ÉCOLE POLYTECHNIQUE EXECUTIVE EDUCATION

DATA SCIENCE STARTER PROGRAM

Conduire un projet de sciences de données

# Valeo

## Defect Prediction on production lines by Valeo

Wajdi Alphonse HAROUNY
DSSP14

06/2020

## Index

- c. Courbes de ROC_AUC et Precision Recall sur Test Set
- d. ROC_AUC sur le test set caché de l'ENS

16. RusBoost et AdaBoost
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
17. Random Forest avec SomteENN:
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
18. Random Forest avec SmoteTomek
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
19. Random Forest avec BorderLineSmote et RandomUnderSampler
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
20. Régression logistique associée à un suréchantillonnage /échantillonnage combiné SmoteENN
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
21. Support Vector Machine Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
22. KNeighbors Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set
    - d. ROC_AUC sur le test set caché de l'ENS
23. GaussianNB Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN
    - a. Caractéristiques de l'algorithme
    - b. Hyperparamètres utilisés en cross validation et mesures
    - c. Courbes de ROC_AUC et Precision Recall sur Test Set

# 1 - Contexte de l'étude

L'étude correspond à un défi industriel du type data proposé par Valéo au Challenge Data organisé par l'ENS et présenté au Collège de France en Janvier 2020.

Le jeu de données objet de l'étude provient d'une vraie ligne de production de démarreurs, avec un démarreur produit toutes les 12 secondes.
La ligne d'assemblage et de process est composée de 15 stations complètement automatisées, gérées par des robots (*Figure 1.a*) qui assemblent les démarreurs et qui stockent toutes les mesures enregistrées par les process : Les angles de vissages et leurs couples, les mesures d'insertions, les emmanchements ainsi que les mesures d'autres process.

*Figure 1.a - Ligne d'assemblage et de process*

En fin de ligne, un banc de test est l'épreuve de vérité : "démarreur en bon état" ou "démarreur en état défectueux" (*Figure 1.b*).
Un "démarreur en état défectueux" engendre du ‘rework’ et du ‘scrap’ pour l'industriel : C'est une perte de temps, d'énergie et d'argent.

*Figure 1.b - Banc de test*

Le problème se ramène à une prédiction de classification binaire mesurée sur le banc de test avec une variable de sortie "démarreur en bon état" ou "démarreur en état défectueux". Pour la suite de l'étude, un "démarreur en bon état" sera annoté comme NEGATIF et un "démarreur en mauvais état" comme POSITIF.

L'objectif de l'étude est de détecter les "démarreurs en mauvais état" sans pour autant rejeter les "démarreur en bon état".

**Mon meilleur modèle se base sur le classifieur "Régression logistique associé à une technique de sur-échantillonnage et de sous échantillonnage SomteEnn".**

**Il occupe la 30ème place sur un nombre total de 137 participants. Mon score (roc_auc) est égal à 0.6904 pour une plage de scores variant entre 0.43 et 0.76.**

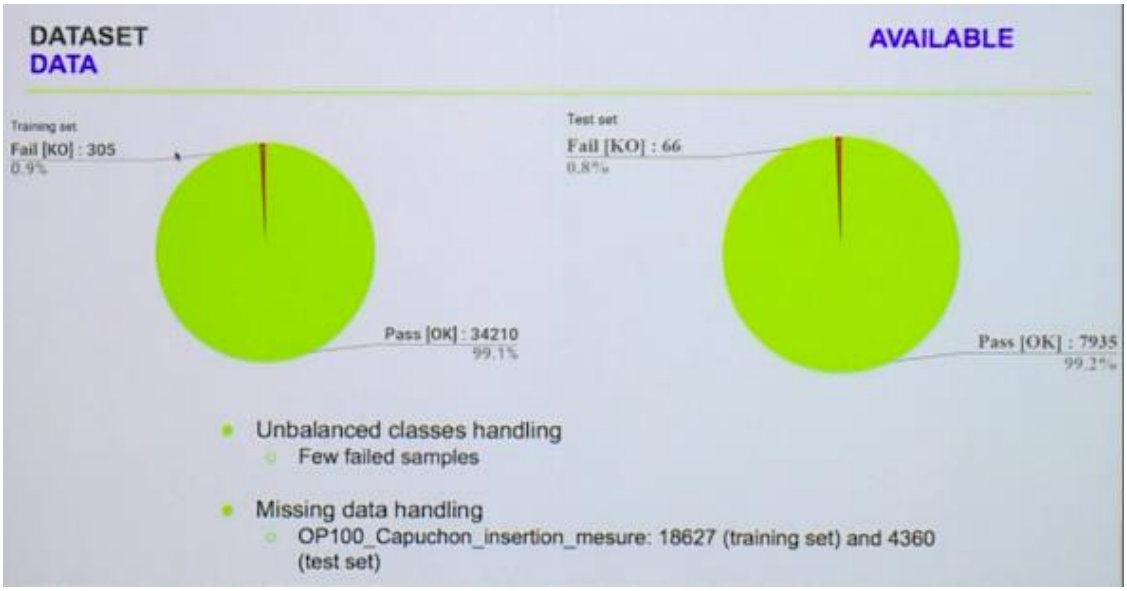
# 2 - Description et particularités du jeu de données

## a - Déséquilibre des données :

**Le jeu de données se distingue par un vrai déséquilibre au niveau de la classification du résultat** *Fig2.a*

- "démarreur en bon état / Pass [OK]" : 99.1% du jeu de données, c'est ce qu'on appellera la classe NEGATIVE de la classification, c'est la classe majoritaire (Classe_0).
- "démarreur en mauvais état / FAIL [KO]" : 0.9% du jeu de données, c'est ce qu'on appellera la classe POSITIVE de la classification, c'est la classe minoritaire (Classe_1). Les problèmes de classification déséquilibrée impliquent deux classes : une classe NEGATIVE avec la majorité des exemples et une classe POSITIVE avec une minorité d'exemples.

*Figure 2.a - Déséquilibre des données*




La particularité du déséquilibre des données est une réalité fréquente dans l'industrie et dans bien d'autres domaines : détection de fraude, détection de spam, domaine médical, ....

Le préjudice de ne pas identifier des "démarreurs en mauvais état" (classe POSITIVE - classe minoritaire) est largement plus élevé que celui de ne pas identifier des "démarreurs en bon état" (classe NEGATIVE - classe majoritaire). Il ne faut pas pour autant tolérer le rejet des "démarreurs en bon état".
**L'objectif est de détecter les "démarreurs en mauvais état" sans rejeter les "démarreurs en bon état.**

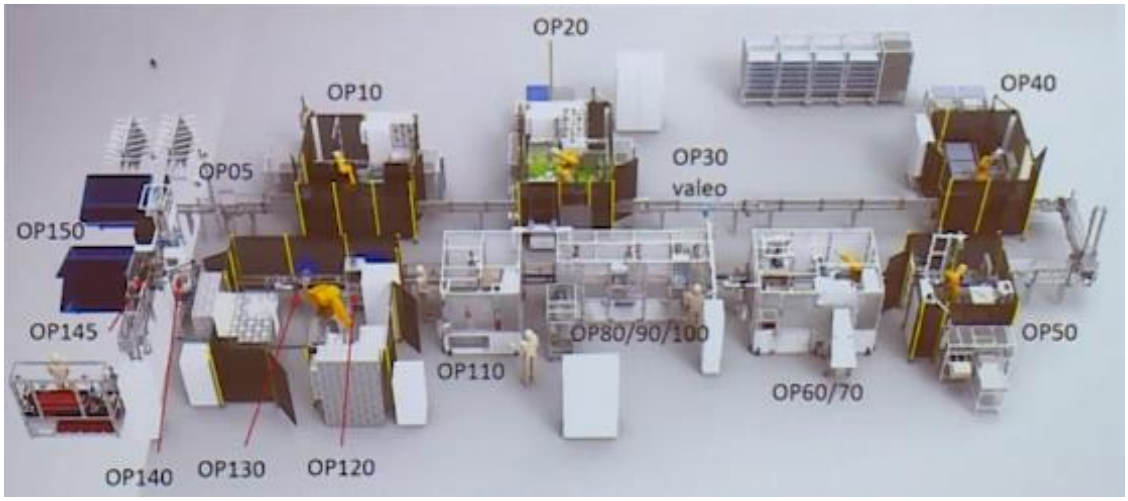
## b - Collecte des mesures des process de fabrication :

Les mesures des process *Figure 2.b.1* sont collectées sur les différentes stations d'assemblage *Figure 2.b.2* avec des capteurs connectés à des contrôleurs logiques programmables qui stockent toutes ces mesures.

*Figure 2.b.1 - Mesures des process*

| | |
|---|---|
| OP070_V_1_angle_value | V1 Valeur d'angle, |
| OP070_V_1_torque_value | V1 Valeur de couple, |
| OP070_V_2_angle_value | V2 Valeur d'angle, |
| OP070_V_2_torque_value | V2 Valeur de couple, |
| OP090_StartLinePeakForce_value | Start Line Peak Force value, |
| OP090_SnapRingMidPointForce_value | Anneau élastique Mid Point Force val, |
| OP090_SnapRingPeakForce_value | Anneau élastique Peak Force value, |
| OP090_SnapRingFinalStroke_value | Valeur finale du coup d'arrêt, |
| OP100_Capuchon_insertion_mesure | Mesure d'insertion capuchon |
| OP110_Vissage_M8_angle_value | Valeur d'angle Vissage M8, |
| OP110_Vissage_M8_torque_value | Valeur de couple Vissage M8, |
| OP120_Rodage_I_mesure_value | Rodage I mesure la valeur, |
| OP120_Rodage_U_mesure_value | Rodage U mesure value, |

*Figure 2.b.2 - Stations de mesures*




**Une autre particularité dans l'industrie est le nombre de mesures manquantes pour certains process.**
Par exemple, il manque plus de la moitié des mesures du process 'OP100_Capuchon_insertion_mesure'.

## c - Classification à la sortie:

Il s'agit de la valeur de résultat de l'OP130 au banc d'essai : Binar OP130_Resultat_Global_v.
La valeur 0 est affectée aux "démarreurs en bon état" OK (réussie) et la valeur 1 "démarreurs en mauvais état" est affectée aux échantillons KO (échoué).
Le résultat sur le banc de test effectué actuellement par Valéo se base sur un combiné de multiples tests électriques, acoustiques et vibro-acoustiques. Valéo ne dispose pas d'un modèle prédictif en place et ils ont un ROC/AUC de 0.635

## d - Mise à disposition des données d'Entrées/Sortie:

L'ensemble de données contient 34515 échantillons d'apprentissage et 8001 échantillons de test.
Les données de training sont réparties dans 2 fichiers csv:
[project-root]/data/train/traininginputs.csv : Les features (X_train)
[project-root]/data/train/trainingoutput.csv : Le résultat de la classification (y_train)

Un identifiant technique 'PROC_TRACEINFO' permet de croiser le fichier d'entrée (X_train) au fichier de sortie (y_train).
C'est un code unique donné attribué au démarreur assemblé. Exemple: I-B-XA1207672-190701-00494.

- XA1207672 est la référence.
- 190701 est la date: ici le 01 juillet de l'année 2019.
- 00494 est le code unique donné au produit, ce nombre est augmenté de 1 pour chaque nouveau produit.

On dispose aussi des données d'entrée du test (X_test) : [project-root]/data/test/testinputs.csv
Les données de prédiction (y_predict) sont générées par cette étude et sont uploadés sur la plateforme 'Data Challenge ENS' https://challengedata.ens.fr/participants/challenges/36/

# 3 - Métriques d'évaluation

## a - Recall, Precision et F1:

Dans une classification de données déséquilibrées, les erreurs de classification impactant la classe minoritaire sont considérées comme plus importantes que celles impactant la classe majoritaire.

Un modèle optimal doit répondre aux critères suivants :

- Il faut qu'il soit capable d'identifier le maximum des vrais classes POSITIVEs, ce qu'on appelle les TRUE POSITIVE.
  Ce critère appelé RECALL constitue une des métriques à utiliser. Un **RECALL élevé** implique une performance plus élevée du modèle.
- Il faut qu'il soit capable de ne pas se tromper sur les vrais classes NEGATIVEs en les qualifiant à tort de classes POSITIVEs, c'est à dire les qualifier de FALSE POSITIVE. Ce critère appelé PRECISION constitue une des métriques à utiliser. Une **PRECISION élevée** implique une performance plus élevée du modèle.

Avant de détailler davantage les métriques, évoquons les notions suivantes:

- TRUE POSITIVE ou TP : C'est une observation identifiée par le modèle comme appartenant à la classe POSITIVE, et dans la réalité elle est POSITIVE. Donc elle a été correctement prédite par le modèle.
- FALSE POSITIVE ou FP : C'est une observation identifiée par le modèle comme appartenant à la classe POSITIVE alors que dans la réalité elle est NEGATIVE. Donc elle n'a pas été correctement prédit par le modèle.
- TRUE NEGATIVE ou TN : C'est une observation identifiée par le modèle comme appartenant à la classe NEGATIVE, et dans la réalité elle est NEGATIVE. Donc elle a été correctement prédite par le modèle.
- FALSE NEGATIVE ou FN : C'est une observation identifiée par le modèle comme appartenant à la classe NEGATIVE alors que dans la réalité elle est POSITIVE. Donc elle n'a pas été correctement prédit par le modèle.

Exprimons le RECALL et la PRECISION en fonction des notions ci-dessus:

- RECALL = TP / (TP + FN)
- PRECISION = TP / (TP + FP)

Dépendance entre RECALL et PRECISION:
Un modèle qui tente de maximiser son RECALL en identifiant le maximum de POSITIF, mécaniquement, s'expose à ramener de faux POSITIF, donc de réduire sa PRECISION. Donc, comme on le constate, les 2 notions de RECALL et de PRECISION sont inter-corrélées, d'où le besoin d'une métrique de mesure qui regroupe les 2 notions.
Cette métrique de mesure correspond à la moyenne harmonique des 2 métriques précédentes, le RECALL et la PRECISION, c'est la métrique F1.

Moyenne harmonique de RECALL et PRECISION = 2 / ( 1/RECALL + 1/PRECISION)
=> F1 = **2 x PRECISON * RECALL / (PRECISION + RECALL)**

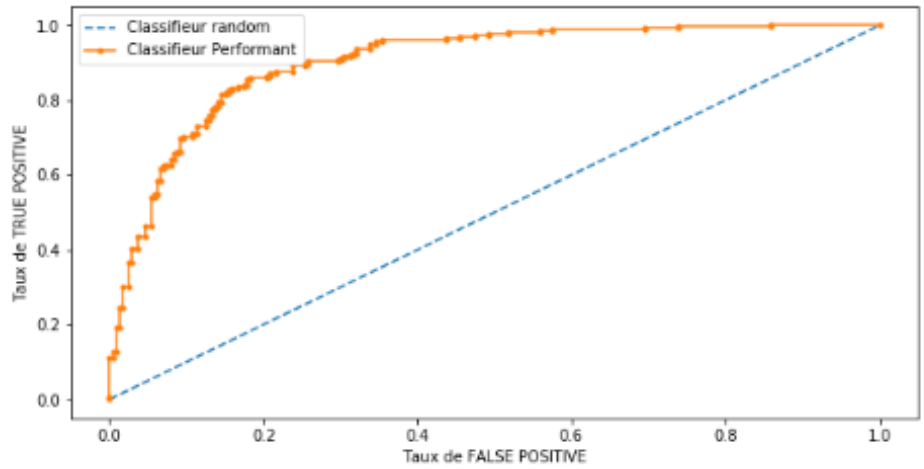
F1 est la métrique qui permet d'agréger les métriques PRECISION et RECALL en une seule.

## b - Courbe ROC et score ROC/AUC:

La courbe ROC (receiver operating characteristic) *Figure 3.b* permet de résumer les performances d'un modèle de classification binaire. L'axe des abscisses indique le taux de FALSE POSITIVE et celui des ordonnées indique le taux de TRUE POSITIVE.
Taux de TRUE POSITIVE = TP / (TP + FN)
Taux de FALSE POSITIVE = FP / (FP + TN)

Un modèle idéal agira de manière à ce que le taux des prédictions TRUE POSITIVE soit 1 (en haut du graphique) et que le taux des prédictions FALSE POSITIVE soit 0 (à gauche du graphique).
Le meilleur modèle est celui qui se rapproche le du coin supérieur gauche du graphique, coordonnée (0,1)

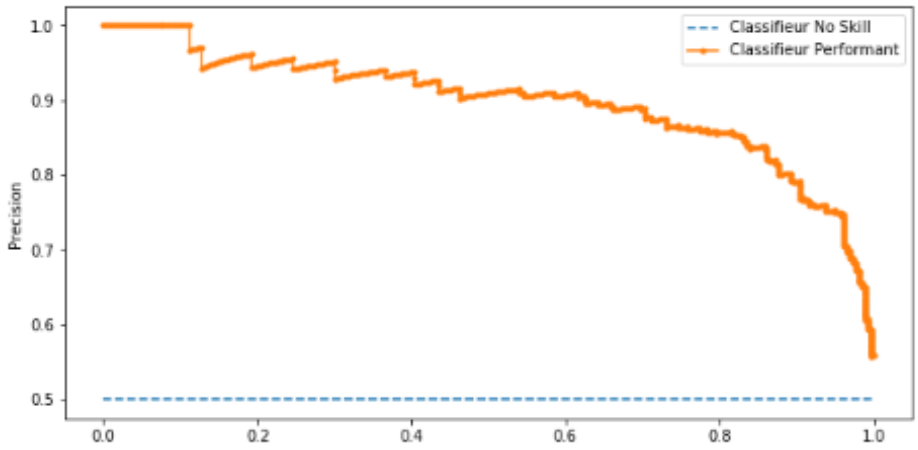
*Figure 3.b - Courbe ROC/AUC*




L'aire sous la courbe peut être calculée pour donner un score unique pour un modèle, c'est ce qu'on appelle le ROC AUC (Area under the curve).
Le score est une valeur comprise entre 0.0 et 1.0 pour un classificateur parfait, il peut être utilisé pour une classification déséquilibrée ainsi que pour une classification équilibrée.

## c - Courbe PR et score PR/AUC:

La courbe PR (PRECISION RECALL) *Figure 3.c* permet de tracer la PRECISION (ordonnée) en fonction du RECALL (abscisse)
Un modèle idéal agira de telle manière que le taux des prédictions TRUE POSITIVE soit 1 (en haut du graphique) et que le taux des prédictions FALSE POSITIVE soit 0 (à gauche du graphique).
*Figure 3.c - Courbe PR/AUC*




L'aire sous la courbe peut être calculée pour donner un score unique pour un modèle, c'est ce qu'on appelle le PR AUC (Area under the curve).
Le score est une valeur comprise entre 0.0 et 1.0 pour un classificateur parfait.

## d - Matrice de confusion:

La matrice de confusion *Figure 3.d* permet de mesurer la qualité d'un système de classification. Elle permet de recenser les classes qui sont prédites correctement et celles qui sont incorrectes avec leurs types d'erreurs.

Elle permet de représenter le nombre de TP/TN/FP et FN sous la forme d'une matrice :

*Figure 3.d - Matrice de confusion*

| | Predicted 0 | Predicted 1 |
|---|---|---|
| Actual 0 | TN | FP |
| Actual 1 | FN | TP |

# 4 - Code du projet Python et du Notebook

## a - Code Python:

Le projet est codé en suivant la programmation orientée objet de Python.

Les fonctionnalités ont été réparties dans des classes de manière à:

- Augmenter la réutilisabilité des modules
- Simplifier la maintenance
- Faciliter l'évolutivilité
- Réduire les bugs
- Rendre les traitements traçable (module logging) les traitements
- Sauvegarder certains résultats (graphes, grid/random search cv, ...)
- Rendre le projet configurable et utilisable au-delà de son environnement initial de développement.

Le code sera fourni en pièce jointe.

A cela s'ajoute un notebook qui a servi pour l'exploration des données, la génération des graphes et l'appel aux différentes méthodes d'évaluation du modèle.
Le notebook importe les modules des librairies publiques utilisées et il fait de même pour les librairies propriétaires du projet tout en activant le chargement automatique.
Par la suite, des extraits de code issus des librairies propriétaires seront utilisés dans le cadre de ce document.

## b - Chargement des données 'Training':

```
data = DfUtil.read_csv([Const.rootDataTrain() , "traininginputs.csv"])
Y_data = DfUtil.read_csv([Const.rootDataTrain(), "trainingoutput.csv"])
```

# 5 - Exploration et analyse tabulaire des données

## a - Visualisation tabulaire des données - Affichage du type 'head()':

```
data.head()
```

| | PROC_TRACEINFO | OP070_V_1_angle_value | OP090_SnapRingPeakForce_value | OP070_V_2_angle_value | OP120_Rodage_I_mesure_value |
|---|---|---|---|---|---|
| 0 | I-B-XA1207672-190429-00688 | 180.4 | 190.51 | 173.1 | 113.64 |
| 1 | I-B-XA1207672-190828-00973 | 138.7 | 147.70 | 163.5 | 109.77 |
| 2 | I-B-XA1207672-190712-03462 | 180.9 | 150.87 | 181.2 | 109.79 |
| 3 | I-B-XA1207672-190803-00051 | 173.5 | 159.56 | 151.8 | 113.25 |
| 4 | I-B-XA1207672-190508-03248 | 174.5 | 172.29 | 177.5 | 112.88 |

| OP090_SnapRingFinalStroke_value | OP110_Vissage_M8_torque_value | OP100_Capuchon_insertion_mesure | OP120_Rodage_U_mesure_value | OP070_V_1_torque_value |
|---|---|---|---|---|
| 12.04 | 12.16 | NaN | 11.97 | 6.62 |
| 12.12 | 12.19 | 0.39 | 11.97 | 6.41 |
| 11.86 | 12.24 | NaN | 11.97 | 6.62 |
| 11.82 | 12.35 | 0.39 | 11.97 | 6.62 |
| 12.07 | 12.19 | NaN | 11.97 | 6.62 |

| OP090_StartLinePeakForce_value | OP110_Vissage_M8_angle_value | OP090_SnapRingMidPointForce_val | OP070_V_2_torque_value |
|---|---|---|---|
| 26.37 | 18.8 | 109.62 | 6.60 |
| 21.03 | 18.5 | 105.48 | 6.40 |
| 25.81 | 17.5 | 100.03 | 6.61 |
| 24.62 | 15.6 | 104.94 | 6.61 |
| 29.22 | 33.6 | 99.19 | 6.61 |

Un simple affichage du type 'head()' permet de voir à quoi ressemble les données.

```
Y_data.head()
```

| | PROC_TRACEINFO | Binar OP130_Resultat_Global_v |
|---|---|---|
| 0 | I-B-XA1207672-190429-00688 | 0 |
| 1 | I-B-XA1207672-190828-00973 | 0 |
| 2 | I-B-XA1207672-190712-03462 | 0 |
| 3 | I-B-XA1207672-190803-00051 | 0 |
| 4 | I-B-XA1207672-190508-03248 | 0 |

## b - Rapport semantique des données - Affichage du type 'info()':

```
data.info()
RangeIndex: 34515 entries, 0 to 34514
Data columns (total 14 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   PROC_TRACEINFO                  34515 non-null  object
 1   OP070_V_1_angle_value           34515 non-null  float64
```

```
 2   OP090_SnapRingPeakForce_value    34515 non-null  float64
 3   OP070_V_2_angle_value            34515 non-null  float64
 4   OP120_Rodage_I_mesure_value      34515 non-null  float64
 5   OP090_SnapRingFinalStroke_value  34515 non-null  float64
 6   OP110_Vissage_M8_torque_value    34515 non-null  float64
 7   OP100_Capuchon_insertion_mesure  15888 non-null  float64
 8   OP120_Rodage_U_mesure_value      34515 non-null  float64
 9   OP070_V_1_torque_value           34515 non-null  float64
 10  OP090_StartLinePeakForce_value   34515 non-null  float64
 11  OP110_Vissage_M8_angle_value     34515 non-null  float64
 12  OP090_SnapRingMidPointForce_val  34515 non-null  float64
 13  OP070_V_2_torque_value           34515 non-null  float64
dtypes: float64(13), object(1)
```

Un affichage sémantique du type 'info()' met en évidence le type des données et le nombre des valeurs manquantes 'missing values'.

On constate que:

- Le jeu de données comporte près de 35.000 entrées, de 14 colonnes chacune.
- Toutes les features sont numériques et continues, il n'y a pas de features catégoriques
- Plus de la moitié des valeurs de la feature 7 ' OP100_Capuchon_insertion_mesure' sont manquants Plusieurs solutions sont possibles pour pallier aux valeurs manquantes :
  - Supprimer toutes les lignes ayant des valeurs manquantes : Ceci n'est pas viable dans notre cas car on risquerait de supprimer la moitié du jeu de données.
  - Supprimer la/les colonne(s) ayant des valeurs manquantes en tant que feature : Ça pourrait être une piste d'exploration pour évaluer l'impact sur la performance du modèle.
  - Utiliser une méthode d'imputation afin de lui attribuer une valeur : On va utiliser un imputer de type IterativeImputer (stratégie 'médiane') at analyser le résultat obtenu.
- PROC_TRACEINFO de type 'object' (=> String), c'est l'identifiant de ligne permettant de croiser les 'features' avec la 'target'. Cette feature porte l'horodatage de l'assemblage des démarreurs, la date sous-jacente sera extraite et utilisée dans la phase de 'features engineering'

Vérifions l'existence ou non de données dupliquées :

```
if len(data[data.duplicated()]) > 0:
    print("No. of duplicated entries: ", len(data[data.duplicated()]))
    print(data[data.duplicated(keep=False)].sort_values(by=list(data.columns)).head())
else:
    print("No duplicated entries found")
```

```
No duplicated entries found
```

## c - Données manquantes par type de 'feature':

```
data.isna().sum()
PROC_TRACEINFO                        0
OP070_V_1_angle_value                 0
OP090_SnapRingPeakForce_value         0
OP070_V_2_angle_value                 0
OP120_Rodage_I_mesure_value           0
OP090_SnapRingFinalStroke_value       0
OP110_Vissage_M8_torque_value         0
OP100_Capuchon_insertion_mesure    18627
OP120_Rodage_U_mesure_value           0
OP070_V_1_torque_value                0
OP090_StartLinePeakForce_value        0
OP110_Vissage_M8_angle_value          0
OP090_SnapRingMidPointForce_val       0
```

```
OP070_V_2_torque_value                0
dtype: int64
```

Une stratégie de base pour utiliser des ensembles de données incomplets consiste à supprimer des lignes et / ou des colonnes entières contenant des valeurs manquantes. Cependant, cela se fait au prix de la perte de données qui peuvent être précieuses (même si elles sont incomplètes). Une meilleure stratégie consiste à imputer les valeurs manquantes, c'est-à-dire à les déduire de la partie connue des données.

## d - Statistiques descriptives

```
X_data = data
X_data.sort_index(axis=1).describe().transpose()
```

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| OP070_V_1_angle_value | 34515.0 | 159.906922 | 15.662650 | 101.80 | 148.70 | 158.00 | 169.30 | 198.30 |
| OP070_V_1_torque_value | 34515.0 | 6.548403 | 0.097602 | 5.67 | 6.41 | 6.61 | 6.62 | 6.67 |
| OP070_V_2_angle_value | 34515.0 | 159.618236 | 15.091490 | 82.00 | 149.40 | 158.70 | 168.90 | 198.10 |
| OP070_V_2_torque_value | 34515.0 | 6.550867 | 0.094814 | 5.74 | 6.42 | 6.61 | 6.61 | 6.67 |
| OP090_SnapRingFinalStroke_value | 34515.0 | 11.970190 | 0.169873 | 0.00 | 11.85 | 12.04 | 12.08 | 12.19 |
| OP090_SnapRingMidPointForce_val | 34515.0 | 97.700978 | 6.837714 | 0.00 | 94.31 | 98.50 | 102.23 | 127.30 |
| OP090_SnapRingPeakForce_value | 34515.0 | 156.915055 | 11.271492 | 0.00 | 149.21 | 156.18 | 164.38 | 196.92 |
| OP090_StartLinePeakForce_value | 34515.0 | 23.630152 | 2.546341 | 0.00 | 22.28 | 23.88 | 25.29 | 43.41 |
| OP100_Capuchon_insertion_mesure | 15888.0 | 0.388173 | 0.024425 | 0.24 | 0.38 | 0.39 | 0.41 | 0.42 |
| OP110_Vissage_M8_angle_value | 34515.0 | 17.878398 | 6.785079 | 6.30 | 13.50 | 16.40 | 20.20 | 84.60 |
| OP110_Vissage_M8_torque_value | 34515.0 | 12.256785 | 0.065319 | 12.03 | 12.21 | 12.26 | 12.30 | 12.50 |
| OP120_Rodage_I_mesure_value | 34515.0 | 113.350222 | 3.528522 | 99.99 | 111.04 | 113.16 | 115.38 | 177.95 |
| OP120_Rodage_U_mesure_value | 34515.0 | 11.971027 | 0.003050 | 11.97 | 11.97 | 11.97 | 11.97 | 11.99 |

*Figure 5.d - Statistiques descriptives du jeu de données*

On constate que :

- OP070_V_2_angle_value : Outlier côté Min => Utiliser un 'robust scaler' pour réduire l'effet Outlier
- OP090_StartLinePeakForce_value, OP090_SnapRingMidPointForce_val, OP090_SnapRingPeakForce_value, OP090_SnapRingFinalStroke_value : Identification de valeurs nulle, 'min' égal à 0. Normalement ces mesures physiques ne doivent pas être nulles, le fait qu'elles soient nulles laisse penser qu'elles sont nulles à tort et par conséquent il faut les considérer comme des valeurs manquantes et les traiter par des imputers.
- OP090_StartLinePeakForce_value : Outlier côté Max => Utiliser un 'robust scaler'
- OP110_Vissage_M8_angle_value : Outlier côté Max => Utiliser un 'robust scaler'
- OP100_Capuchon_insertion_mesure: Plus de la moitié sans valeurs => Utiliser 'missing values' Imputer
- OP120_Rodage_U_mesure_value : Très petite variance, valeurs presque constante (min, Q1, Q2 et Q3) => Eventuellement ne pas tenir compte de cette feature
- OP120_Rodage_I_mesure_value : Outlier cote Max => Utiliser un 'robust scaler'

### Nombre de features égales à Nulle:

```
data.query('OP090_StartLinePeakForce_value == 0 or OP090_SnapRingMidPointForce_val == 0 or
            OP090_SnapRingPeakForce_value == 0 or OP090_SnapRingFinalStroke_value == 0')
```

| | PROC_TRACEINFO | OP070_V_1_angle_value | OP090_SnapRingPeakForce_value | OP070_V_2_angle_value | OP120_Rodage_I_mesure_value |
|---|---|---|---|---|---|
| 549 | I-B-XA1207672-190907-01953 | 137.4 | 0.0 | 166.7 | 105.51 |
| 1651 | I-B-XA1207672-190821-01367 | 178.7 | 0.0 | 170.4 | 112.95 |
| 22483 | I-B-XA1207672-190424-02168 | 166.4 | 0.0 | 171.5 | 117.26 |

Seulement 3 observations dont les valeurs des features sont égales à 0.\ A cela s'ajoute la moitié des valeurs de 'OP100_Capuchon_insertion_mesure' qui sont manquantes.

### Ratio d'observations ayant des features en outlier:

```
@classmethod
def outlier_filter(cls, df:pd.DataFrame, qtile1=0.25, qtile3=0.75) -> pd.Series:
    num_col = cls.numerical_cols(df)
    Q1 = df[num_col].quantile(qtile1)
    Q3 = df[num_col].quantile(qtile3)
    IQR = Q3 - Q1
    #
    return ((df[num_col] < (Q1 - 1.5 * IQR)) |(df[num_col] > (Q3 + 1.5 * IQR))).any(axis=1)

@classmethod
def outlier_ratio(cls, df:pd.DataFrame, qtile1=0.25, qtile3=0.75) -> float:
    return len(df[cls.outlier_filter(df, qtile1=qtile1, qtile3=qtile3)].index)/len(df.index)

outliers = DfUtil.outlier_filter(X_data)
print(f"Le ratio d'outlier est de {'%.4f' % DfUtil.outlier_ratio(X_data)}")
```

```
Le ratio d'outlier est de 0.2426
```

C’est un ratio élevé => L'éventualité de supprimer les observations n'est pas viable. D'ailleurs, on ne connait pas la raison de ces outliers :

- S’agit-il d'une erreur ?
- Ou bien est-ce une vraie donnée dont le pattern est différent ?

Pour limiter l’effet des outliers :

- Utiliser un modèle résistant aux outliers, comme les arbres
- Etudier la possibilité de transformer les données en utilisant la fonction Log.
  Lors de la visualisation graphique des données on va retrouver des distributions biaisée (skewed)

### Comparaison des valeurs statistiques entres le dataset initiale et le dataset dépourvu des outliers

```
# 1 - Le dataset dépourvu des outliers
X_data_out = X_data[~outliers]

# 2 - Creéer les 2 dataframes des valeurs statistiques descriptives
Xt =  X_data.sort_index(axis=1).describe().transpose()
Xt_out =  X_data_out.sort_index(axis=1).describe().transpose()

# 3 - Fusionner les afin de pouvoir les comparer
xt_merged = pd.merge(left=Xt, right=Xt_out, how='inner', left_on=Xt.index, right_on=Xt_out.index,
suffixes=('','-o'))
xt_merged = xt_merged.set_index(['key_0'])
xt_merged.sort_index(axis=1)
```

| key_0 | 25% | 25%-o | 50% | 50%-o | 75% | 75%-o | count | count-o | max | max-o | mean | mean-o | min | min-o |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP070_V_1_angle_value | 148.70 | 148.80 | 158.00 | 158.30 | 169.30 | 169.700 | 34515.0 | 26143.0 | 198.30 | 198.20 | 159.906922 | 160.147298 | 101.80 | 118.00 |
| OP070_V_1_torque_value | 6.41 | 6.41 | 6.61 | 6.61 | 6.62 | 6.620 | 34515.0 | 26143.0 | 6.67 | 6.67 | 6.548403 | 6.547524 | 5.67 | 6.10 |
| OP070_V_2_angle_value | 149.40 | 149.60 | 158.70 | 158.80 | 168.90 | 169.100 | 34515.0 | 26143.0 | 198.10 | 197.90 | 159.618236 | 159.847145 | 82.00 | 120.20 |
| OP070_V_2_torque_value | 6.42 | 6.42 | 6.61 | 6.61 | 6.61 | 6.610 | 34515.0 | 26143.0 | 6.67 | 6.67 | 6.550867 | 6.549834 | 5.74 | 6.15 |
| OP090_SnapRingFinalStroke_value | 11.85 | 11.85 | 12.04 | 12.04 | 12.08 | 12.080 | 34515.0 | 26143.0 | 12.19 | 12.19 | 11.970190 | 11.967375 | 0.00 | 11.67 |
| OP090_SnapRingMidPointForce_val | 94.31 | 94.91 | 98.50 | 98.82 | 102.23 | 102.365 | 34515.0 | 26143.0 | 127.30 | 114.10 | 97.700978 | 98.384817 | 0.00 | 82.43 |
| OP090_SnapRingPeakForce_value | 149.21 | 149.29 | 156.18 | 156.06 | 164.38 | 163.990 | 34515.0 | 26143.0 | 196.92 | 186.87 | 156.915055 | 156.769002 | 0.00 | 126.52 |
| OP090_StartLinePeakForce_value | 22.28 | 22.43 | 23.88 | 23.91 | 25.29 | 25.300 | 34515.0 | 26143.0 | 43.41 | 29.80 | 23.630152 | 23.789862 | 0.00 | 17.77 |
| OP100_Capuchon_insertion_mesure | 0.38 | 0.38 | 0.39 | 0.40 | 0.41 | 0.410 | 15888.0 | 12028.0 | 0.42 | 0.42 | 0.388173 | 0.392272 | 0.24 | 0.34 |
| OP110_Vissage_M8_angle_value | 13.50 | 13.40 | 16.40 | 16.10 | 20.20 | 19.400 | 34515.0 | 26143.0 | 84.60 | 30.20 | 17.878398 | 16.754443 | 6.30 | 6.30 |
| OP110_Vissage_M8_torque_value | 12.21 | 12.21 | 12.26 | 12.26 | 12.30 | 12.300 | 34515.0 | 26143.0 | 12.50 | 12.43 | 12.256785 | 12.258490 | 12.03 | 12.08 |
| OP120_Rodage_I_mesure_value | 111.04 | 111.17 | 113.16 | 113.25 | 115.38 | 115.420 | 34515.0 | 26143.0 | 177.95 | 121.88 | 113.350222 | 113.341812 | 99.99 | 104.56 |
| OP120_Rodage_U_mesure_value | 11.97 | 11.97 | 11.97 | 11.97 | 11.97 | 11.970 | 34515.0 | 26143.0 | 11.99 | 11.97 | 11.971027 | 11.970000 | 11.97 | 11.97 |

| min-o | std | std-o |
|---|---|---|
| 118.00 | 15.662650 | 1.561998e+01 |
| 6.10 | 0.097602 | 9.687328e-02 |
| 120.20 | 15.091490 | 1.497825e+01 |
| 6.15 | 0.094814 | 9.459255e-02 |
| 11.67 | 0.169873 | 1.257830e-01 |
| 82.43 | 6.837714 | 5.792290e+00 |
| 126.52 | 11.271492 | 1.104504e+01 |
| 17.77 | 2.546341 | 2.186637e+00 |
| 0.34 | 0.024425 | 1.951393e-02 |
| 6.30 | 6.785079 | 4.503697e+00 |
| 12.08 | 0.065319 | 6.206688e-02 |
| 104.56 | 3.528522 | 3.106980e+00 |
| 11.97 | 0.003050 | 3.636272e-12 |

D'une manière générale, les valeurs sont approximativement similaires, sauf pour le 'max' et le 'std' de quelques features :

- OP090_StartLinePeakForce_value, OP110_Vissage_M8_angle_value, OP120_Rodage_I_mesure_value: Le 'max' a chuté considérablement
- OP110_Vissage_M8_angle_value : Variance considérablement plus petite
  Les outliers impactent la moyenne et la variance.
  Dans le cas échéant, il n'y a pas de différence importante pour ces 2 mesures entre le jeu de données avec ou sans outliers.

## e - Distribution du jeu des données:

```
starter_count = len(Y_data[Const.Binar_OP130_Resultat_Global_v])
starter_count_ok = Y_data[Const.Binar_OP130_Resultat_Global_v].value_counts()[0]
starter_count_ko = Y_data[Const.Binar_OP130_Resultat_Global_v].value_counts()[1]
#
print(f'Nombre total des démarreurs : {starter_count}')
print(f'Nombre total des démarreurs OK => Nombre de Classes Negatives : {starter_count_ok} soit
{round(starter_count_ok/starter_count * 100,2)} % du dataset')
print(f'Nombre total des démarreurs KO => Nombre de Classes Positives : {starter_count_ko} soit
{round(starter_count_ko/starter_count * 100,2)} % du dataset')
```

```
Nombre total des démarreurs : 34515
Nombre total des démarreurs OK => Nombre de Classes Negatives : 34210 soit 99.12 % du dataset
Nombre total des démarreurs KO => Nombre de Classes Positives : 305 soit 0.88 % du dataset
```

L'étude concerne une classification de données déséquilibrées avec des valeurs de données manquantes. C'est un problème classique dans l'industrie et dans bien d'autres domaines : détection de fraude, détection de spam, domaine médical, ....

On constate qu'on est sur une classification déséquilibrée dans la répartition ce qui pose un défi pour la modélisation prédictive. La classe minoritaire est plus importante et donc le problème est plus sensible aux erreurs de classification pour la classe minoritaire que pour la classe majoritaire.

## f - Approche de la suite de l'exploration - Plan général:

- Analyse univariable des features numériques par des histogrammes / Chapitre 6
- Analyse univariable de la distribution des features numériques par la superposition des 'violons et des boîtes à moustaches' / Chapitre 6
- Analyse univariable des features catégoriques par des barres verticales représentant le nombre de chaque catégorie / Chapitre 8
- Analyse bivariables de toutes les features numériques avec le résultat de la classification / Chapitre 7:
  - Nuage de points
  - Matrice de corrélation
- Analyse bivariables de toutes les features catégoriques avec le résultat de la classification / Chapitre 8:
  - Barres verticales représentant le nombre de chaque catégorie pour chaque distribution
- Analyse de la target / Chapitre 9

# 6 - Exploration graphique univariable des données

## a - Histogramme des features après gestion des valeurs manquantes:

```
X_data = data.drop(columns= 'PROC_TRACEINFO')
tsf = transf.Transformer()
X_data_transformed = tsf.iterative_imputer_transform(X_data)
ImgUtil.save_df_hist_plot(X_data_transformed,"X_data_imputed", ncols=3, figsize=(20,15), bins=50)
plt.show()
```

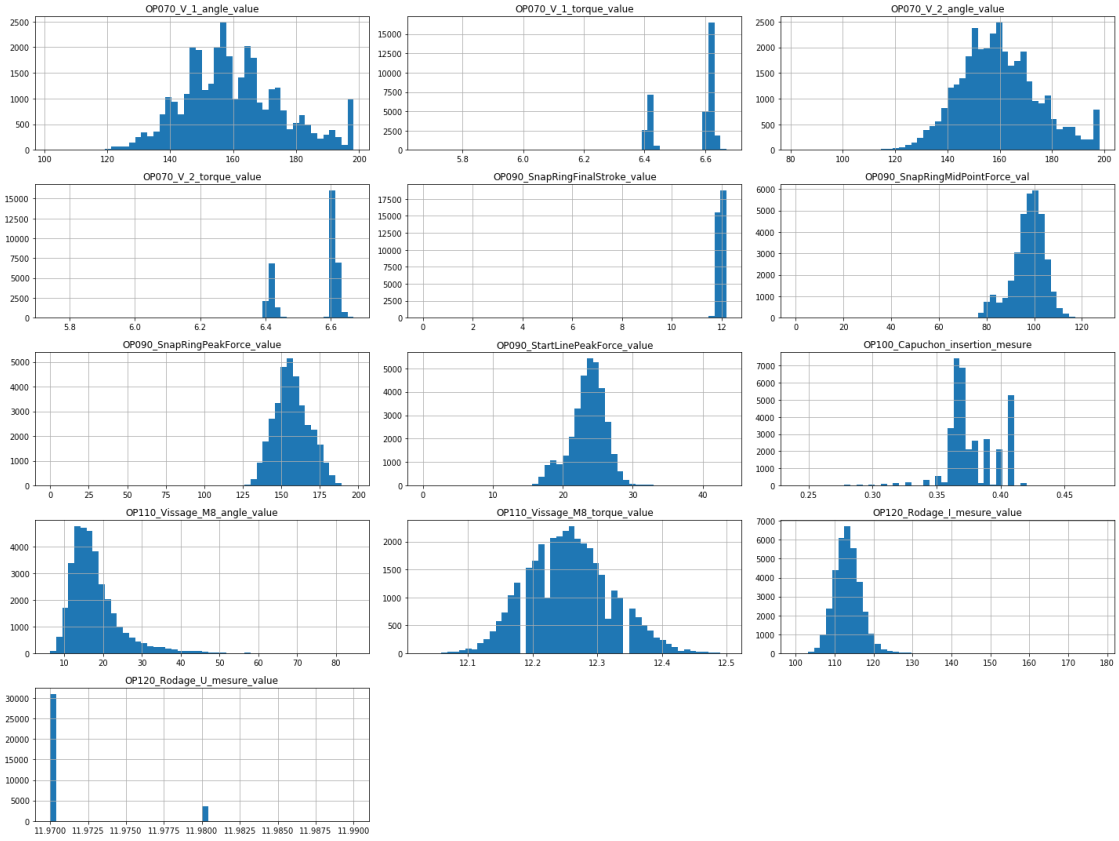



*Figure 6.a.1 - Histogramme des distributions des features*

NB:

tsf.iterative_imputer_transform(X_data) méthode de la classe 'Transformer' du module Python valeo.infrastructure.Transformer. Elle applique un imputer du type IterativeImputer(estimator=BayesianRidge, missing_values, initial_strategy = 'median')

En observant les graphes des différents features *Figure 6.a.1*, on constate que :

- La plupart des distributions sont asymétriques notamment pour :
  - OP070_V_1_angle_value : légèrement asymétrique à droite (right skewed)
  - OP090_SnapRingMidPointForce_val : asymétrique à gauche (left skewed)
  - OP090_StartLinePeakForce_value : asymétrique à droite
  - OP100_Capuchon_insertion_mesure : feature dont la moitié des mesures ont été générées par IterativeImputer avec la stratégie médiane

  - OP110_Vissage_M8_angle_value : la plus asymétrique à droite parmi l'ensemble
    Explorer le résultat d'une transformation logarithmique pour les distributions asymétriques notamment pour OP110_Vissage_M8_angle_value.
- Une valeur de plafonnement (capping value) pour :
  - OP070_V_1_angle_value
  - OP070_V_2_angle_value
- Les distributions suivantes représentent 2 familles de distribution indépendantes :
  - OP070_V_1_torque_value
  - OP070_V_2_torque_value
  - OP120_Rodage_U_mesure_value

Histogramme comparant la feature 'OP100_Capuchon_insertion_mesure' avant et après la gestion des valeurs manquantes:

```
dff_ = pd.DataFrame(X_data[Const.OP100_Capuchon_insertion_mesure])
dff_[Const.OP100_Capuchon_insertion_mesure + "_imputed_iterative_imputer"] =
X_data_transformed[Const.OP100_Capuchon_insertion_mesure]
ImgUtil.save_df_hist_plot(dff_,Const.OP100_Capuchon_insertion_mesure, ncols=2, figsize=(20,7))
plt.show()
```

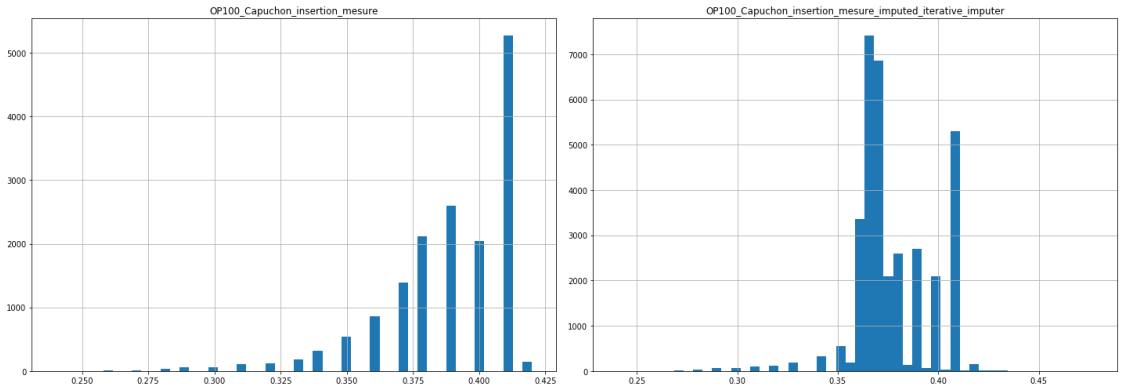



*Figure 6.a.2 - Mesures du process 'OP100_Capuchon_insertion_mesure'avec valeurs manquantes (à gauche) vs mesures avec imputation 'iterative imputer' (à droite)*

Comme la moitié des valeurs de la features OP100_Capuchon_insertion_mesure manquaient et qu'on a dû les remplacer par des imputers, essayons de comparer les 2 distributions *Figure 6.a.2*, celle d'avant et celle d'après : On constate qu'elles ne se ressemblent pas.
La distribution à droite correspond à 'OP100_Capuchon_insertion_mesure' après la gestion des valeurs manquantes par IterativeImputer. On constate que cette gestion a introduit un biais car la forme de la distribution a largement changé.

## b - Histogramme des features avec gestion des valeurs manquantes et Application de 'RobustScaler':

### Pourquoi appliquer un scaling :

Les algorithmes d'apprentissage automatique prennent en compte uniquement la magnitude des mesures, mais pas les unités de ces mesures.
Par la suite, une caractéristique exprimée en une magnitude (nombre) très élevée, peut affecter la

prévision beaucoup plus qu'une autre caractéristique tout aussi importante mais exprimée en petite magnitude (dû à l'unité de mesure).

On note que tous les algorithmes ne se comportent pas de cette façon et par la suite l'application du scaling n'est pas un prérequis pour tous les algorithmes.

Les algorithmes à base d'arbres et de Naïve Bayes ne nécessitent pas de mise à l'échelle des fonctionnalités.
Les algorithmes qui exploitent des distances ou des similitudes (par exemple sous forme de produit scalaire) entre les échantillons de données, tels que k-NN et SVM, nécessitent souvent une mise à l'échelle des fonctionnalités.

```
X_data_transformed_scaled = tsf.robust_scaler_transform(X_data_transformed)
ImgUtil.save_df_hist_plot(X_data_transformed_scaled,"X_data_imputed_robust_scaled", ncols=3, bins=50)
plt.show()
```

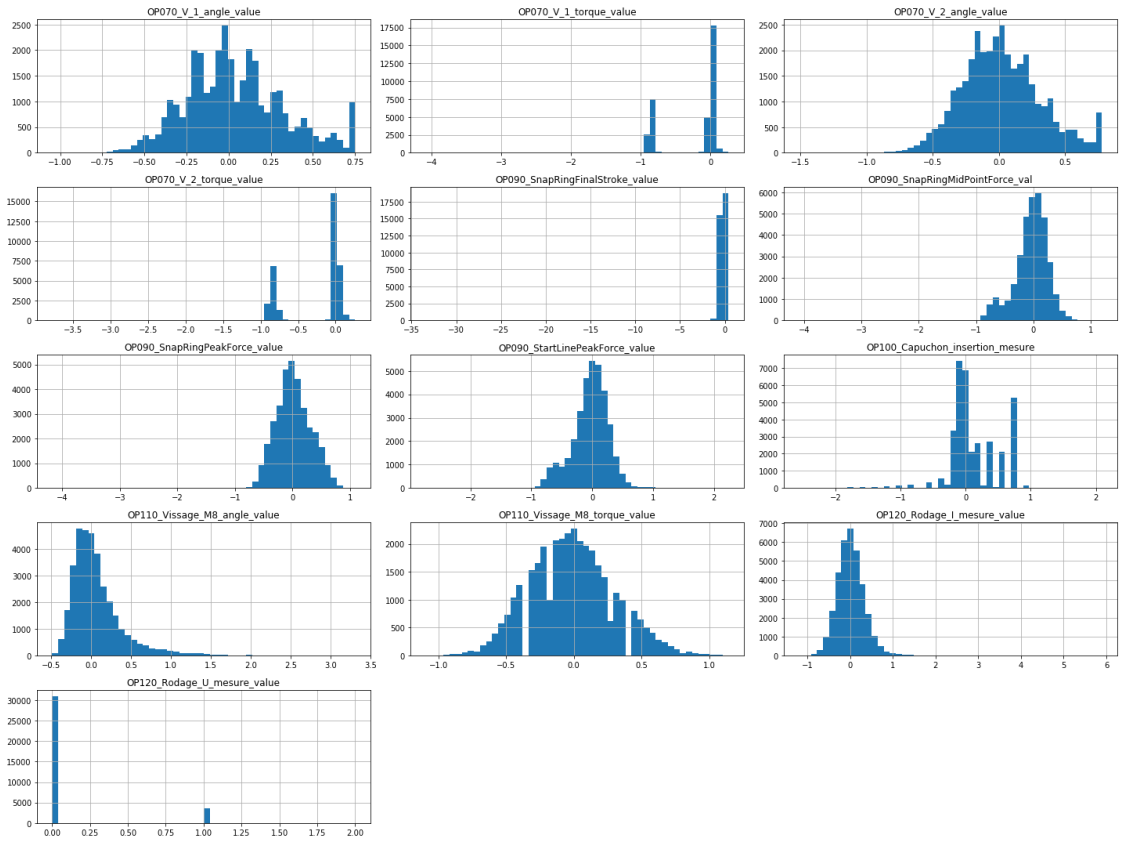



*Figure 6.b.1 - Histogramme des distributions des features après application du 'robust scaler'*

On constate que l'application du RobustScaler a rapproché davantage les distributions des distributions gaussiennes *Figure 6.b.1*, à l'exception de la distribution OP110_Vissage_M8_Angle_Value qui reste asymétrique à droite.

### c - Histogramme des features après gestion des valeurs manquantes et Transformation 'log10' où 'log2':

```
tsf = transf.Transformer()
X_data_offset_1 = X_data[[Const.OP110_Vissage_M8_angle_value]] + 1
X_data_transformed_log10 = X_data_offset_1.applymap(np.log10).rename(columns =
```

```
{Const.OP110_Vissage_M8_angle_value:Const.OP110_Vissage_M8_angle_value + '_log10'}, inplace = False)
X_data_transformed_log2 = X_data_offset_1.applymap(np.log2).rename(columns =
{Const.OP110_Vissage_M8_angle_value:Const.OP110_Vissage_M8_angle_value + '_log2'}, inplace = False)
#
X_data_transformed_log2_10 = X_data_offset_1.join(X_data_transformed_log10.join(X_data_transformed_log2))
ImgUtil.save_df_hist_plot(X_data_transformed_log2_10,"X_data_imputed_log2_log10", bins=100,  ncols=3,
figsize=(20,7))
plt.show()
```

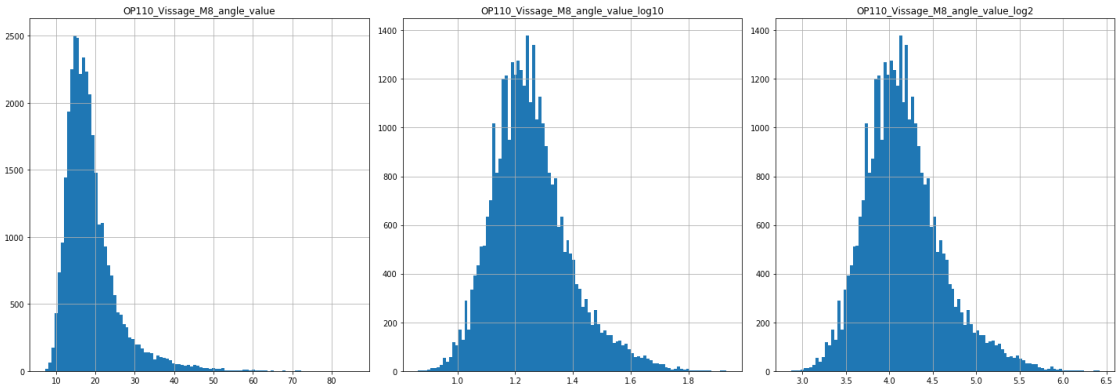



*Figure 6.c.1 - Histogrammes comparatifs de distribution de 'OP110_Vissage_M8_angle_value' features après transformation par log10 et log2*

On constate que les distributions ci-dessus *Figure 6.c.1* transformées par log10 / log2 **ressemblent plus** à des distributions Normales.
Cependant, comme dans la suite de l'étude les algorithmes de machine learning donneront des meilleurs scores sans cette transformation, alors cette transformation ne sera pas retenue.

### d - 'Violon' et 'boîte à moustaches' des features après gestion des valeurs manquantes:

Représenter la distribution d'une feature par un 'violin plot' superposé à un 'box plot' permet de:

- Visualiser la densité de la distribution d'une feature en fonction de sa valeur
- Afficher Q1, Q3 et la médiane
- Afficher les moustaches inférieur et supérieur ainsi que les outliers

Chacun des graphes suivants correspond à la superposition de 2 graphes : Celui d'une 'boîte à moustaches' et celui d'un 'violon'.

```
ImgUtil.save_df_violin_plot(X_data_transformed, 'X_data_distribution', ncols=3, figsize=(20,15))
```

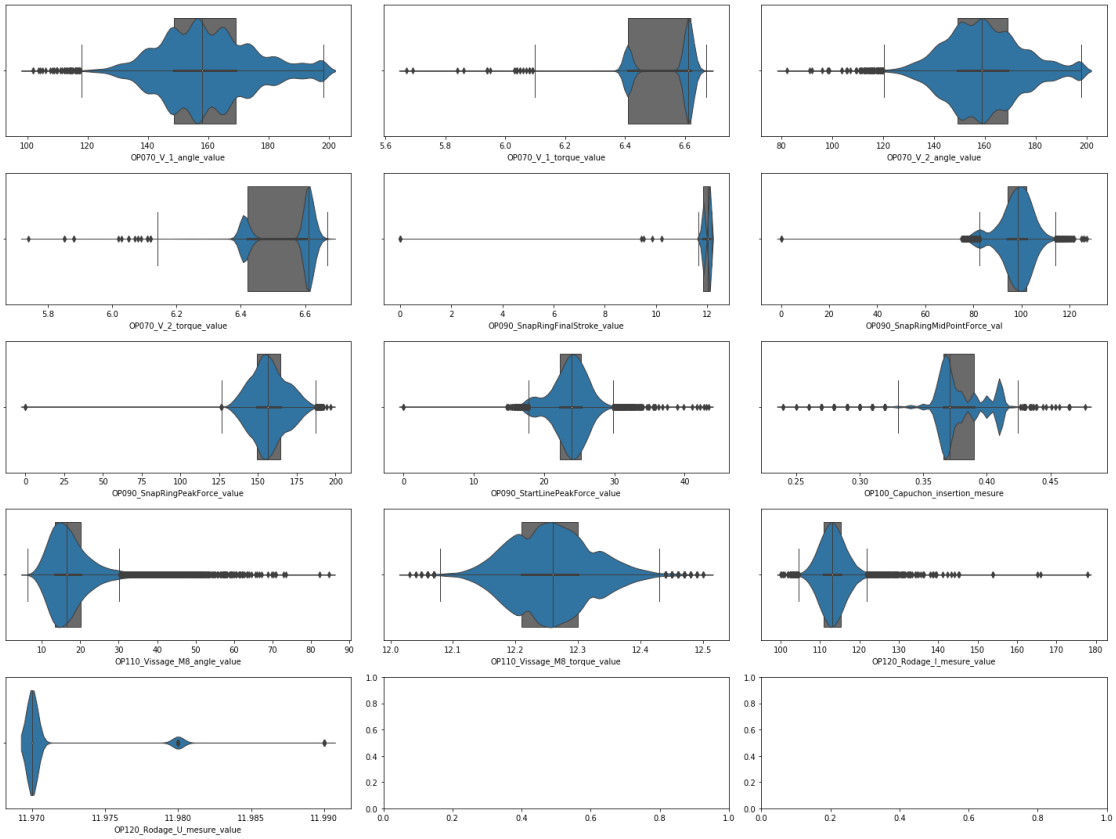


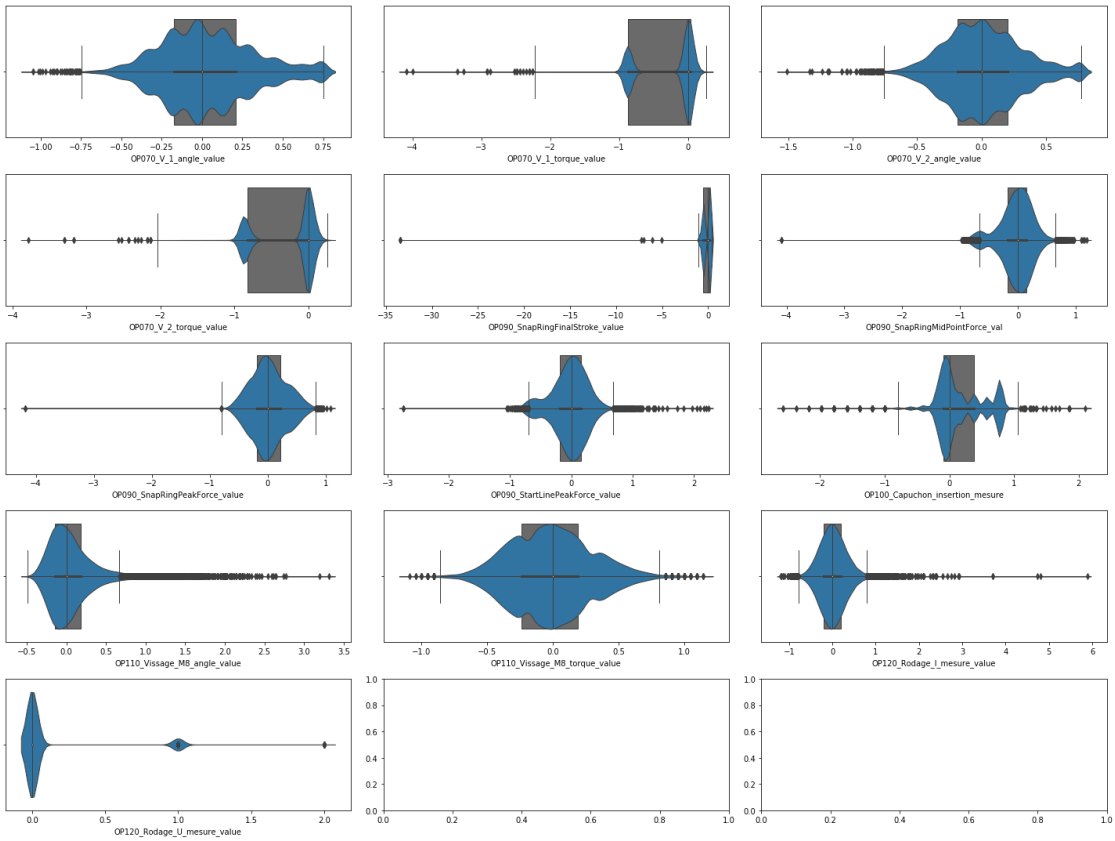

*Figure 6.d - 'Violon et boîte à moustaches' des distributions des features*

Autres observations *Figure 6.d*:

- Les densités des distributions "OP070_V_1_torque_value, OP070_V_2_torque_value et OP120_Rodage_U_mesure_value" sont bien concentrées autour de Q1 et Q3. Ceci était prévu d'après leurs histogrammes.
- Les outliers inférieures et supérieures identifiées dans la partie précédente "Statistiques descriptives" sont bien visibles dans cette famille de graphe :
  Outlier inférieur : OP070_V_1_torque_value et OP070_V_2_torque_value
  Outlier supérieur : OP090_StartLinePeakForce_value, OP110_Vissage_M8_angle_value et OP120_Rodage_U_mesure_value

## e - Violon et boîte à moustaches des features avec gestion des valeurs manquantes et application de 'RobustScaler':

```
ImgUtil.save_df_violin_plot(X_data_transformed_scaled, 'X_data_scaled_distribution_scaled', ncols=3,
figsize=(20,15))
```




*<u>Figure 6.e</u> - 'Violon et boîte à moustache' des distributions des features après application de 'robust scaler'*

A l'exception des valeurs des mesures, on constate qu'il n'y a pas de différence au niveau de la forme entre les 2 graphes <u>Figure 6.e</u> et <u>Figure 6.d</u>

# 7 - Exploration graphique bivariables 'feature/target' des données

## a - Matrice de corrélation (pearson) et heatmap après gestion des valeurs manquantes :

La corrélation des données est un moyen de comprendre la relation entre les features et la target d'un jeu de données.

```
# 1 - Charger les features et la target en les croisant:
XY_data_with_id = pd.merge(left=data, right=Y_data, how='inner', left_on=Const.PROC_TRACEINFO,
right_on=Const.PROC_TRACEINFO)
XY_data_with_id.info()

Int64Index: 34515 entries, 0 to 34514
Data columns (total 15 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   PROC_TRACEINFO                   34515 non-null  object
 1   OP070_V_1_angle_value            34515 non-null  float64
 2   OP090_SnapRingPeakForce_value    34515 non-null  float64
 3   OP070_V_2_angle_value            34515 non-null  float64
 4   OP120_Rodage_I_mesure_value      34515 non-null  float64
 5   OP090_SnapRingFinalStroke_value  34515 non-null  float64
 6   OP110_Vissage_M8_torque_value    34515 non-null  float64
 7   OP100_Capuchon_insertion_mesure  15888 non-null  float64
 8   OP120_Rodage_U_mesure_value      34515 non-null  float64
 9   OP070_V_1_torque_value           34515 non-null  float64
 10  OP090_StartLinePeakForce_value   34515 non-null  float64
 11  OP110_Vissage_M8_angle_value     34515 non-null  float64
 12  OP090_SnapRingMidPointForce_val  34515 non-null  float64
 13  OP070_V_2_torque_value           34515 non-null  float64
 14  Binar OP130_Resultat_Global_v    34515 non-null  int64
dtypes: float64(13), int64(1), object(1)


# 2 - Rajout des missing values afin d'avoir une meilleure représentation
XY_data = XY_data_with_id.drop(columns = Const.PROC_TRACEINFO)
XY_data_transformed = tsf.iterative_imputer_transform(XY_data)


# 3 - Corrélation entre la target "Binar OP130_Resultat_Global_v" et les autres attributs
corr_matrix = XY_data_transformed.corr()
corr_matrix[Const.Binar_OP130_Resultat_Global_v].sort_values(ascending=False)
Out[24]:
Binar OP130_Resultat_Global_v      1.000000
OP100_Capuchon_insertion_mesure    0.040366
OP090_SnapRingFinalStroke_value    0.015148
OP090_SnapRingMidPointForce_val    0.014273
OP090_StartLinePeakForce_value     0.010720
OP110_Vissage_M8_angle_value       0.005470
OP120_Rodage_I_mesure_value        0.003763
OP110_Vissage_M8_torque_value     -0.002984
OP070_V_2_angle_value             -0.006342
OP090_SnapRingPeakForce_value     -0.007290
OP120_Rodage_U_mesure_value       -0.010492
OP070_V_1_angle_value             -0.012793
OP070_V_1_torque_value            -0.037438
OP070_V_2_torque_value            -0.039752
Name: Binar OP130_Resultat_Global_v, dtype: float64
```

Le coefficient de corrélation varie entre -1 et 1.
Quand il est proche de 1, ça veut dire qu'il y a une corrélation positive forte.
Quand il est proche de -1, ça veut dire qu'il y a une corrélation négative forte.
Quand il est proche de 0, ça veut dire qu'il n'y a pas de **corrélation linéaire.**
**Le coefficient de corrélation mesure uniquement les corrélations linéaires.**

NB:
C'est la corrélation Pearson qui évalue la relation linéaire entre deux variables continues.
Une relation est linéaire lorsqu'un changement dans une variable est associé à un changement proportionnel dans l'autre variable.
On aurait également pu calculer la corrélation Spearman qui évalue la relation monotone entre deux variables continues ou ordinales.
Dans une relation monotone, les variables ont tendance à changer ensemble, mais pas nécessairement à un rythme constant.

```
# 4 - Dessiner la Heatmap
title = 'Corrélation matrix - features [{0}]'
ImgUtil.save_df_heatmap_plot(corr_matrix,title.format('imputed'))
```

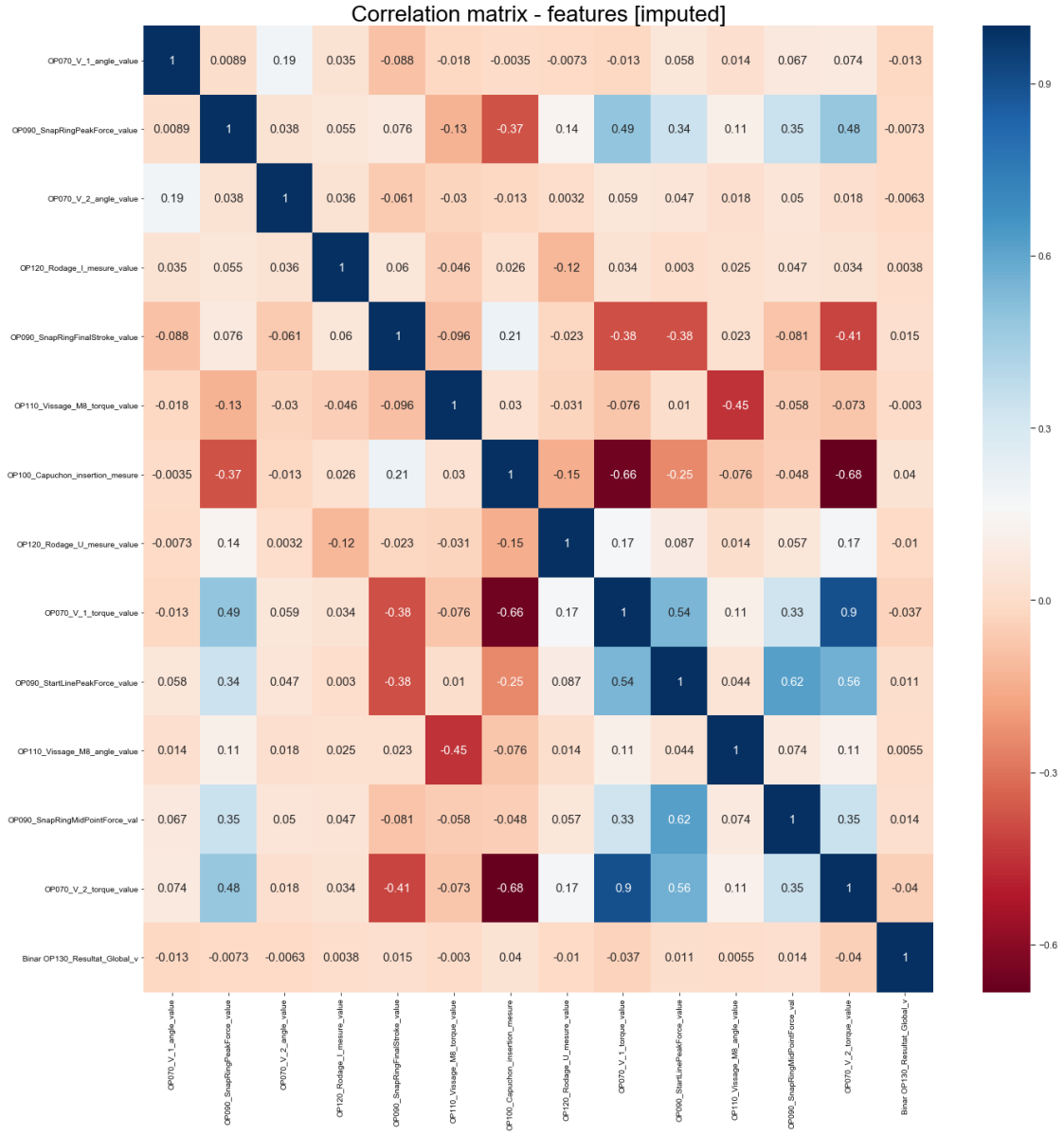



*Figure 7.a - Heatmap des features et de la target*

En observant la matrice de corrélation *Figure 7.a*, on constate :

- L'inexistence d'aucune corrélation forte entre la target 'Binar OP130_Resultat_Global_v' et n'importe quel feature.
- L'existence de corrélations positives (0.54, 0.49, 0.48, ... ) et négatives (-0.68, -0.45, -0.38, ... ) entre les autres features

## b - Matrice de corrélation et heatmap avec gestion des valeurs manquantes et rescale:

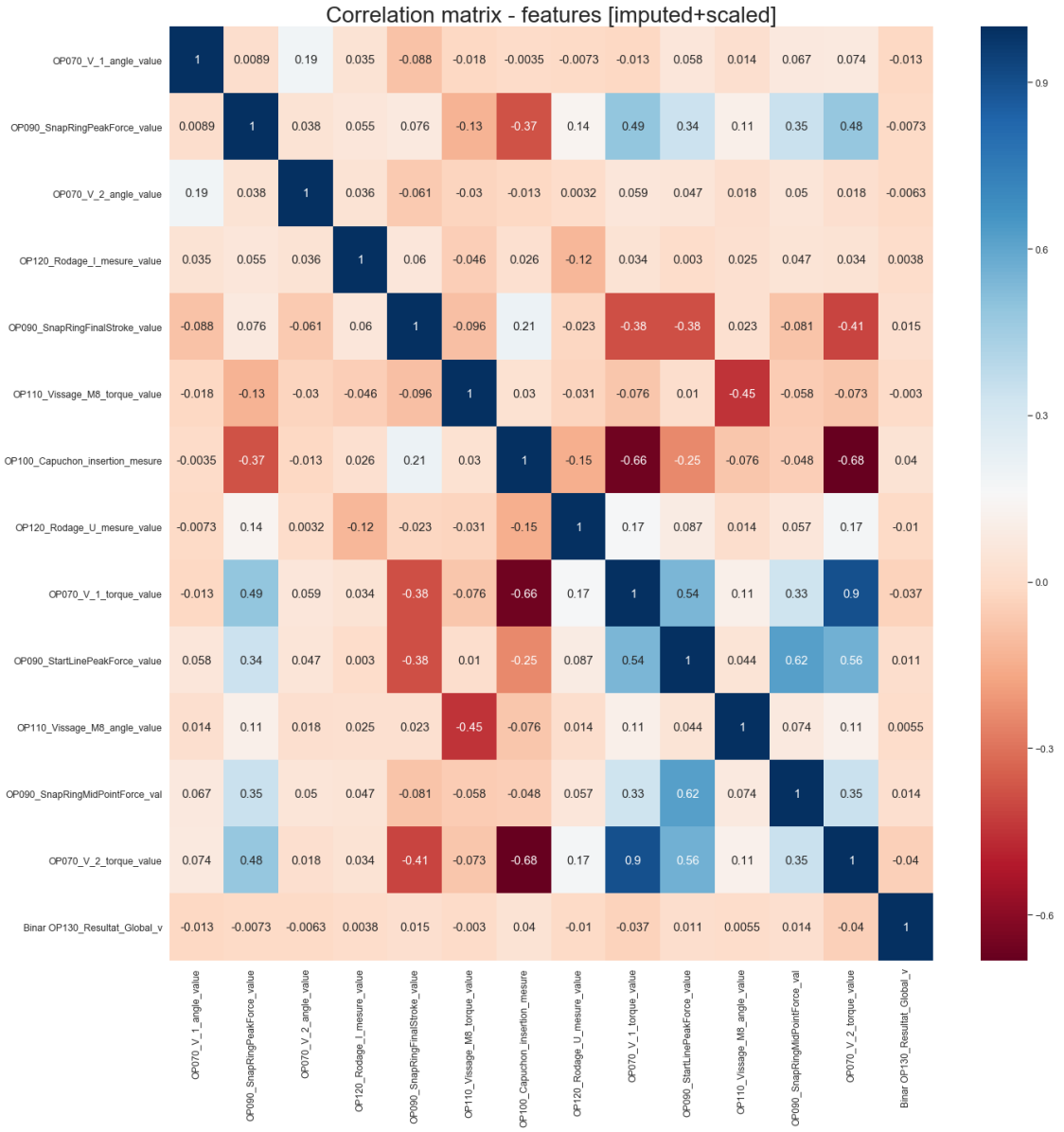
```
# 1 - Appliquer la transformation 'Robust Scaler'
XY_data_transformed_scaled =
tsf.robust_scaler_transform(XY_data_transformed.drop(columns=Const.Binar_OP130_Resultat_Global_v, axis=1))

# 2 - Rajouter la target à la dataframe
XY_data_transformed_scaled[Const.Binar_OP130_Resultat_Global_v] =
XY_data_transformed[Const.Binar_OP130_Resultat_Global_v]

# 3 - Corrélation entre la target "Binar OP130_Resultat_Global_v" et les autres attributs
corr_matrix_scaled = XY_data_transformed_scaled.corr()
corr_matrix_scaled[Const.Binar_OP130_Resultat_Global_v].sort_values(ascending=False)

Binar OP130_Resultat_Global_v      1.000000
OP100_Capuchon_insertion_mesure    0.040366
OP090_SnapRingFinalStroke_value    0.015148
OP090_SnapRingMidPointForce_val    0.014273
OP090_StartLinePeakForce_value     0.010720
OP110_Vissage_M8_angle_value       0.005470
OP120_Rodage_I_mesure_value        0.003763
OP110_Vissage_M8_torque_value     -0.002984
OP070_V_2_angle_value             -0.006342
OP090_SnapRingPeakForce_value     -0.007290
OP120_Rodage_U_mesure_value       -0.010492
OP070_V_1_angle_value             -0.012793
OP070_V_1_torque_value            -0.037438
OP070_V_2_torque_value            -0.039752
Name: Binar OP130_Resultat_Global_v, dtype: float64

# 4 - Dessiner la Heatmap
ImgUtil.save_df_heatmap_plot(corr_matrix,title.format('imputed+scaled'))
Generating 'heatmap' plot: figsize:(20, 20)
```

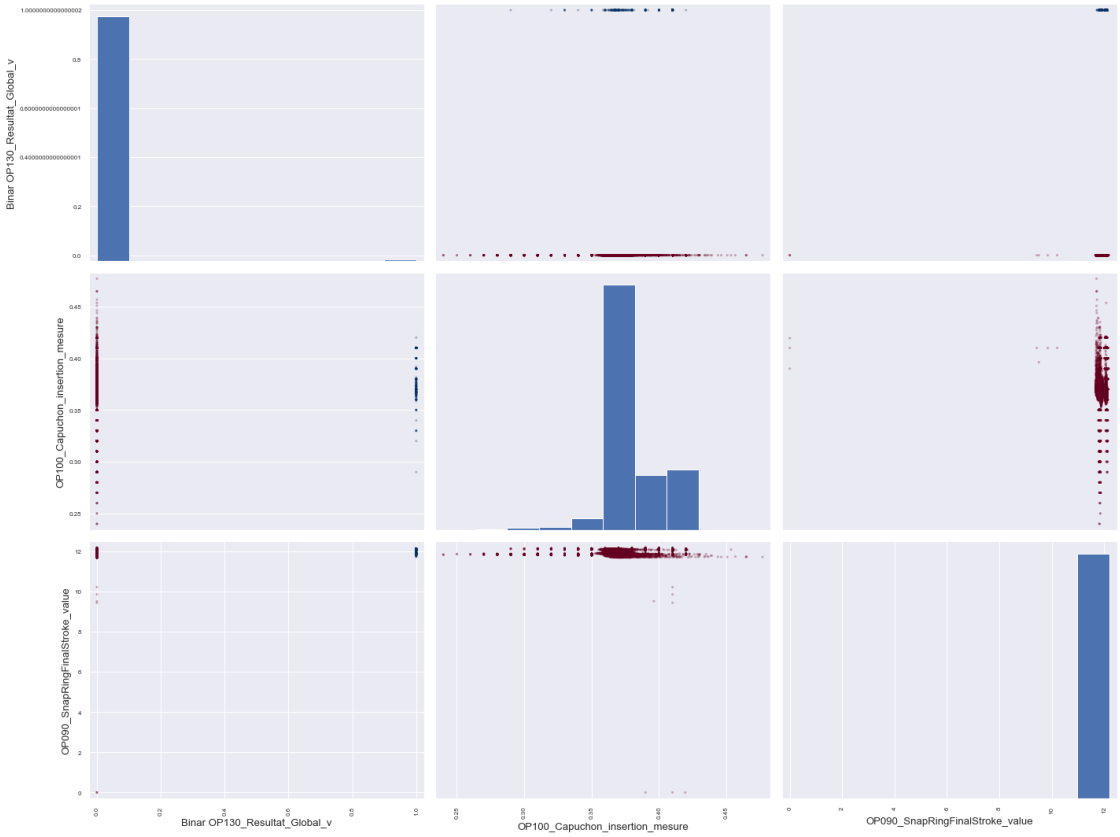



*<u>Figure 7.b</u> - Heatmap des features redimensionnés et de la target*

## c - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et les autres features:

Une autre façon de chercher une corrélation entre les différents features et target serait de tracer un nuage de points entre les différents attributs. D'une manière générale, si on connait le cas métier qu'on tente de résoudre, on se limiterait à tracer le nuage des features qui semblent impacter au plus le résultat.
Comme dans notre cas, on n'a pas une bonne connaissance métier du sujet, alors on va tracer le nuage de points pour la totalité des attributs. On les regroupe par lot de 3 attributs.

```
features = [Const.Binar_OP130_Resultat_Global_v,Const.OP100_Capuchon_insertion_mesure,
Const.OP090_SnapRingFinalStroke_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_corr_pos_1",
cfield=Const.Binar_OP130_Resultat_Global_v)
```

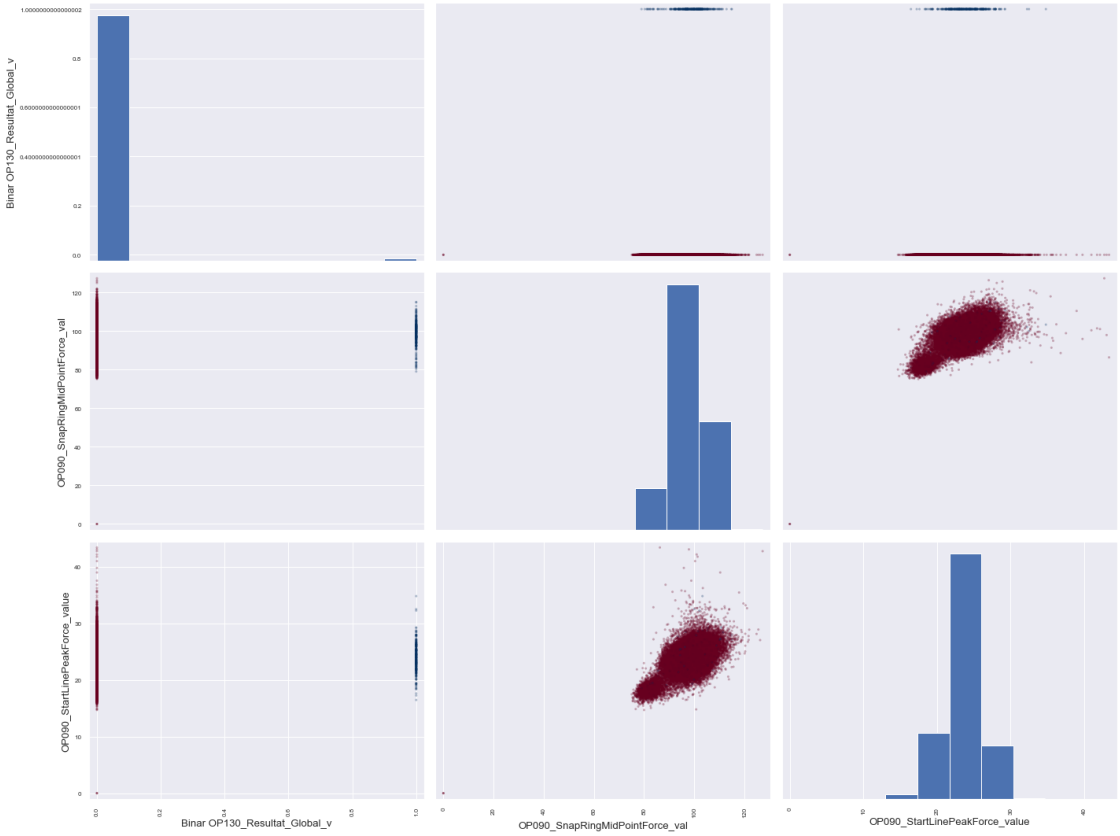



*<u>Figure 7.c.1</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et les autres features*

La diagonale allant du coin-gauche-haut au coin-droite-bas *<u>Figure 7.c.1</u>* représente des barres droites d'histogramme, ces graphes représentent le nombre d'observations d'une feature (ou de la target) en fonction des différentes valeurs que cette feature peut prendre.
Pour le reste de la grille, le nuage rouge représente les démarreurs en bon état (Classe_O majoritaire) et le bleu représente les démarreurs en mauvais état(Classe_1 minoritaire)

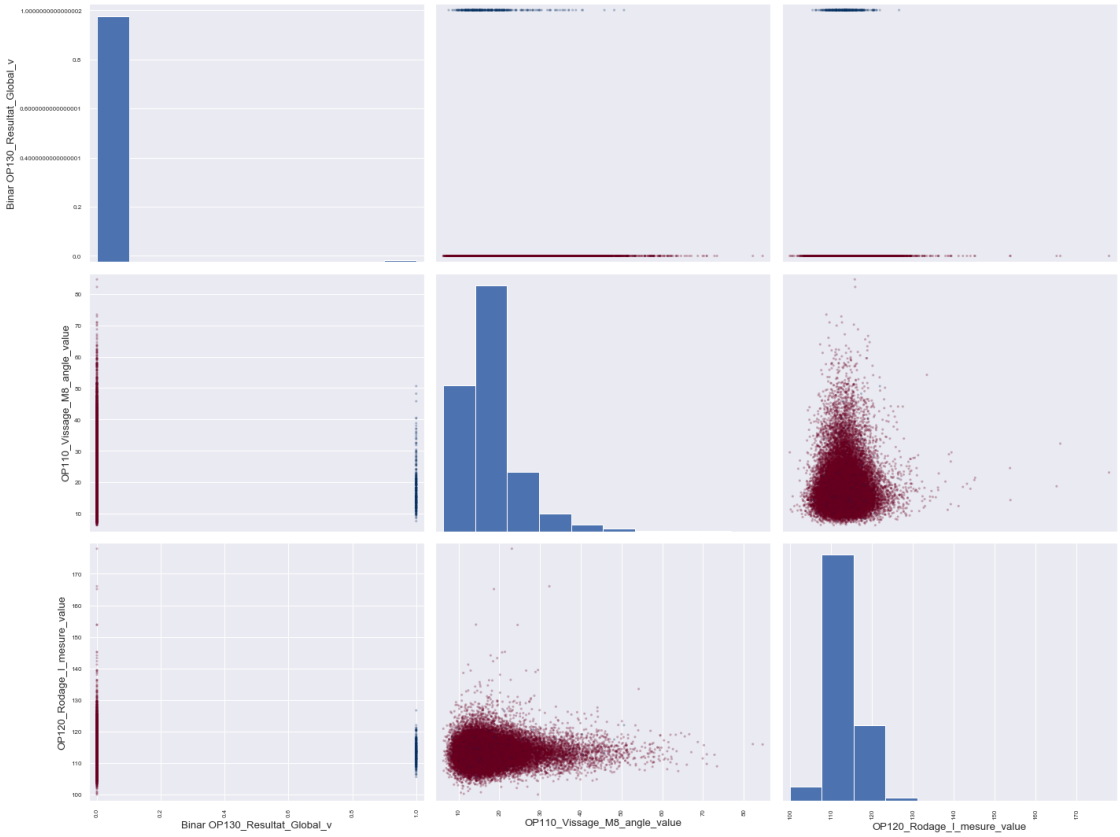
On constate que le graphe correspondant à la target (Binar OP130_Resultat_Global_v) représente une distribution fortement déséquilibrée entre les 2 valeurs '0' et '1' que peut prendre la target.

```
features = [Const.Binar_OP130_Resultat_Global_v,Const.OP090_SnapRingMidPointForce_val,
Const.OP090_StartLinePeakForce_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_corr_pos_2",
cfield=Const.Binar_OP130_Resultat_Global_v)
```



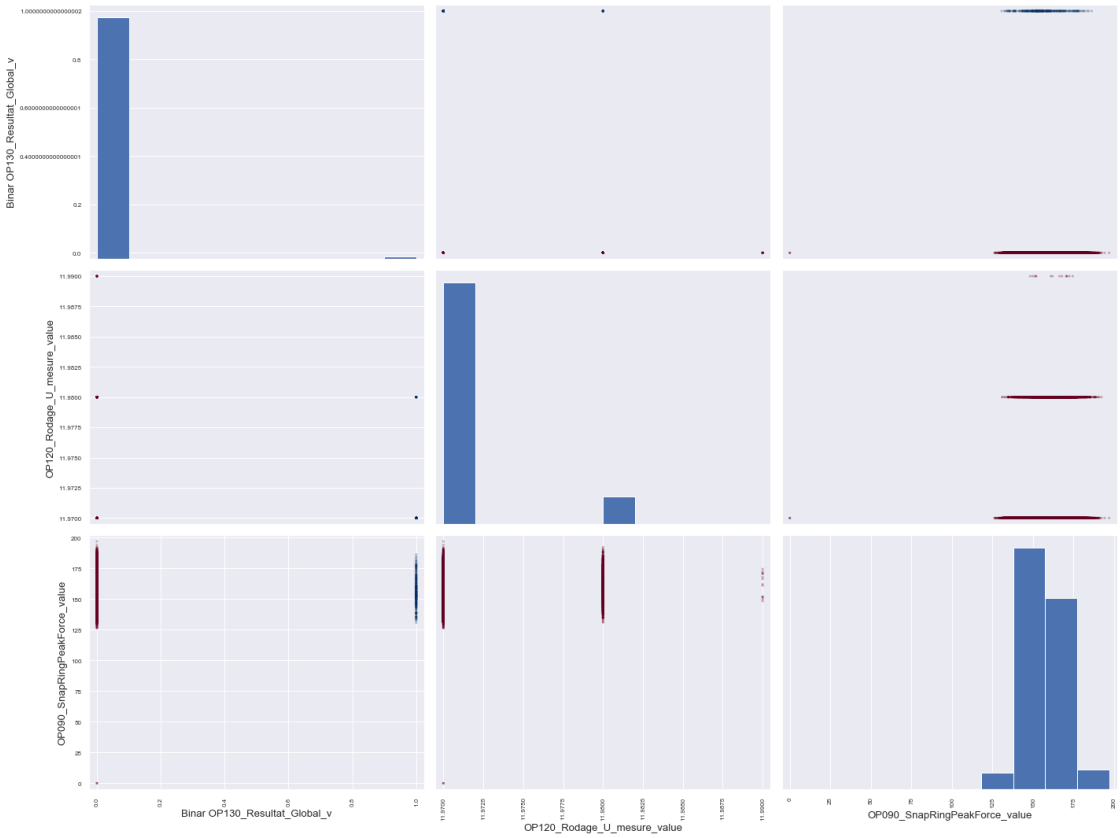

*<u>Figure 7.c.2</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et le reste des features*

```
features = [Const.Binar_OP130_Resultat_Global_v, Const.OP110_Vissage_M8_angle_value,
Const.OP120_Rodage_I_mesure_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_corr_pos_4",
cfield=Const.Binar_OP130_Resultat_Global_v)
```



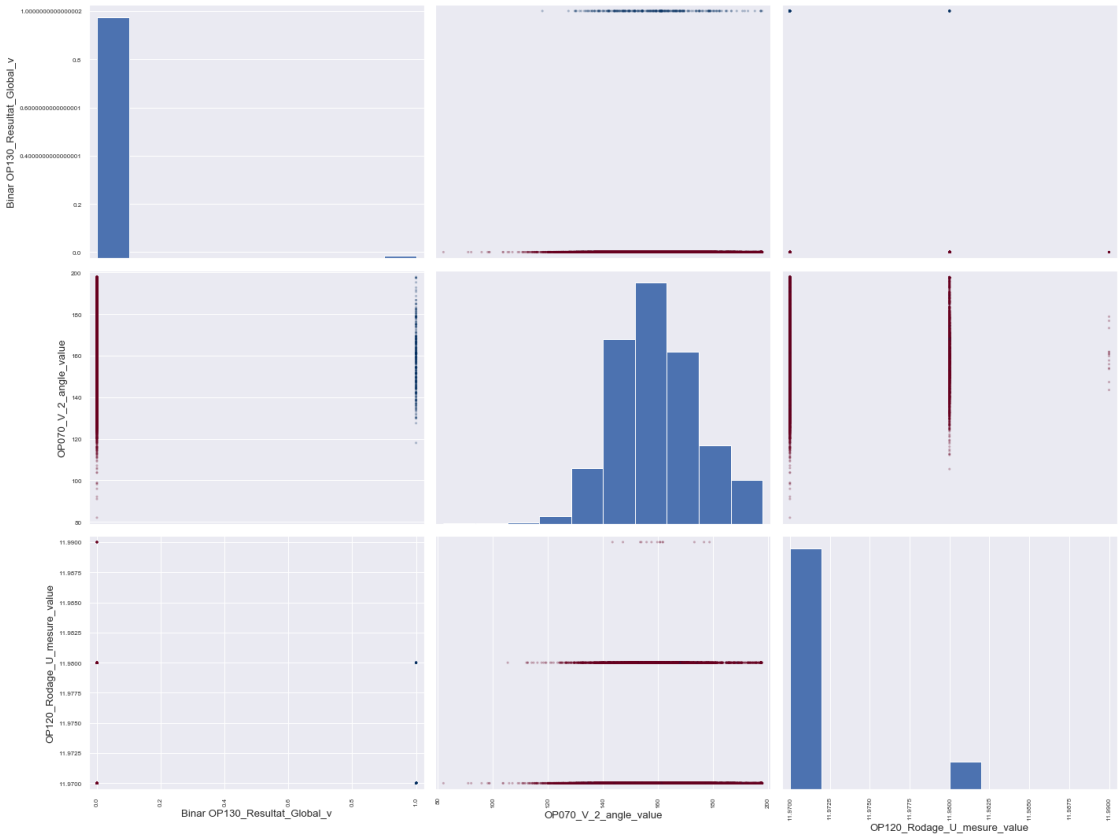

*<u>Figure 7.c.3</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et le reste des features*

```
features = [Const.Binar_OP130_Resultat_Global_v, Const.OP120_Rodage_U_mesure_value,
Const.OP090_SnapRingPeakForce_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_neg_1",
cfield=Const.Binar_OP130_Resultat_Global_v)
```



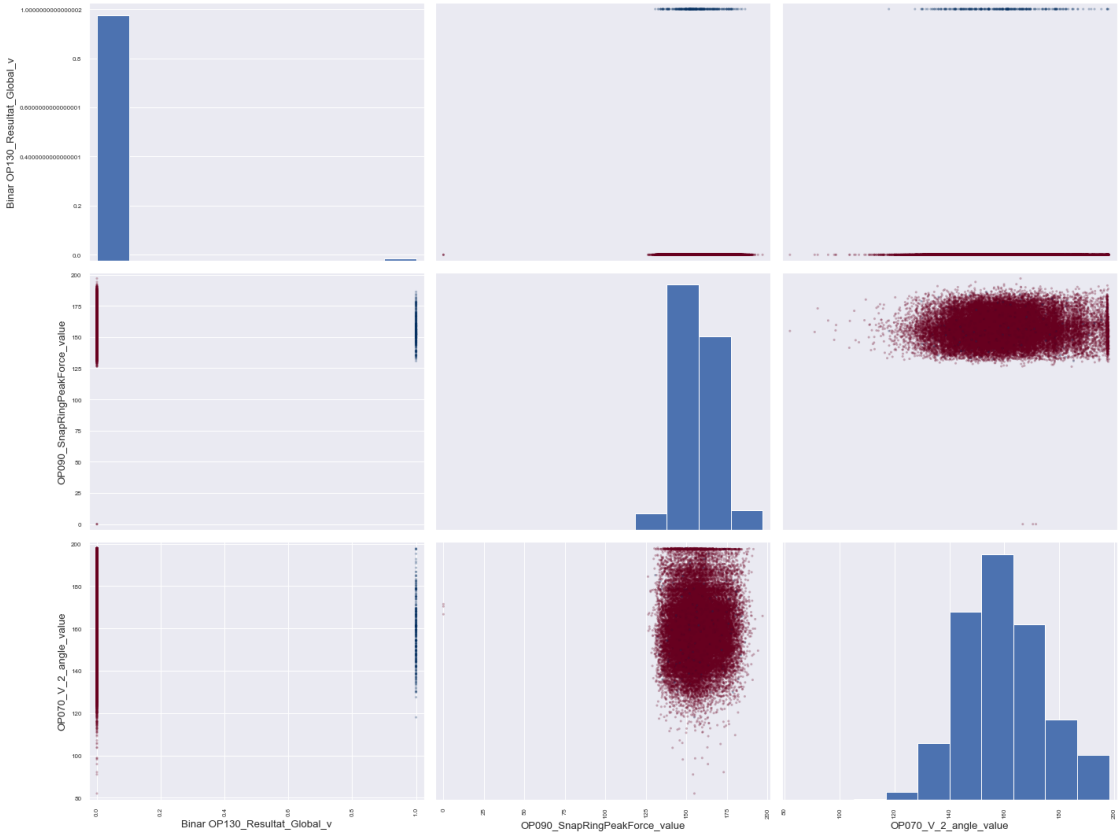

*<u>Figure 7.c.4</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et le reste des features*

```
features = [Const.Binar_OP130_Resultat_Global_v, Const.OP070_V_2_angle_value,
Const.OP120_Rodage_U_mesure_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_neg_2",
cfield=Const.Binar_OP130_Resultat_Global_v)
```




*<u>Figure 7.c.5</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et le reste des features*

```
features = [Const.Binar_OP130_Resultat_Global_v,
Const.OP090_SnapRingPeakForce_value,Const.OP070_V_2_angle_value]
ImgUtil.save_df_scatter_matrix_plot(XY_data_transformed[features], "XY_data_imputed_neg_3",
cfield=Const.Binar_OP130_Resultat_Global_v)
```




*<u>Figure 7.c.5</u> - Nuage de points entre la target 'Binar OP130_Resultat_Global_v' et le reste des features*

D'après les graphes, on constate qu'il n'y aucune relation linéaire !!

# 8 - Feature Engineering/Sélection et choix effectués

En observant les graphes des différents features, on constate que :

- La plupart des distributions sont asymétriques notamment pour :
  - OP070_V_1_angle_value : légèrement asymétrique à droite (right skewed)
  - OP090_SnapRingMidPointForce_val : asymétrique à gauche (left skewed)
  - OP090_StartLinePeakForce_value : asymétrique à droite
  - OP100_Capuchon_insertion_mesure : feature dont la moitié des mesures ont été générées par IterativeImputer avec la stratégie médiane
  - OP110_Vissage_M8_angle_value : asymétrique à droite

## a - Extraction de nouvelles features:

Le pattern de l'identifiant technique 'PROC_TRACEINFO' attribué au démarreur assemblé ressemble à 'I-B-XA1207672-190701-00494'.

- XA1207672 est la référence.
- 190701 est la date : ici le 01 juillet de l'année 2019.
- 00494 est le code unique donné au produit, ce nombre est augmenté de 1 pour chaque nouveau produit, mais il peut être remis à zéro de temps en temps et d'une manière aléatoire.

La partie ‘date’ de 'PROC_TRACEINFO' permet d'enrichir le jeu de données par des nouvelles informations utiles au problème métier qu'on résout. D'autre part, les données de Training et de Test s'étalent uniquement sur l'année 2019, donc l'année est invariante.
Pour cela, on va retenir uniquement les éléments discriminants que la date porte:

- mois de l'année : proc_month
- semaine de l'année : proc_week
- jour de la semaine : proc_weekday

Les 3 nouvelles features extraites ne représentent pas des variables quantitatives, donc elles seront considérées comme des variables qualitatives.
De plus, elles ne portent aucune notion d'ordre ou bien de 'distance'. Par exemple le 'dimanche' portant le code '6' précède le 'lundi' qui porte le code '0'.
Pour cela, ces 3 nouvelles features seront représentées par des HotEncoder lors des traitements par les algorithmes des classifieurs. Elles ont amélioré score roc_auc de 4.3%

L'identifiant 'PROC_TRACEINFO' servira aussi à faire le croisement entre le X (features) et le y (target).

Pour information, l'implémentation de cette partie au niveau du code se trouve dans la classe 'ProcDateTransformer' du module 'Preprocessor'.

**NB**
Les librairies sklearn proposent 2 classes OneHotEncoder :

- Une première classe appartenant au module ‘sklearn.preprocessing’ :
  - Cette classe opère uniquement sur des variables numériques => donc une variable catégorique chaine de caractère requiert qu'elle soit encodée en valeur numérique au préalable
  - Cette classe requiert aussi que les données traitées 'hors phase de training' ne contiennent pas de valeurs non recensées lors du training par OneHotEncoder.fit(X_train) sous réserve qu'elle génère l'erreur Python 'ValueError'. Un contournement serait possible en positionnant la propriété 'handle_unknown' à 'ignore'
- Une deuxième classe appartenant au module category_encoders (scikit-learn-contrib):
  - Cette classe n'est pas assujettie aux contraintes ci-dessus, en plus elle préserve les dataframe Pandas, ce qui permet de pouvoir vérifier les transformations en suivant les noms des colonnes.

## b - Transformation de certaines features asymétriques:

Les distributions les plus asymétriques du jeu de données sont: OP090_SnapRingMidPointForce_val, OP090_StartLinePeakForce_value et OP110_Vissage_M8_angle_value.

Différents types de transformations (log2 / log10 / sqrt) ont été effectués sur chacune de ces features. Après chaque transformation, on a évalué l'impact sur le score (roc_auc,f1) de l'estimation finale, on a constaté qu'un score plus haut est atteint avec le jeu de données initiales.
On retrouve ces transformers 'LogTransformer / SqrtTransformer' dans le module 'Preprocessor'.

## c - Graphes des nouvelles features:

```
data_with_date = DfUtil.read_csv([Const.rootDataTrain() , "traininginputs.csv"])
#
data_with_date = pp.ProcDateTransformer().transform(data_with_date)
data_with_date[[Const.proc_month, Const.proc_week, Const.proc_weekday]].head()
#
ImgUtil.save_df_bar_plot(data_with_date[[Const.proc_month, Const.proc_week, Const.proc_weekday]]
,"data_with_date", ncols=1, figsize=(20,15), use_same_figure=True)
```

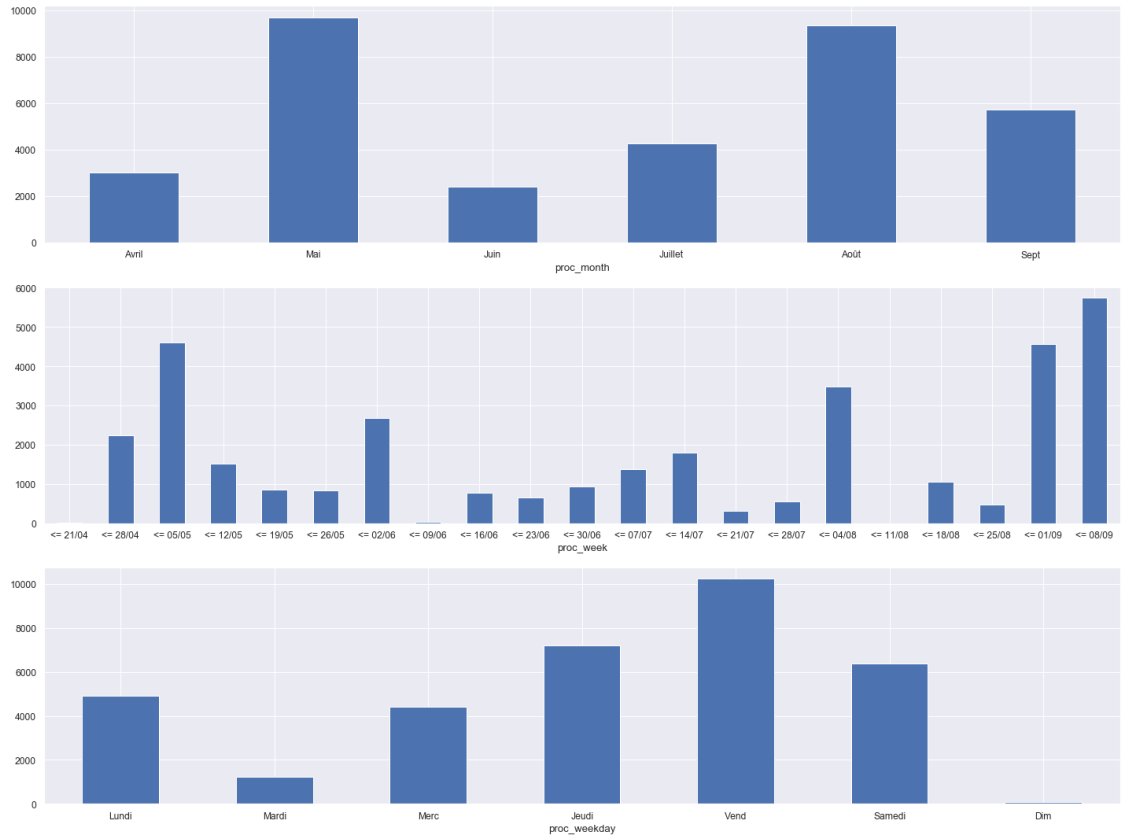



*Figure 8.c - Nouvelles features de type catégorie*

## d - Matrice de corrélation après rajout des nouvelles features:

Il est possible d'établir la nouvelle matrice de corrélation entre les nouvelles features et la target.

```
# 1 - Lecture des données.
XY_data_with_date = pd.merge(left=data, right=Y_data, how='inner', left_on=Const.PROC_TRACEINFO,
right_on=Const.PROC_TRACEINFO)
cat_date = pp.ProcDateTransformer().transform(XY_data_with_date[[Const.PROC_TRACEINFO]])

# 2 - Préparation des données
XY_data_with_date = tsf.iterative_imputer_transform( XY_data_with_date.drop(Const.PROC_TRACEINFO, axis=1)
)
cat_date_encoded = OneHotEncoder(use_cat_names=True).fit_transform(cat_date)
XY_data_with_encoded_date = pd.concat( [XY_data_with_date,cat_date_encoded] , axis=1)

# 3 - Matrice de corrélation
corr_matrix = XY_data_with_encoded_date.corr()
corr_matrix[Const.Binar_OP130_Resultat_Global_v].sort_values(ascending=False)
Out[35]:
Binar_OP130_Resultat_Global_v      1.000000
proc_week_<= 19/05                 0.060431
proc_week_<= 08/09                 0.056733
proc_month_Sept                    0.056733
OP100_Capuchon_insertion_mesure    0.040366
proc_week_<= 25/08                 0.028336
OP090_SnapRingFinalStroke_value    0.015148
OP090_SnapRingMidPointForce_val    0.014273
```

```
OP090_StartLinePeakForce_value     0.010720
proc_weekday_Merc                  0.007411
proc_weekday_Dim                   0.006770
OP110_Vissage_M8_angle_value       0.005470
proc_week_<= 12/05                 0.003831
OP120_Rodage_I_mesure_value        0.003763
proc_weekday_Jeudi                 0.003210
proc_weekday_Lundi                 0.001293
proc_weekday_Vend                  0.000958
proc_week_<= 21/04                -0.001345
proc_week_<= 18/08                -0.002349
proc_week_<= 21/07                -0.002551
proc_week_<= 09/06                -0.002785
OP110_Vissage_M8_torque_value     -0.002984
proc_month_Août                   -0.005376
proc_week_<= 01/09                -0.005741
OP070_V_2_angle_value             -0.006342
OP090_SnapRingPeakForce_value     -0.007290
proc_week_<= 28/04                -0.007310
proc_month_Mai                    -0.008133
proc_weekday_Samedi               -0.008310
proc_week_<= 23/06                -0.008554
proc_week_<= 28/07                -0.009715
proc_weekday_Mardi                -0.009736
proc_week_<= 16/06                -0.010107
OP120_Rodage_U_mesure_value       -0.010492
proc_week_<= 26/05                -0.010757
proc_week_<= 14/07                -0.010952
proc_month_Avril                  -0.011780
proc_week_<= 30/06                -0.012016
OP070_V_1_angle_value             -0.012793
proc_week_<= 07/07                -0.012958
proc_week_<= 04/08                -0.013178
proc_month_Juin                   -0.018488
proc_week_<= 02/06                -0.019278
proc_month_Juillet                -0.021399
proc_week_<= 05/05                -0.025202
OP070_V_1_torque_value            -0.037438
OP070_V_2_torque_value            -0.039752
proc_week_<= 11/08                      NaN
Name: Binar OP130_Resultat_Global_v, dtype: float64
```

- On voit bien que les features rajoutées contribuent à développer la corrélation avec la target:
  Binar OP130_Resultat_Global_v 1.000000
  proc*week* <= 19/05 0.060431
  proc*week* <= 08/09 0.056733
  proc_month_Sept 0.056733
  .....

- Dans cette matrice de corrélation, on ne peut pas voir clairement la grandeur de la corrélation entre la target et la date car la date de type catégorie a dû être transformée et décomposée en plusieurs bins.
  Il existe d'autres méthodes statistiques qui permettent d'établir la corrélation avec des variables de type catégorie, mais ceci sort du scope de cette étude.

# 9 - Analyse de la target

## a - Vérification de l'équilibre des données:

```
starter_count = len(Y_data[Const.Binar_OP130_Resultat_Global_v])
starter_count_ok = Y_data[Const.Binar_OP130_Resultat_Global_v].value_counts()[0]
starter_count_ko = Y_data[Const.Binar_OP130_Resultat_Global_v].value_counts()[1]
#
print(f'Nombre total des démarreurs : {starter_count}')
print(f'Nombre total des démarreurs OK / Classe majoritaire - Les TRUE NEGATIVE - Classe_0 :
{starter_count_ok} soit {round(starter_count_ok/starter_count * 100,2)} % du dataset')
print(f'Nombre total des démarreurs KO / Classe minoritaire - Les TRUE POSITIVE - Classe_1 :
{starter_count_ko} soit {round(starter_count_ko/starter_count * 100,2)} % du dataset')
```

```
Nombre total des démarreurs : 34515
Nombre total des démarreurs OK / Classe majoritaire - Les TRUE NEGATIVE - Classe_0 :
34210 soit 99.12 % du dataset
Nombre total des démarreurs KO / Classe minoritaire - Les TRUE POSITIVE - Classe_1 :
305 soit 0.88 % du dataset
```

## b - Distribution de la target selon ses propres classes:

```
plt.figure(figsize=(8, 10))
sns.countplot(Const.Binar_OP130_Resultat_Global_v, data=XY_data_transformed)
```

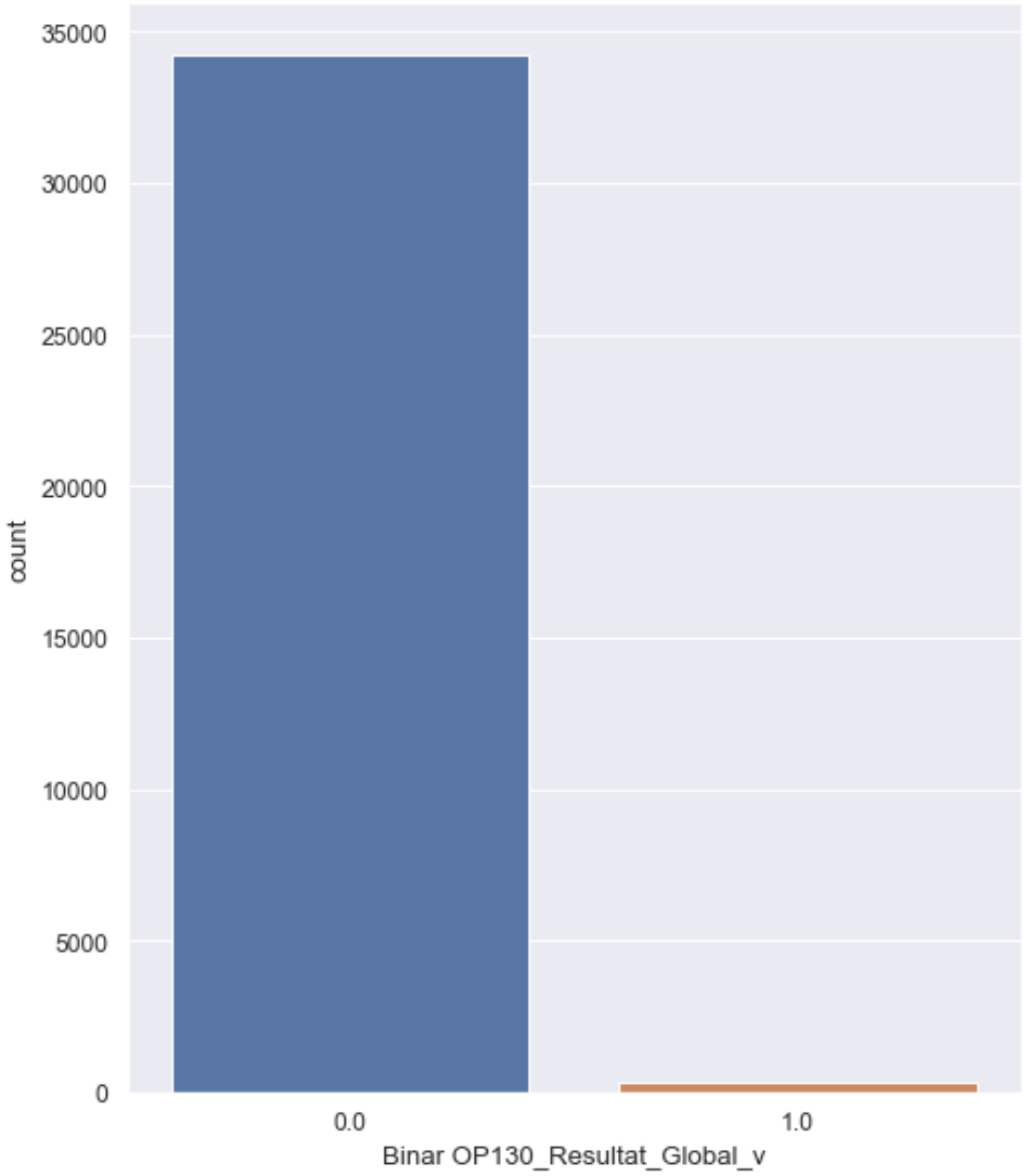



*Figure 9.b - Distribution des valeurs de la target*

On constate que le jeu de données *Figure 9.b* est fortement déséquilibré.
La presque totalité des démarreurs (99.12%) ne sont pas défectueux lors de la sortie de la ligne de production.

En utilisant cette base de données comme base pour les modèles prédictifs et pour les analyses, on pourrait obtenir beaucoup d'erreurs par des algorithmes inadaptés car ils 'supposeront' que les 'démarreurs' ne sont pas défectueux.
On cherche un modèle capable de déceler les patterns qui prédisent les défauts sur les lignes de production du démarreur.

## c - Distribution des features selon les classes de la target:

Histogramme des features de type catégorie :

```
# 1 - Lecture des données.
XY_data_with_date = pd.merge(left=data, right=Y_data, how='inner', left_on=Const.PROC_TRACEINFO,
right_on=Const.PROC_TRACEINFO)
cat_date = pp.ProcDateTransformer().transform(XY_data_with_date[[Const.PROC_TRACEINFO]])
XY_data_with_date = pd.concat( [XY_data_with_date,cat_date] , axis=1)

# 2 - Ventilation de la target selon les dates
print(f"{XY_data_with_date.groupby([Const.proc_month,
Const.Binar_OP130_Resultat_Global_v]).count()['PROC_TRACEINFO']}\n")
print(f"{XY_data_with_date.groupby([Const.proc_week,
Const.Binar_OP130_Resultat_Global_v]).count()['PROC_TRACEINFO']}\n")
print(f"{XY_data_with_date.groupby([Const.proc_weekday,
Const.Binar_OP130_Resultat_Global_v]).count()['PROC_TRACEINFO']}")
```

```
proc_month  Binar OP130_Resultat_Global_v
Avril       0                              3013
            1                                16
Mai         0                              9637
            1                                74
Juin        0                              2391
            1                                 6
Juillet     0                              4259
            1                                15
Août        0                              9286
            1                                75
Sept        0                              5624
            1                               119
Name: PROC_TRACEINFO, dtype: int64

proc_week  Binar OP130_Resultat_Global_v
<= 21/04   0                               7.0
           1                               NaN
<= 28/04   0                            2229.0
           1                              14.0
<= 05/05   0                            4590.0
           1                              13.0
<= 12/05   0                            1507.0
           1                              16.0
<= 19/05   0                             821.0
           1                              38.0
<= 26/05   0                             826.0
           1                               2.0
<= 02/06   0                            2670.0
           1                               7.0
<= 09/06   0                              30.0
           1                               NaN
<= 16/06   0                             771.0
           1                               2.0
<= 23/06   0                             650.0
           1                               2.0
<= 30/06   0                             940.0
           1                               2.0
<= 07/07   0                            1377.0
           1                               4.0
<= 14/07   0                            1786.0
           1                               8.0
```

```
<= 21/07   0                                  313.0
           1                                    2.0
<= 28/07   0                                  562.0
           1                                    1.0
<= 04/08   0                                 3471.0
           1                                   18.0
<= 11/08   0                                    NaN
           1                                    NaN
<= 18/08   0                                 1045.0
           1                                    8.0
<= 25/08   0                                  467.0
           1                                   15.0
<= 01/09   0                                 4524.0
           1                                   34.0
<= 08/09   0                                 5624.0
           1                                  119.0
Name: PROC_TRACEINFO, dtype: float64

proc_weekday  Binar OP130_Resultat_Global_v
Lundi         0                                4882
              1                                  45
Mardi         0                                1219
              1                                   5
Merc          0                                4367
              1                                  47
Jeudi         0                                7150
              1                                  68
Vend          0                               10159
              1                                  92
Samedi        0                                6339
              1                                  46
Dim           0                                  94
              1                                   2
Name: PROC_TRACEINFO, dtype: int64
```

*<u>Tableau 9.c.1</u> - Distribution des features de type catégorie selon les classes de la target*

```
# 3 - Graphe de la distribution
ImgUtil.save_df_XY_bar_plot(XY_data_with_date, "XY_imputed_with_date", ncols=1, figsize=(20,30),
y_target_name=Const.Binar_OP130_Resultat_Global_v, ts_type=Const.ts_sfix)
```

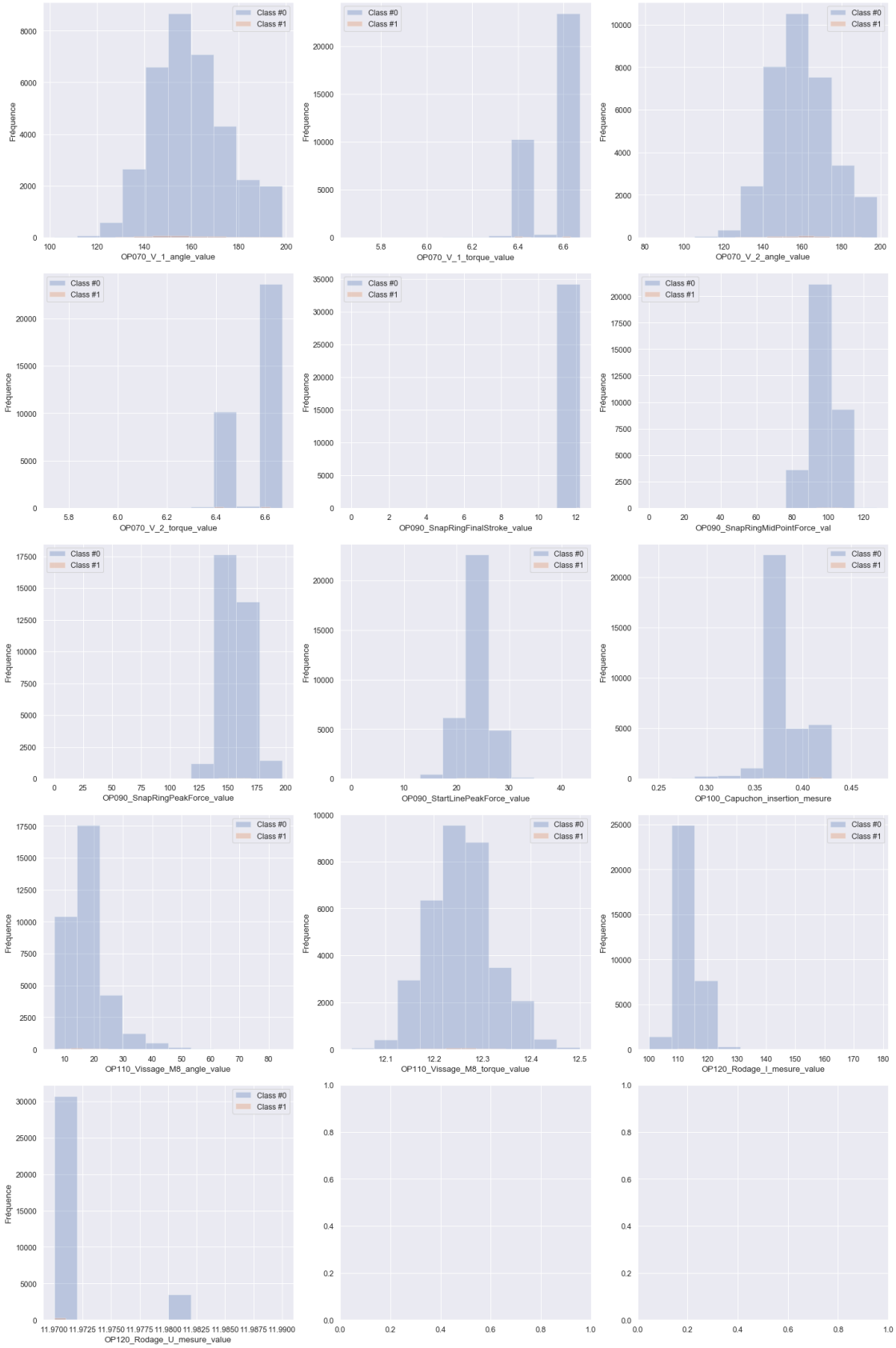

*<u>Figure 9.c.2</u> - Distribution des features de type catégorie selon les classes de la target*

Le déséquilibre des classes est bien visible dans les graphes *Figure 9.c.2* ainsi que dans le tableau *Figure 9.c.1*

Histogrammes des features de type continue:

```
ImgUtil.save_df_XY_hist_plot(XY_data_transformed, "XY_imputed", 3, figsize=(20, 30),
y_target_name=Const.Binar_OP130_Resultat_Global_v, bins=10)
Generating 'XY_hist' plot: bins=10 - figsize=(20, 30)
```

Une fois de plus, le déséquilibre des classes est bien visible dans le graphe ci dessous *Figure 9.c.3*.
Les observations appartenant à la classe minoritaire Classe_1 sont à peine visibles par rapport à la densité des observations de la classe majoritaire Classe_0.




*<u>Figure 9.c.3</u> - Distribution des features de type continu selon les classes de la target*

## d - Défis de la distribution déséquilibrée:

Le graphe suivant correspond à la distribution de la feature 'OP070_V_1_angle_value' sélectionné parmi les distributions précédentes.
On a choisi le graphe dont la classe minoritaire Classe_1 apparait le plus et on l'a agrandi.

```
ImgUtil.save_df_XY_hist_plot(XY_data_transformed[[Const.OP070_V_1_angle_value,Const.Binar_OP130_Resultat_G
lobal_v]], "XY_imputed", 1, y_target_name=Const.Binar_OP130_Resultat_Global_v,
                             figsize=(20, 15), bins=10)
```

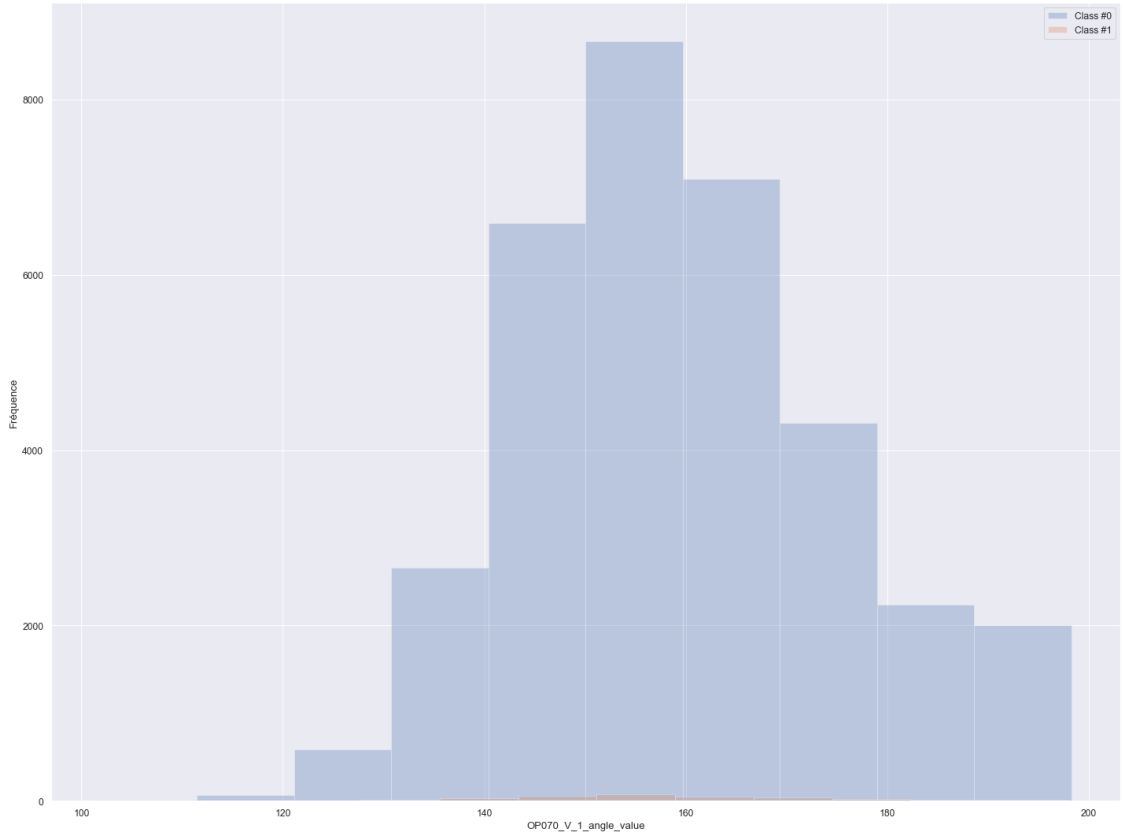



*Figure 9.d - Distribution de OP070_V_1_angle_value*

Il est bien visible dans le graphe *Figure 9.d* que la classe majoritaire (Classe_0 de couleur bleue) domine largement la classe minoritaire (Classe_1 de couleur orange).
Donc pour n'importe quelle valeur de OP070_V_1_angle_value, la probabilité de tomber sur la Classe_0 est largement supérieure à celle de tomber sur la Classe_1
=> P(Classe_0 | OP070_V_1_angle_value) > P(Classe_1 | OP070_V_1_angle_value)

Si on avait à entrainer un modèle, la précision du classifieur serait maximale en répondant toujours Classe_0 au détriment de la classe minoritaire Classe_1 qui représente l'intérêt métier.

Le classifieur idéale serait celui qui est capable d'identifier le maximum d'occurrences de la classe minoritaire 'Classe_1' tout en gardant une bonne précision globale.

Les contraintes imposées par le jeux de données que le classifieur doit gérer correspondent à:

- Une Classe_1 très minoritaire, seulement 0.88% du jeu de données.
- Les distributions Classe_0 et Classe_1 des différents features se recouvrent, tel que ceci paraît dans les graphes, d'où le défi de séparabilité.

```
filter_on_0=  XY_data_transformed[Const.Binar_OP130_Resultat_Global_v]==0
filter_on_1=  XY_data_transformed[Const.Binar_OP130_Resultat_Global_v]==1
#
OP070_on_0 = XY_data_transformed[filter_on_0][[Const.OP070_V_1_angle_value]]
OP070_on_1 = XY_data_transformed[filter_on_1][[Const.OP070_V_1_angle_value]]
#
print(f"Classe 0 -> Moyenne:{OP070_on_0.mean()['OP070_V_1_angle_value']} -
Std:{OP070_on_0.std()['OP070_V_1_angle_value']}")
print(f"Classe 1 -> Moyenne:{OP070_on_1.mean()['OP070_V_1_angle_value']} -
Std:{OP070_on_1.std()['OP070_V_1_angle_value']}")
```

```
Classe 0 -> Moyenne:159.92584039754314 - Std:15.66772829335089
Classe 1 -> Moyenne:157.78491803278683 - Std:14.95539317249524
```

Le manque de séparabilité se traduit aussi par la moyenne et la variance qui sont de même ordre de grandeur pour les 2 classes Classe_0 et Classe_1.

# 10 - Analyse de la target après un oversampling SMOTE

## a - Suréchantillonnage de la classe minoritaire de la target:

Le défi de travailler avec des ensembles de données déséquilibrés est dû au fait que la plupart des techniques d'apprentissage automatique ignorent la classe minoritaire et par conséquent ont de mauvaises performances sur cette classe, bien que généralement les performances sur la classe minoritaire soient les plus importantes.
Une approche pour remédier aux ensembles de données déséquilibrés consiste à suréchantillonner la classe minoritaire. L'approche la plus simple consiste à synthétiser de nouveaux exemples à partir des exemples existants.
Il s'agit d'un type d'augmentation de données pour la classe minoritaire et désigné sous le nom de Synthetic Minority Oversampling Technic ou SMOTE.

## b - Comment fonctionne la technique SMOTE:

SMOTE fonctionne en sélectionnant des exemples de Classe_1 (minoritaire) proches dans l'espace des features.
Puis en traçant une ligne (ou plus généralement un hyperplan) entre les exemples et en générant un échantillon correspondant à un point le long de cette ligne.

Donc:

- Un exemple aléatoire de la classe minoritaire est d'abord choisi, soit Pt_1
- Ensuite, k des voisins de la classe minoritaire les plus proches pour cet exemple sont trouvés (généralement k = 5).
- Un des k-voisins est sélectionné au hasard, soit Pt_2
- Par la suite un exemple synthétique est créé à un point sélectionné au hasard entre les deux exemples ci-dessus Pt_1 et Pt_2.

Cette procédure peut être utilisée pour créer autant d'exemples synthétiques pour la classe minoritaire que nécessaire. Cependant il est recommandé d'effectuer conjointement un sous-échantillonnage aléatoire pour réduire le nombre d'exemples dans la classe majoritaire, et d'utiliser SMOTE pour suréchantillonner la classe minoritaire afin d'équilibrer la distribution des classes.

Un inconvénient général de l'approche est que les exemples synthétiques sont créés sans tenir compte de la classe majoritaire, ce qui peut entraîner des exemples ambigus s'il y a un fort chevauchement pour les classes.

## c - Différentes techniques de SMOTE:

L'idée générale est d'appliquer un suréchantillonnage de la classe minoritaire suivi d'un sous-échantillonnage de la classe majoritaire.
Il existe plusieurs variantes de la technique de suréchantillonnage décrite ci-dessus, elles diffèrent par la stratégie utilisée pour choisir le "voisin" qui servira à la création du nouvel exemple synthétique :

- SMOTE (Synthetic Minority Oversampling TEchnique): Cette implémentation utilise les "k voisins de Classe_1 les plus proches" pour créer le nouvel exemple. La valeur fixée de k par défaut est '5' pour toutes les variantes.
- SMOTE SVM : Implémentation similaire à SMOTE mais utilise SVM pour créer le nouvel exemple.
- Borderline SMOTE_1 : En utilisant classification des "k voisins de Classe_1 les plus proches", on va choisir les instances de la Classe_1 se trouvant à côté de la frontière et qui sont mal classées. On suréchantillonne uniquement ces cas difficiles, en fournissant plus de résolution uniquement là où cela peut être nécessaire. La sélection se fait comme avec un modèle de classification du voisin le plus proche
- Borderline SMOTE_2 : Semblable à Borderline SMOTE_1 en suréchantillonnant aussi les exemples de Classe_0, la classe majoritaire. L'idée consiste à suréchantillonner les exemples à la frontière et qui sont mal classés.
- Borderline SMOTE SVM : Implémentation similaire à Borderline SMOTE, mais en utilisant l'algorithme SVM au lieu de KNN pour identifier les exemples mal classifiés autour de la frontière.
- ASADYN (Adaptive Synthetic Sampling) : C'est une autre approche qui consiste à générer des échantillons synthétiques inversement proportionnels à la densité des exemples dans la classe minoritaire => Il s'agit de générer des données pour les échantillons de classe minoritaire qui sont plus difficiles à apprendre par rapport aux échantillons minoritaires qui sont plus faciles à apprendre. Avant d'appliquer cette méthode, il faut supprimer les valeurs aberrantes afin de ne pas les suréchantillonner.
- RandomOverSampler : Suréchantillonnage d'une manière aléatoire.

### d- Techniques de combinaison du sous échantillonnage et du suréchantillonnage :

Les techniques de SMOTE sont enclines à générer des échantillons bruyants en interpolant de nouveaux points entre des valeurs marginales et des valeurs aberrantes. Ce problème peut être résolu en nettoyant l'espace résultant d'un suréchantillonnage.
Dans les chapitres 14 et 15, on utilisera des méthodes de traitements proposées par la librairie scikit learn qui combinent en pipeline un suréchantillonnage SMOTE suivi d'un sous échantillonnage TOMEK ou ENN (Edited Nearest Neighbours) pour obtenir un espace plus propre.

- TOMEK appelé aussi 'Tomek's links' consiste à :
  - Trouver 2 échantillons, '2 voisins les plus proches', dont chacun appartient à une classe différente
  - Supprimer un échantillon (l'échantillon majoritaire ou minoritaire) des 2 ou bien de supprimer les 2. Le choix par défaut étant de supprimer l'échantillon majoritaire.
- ENN consite à:
  - Pour tout 'échantillon' de la classe à sous échantillonner, trouver le 'voisin le plus proche'
  - Si ce voisin n'appartient pas à la classe du dit 'échantillon', supprimer alors l'échantillon'

### e - Suréchantillonnage et sous-échantillonnage des classes de la target:

Dans le reste de ce paragraphe, on va :

- Augmenter la classe minoritaire pour atteindre 15% du jeu de données en utilisant Borderline SMOTE 1
- Réduire la classe majoritaire de manière à ce qu'elle devienne le double de la classe minoritaire

```
# 1 - Oversample & Undersample
X = XY_data_transformed.drop(Const.Binar_OP130_Resultat_Global_v, axis=1)
y = XY_data_transformed[Const.Binar_OP130_Resultat_Global_v]

# 2 - Appliquer BorderlineSMOTE + UnderSample
over = BorderlineSMOTE(sampling_strategy=0.15, k_neighbors=5, kind='borderline-1')
#SMOTE(sampling_strategy=0.15, k_neighbors=5)
X, y = over.fit_resample(X, y)
under = RandomUnderSampler(sampling_strategy=0.5)
X, y = under.fit_resample(X, y)
oversampled_X_num = X

# 3 - Afficher
oversampled_XY = pd.concat([X,y], axis = 1)
oversampled_XY.columns = XY_data_transformed_scaled.columns
oversampled_XY.sort_index(axis=1).describe().transpose()
```

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| Binar OP130_Resultat_Global_v | 15393.0 | 0.333333 | 0.471420 | 0.00 | 0.000000 | 0.000000 | 1.000000 | 1.000000 |
| OP070_V_1_angle_value | 15393.0 | 159.184580 | 15.036566 | 101.90 | 148.800000 | 157.177522 | 166.800000 | 198.300000 |
| OP070_V_1_torque_value | 15393.0 | 6.539497 | 0.096040 | 5.67 | 6.420000 | 6.600000 | 6.612538 | 6.660000 |
| OP070_V_2_angle_value | 15393.0 | 159.363573 | 14.379885 | 103.60 | 148.500000 | 159.300000 | 168.500000 | 198.000000 |
| OP070_V_2_torque_value | 15393.0 | 6.543712 | 0.091488 | 5.85 | 6.428480 | 6.600000 | 6.610000 | 6.670000 |
| OP090_SnapRingFinalStroke_value | 15393.0 | 11.976121 | 0.153017 | 0.00 | 11.871442 | 12.027837 | 12.080000 | 12.180000 |
| OP090_SnapRingMidPointForce_val | 15393.0 | 98.321599 | 6.597832 | 0.00 | 95.420000 | 99.050000 | 102.450000 | 119.630000 |
| OP090_SnapRingPeakForce_value | 15393.0 | 156.782164 | 10.918744 | 0.00 | 149.920000 | 156.443534 | 163.190000 | 191.020000 |
| OP090_StartLinePeakForce_value | 15393.0 | 23.887627 | 2.552794 | 0.00 | 22.560000 | 24.143260 | 25.441078 | 34.930000 |
| OP100_Capuchon_insertion_mesure | 15393.0 | 0.379649 | 0.018958 | 0.24 | 0.366972 | 0.373824 | 0.392830 | 0.464825 |
| OP110_Vissage_M8_angle_value | 15393.0 | 18.816977 | 7.161284 | 6.30 | 14.000000 | 17.034965 | 21.384349 | 82.200000 |
| OP110_Vissage_M8_torque_value | 15393.0 | 12.255763 | 0.061124 | 12.05 | 12.210000 | 12.257278 | 12.290000 | 12.490000 |

*Figure 10.e - Statistiques descriptives du jeu de données après suréchantillonnage et sous-échantillonnage*

Les nouvelles valeurs statistiques *Figure 10.e*, obtenues après les 2 opérations, sont très semblables aux valeurs obtenues au début *Figure 5.d*

A noter qu'après la phase de 'Feature Engineering' le jeu de données disposera de 3 variables catégoriques extraites de la date de fabrication (PROC_TRACEINFO). La librairie Scikit Learn donne la possibilité d'appliquer aussi une technique de SMOTE sur des données catégoriques, c'est la fonctionnalité SMOTENC. A noter que cette fonctionnalité utilise le SMOTE au lieu du BorderLineSMOTE qui est plus efficace. Lors de l'évaluation du classifieur, il serait intéressant quand même de vérifier l'impact sur le score.

```
# 1 - Oversample & Undersample NC/WITH CATEGORY
X = XY_data_with_date.drop([Const.Binar_OP130_Resultat_Global_v, Const.PROC_TRACEINFO], axis=1)
cols = DfUtil.categorical_cols(X) + DfUtil.object_or_bool_cols(X)
tsf = transf.Transformer()
X = pd.concat( [X[cols], tsf.iterative_imputer_transform(X.drop(columns=cols, axis=1) )], axis = 1)
y = XY_data_with_date[Const.Binar_OP130_Resultat_Global_v]

# 2 - Appliquer SMOTENC + UnderSample
over = SMOTENC(categorical_features=[0, 1, 2], sampling_strategy=0.15, k_neighbors=5, random_state=42)
X, y = over.fit_resample(X, y)
under = RandomUnderSampler(sampling_strategy=0.5)
```

```
X, y = under.fit_resample(X, y)

# 3 - Afficher
oversampled_XY_NC = pd.concat([X,y], axis = 1)
oversampled_XY_NC.info()

Data columns (total 17 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   proc_month                       15393 non-null  category
 1   proc_week                        15393 non-null  category
 2   proc_weekday                     15393 non-null  category
 3   OP070_V_1_angle_value            15393 non-null  float64
 4   OP090_SnapRingPeakForce_value    15393 non-null  float64
 5   OP070_V_2_angle_value            15393 non-null  float64
 6   OP120_Rodage_I_mesure_value      15393 non-null  float64
 7   OP090_SnapRingFinalStroke_value  15393 non-null  float64
 8   OP110_Vissage_M8_torque_value    15393 non-null  float64
 9   OP100_Capuchon_insertion_mesure  15393 non-null  float64
 10  OP120_Rodage_U_mesure_value      15393 non-null  float64
 11  OP070_V_1_torque_value           15393 non-null  float64
 12  OP090_StartLinePeakForce_value   15393 non-null  float64
 13  OP110_Vissage_M8_angle_value     15393 non-null  float64
 14  OP090_SnapRingMidPointForce_val  15393 non-null  float64
 15  OP070_V_2_torque_value           15393 non-null  float64
 16  Binar OP130_Resultat_Global_v    15393 non-null  int64
dtypes: category(3), float64(13), int64(1)
```

## f - Nouvelle distribution équilibrée du nouveau dataset:

```
plt.figure(figsize=(5, 5))
sns.countplot(Const.Binar_OP130_Resultat_Global_v, data=oversampled_XY)
```

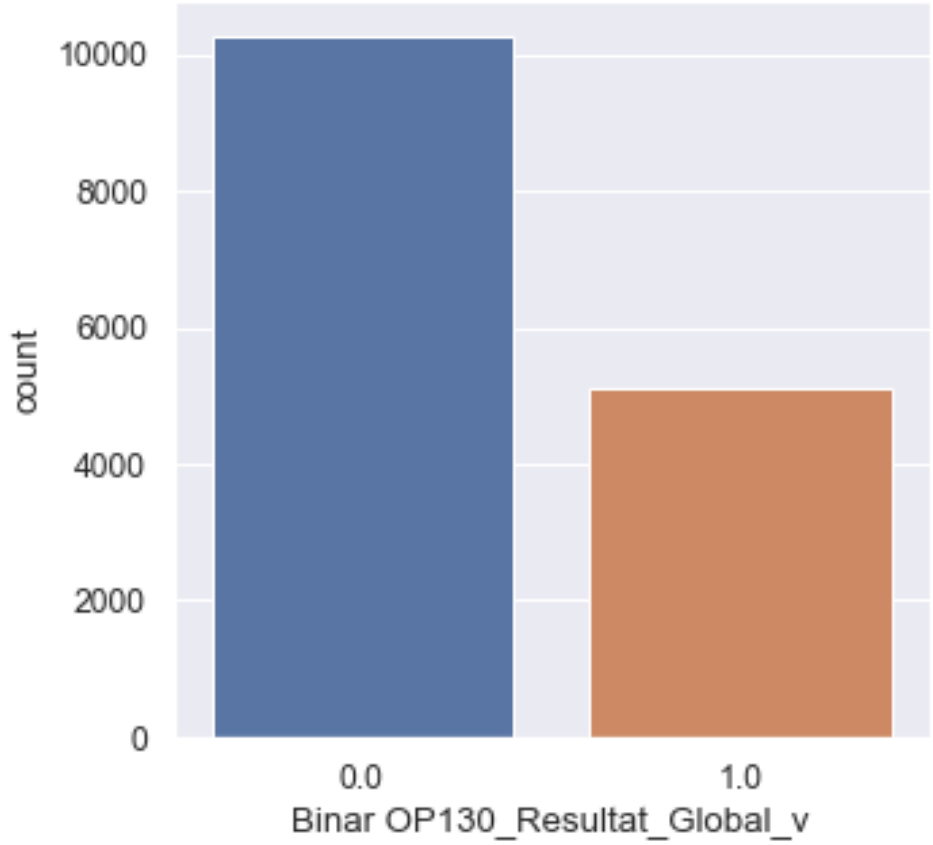



*Figure 10.f - Nouvelle distribution du dataset*

## g - Matrice de corrélation et heatmap du nouveau dataset:

```
# Corrélation entre la target "Binar OP130_Resultat_Global_v" et les autres attributs
corr_matrix_oversampled = oversampled_XY.corr()
corr_matrix_oversampled[Const.Binar_OP130_Resultat_Global_v].sort_values(ascending=False)
```

```
Binar OP130_Resultat_Global_v      1.000000
OP110_Vissage_M8_angle_value       0.190686
OP100_Capuchon_insertion_mesure    0.165045
OP090_StartLinePeakForce_value     0.132787
OP090_SnapRingMidPointForce_val    0.112513
OP120_Rodage_U_mesure_value        0.073585
OP090_SnapRingFinalStroke_value    0.067536
OP090_SnapRingPeakForce_value     -0.013993
OP070_V_2_angle_value             -0.018439
OP110_Vissage_M8_torque_value     -0.038203
OP070_V_1_angle_value             -0.055074
OP120_Rodage_I_mesure_value       -0.057177
OP070_V_2_torque_value            -0.123385
```

```
OP070_V_1_torque_value            -0.143730
Name: Binar OP130_Resultat_Global_v, dtype: float64
```

- On constate l'accroissement de la corrélation après l'opération 'suréchantillonnage classe minoritaire + sous échantillonnage classe majoritaire' par rapport à la corrélation initiale calculée en '7-a'.
- A noter que l'accroissement de la corrélation est induit par les 2 actions réunis 'suréchantillonnage et sous échantillonnage '.
- Se limiter uniquement à une action de 'suréchantillonnage ' n'aurait pas eu cet effet escompté.

```
# 4 - Dessiner la Heatmap
ImgUtil.save_df_heatmap_plot(corr_matrix_oversampled,title.format('oversampled_imputed+scaled'))
```

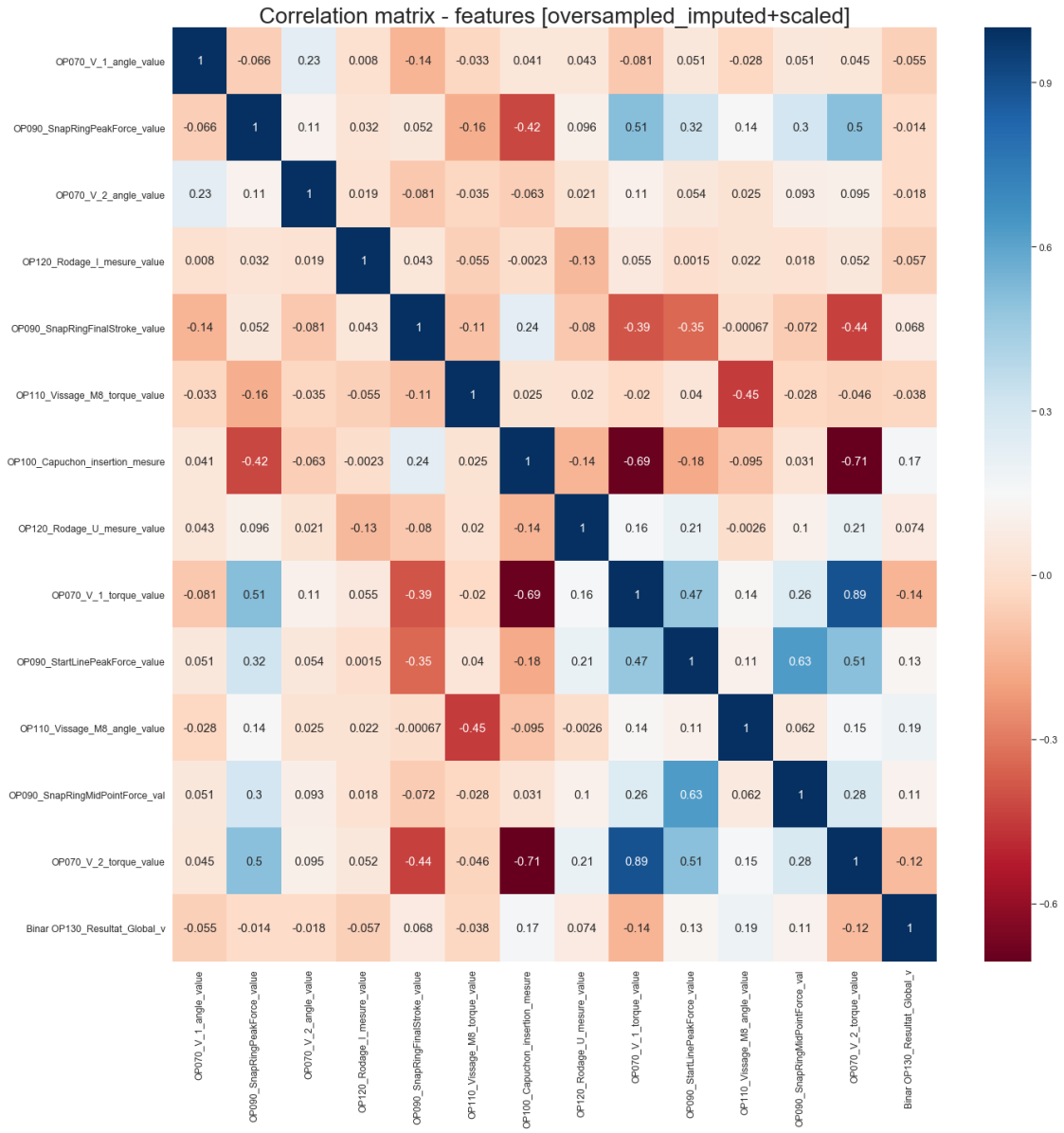



*Figure 10.g - Heatmap de la nouveau dataset*

## h - Violon et boîte à moustaches des features du nouveau dataset:

```
ImgUtil.save_df_XY_violin_plot(oversampled_XY, Const.Binar_OP130_Resultat_Global_v,
'XY_oversampled_data_distribution', 3)
```

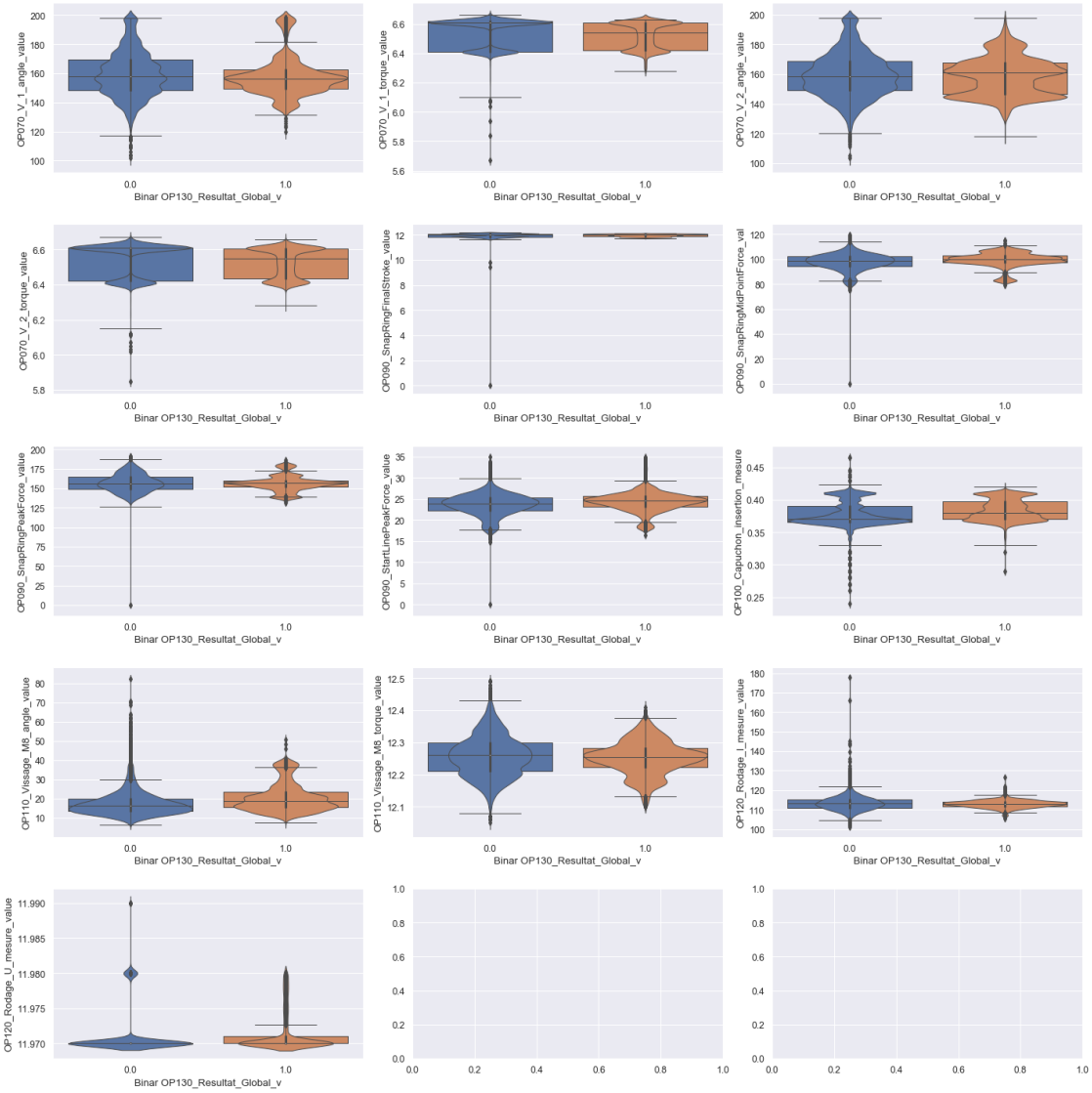



*Figure 10.h - Violon et boîte à moustaches des features du nouveau dataset*

## i - Ratio d'observations ayant des features en outlier du nouveau dataset:

```
Q1 = oversampled_X_num.quantile(0.25)
Q3 = oversampled_X_num.quantile(0.75)
IQR = Q3 - Q1
#
```

```
outliers = ((oversampled_X_num < (Q1 - 1.5 * IQR)) |(oversampled_X_num > (Q3 + 1.5 * IQR))).any(axis=1)
print(f"Le ratio d'outlier est de {len(oversampled_X_num[outliers].index)/len(oversampled_X_num.index)}")
Le ratio d'outlier est de 0.3602286753719223
```

Le ratio d'outlier est considérable => On ne peut pas supprimer les observations correspondantes.

# 11 - Approche suivie pour l'apprentissage, la validation et le test des modèles

## a - Approche suivie pour l'évaluation des modèles:

Pour chacun des algorithmes utilisés, la trame suivante a été suivie en phase d'étude :

- Chargement du jeu de données de l'ENS et initialisation des features (X) et de la target (y).
- Compréhension du fonctionnement l'algorithme et de ses caractéristiques.
  Compréhension de l'impact des hyperparamètres sur le fonctionnement de l'algorithme.
- Passage à l'étape exploration et expérimentation en évaluant les performances.
  Trois types de train/test seront utilisés :
  - Un simple train_test_split() : qui permet de donner une évaluation rapide du modèle
  - Une cross_validation() : qui permet de donner une évaluation plus précises du modèle et une orientation sur l'impact des hyperparamètres.
    Le jeu de données est décomposé en K-folds qui seront traités d'une manière itérative tel que: k-1 folds serviront pour le training et 1 fold pour la validation. Cette étape permet aussi d'évaluer la capacité du modèle à généraliser sur les données.

    - La moyenne des performances mesurées (roc_auc, precision, recall, f1) donne une idée sur la performance globale attendue
    - La variance des performances mesurées donne une idée de comment le modèle va réagir en production sur des nouvelles données qu'il n'a jamais vues.
    - Ceci fournit aussi des éléments de comparaison entre les différentes méthodes utilisées.
  - Dans le code Python fourni on retrouve la possibilité d'utiliser les méthodes BayesSearchCV(), GridSearchCV() et RandomizedSearch() qui sont des méthodes de recherches et de cross validation pour retrouver les paramètres optimaux du modèle.
    - BayesSearchCV : Méthode intelligente bayésienne pour retrouver les paramètres optimaux. On lui définit l'espace de recherche dans lequel elle va retrouver les paramètres optimaux. De même on lui spécifie le nombre maximal d'itérations (de recherches) à effectuer.
    - GridSearchCV : On définit l'espace de recherche à partir duquel elle va retrouver les paramètres optimaux
      Elle teste la totalité des combinaisons définies afin d'identifier la plus performante. Elle est consommatrice de temps machine.
    - RandomizedSearchCV : On définit l'espace de recherche à partir duquel elle va choisir 'selon une distribution' des paramètres, les tester sur le modèle et rendre la combinaison la plus performante.
      De même on lui spécifie le nombre maximal d'itérations à effectuer, donc on peut limiter ainsi la consommation de temps machine. Cependant la combinaison de paramètres obtenue n'est pas forcément optimale car le choix des paramètres initiaux n'est pas basé sur une méthode d'optimisation, comme c'est le cas pour BayesSearhCV.
      NB:
      Le résultat de test de ces 3 méthodes pour les différents algorithmes, ne figurera

pas dans de ce document car le temps d'exécution est relativement long sur ma machine à ressources limitées.
On se limitera à un cas représentatif et on référencera au niveau du code la mise en œuvre de ces 3 méthodes de 'recherche de paramètres + cross validation'.

- Mesures de performance: roc_auc, recall, precision, f1
- Graphes de ROC_AUC et PR_AUC

Pour information:

- Cette approche fera l'objet des chapitres 12 à 23 avec une vue synthétique récapitulative au chapitre 24.
- Les indicateurs retenus pour mesurer la performance ont été détaillés au chapitre 3.

## b - Echantillonnage Train/Validation/Test:

Les points principaux à considérer lors de l'échantillonnage se résument par :

- Il faut que les jeux de données Train / Validation et Test soient totalement distincts et étanches de manière à ce que le modèle ne s'entraine pas sur le jeu de validation ou de test. Si le modèle s'entraine sur le jeu de données du test, alors il va apprendre les spécificités de ce jeu de données et par conséquent les mesures de performances seront excellentes mais elles ne reflèteront pas la réalité.
  Les mesures seront biaisées car on aurait testé le modèle sur des données qu'il connait déjà.
- Il faut que l'échantillonnage de chacun des jeux de données soit représentatif des vraies données que le modèle sera emmené à prédire, sinon on risque d'introduire un biais d'échantillonnage.

Un échantillonnage représentatif, appelé aussi échantillonnage stratifié, consiste à :

- Identifier le critère (la feature) le plus déterminant pour aboutir au résultat,
- Et par la suite générer un échantillonnage de manière à ce que les jeux de données Train, Validation et Test aient la même proportion de distribution de ce critère déterminant. La librairie de Scikit Learn fournit des méthodes pour effectuer des échantillonnage s stratifiés

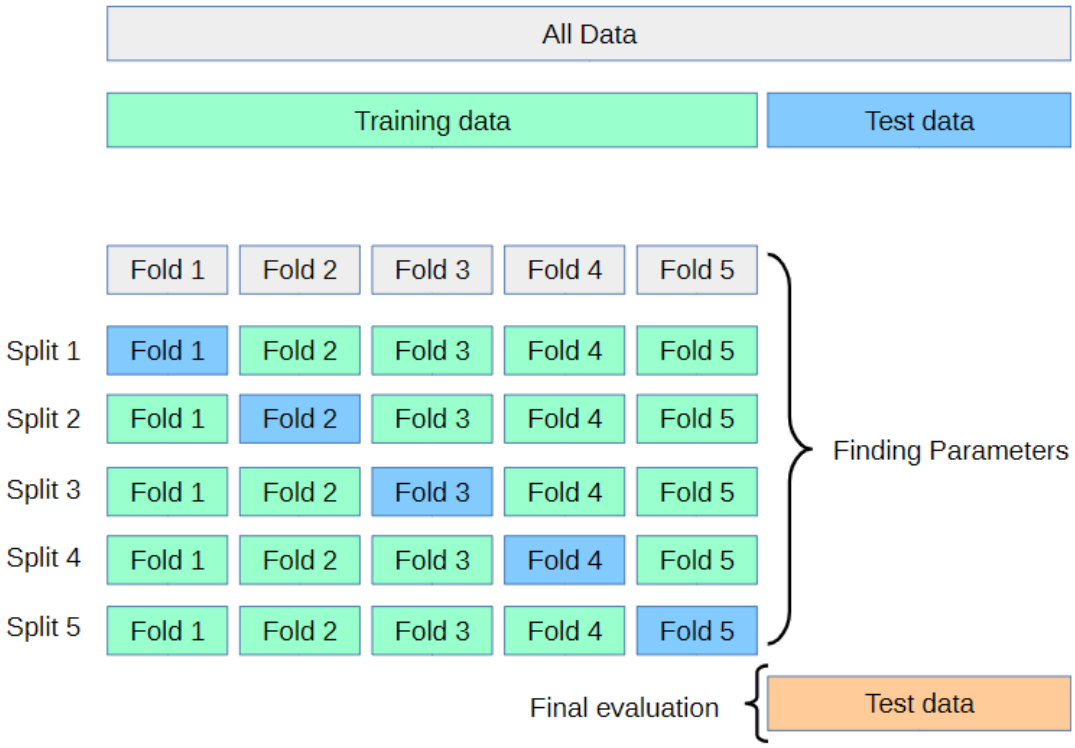
D'habitude le critère déterminant d'un problème est bien connu auprès des experts métier dans l'entreprise, donc il faut les intégrer au projet de Data Science le plus tôt pour mieux connaitre les enjeux métiers d'une part et pour préparer l'adoption du projet pour la suite.

Comme dans le cadre de cette étude on ne dispose pas d'expertise métier, le critère de stratification choisi est la target. Donc chacun des 2 échantillons Test et Train contiendra la même proportion de

Classe_0 (classe majoritaire) et de Classe_1 (classe minoritaire).
Pour synthétiser, au début du traitement pour chaque algorithme :

- On créera les échantillons Train/Test de la façon suivante :
  X_train, X_test, y_train, y_test = train_test_split(X_df, y_df, test_size=0.2, random_state=48)
  => L'échantillon de test sera 20% du jeu de données initiales.
  Les X_train et y_train serviront pour la cross_validation. Les X_test et y_test serviront pour le test en local.
- La cross validation stratifiée en fonction de la classe de sortie sera définie tel que: CV = StratifiedKFold(n_splits=8) On fixera le nombre de splits à 8 (7 splits de Train, 1 split de Test), c'est le nombre optimale constaté après plusieurs essais.

*Figure 11.b - Découpage du jeu de données*




Le principe général d'une cross validation est que le jeu de données soit divisé en K parties égales appelées folds.
Le processus est répété K fois : K-1 folds serviront au training et 1 fold à la validation.
Lors de chaque itération, le fold de validation change de données.

La librairie scikit learn propose des variantes diverses et variées de ce principe. La cross validation retenue dans le cadre de ce projet correspond à la StratifiedKFold : Lors de chaque itération, les folds d'entrainements et celui de validation changent de données de manière à ce que suite aux différentes itérations, le fold de validation aura parcouru la totalité des folds (shuffle=False) tout en respectant la stratification. Il en sera de même pour les (k - 1) folds d'entrainements.
Le tout sera réalisé en respectant que les jeux de données validation et training soient étanches.

## c - Pipeline, préprocesseurs et classifieur:

Un pipeline de traitements sera associé au modèle, il est composé de plusieurs étapes de préprocessing qui s'appliquent sur les colonnes.
La dernière étape du pipeline correspond au classifieur.

Le traitement en pipeline est implémenté au niveau de la classe 'ValeoModeler.py' du module 'valeo.domain'.
Il se traduit par:

- Appliquer le traitement suivant aux colonnes de types numériques :
  - Remplacer les valeurs manquantes par application d'un IterativeImputer selon la stratégie 'median' de l'algorithme.
  - Remplacer les valeurs égales à 0 par application d'un IterativeImputer similaire.
  - Appliquer un RobustScaler qui atténue l'impact des outliers.

- Extraire des features relatives à la date à partir de PROC_TRACEINFO
- Supprimer la feature invariante OP120_Rodage_U_mesure_value
- Appliquer l'encodage OneHotEncoder aux variables de type catégorie
- Appliquer un suréchantillonnage de la classe minoritaire et un sous échantillonnage de la classe majoritaire pour certains algorithmes de classification.
  Voir la méthode 'compute_first_level_classifier(self, clfTypes:[str]) -> [(str, BaseEstimator)]' de 'ValeoModeler.py'
- Appliquer le modèle de classification
  Ces différentes étapes correspondent au modèle, donc elles seront appliquées à travers le modèle à toute les étapes 'Training', 'Validation' et 'Test'; Ainsi qu'en 'Prédiction' une fois que le modèle est mis en production.

Le code relative au pipeline se trouve dans 'ValeoModeler.py' dans le package valeo.domain.
Ci suit des extraits du code:

```
- ColumnTransformer([
                        ('num_transformers_pipeline',num_transformers_pipeline, numerical_features),
                        ('cat_proc_date', pp.ProcDateTransformer(), [C.PROC_TRACEINFO]),
                        ('drop_unecessary_features', pp.DropUnecessaryFeatures(), [C.OP120_Rodage_U_mesure_value]),
                        ], remainder='passthrough')

 Avec
    num_transformers_pipeline = Pipeline([ ('nan_imputer', nan_imputer),
                                           ('zeroes_imputer', zeroes_imputer),
                                           ('scaler', scaler)
                                           ])
   Et
     nan_imputer = IterativeImputer(estimator=BayesianRidge(), missing_values=np.nan,  max_iter=10,
              initial_strategy='median', add_indicator=True, random_state=rand_state)
```

```
    zeroes_imputer = IterativeImputer(estimator=BayesianRidge(), missing_values=0,  max_iter=10,
            initial_strategy='median', add_indicator=True, random_state=rand_state)
    scaler = RobustScaler(with_centering=True, with_scaling=True, quantile_range=(5.0, 95.0))

- Pipeline([ ('preprocessor', self._build_transformers_pipeline(X_df) ) ,
            ('hotencoder_transformer', OneHotEncoder()),
            *self.compute_first_level_classifier(clfTypes) ,
            ('classifier', clfs[clfTypes[0]])
          ])

 Avec :
  def compute_first_level_classifier(self, clfTypes:[str]) -> [(str, BaseEstimator)]:
    if clfTypes[0] in {C.RFC_BLINESMT_RUS, C.HGBC} :
      return  [('over_sampler', BorderlineSMOTE(sampling_strategy=0.1, m_neighbors=5)),
               ('under_sampler', RandomUnderSampler(sampling_strategy=0.5))]
    else :
      return [('combined_over_and_under_sampler',
              SMOTEENN(sampling_strategy='auto')  if clfTypes[0] in
                          {C.RFC_SMOTEEN, C.LRC_SMOTEEN, C.SVC_SMOTEEN, C.KNN_SMOTEEN, C.GNB_SMOTENN} else
              SMOTETomek(sampling_strategy='auto')  if clfTypes[0] in { C.RFC_SMOTETOMEK} else
              pp.EmtpyTransformer() )]
```

## d - Point d'entrée au code Python:

Le code exécuté pour les différents algorithmes, du chapitres 12 au chapitre 23, correspond à cet extrait ci-dessous en provenance de 'main.py'.
On distingue 2 parties :

- Initialisation du contexte de la prédiction
- Génération de la prédiction

```
if __name__ == "__main__" :
    clfTypes     = <Choisir le classifieur utilisé pour la classification>
    clfSelection = <Choisir le modèle train/validation/test utilisé par le classifieur>
    rand_or_optim_iteration_count = 0
    #
    generate_ens_prediction(clfTypes, clfSelection, rand_or_optim_iteration_count)
```

Initialisation du contexte de la prédiction:

- 'clfTypes' correspond au 'type de classifieur (algorithme)' qu'on souhaite utiliser. Ci suit les valeurs possibles :
  - BRFC : Balanced Random Forest - Chapitre 12
  - BBC_ADABOOST : Balanced Bagging Classifier avec AdaBoost - Chapitre 13
  - BBC_GBC : Balanced Bagging Classifier avec GradientBoost - Chapitre 14
  - BBC_HGBC : Balanced Bagging Classifier avec HistGradientBoost - Chapitre 15
  - RUSBoost_ADABoost : RusBoost et AdaBoost - Chapitre 16

  - RFC_SMOTEEN : Random Forest avec SomteENN - Chapitre 17
  - RFC_SMOTETOMEK : Random Forest avec SmoteTomek - Chapitre 18
  - RFC_BLINESMT_RUS : Random Forest avec BorderLineSmote et RandomUnderSampler - Chapitre 19
  - LRC_SMOTEEN : Régression Logistique avec SmoteENN - Chapitre 20
  - SVC_SMOTEEN : Support Vector Machine avec SmoteENN - Chapitre 21
  - KNN_SMOTEEN : KNeighbors Classifier avec SmoteENN - Chapitre 22
  - GNB_SMOTEEN : Gaussian Naïve Bayes avec SmoteENN - Chapitre 23

- 'clfSelection' correspond au 'modèle de train/validation/test' qu'on souhaite utiliser pour le classifieur choisi, ci suit les valeurs possibles :
  - simple_train_test : Train / Test basique
  - cross_validation (cross validate) : Train/Validate sur le split 'train' et Test du modèle sur le split de 'test'.
  - grid_cv (grid search cv) : Train/Validate en utilisant toute les combinaisons des hyperparamètres de GridSearch.
    Train/Validate sur le split 'train' et Test du modèle sur le split de 'test'.
  - rand_cv (randomized search cv) : Train/Validate en utilisant une partie des combinaisons des hyperparamètres 'selon une loi de distribution' et par la suite Test du modèle sur le split de 'test'.
  - optim_cv (optimized search cv) : A partir d'un espace défini, le système applique une optimisation bayésienne en choisissant les hyperparamètres optimaux lors du Train/Validate sur le split 'train'. Par la suite Test du modèle sur le split de 'test'.
  - view_hyp (view hyperparamter) : Cette option permet d'afficher les identificateurs des hyperparamètres du modèle. Aucun Train/Validate ou Test n'est effectué.
    NB: 'clfTypes' et 'clfSelection' sont définies comme des variables constantes dans 'Const.py' du module 'valeo.infrastructure'

- 'rand_or_optim_iteration_count' : Nombre d'itérations de recherche à effectuer lors de l'utilisation de 'rand_cv' et 'optim_cv'.

Ci-dessous un extrait de code, afin de rendre l'explication plus claire:

```
def generate_ens_prediction(
        clfTypes:[str],
        clfSelection:Union[C.simple_train_test, C.cross_validation,  C.grid_cv, C.rand_cv, C.optim_cv,
C.view_hyp],
        rand_or_optim_iteration_count:int = 0) :

    # 1 - Load the data
    mt_train = XY_metadata(
                [C.rootDataTrain(), 'traininginputs.csv'], [C.rootDataTrain(), 'trainingoutput.csv'],
                [C.PROC_TRACEINFO], # Colonne de jointure côté X 'traininginputs.csv'
                [C.PROC_TRACEINFO], # Colonne de jointure côté y 'trainingoutput.csv'
                C.Binar_OP130_Resultat_Global_v) # Colonne représentant la Target
```

```
    xy_loader = XY_Loader()
    X_df, y_df = xy_loader.load_XY_df(mt_train, delete_XY_join_cols=False)

    # 2 - Instantiate ValeoPredictor
    pred = ValeoPredictor()

    # 2.a - Split, fit and validate on  X_train, X_test
    X_train, X_test, y_train, y_test = train_test_split(X_df, y_df, test_size=0.2,
random_state=48)
    fitted_model =
        pred.fit(X_train, y_train, clfTypes) if clfSelection == C.simple_train_test else
        pred.fit_cv_best_score(X_train, y_train, clfTypes, n_splits=8)
                                            if clfSelection == C.cross_validation else
        pred.view_model_params_keys(X_train, clfTypes) if clfSelection == C.view_hyp else
        pred.fit_cv_grid_or_random_or_opt_search(X_train, y_train, clfTypes,
                    cv_type=clfSelection, n_iter=rand_or_optim_iteration_count, n_splits=8)

    # 3 - Predict, plot and generate ENS artefact
    if fitted_model != None:
        pred.predict_and_plot(fitted_model, X_test, y_test, clfTypes)
        generate_y_ens(pred.fit(X_df, y_df, clfTypes),  f'{clfTypes[0]}_{clfSelection}')
```

# 12 - Balanced Random Forest Classifier

## a - Caractéristiques de l'algorithme :

L'algorithme choisi est à base d'arbres, il correspond à un modèle de Random Forest avec un support natif (suréchantillonnage /sous-échantillonnage) pour la gestion des classes déséquilibrées.
Les principales hyperparamètres sont :

- n_estimators : Nombre d'arbres binaires (DecisionTree) constituant la forêt. Chaque arbre sert à effectuer un cycle entier de classification des échantillons afin d'aboutir à un résultat.

- criterion : C'est le type de mesure utilisé pour évaluer l'impureté (l'imprécision) du résultat de l'arbre. Deux valeurs sont possibles 'gini' et 'entropy'.
- max_depth : C'est la profondeur maximale de l'arbre. Aide à améliorer la performance, mais risque aussi d'induire de l'overfit.
- min_samples_split : Le nombre minimal d'échantillons requis par le nœud courant afin de poursuivre le tri au niveau du nœud. Il impacte l'overfit.
- min_samples_leaf : Le nombre minimal d'échantillons autorisés pour former une feuille terminale. Il impacte l'overfit.
- max_features : Nombre de features utilisées pour effectuer les classifications au niveau de l'arbre. Il impacte l'overfit.
- max_leaf_nodes : Nombre maximale autorisé de feuilles terminales au niveau de l'arbre
- bootstrap : Paramètre activé par défaut. Technique d'échantillonnage permettant la sélection des échantillons pour construire chacun des arbres de la forêt.
- oob_score : Lors de l'opération d'échantillonnage s bootstrap, certaines observations du jeu de données ne feront pas partie des échantillons construisant l'arbre.
  Au cas ou ce flag est positionné à True alors ces observations serviront à valider le résultat de l'arbre. Ceci aide à améliorer la généralisation du modèle, donc à réduire l'overfit.
- sampling_strategy : C'est la stratégie de sur échantillonnage et de sous échantillonnage des données.
- Replacement : Si ce flag est égal à True, alors un échantillon choisi pour la construction de l'arbre, est remis dans l'ensemble des observations ce qui le rend potentiellement éligible à être rechoisi en tant qu'échantillon pour la construction du même arbre => Les échantillons peuvent contenir plusieurs fois le même échantillon.

Lors de l'optimisation des hyperparamètres dans le but d'améliorer la performance, il faut éviter d'induire de l'overfit. L'overfit est un signe de variance élevé du modèle, c'est aussi un déficit de généralisation du modèle.

La technique d'échantillonnage utilisée pour construire chacun des arbres de la forêt est appelé bootstrapping.
Le bootstrapping est couplé à l'agrégation du résultat de la classification d'un 'estimateur courant' avec le résultat des 'estimateurs qui l'ont précédé' s'appelle bagging, raccourci de 'bootstrapping + aggregating'.
La technique de bagging permet de réduire l'overfit en réduisant la variance du modèle.

## b - Hyperparamètres utilisés en cross validation et mesures :

- BalancedRandomForestClassifier(criterion= 'gini', max_depth= 10, max_features= 'log2', min_samples_split= 18, n_estimators= 300, oob_score= True, sampling_strategy= 'auto')

- *Tableau 12.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.3700 | 4.1627 |
| ROC_AUC | 0.7027 | 0.8451 |
| Recall | 0.6116 | 0.8107 |
| Precision | 0.0202 | 0.0267 |
| F1 | 0.0390 | 0.0518 |
| Best ROC_AUC on Validation Set (fold 6) | 0.7491 | 0.8423 |

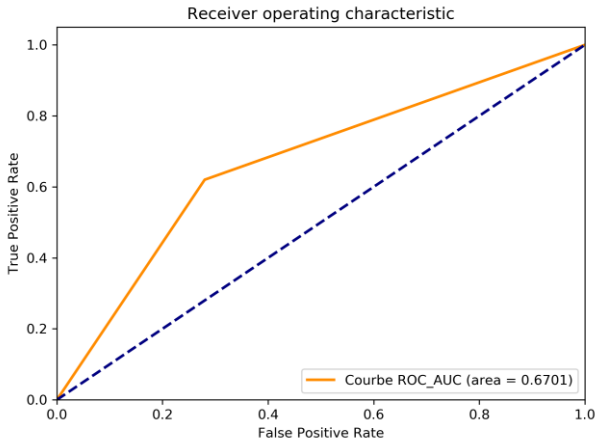
- *Tableau 12.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6701 |
| Recall | 0.6200 |
| Precision | 0.0159 |
| F1 | 0.0310 |
| Matrice de confusion | [4935 1918]<br>[19 31] |

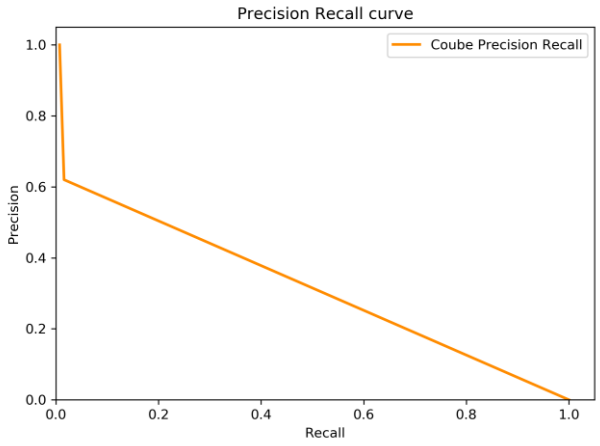
## c - Courbes de ROC AUC et Precision Recall sur Test Set:

*Figure 12.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall



### d - ROC  AUC sur le test set caché de l'ENS: 0.673

NB: La méthode actuelle utilisée par Valéo pour la classification des moteurs, et qui se base uniquement sur les mesures physiques 'sans assistance du machine learning', atteint un ROC_AUC de 0.635.

# 13 - Balanced Bagging Classifier avec AdaBoost

## a - Caractéristiques de l'algorithme :

Cet algorithme est à base d'arbres aussi, avec un support natif (suréchantillonnage /sous-échantillonnage) pour la gestion des classes déséquilibrées.
Il ressemble beaucoup au 'Balanced Random Forest Classifier', la différence entre les 2 modèles se résume à :

- Dans le cas de Randon Forest l'estimateur utilisé est forcément un DecisionTree alors dans le cas du Bagging Classifier, il peut être d'un autre type.
- Dans le cas du Bagging Classifier l'ordre des features utilisées pour effectuer le split au niveau des nœuds est toujours le même, alors que dans le cas du Randon Forest cet ordre est aléatoire, ce qui réduit le surapprentissage et rend le modèle plus robuste.

Dans le cas présent, on va utiliser l'estimateur AdaBoost, qui est un 'weak learner'. Un weak leaner est formé d'un estimateur qui n'est pas performant d'une manière isolée, CEPENDANT, ce qui le rend performant c'est le cycle d'apprentissage suivi:

- Un weak learner est un arbre de 1 niveau de profondeur appelé 'stump'. Dans le cas d'AdaBoost c'est une RF de de profondeur 1.
- La forêt de stumps s'entraine sur le même échantillon qui aurait été construit précédemment par le Bagging Classifier
- Lors des itérations des classifications effectuées par chaque arbre de la forêt, la pondération d'une observation mal classifiée est augmentée afin que les itérations suivantes se focalisent sur les observations les plus difficiles à être correctement classifier ; D'ou le raccourci Ada correspondant à 'adaptive'.
- Cette technique d'entrainement séquentielle s'appelle 'boosting', le modèle de chaque itération tente de corriger les erreurs de son prédécesseur.

## b - Hyperparamètres utilisés en cross validation et mesures:

- BalancedBaggingClassifier(base_estimator=AdaBoostClassifier(), n_estimators= 300, max_samples=0.7, max_features= 8, oob_score= True, replacement=True , sampling_strategy= 'auto', n_jobs=-1)

- *Tableau 13.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 8.7849 | 21.6472 |
| ROC_AUC | 0.6694 | 0.8171 |
| Recall | 0.5295 | 0.7535 |
| Precision | 0.0188 | 0.0270 |
| F1 | 0.0363 | 0.0522 |
| Best ROC_AUC on Validation Set (fold 1) | 0.7178 | 0.8075 |

- NB :
  Un essai de BalancedBaggingClassifier a été effectué avec l'estimateur par défaut, DecisionTree.
  Le *Tableau 13.b.2* montre une très grande différence entre les performances du Training et ceux de la Validation, un défaut de généralisation. On voit bien la meilleure capacité de généralisation de la technique 'boost weak learner' au *Tableau 13.b.1* .

  BalancedBaggingClassifier(n_estimators= 300, max_samples=0.7, max_features= 8, oob_score= True, replacement=True , sampling_strategy= 'auto', n_jobs=-1)

- *Tableau 13.b.2 - Moyenne des mesures par folds de CV / BalancedBaggingClassifier avec DecisionTree*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | | |
| ROC_AUC | 0.6992 | 0.9981 |
| Recall | 0.4921 | 1.0000 |
| Precision | 0.0240 | 0.0482 |
| F1 | 0.0457 | 0.0921 |
| Best ROC_AUC on Validation Set (fold 1) | 0.7353 | 0.9984 |

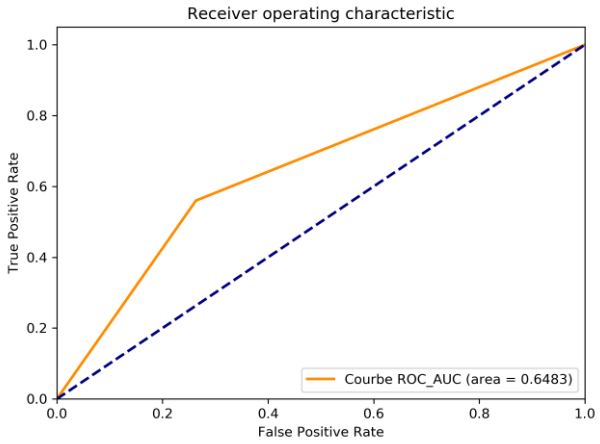
- *Tableau 13.b.3 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6483 |
| Recall | 0.5600 |
| Precision | 0.0153 |
| F1 | 0.0297 |
| Matrice de confusion | [5048 1805]<br>[22 28] |

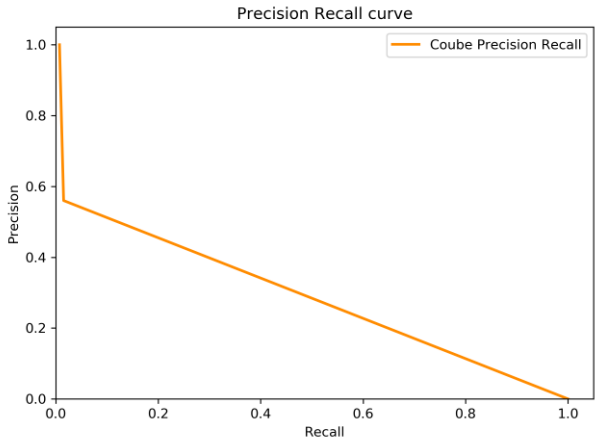
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 13.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.655

# 14 - Balanced Bagging Classifier avec GradientBoost

## a - Caractéristiques de l'algorithme :

Cet algorithme est très similaire à celui du chapitre_12 avec l'utilisation du GradientBoost au lieu de l'AdaBoost. Le GradientBoost est aussi un weak learner et adopte aussi l'apprentissage de l'ensemble comme l'AdaBoost

.

La différence entre les deux réside dans la méthode utilisée par chaque algorithme lors de la correction du modèle. Le modèle est corrigé à la fin d'une séquence de classification :

- AdaBoost: L'algorithme attribue plus de poids aux observations mal classifiées.
- GradientBoost: L'algorithme crée un nouveau modèle ayant pour target les erreurs des observations mal classifiées.
  Les prédictions du nouveau modèle sont combinées aux prédictions précédentes et une nouvelle séquence de classification est entamé.

## b - Hyperparamètres utilisés en cross validation et mesures:

- BalancedBaggingClassifier(
  base_estimator=GradientBoostingClassifier(learning_rate= 0.1, max_depth= 10,
  max_features= 'log2', min_samples_split= 18),
  n_estimators= 200, max_samples=0.7, max_features= 8, oob_score= True, replacement=True )

- *Tableau 14.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 3.7217 | 21.0091 |
| ROC_AUC | 0.6999 | 0.9468 |
| Recall | 0.4981 | 0.9905 |
| Precision | 0.0224 | 0.0441 |
| F1 | 0.0429 | 0.0844 |
| Best ROC_AUC on Validation Set (fold 4) | 0.7529 | 0.9539 |

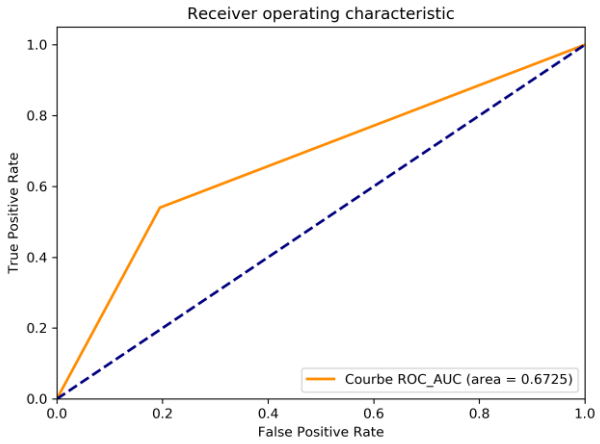
- *Tableau 14.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6725 |
| Recall | 0.5400 |
| Precision | 0.0198 |
| F1 | 0.0382 |
| Matrice de confusion | [5516 1337]<br>[23 27] |

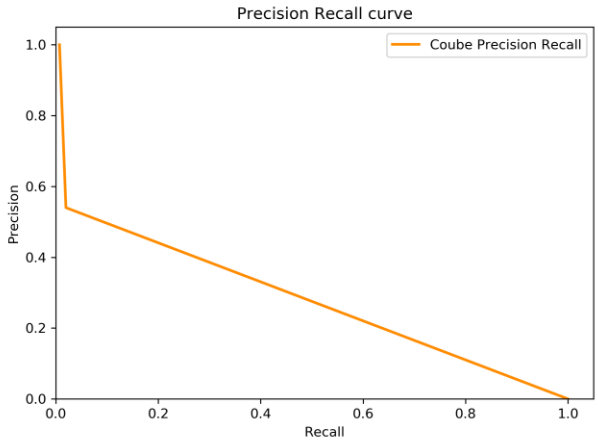
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 14.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.673

# 15 - Balanced Bagging Classifier avec HistGradientBoost

## a - Caractéristiques de l'algorithme :

Cet algorithme est très similaire à celui du chapitre_13 avec utilisation du HistGradientBoost au lieu du GradientBoost. Le HistGradientBoost est aussi un weak learner et adopte aussi l'apprentissage de l'ensemble.

Les features des observations sont réparties sur des bins ou des buckets ce qui réduit considérablement le nombre de valeurs utilisées pour trier (splitter) les données au niveau de l'arbre.
L'algorithme gère aussi les valeurs manquantes en les isolant dans un bucket à part. Le nombre de bucket par défaut est de 255.

## b - Hyperparamètres utilisés en cross validation et mesures :

- BalancedBaggingClassifier(
  base_estimator=HistGradientBoostingClassifier(max_iter = 100, max_depth=5,learning_rate=0.10,
  l2_regularization=15, scoring='roc_auc'),
  n_estimators= 200, max_samples=0.7, max_features= 8, oob_score= True, replacement=True,
  sampling_strategy= 'auto')

- *Tableau 15.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 6.0610 | 22.5675 |
| ROC_AUC | 0.6761 | 0.8164 |
| Recall | 0.4942 | 0.6560 |
| Precision | 0.0206 | 0.0276 |
| F1 | 0.0395 | 0.0530 |
| Best ROC_AUC on Validation Set (fold 4) | 0.7421 | 0.8245 |

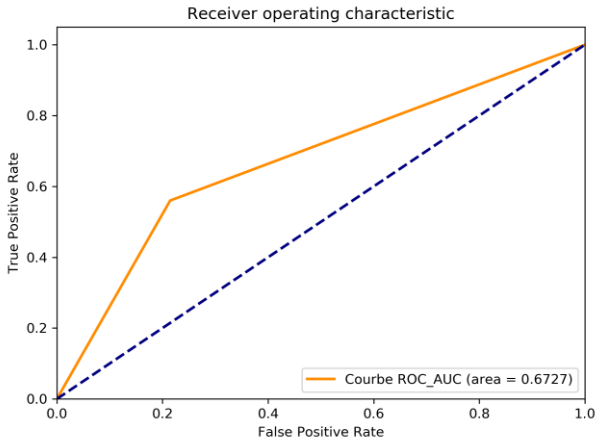
- *Tableau 15.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6727 |
| Recall | 0.5600 |
| Precision | 0.0187 |
| F1 | 0.0362 |
| Matrice de confusion | [5383 1470]<br>[22 28] |

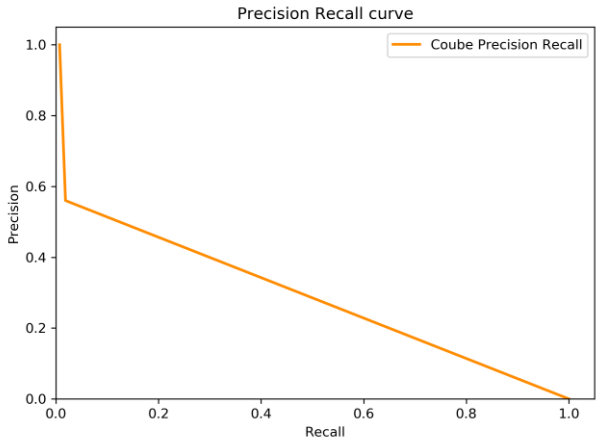
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 15.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.649

# 16 - RusBoost et AdaBoost

## a - Caractéristiques de l'algorithme :

Cet algorithme combine un algorithme de sous échantillonnage à un algorithme d'AdaBoost. Lors de l'apprentissage, le problème du déséquilibre est atténué par un sous échantillonnage lors de chaque itération au niveau d'AdaBoost.

## b - Hyperparamètres utilisés en cross validation et mesures :

- RUSBoostClassifier(base_estimator = AdaBoostClassifier(), n_estimators = 50 , algorithm='SAMME.R', random_state=42)

- *Tableau 16.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 4.1073 | 23.7319 |
| ROC_AUC | 0.6478 | 0.8348 |
| Recall | 0.6237 | 0.9120 |
| Precision | 0.0148 | 0.0218 |
| F1 | 0.0290 | 0.0426 |
| Best ROC_AUC on Validation Set (fold 1) | 0.7155 | 0.8264 |

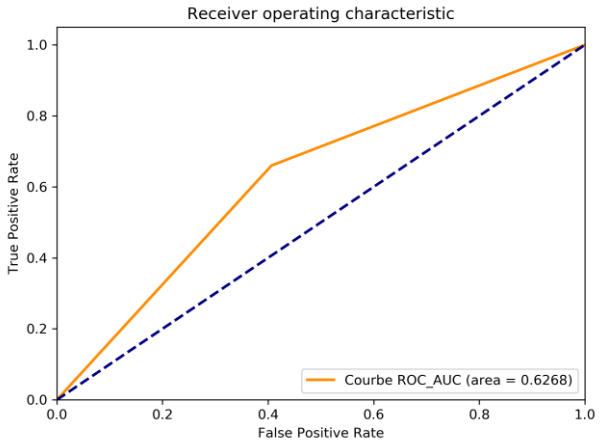
- *Tableau 16.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6268 |
| Recall | 0.6600 |
| Precision | 0.0117 |
| F1 | 0.0230 |
| Matrice de confusion | [4068 2785]<br>[17 33] |

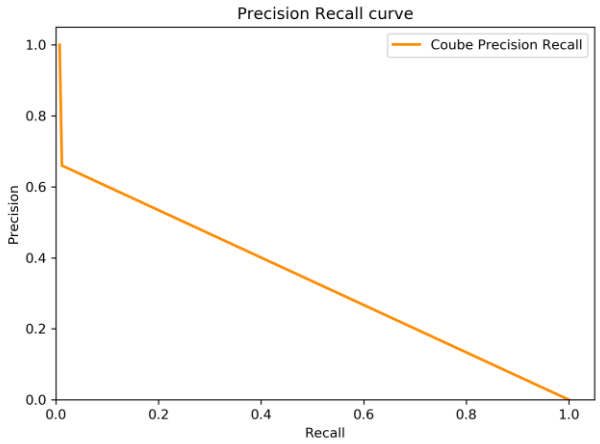
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 16.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.662

# 17 - Random Forest avec SomteENN

## a - Caractéristiques de l'algorithme :

Dans ce cas de figure on essayé un Random Forest classique précédé d'une opération combinée suréchantillonne/sous-échantillonnage SMOTE/ENN (Edited Nearest Neighboors).
*Extrait du paragraphe 10.d*

- ENN consiste à:
  - Pour chaque 'échantillon' de la classe à sous échantillonner, trouver le 'voisin le plus proche'
  - Si ce voisin n'appartient pas à la classe du dit 'échantillon', alors supprimer cet 'échantillon'

## b - Hyperparamètres utilisés en cross validation et mesures :

- RandomForestClassifier(criterion= 'gini', max_depth= 8, max_features= 'log2', min_samples_split= 25, n_estimators=100, oob_score= True)
  + SMOTEENN(sampling_strategy='auto')

- *Tableau 17.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.3572 | 29.6008 |
| ROC_AUC | 0.6798 | 0.8590 |
| Recall | 0.4316 | 0.7081 |
| Precision | 0.0213 | 0.0356 |
| F1 | 0.0406 | 0.0677 |
| Best ROC_AUC on Validation Set (fold 6) | 0.7532 | 0.8578 |

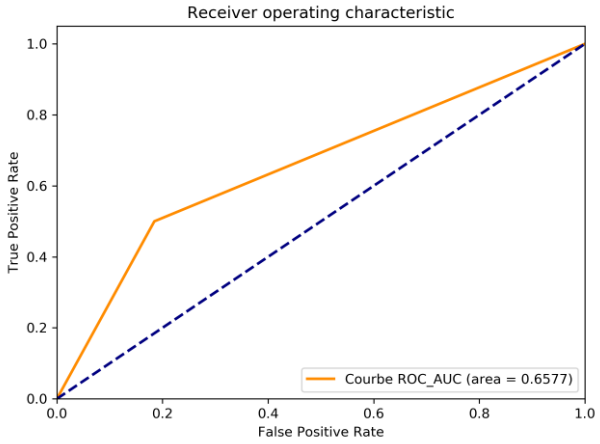
- *Tableau 17.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6577 |
| Recall | 0.5000 |
| Precision | 0.0194 |
| F1 | 0.0373 |
| Matrice de confusion | [5588 1265]<br>[25 25] |

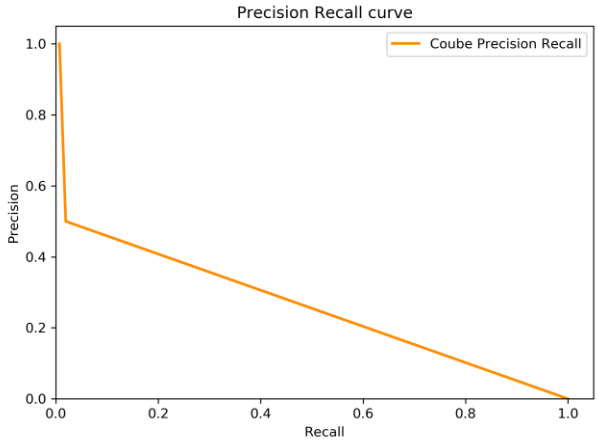
## c - Courbes de ROC_AUC et Precision Recall sur Test Set:

*Figure 17.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC_AUC sur le test set caché de l'ENS: 0.663

# 18 - Random Forest avec SmoteTomek

## a - Caractéristiques de l'algorithme :

Dans ce cas de figure on a essayé un Random Forest classique précédé d'une opération combinée suréchantillonnage/sous-échantillonnage SMOTE/TOMEK (Tomek links).
*Extrait du paragraphe 10.d*

- TOMEK consiste à:
  - Trouver 2 échantillons '2 voisins les plus proches' dont chacun appartient à une classe différente
  - Supprimer un échantillon (l'échantillon majoritaire ou minoritaire) des 2 ou bien de supprimer les 2. Le choix par défaut étant de supprimer l'échantillon majoritaire.

## b - Hyperparamètres utilisés en cross validation et mesures :

- RandomForestClassifier(criterion= 'gini', max_depth= 8, max_features= 'log2', min_samples_split= 25, n_estimators=100, oob_score= True)
  + SMOTETomek(sampling_strategy='auto')

- *Tableau 18.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.3581 | 23.8095 |
| ROC_AUC | 0.6843 | 0.8666 |
| Recall | 0.3572 | 0.6275 |
| Precision | 0.0227 | 0.0410 |
| F1 | 0.0427 | 0.0769 |
| Best ROC_AUC on Validation Set (fold 6) | 0.7401 | 0.8687 |

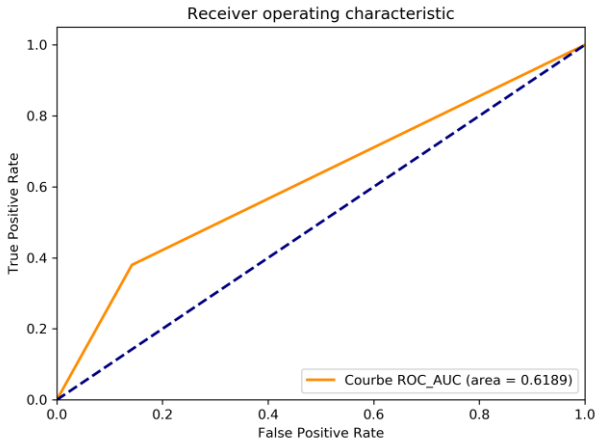
- *Tableau 18.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6189 |
| Recall | 0.3800 |
| Precision | 0.0191 |
| F1 | 0.0364 |
| Matrice de confusion | [5879 974]<br>[31 19] |

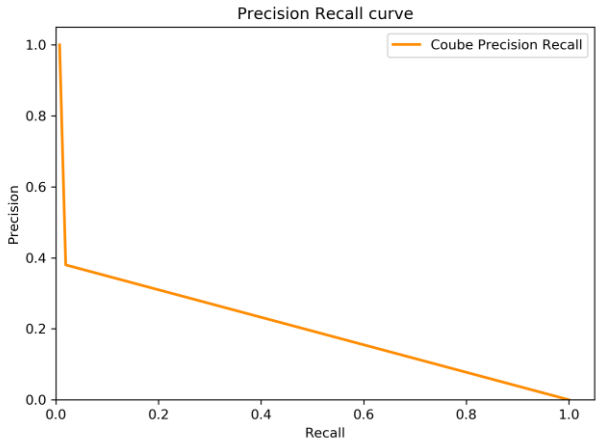
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 18.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.613

# 19 - Random Forest avec BorderLineSmote et RandomUnderSampler

## a - Caractéristiques de l'algorithme :

Dans ce cas de figure on a essayé un Random Forest classique précédé d'une opération de suréchantillonnage BorderLineSmote et d'une opération de sous-échantillonnage RandomUnderSample.

*Extrait du paragraphe 10.c*

- Il existe 2 types de BorderLineSmote:
  - Borderline SMOTE_1 : En utilisant la classification des "k voisins de Classe_1 les plus proches", il va choisir les instances de la Classe_1 se trouvant à côté de la frontière et qui sont mal classées puis suréchantillonner uniquement ces cas difficiles, en fournissant plus de résolution uniquement là où cela peut être nécessaire. La sélection se fait comme avec un modèle de classification du voisin le plus proche.
  - Borderline SMOTE_2 : Semblable à Borderline SMOTE_1 en suréchantillonnant aussi les exemples de Classe_0, la classe majoritaire ; l'idée consiste à suréchantillonner les exemples à la frontière et qui sont mal classés.

## b - Hyperparamètres utilisés en cross validation et mesures :

- RandomForestClassifier(criterion= 'gini', max_depth= 8, max_features= 'log2', min_samples_split= 25, n_estimators=100, oob_score= True)
  + BorderlineSMOTE(sampling_strategy=0.1, m_neighbors=5)
  + RandomUnderSampler(sampling_strategy=0.5)

- *Tableau 19.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.3894 | 3.8254 |
| ROC_AUC | 0.6341 | 0.7984 |
| Recall | 0.0237 | 0.1585 |
| Precision | 0.0195 | 0.1018 |
| F1 | 0.0210 | 0.1229 |
| Best ROC_AUC on Validation Set (fold 6) | 0.6605 | 0.7835 |

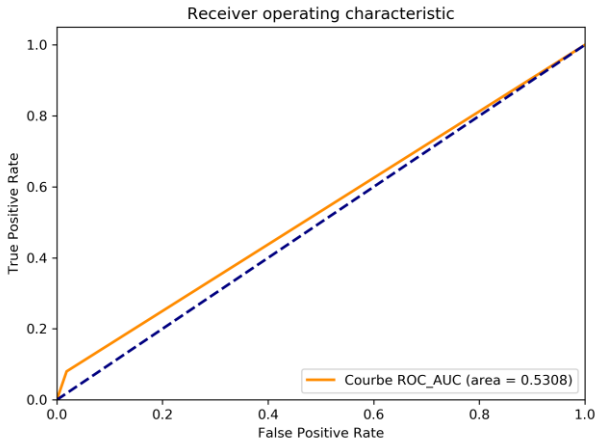
- *Tableau 19.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.5308 |
| Recall | 0.0800 |
| Precision | 0.0308 |
| F1 | 0.0444 |
| Matrice de confusion | [6727 126]<br>[46 4] |

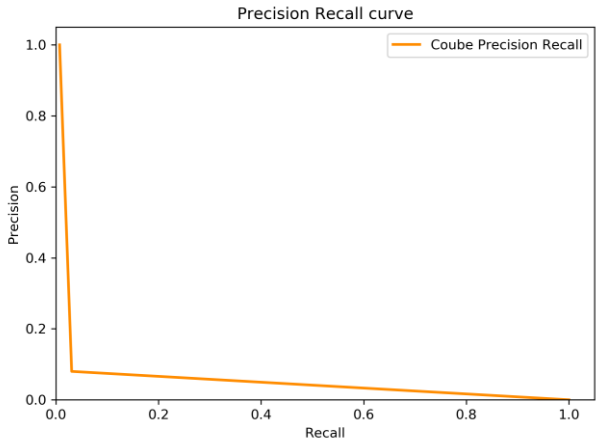
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 19.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.646

# 20 - Régression logistique associée à un suréchantillonnage /échantillonnage combiné SmoteENN

## a - Caractéristiques de l'algorithme :

La régression logistique est un algorithme de classification qui permet d'estimer la probabilité qu'une instance appartient à une classe particulière. Si la probabilité estimée est supérieure à 50%, alors le modèle prédit que l'instance appartient à cette classe.
L'utilisation de cet algorithme requiert que toutes les features soient ramenées à une même échelle de grandeur, donc normalisées.
L'algorithme tente de réduire la différence entre la valeur exacte du résultat et la valeur calculée, c'est ce qu'on appelle la fonction de coût.

- solver : L'algorithme utilisé pour optimiser l'erreur.
  On distingue les solveurs suivants:
  - newton-cg : Il est lent sur les grands jeux de données car il calcule les dérivées secondes.
  - lbfgs : Il n'est pas rapide non plus, ce solver approxime la dérivée seconde et ne sauvegarde que les dernières valeurs.
  - liblinear : Cet algorithme performe bien quand il s'agit de beaucoup de dimensions.
  - sag : Stochastic gradient descent. Rapide pour les grands jeux de données.
  - saga: Extension de sag, permet l'utilisation de la norme L1

  *Figure 20.a - Solver de Régression logistique*

| Penalties | 'liblinear' | 'lbfgs' | 'newton-cg' | 'sag' | 'saga' |
|---|---|---|---|---|---|
| Multinomial + L2 penalty | no | yes | yes | yes | yes |
| OVR + L2 penalty | yes | yes | yes | yes | yes |
| Multinomial + L1 penalty | no | no | no | no | yes |
| OVR + L1 penalty | yes | no | no | no | yes |
| Elastic-Net | no | no | no | no | yes |
| No penalty ('none') | no | yes | yes | yes | yes |
| **Behaviors** | | | | | |
| Penalize the intercept (bad) | yes | no | no | no | no |
| Faster for large datasets | no | no | no | yes | yes |
| Robust to unscaled datasets | yes | yes | yes | no | no |

- C : L'inverse de la régularisation. La régularisation permet de réduire l'overfit d'un modèle.
- max_iter : C'est le seuil d'itérations maximales pour que l'algorithme converge.
- fit_intercept : Si ce flag est égal à True, alors l'axe de la target 'y' sera intercepté pour 'x égal à 0' au cas où ceci correspondrait au best fit => Un biais est rajouté.

Une phase de suréchantillonnage et sous échantillonnage SMOTEENN vont précéder l'appel à ce classifieur.

## b - Hyperparamètres utilisés en cross validation et mesures :

- LogisticRégression(C= 1000, fit_intercept= False, max_iter= 1000, penalty= 'l2', solver= 'saga') + SMOTEENN(sampling_strategy='auto')

- *Tableau 20.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.1571 | 60.4653 |
| ROC_AUC | 0.7133 | 0.7439 |
| Recall | 0.6352 | 0.6644 |
| Precision | 0.0190 | 0.0199 |
| F1 | 0.0369 | 0.0387 |
| Best ROC_AUC on Validation Set (fold 1) | 0.7671 | 0.7332 |

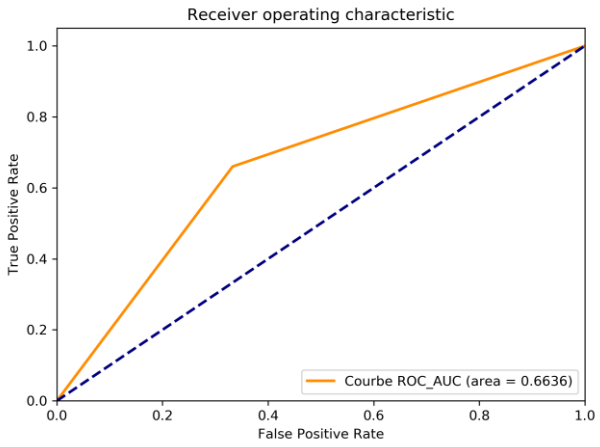
- *Tableau 20.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.6636 |
| Recall | 0.6600 |
| Precision | 0.0143 |
| F1 | 0.0279 |
| Matrice de confusion | [4572 2281]<br>[17 33] |

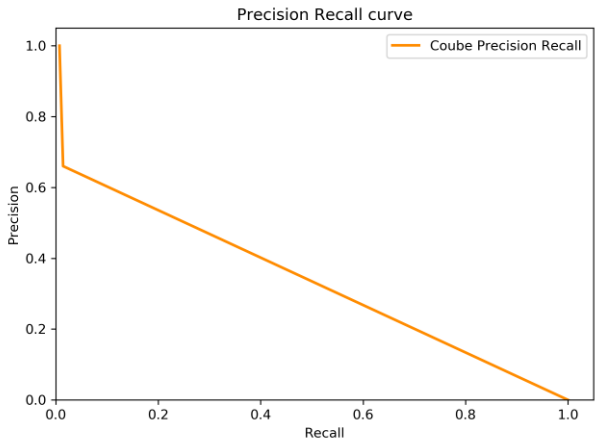
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 20.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall



**d - ROC  AUC sur le test set caché de l'ENS: 0.6904**

## 21 - Support Vector Machine Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN

### a - Caractéristiques de l'algorithme :

- L'algorithme SVM tente de trouver une frontière de décision, un hyperplan qui divise au mieux les exemples en deux classes.

- L'hyperplan est défini par une marge qui maximise la distance entre la frontière de décision et les exemples les plus proches de chacune des deux classes.

- La séparation est assouplie grâce à l'utilisation d'une marge qui permet à certains points d'être mal classés.

- Cette marge favorise la classe majoritaire sur les ensembles de données déséquilibrés. Cependant elle peut être adaptée pour tenir compte de l'importance de chaque classe et améliorer les performances de l'algorithme, c'est le SVM pondéré.

- Les données issues des observations peuvent être transformées à l'aide d'un noyau pour permettre de définir des hyperplans de séparations linéaires des classes.

- Cette transformation ramène les entités d'un espace où l'hyperplan de séparation des classes n'était pas linéaire, vers un autre espace ou l'hyperplan de séparation est linéaire, donc les classes seront plus facilement séparables.

- La transformation des données connue par le 'kernel trick' supporte des transformations dont l'hyperplan de séparation d'origine était polynomial, radial, gaussien ou autres.

- Cet algorithme est adapté aux classification complexes avec un jeu de données de taille moyenne ou petite

Dans ce premier exemple, on va utiliser une la technique SmoteENN pour la pondération des données ; Il aurait été possible d'utiliser la pondération native intégrée à l'algorithme.

### b - Hyperparamètres utilisés en cross validation et mesures :

- SVC(kernel='rbf', gamma='scale', C=10, probability=True, random_state=42)
  + SMOTEENN(sampling_strategy='auto')

  – Kernel : Le kernel de transformation Gaussien RBF (radial basis function) est utilisé, il s'adapte bien à la plupart des cas.
  – C : L'inverse de la régularisation. La régularisation permet de réduire l'overfit d'un modèle.
  – gamma : Propriétés relatives au kernel dans le cas du 'rbf'. Si gamma augmente alors la distribution gaussienne de la fonction de transformation correspondra à une courbe gaussienne plus étroite.

'gamma' agit comme un paramètre de régularisation et est proportionnel à cette dernière. Au niveau de l'algorithme, il prend une des 2 valeurs suivantes :

- 'auto' : => gamme est égal à 1 / nbre_features
- 'scale': => gamma est égal à 1 / 'nbre_features * variance)

– probability: Active ou non les estimations de probabilité.

- *Tableau 21.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 3.5207 | 361.6414 |
| ROC_AUC | 0.6400 | 0.9399 |
| Recall | 0.2708 | 0.9384 |
| Precision | 0.0191 | 0.0666 |
| F1 | 0.0357 | 0.1244 |
| Best ROC_AUC on Validation Set (fold 6) | 0.7199 | 0.9373 |

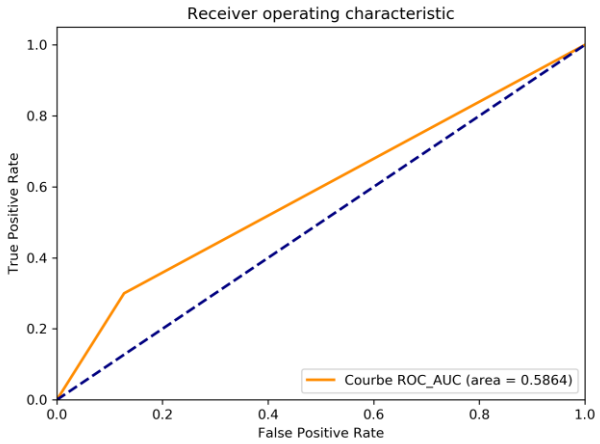
- *Tableau 21.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.5864 |
| Recall | 0.3000 |
| Precision | 0.0169 |
| F1 | 0.0320 |
| Matrice de confusion | [5981 872]<br>[35 15] |

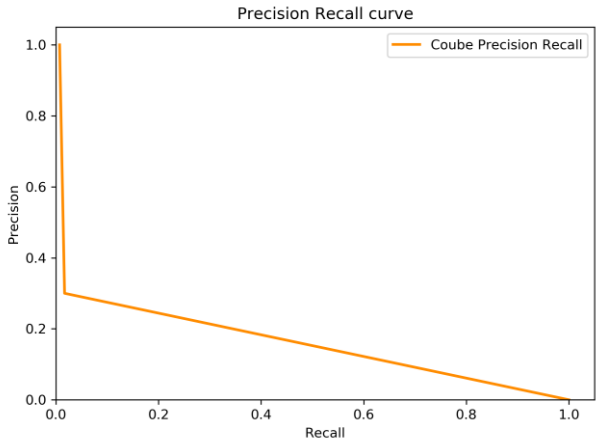
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 21.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.6798

# 22 - KNeighbors Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN

## a - Caractéristiques de l'algorithme :

Cet algorithme connu par KNN, K est le nombre de voisins les plus proches. Le nombre de voisins est le principal facteur décisif dans le contexte de cet algorithme, il est généralement un nombre impair.

Supposons que Pt_1 soit le point pour lequel la classe doit être prédite:

- L'algorithme cherche les K points les plus proches à Pt_1 en calculant la distance entre Pt_1 et ses voisins.
- Par la suite Pt_1 est classé en fonction du vote majoritaire de ses voisins choisis.
  La valeur de K par défaut est 5. Il n'a pas de valeurs par défaut, chaque jeu de données a ses propres exigences.

Dans le cas d'un petit nombre de voisins, le bruit aura une plus grande influence sur le résultat, et un grand nombre de voisins le rendent coûteux en calcul. Un petit nombre de voisins génère un faible biais mais une variance élevée => Induit de l'overfit.
Un grand nombre de voisins génère une frontière de décision plus lisse, ce qui signifie une variance plus faible mais une valeur de biais plus élevée => Induit de l'underfit.

## b - Hyperparamètres utilisés en cross validation et mesures :

- KNeighborsClassifier(n_neighbors=7, weights='uniform')
  + SMOTEENN(sampling_strategy='auto')

- *Tableau 22.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 6.6173 | 25.8993 |
| ROC_AUC | 0.5931 | 0.9648 |
| Recall | 0.2703 | 0.9535 |
| Precision | 0.0165 | 0.0660 |
| F1 | 0.0312 | 0.1234 |
| Best ROC_AUC on Validation Set (fold 6) | 0.6255 | 0.9634 |

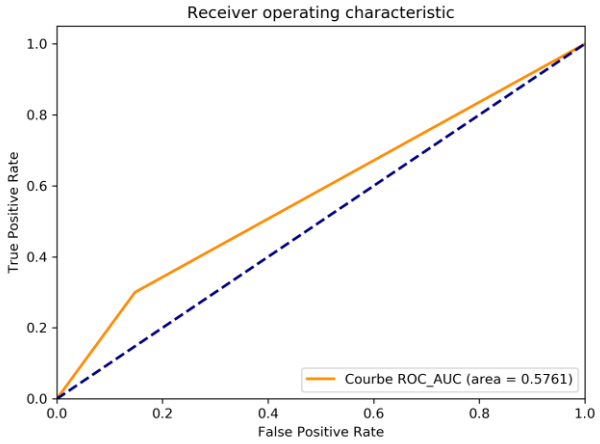
- *Tableau 22.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.5761 |
| Recall | 0.3000 |
| Precision | 0.0146 |
| F1 | 0.0278 |
| Matrice de confusion | [5840 1013]<br>[35 15] |

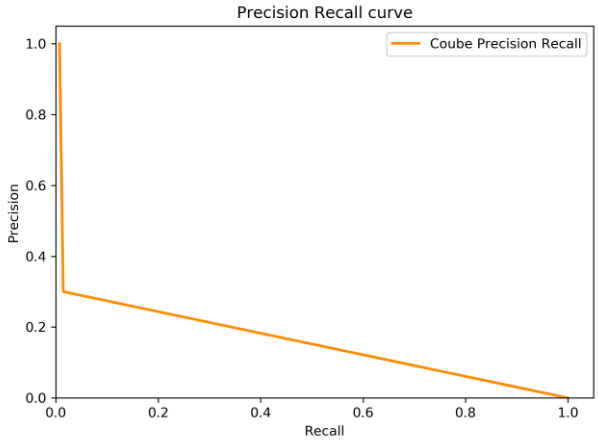
## c - Courbes de ROC_AUC et Precision Recall sur Test Set:

*Figure 22.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC_AUC sur le test set caché de l'ENS: 0.605

# 23 - GaussianNB Classifier associée à un suréchantillonnage /échantillonnage combiné SmoteENN

## a - Caractéristiques de l'algorithme :

Les modèles Naïve Bayes peuvent être utilisés pour résoudre des problèmes de classification à grande échelle pour lesquels l'ensemble complet d'entraînement pourrait ne pas tenir en mémoire.

## b - Hyperparamètres utilisés en cross validation et mesures :

- KGaussianNB()
  + SMOTEENN(sampling_strategy='auto')

- *Tableau 23.b.1 - Moyenne des mesures par folds de CV*

| Moyenne par folds de CV | Validation Set | Train Set |
|---|---|---|
| Temps consommé | 0.1463 | 24.2888 |
| ROC_AUC | 0.5715 | 0.5854 |
| Recall | 0.9335 | 0.9384 |
| Precision | 0.0105 | 0.0105 |
| F1 | 0.0207 | 0.0208 |
| Best ROC_AUC on Validation Set (fold 0) | 0.6418 | 0.6665 |

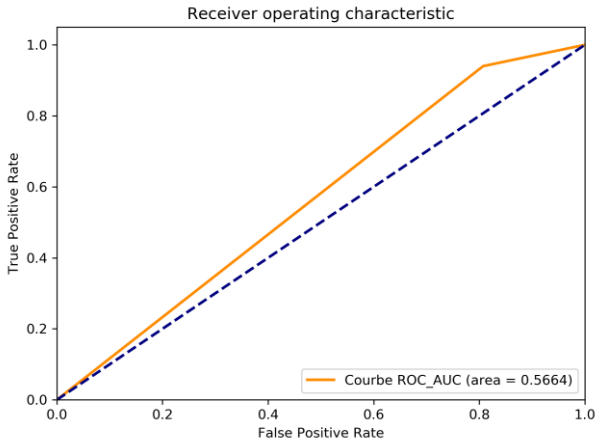
- *Tableau 23.b.2 - Résultat de la prédiction sur Test Set*

| Mesure | Valeur |
|---|---|
| ROC_AUC | 0.5664 |
| Recall | 0.9400 |
| Precision | 0.0084 |
| F1 | 0.0167 |
| Matrice de confusion | [1321 5532]<br>[ 3 47] |

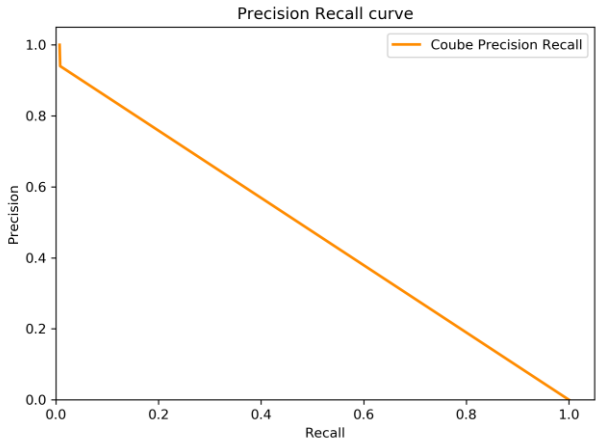
## c - Courbes de ROC  AUC et Precision Recall sur Test Set:

*Figure 23.c - Courbes ROC_AUC et Precision/Recall*

Courbe roc auc




Courbe Precision Recall




## d - ROC  AUC sur le test set caché de l'ENS: 0.580

# 24 - Récapitulatif et synthèses des méthodes utilisées

## a - Introduction :

Ce chapitre va regrouper dans une même vue les différentes métriques exposées dans les chapitres précédents pour chacune des méthodes.
On ne reviendra pas sur le détail de fonctionnement de chacun des algorithmes, ceci ayant déjà été exposé dans le chapitre correspondant.
On va commencer par énumérer les métriques à afficher en soulignant leurs spécificités. On va enrichir cette vue par des nouvelles propriétés qui n'étaient pas évoquées dans les chapitres précédents.

## b - Jeu de données :

Pour rappel, le jeu de données entier fait 34515 observations :

- Chaque observation fait 13 features numériques.
- Suite à la création de nouvelles features, 3 nouvelles features de type ‘catégorie’ ont été rajoutées.
  - Une première feature qui compte 6 catégories
  - une deuxième qui compte 21 catégories
  - une troisième qui compte 7 catégories Ces catégories ont dû être transformées par un HotEncoder.
- Avant de commencer le traitement, le jeu de données a été divisé par train_test_split() en:
  - un split de 80% qui servira pour la cross validation (CV=8) => 7 folds de train et 1 fold de validation
  - un split de 20% qui servira comme test sur mon poste local
- A la fin, un deuxième test est effectué sur le jeu de données (features) de l'ENS dont on ne dispose pas de la valeur réelle de la target. Par la suite, le résultat est téléversé sur le site de l'ENS.

## c - Métriques et autres propriétés :

On retrouvera :

- Performance de l'algorithme :
  - La moyenne du roc_auc, du recall, de la précision et du f1 des différentes itérations 'train' et 'validation' de la cross validation
  - Roc_auc du test de l'ENS.
- Coût en terme de ressources temps (temps machine) :
  - La moyenne du 'fit time' d'une itération de cross validation
  - La moyenne du 'test time' du test de validation correspondant.
- Propriétés de l'algorithme:
  - Bootstrapping, bagging, boosting, support natif undesampling/oversampling du classifieur au cas ou il existe.
  - Combinaison d'un algorithme supplémentaire d'oversampling/undersampling (SmoteENN, SmoteLink, BorderLineSmote/RandomUnderSample)

- Paramétrique (oui/non):
  - Modèle paramétrique:
    - Il possède un nombre fini de paramètres ce qui permet de le définir en une structure fixe.
    - Il est plus rapide en phase d'apprentissage
    - Il requiert moins de données pour apprendre
    - D'une manière générale, il est adapté au problème simple et il est très lié à la fonction qui le représente
  - Modèle non paramétrique:
    - S'il possède potentiellement un nombre infini de paramètres. Le modèle devient plus complexe en augmentant le volume de training data.
    - Temps d'apprentissage plus long
    - Le risque d'overfit est plus grand
- Lazy où Eager:
  - Modèle 'lazy':
    - En phase d'apprentissage, il stocke uniquement les données d'apprentissage au lieu d'en tirer des règles à utiliser en phase de test.
    - le temps d'apprentissage est moins long mais le temps de prédiction est plus long.
  - Modèle 'eager':
    - En phase d'apprentissage, il construit un model de classification avant de recevoir les données à tester ou à prédire.
    - Le temps d'apprentissage est plus long mais le temps de prédiction est plus court.
- Discriminatif ou Génératif :
  - Le modèle discriminatif est un modèle qui classifie la target en fonction de la frontière de décision entre ses différentes classes.
  - Le modèle génératif est un modèle qui classifie la target en comparant ses features d'origine aux features des autres targets d'entrainement.

## d - Vue synthétique :

Dans les tableaux suivants *Figure 24.d.1* et *Figure 24.d.2* on a regroupé les méthodes utilisées en 3 groupes:

- Groupe 1: Classifications basées sur les arbres issues du package imblearn de scikit learn.
- Groupe 2: Classifications basées sur les arbres associées à du suréchantillonnage et de sous échantillonnage.
- Groupe 3: Classifications basées sur les distances euclidiennes associées à du suréchantillonnage et du sous échantillonnage + Une classification Naïve Bayes.

Dans le premier tableau on distingue 3 séries de métriques :

- Les moyennes des métriques pour les folds de train d'une cross validation à 8 folds + le 'roc_auc/train' correspondant au fold dont le 'roc_auc/validation' est le plus élevé.
- Les moyennes des mêmes métriques pour le fold validation + le roc_auc/validation le plus élevé.
- Les moyennes des mêmes métriques pour les split de test + le roc_auc de l'algorithme calculée par l'ENS

*Figure 24.d.1 - Performances des méthodes utilisées*

| Méthodes | Perfomance de l'algorithme | | | | | | | | | | | | | | | Chapitre |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Moy. du split train de la CV | | | | bst split | Moy. du split valid. de la CV | | | | bst split | Split test local | | | | Ens split | |
| | roc_auc | recall | prec. | f1 | roc_auc | roc_auc | recall | prec. | f1 | roc_auc | roc_auc | recall | prec. | f1 | roc_auc | |
| Balanced RF | 0.8451 | 0.8107 | 0.0267 | 0.0518 | 0.8423 | 0.7027 | 0.6116 | 0.0202 | 0.0390 | 0.7491 | 0.6701 | 0.6200 | 0.0159 | 0.0310 | **0.673** | 12 |
| Balanced Bag + Adaboost | 0.8171 | 0.7535 | 0.0270 | 0.0522 | 0.8075 | 0.6694 | 0.5295 | 0.0188 | 0.0363 | 0.7178 | 0.6483 | 0.5600 | 0.0153 | 0.0297 | 0.655 | 13 |
| Balanced Bag + Gradboost | 0.9468 | 0.9905 | 0.0441 | 0.0844 | 0.9539 | 0.6999 | 0.4981 | 0.224 | 0.0429 | 0.7569 | 0.6725 | 0.5400 | 0.0198 | 0.0382 | **0.673** | 14 |
| Balanced Bag + HistGradientBoost | 0.8164 | 0.6560 | 0.0276 | 0.0530 | 0.8245 | 0.6761 | 0.4942 | 0.0206 | 0.0395 | 0.7421 | 0.6727 | 0.5600 | 0.0187 | 0.0362 | 0.649 | 15 |
| Rusboost + AdaBoost | 0.8348 | 0.9120 | 0.0218 | 0.0426 | 0.8264 | 0.6478 | 0.6237 | 0.0148 | 0.0290 | 0.7155 | 0.6268 | 0.6600 | 0.0117 | 0.0230 | 0.662 | 16 |
| RF + SmoteENN | 0.8590 | 0.7081 | 0.0356 | 0.0677 | 0.8578 | 0.6798 | 0.4316 | 0.0213 | 0.0406 | 0.7532 | 0.6577 | 0.5000 | 0.0194 | 0.0373 | 0.663 | 17 |
| RF + SmoteTomek | 0.8666 | 0.6275 | 0.0410 | 0.0769 | 0.8687 | 0.6843 | 0.3572 | 0.0227 | 0.0427 | 0.7401 | 0.6189 | 0.3800 | 0.0191 | 0.0364 | 0.613 | 18 |
| RF + BLineSmote + RandUnderSample | 0.7984 | 0.1585 | 0.1018 | 0.1229 | 0.7835 | 0.6341 | 0.0237 | 0.0195 | 0.0210 | 0.6605 | 0.5308 | 0.0800 | 0.0308 | 0.0444 | 0.646 | 19 |
| Reg Logistique + SmoteENN | 0.7439 | 0.6644 | 0.0199 | 0.0387 | 0.7332 | 0.7133 | 0.6352 | 0.0190 | 357 | 0.7671 | 0.6636 | 0.6600 | 0.0143 | 0.0279 | **0.6904** | 20 |
| SVM(rbf) + SmoteENN | 0.9399 | 9384 | 0.0666 | 0.1244 | 0.9373 | 0.6400 | 0.2708 | 0.0191 | 0.0337 | 7199 | 0.5742 | 0.2787 | 0.0187 | 0.0351 | **0.6798** | 21 |
| KNN + SmoteENN | 0.9648 | 0.9535 | 0.0660 | 0.1234 | 0.9634 | 0.5931 | 0.2703 | 0.0165 | 0.0312 | 0.6255 | 0.5761 | 0.3000 | 0.0146 | 0.0278 | 0.605 | 22 |
| GaussianNB + SmoteENN | 0.5854 | 0.9384 | 0.0105 | 0.0208 | 0.6665 | 0.5715 | 0.9335 | 0.0105 | 0.0207 | 0.6418 | 0.5664 | 0.9400 | 0.0084 | 0.0167 | | 23 |

D'une manière générale les résultats obtenus sont satisfaisants, le roc_auc de la plupart des méthodes dépasse celui qui est obtenu actuellement par Valéo (63.5) et qui est calculé en se basant sur les abaques de mesures physiques.

Cependant, on constate qu'il y a un décalage entre les performances du split 'train' et celles de 'validation', ce qui signifie qu'il y a un overfit dans le choix des hyperparamètres. Plus la différence entre 'Train' et 'Validation' se réduit, plus l'algorithme a tendance à généraliser et d'être plus efficace sur un nouveau jeu de données, ce qui augmente la performance sur le split de l'ENS.

Les hyperparamètres ont été choisis suite à plusieurs essais itératifs selon la méthodologie suivante :

- Un premier choix des hyperparamètres est effectué après la compréhension de l'algorithme et l'évaluation de l'impact des paramètres suite à un 'train/test' simple.
- Une mise au point des hyperparamètres est effectuée après des séquences de 'cross_validate'.
- L'affinement de ce choix peut être effectué en appliquant une des méthodes d'optimisations des paramètres : BayesSearchCV, GridSearchCV ou RandomSearchCV.
  Cependant l'utilisation de GridSearchCV et RandomSearchCV n'est pas guidée par un algorithme intelligent pour trouver la meilleure combinaison d'hyperparamètres.

  GridSearchCV est consommatrice de ressources machine, elle essaie toutes les combinaisons spécifiées sans faire recours à aucune méthode d'optimisation.

RandomSearchCV essaie, selon une distribution, les combinaisons à partir d'une plage de paramètres spécifiées. Cette méthode ne fait pas recours non plus à une méthode d'optimisation.

On note la bonne généralisation du modèle basé sur la Régression Logistique avec du SomteEnn, les performances tain/validation et tests sont très proches les unes des autres, donc un choix optimal d'hyperparamètres, ce qui s'est traduit par le score le plus élevé au niveau du test de l'ENS (0.6904)

Dans le second tableau on distingue les spécificités des méthodes utilisées ainsi que le temps machine consomé en phases de 'training' et de 'validation'.

*Figure 24.d.2 - Autres Métriques*

| Méthodes | Temps machine par cyle de CV | | Bagging / Boosting SmoteBLine / R. undersampling SmoteEnn / SmoteTomek | Paramétrique Oui / Non | Lazy Eager | Discriminatif Generatif | Chap |
|---|---|---|---|---|---|---|---|
| | train(s) | valid(s) | | | | | |
| Balanced RF | 4.1627 | 0.3700 | bagging + under sampling natif à l'algorithme | Non | Eager | Discrim. | 12 |
| Balanced Bag + Adaboost | 21.6472 | 8.7849 | bagging + boosting + under sampling natif à l'algorithme | Non | Eager | Discrim. | 13 |
| Balanced Bag + Gradboost | 21.0091 | 3.7217 | bagging + boosting + under sampling natif à l'algorithme | Non | Eager | Discrim. | 14 |
| Balanced Bag + HistGradientBoost | 22.5675 | 6.0610 | boosting + under sampling natif à l'algorithme | Non | Eager | Discrim. | 15 |
| Rusboost + AdaBoost | 23.7319 | 4.1073 | bagging + boosting + under sampling natif à l'algorithme | Non | Eager | Discrim. | 16 |
| RF + SmoteENN | 29.6008 | 0.3572 | bagging + SmoteEnn | Non | Eager | Discrim. | 17 |
| RF + SmoteTomek | 23.8095 | 0.3581 | bagging + SmoteTomek | Non | Eager | Discrim. | 18 |
| RF + BlineSmote + RandUnderSampler | 3.8254 | 0.3894 | bagging + SmoteBLine + Random undersampling | Non | Eager | Discrim. | 19 |
| Reg Logistique + SmoteENN | 60.4653 | 0.1571 | SmoteEnn | Oui pour RL Non (SmtENN) | | Discrim. | 20 |
| SVM(rbf) + SmoteENN | 361.6414 | 3.5207 | SmoteEnn | Non | Lazy | Discrim. | 21 |
| KNN + SmoteENN | 25.8993 | 6.6173 | SmoteEnn | Non | Lazy | Discrim. | 22 |
| GaussianNB + SmoteENN | 24.2888 | 0.1463 | SmoteEnn | Oui pour GNB Non (SmtENN) | Eager | Generat. | 23 |

On distingue des temps d'entrainements et de validation très rapides pour les méthodes basées sur les arbres, notamment à la méthode 'Balanced Random Forest'.
Un temps d'entrainement très long pour la méthode SVM(rbf) et un temps de validation très court pour la Régression Logistique.

# 25 - Optimisation de paramètres avec BayesGridSearchCV

## a - Approche suivie

En vue d'appliquer une optimisation de paramètres, on avait le choix entre les 3 méthodes : GridSearchCV, RandomizedSearchCV et BayesSearchCV déjà exposés au chapitre 11 (paragraphes 'a' et 'd').
Comme ces méthodes sont consommatrices de temps machine, on va se limiter à l'usage de BayesGridSearchCV sur un des algorithmes étudiés :

- L'idée est de choisir un algorithme représentant un overfit.
- Puis appliquer un BayesSearchCV afin de trouver un modèle plus optimal, avec moins d'overfit et qui conserve ou améliore la performance du roc_auc.

Pour cela, on va choisir l'algorithme 'Balanced Bagging Classifier avec GradientBoost - Chapitre 14' qui représente un overfit important :

- 'roc_auc' de train: 0.9468
- 'roc_auc' de validation: 0.6999
- paramètres utilisés pour ces roc_auc:
  BalancedBaggingClassifier(
  base_estimator = GradientBoostingClassifier(learning_rate= 0.1, max_depth= 10,
  max_features= 'log2', min_samples_split= 18),
  n_estimators= 200, max_samples=0.7, max_features= 8, oob_score= True, replacement=True ,
  sampling_strategy= 'auto', n_jobs=-1)

## b - Mise en place de la solution

Pour mettre en place l'utilisation de BayesGridSearchCV, on va commencer par définir l'espace de paramètres qui sera parcouru par la fonction d'optimisation bayésienne.

Ci suit un extrait de 'ClassifierParam.py' du module 'valeo.domain', une classe conçue pour la définition des différentes combinaisons de paramètres en fonction de l'algorithme à utiliser et de la méthode (bayes, grid, randomized).
Pour le choix des paramètres initiaux de BayesSearchCV, on propose des intervalles de valeurs entourant les valeurs déjà choisies pour 'BBC_GBC - Chap 14'.

```
self.o_param[C.BBC_GBC] = {
        'classifier__base_estimator__learning_rate': Real(0.1, 10),
        'classifier__base_estimator__n_estimators': Integer(50,100),
        'classifier__base_estimator__max_depth' : Integer(5,10),
        'classifier__base_estimator__max_features' : ['sqrt', 'log2'],
        'classifier__base_estimator__min_samples_split' :Integer(15, 25),
        'classifier__n_estimators' : Integer(10,200),
        'classifier__max_samples' : Real(0.5, 0.7),
        'classifier__max_features' : Integer(5,30),
```

'classifier__oob_score' : [True, False] }

## c - Résultats et comparaisons

Au bout de 56 minutes de traitement (30 itérations), la méthode a identifié les paramètres suivants :

Best Params: OrderedDict([
('classifier__base_estimator__learning_rate', 0.1),
('classifier__base_estimator__n_estimators', 50),
('classifier__base_estimator__max_depth', 5),
('classifier__base_estimator__max_features', 'log2'),
('classifier__base_estimator__min_samples_split', 25),
('classifier__n_estimators', 200),
('classifier__max_samples', 0.7),
('classifier__max_features', 30),
('classifier__oob_score', True)])

Comparatif des paramètres - Tableau 24.c.1 :

- *Tableau 24.c.1*

| Paramètres | Espace de param. | Best params | Params du BBC_GBC - Chap 14 |
|---|---|---|---|
| classifier__base_estimator__learning_rate | Real(0.1, 10) | 0.1 | 0.1 |
| classifier__base_estimator__n_estimators | Integer(50,100) | 50 | par défaut(100) |
| classifier__base_estimator__max_depth | Integer(5,10) | 5 | 10 |
| classifier__base_estimator__max_features | ['sqrt', 'log2'] | log2 | log2 |
| classifier__base_estimator__min_samples_split | Integer(15, 25) | 25 | 18 |
| classifier__n_estimators | Integer(10,200) | 200 | 200 |
| classifier__max_samples | Real(0.5, 0.7) | 0.7 | 0.7 |
| classifier__max_features | Integer(5,30) | 30 | 8 |
| classifier__oob_score | [True, False] | True | True |

Comparatif des splits Train et Validation - Tableau 24.c.2 :

- *Tableau 24.c.2*

| Valeur | BayesSearchCV best param | BBC_GBC - Chap 14 |
|---|---|---|
| Moy. ROC_AUC Train | 0.8552 | 0.9468 |
| Moy. ROC_AUC Validation | 0.7066 | 0.6999 |
| Temps(s) Train | 10.8653 | 21.0091 |
| Temps(s) Validation | 0.8941 | 3.7217 |


| Valeur | BayesSearchCV best param | BBC_GBC - Chap 14 |
|---|---|---|
| Moy. ROC_AUC Train | 0.8552 | 0.9468 |
| Moy. ROC_AUC Validation | 0.7066 | 0.6999 |
| Temps(s) Train | 10.8653 | 21.0091 |

| | | |
|---|---|---|
| Temps(s) Validation | 0.8941 | 3.7217 |

Comparatif des performances sur split Test - Tableau 24.c.3 :

- *Tableau 24.c.3*

| Valeur | BayesSearchCV best param | BBC_GBC - Chap 14 |
|---|---|---|
| ROC_AUC | 0.6828 | 0.6725 |
| Recall | 0.6000 | 0.5400 |
| Precision | 0.0183 | 0.0198 |
| F1 | 0.0356 | 0.0382 |
| Matrice de confusion | [5246 1607] | [5516 1337] |
| | [20 30] | [23 27] |

## 26 - Conclusion et perspective

Ce projet de classification à partir d'un jeu de données déséquilibrées vient clore mon cursus de Data Science à l'Ecole Polytechnique Executive Education. Ce projet était une occasion de travailler plusieurs facettes du problème, monter en compétence en Data Science et acquérir de nouvelles connaissances. J'en suis ravi.

Lors de mon étude, je me suis posé des questions autour de la pertinence industrielle du problème qu'on tente de résoudre.
Valéo dispose déjà d'un banc de tests pour évaluer et classifier les démarreurs. Quelle plus-value lui apporterait alors la mise en place d'une nouvelle méthode basée sur les algorithmes des sciences de données ?
Le démarreur étant déjà produit, la nouvelle méthode viendra donc comme outil d'aide à la décision aux contrôleurs de qualité.

J'ai réécouté à plusieurs reprises la présentation de Valéo au Collège de France http://www.college-de-france.fr/video/stephane-mallat/2020/06-sem-mallat-challenge-valeo-20200129-06.mp4 afin de deviner la forte plus-value que Valéo tente à trouver.
Afin de mieux comprendre, j'ai contacté par e-mail le responsable du pôle intelligence artificielle de l'unité de production de Valéo et nous avons échangé quelques emails.

**Ce qui intéresse Valéo est d'identifier les facteurs/les mesures qui influent sur le résultat afin de mieux régler les postes de production en amont.**

La réponse de Valéo a fortement résonné dans ma tête. La réponse est claire mais il fallait y penser.
De mon côté j'ai fait ressortir les facteurs influents à travers '*fitted_model.feature_importances_*', mais il fallait remonter la chaîne des transformations afin d'identifier les features sous leurs formes d'entrée.
Je compte revenir sur ce point et l'élucider.