



ONCFM

ORDRE DU JOUR

Elaboration d'un algorithme capable de déterminer le type de billet (vrai/faux), à partir des seules dimensions géométriques.

1

Analyse exploratoire

- Structure
- Valeurs manquantes & doublons
- Distribution

2

Choix du modèle

- Compléter les NaN : Régression Linéaire & Random Forest
- Prédiction de type de billet (vrai/faux): K-means, Régression Logistique, k-NN

3

Algorithme fonctionnel

- Exportation du modèle final
- Démonstration

I) ANALYSE EXPLORATOIRE DES DONNÉES



1 Import

1.1 Librairies

```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
```

1.2 Fichier

```
billets = pd.read_csv('billets.csv', sep=';', encoding='latin-1')
billets.head()
```

	is_genuine	diagonal	height_left	height_right	margin_low	margin_up	length
0	True	171.81	104.86	104.95	4.52	2.89	112.83
1	True	171.46	103.36	103.66	3.77	2.99	113.09
2	True	172.69	104.48	103.50	4.40	2.94	113.16
3	True	171.36	103.91	103.94	3.62	3.01	113.51
4	True	171.73	104.28	103.46	4.04	3.48	112.54

2 Analyse exploratoire

2.1 Structure

Entrée [3]: `billets.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1500 entries, 0 to 1499
Data columns (total 7 columns):
#   Column          Non-Null Count  Dtype
---  -
0   is_genuine      1500 non-null   bool
1   diagonal        1500 non-null   float64
2   height_left     1500 non-null   float64
3   height_right    1500 non-null   float64
4   margin_low      1463 non-null   float64
5   margin_up       1500 non-null   float64
6   length          1500 non-null   float64
dtypes: bool(1), float64(6)
memory usage: 71.9 KB
```

Entrée [4]: `billets.nunique()`

```
Out[4]: is_genuine      2
diagonal      159
height_left   155
height_right  170
margin_low    285
margin_up     123
length       336
dtype: int64
```

2.3 Data inspection

Entrée [8]: `billets.describe()`

Out[8]:

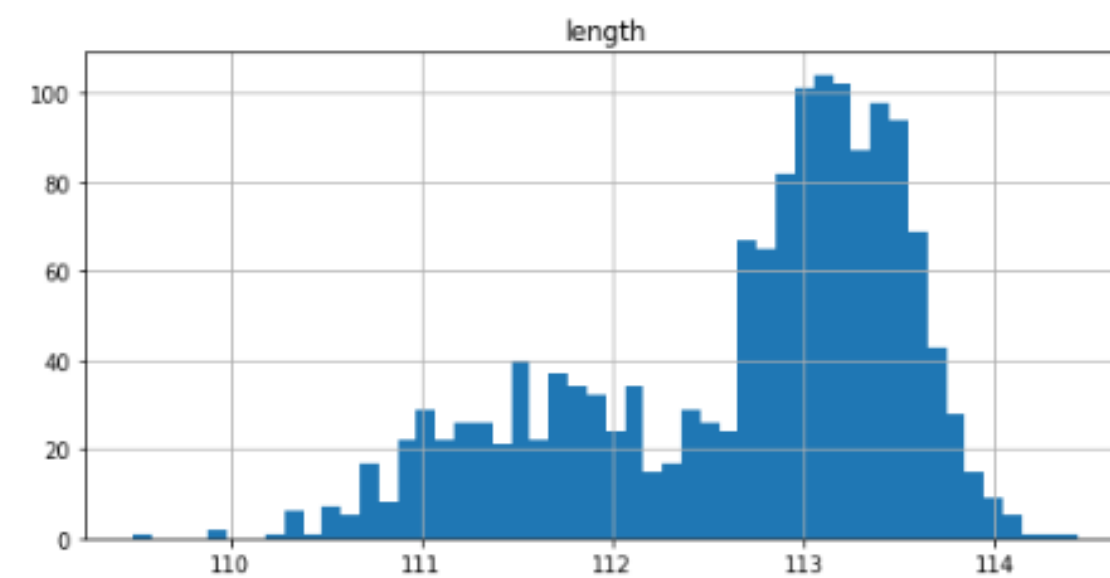
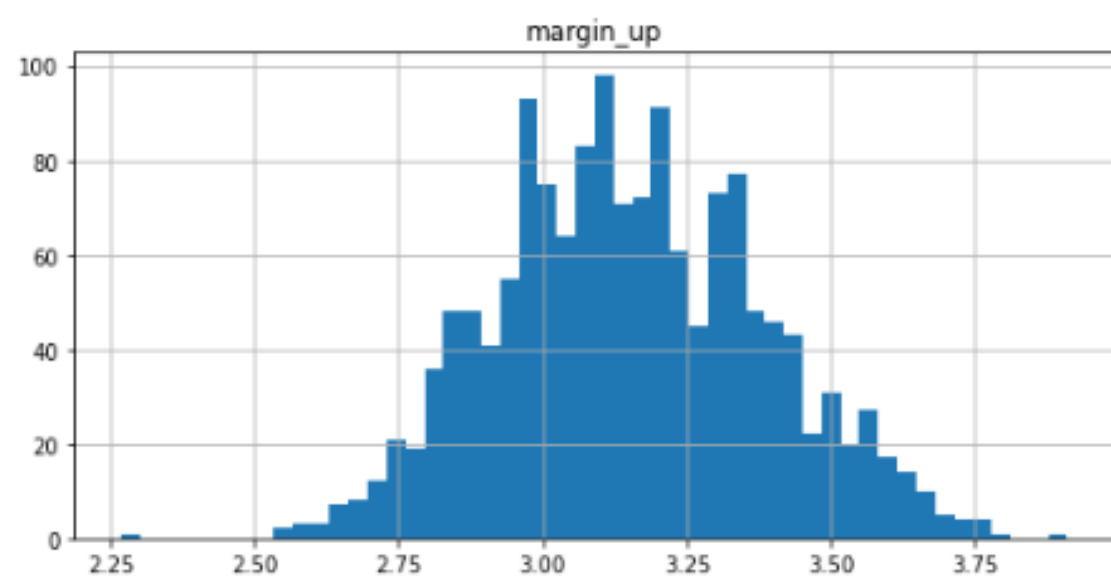
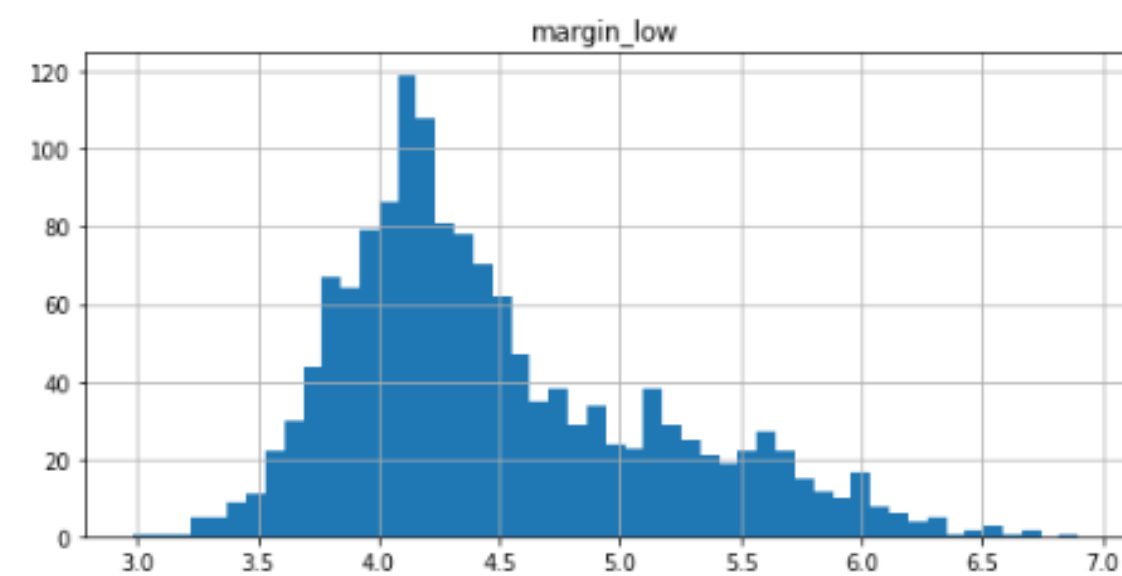
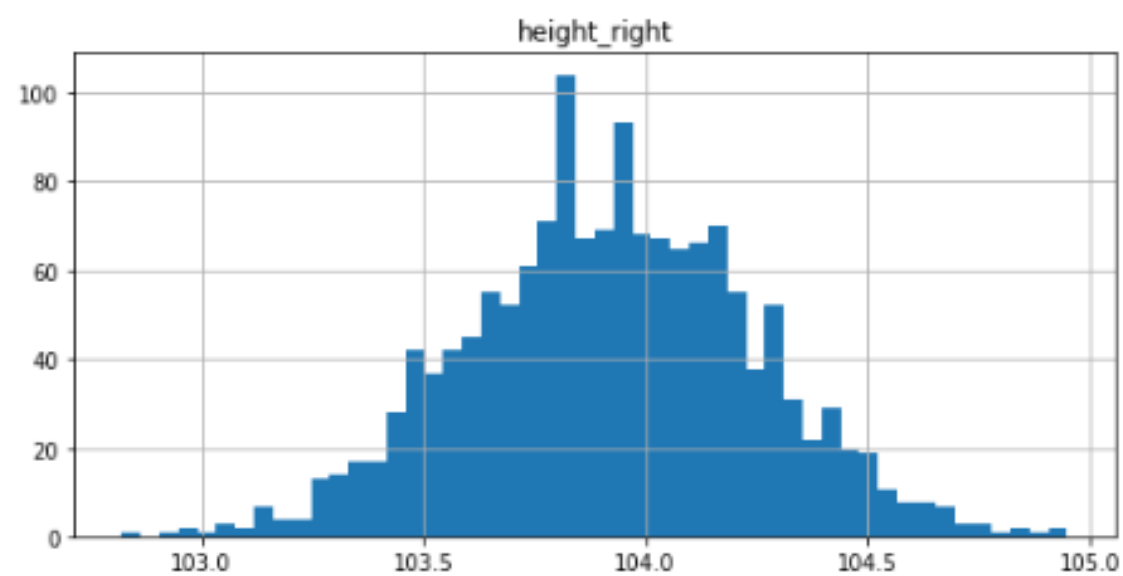
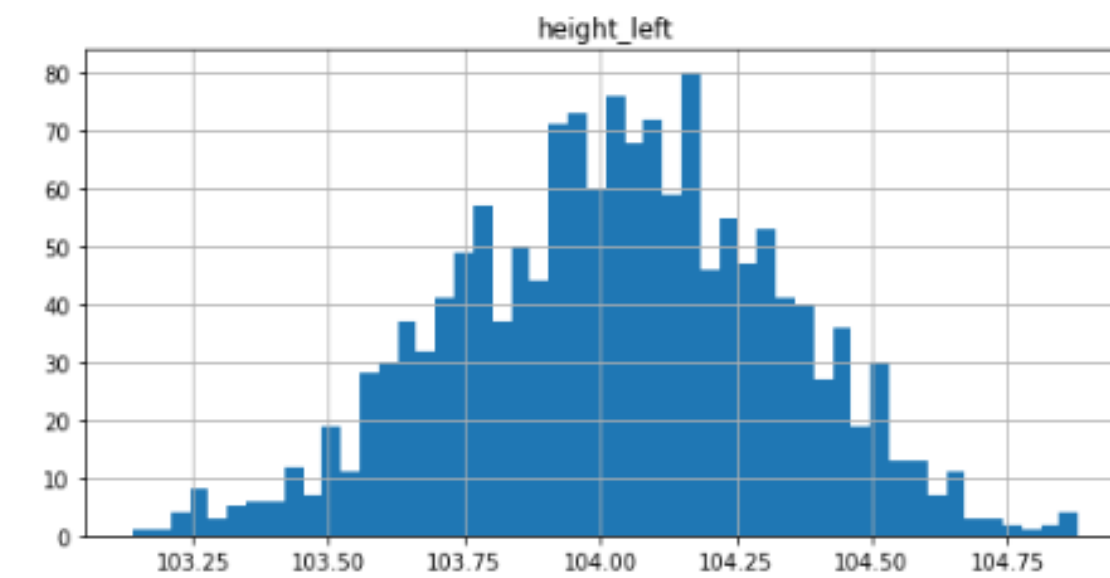
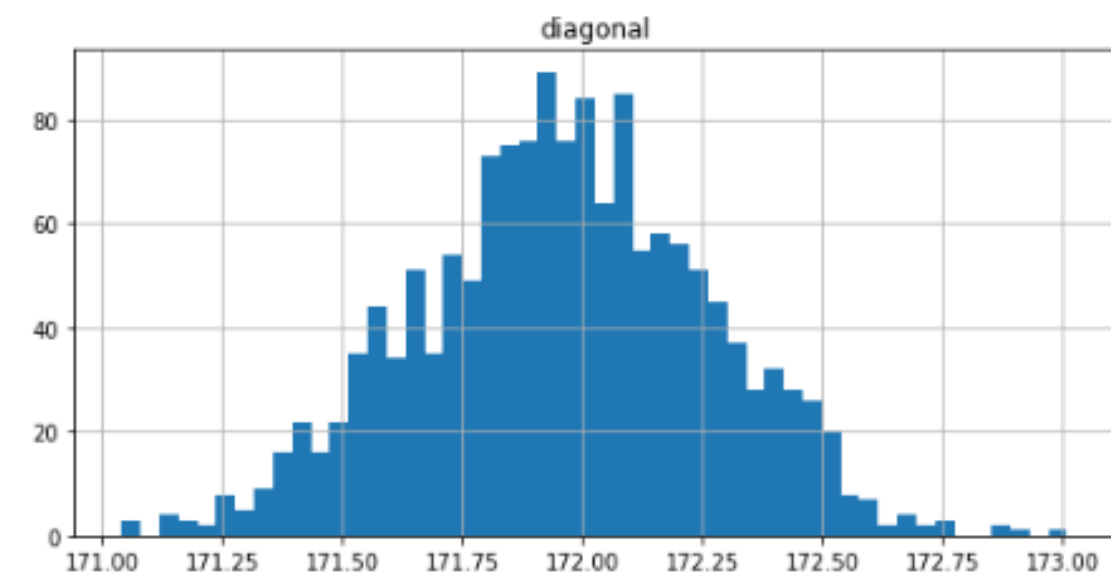
	diagonal	height_left	height_right	margin_low	margin_up	length
count	1500.000000	1500.000000	1500.000000	1463.000000	1500.000000	1500.000000
mean	171.958440	104.029533	103.920307	4.485967	3.151473	112.67850
std	0.305195	0.299462	0.325627	0.663813	0.231813	0.87273
min	171.040000	103.140000	102.820000	2.980000	2.270000	109.49000
25%	171.750000	103.820000	103.710000	4.015000	2.990000	112.03000
50%	171.960000	104.040000	103.920000	4.310000	3.140000	112.96000
75%	172.170000	104.230000	104.150000	4.870000	3.310000	113.34000
max	173.010000	104.880000	104.950000	6.900000	3.910000	114.44000

Entrée [9]: `# Observer la répartition de ma variable d'étude principale (ici si le billet est vrai ou non)`
`billets.iloc[:,0].value_counts()`

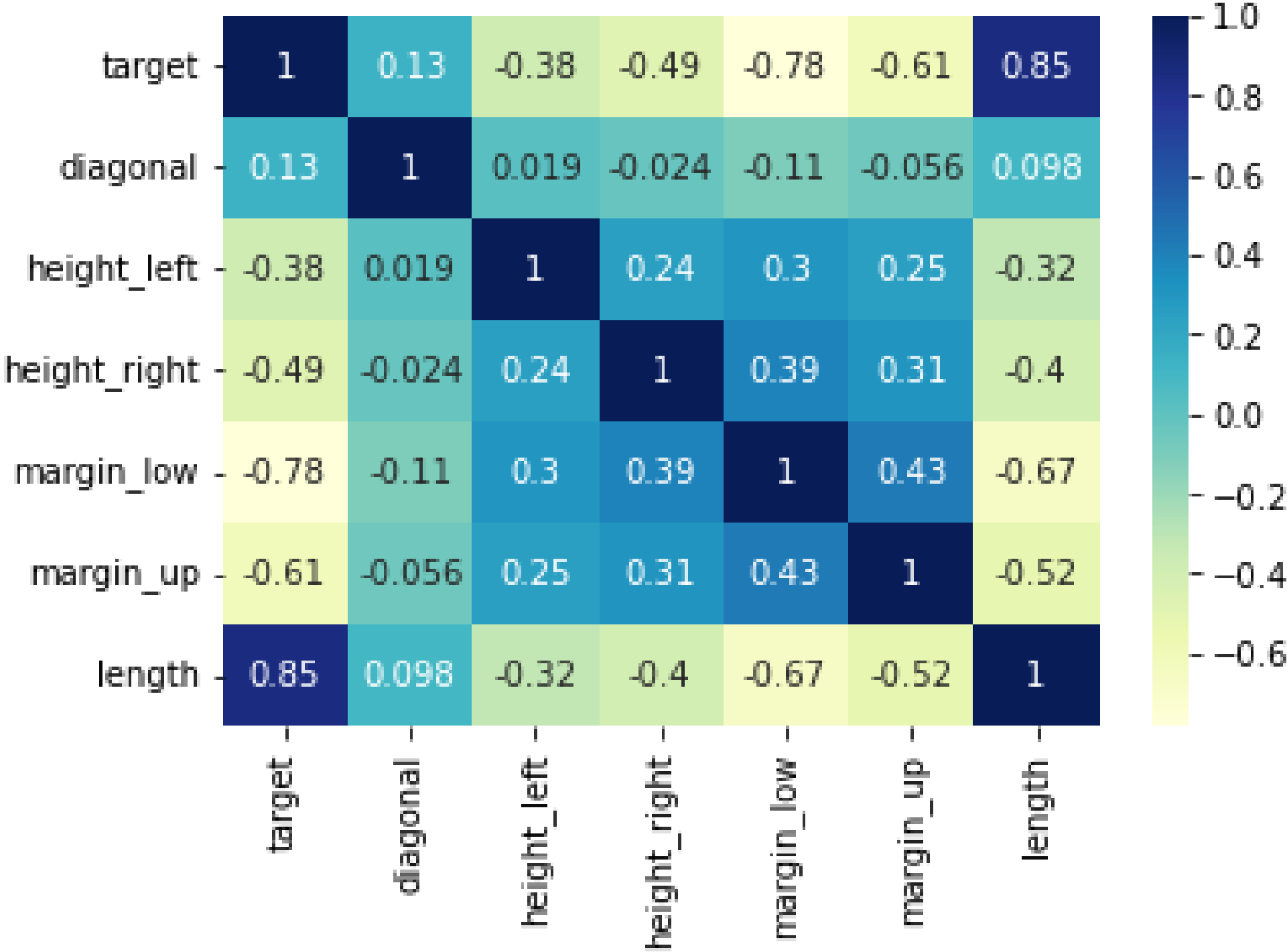
```
Out[9]: True      1000
False      500
Name: is_genuine, dtype: int64
```

Nous avons bien 1000 True et 500 False dans ce jeu de données

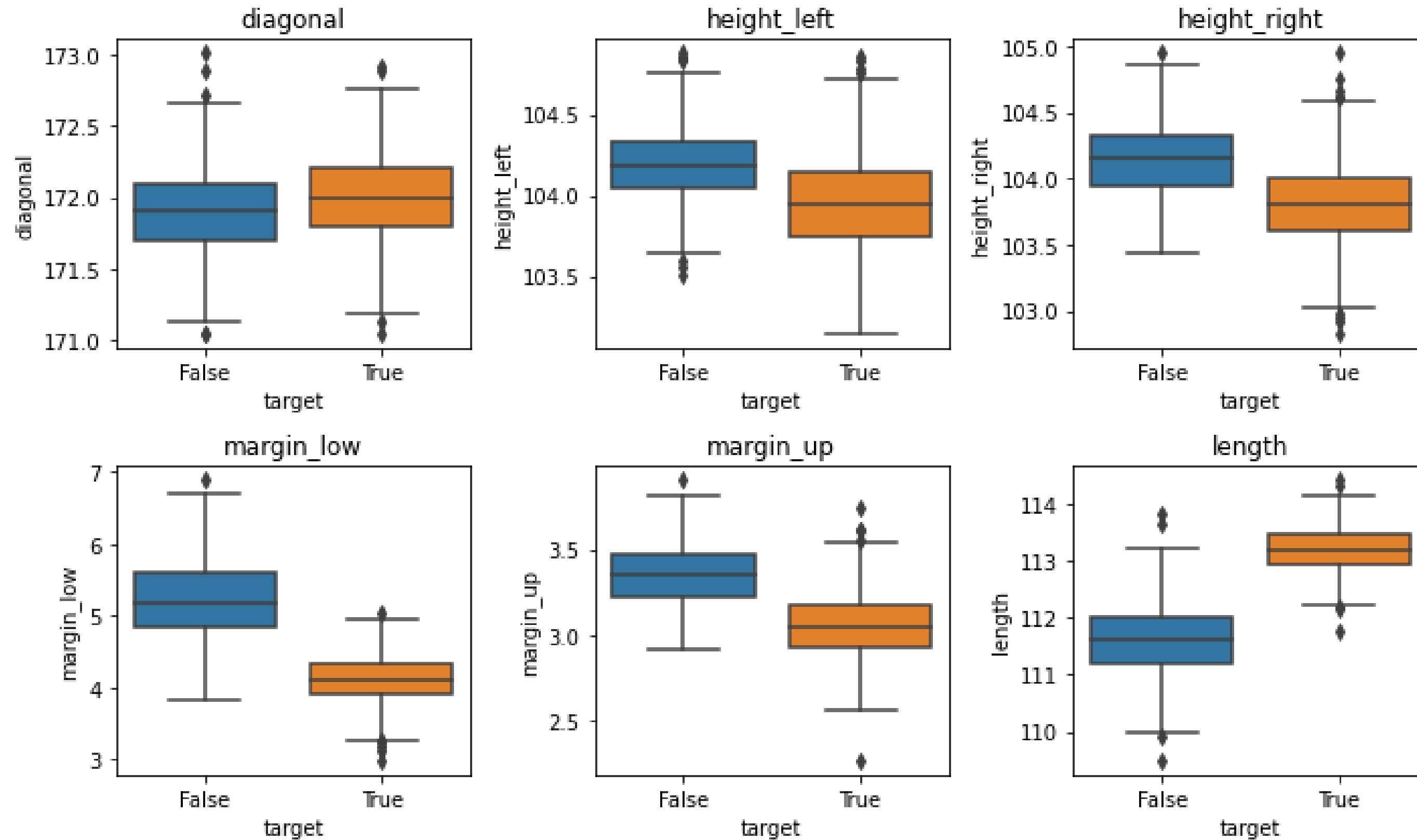
Distribution du nombre d'observations par paramètre



Matrice des corrélations



Différence de paramètres par type de billet



II) CHOIX DU MODÈLE

A) PREDICTION DE VALEURS MANQUANTES



1 Import ¶

1.1 Bibliothèques

```
Entrée [1]: # Base
import pandas as pd
import numpy as np
import pickle

# Visualisation
import matplotlib.pyplot as plt
import seaborn as sns

# ML
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.metrics import roc_curve, auc, confusion_matrix
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsRegressor

# Statistiques
import statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.stats.outliers_influence import variance_inflation_factor
from scipy.stats import t, shapiro
```

1.2 Data Frame

```
Entrée [2]: billets = pd.read_csv('billets_clean.csv')
billets
```

1.3 Fonctions

1.3.1 Afficher score train & score test

```
Entrée [4]: def score(estimator):  
    """ compute and print train score and test score """  
  
    tr_score = estimator.score(X_train, y_train).round(4)  
    te_score = estimator.score(X_test, y_test).round(4)  
  
    print(f"score train : {tr_score} score test : {te_score}")
```

1.3.2 Afficher la matrice de confusion

```
Entrée [5]: def confusion(y_test, y_pred):  
    """display a fancy confusion matrix"""  
    mat = confusion_matrix(y_test, y_pred)  
    mat = pd.DataFrame(mat)  
    mat.columns = [f"pred_{i}" for i in mat.columns]  
    mat.index = [f"test_{i}" for i in mat.index]  
  
    return mat
```

1.3.3 Afficher les meilleurs hyperparamètres

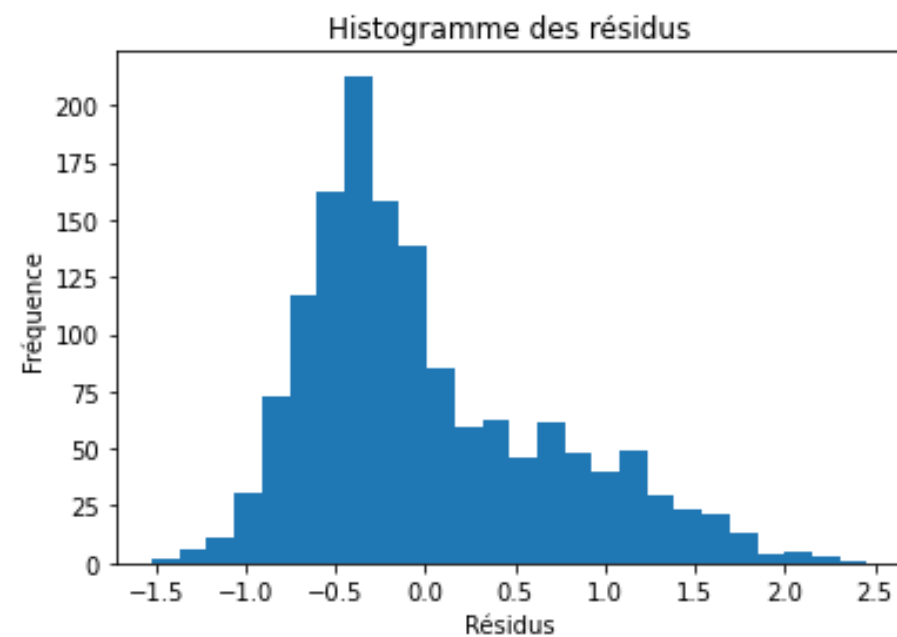
```
Entrée [6]: def resultize(grid):  
    """make a fancy df from grid.cv_results"""  
    # Pour avoir plus d'info au niveau de notre modèle  
    res = grid.cv_results_  
    res = pd.DataFrame(res)  
  
    # On ne garde que des colonnes qui nous intéresse (en enlevant les col "split")  
    cols = [i for i in res.columns if "split" not in i]  
  
    #afficher les meilleurs résultats en premier  
    res = res[cols]  
    res = res.sort_values("rank_test_score")  
  
    return res
```

2 Choix du modèle de ML pour la prédiction des NaN

2.1 Régression linéaire multiple

2.1.1 Validité du modèle

Normalité des résidus



p-value du test de normalité : 7.239837024766648e-25

Homoscédasticité

```
X = billets_without_NaN[['diagonal', 'height_left', 'height_right', 'margin_up', 'length']]
y = billets_without_NaN['margin_low']

levене_test = levene(X['diagonal'], X['height_left'], X['height_right'], X['margin_up'], X['length'], y, center='median')

print(f"Statistic: {levене_test.statistic}")
print(f"p-value: {levене_test.pvalue}")

Statistic: 477.3663829118897
p-value: 0.0
```

**les résidus ne suivent pas
une distribution normale**

+

**homoscédasticité
non respectée**

=

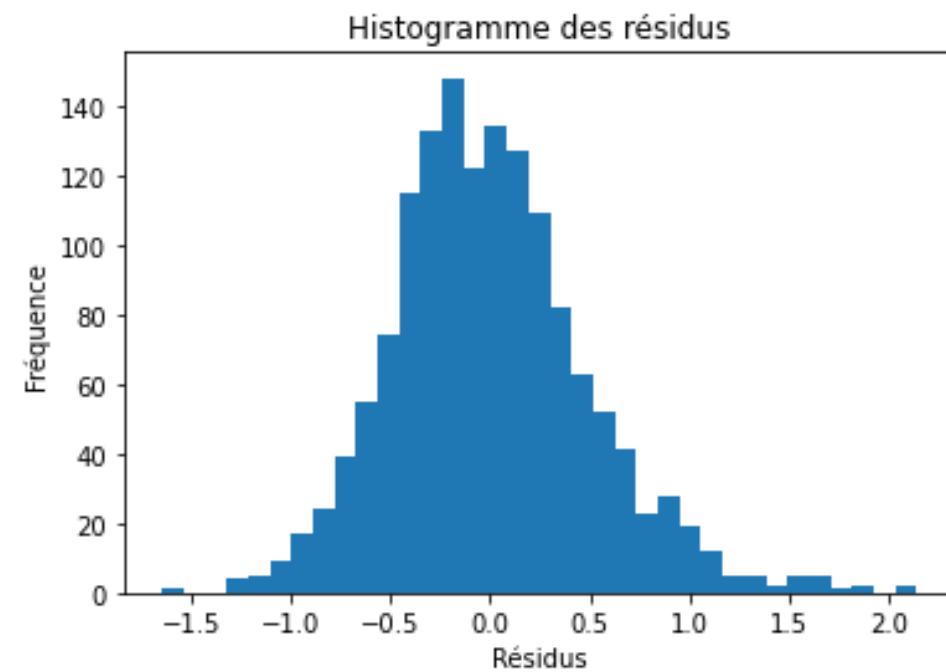
**Modèle de RLM
non valide**

2 Choix du modèle de ML pour la prédiction des NaN

2.2 Régression linéaire simple

2.2.1 Validité du modèle

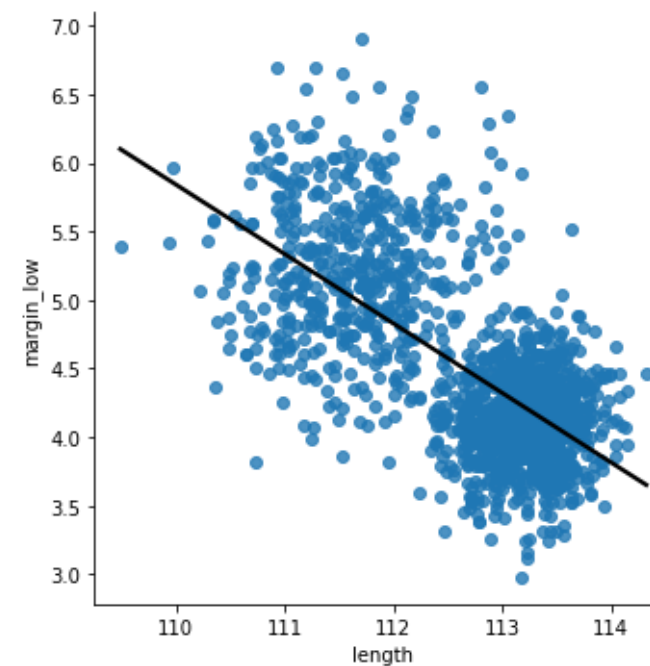
Normalité des résidus



p-value du test de normalité : 2.3925759377180222e-12

Homoscédasticité

```
ax = sns.lmplot(x="length", y="margin_low", data=billets_without_NaN, ci=None, line_kws={'color':'black'})  
ax.set(xlabel='length', ylabel='margin_low')  
plt.show()
```



**les résidus ne suivent pas
une distribution normale
avec la p-value proche de 0**

+

**homoscédasticité
non respectée**

=

**Modèle de RLS
non valide**

2.2 Random Forest

2.2.1 Construction du modèle RF

```
Entrée [27]: # Sélectionner les colonnes pour l'entraînement et la prédiction

X = billets_without_NaN[['diagonal', 'height_left', 'height_right', 'margin_up', 'length']]
y = billets_without_NaN['margin_low']
```

```
Entrée [28]: # Diviser les données en ensembles d'entraînement et de test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
Entrée [29]: # Construction du modèle de forêt aléatoire à partir des données d'entraînement

estimator_rf = RandomForestRegressor(n_estimators=1000, random_state=42)
estimator_rf.fit(X_train, y_train)
```

```
Out[29]: RandomForestRegressor(n_estimators=1000, random_state=42)
```

```
Entrée [30]: # Prédiction de valeurs de notre target (margin_low) à partir des variables de départ (X_test)

y_pred = estimator_rf.predict(X_test)
```


Evaluation des performances du modèle

```
# Evaluation des performances du modèle  
  
mse_RF_base = mean_squared_error(y_test, y_pred)  
r2_RF_base = r2_score(y_test, y_pred)  
  
print("MSE: {:.3f}".format(mse_RF_base))  
print("R2 Score: {:.3f}".format(r2_RF_base))
```

```
MSE: 0.170  
R2 Score: 0.596
```

```
rmse_base_RF = np.sqrt(mse_RF_base)  
print('RMSE : {:.3f}'.format(rmse_base_RF))
```

```
RMSE : 0.413
```

2.2.2 Recherche de meilleurs hyperparamètres

```
: # Sélectionner les colonnes pour l'entraînement et la prédiction
X = billets_without_NaN[['diagonal', 'height_left', 'height_right', 'margin_up', 'length']]
y = billets_without_NaN['margin_low']

# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Construction de l'estimateur de base
estimator_RF_best = RandomForestRegressor(random_state=42)

# Grille de valeurs d'hyperparamètres à tester
param_grid = {
    'n_estimators': [100, 500, 1000], #nombre d'arbres dans la foret (si gd = + précis -lent)
    # 'max_leaf_nodes': [None, 3, 10, 20], #permet de choisir le nb de feuilles de notre arbre(plus il est grand et plus on va tendre
    'max_depth': [None, 10, 20], #la longueur max de l'arbre (de préférence il faut bp d'arbre, peu prodonds)

    'max_features': ['auto', 'sqrt'], #le nb de variables à tester pour trouver la meilleure séparation de notre arbre
    'min_samples_split': [2, 5, 10], #(plus il est gd et plus l'arbre sera simple et généralisable)
    'min_samples_leaf': [1, 2, 4] #le min d'exemple requis pour créer une feuille (si pas assez d'exemples la feuille ne sera pas
}

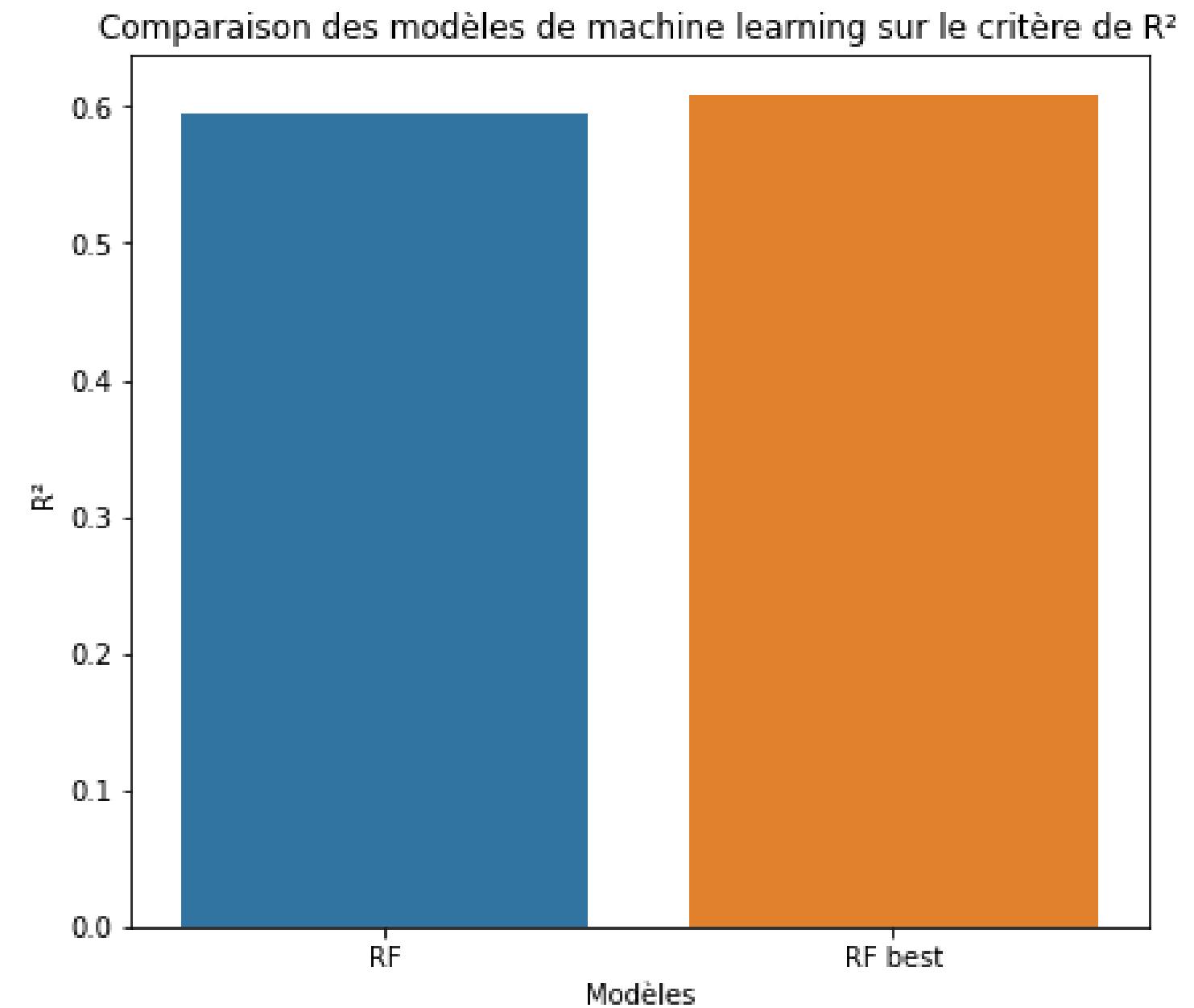
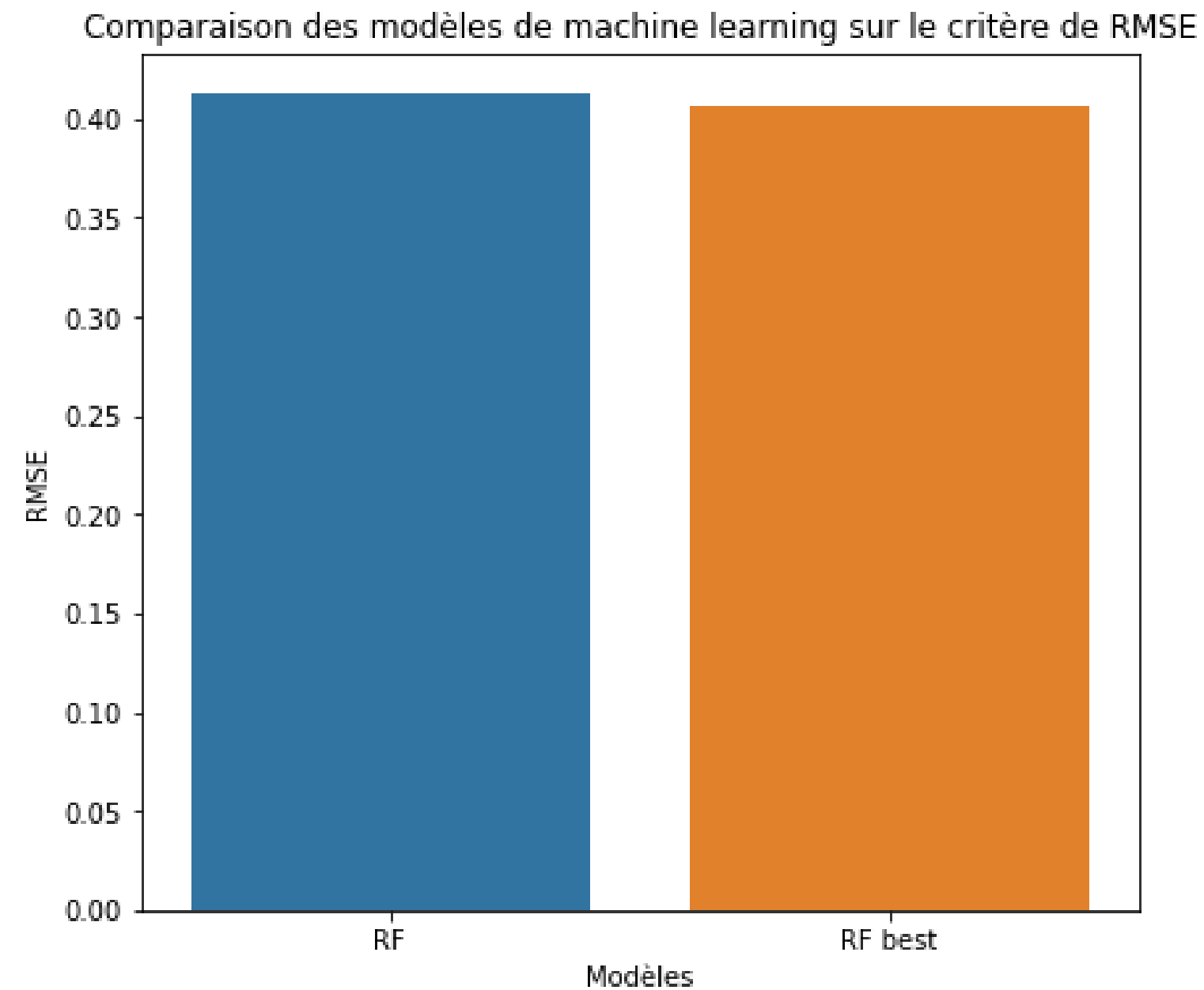
# Recherche des meilleurs hyperparamètres par validation croisée
grid_search = GridSearchCV(estimator=estimator_RF_best, param_grid=param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Affichage des meilleurs hyperparamètres et de la performance du modèle
print("Meilleurs hyperparamètres : ", grid_search.best_params_)
print("Performance du modèle : ", grid_search.best_score_)
```

Meilleurs hyperparamètres : {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}

Performance du modèle : 0.5111088209531769

Choix du modèle en fonction de performances R^2 & RMSE



On cherche à minimiser le MSE et à maximiser le R^2 . Ici c'est bien le RF best qui a les meilleurs performances.


2.4 Prédiction

On sélectionne les colonnes nécessaires pour l'entraînement et la prédiction, qui sont les mêmes que celles utilisées lors de la construction de votre modèle.

```
Entrée [50]: X_train = billets.loc[billets['margin_low'].notnull(), ['diagonal', 'height_left', 'height_right', 'margin_up', 'length']]
y_train = billets.loc[billets['margin_low'].notnull(), 'margin_low']
X_pred = billets.loc[billets['margin_low'].isnull(), ['diagonal', 'height_left', 'height_right', 'margin_up', 'length']]
```

On construit un modèle de Random Forest en utilisant les meilleurs hyperparamètres trouvés précédemment.

```
Entrée [51]: estimator_RF_best = RandomForestRegressor(n_estimators=100, max_depth=10, max_features='sqrt', min_samples_leaf=1, min_samples_sp
```



On entraine le modèle

```
Entrée [52]: estimator_RF_best.fit(X_train, y_train)
```

```
Out[52]: RandomForestRegressor(max_depth=10, max_features='sqrt', min_samples_split=5,
                                random_state=42)
```

On fait des prédictions sur les données à remplacer.

```
Entrée [53]: y_pred = estimator_RF_best.predict(X_pred)
```

On remplace les valeurs manquantes de la colonne "margin_low" dans le data frame "billets" par les valeurs prédites.

```
Entrée [54]: billets.loc[billets['margin_low'].isnull(), 'margin_low'] = y_pred
```

2.5 Vérification

Entrée [55]: `billets.describe()`

Out[55]:

	diagonal	height_left	height_right	margin_low	margin_up	length
count	1500.000000	1500.000000	1500.000000	1500.000000	1500.000000	1500.000000
mean	171.958440	104.029533	103.920307	4.483625	3.151473	112.67850
std	0.305195	0.299462	0.325627	0.659622	0.231813	0.87273
min	171.040000	103.140000	102.820000	2.980000	2.270000	109.49000
25%	171.750000	103.820000	103.710000	4.027900	2.990000	112.03000
50%	171.960000	104.040000	103.920000	4.310000	3.140000	112.96000
75%	172.170000	104.230000	104.150000	4.870000	3.310000	113.34000
max	173.010000	104.880000	104.950000	6.900000	3.910000	114.44000

Entrée [56]: `billets.isna().sum()`

Out[56]:

target	0
diagonal	0
height_left	0
height_right	0
margin_low	0
margin_up	0
length	0

dtype: int64

Entrée [57]: `billets.head(5)`

Out[57]:

	target	diagonal	height_left	height_right	margin_low	margin_up	length
0	True	171.81	104.86	104.95	4.52	2.89	112.83
1	True	171.46	103.36	103.66	3.77	2.99	113.09
2	True	172.69	104.48	103.50	4.40	2.94	113.16
3	True	171.36	103.91	103.94	3.62	3.01	113.51
4	True	171.73	104.28	103.46	4.04	3.48	112.54

II) CHOIX DU MODÈLE

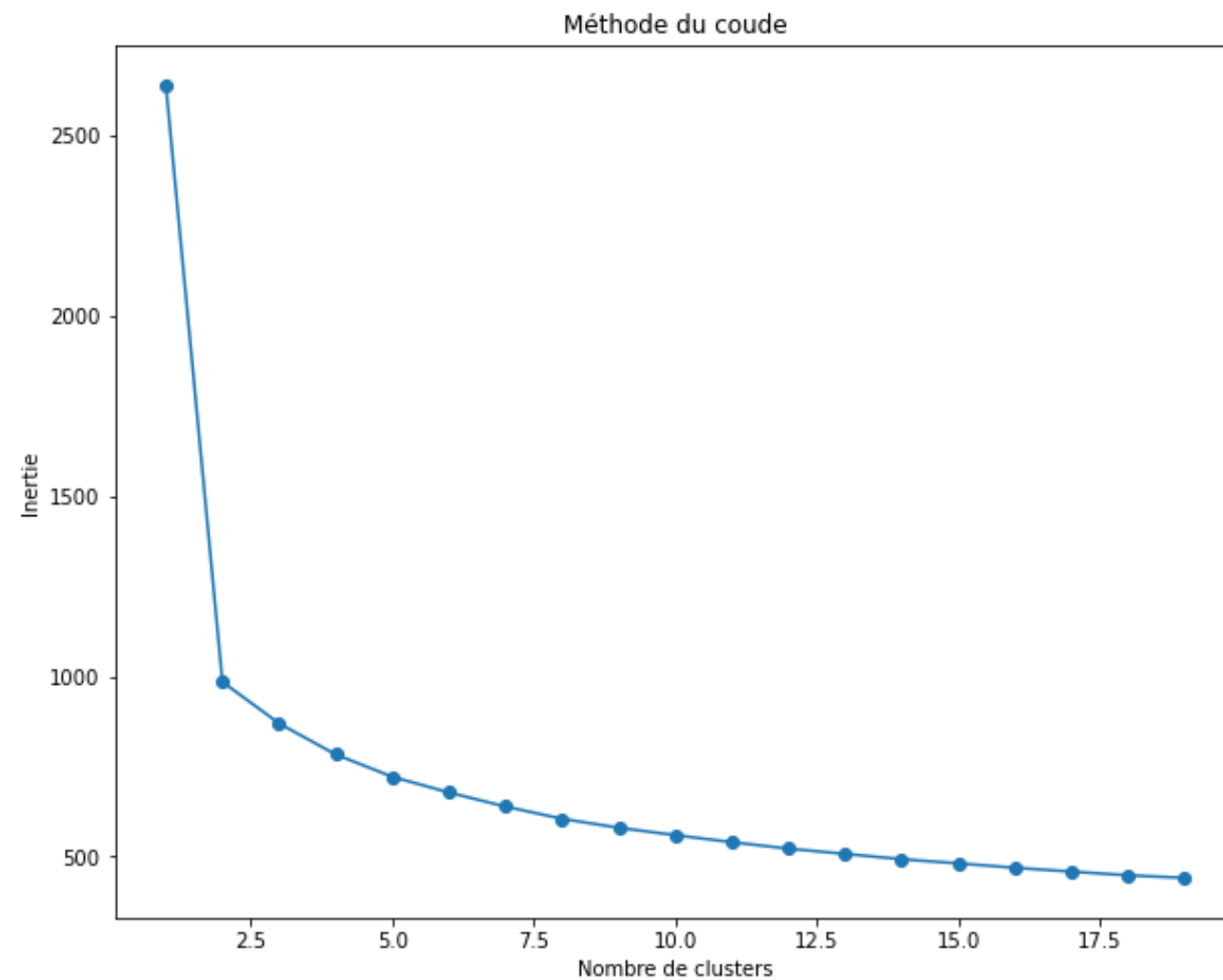
B) PREDICTION DE TYPE DE BILLET



K-Means

Le nombre de clusters

Méthode du coude



2 clusters

Méthode de la silhouette

```
silhouette_scores = []

for n_clusters in range(2, 11):
    kmeans = KMeans(n_clusters=n_clusters)
    kmeans.fit(X)
    silhouette_avg = silhouette_score(X, kmeans.labels_)
    silhouette_scores.append(silhouette_avg)

best_num_clusters = silhouette_scores.index(max(silhouette_scores)) + 2

print("Nombre de clusters optimal : ", best_num_clusters)
```

Nombre de clusters optimal : 2

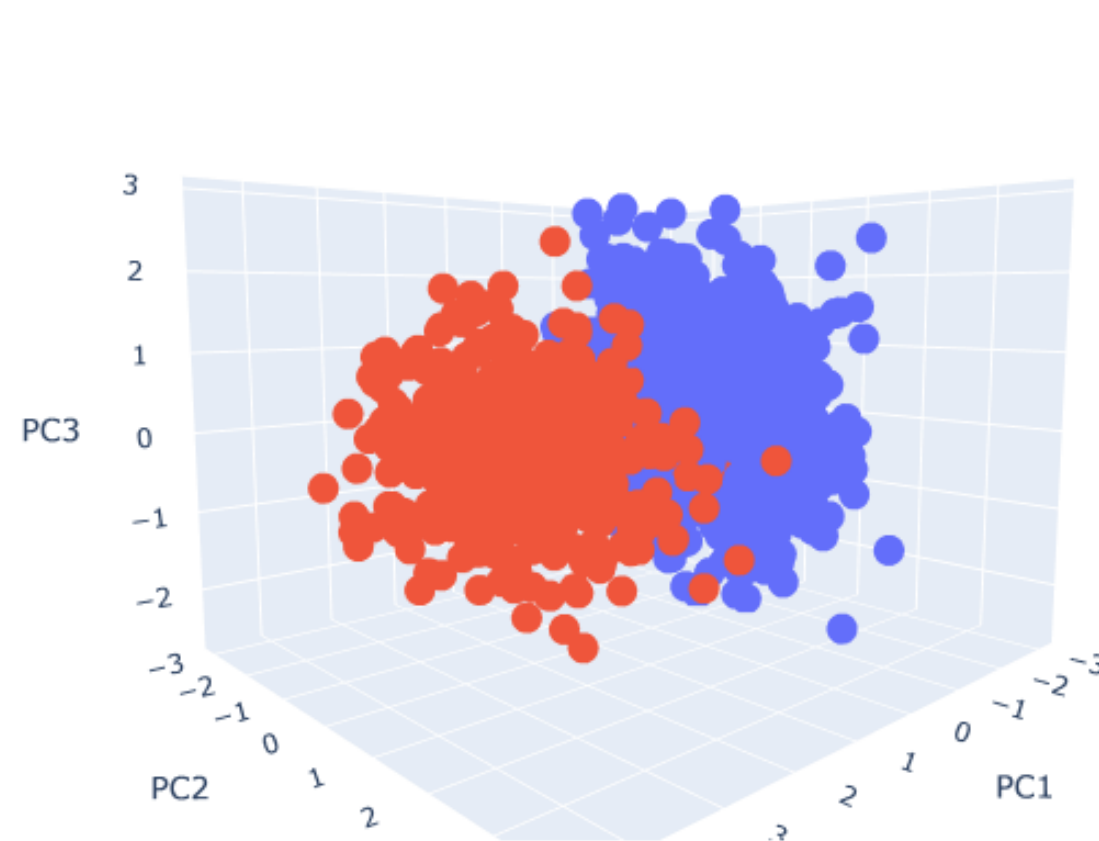
2 clusters

=

K-Means

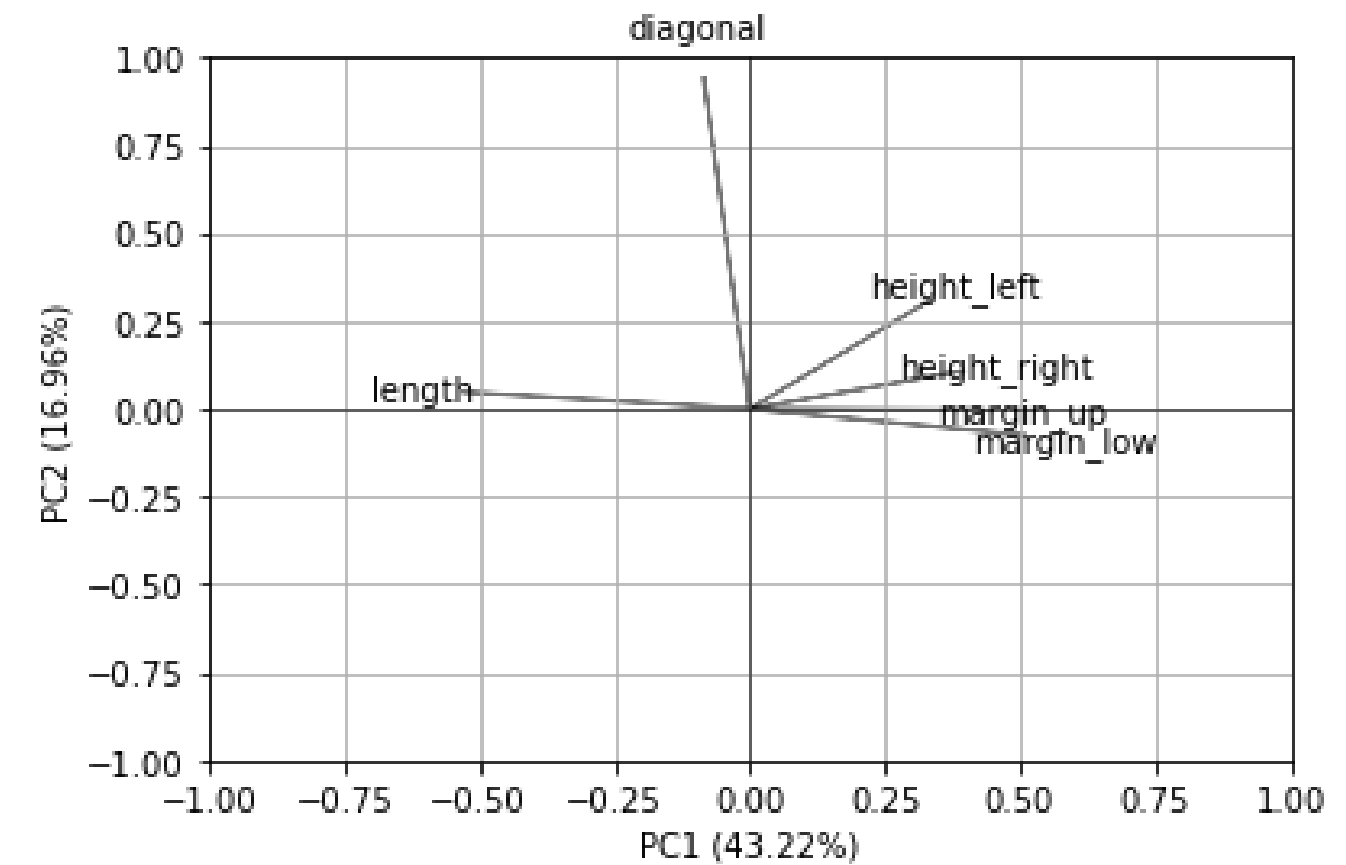
ACP

Projection des individus



2 clusters

Cercle des corrélations



**Influence des variables sur les
composantes principales**

3.1 K-means

3.1.1 Construction du modèle KM

```
# Sélectionner les colonnes pour l'entraînement et la prédiction
```

```
X = billets[['diagonal', 'height_left', 'height_right', 'margin_up', 'length', 'margin_low']] #variable de départ  
y = billets['target'] #variable à prédire
```

```
# Diviser les données en ensembles d'entraînement et de test
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Construction du modèle K-means à partir des données d'entraînement
```

```
kmeans = KMeans(n_clusters=2, random_state=42)  
kmeans.fit(X_train, y_train)
```

```
KMeans(n_clusters=2, random_state=42)
```

```
# Prédiction des groupes d'appartenance pour les données de test
```

```
y_pred = kmeans.predict(X_test)
```

```
# Inverser les prédictions (valable pour une cellule uniquement)
```

```
y_pred = 1 - y_pred
```

```
# Matrice de confusion
```

```
confusion(y_test, y_pred)
```

	pred_0	pred_1
test_0	104	6
test_1	0	190

3.1.3 Recherche de meilleurs hyperparamètres

```
# Sélectionner les colonnes pour l'entraînement et la prédiction
X = billets[['diagonal', 'height_left', 'height_right', 'margin_up', 'length', 'margin_low']]
y = billets['target']

# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Construction de l'estimateur de base
estimator = KMeans(random_state=42)

# Grille de valeurs d'hyperparamètres à tester
param_grid = {
    'init': ['k-means++', 'random'],
    'n_init': [5, 10, 15], # Le nombre de fois que l'algorithme sera exécuté avec des centroides différents
    'max_iter': [100, 200, 300, 400, 500] # Le nombre maximum d'itérations pour une seule exécution
}

# Recherche des meilleurs hyperparamètres par validation croisée
grid_search = GridSearchCV(estimator=estimator, param_grid=param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Affichage des meilleurs hyperparamètres et de la performance du modèle
print("Meilleurs hyperparamètres : ", grid_search.best_params_)
```

Meilleurs hyperparamètres : {'init': 'k-means++', 'max_iter': 100, 'n_init': 5}

	pred_0	pred_1
test_0	103	7
test_1	0	190

Régression Logistique

```
reg_log1 = smf.glm('target ~ diagonal+height_left+height_right+margin_up+length+margin_low', data=billets, family=sm.families.Binomial()).fit()  
print(reg_log1.summary())
```

Generalized Linear Model Regression Results						
=====						
Dep. Variable:	['target[False]', 'target[True]']			No. Observations:	1500	
Model:	GLM			Df Residuals:	1493	
Model Family:	Binomial			Df Model:	6	
Link Function:	Logit			Scale:	1.0000	
Method:	IRLS			Log-Likelihood:	-42.800	
Date:	Mon, 29 May 2023			Deviance:	85.601	
Time:	15:37:38			Pearson chi2:	2.14e+03	
No. Iterations:	10			Pseudo R-squ. (CS):	0.7036	
Covariance Type:	nonrobust					
=====						
	coef	std err	z	P> z	[0.025	0.975]

Intercept	207.9165	240.242	0.865	0.387	-262.949	678.782
diagonal	-0.1280	1.081	-0.118	0.906	-2.247	1.991
height_left	1.5802	1.087	1.454	0.146	-0.550	3.710
height_right	2.3188	1.049	2.210	0.027	0.262	4.376
margin_up	10.2749	2.136	4.811	0.000	6.089	14.461
length	-5.7783	0.819	-7.053	0.000	-7.384	-4.173
margin_low	5.6115	0.896	6.266	0.000	3.856	7.367
=====						

6 variables explicatives



```
reg_log1 = smf.glm('target ~ height_right+margin_up+length+margin_low', data=billets, family=sm.families.Binomial()).fit()  
print(reg_log1.summary())
```

=====						
Dep. Variable:	['target[False]', 'target[True]']			No. Observations:	1500	
Model:	GLM			Df Residuals:	1495	
Model Family:	Binomial			Df Model:	4	
Link Function:	Logit			Scale:	1.0000	
Method:	IRLS			Log-Likelihood:	-43.890	
Date:	Mon, 29 May 2023			Deviance:	87.779	
Time:	15:37:38			Pearson chi2:	2.59e+03	
No. Iterations:	10			Pseudo R-squ. (CS):	0.7031	
Covariance Type:	nonrobust					
=====						
	coef	std err	z	P> z	[0.025	0.975]

Intercept	311.3898	136.206	2.286	0.022	44.431	578.349
height_right	2.7567	1.052	2.619	0.009	0.694	4.819
margin_up	10.3617	2.121	4.885	0.000	6.205	14.519
length	-5.8490	0.804	-7.277	0.000	-7.424	-4.274
margin_low	5.8580	0.858	6.830	0.000	4.177	7.539
=====						

4 variables explicatives
(significatives uniquement)

Régression Logistique

```
# Sélectionner les colonnes pour l'entraînement et la prédiction  
|  
X = billets[['height_right', 'margin_up', 'length', 'margin_low']] #variable de départ  
y = billets['target'] #variable à prédire
```

```
# Diviser les données en ensembles d'entraînement et de test  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
estimator_RL_base = LogisticRegression(solver="liblinear")  
  
estimator_RL_base.fit(X_train,y_train)
```

```
LogisticRegression(solver='liblinear')
```

```
score(estimator_RL_base)
```

```
score train : 0.9883 score test : 0.9867
```

	pred_0	pred_1
test_0	106	4
test_1	0	190

3.2.3 Recherche de meilleurs hyperparamètres

```
estimator = LogisticRegression()

# ici ce sont tous des hyper paramètres qu'on fixe en avance

params = { 'C': np.logspace(-3,3,7), #est une valeur numérique que nous prenons ici avec des échelles logarytmiques
            'penalty': ['l1','l2'], # est le coef de régularisation de niveau 1(l1) ou 2 (l2)
            'solver': ['newton-cg', 'lbfgs', 'liblinear']
          }
#params c'est toujours un dictionnaire + liste itérable à droite (valeurs)
```

```
grid = GridSearchCV(estimator, # précisé avant
                    params, # Les params déterminés avant
                    cv = 10, # 10 sous échantillons = "10 folds"
                    n_jobs = -1, #force notre GridSearch à travailler sur l'ensemble des CPU de notre machine
                    return_train_score = True, #il nous renvoie les scores de train
                    verbose = 1) # la quantité d'info qui sera renvoyé par GridSearch lors de son déroulement
grid.fit(X_train, y_train)
```

Fitting 10 folds for each of 42 candidates, totalling 420 fits

```
# Les meilleurs hyperparamètres pour notre modèle

best_params = grid.best_params_
best_params
```

- {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}

-

- | | pred_0 | pred_1 |
|--------|--------|--------|
| test_0 | 106 | 4 |
| test_1 | 0 | 190 |

k-NN

```
# Sélectionner les colonnes pour l'entraînement et la prédiction

X = billets[['height_right', 'margin_up', 'length', 'margin_low']] #variable de départ
y = billets['target'] #variable à prédire

# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Construction du modèle de K-NN
estimator_knn = KNeighborsRegressor(n_neighbors=5) #(par défaut avec la distance euclidienne)

# Entraînement du modèle
estimator_knn.fit(X_train, y_train)

# Prédiction sur les données de test
y_pred = estimator_knn.predict(X_test)

# Définir un seuil de classification pour les valeurs prédites (ici 0.5)
y_pred_discret = (y_pred > 0.5).astype(int)

# Créer la matrice de confusion
conf_mat = confusion_matrix(y_test, y_pred_discret)

# Afficher la matrice de confusion sous forme de DataFrame
conf_df = pd.DataFrame(conf_mat, index=['test_0', 'test_1'], columns=['pred_0', 'pred_1'])
conf_df
```

	pred_0	pred_1
test_0	108	2
test_1	0	190

3.3.2 Recherche de meilleurs hyperparamètres

```
# Définir Les valeurs des hyperparamètres à tester
param_grid = {
    'n_neighbors': [3, 5, 7, 10, 15, 25, 30],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}

# Créer un objet GridSearchCV avec Le modèle KNN et Les hyperparamètres à tester
grid_search = GridSearchCV(KNeighborsRegressor(), param_grid, cv=5)

# Exécuter La validation croisée sur Les données d'entraînement
grid_search.fit(X_train, y_train)

# Afficher Les meilleurs hyperparamètres trouvés et Leur score de validation croisée
print("Meilleurs hyperparamètres : ", grid_search.best_params_)
print("Score de validation croisée : ", grid_search.best_score_)
```

```
Meilleurs hyperparamètres : {'n_neighbors': 25, 'p': 1, 'weights': 'distance'}
Score de validation croisée : 0.959330907860069
```

```
best_params = {'n_neighbors': 25, 'p': 1, 'weights': 'distance'}
```

```
# Construction du modèle de K-NN
knn_best = KNeighborsRegressor(**best_params)

# Entraînement du modèle
knn_best.fit(X_train, y_train)

# Prédiction sur Les données de test
y_pred = knn_best.predict(X_test)
```

-
-
-

	pred_0	pred_1
test_0	106	4
test_1	0	190

COMPARAISON DES PERFORMANCES

	K-means_base	K-means_best	Régression Logistique_base	Régression Logistique_best	k-NN_base	k-NN_best
RocAuc (taux de vrai positif/négatif)	0.9139	0.9727	0.9818	0.9818	0.9954	0.995
R ²	0.9139	0.9563	0.9426	0.9426	0.9569	0.9563
Train/test score	Non disponible	Non disponible	score train : 0.9883 score test : 0.9867	score train : 0.9892 score test : 0.9867	score train : 0.9719 score test : 0.965	score train : 1.0 score test : 0.9563
Choix						

DÉMONSTRATION DU MODÈLE

Régression Logistique avec les hyperparamètres suivant: {'C':
1, 'penalty': 'l2', 'solver' : 'newton-cg'}

