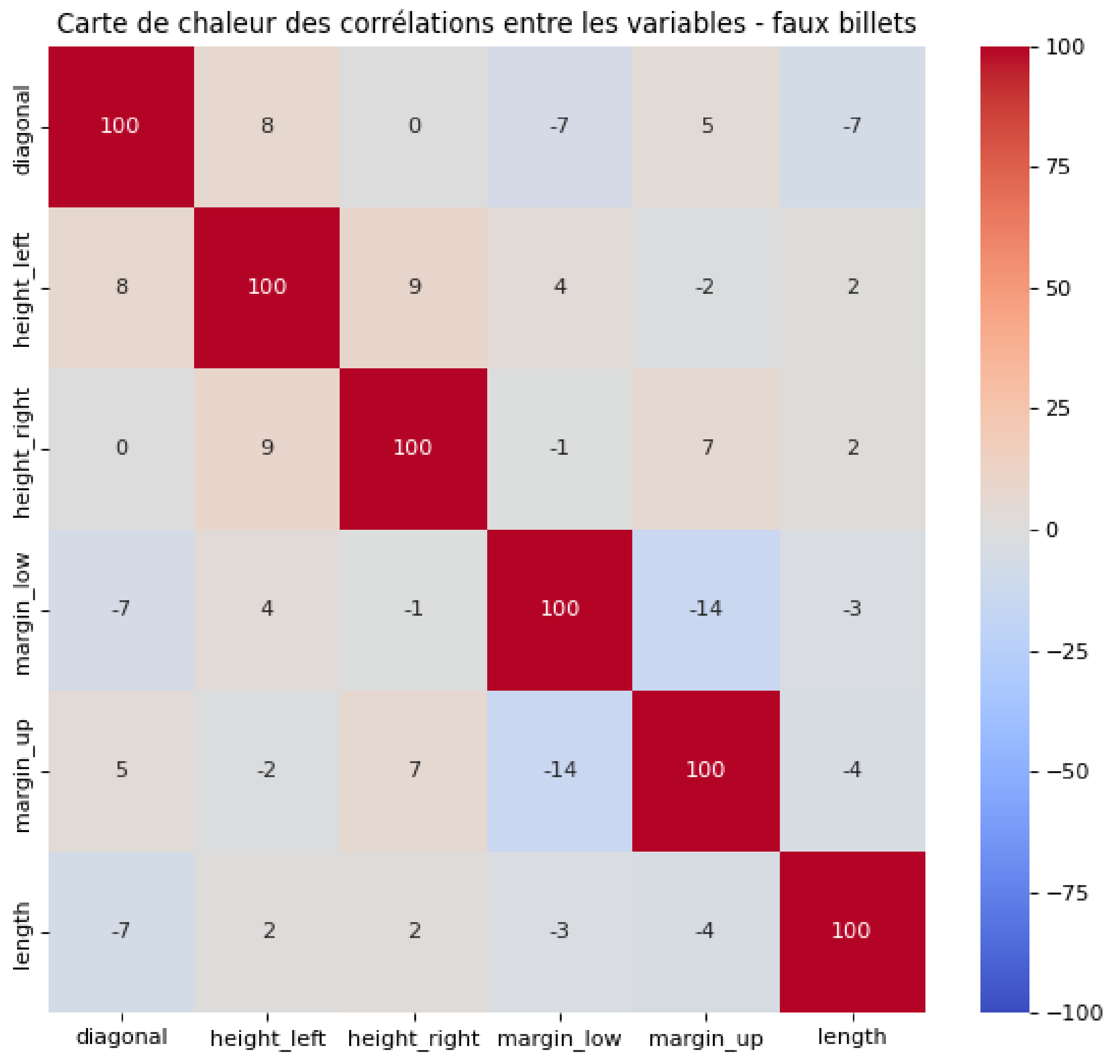
[151]:
```
fig = plt.figure(figsize=(9,8), dpi=80)

sb.heatmap(100*df_work.corr(method='pearson'),\
           vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
#plt.xticks(rotation=45)
plt.title("Carte de chaleur des corrélations entre les variables - échantillon␣
 ↪complet")
#plt.savefig("Heatmap.png")
plt.show()
```



[152]:
```
fig = plt.figure(figsize=(9,8), dpi=80)

sb.heatmap(100*df_work.loc[df_work["is_genuine"]==True,list(df_work.columns)[1:
↪]].dropna().corr(method='pearson'),\
           vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
#plt.xticks(rotation=45)
plt.title("Carte de chaleur des corrélations entre les variables - vrais␣
↪billets")
#plt.savefig("Heatmap.png")
plt.show()
```



[153]:
```
fig = plt.figure(figsize=(9,8), dpi=80)

sb.heatmap(100*df_work.loc[df_work["is_genuine"]==False,list(df_work.columns)[1:
  ↪]].dropna().corr(method='pearson'),\
            vmin=-100,vmax=100,cmap='coolwarm', fmt='.0f', annot=True)
#plt.xticks(rotation=45)
plt.title("Carte de chaleur des corrélations entre les variables - faux␣
  ↪billets")
#plt.savefig("Heatmap.png")
plt.show()
```



Les imputations n'ont pas eu d'impact significatif sur les indicateurs statistiques des données.

Partie 4 - Modèles prédictifs

La variable qu'on cherche à prédire est *is_genuine*, à savoir si le billet est vrai ou faux, selon la mesure de cotes sur le billet, contenues dans les autres variables du jeu de données. On considère un faux billet comme étant un "positif" et un vrai billet comme étant un "négatif".

[154]:
```
X = df_work[list(df_work.columns)[1:]]
y = df_work["is_genuine"]
display(X)
```

	diagonal	height_left	height_right	margin_low	margin_up	length
0	171.81	104.86	104.95	4.52	2.89	112.83
1	171.46	103.36	103.66	3.77	2.99	113.09
2	172.69	104.48	103.50	4.40	2.94	113.16

```
3       171.36     103.91      103.94      3.62      3.01  113.51
4       171.73     104.28      103.46      4.04      3.48  112.54
…        …        …         …         …        …       …
1495    171.75     104.38      104.17      4.42      3.09  111.28
1496    172.19     104.63      104.44      5.27      3.37  110.97
1497    171.80     104.01      104.12      5.51      3.36  111.95
1498    172.06     104.28      104.06      5.17      3.46  112.25
1499    171.47     104.15      103.82      4.63      3.37  112.07

[1500 rows x 6 columns]
```

[155]:
```python
# Découpage de l'échantillon en folds pour la validation croisée
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Performances des modèles testés, ligne par ligne
modrows=[]
```

Il faut définir la métrique par laquelle nous allons éprouver les différents modèles prédictifs afin de les classer. Le cahier des charges indique sans plus de précision qu'il faut simplement détecter un maximum de faux billets. Nous concernant, cela revient à dire qu'il faut maximiser le score de rappel, ce qui revient à minimiser le nombre de faux-négatifs. Mais cette approche peut facilement être prise en défaut : par exemple il nous suffit de prédire que n'importe quel billet testé sera faux, et on obteint un modèle très simple qui est optimal vis-à-vis du score de rappel. Il faut aller plus loin et se fixer un ratio *rc* représentatif du différentiel entre le coût de traitement d'un faux-négatif et le coût de traitement d'un faux-positif.

[156]:
```python
# par exemple
rc = 20
# signifie qu'un faux-négatif coûte rc fois plus cher qu'un faux-positif.
```

À partir de ce ratio, on peut se définir un F_β-score qui va nous permettre de pondérer l'importance relative du rappel (c'est-à-dire de limiter le nombre de faux-négatifs) par rapport à la précision (limiter le nombre de faux-positifs). La formulation du F_β-score fait apparaître une pondération β^2 devant le terme 1/rappel (avant inversion), on peut donc retenir $\beta^2 = rc$. Notre métrique à maximiser devient donc :

$$F_\beta = (1 + rc)\frac{precision * rappel}{rc * precision + rappel}$$

On cherchera à maximiser ce F-score pour les différents modèles.

4.1 - Modèle K-Means

L'algorithme K-Means est une méthode d'apprentissage non-supervisée, basée sur les distances entre individus. Une normalisation des données est nécessaire de façon à ce que les variables mesurant les marges ne soient pas masquées par les autres cotes. En guise de test, et pour mesurer la performance maximale possible de ce modèle, on applique K-Means sur l'échantillon complet.

4.1.1 - K-Means sur l'échantillon complet

[157]:

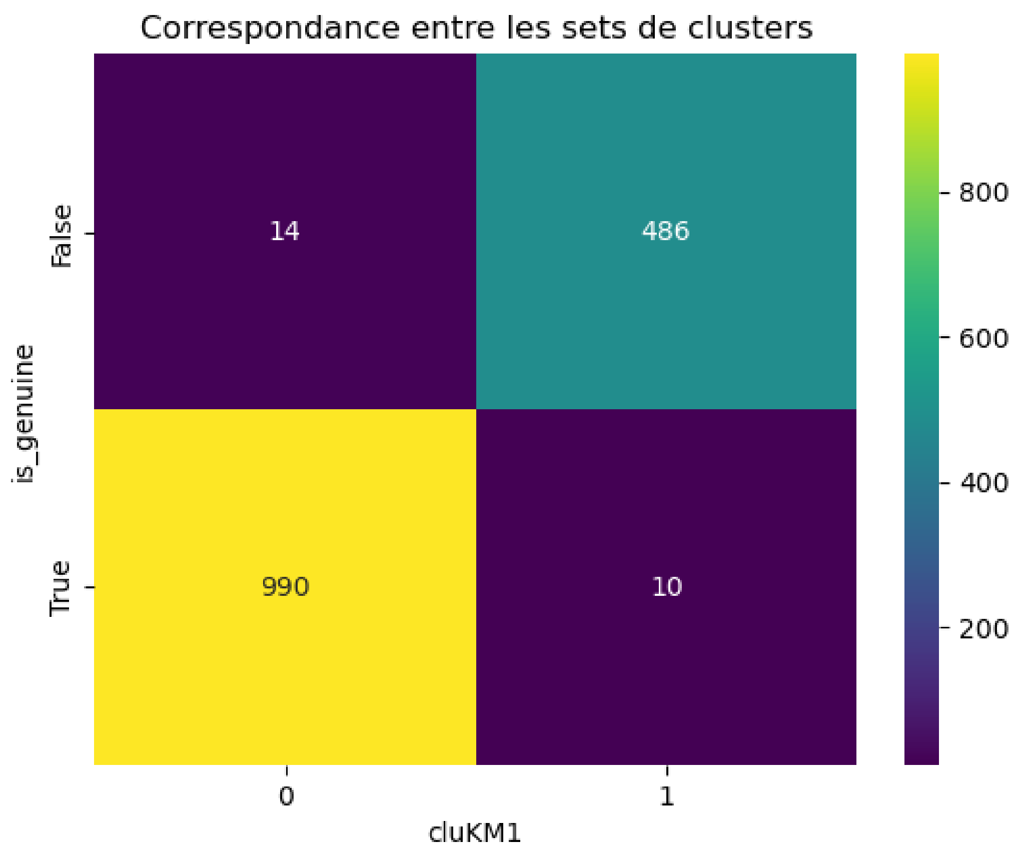
```
# Normalisation des données
scaler_full = StandardScaler(with_std=True)

scaler_full.fit(X)

#Transformation
data_scaled = scaler_full.transform(X)

idx = ["mean", "std"]

df_scaled = pd.DataFrame(data_scaled)
display(pd.DataFrame(data_scaled).describe().round(2).loc[idx, :])
df_scaled.rename(mapper={i: "z_"+list(X.columns)[i] for i in range(df_scaled.
 ↪shape[1])}, axis=1, inplace=True)
display(df_scaled)
```

```
        0    1    2    3    4    5
mean -0.0  0.0 -0.0 -0.0 -0.0  0.0
std   1.0  1.0  1.0  1.0  1.0  1.0

      z_diagonal  z_height_left  z_height_right  z_margin_low  z_margin_up  \
0      -0.486540       2.774123        3.163240      0.056202    -1.128325
1      -1.633729      -2.236535       -0.799668     -1.080566    -0.696799
2       2.397823       1.504756       -1.291191     -0.125681    -0.912562
3      -1.961498      -0.399294        0.060498     -1.307919    -0.610494
4      -0.748754       0.836669       -1.414072     -0.671329     1.417677
…            …            …             …             …           …
1495   -0.683201       1.170713        0.767063     -0.095367    -0.265273
1496    0.758981       2.005822        1.596509      1.192969     0.942999
1497   -0.519316      -0.065250        0.613462      1.556735     0.899846
1498    0.332882       0.836669        0.429141      1.041400     1.331372
1499   -1.600953       0.402412       -0.308144      0.222928     0.942999

      z_length
0     0.173651
1     0.471666
2     0.551901
3     0.953075
4    -0.158750
…          …
1495 -1.602978
1496 -1.958303
1497 -0.835016
1498 -0.491152
1499 -0.697470

[1500 rows x 6 columns]
```

[158]:

```
n_clust=2
kmeans = KMeans(n_clusters=n_clust, random_state=42)
kmeans.fit(df_scaled)
clusters = kmeans.fit_predict(df_scaled)
display(pd.Series.value_counts(clusters).sort_index())
df_work["cluKM1"] = clusters

contingency_table = pd.crosstab(df_work['is_genuine'], df_work['cluKM1'])
fig=plt.figure()
sb.heatmap(contingency_table, annot=True, fmt='d', cmap='viridis')
plt.title("Correspondance entre les sets de clusters")
plt.show()

display(df_work.
  ↪loc[df_work["is_genuine"]==df_work["cluKM1"],["is_genuine","cluKM1"]])
```

```
0    1004
1     496
Name: count, dtype: int64
```



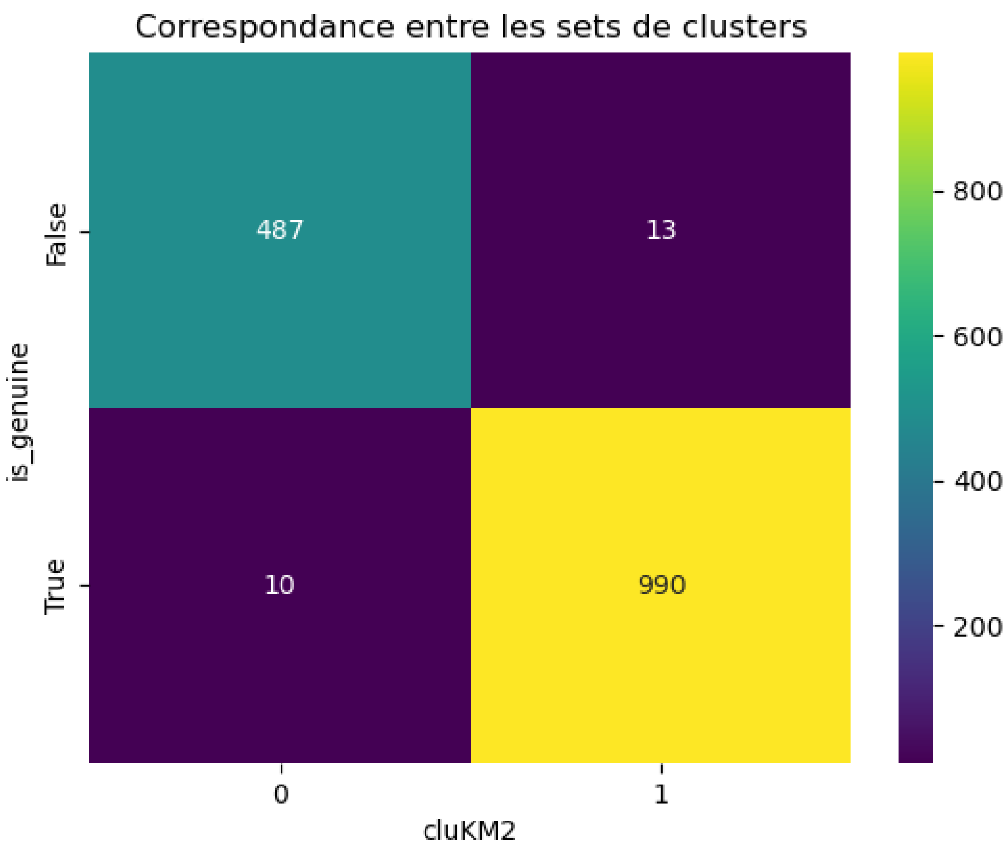

```
is_genuine  cluKM1
```

```
0           True       1
253         True       1
341         True       1
580         True       1
626         True       1
669         True       1
724         True       1
728         True       1
743         True       1
946         True       1
1025       False       0
1073       False       0
1081       False       0
1083       False       0
1103       False       0
1104       False       0
1122       False       0
1160       False       0
1267       False       0
1362       False       0
1383       False       0
1407       False       0
1412       False       0
1482       False       0
```

[159]:

```
n_clust=2
kmeans = KMeans(n_clusters=n_clust, random_state=123)
kmeans.fit(df_scaled)
clusters = kmeans.fit_predict(df_scaled)
display(pd.Series.value_counts(clusters).sort_index())
df_work["cluKM2"] = clusters

contingency_table = pd.crosstab(df_work['is_genuine'], df_work['cluKM2'])
fig=plt.figure()
sb.heatmap(contingency_table, annot=True, fmt='d', cmap='viridis')
plt.title("Correspondance entre les sets de clusters")
plt.show()

display(df_work.loc[df_work["is_genuine"]!
 ↪=df_work["cluKM2"],["is_genuine","cluKM2"]])
```

```
0     497
1    1003
Name: count, dtype: int64
```

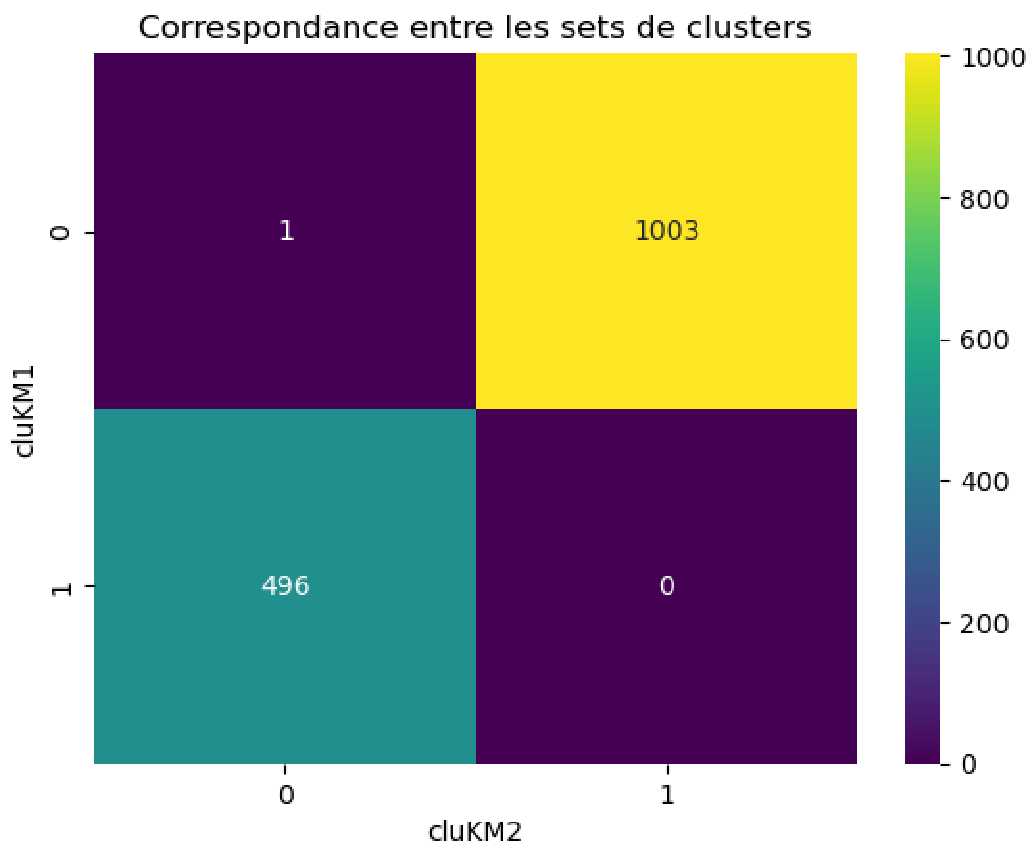


	is_genuine	cluKM2
0	True	0
253	True	0
341	True	0
580	True	0
626	True	0
669	True	0
724	True	0
728	True	0
743	True	0
946	True	0
1025	False	1
1073	False	1
1081	False	1
1083	False	1
1103	False	1
1122	False	1
1160	False	1
1267	False	1
1362	False	1

```
1383       False       1
1407       False       1
1412       False       1
1482       False       1
```

[160]:

```
contingency_table = pd.crosstab(df_work['cluKM1'], df_work['cluKM2'])
fig=plt.figure()
sb.heatmap(contingency_table, annot=True, fmt='d', cmap='viridis')
plt.title("Correspondance entre les sets de clusters")
plt.show()
```



Comme la numérotation des clusters est arbitraire, on peut considérer que le cluster avec le plus d'individus est le cluster des vrais billets.

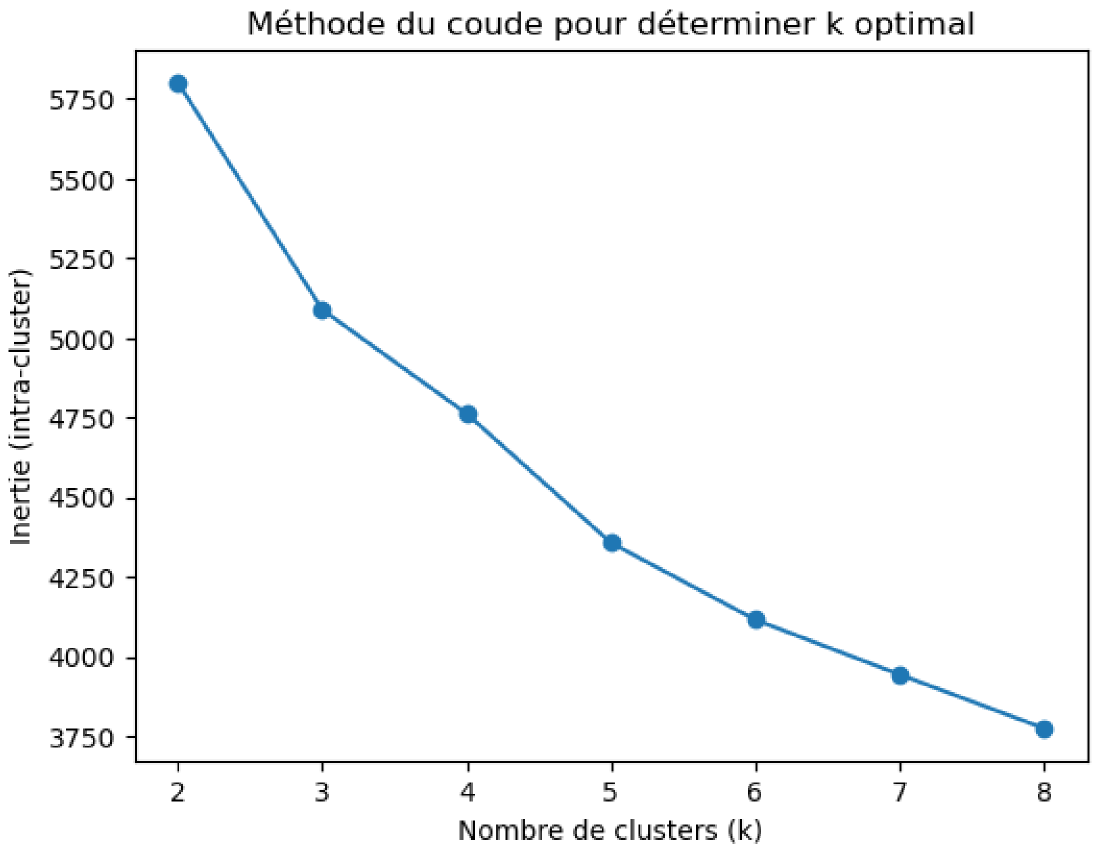
[161]:

```
eff = df_work["cluKM1"].sort_index()
if eff.iloc[0] < eff.iloc[1]:
    df_work["clu"] = (df_work["cluKM1"]==1)
else:
    df_work["clu"] = (df_work["cluKM1"]==0)
```

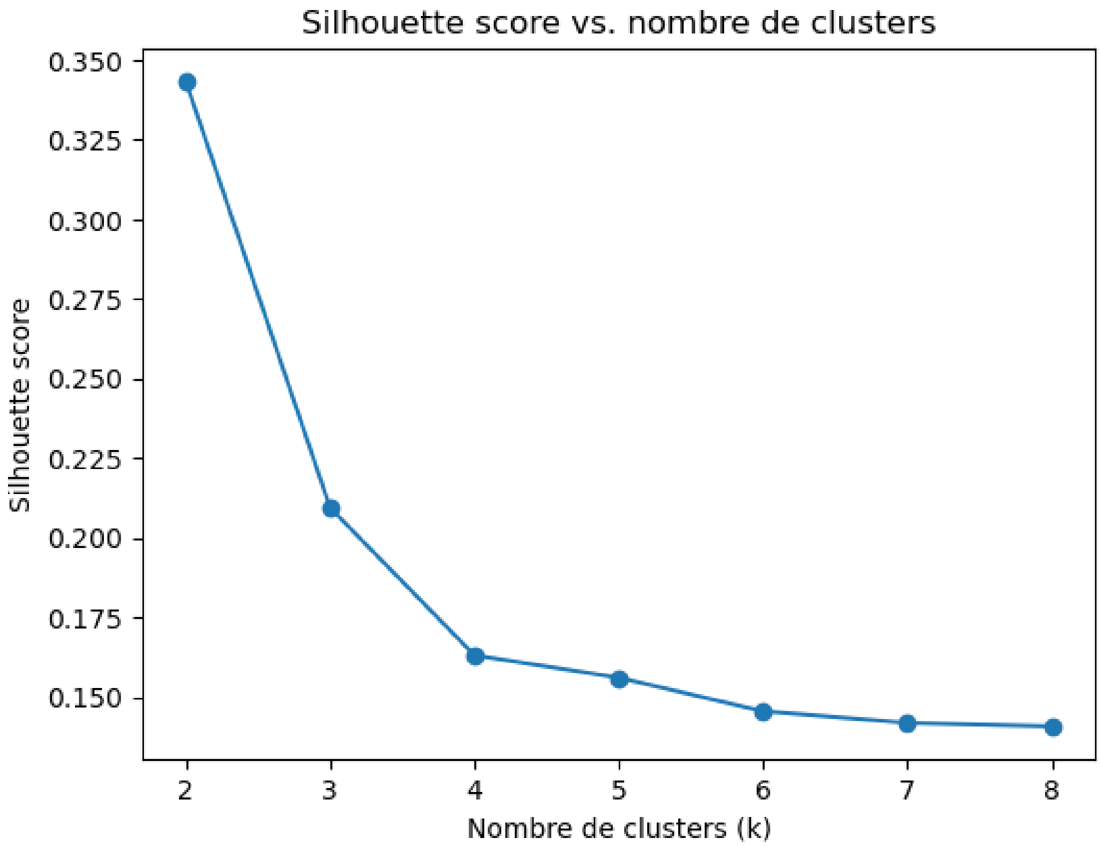
[162]:
```
cfm = confusion_matrix(y_true=df_work["is_genuine"], y_pred=df_work["clu"],␣
↪labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")
print(df_work["clu"].value_counts())
print(df_work["is_genuine"].value_counts())
print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion :
 [[486  14]
 [ 10 990]]
[False, True]

clu
True     1004
False     496
Name: count, dtype: int64
is_genuine
True     1000
False     500
Name: count, dtype: int64

Exactitude du modèle : 0.984
Précision du modèle : 0.9798387096774194
Rappel du modèle : 0.972
```

[163]:
```
inertias = []
k_range = range(2, 9)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=127)
    kmeans.fit(df_scaled)
    inertias.append(kmeans.inertia_)

plt.plot(k_range, inertias, marker='o')
plt.xlabel('Nombre de clusters (k)')
plt.ylabel('Inertie (intra-cluster)')
plt.title('Méthode du coude pour déterminer k optimal')
plt.show()
```

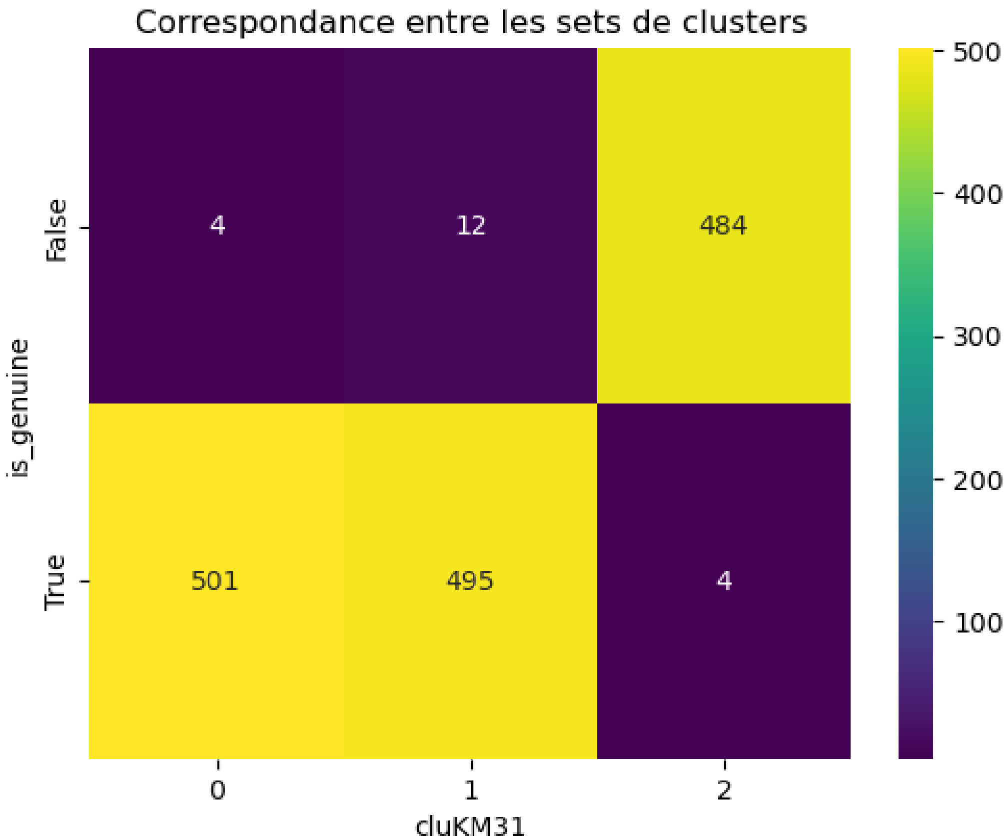


[164]:
```
silhouette_scores = []

for k in range(2, 9):
    kmeans = KMeans(n_clusters=k, random_state=127)
    labels = kmeans.fit_predict(df_scaled)
    score = silhouette_score(df_scaled, labels)
    silhouette_scores.append(score)

plt.plot(range(2, 9), silhouette_scores, marker='o')
plt.xlabel('Nombre de clusters (k)')
plt.ylabel('Silhouette score')
plt.title('Silhouette score vs. nombre de clusters')
plt.show()
```

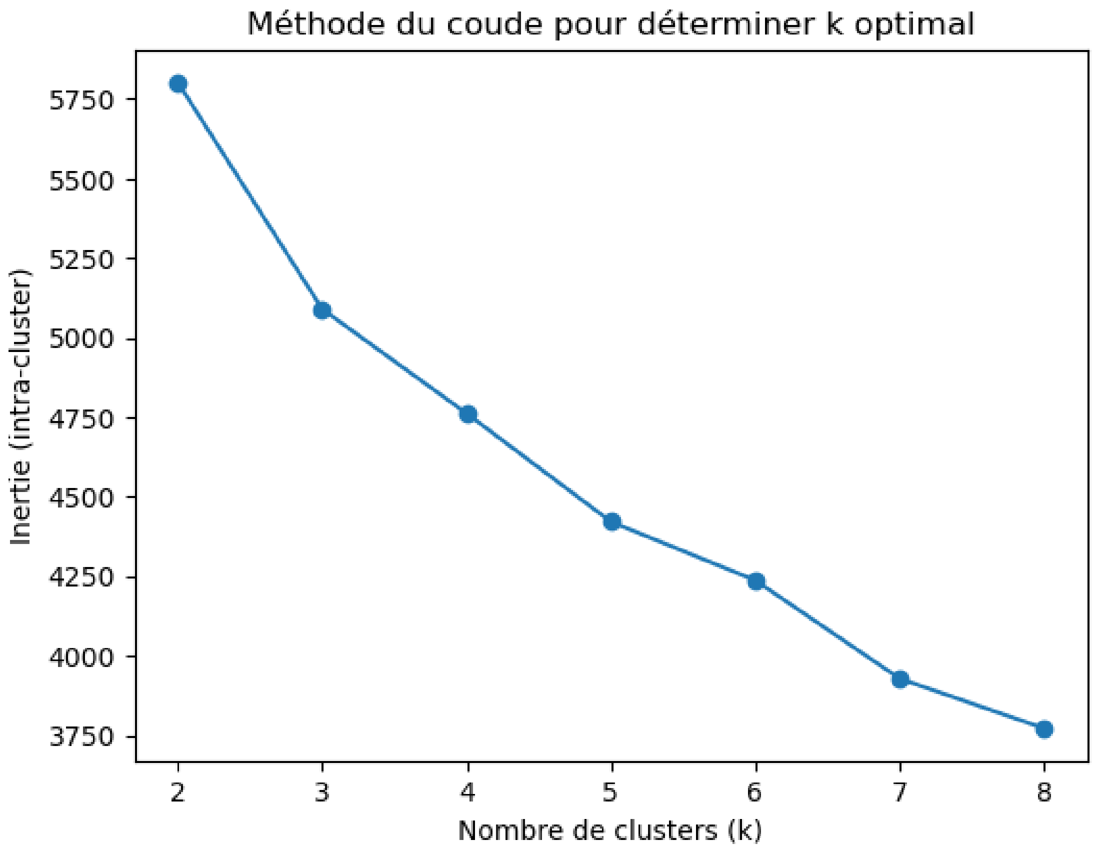


[165]:
```
inertias = []
n_clust=3
kmeans = KMeans(n_clusters=n_clust, random_state=127)
kmeans.fit(df_scaled)
clusters = kmeans.fit_predict(df_scaled)
display(pd.Series.value_counts(clusters).sort_index())
df_work["cluKM31"] = clusters

contingency_table = pd.crosstab(df_work['is_genuine'], df_work['cluKM31'])
fig=plt.figure()
sb.heatmap(contingency_table, annot=True, fmt='d', cmap='viridis')
plt.title("Correspondance entre les sets de clusters")
plt.show()
```

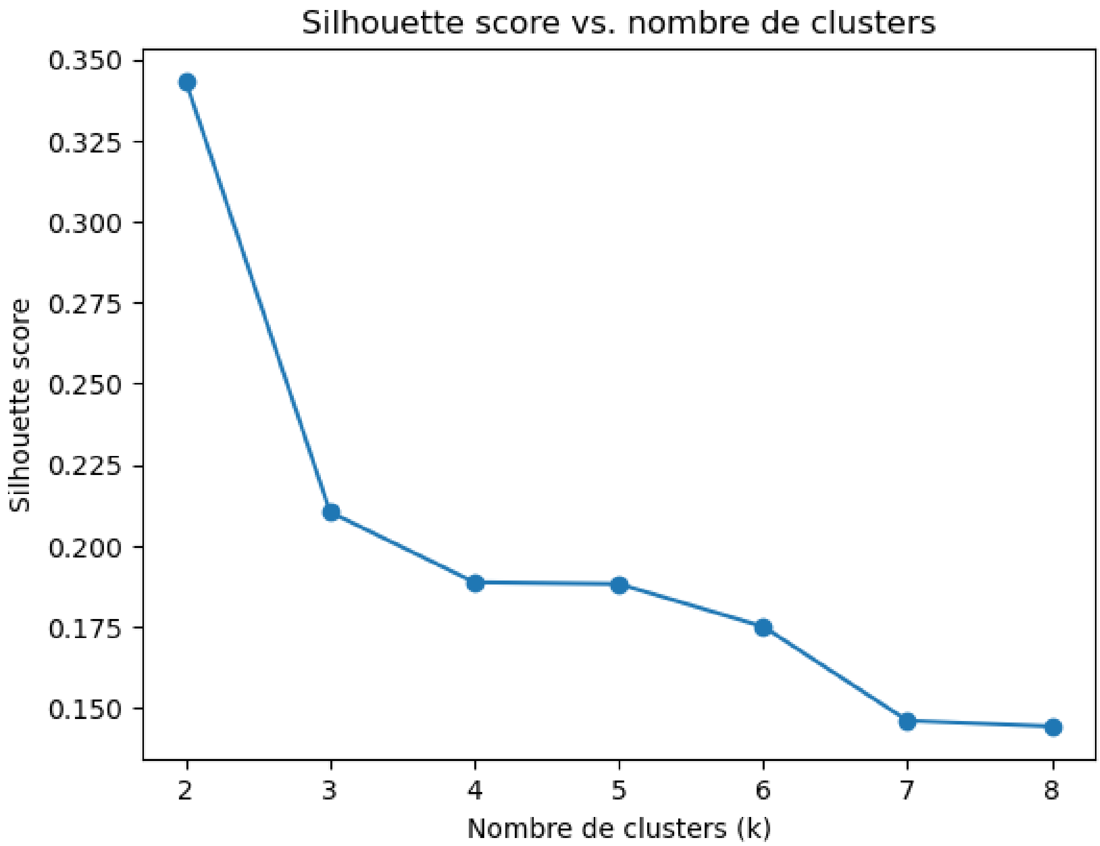
```
0    505
1    507
2    488
Name: count, dtype: int64
```




```
[166]: inertias = []
k_range = range(2, 9)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=808)
    kmeans.fit(df_scaled)
    inertias.append(kmeans.inertia_)

plt.plot(k_range, inertias, marker='o')
plt.xlabel('Nombre de clusters (k)')
plt.ylabel('Inertie (intra-cluster)')
plt.title('Méthode du coude pour déterminer k optimal')
plt.show()
```

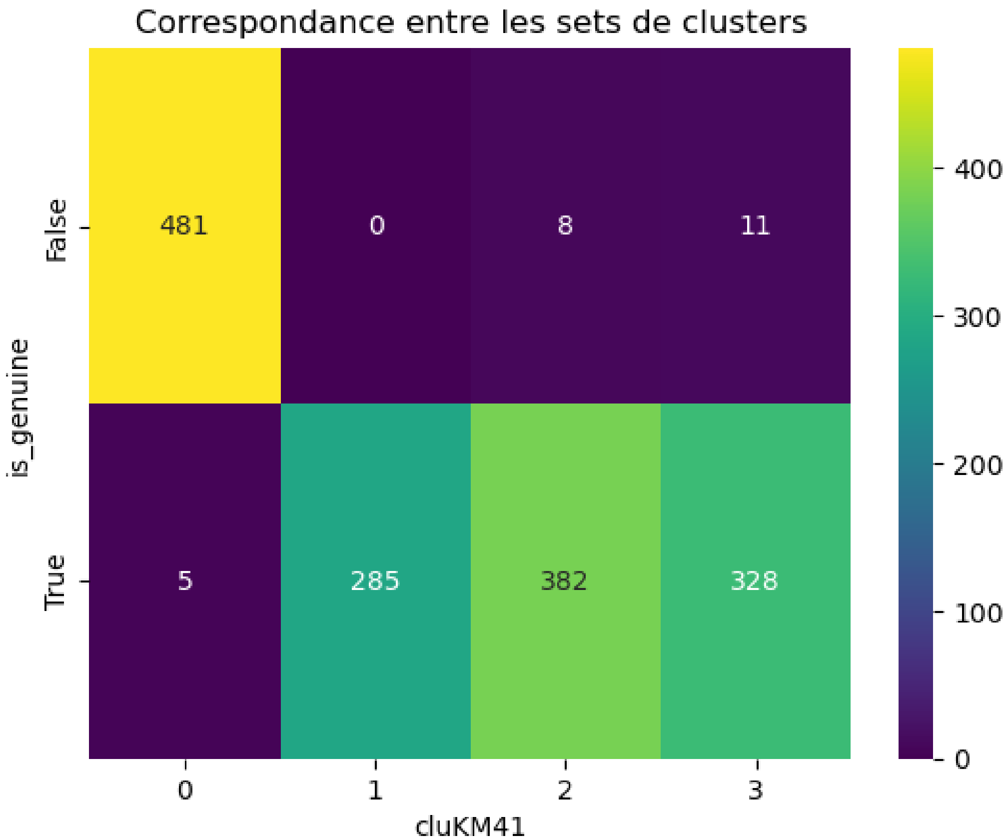


[167]:
```
silhouette_scores = []

for k in range(2, 9):
    kmeans = KMeans(n_clusters=k, random_state=808)
    labels = kmeans.fit_predict(df_scaled)
    score = silhouette_score(df_scaled, labels)
    silhouette_scores.append(score)

plt.plot(range(2, 9), silhouette_scores, marker='o')
plt.xlabel('Nombre de clusters (k)')
plt.ylabel('Silhouette score')
plt.title('Silhouette score vs. nombre de clusters')
plt.show()
```



[168]:
```
inertias = []
n_clust=4
kmeans = KMeans(n_clusters=n_clust, random_state=808)
kmeans.fit(df_scaled)
clusters = kmeans.fit_predict(df_scaled)
display(pd.Series.value_counts(clusters).sort_index())
df_work["cluKM41"] = clusters

contingency_table = pd.crosstab(df_work['is_genuine'], df_work['cluKM41'])
fig=plt.figure()
sb.heatmap(contingency_table, annot=True, fmt='d', cmap='viridis')
plt.title("Correspondance entre les sets de clusters")
plt.show()
```

```
0    486
1    285
2    390
3    339
Name: count, dtype: int64
```



K-Means pose un problème théorique : son caractère aléatoire fait qu'il n'est pas robuste au-delà de 2 clusters pour notre jeu de données. Il n'y a a priori aucune raison de considérer qu'il faille seulement 2 clusters - en particulier pour les faux billets, les contrefaçons possibles pouvant se manifester par une plus grande variabilité des cotes que par rapport aux vrais billets - à moins qu'on suppose que chaque groupe (vrai/faux) de billets oscille autour d'un comportement moyen, ce qui n'a a priori aucune raison d'être vrai pour notre jeu de données. Cette intuition se trouve être démentie par nos itérations : il est plus facile de créer des clusters différents de vrais billets que de faux billets, cela semble contre-intuitif. Le fait de se limiter à 2 clusters est cohérent avec le fait que la variable prédite ne peut prendre que 2 valeurs, et permet de limiter les effets aléatoires de l'agorithme.

4.1.2 - K-Means sur échantillon scindé train-test

[169]:
```
df_work.drop(columns=["clu","cluKM1","cluKM2","cluKM31","cluKM41"],inplace=True)
display(df_work)
```

```
     is_genuine  diagonal  height_left  height_right  margin_low  margin_up  \
0          True    171.81       104.86        104.95        4.52       2.89
1          True    171.46       103.36        103.66        3.77       2.99
2          True    172.69       104.48        103.50        4.40       2.94
3          True    171.36       103.91        103.94        3.62       3.01
```

```
4          True    171.73       104.28       103.46       4.04       3.48
…          …       …            …            …            …          …
1495       False   171.75       104.38       104.17       4.42       3.09
1496       False   172.19       104.63       104.44       5.27       3.37
1497       False   171.80       104.01       104.12       5.51       3.36
1498       False   172.06       104.28       104.06       5.17       3.46
1499       False   171.47       104.15       103.82       4.63       3.37

      length
0     112.83
1     113.09
2     113.16
3     113.51
4     112.54
…      …
1495  111.28
1496  110.97
1497  111.95
1498  112.25
1499  112.07

[1500 rows x 7 columns]
```

[170]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
↪random_state=8)

# Normalisation des données
scaler = StandardScaler(with_std=True)
scaler.fit(X_train)

# Problème : on perd les index !
X_train_scaled = scaler.transform(X_train)

# On récupère les index
df_train_scaled = pd.DataFrame(X_train_scaled, index=X_train.index)
df_train_scaled.rename(mapper={i: "z_"+list(X.columns)[i] for i in␣
↪range(df_train_scaled.shape[1])}, axis=1, inplace=True)

k_means = KMeans(n_clusters=2, random_state = 808)
k_means.fit(df_train_scaled)
print("Coordonnées des centroïdes :\n",k_means.cluster_centers_.round(3))
print("\nValeurs moyennes des cotes billets :")
display(df_train_scaled.merge(df_work["is_genuine"], how='left',␣
↪left_index=True, right_index=True).groupby("is_genuine").mean().round(3))
```

```
Coordonnées des centroïdes :
 [[ 0.103 -0.272 -0.342 -0.555 -0.422  0.595]
 [-0.211  0.555  0.697  1.13   0.859 -1.212]]
```

```
Valeurs moyennes des cotes billets :

            z_diagonal  z_height_left  z_height_right  z_margin_low  \
is_genuine
False           -0.198          0.521           0.672         1.105
True             0.100         -0.263          -0.339        -0.557

            z_margin_up  z_length
is_genuine
False             0.849    -1.201
True             -0.428     0.605
```

On peut observer que l'entraînement n'est pas parfait : les centroïdes ne correspondent pas exactement aux moyennes des vrais billets d'une part et des faux-billets d'autre part. Mais l'algorithme fonctionne comme il le devrait, en regroupant d'abord les points en fonction de leurs corrdonnées, il faudrait ensuite étiqueter chaque cluster selon son *is_genuine* majoritaire.

[171]:
```
# fonction avec pour paramètres l1 et l2 2 listes donnant les coordonnées de 2␣
↪points
# et renvoie une fonction lambda
# qui teste si le point en argument (V) est plus proche de l1 (retourne alors␣
↪True) que de l2
def affunc(l1, l2) :
    return lambda V: (math.dist([V[col] for col in list(df_train_scaled.
↪columns)[:6]],l1) <
                      math.dist([V[col] for col in list(df_train_scaled.
↪columns)[:6]],l2))
```

Problème : K-Means ne fournit pas d'étiquetage, on sait seulement que notre échantillonnage est fidèle aux proportions de l'échantillon complet, et donc qu'il y a plus de vrais billets que de faux billets dans les sous-échantillons *train* et *test*. On peut reconstituer les étiquettes par ce biais.

[172]:
```
# Coordonnées des centroïdes
if list(k_means.labels_).count(0) > list(k_means.labels_).count(1):
    p1 = k_means.cluster_centers_[0]
    p0 = k_means.cluster_centers_[1]
else:
    p1 = k_means.cluster_centers_[1]
    p0 = k_means.cluster_centers_[0]
```

[173]:
```
# Vérification de la performance de l'entraînement
y_check = df_train_scaled.apply(affunc(p1,p0), axis=1)
y_check.name = "check"
cfm = confusion_matrix(y_true=y_train, y_pred=y_check, labels=[False, True])
print("Matrice de confusion de l'entraînement :\n",cfm)
print([False, True],"\n")
print(y_train.value_counts())
print(y_check.value_counts())
```

```
print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion de l'entraînement :
 [[390  12]
 [  5 793]]
[False, True]

is_genuine
True     798
False    402
Name: count, dtype: int64
check
True     805
False    395
Name: count, dtype: int64

Exactitude du modèle : 0.9858333333333333
Précision du modèle : 0.9873417721518988
Rappel du modèle : 0.9701492537313433
```

[174]:

```
# récupérer les index des prédictions erronées sur les données d'entraînement
df_check = pd.merge(y_train, y_check, how='inner', left_index=True,␣
 ↪right_index=True)
display(df_check.loc[df_check["is_genuine"]!=df_check["check"]])
```

	is_genuine	check
253	True	False
1103	False	True
728	True	False
1267	False	True
0	True	False
1383	False	True
1104	False	True
1073	False	True
1025	False	True
1083	False	True
1160	False	True
1412	False	True
1482	False	True
1122	False	True
626	True	False
743	True	False
1407	False	True

[175]:
```python
# On teste pour chaque individu test s'il est plus proche du centroïde "vrai"␣
↪que du centroïde "faux", obtenus à l'issue de l'entraînement
# mais il faut normaliser les données, sur la base des données d'entraînement

# Problème : on perd les index !
X_test_scaled = scaler.transform(X_test)

# On récupère les index
df_test_scaled = pd.DataFrame(X_test_scaled, index=X_test.index)
df_test_scaled.rename(mapper={i: "z_"+list(X.columns)[i] for i in␣
↪range(df_test_scaled.shape[1])}, axis=1, inplace=True)

y_pred = df_test_scaled.apply(affunc(p1,p0), axis=1)
y_pred.name = "pred"
display(y_pred)
```

```
1237    False
437      True
51       True
579      True
238      True
         …
916      True
1337    False
359      True
1438    False
1298    False
Name: pred, Length: 300, dtype: bool
```

[176]:
```python
cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")
print(y_test.value_counts())
print(y_pred.value_counts())
print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion :
 [[ 96   2]
 [  3 199]]
[False, True]

is_genuine
True     202
False     98
Name: count, dtype: int64
pred
```

```
True     201
False     99
Name: count, dtype: int64

Exactitude du modèle : 0.9833333333333333
Précision du modèle : 0.9696969696969697
Rappel du modèle : 0.9795918367346939
```

[177]:

```
# récupérer les index des prédictions erronnées
df_res = pd.merge(y_test, y_pred, how='inner', left_index=True,␣
 ↪right_index=True)
display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
      is_genuine   pred
946         True  False
341         True  False
1081       False   True
580         True  False
1362       False   True
```

4.1.3 - Validation croisée du modèle K-Means

Il suffit de reprendre les étapes précédentes en divisant l'échantillon complet en folds de taille égale, chaque fold étant testé sur la base des entraînements par les autres folds, puis de calculer les métriques et enfin de rassembler les résultats.

[178]:

```
l_score = []
l_exa = []
l_pre = []
l_rap = []
S_pred = pd.Series(name="pred")

for train_ids, test_ids in kf.split(X):

    # Répartir les individus dans les sous-échantillons train/test
    X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
    y_test = y.iloc[test_ids]

    # Entraîner K-Means sur X_train

    # Normalisation des données
    scaler = StandardScaler(with_std=True)
    scaler.fit(X_train)

    # Problème : on perd les index !
    X_train_scaled = scaler.transform(X_train)

    # On récupère les index
    df_train_scaled = pd.DataFrame(X_train_scaled, index=X_train.index)
```

```
    df_train_scaled.rename(mapper={i: "z_"+list(X.columns)[i] for i in␣
 ↪range(df_train_scaled.shape[1])}, axis=1, inplace=True)

    kmeans = KMeans(n_clusters=2, random_state=42).fit(df_train_scaled)

    # Récupérer les centroïdes
    if list(k_means.labels_).count(0) > list(k_means.labels_).count(1):
        p1 = k_means.cluster_centers_[0]
        p0 = k_means.cluster_centers_[1]
    else:
        p1 = k_means.cluster_centers_[1]
        p0 = k_means.cluster_centers_[0]

    # Normaliser les données de test à partir des données d'entraînement
    # /!\ Source de mauvaises prédictions
    # Problème : on perd les index !
    X_test_scaled = scaler.transform(X_test)

    # On récupère les index
    df_test_scaled = pd.DataFrame(X_test_scaled, index=X_test.index)
    df_test_scaled.rename(mapper={i: "z_"+list(X.columns)[i] for i in␣
 ↪range(df_test_scaled.shape[1])}, axis=1, inplace=True)

    # Faire les prédictions
    y_pred = df_test_scaled.apply(affunc(p1,p0), axis=1)
    y_pred.name = "pred"
    if len(S_pred) == 0:
        S_pred = y_pred
    else:
        S_pred = pd.concat([S_pred, y_pred], axis=0)

    # Calculer les scores
    cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False, True])
    pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
    rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
    fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
    l_score.append(fb_score)
    l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
    l_pre.append(pre)
    l_rap.append(rap)
```

[179]:
```
df_res = pd.merge(df_work["is_genuine"], S_pred, how='left', left_index=True,␣
 ↪right_index=True)

cfm = confusion_matrix(y_true=df_res["is_genuine"], y_pred=df_res["pred"],␣
 ↪labels=[False, True])
print("Matrice de confusion :\n",cfm)
```

```python
print([False, True],"\n")

display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
Matrice de confusion :
 [[488  12]
 [  9 991]]
[False, True]


      is_genuine   pred
0           True  False
253         True  False
341         True  False
580         True  False
626         True  False
724         True  False
728         True  False
743         True  False
946         True  False
1025       False   True
1073       False   True
1081       False   True
1083       False   True
1103       False   True
1122       False   True
1160       False   True
1267       False   True
1362       False   True
1407       False   True
1412       False   True
1482       False   True
```

[180]:

```python
print(f"Fb-score moyen : {round(np.mean(l_score),4)}")
print(f"\nScore moyen exactitude : {round(np.mean(l_exa),4)}")
print(f"Écart-type exactitude : {round(np.std(l_exa,ddof=1),4)}")
print(f"Score min exactitude : {round(np.min(l_exa),4)}")
print(f"\nScore moyen précision : {round(np.mean(l_pre),4)}")
print(f"Écart-type précision : {round(np.std(l_pre,ddof=1),4)}")
print(f"Score min précision : {round(np.min(l_pre),4)}")
print(f"\nScore moyen rappel : {round(np.mean(l_rap),4)}")
print(f"Écart-type rappel : {round(np.std(l_rap,ddof=1),4)}")
print(f"Score min rappel : {round(np.min(l_rap),4)}")

dicr = {'nom': "K-Means", 'param_value': 2, 'Fb-score': round(np.
 ↪mean(l_score),4),
        'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.min(l_exa),4),
        'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.min(l_pre),4),
```

```
        'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.min(l_rap),4)}
modrows.append(dicr)
```

```
Fb-score moyen : 0.9763

Score moyen exactitude : 0.986
Écart-type exactitude : 0.0049
Score min exactitude : 0.98

Score moyen précision : 0.9819
Écart-type précision : 0.0176
Score min précision : 0.9556

Score moyen rappel : 0.9761
Écart-type rappel : 0.0098
Score min rappel : 0.9652
```

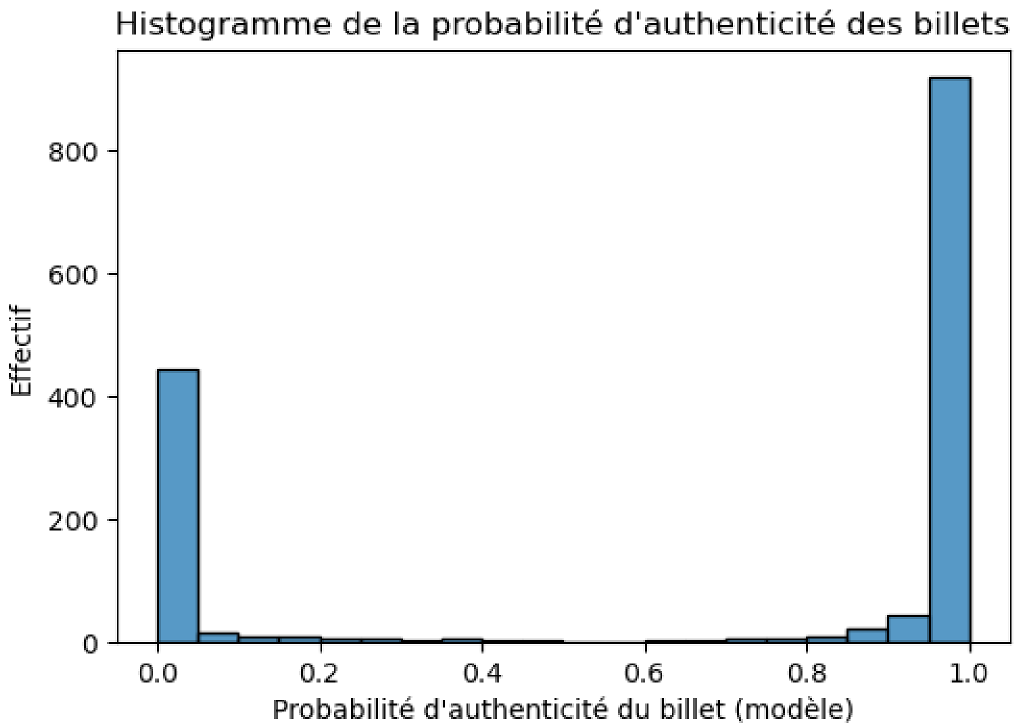
4.2 - Modèle de régression logistique

Avec ce modèle d'apprentissage supervisé, on se base sur une potentielle relation (ce qu'on appelle une régression) entre la variable prédite *is_genuine* et les données géométriques (prédicteurs) via une fonction particulière, dite fonction logistique (ou fonction logit selon les formulations), qui essaie de quantifier de façon probabiliste l'appartenance de la variable prédite à telle catégorie (0=False=positif ou bien 1=True=négatif).

4.2.1 - Modèle de régression logistique sur l'échantillon complet

[181]:
```
clf = LogisticRegression().fit(X, y)
```

[182]:
```
y_hat_proba = clf.predict_proba(X)[:,1]

plt.figure(figsize=(6, 4))
sb.histplot(y_hat_proba, binwidth=0.05)
plt.xlabel("Probabilité d'authenticité du billet (modèle)")
plt.ylabel("Effectif")
plt.title("Histogramme de la probabilité d'authenticité des billets")
plt.show()
```



[183]:
```
y_pred = pd.Series(clf.predict(X), name="pred")
cfm = confusion_matrix(y, y_pred, labels=[False, True])

print("Matrice de confusion sur échantillon complet auto-entraîné :\n",cfm)
print([False, True],"\n")
print(y.value_counts())
print(y_pred.value_counts())

print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion sur échantillon complet auto-entraîné :
 [[491   9]
 [  5 995]]
[False, True]

is_genuine
True     1000
False     500
Name: count, dtype: int64
pred
True     1004
False     496
```

```
Name: count, dtype: int64

Exactitude du modèle : 0.9906666666666667
Précision du modèle : 0.9899193548387096
Rappel du modèle : 0.982
```

[184]:
```
df_res = pd.merge(df_work["is_genuine"], y_pred, how='left', left_index=True,␣
↪right_index=True)
y_proba = pd.Series(clf.predict_proba(X)[:,1], name="proba").round(3)
df_res = df_res.merge(y_proba, how='left', left_index=True, right_index=True)
display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

	is_genuine	pred	proba
0	True	False	0.390
591	True	False	0.380
626	True	False	0.494
669	True	False	0.406
728	True	False	0.145
1025	False	True	0.815
1073	False	True	0.699
1083	False	True	0.766
1103	False	True	0.807
1122	False	True	0.997
1160	False	True	0.735
1190	False	True	0.814
1407	False	True	0.811
1412	False	True	0.915

4.2.2 - Modèle de régression logistique sur échantillon scindé train-test

[185]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
↪random_state=8)
clf = LogisticRegression().fit(X_train, y_train)

# Performance de l'entraînement
y_check = pd.Series(clf.predict(X_train), name="check")
cfm = confusion_matrix(y_train, y_check, labels=[False, True])

print("Matrice de confusion sur échantillon d'entraînement :\n",cfm)
print([False, True],"\n")
print(y_train.value_counts())
print(y_check.value_counts())

print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion sur échantillon d'entraînement :
 [[393   9]
```

```
 [  4 794]]
[False, True]

is_genuine
True     798
False    402
Name: count, dtype: int64
check
True     803
False    397
Name: count, dtype: int64

Exactitude du modèle : 0.9891666666666666
Précision du modèle : 0.9899244332493703
Rappel du modèle : 0.9776119402985075
```

[186]:

```python
# Performance dy test
y_pred = pd.Series(clf.predict(X_test), name="pred")
cfm = confusion_matrix(y_test, y_pred, labels=[False, True])

print("Matrice de confusion sur échantillon de test :\n",cfm)
print([False, True],"\n")
print(y_test.value_counts())
print(y_pred.value_counts())

print(f"\nExactitude du modèle : {(cfm[0][0]+cfm[1][1])/sum(sum(cfm))}")
print(f"Précision du modèle : {cfm[0][0]/(cfm[0][0]+cfm[1][0])}")
print(f"Rappel du modèle : {cfm[0][0]/(cfm[0][0]+cfm[0][1])}")
```

```
Matrice de confusion sur échantillon de test :
 [[ 97   1]
 [  0 202]]
[False, True]

is_genuine
True     202
False     98
Name: count, dtype: int64
pred
True     203
False     97
Name: count, dtype: int64

Exactitude du modèle : 0.9966666666666667
Précision du modèle : 1.0
Rappel du modèle : 0.9897959183673469
```

4.2.3 - Validation croisée du modèle de régression logistique

[187]:
```python
l_score = []
l_exa = []
l_pre = []
l_rap = []
df_pred = pd.DataFrame(columns=["pred","proba"])

for train_ids, test_ids in kf.split(X):

    # Répartir les individus dans les sous-échantillons train/test
    X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
    y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

    # Entraîner la régression sur (X_train,y_train)
    clf = LogisticRegression().fit(X_train, y_train)

    # Faire les prédictions (et conserver les probabilités d'authenticité)
    y_pred = pd.DataFrame(data={"pred": clf.predict(X_test),\
                                "proba": (clf.predict_proba(X_test)[:,1]).
 ↪round(3)},\
                          index=test_ids)

    if len(df_pred) == 0:
        df_pred = y_pred
    else:
        df_pred = pd.concat([df_pred, y_pred], axis=0)

    # Calculer les scores
    cfm = confusion_matrix(y_true=y_test, y_pred=y_pred["pred"], labels=[False,␣
 ↪True])
    pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
    rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
    fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
    l_score.append(fb_score)
    l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
    l_pre.append(pre)
    l_rap.append(rap)
```

[188]:
```python
print(f"Fb-score moyen : {round(np.mean(l_score),4)}")
print(f"\nScore moyen exactitude : {round(np.mean(l_exa),4)}")
print(f"Écart-type exactitude : {round(np.std(l_exa,ddof=1),4)}")
print(f"Score min exactitude : {round(np.min(l_exa),4)}")
print(f"\nScore moyen précision : {round(np.mean(l_pre),4)}")
print(f"Écart-type précision : {round(np.std(l_pre,ddof=1),4)}")
print(f"Score min précision : {round(np.min(l_pre),4)}")
print(f"\nScore moyen rappel : {round(np.mean(l_rap),4)}")
print(f"Écart-type rappel : {round(np.std(l_rap,ddof=1),4)}")
print(f"Score min rappel : {round(np.min(l_rap),4)}")
```

```
Fb-score moyen : 0.9807

Score moyen exactitude : 0.9907
Écart-type exactitude : 0.0028
Score min exactitude : 0.9867

Score moyen précision : 0.9914
Écart-type précision : 0.0118
Score min précision : 0.9775

Score moyen rappel : 0.9802
Écart-type rappel : 0.0059
Score min rappel : 0.9727
```

[189]:
```
df_res = pd.merge(df_work["is_genuine"], df_pred, how='left', left_index=True,␣
 ↪right_index=True)
display(df_res[["is_genuine","pred"]].groupby(["is_genuine","pred"]).
 ↪value_counts())
display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
is_genuine  pred
False       False    490
            True      10
True        False      4
            True     996
Name: count, dtype: int64
```

	is_genuine	pred	proba
0	True	False	0.340
591	True	False	0.398
669	True	False	0.408
728	True	False	0.136
1025	False	True	0.842
1073	False	True	0.692
1083	False	True	0.775
1103	False	True	0.800
1122	False	True	0.997
1160	False	True	0.761
1190	False	True	0.848
1325	False	True	0.581
1407	False	True	0.789
1412	False	True	0.917

[190]:
```
# Variante en faisant varier le seuil de décision (par défaut psd = 0.5)
# On va stocker sous forme de lignes d'un DataFrame les indicateurs de␣
 ↪performance de la régression
# 1 ligne = 1 valeur de psd
krows=[]
```

```
psd_val=np.linspace(0.5, 0.95, 46)

for psd in psd_val:

    l_score = []
    l_exa = []
    l_pre = []
    l_rap = []
    df_pred = pd.DataFrame(columns=["pred","proba"])

    for train_ids, test_ids in kf.split(X):

        # Répartir les individus dans les sous-échantillons train/test
        X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
        y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

        # Entraîner la régression sur (X_train,y_train)
        clf = LogisticRegression().fit(X_train, y_train)

        # Faire les prédictions (et conserver les probabilités d'authenticité)
        y_pred = pd.DataFrame(data={"pred": clf.predict(X_test),\
                                    "proba": (clf.predict_proba(X_test)[:,1]).
 ↪round(3)},\
                              index=test_ids)

        # Appliquer le seuil de décision
        y_pred["pred"] = (y_pred["proba"] > psd)

        if len(df_pred) == 0:
            df_pred = y_pred
        else:
            df_pred = pd.concat([df_pred, y_pred], axis=0)

        # Calculer les scores
        cfm = confusion_matrix(y_true=y_test, y_pred=y_pred["pred"],␣
 ↪labels=[False, True])
        pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
        rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
        fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
        l_score.append(fb_score)
        l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
        l_pre.append(pre)
        l_rap.append(rap)

    # On stocke les résultats pour l'itération psd courante
    dicr = {'Fb-score': round(np.mean(l_score),4),
```

```
            'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.
 ↪min(l_exa),4),
            'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.
 ↪min(l_pre),4),
            'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.
 ↪min(l_rap),4)}
    krows.append(dicr)

df_psdperf = pd.DataFrame(krows, index=list(psd_val))

for col in list(df_psdperf.columns):
    print("Meilleurs modèles selon",col)
    display(df_psdperf.sort_values(col, ascending=False).head())

hyper_psd = list(df_psdperf.sort_values("Fb-score", ascending=False).index)[0]
print("Valeur de psd retenue :", round(hyper_psd,2))
```

```
Meilleurs modèles selon Fb-score
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
0.85	0.9942	0.9833	0.9767	0.9559	0.9462	0.9962	0.9898
0.86	0.9936	0.9793	0.9733	0.9449	0.9362	0.9962	0.9898
0.87	0.9935	0.9787	0.9733	0.9429	0.9362	0.9962	0.9898
0.88	0.9932	0.9767	0.9733	0.9370	0.9167	0.9962	0.9898
0.92	0.9931	0.9633	0.9500	0.9015	0.8544	0.9983	0.9913

```
Meilleurs modèles selon avg_exa
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
0.50	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727
0.59	0.9824	0.9907	0.9867	0.9896	0.9775	0.9820	0.9775
0.64	0.9824	0.9907	0.9867	0.9896	0.9775	0.9820	0.9775
0.63	0.9824	0.9907	0.9867	0.9896	0.9775	0.9820	0.9775
0.62	0.9824	0.9907	0.9867	0.9896	0.9775	0.9820	0.9775

```
Meilleurs modèles selon min_exa
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
0.50	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727
0.61	0.9824	0.9907	0.9867	0.9896	0.9775	0.9820	0.9775
0.71	0.9838	0.9887	0.9867	0.9822	0.9667	0.9839	0.9775
0.70	0.9838	0.9893	0.9867	0.9839	0.9667	0.9839	0.9775
0.69	0.9822	0.9893	0.9867	0.9860	0.9775	0.9820	0.9775

```
Meilleurs modèles selon avg_pre
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
0.50	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727
0.52	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727
0.53	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727
0.51	0.9807	0.9907	0.9867	0.9914	0.9775	0.9802	0.9727

```
0.59    0.9824   0.9907   0.9867   0.9896   0.9775   0.9820   0.9775

Meilleurs modèles selon min_pre

      Fb-score  avg_exa  min_exa  avg_pre  min_pre  avg_rap  min_rap
0.50    0.9807   0.9907   0.9867   0.9914   0.9775   0.9802   0.9727
0.61    0.9824   0.9907   0.9867   0.9896   0.9775   0.9820   0.9775
0.51    0.9807   0.9907   0.9867   0.9914   0.9775   0.9802   0.9727
0.69    0.9822   0.9893   0.9867   0.9860   0.9775   0.9820   0.9775
0.68    0.9823   0.9900   0.9867   0.9878   0.9775   0.9820   0.9775

Meilleurs modèles selon avg_rap

      Fb-score  avg_exa  min_exa  avg_pre  min_pre  avg_rap  min_rap
0.95    0.9898   0.9393   0.9267   0.8466   0.8073   0.9983   0.9913
0.94    0.9911   0.9487   0.9333   0.8672   0.8148   0.9983   0.9913
0.93    0.9925   0.9587   0.9467   0.8908   0.8462   0.9983   0.9913
0.92    0.9931   0.9633   0.9500   0.9015   0.8544   0.9983   0.9913
0.88    0.9932   0.9767   0.9733   0.9370   0.9167   0.9962   0.9898

Meilleurs modèles selon min_rap

      Fb-score  avg_exa  min_exa  avg_pre  min_pre  avg_rap  min_rap
0.95    0.9898   0.9393   0.9267   0.8466   0.8073   0.9983   0.9913
0.94    0.9911   0.9487   0.9333   0.8672   0.8148   0.9983   0.9913
0.93    0.9925   0.9587   0.9467   0.8908   0.8462   0.9983   0.9913
0.92    0.9931   0.9633   0.9500   0.9015   0.8544   0.9983   0.9913
0.88    0.9932   0.9767   0.9733   0.9370   0.9167   0.9962   0.9898

Valeur de psd retenue : 0.85
```

[191]:

```
l_score = []
l_exa = []
l_pre = []
l_rap = []
df_pred = pd.DataFrame(columns=["pred","proba"])

for train_ids, test_ids in kf.split(X):

    # Répartir les individus dans les sous-échantillons train/test
    X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
    y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

    # Entraîner la régression sur (X_train,y_train)
    clf = LogisticRegression().fit(X_train, y_train)

    # Faire les prédictions (et conserver les probabilités d'authenticité)
    y_pred = pd.DataFrame(data={"pred": clf.predict(X_test),\
                                "proba": (clf.predict_proba(X_test)[:,1]).
 ↪round(3)},\
                          index=test_ids)
```

```python
    # Appliquer le seuil de décision
    y_pred["pred"] = (y_pred["proba"] > hyper_psd)

    if len(df_pred) == 0:
        df_pred = y_pred
    else:
        df_pred = pd.concat([df_pred, y_pred], axis=0)

    # Calculer les scores
    cfm = confusion_matrix(y_true=y_test, y_pred=y_pred["pred"], labels=[False,␣
↪True])
    pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
    rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
    fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
    l_score.append(fb_score)
    l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
    l_pre.append(pre)
    l_rap.append(rap)
```

[192]:

```python
print(f"Fb-score moyen : {round(np.mean(l_score),4)}")
print(f"\nScore moyen exactitude : {round(np.mean(l_exa),4)}")
print(f"Écart-type exactitude : {round(np.std(l_exa,ddof=1),4)}")
print(f"Score min exactitude : {round(np.min(l_exa),4)}")
print(f"\nScore moyen précision : {round(np.mean(l_pre),4)}")
print(f"Écart-type précision : {round(np.std(l_pre,ddof=1),4)}")
print(f"Score min précision : {round(np.min(l_pre),4)}")
print(f"\nScore moyen rappel : {round(np.mean(l_rap),4)}")
print(f"Écart-type rappel : {round(np.std(l_rap,ddof=1),4)}")
print(f"Score min rappel : {round(np.min(l_rap),4)}")

dicr = {'nom': "LogReg", 'param_value': hyper_psd, 'Fb-score': round(np.
↪mean(l_score),4),
        'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.min(l_exa),4),
        'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.min(l_pre),4),
        'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.min(l_rap),4)}
modrows.append(dicr)
```

```
Fb-score moyen : 0.9942

Score moyen exactitude : 0.9833
Écart-type exactitude : 0.0053
Score min exactitude : 0.9767

Score moyen précision : 0.9559
Écart-type précision : 0.0096
Score min précision : 0.9462
```

```
Score moyen rappel : 0.9962
Écart-type rappel : 0.0052
Score min rappel : 0.9898
```

[193]:
```
df_res = pd.merge(df_work["is_genuine"], df_pred, how='left', left_index=True,␣
↪right_index=True)

cfm = confusion_matrix(y_true=df_res["is_genuine"], y_pred=df_res["pred"],␣
↪labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")

display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
Matrice de confusion :
 [[498   2]
 [ 23 977]]
[False, True]
```

	is_genuine	pred	proba
0	True	False	0.340
6	True	False	0.768
38	True	False	0.712
167	True	False	0.837
341	True	False	0.696
357	True	False	0.848
392	True	False	0.759
441	True	False	0.813
591	True	False	0.398
623	True	False	0.734
626	True	False	0.533
648	True	False	0.814
665	True	False	0.832
669	True	False	0.408
693	True	False	0.823
724	True	False	0.643
728	True	False	0.136
743	True	False	0.789
931	True	False	0.828
946	True	False	0.775
951	True	False	0.707
975	True	False	0.685
985	True	False	0.832
1122	False	True	0.997
1412	False	True	0.917

4.3 - Modèle des k plus proches voisins (KNN)

L'algorithme KNN est une méthode d'apprentissage supervisée, basée sur les distances entre individus. Une normalisation des données est nécessaire de façon à ce que les variables mesurant les marges ne soient pas masquées par les autres cotes. En guise de test, et pour mesurer la performance maximale possible de ce modèle, on applique K-Means sur l'échantillon complet. Il nous faudra également déterminer l'hyper-paramètre k du modèle, à savoir : pour un point donné dans l'espace des prédicteurs centrés-réduits, de combien de points voisins (l'hyper-paramètre k) avons-nous besoin pour réaliser une prédiction pertinente. Pour ce modèle, on va passer directement à la validation croisée (en cherchant le k optimal) car faire des prédictions sur l'échantillon complet avec pour données d'entraînement ce même échantillon complet n'a pas vraiment de sens, surtout si nous avons la valeur d'un hyper-paramètre à déterminer.

[194]:
```
# On va stocker sous forme de lignes d'un DataFrame les indicateurs de␣
↪performance de KNN
# 1 ligne = 1 valeur de K
krows=[]
k_val=range(1,21)

for k_voisins in k_val:

    l_score = []
    l_exa = []
    l_pre = []
    l_rap = []
    S_pred = pd.Series(name="pred")

    for train_ids, test_ids in kf.split(X):

        # Répartir les individus dans les sous-échantillons train/test
        X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
        y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

        # Entraîner KNN sur X_train
        # Normalisation des données
        scaler = StandardScaler(with_std=True)
        scaler.fit(X_train)

        # Problème : on perd les index !
        X_train_scaled = scaler.transform(X_train)

        # On récupère les index
        df_train_scaled = pd.DataFrame(X_train_scaled, index=X_train.index,␣
↪columns=list(X.columns))

        # Instanciation du modèle pour un k donné
        knn = sk_n.KNeighborsClassifier(n_neighbors=k_voisins).
↪fit(df_train_scaled, y_train)
```

```
        # Normaliser les données de test à partir des données d'entraînement
        # /!\ Source de mauvaises prédictions
        # Problème : on perd les index !
        X_test_scaled = scaler.transform(X_test)

        # On récupère les index
        df_test_scaled = pd.DataFrame(X_test_scaled, index=X_test.index,␣
 ↪columns=list(X.columns))

        # Faire les prédictions
        y_pred = pd.Series(knn.predict(df_test_scaled), index=y_test.index,␣
 ↪name="pred")
        if len(S_pred) == 0:
            S_pred = y_pred
        else:
            S_pred = pd.concat([S_pred, y_pred], axis=0)

        # Calculer les scores
        cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False,␣
 ↪True])
        pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
        rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
        fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
        l_score.append(fb_score)
        l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
        l_pre.append(pre)
        l_rap.append(rap)

    # On stocke les résultats pour l'itération k courante
    dicr = {'Fb-score': round(np.mean(l_score),4),
            'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.
 ↪min(l_exa),4),
            'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.
 ↪min(l_pre),4),
            'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.
 ↪min(l_rap),4)}
    krows.append(dicr)

df_kperf = pd.DataFrame(krows, index=list(k_val))

for col in list(df_kperf.columns):
    print("Meilleurs modèles selon",col)
    display(df_kperf.sort_values(col, ascending=False).head())

hyper_kv = list(df_kperf.sort_values("Fb-score", ascending=False).index)[0]
print("Valeur de k retenue :", hyper_kv)
```

Meilleurs modèles selon Fb-score

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
2	0.9851	0.9840	0.9700	0.9656	0.9158	0.9861	0.9796
14	0.9808	0.9913	0.9900	0.9934	0.9775	0.9802	0.9727
6	0.9800	0.9887	0.9800	0.9851	0.9556	0.9798	0.9773
12	0.9790	0.9900	0.9867	0.9913	0.9667	0.9784	0.9636
8	0.9789	0.9893	0.9867	0.9893	0.9667	0.9784	0.9636

Meilleurs modèles selon avg_exa

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
14	0.9808	0.9913	0.9900	0.9934	0.9775	0.9802	0.9727
18	0.9787	0.9907	0.9833	0.9933	0.9663	0.9779	0.9727
12	0.9790	0.9900	0.9867	0.9913	0.9667	0.9784	0.9636
17	0.9754	0.9900	0.9867	0.9955	0.9773	0.9744	0.9636
16	0.9774	0.9900	0.9867	0.9934	0.9775	0.9767	0.9636

Meilleurs modèles selon min_exa

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
14	0.9808	0.9913	0.9900	0.9934	0.9775	0.9802	0.9727
11	0.9753	0.9893	0.9867	0.9934	0.9773	0.9744	0.9636
12	0.9790	0.9900	0.9867	0.9913	0.9667	0.9784	0.9636
17	0.9754	0.9900	0.9867	0.9955	0.9773	0.9744	0.9636
16	0.9774	0.9900	0.9867	0.9934	0.9775	0.9767	0.9636

Meilleurs modèles selon avg_pre

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
17	0.9754	0.9900	0.9867	0.9955	0.9773	0.9744	0.9636
11	0.9753	0.9893	0.9867	0.9934	0.9773	0.9744	0.9636
16	0.9774	0.9900	0.9867	0.9934	0.9775	0.9767	0.9636
15	0.9753	0.9893	0.9867	0.9934	0.9773	0.9744	0.9636
9	0.9769	0.9900	0.9867	0.9934	0.9773	0.9761	0.9636

Meilleurs modèles selon min_pre

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
16	0.9774	0.9900	0.9867	0.9934	0.9775	0.9767	0.9636
14	0.9808	0.9913	0.9900	0.9934	0.9775	0.9802	0.9727
11	0.9753	0.9893	0.9867	0.9934	0.9773	0.9744	0.9636
13	0.9769	0.9900	0.9867	0.9934	0.9773	0.9761	0.9636
17	0.9754	0.9900	0.9867	0.9955	0.9773	0.9744	0.9636

Meilleurs modèles selon avg_rap

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
2	0.9851	0.9840	0.9700	0.9656	0.9158	0.9861	0.9796
14	0.9808	0.9913	0.9900	0.9934	0.9775	0.9802	0.9727
6	0.9800	0.9887	0.9800	0.9851	0.9556	0.9798	0.9773
8	0.9789	0.9893	0.9867	0.9893	0.9667	0.9784	0.9636
12	0.9790	0.9900	0.9867	0.9913	0.9667	0.9784	0.9636

```
Meilleurs modèles selon min_rap

    Fb-score  avg_exa  min_exa  avg_pre  min_pre  avg_rap  min_rap
2     0.9851   0.9840   0.9700   0.9656   0.9158   0.9861   0.9796
6     0.9800   0.9887   0.9800   0.9851   0.9556   0.9798   0.9773
18    0.9787   0.9907   0.9833   0.9933   0.9663   0.9779   0.9727
14    0.9808   0.9913   0.9900   0.9934   0.9775   0.9802   0.9727
11    0.9753   0.9893   0.9867   0.9934   0.9773   0.9744   0.9636

Valeur de k retenue : 2
```

Comme dit plus haut, on cherche le modèle qui minimise le Fb-score. Pour les k plus proches voisins, il s'agit du modèle k=2.

[195]:
```
l_score = []
l_exa = []
l_pre = []
l_rap = []
S_pred = pd.Series(name="pred")

for train_ids, test_ids in kf.split(X):

    # Répartir les individus dans les sous-échantillons train/test
    X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
    y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

    # Entraîner KNN sur X_train
    # Normalisation des données
    scaler = StandardScaler(with_std=True)
    scaler.fit(X_train)

    # Problème : on perd les index !
    X_train_scaled = scaler.transform(X_train)

    # On récupère les index
    df_train_scaled = pd.DataFrame(X_train_scaled, index=X_train.index,␣
 ↪columns=list(X.columns))

    # Instanciation du modèle pour un k donné
    knn = sk_n.KNeighborsClassifier(n_neighbors=hyper_kv).fit(df_train_scaled,␣
 ↪y_train)

    # Normaliser les données de test à partir des données d'entraînement
    # /!\ Source de mauvaises prédictions
    # Problème : on perd les index !
    X_test_scaled = scaler.transform(X_test)

    # On récupère les index
```

```
    df_test_scaled = pd.DataFrame(X_test_scaled, index=X_test.index,␣
↪columns=list(X.columns))

    # Faire les prédictions
    y_pred = pd.Series(knn.predict(df_test_scaled), index=y_test.index,␣
↪name="pred")
    if len(S_pred) == 0:
        S_pred = y_pred
    else:
        S_pred = pd.concat([S_pred, y_pred], axis=0)

    # Calculer les scores
    cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False, True])
    pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
    rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
    fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
    l_score.append(fb_score)
    l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
    l_pre.append(pre)
    l_rap.append(rap)
```

[196]:

```
print(f"Fb-score moyen : {round(np.mean(l_score),4)}")
print(f"\nScore moyen exactitude : {round(np.mean(l_exa),4)}")
print(f"Écart-type exactitude : {round(np.std(l_exa,ddof=1),4)}")
print(f"Score min exactitude : {round(np.min(l_exa),4)}")
print(f"\nScore moyen précision : {round(np.mean(l_pre),4)}")
print(f"Écart-type précision : {round(np.std(l_pre,ddof=1),4)}")
print(f"Score min précision : {round(np.min(l_pre),4)}")
print(f"\nScore moyen rappel : {round(np.mean(l_rap),4)}")
print(f"Écart-type rappel : {round(np.std(l_rap,ddof=1),4)}")
print(f"Score min rappel : {round(np.min(l_rap),4)}")

dicr = {'nom': "KNN", 'param_value': hyper_kv, 'Fb-score': round(np.
↪mean(l_score),4),
        'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.min(l_exa),4),
        'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.min(l_pre),4),
        'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.min(l_rap),4)}
modrows.append(dicr)
```

```
Fb-score moyen : 0.9851

Score moyen exactitude : 0.984
Écart-type exactitude : 0.0083
Score min exactitude : 0.97

Score moyen précision : 0.9656
Écart-type précision : 0.0283
```

```
Score min précision : 0.9158

Score moyen rappel : 0.9861
Écart-type rappel : 0.0048
Score min rappel : 0.9796
```

[197]:
```python
df_res = pd.merge(df_work["is_genuine"], S_pred, how='left', left_index=True,␣
 ↪right_index=True)

cfm = confusion_matrix(y_true=df_res["is_genuine"], y_pred=df_res["pred"],␣
 ↪labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")

display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
Matrice de confusion :
 [[493   7]
 [ 17 983]]
[False, True]
```

	is_genuine	pred
4	True	False
66	True	False
75	True	False
91	True	False
270	True	False
277	True	False
341	True	False
591	True	False
623	True	False
670	True	False
685	True	False
728	True	False
739	True	False
857	True	False
934	True	False
946	True	False
980	True	False
1025	False	True
1073	False	True
1083	False	True
1103	False	True
1122	False	True
1407	False	True
1412	False	True

4.4 - Modèle de forêt aléatoire

```
[198]: # On va stocker sous forme de lignes d'un DataFrame les indicateurs de␣
        ↪performance
       # 1 ligne = 1 valeur de nombre d'arbres (k_trees)
       krows=[]

       k_val = range(1,31)

       for k_trees in k_val:

           l_score = []
           l_exa = []
           l_pre = []
           l_rap = []
           S_pred = pd.Series(name="pred")

           for train_ids, test_ids in kf.split(X):

               # Répartir les individus dans les sous-échantillons train/test
               X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
               y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

               # Instanciation du modèle pour un k donné
               rndfor = sk_e.RandomForestClassifier(n_estimators=k_trees,␣
        ↪random_state=42).fit(X_train, y_train)

               # Faire les prédictions
               y_pred = pd.Series(rndfor.predict(X_test), index=y_test.index,␣
        ↪name="pred")
               if len(S_pred) == 0:
                   S_pred = y_pred
               else:
                   S_pred = pd.concat([S_pred, y_pred], axis=0)

               # Calculer les scores
               cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False,␣
        ↪True])
               pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
               rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
               fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
               l_score.append(fb_score)
               l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
               l_pre.append(pre)
               l_rap.append(rap)

           # On stocke les résultats pour l'itération k courante
           dicr = {'Fb-score': round(np.mean(l_score),4),
```

```
            'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.
↪min(l_exa),4),
            'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.
↪min(l_pre),4),
            'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.
↪min(l_rap),4)}
    krows.append(dicr)

df_kperf = pd.DataFrame(krows, index=list(k_val))

for col in list(df_kperf.columns):
    print("Meilleurs modèles selon",col)
    display(df_kperf.sort_values(col, ascending=False).head())

hyper_ka = list(df_kperf.sort_values("Fb-score", ascending=False).index)[0]
print("Valeur de k retenue :", hyper_ka)
```

Meilleurs modèles selon Fb-score

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
2	0.9863	0.9660	0.9467	0.9156	0.8898	0.9902	0.9796
4	0.9847	0.9807	0.9700	0.9570	0.9355	0.9862	0.9739
18	0.9842	0.9920	0.9900	0.9917	0.9775	0.9839	0.9775
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
22	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

Meilleurs modèles selon avg_exa

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
18	0.9842	0.9920	0.9900	0.9917	0.9775	0.9839	0.9775
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
28	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
24	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
23	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

Meilleurs modèles selon min_exa

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
17	0.9788	0.9907	0.9900	0.9934	0.9775	0.9781	0.9663
18	0.9842	0.9920	0.9900	0.9917	0.9775	0.9839	0.9775
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
29	0.9804	0.9907	0.9867	0.9917	0.9775	0.9799	0.9663
28	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

Meilleurs modèles selon avg_pre

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
17	0.9788	0.9907	0.9900	0.9934	0.9775	0.9781	0.9663
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
29	0.9804	0.9907	0.9867	0.9917	0.9775	0.9799	0.9663
28	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

```
27    0.9791   0.9900   0.9867   0.9917   0.9775   0.9785   0.9727
```

```
Meilleurs modèles selon min_pre
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
17	0.9788	0.9907	0.9900	0.9934	0.9775	0.9781	0.9663
29	0.9804	0.9907	0.9867	0.9917	0.9775	0.9799	0.9663
28	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
27	0.9791	0.9900	0.9867	0.9917	0.9775	0.9785	0.9727

```
Meilleurs modèles selon avg_rap
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
2	0.9863	0.9660	0.9467	0.9156	0.8898	0.9902	0.9796
4	0.9847	0.9807	0.9700	0.9570	0.9355	0.9862	0.9739
18	0.9842	0.9920	0.9900	0.9917	0.9775	0.9839	0.9775
6	0.9826	0.9833	0.9767	0.9665	0.9355	0.9834	0.9663
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

```
Meilleurs modèles selon min_rap
```

	Fb-score	avg_exa	min_exa	avg_pre	min_pre	avg_rap	min_rap
2	0.9863	0.9660	0.9467	0.9156	0.8898	0.9902	0.9796
18	0.9842	0.9920	0.9900	0.9917	0.9775	0.9839	0.9775
30	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739
4	0.9847	0.9807	0.9700	0.9570	0.9355	0.9862	0.9739
28	0.9826	0.9913	0.9867	0.9917	0.9775	0.9821	0.9739

```
Valeur de k retenue : 2
```

[199]:
```python
l_score = []
l_exa = []
l_pre = []
l_rap = []
S_pred = pd.Series(name="pred")

for train_ids, test_ids in kf.split(X):

    # Répartir les individus dans les sous-échantillons train/test
    X_train, X_test = X.iloc[train_ids], X.iloc[test_ids]
    y_train, y_test = y.iloc[train_ids], y.iloc[test_ids]

    rndfor = sk_e.RandomForestClassifier(n_estimators=hyper_ka,␣
 ↪random_state=42).fit(X_train, y_train)

    # Faire les prédictions
    y_pred = pd.Series(rndfor.predict(X_test), index=y_test.index, name="pred")
    if len(S_pred) == 0:
        S_pred = y_pred
    else:
```

```python
        S_pred = pd.concat([S_pred, y_pred], axis=0)

    # Calculer les scores
    cfm = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[False, True])
    pre = cfm[0][0]/(cfm[0][0]+cfm[1][0])
    rap = cfm[0][0]/(cfm[0][0]+cfm[0][1])
    fb_score = (1+rc)*(rap*pre)/(rc*pre+rap)
    l_score.append(fb_score)
    l_exa.append((cfm[0][0]+cfm[1][1])/sum(sum(cfm)))
    l_pre.append(pre)
    l_rap.append(rap)
```

[200]:
```python
print(f"Fb-score moyen : {round(np.mean(l_score),4)}")
print(f"\nScore moyen exactitude : {round(np.mean(l_exa),4)}")
print(f"Écart-type exactitude : {round(np.std(l_exa,ddof=1),4)}")
print(f"Score min exactitude : {round(np.min(l_exa),4)}")
print(f"\nScore moyen précision : {round(np.mean(l_pre),4)}")
print(f"Écart-type précision : {round(np.std(l_pre,ddof=1),4)}")
print(f"Score min précision : {round(np.min(l_pre),4)}")
print(f"\nScore moyen rappel : {round(np.mean(l_rap),4)}")
print(f"Écart-type rappel : {round(np.std(l_rap,ddof=1),4)}")
print(f"Score min rappel : {round(np.min(l_rap),4)}")

dicr = {'nom': "RandForest", 'param_value': hyper_ka, 'Fb-score': round(np.
 ↪mean(l_score),4),
        'avg_exa': round(np.mean(l_exa),4), 'min_exa': round(np.min(l_exa),4),
        'avg_pre': round(np.mean(l_pre),4), 'min_pre': round(np.min(l_pre),4),
        'avg_rap': round(np.mean(l_rap),4), 'min_rap': round(np.min(l_rap),4)}
modrows.append(dicr)
```

```
Fb-score moyen : 0.9863

Score moyen exactitude : 0.966
Écart-type exactitude : 0.0132
Score min exactitude : 0.9467

Score moyen précision : 0.9156
Écart-type précision : 0.0228
Score min précision : 0.8898

Score moyen rappel : 0.9902
Écart-type rappel : 0.0095
Score min rappel : 0.9796
```

[201]:
```python
df_res = pd.merge(df_work["is_genuine"], S_pred, how='left', left_index=True,␣
 ↪right_index=True)
```

```
cfm = confusion_matrix(y_true=df_res["is_genuine"], y_pred=df_res["pred"],
 ↪labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")

display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
Matrice de confusion :
 [[495   5]
 [ 46 954]]
[False, True]
```

	is_genuine	pred
4	True	False
48	True	False
56	True	False
58	True	False
75	True	False
95	True	False
181	True	False
193	True	False
197	True	False
201	True	False
239	True	False
241	True	False
253	True	False
332	True	False
341	True	False
357	True	False
406	True	False
436	True	False
449	True	False
454	True	False
455	True	False
525	True	False
562	True	False
570	True	False
575	True	False
576	True	False
577	True	False
591	True	False
636	True	False
669	True	False
670	True	False
687	True	False
728	True	False
739	True	False

```
804          True  False
832          True  False
875          True  False
877          True  False
892          True  False
913          True  False
916          True  False
931          True  False
951          True  False
970          True  False
980          True  False
985          True  False
1025        False   True
1083        False   True
1122        False   True
1407        False   True
1412        False   True
```

4.5 - Comparaison des performances des modèles

[202]:
```python
df_mods = pd.DataFrame(modrows)
display(df_mods)
```

```
          nom  param_value  Fb-score  avg_exa  min_exa  avg_pre  min_pre  \
0     K-Means         2.00    0.9763   0.9860   0.9800   0.9819   0.9556
1      LogReg         0.85    0.9942   0.9833   0.9767   0.9559   0.9462
2         KNN         2.00    0.9851   0.9840   0.9700   0.9656   0.9158
3  RandForest         2.00    0.9863   0.9660   0.9467   0.9156   0.8898

   avg_rap  min_rap
0   0.9761   0.9652
1   0.9962   0.9898
2   0.9861   0.9796
3   0.9902   0.9796
```

On considère que le modèle optimal pour notre étude est celui qui maximise le F-score. Le modèle retenu est donc le modèle de régression logistique avec seuil de décision à 0.85.

Partie 5 - Enregistrement du modèle optimal

[203]:
```python
scaler_full = StandardScaler()

# Pipeline = modèle
pipeline = Pipeline([
    ("modele", LogisticRegression())
])

# Entraînement
pipeline.fit(X, y)
```

```python
# Sauvegarde en local
joblib.dump(pipeline, "pipeline.joblib")
```

[203]: ['pipeline.joblib']

[204]:
```python
# Chargement du modèle
loaded_pipeline = joblib.load("pipeline.joblib")

# Prédiction des billets à tester
df_pred = pd.DataFrame(data={"pred": loaded_pipeline.predict(X),\
                             "proba": loaded_pipeline.predict_proba(X)[:,1]},\
                       index=X.index)

df_pred["pred"] = (df_pred["proba"] > hyper_psd)

cfm = confusion_matrix(y_true=y, y_pred=df_pred["pred"], labels=[False, True])
print("Matrice de confusion :\n",cfm)
print([False, True],"\n")

df_res = pd.merge(df_work["is_genuine"], df_pred, how='left', left_index=True,␣
↪right_index=True)
display(df_res.loc[df_res["is_genuine"]!=df_res["pred"]])
```

```
Matrice de confusion :
 [[498   2]
 [ 22 978]]
[False, True]
```

	is_genuine	pred	proba
0	True	False	0.390189
6	True	False	0.780080
38	True	False	0.733469
341	True	False	0.720833
392	True	False	0.771722
441	True	False	0.799395
591	True	False	0.379780
623	True	False	0.713768
626	True	False	0.493959
648	True	False	0.829418
665	True	False	0.844109
669	True	False	0.406165
693	True	False	0.838538
700	True	False	0.849994
724	True	False	0.623198
728	True	False	0.145269
743	True	False	0.773188

```
931          True  False  0.832764
946          True  False  0.804088
951          True  False  0.734315
975          True  False  0.683044
985          True  False  0.831133
1122        False   True  0.997135
1412        False   True  0.915315
```

```
[ ]:
```