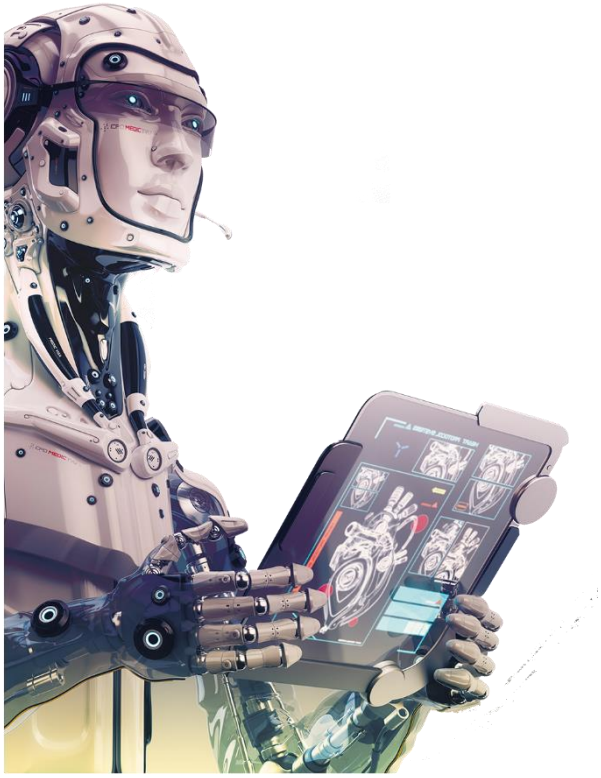
Royaume du Maroc
Haut-Commissariat au Plan (HCP)
Ecole des Sciences de l'Information (ESI)

Projet Data Mining :

L'Utilisation du Machine Learning pour Prédire les Maladies Cardiovasculaires



Groupe :

Fatima Ezzahra Assane

Achraf El Maouaj

Amine Salim

Issam Zaidouhadou

Encadrante :

Mme. Najima Daoudi

Table des Matières :

Introduction :

Dans ce projet nous introduisons des concepts de Machine Learning et de Data Science en explorant le problème de classification par rapport aux maladies cardiovasculaires.

La classification consiste à décider si un échantillon fait partie d'une classe ou une autre, et on parle dans ce cas de la classification uni-classe. Cependant, Si nous avons plusieurs options de classes, il s'agit d'une classification multi-classes.

1. La Définition du Problème :

Dans notre cas, il s'agit d'une classification binaire (les variables se sont labilisé donc Supervised Learning / la variable réponse (target) est discréte (2 cas)). En effet, nous utiliserons plusieurs features afin de prédire si ou non, une personne a une maladie cardiovasculaire. Plus précisément, répondre à la question ; étant donné les paramètres cliniques d'un patient, pouvons-nous prédire s'il souffre ou non d'une maladie cardiaque ?

2. La base de données :

La source de la base de données originale est Cleveland Database de UCI Machine Learning Repository. Toutefois, Nous avons télécharger sa forme formatée à partir de Kaggle.

La base de données original contient 76 attributs, mais dans notre base de données nous avons les 14 attributs qui vont nous servir le plus à prédire la variable cible.

3. Points abordés dans le projet :

Analyse exploratoire de données : dans cette étape nous allons naviguer à travers la base de données et la découvrir.

- **Le Training du modèle** : créer un ou des modèles qui apprennent à prédire une variable cible en se basant sur d'autres variables.
- **L'évaluation du modèle** : l'évaluation de la prédiction des modèles en utilisant les métriques d'évaluation.
- **La comparaison des modèles** : comparer entre plusieurs modèles afin de trouver le meilleur.
- **Le tuning du modèle** : dès que nous trouverons un bon modèle. Comment peut-on l'améliorer ?

- **L’importance des features** : En termes de prédiction, est-ce qu’il y a des features plus importantes que d’autres ?
- **Cross-validation** : Si nous avons créé un bon modèle, est-ce qu’on peut garantir qu’il fonctionnera dans d’autres bases de données.
- **Le Reporting** : Comment nous allons présenter nos résultats ?
- **L’évaluation** : Si nous pouvons atteindre une précision de 95 % pour prédire si un patient souffre ou non d'une maladie cardiaque pendant la phase de validation du concept, nous assurerons ce projet.

4. Le dictionnaire des variables :

1. **age** : l’âge en années.
2. **genre** : (1 = male, 0 = female).
3. **td** : type de la douleur thoracique :
    - 0 = Angine typique : douleur thoracique liée à la circulation sanguine ;
    - 1 = Angine Atypique : douleur thoracique indépendante du cœur ;
    - 2 = Douleur non angineuse : spasmes œsophagiens typique (indépendante du cœur) ;
    - 3 = Asymptomatique.
4. **psr** : pression sanguine au repos (en mm Hg), toute valeur supérieure à 130-140 représente un risque.
5. **chol** : cholestérol du sérum en mg/dl, toute valeur supérieure à 200 représente un risque.
6. **gaj** : glycémie à jeun en mg/dl (1 = true, 0 = false), toute valeur supérieure à 126 signifie le diabète.
7. **ecgr** : résultats de l’électrocardiogramme de repos :
    - 0 = Rien à noter ;
    - 1 = Anomalie de l’onde ST-S ;
    - 2 = Hypertrophie ventriculaire gauche (HVG).
8. **fcm** : fréquence cardiaque maximale.
9. **effang** : l'angine de poitrine provoquée par l'effort (1 = yes, 0 = no).
10. **dinde** : Dépression ST induite par l'effort par rapport à celle au repos
11. **penteff** : la pente du segment ST de l'effort de pointe.
    - 0 = Meilleur rythme cardiaque avec l'effort (peu fréquent)

- 1 = Changement minimal (cœur sain typique)
- 2 = Signes d'un cœur malade

12. **nvc** : nombre de grands vaisseaux (0-3) colorés par fluoroscopie.
13. **rshal** : résultat du stress au thallium.
    - 1, 3 = Normal ;
    - 6 = Défaut corrigé : c'était un défaut avant, mais c'est bon maintenant ;
    - 7 = Défaut réversible : pas de circulation sanguine correcte lors de l'effort.
14. **target** : avoir ou non une maladie cardiovasculaire (1=Yes, 0= No) (= l'attribut prédit).

5. Code, Sortie et Interprétation :

5.1. L'importation des bibliothèques :

Entrée [1]:
```
#Bibliothèques pour l'analyse exploratoire
import numpy as np #pour les opérations numériques
import pandas as pd #Pandas pour l'analyse de données

#Bibliothèques pour la visualisation de données
import matplotlib.pyplot as plt
import seaborn as sns

#Pour que les figures apparaissent dans le notebook
%matplotlib inline

#Pour créer les modèles
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

#Pour évaluer les modèles
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import plot_roc_curve
```

5.2. L'importation de la base de données :

Entrée [2]:
```
#Importation de la base de données
cardio=pd.read_csv('Maladie_Cardio.csv')

#Afficher les dimension de la base de données
cardio.shape # (lignes, colonnes)
```

Out[2]: (303, 14)

5.3. Analyse exploratoire des données :

Entrée [3]:
```
#L'analyse exploratoire des données
#Afficher les 5 premieres lignes
cardio.head()
```

Out[3]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

value_counts() nous permet de montrer combien de fois apparaît chacune des valeurs de la variable « target ».

Entrée [5]:
```
#Afficher combien des 1 et des 0 nous avons dans la variable target
cardio.target.value_counts()
```

Out[5]:
```
1    165
0    138
Name: target, dtype: int64
```

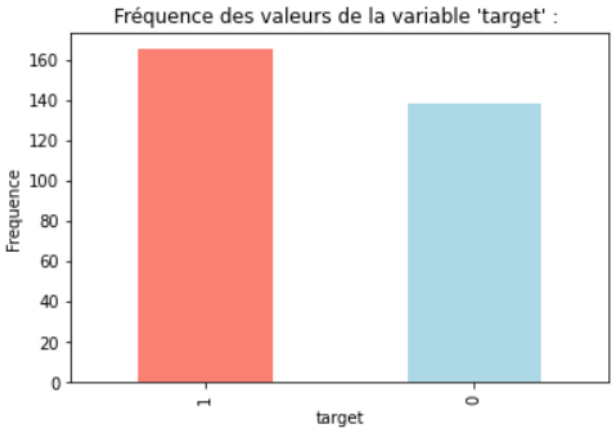
Entrée [6]:
```
#Afficher les pourcentage de chaque valeur de la variable target (Normaliser)
cardio.target.value_counts(normalize=True)
```

Out[6]:
```
1    0.544554
0    0.455446
Name: target, dtype: float64
```

Comme ces deux valeurs sont proches, notre colonne cible « target » peut être considérée comme équilibrée. Une colonne cible déséquilibrée, c'est-à-dire que certaines classes ont beaucoup plus d'échantillons, ce qui peut être plus difficile à modéliser qu'un ensemble équilibré. Idéalement, toutes les classes cibles doivent avoir le même nombre d'échantillons.

Nous pouvons visualiser les valeurs de la colonne « target » en appelant la fonction *plot()* et en précisant quel type de graphe nous voulons, dans ce cas, un graphique à barres.

Entrée [7]:
```
#Afficher un graphe à barres representant les valeurs de la variable target
plt.xlabel('target')
plt.ylabel('Frequence')
plt.title("Fréquence des valeurs de la variable 'target' : ")
cardio.target.value_counts().plot(kind='bar', color=['salmon', 'lightblue'])
plt.show()
```



info() nous donne un aperçu du nombre de valeurs manquantes dont nous disposons et du type de données avec lesquelles nous travaillons.

Dans notre cas, il n'y a pas de valeurs manquantes et toutes nos colonnes sont de nature numérique.

Entrée [8]:
```
cardio.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   age      303 non-null    int64
 1   genre    303 non-null    int64
 2   td       303 non-null    int64
 3   psr      303 non-null    int64
 4   chol     303 non-null    int64
 5   gaj      303 non-null    int64
 6   ecgr     303 non-null    int64
 7   fcm      303 non-null    int64
 8   effang   303 non-null    int64
 9   dinde    303 non-null    float64
 10  penteff  303 non-null    int64
 11  nvc      303 non-null    int64
 12  rshal    303 non-null    int64
 13  target   303 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

Une autre façon d'obtenir des informations sur nos données est d'utiliser *describe()*. *describe()* montre une série de mesures différentes sur les colonnes numériques telles que la moyenne, le maximum et l'écart type.

```
Entrée [9]: cardio.describe()
```

Out[9]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.313531
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.612277
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.000000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.000000

5.4. Fréquence des maladies cardiaques selon le genre :

Comparons notre colonne cible « target » avec la colonne genre.

Rappelons notre dictionnaire de données, pour la colonne « target », 1 = présence d'une maladie cardiaque, 0 = aucune maladie cardiaque. Et pour le genre, 1 = male, 0 = female.

```
Entrée [10]: #Afficher combien des 1 et des 0 nous avons dans la variable 'genre'
             cardio.genre.value_counts()
```

```
Out[10]: 1    207
         0     96
         Name: genre, dtype: int64
```

Il y a 207 hommes et 96 femmes dans notre étude.

```
Entrée [11]: #Etudions la relation entre les variables 'target' et 'genre'
             pd.crosstab(cardio.target, cardio.genre)
```

Out[11]:

genre	0	1
target		
0	24	114
1	72	93

Comme il y a environ 100 femmes et que 72 d'entre elles ont une valeur positive de la présence d'une maladie cardiaque, nous pourrions déduire, sur la base de cette seule variable si le participant est une femme, qu'il y a 75% de chances qu'elle ait une maladie cardiaque.

Quant aux hommes, il y en a environ 200 au total, dont la moitié environ indique la présence d'une maladie cardiaque. Nous pourrions donc prédire, si le participant est un homme, qu'il aura une maladie cardiaque dans 50% des cas.

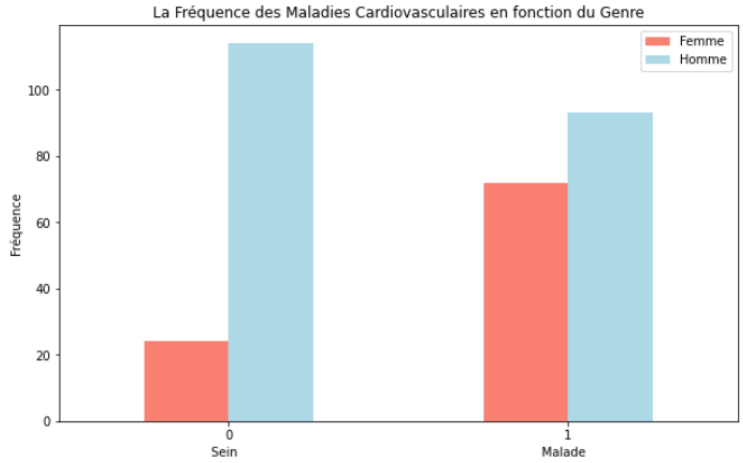
En faisant la moyenne de ces deux valeurs, nous pouvons supposer, sur la base d'aucun autre paramètre, que si une personne est vivante, il y a 62,5 % de chances qu'elle ait une maladie cardiaque.

Cela peut être notre base de référence très simple, nous allons essayer de la surpasser avec le Machine Learning.

Visualisons le tableau croisé que nous avons obtenu :

Entrée [12]:
```
#Modeliser le tableau ci-dessus

pd.crosstab(cardio.target, cardio.genre).plot(ylabel ="Fréquence",kind='bar',
                                              figsize=(10,6),
                                              color=["salmon", "lightblue"])

plt.title("La Fréquence des Maladies Cardiovasculaires en fonction du Genre")
plt.xlabel("Sein                                                                    Malade")
plt.ylabel("Fréquence")
plt.legend(["Femme", "Homme"])
plt.xticks(rotation=0)
```

Out[12]:
```
(array([0, 1]), [Text(0, 0, '0'), Text(1, 0, '1')])
```



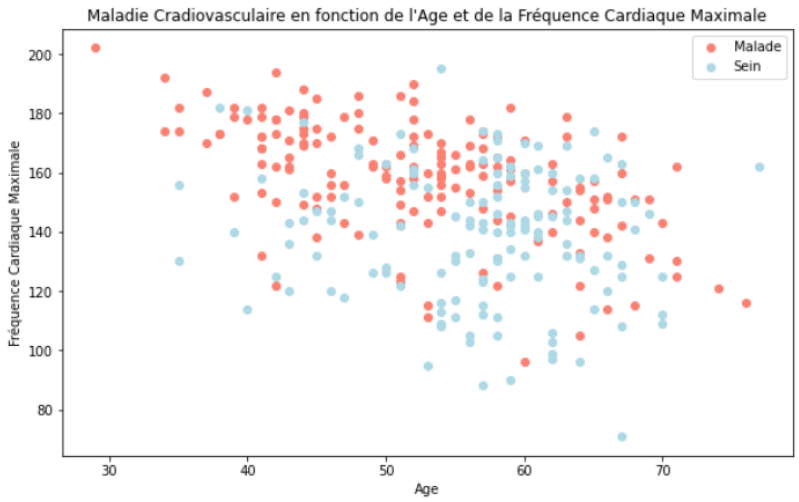
5.5. Les maladies cardiaques en fonctions de l'âge et la fréquence cardiaque maximale :

Essayons de combiner quelques variables indépendantes, telles que l'âge et la fréquence cardiaque maximale (fcm), puis de les comparer à notre variable « target », la maladie cardiaque.

Comme il y a tant de valeurs différentes pour l'âge et la fréquence cardiaque maximale, nous utiliserons un scatter plot.

Entrée [13]:
```
#Etudions la relation en entre l'age et la fréquence cardiaque maximale
plt.figure(figsize=(10,6))

#Considerons les examples positives
plt.scatter(cardio.age[cardio.target==1],
            cardio.fcm[cardio.target==1],
            c='salmon')
#Considérons les examples négative toujours dans le même graphe
plt.scatter(cardio.age[cardio.target==0],
            cardio.fcm[cardio.target==0],
            c='lightblue')

plt.title("Maladie Cradiovasculaire en fonction de l'Age et de la Fréquence Cardiaque Maximale")
plt.xlabel("Age")
plt.ylabel("Fréquence Cardiaque Maximale")
plt.legend(["Malade", "Sein"])
```

Out[13]: <matplotlib.legend.Legend at 0x1a9a1d0a488>



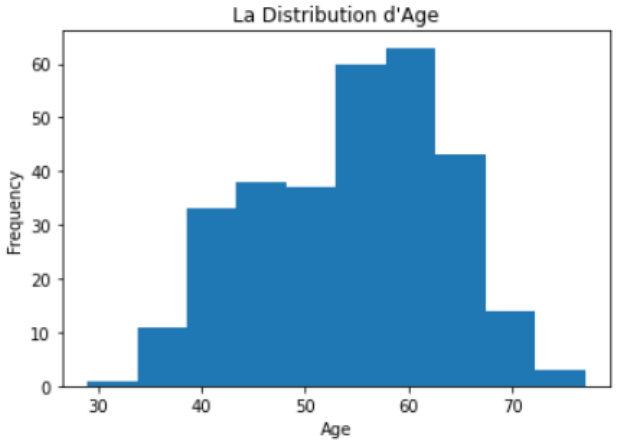
Il semble que plus une personne est jeune, plus sa fréquence cardiaque maximale est élevée (les points rouges sont plus hauts à gauche du graphique) et plus une personne est âgée, plus il y a de points bleus. Mais cela peut être dû au fait qu'il y a plus de points sur le côté droit du graphique (participants plus âgés).

Ces deux facteurs sont bien sûr des facteurs d'observation, mais c'est ce que nous essayons de faire, en d'autres termes comprendre les données.

5.6. La distribution d'âge :

Entrée [14]:
```
#Etudions la distribution de l'age

cardio.age.plot.hist()
plt.title('La Distribution d\'Age')
plt.xlabel('Age')
```

Out[14]: Text(0.5, 0, 'Age')



Nous pouvons voir qu'il s'agit d'une distribution normale, mais qui oscille légèrement vers la droite, ce qui se reflète dans le scatter plot ci-dessus.

5.7. La fréquence des maladies cardiaques en fonction du type de la douleur thoracique :

Nous utiliserons le même procédé qu'auparavant pour le genre.

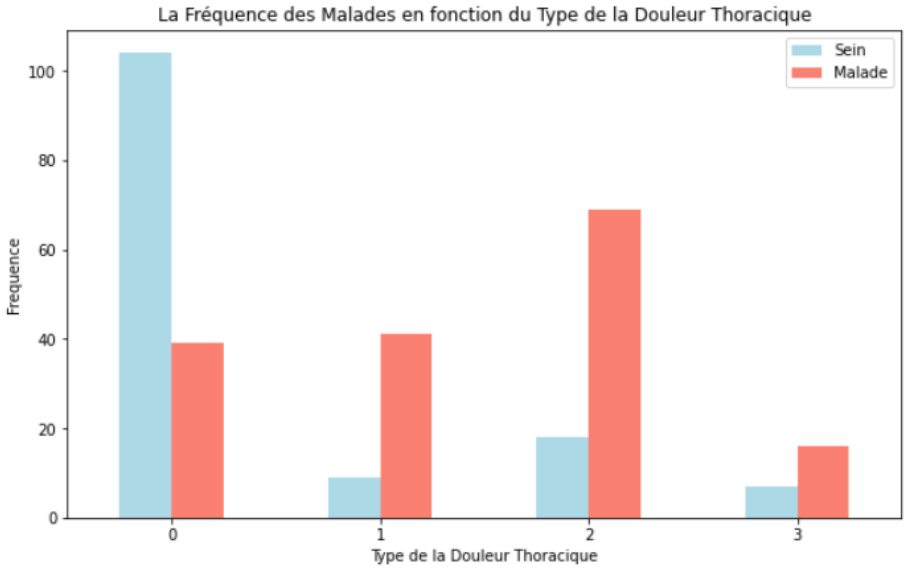
Entrée [15]:
```
#Etudions la relation entre les variables 'target' et 'td (type de la douleur thoracique)'
pd.crosstab(cardio.td, cardio.target)
```

Out[15]:

target	0	1
td		
0	104	39
1	9	41
2	18	69
3	7	16

Entrée [16]:
```
#Modélisons les deux variables
pd.crosstab(cardio.td, cardio.target).plot(kind='bar',
                                           figsize=(10,6),
                                           color=["lightblue", "salmon"])

#Ajoutons les descriptions
plt.title("La Fréquence des Malades en fonction du Type de la Douleur Thoracique")
plt.xlabel("Type de la Douleur Thoracique")
plt.ylabel("Frequence")
plt.legend(["Sein", "Malade"])
plt.xticks(rotation = 0)
```

Out[16]:
```
(array([0, 1, 2, 3]),
 [Text(0, 0, '0'), Text(1, 0, '1'), Text(2, 0, '2'), Text(3, 0, '3')])
```



Rappelons notre dictionnaire de données :

td - type de douleur thoracique

0 : Angine typique : douleur thoracique liée à une diminution de l'apport sanguin au cœur

1 : Angine atypique : douleur thoracique sans rapport avec le cœur

2 : Douleur non angineuse : spasmes oesophagiens typiques (non liés au cœur)

3 : Asymptomatique : douleur thoracique ne présentant pas de signes de maladie

Il est intéressant de noter que l'angine atypique (valeur 1) indique qu'elle n'est pas liée au cœur, mais semble avoir un pourcentage plus élevé chez les personnes souffrant de maladies cardiaques que celles qui n'en souffrant pas.

Selon PubMed, il semble que même certains professionnels de la santé soient confus par ce terme.

« Aujourd'hui, 23 ans plus tard, les "douleurs thoraciques atypiques" sont toujours populaires dans les milieux médicaux. Sa signification reste cependant floue. Quelques articles ont le terme dans leur titre, mais ne le définissent pas et n'en parlent pas dans leur texte. Dans d'autres articles, le terme fait référence aux causes non cardiaques des douleurs thoraciques. »

Bien qu'il ne soit pas concluant, le graphique ci-dessus est un indice de la confusion des définitions représentées dans les données.

5.8. Corrélation entre les variables indépendantes :

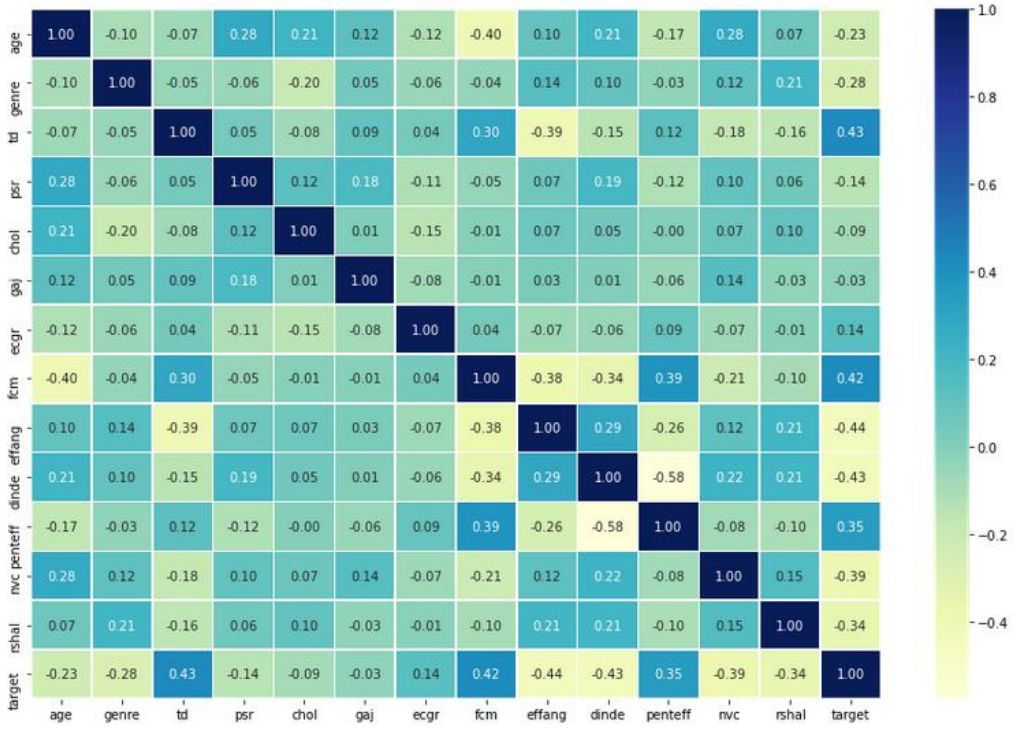
Entrée [17]:
```
#La correlation entre les variables
corr_matrix = cardio.corr()
corr_matrix
```

Out[17]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal	target
age	1.000000	-0.098447	-0.068653	0.279351	0.213678	0.121308	-0.116211	-0.398522	0.096801	0.210013	-0.168814	0.276326	0.068001	-0.225439
enre	-0.098447	1.000000	-0.049353	-0.056769	-0.197912	0.045032	-0.058196	-0.044020	0.141664	0.096093	-0.030711	0.118261	0.210041	-0.280937
td	-0.068653	-0.049353	1.000000	0.047608	-0.076904	0.094444	0.044421	0.295762	-0.394280	-0.149230	0.119717	-0.181053	-0.161736	0.433798
psr	0.279351	-0.056769	0.047608	1.000000	0.123174	0.177531	-0.114103	-0.046698	0.067616	0.193216	-0.121475	0.101389	0.062210	-0.144931
chol	0.213678	-0.197912	-0.076904	0.123174	1.000000	0.013294	-0.151040	-0.009940	0.067023	0.053952	-0.004038	0.070511	0.098803	-0.085239
gaj	0.121308	0.045032	0.094444	0.177531	0.013294	1.000000	-0.084189	-0.008567	0.025665	0.005747	-0.059894	0.137979	-0.032019	-0.028046
ecgr	-0.116211	-0.058196	0.044421	-0.114103	-0.151040	-0.084189	1.000000	0.044123	-0.070733	-0.058770	0.093045	-0.072042	-0.011981	0.137230
fcm	-0.398522	-0.044020	0.295762	-0.046698	-0.009940	-0.008567	0.044123	1.000000	-0.378812	-0.344187	0.386784	-0.213177	-0.096439	0.421741
fang	0.096801	0.141664	-0.394280	0.067616	0.067023	0.025665	-0.070733	-0.378812	1.000000	0.288223	-0.257748	0.115739	0.206754	-0.436757
nde	0.210013	0.096093	-0.149230	0.193216	0.053952	0.005747	-0.058770	-0.344187	0.288223	1.000000	-0.577537	0.222682	0.210244	-0.430696
nteff	-0.168814	-0.030711	0.119717	-0.121475	-0.004038	-0.059894	0.093045	0.386784	-0.257748	-0.577537	1.000000	-0.080155	-0.104764	0.345877
nvc	0.276326	0.118261	-0.181053	0.101389	0.070511	0.137979	-0.072042	-0.213177	0.115739	0.222682	-0.080155	1.000000	0.151832	-0.391724
shal	0.068001	0.210041	-0.161736	0.062210	0.098803	-0.032019	-0.011981	-0.096439	0.206754	0.210244	-0.104764	0.151832	1.000000	-0.344029
rget	-0.225439	-0.280937	0.433798	-0.144931	-0.085239	-0.028046	0.137230	0.421741	-0.436757	-0.430696	0.345877	-0.391724	-0.344029	1.000000

```
Entrée [18]: #Modélisons le résultat ci-dessus
corr_matrix = cardio.corr()
plt.figure(figsize=(15, 10))
sns.heatmap(corr_matrix,
            annot=True,
            linewidths=0.5,
            fmt=".2f",
            cmap="YlGnBu")
```

```
Out[18]: <AxesSubplot:>
```



Une valeur positive plus élevée signifie une corrélation positive potentielle (augmentation) et une valeur négative plus élevée signifie une corrélation négative potentielle (diminution).

5.9. Le Modeling :

Nous avons exploré les données, nous allons maintenant essayer d'utiliser le Machine Learning pour prédire notre variable cible sur la base des 13 variables indépendantes. Rappelons notre problématique : Étant donné les paramètres cliniques d'un patient, pouvons-nous prédire s'il souffre ou non d'une maladie cardiaque ? Et rappelons notre mesure d'évaluation ; si nous pouvons atteindre une précision de 95 % pour prédire si un patient souffre ou non d'une maladie cardiaque pendant la phase de validation du concept, nous assurerons ce projet. C'est ce que nous viserons.

Mais avant de construire un modèle, nous devons préparer notre base de données. Nous essayons de prévoir notre variable cible en utilisant toutes les autres variables. Pour ce faire, nous allons séparer la variable cible du reste.

Entrée [20]:
```python
#Séparons la variable 'target' du reste des variables
X = cardio.drop("target", axis=1)

#La variable 'target'
y=cardio.target.values
```

Entrée [21]:
```python
#(Pas de variable 'target')
X.head()
```

Out[21]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2

Entrée [22]:
```python
#Les Cibles
y
```

Out[22]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int6
4)
```

5.10. Répartition des données en données du training et données de test :

Le paramètre test_size est utilisé pour indiquer à la fonction train_test_split() la quantité de données que nous souhaitons utilisé dans le test. La règle générale est d'utiliser 80 % des données pour le training et les 20 % restants pour le test.

Entrée [23]:
```
#Separons les données en deux, une partie pour le training(80%) et une autre(20%) pour le test

#random.seed pour la reproductibilité
np.random.seed(42)

#Separons la dataset (train et test)
X_train, X_test, y_train, y_test = train_test_split(X, #Variables indépendantes
                                                    y, #Variable dépendante
                                                    test_size = 0.2) #pourcentage des données à utliser dans le traini
```

Entrée [24]:
```
X_train.head()
```

Out[24]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal
132	42	1	1	120	295	0	1	162	0	0.0	2	0	2
202	58	1	0	150	270	0	0	111	1	0.8	2	0	3
196	46	1	2	150	231	0	1	147	0	3.6	1	0	2
75	55	0	1	135	250	0	0	161	0	1.4	1	0	2
176	60	1	0	117	230	1	1	160	1	1.4	2	2	3

```
Entrée [25]: y_train, len(y_train)
```

```
Out[25]: (array([1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0,
        1,
               1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0,
        0,
               1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,
        1,
               0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1,
        0,
               0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1,
        0,
               1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0,
        1,
               1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1,
        1,
               1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1,
        0,
               0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1,
        1,
               1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0,
        1,
               1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0,
        1],
              dtype=int64),
 242)
```

```
Entrée [26]: X_test.head()
```

Out[26]:

	age	genre	td	psr	chol	gaj	ecgr	fcm	effang	dinde	penteff	nvc	rshal
179	57	1	0	150	276	0	0	112	1	0.6	1	1	1
228	59	1	3	170	288	0	0	159	0	0.2	1	0	3
111	57	1	2	150	126	1	1	173	0	0.2	2	1	3
246	56	0	0	134	409	0	0	150	1	1.9	1	2	3
60	71	0	2	110	265	1	0	130	0	0.0	2	1	2

```
Entrée [27]: y_test, len(y_test)
```

```
Out[27]: (array([0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1,
        0,
               0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
        1,
               1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0], dtype=int6
        4),
 61)
```

5.11. Le choix du modèle :

Maintenant que nous avons préparé nos données, nous pouvons commencer à ajuster les modèles. Nous allons utiliser les éléments suivants et comparer leurs résultats :

- Régression logistique - LogisticRegression()
- K-Nearest Neighbors - KNeighboursClassifier()
- RandomForest - RandomForestClassifier()

Entrée [28]:
```
#Le choix du modèle

#Mettons les modèles dans un dictionnaire
models={'KNN' : KNeighborsClassifier(),
        'Logistic Regression' : LogisticRegression(),
        'Random Forest' : RandomForestClassifier()}

#Créer une fonction réalisons le fit et le score des modèles
def fit_and_score(models, X_train, X_test, y_train, y_test):
    """
    Ajuste et evalue des modèles de machine learning donnés
    models : un dictionnaire pour les differents modèles de Scikit-Lea
    X_train : les données du training
    X_test : les données du test
    y_train : labels associés avec les données du training
    y_test : labels associés avec les données du test
    """
    #Random seed pour des résultats reproductible
    np.random.seed(42)
    #liste pour enregistrer les scores des modèles
    model_scores = {}
    #Boucler sur les modèles
    for name, model in models.items():
        #fit le modèle pour la data
        model.fit(X_train, y_train)
        #Evaluer le modèle et ajouter son score à model_scores
        model_scores[name] = model.score(X_test, y_test)
    return model_scores
```

Entrée [29]:
```
model_scores = fit_and_score(models=models,
                             X_train=X_train,
                             X_test=X_test,
                             y_train=y_train,
                             y_test=y_test)
model_scores
```

```
D:\Programmes\anaconda3\lib\site-packages\sklearn\linear_model\_logist
ic.py:764: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as show
n in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic
-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

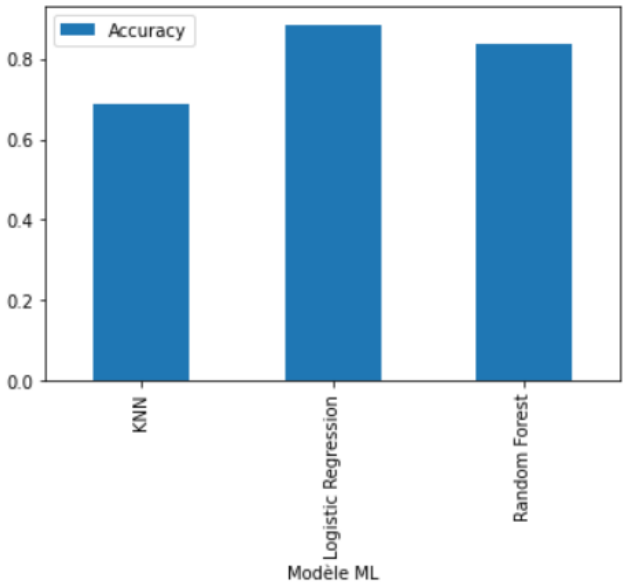
Out[29]:
```
{'KNN': 0.6885245901639344,
 'Logistic Regression': 0.8852459016393442,
 'Random Forest': 0.8360655737704918}
```

5.12. Comparaison des modèles :

Comme nous avons enregistré les scores de nos modèles dans un dictionnaire, nous pouvons les visualiser en les convertissant d'abord en un DataFrame.

```
Entrée [30]: model_compare = pd.DataFrame(model_scores, index=['Accuracy'])
             model_compare.T.plot.bar(xlabel = 'Modèle ML')
```

```
Out[30]: <AxesSubplot:xlabel='Modèle ML'>
```



5.13. Le tuning des hypermètres et la cross-validation :

Voici le plan que nous allons aborder :

- Ajuster les hyperparamètres du modèle, voir lequel est le plus performant ;
- Effectuer une validation croisée ;
- Tracez les courbes ROC ;
- Faire une matrice de confusion ;
- Obtenir des mesures de précision (accuracy), de rappel et de score F1 ;
- Trouvez les caractéristiques les plus importantes du modèle.

5.13.1. Le tuning du KneighborsClassifier :

Il y a un hyperparamètre principal que nous pouvons régler pour l'algorithme K-Nearest Neighbors (KNN), et c'est le nombre de voisins. La valeur par défaut est 5 (n_neigbors=5).

Pour l'instant, essayons quelques valeurs différentes de n_neighbors.

Entrée [31]:
```
#L'amélioration (tuning) des hyperparametres et la cross-validation

#Améliorer le KNN (Tuning du KNN)

#Créons une liste des scores du training
train_scores = []

#Créons une liste des scores du test
test_scores = []

#Créons une liste des différentes valeurs pour les n_neighbors
neighbors = range(1, 21)

#Définir l'algorithme
knn = KNeighborsClassifier()

#Boucler sur les différentes valeurs de neighbors
for i in neighbors :
    knn.set_params(n_neighbors = i) #Définir la valeur de neighbors

    #Fit
    knn.fit(X_train, y_train)

    #Mettre à jour les scores du training
    train_scores.append(knn.score(X_train, y_train))

    #Mettre à jour les scores du test
    test_scores.append(knn.score(X_test, y_test))
```

Visualisons les scores du test et du training de KNN.

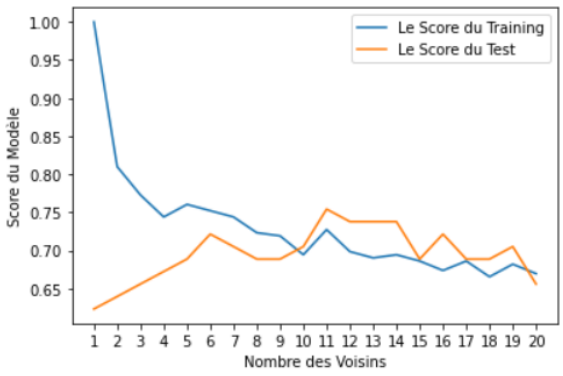
Entrée [32]:
```
train_scores
```

Out[32]:
```
[1.0,
 0.8099173553719008,
 0.7727272727272727,
 0.743801652892562,
 0.7603305785123967,
 0.7520661157024794,
 0.743801652892562,
 0.7231404958677686,
 0.71900826446281,
 0.6942148760330579,
 0.7272727272727273,
 0.6983471074380165,
 0.6900826446280992,
 0.6942148760330579,
 0.6859504132231405,
 0.6735537190082644,
 0.6859504132231405,
 0.6652892561983471,
 0.6818181818181818,
 0.6694214876033058]
```

```
Entrée [33]: test_scores

Out[33]: [0.6229508196721312,
          0.639344262295082,
          0.6557377049180327,
          0.6721311475409836,
          0.6885245901639344,
          0.7213114754098361,
          0.7049180327868853,
          0.6885245901639344,
          0.6885245901639344,
          0.7049180327868853,
          0.7540983606557377,
          0.7377049180327869,
          0.7377049180327869,
          0.7377049180327869,
          0.6885245901639344,
          0.7213114754098361,
          0.6885245901639344,
          0.6885245901639344,
          0.7049180327868853,
          0.6557377049180327]
```

```
Entrée [33]: #Modeliser les valeurs ci-dessus

plt.plot(neighbors, train_scores, label='Le Score du Training')
plt.plot(neighbors, test_scores, label='Le Score du Test')
plt.xticks(np.arange(1,21,1))
plt.xlabel("Nombre des Voisins")
plt.ylabel("Score du Modèle")
plt.legend()

print(f'Le Score KNN maximum sur la dataset du test est: {max(test_scores)*100:.2f}%')
```

```
Le Score KNN maximum sur la dataset du test est: 75.41%
```



En regardant le graphique, n_neighbors = 11 semble le mieux. Même en sachant cela, la performance du modèle KNN n'a pas été aussi bonne que celle de LogisticRegression ou du RandomForestClassifier. Pour cette raison, nous allons écarter KNN et nous concentrer sur les deux autres. Nous avons réglé KNN à la main mais voyons comment nous pouvons utiliser LogisticsRegression et RandomForestClassifier en se basant sur RandomizedSearchCV.

Au lieu de devoir essayer manuellement différents hyperparamètres à la main, RandomizedSearchCV essaie un certain nombre de combinaisons différentes, les évalue et sauvegarde la meilleure.

5.13.2. Le tuning des modèles en utilisant RandomizedSearchCV :

En lisant la documentation de Scikit-Learn sur LogisticRegression, nous constatons qu'il existe un certain nombre d'hyperparamètres différents que nous pouvons fixer. Il en va de même pour le RandomForestClassifier.

Créons un dictionnaire de différents hyperparamètres pour chacun des modèles et testons-les.

Entrée [34]:
```
#Améliorer le modèle par RandomizedSearchCV

#Differents hyperparametres de LogisticRegression
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                "solver": ["liblinear"]}

#Differents hyperparametres de RandomForrestClassifier
rf_grid = {"n_estimators" : np.arange(10, 1000, 50),
           "max_depth": [None, 3, 5, 10],
           "min_samples_split": np.arange(2, 20, 2),
           "min_samples_leaf" : np.arange(1, 20, 2)}
```

Utilisons maintenant RandomizedSearchCV pour essayer d'améliorer notre modèle de régression logistique. Nous allons lui passer les différents hyperparamètres de log_reg_grid ainsi que le paramètre n_iter = 20. Cela signifie que RandomizedSearchCV essaiera 20 combinaisons différentes d'hyperparamètres de log_reg_grid et enregistrera les meilleures.

Entrée [34]:
```
#Améliorer le modèle par RandomizedSearchCV

#Differents hyperparametres de LogisticRegression
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                "solver": ["liblinear"]}

#Differents hyperparametres de RandomForrestClassifier
rf_grid = {"n_estimators" : np.arange(10, 1000, 50),
           "max_depth": [None, 3, 5, 10],
           "min_samples_split": np.arange(2, 20, 2),
           "min_samples_leaf" : np.arange(1, 20, 2)}
```

Entrée [35]:
```
#Définir random seed
np.random.seed(42)

#Definir une recherche à hyperparametre aléatoire pour Logistique Regression
rs_log_reg = RandomizedSearchCV(LogisticRegression(),
                                param_distributions=log_reg_grid,
                                cv=5,
                                n_iter=20,
                                verbose=True)

#Fit le modèle de recherche à hyperparametre aléatoire
rs_log_reg.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.3s finished
```

Out[35]:
```
RandomizedSearchCV(cv=5, estimator=LogisticRegression(), n_iter=20,
                   param_distributions={'C': array([1.00000000e-04, 2.63665090e-04, 6.95192796e-04, 1.83298071e-03,
       4.83293024e-03, 1.27427499e-02, 3.35981829e-02, 8.85866790e-02,
       2.33572147e-01, 6.15848211e-01, 1.62377674e+00, 4.28133240e+00,
       1.12883789e+01, 2.97635144e+01, 7.84759970e+01, 2.06913808e+02,
       5.45559478e+02, 1.43844989e+03, 3.79269019e+03, 1.00000000e+04]),
                                        'solver': ['liblinear']},
                   verbose=True)
```

```
Entrée [36]: rs_log_reg.best_params_

Out[36]: {'solver': 'liblinear', 'C': 0.23357214690901212}

Entrée [37]: rs_log_reg.score(X_test, y_test)

Out[37]: 0.8852459016393442
```

Maintenant que nous avons amélioré LogisticRegression en utilisant RandomizedSearchCV, nous allons faire de même pour RandomForestClassifier.

```
Entrée [38]: #Améliorer le modèle RandomForestClassifier
             np.random.seed(42)

             #Definir une recherche à hyperparametre aléatoire pour RandomForestClassifier
             rs_rf = RandomizedSearchCV(RandomForestClassifier(),
                                        param_distributions=rf_grid,
                                        cv=5,
                                        n_iter=20,
                                        verbose=True)

             #Fit le modèle de recherche à hyperparametre aléatoire
             rs_rf.fit(X_train, y_train)

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   52.6s finished

Out[38]: RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(), n_iter=20,
                            param_distributions={'max_depth': [None, 3, 5, 10],
                                                 'min_samples_leaf': array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19]),
                                                 'min_samples_split': array([ 2,  4,  6,  8, 10, 12, 14, 16, 18]),
                                                 'n_estimators': array([ 10,  60, 110, 160, 210, 260, 310, 360, 410, 460, 510,
         560, 610,
                660, 710, 760, 810, 860, 910, 960])},
                            verbose=True)

Entrée [39]: #Trouver les meiller hyperparametres
             rs_rf.best_params_

Out[39]: {'n_estimators': 210,
          'min_samples_split': 4,
          'min_samples_leaf': 19,
          'max_depth': 3}

Entrée [40]: #Evaluer le modèle de recherche aléatoire de RandomForest
             rs_rf.score(X_test, y_test)

Out[40]: 0.8688524590163934
```

Le réglage des hyperparamètres pour chaque modèle a permis d'améliorer légèrement les performances du RandomForestClassifier et du LogisticRegression. Mais comme LogisticRegression montre plus de performance, nous allons essayer de l'améliorer davantage avec GridSearchCV.

5.13.3. Le tuning du modèle en utilisant GridSearchCV :

La différence entre RandomizedSearchCV et GridSearchCV réside dans le fait que RandomizedSearchCV effectue des recherches sur une grille d'hyperparamètres en effectuant des combinaisons de n_itres, GridSearchCV testera toutes les combinaisons possibles. Plus précisemment, RandomizedSearchCV - essaie des combinaisons de n_iter d'hyperparamètres et enregistre la meilleure, alors que GridSearchCV - essaie chaque combinaison d'hyperparamètres et enregistre la meilleure.

```
Entrée [41]: #Améliorer le modèle avec GridSearchCV

             #Differents hyperparametres de LogisticRegression
             log_reg_grid = {"C": np.logspace(-4, 4, 20),
                             "solver": ["liblinear"]}

             #Definir une recherche à hyperparametre aléatoire pour Logistique Regression
             gs_log_reg = GridSearchCV(LogisticRegression(),
                                       param_grid=log_reg_grid,
                                       cv=5,
                                       verbose=True)

             #Fit le modèle de recherche à hyperparametre Grid
             gs_log_reg.fit(X_train, y_train)

          Fitting 5 folds for each of 20 candidates, totalling 100 fits

          [Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
          [Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.3s finished

Out[41]: GridSearchCV(cv=5, estimator=LogisticRegression(),
                       param_grid={'C': array([1.00000000e-04, 2.63665090e-04, 6.95192796e-04, 1.83298071e-03,
                4.83293024e-03, 1.27427499e-02, 3.35981829e-02, 8.85866790e-02,
                2.33572147e-01, 6.15848211e-01, 1.62377674e+00, 4.28133240e+00,
                1.12883789e+01, 2.97635144e+01, 7.84759970e+01, 2.06913808e+02,
                5.45559478e+02, 1.43844989e+03, 3.79269019e+03, 1.00000000e+04]),
                                   'solver': ['liblinear']},
                       verbose=True)

Entrée [42]: #Verifier le meiller hyperparametres
             gs_log_reg.best_params_

Out[42]: {'C': 0.23357214690901212, 'solver': 'liblinear'}

Entrée [43]: #Evaluer le modèle
             gs_log_reg.score(X_test, y_test)

Out[43]: 0.8852459016393442
```

Dans ce cas, nous obtenons les mêmes résultats qu'auparavant puisque notre grille ne comporte qu'un maximum de 20 combinaisons d'hyperparamètres différentes.

5.14. L'évaluation du modèle au-delà de la précision (accuracy) :

Maintenant que nous disposons d'un modèle optimisé, obtenons des mesures d'évaluation. Nous voulons :

- Courbe ROC et score AUC - plot_roc_curve()
- Matrice de confusion - confusion_matrix()
- Rapport de classification - classification_report()
- Précision - precision_score()
- Rappel - recall_score()
- F1-score - f1_score()

Pour y accéder, nous devrons utiliser notre modèle pour faire des prédictions sur l'ensemble des tests. Nous pouvons faire des prédictions en appelant predict() sur le modèle formé et en lui transmettant les données sur lesquelles nous souhaitons faire des prédictions.

```
Entrée [44]: #Evaluer le modèle de classification au delà de l'accuracy

             #Mener des prédictions sur la data set
             y_preds = gs_log_reg.predict(X_test)
```

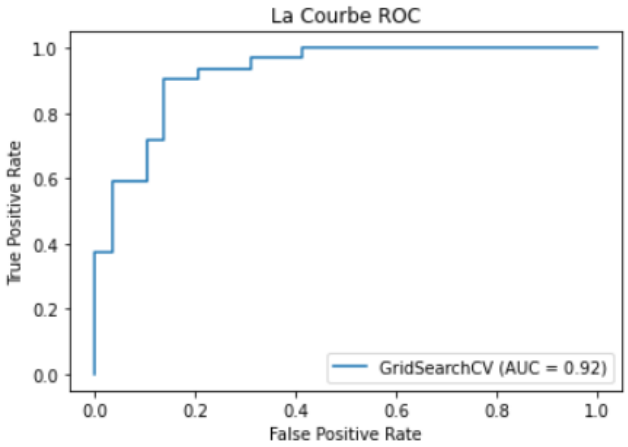
```
Entrée [45]: y_preds

Out[45]: array([0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0,
                0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0], dtype=int64)
```

```
Entrée [46]: y_test

Out[46]: array([0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0,
                0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0], dtype=int64)
```

y_preds ressemblent à nos données de test originales, sauf qu'ils sont différents dans les points où le modèle a fait des prédictions fausses.

5.14.1. La courbe ROC et le score AUC :

```
Entrée [47]: #Courbe ROC et Scores AUC

             #Importer la fonction de la courbe ROC à partir du module metrics
             from sklearn.metrics import plot_roc_curve

             #Afficher la courbe ROC et calculer l'AUC
             plot_roc_curve(gs_log_reg, X_test, y_test)
             plt.title("La Courbe ROC")

Out[47]: Text(0.5, 1.0, 'La Courbe ROC')
```



Notre modèle fait beaucoup mieux que de deviner quelle serait une ligne allant du coin inférieur gauche au coin supérieur droit, AUC = 0,5. Mais un modèle parfait atteindrait un score AUC de 1,0, donc il y a encore des possibilités d'amélioration.

Passons à la prochaine demande d'évaluation, la matrice de confusion.

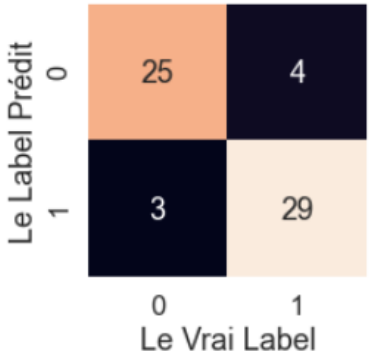
5.14.2. La matrice de confusion :

La matrice de confusion est un moyen visuel de montrer où notre modèle a fait les bonnes prédictions et où il a fait les mauvaises (ou, en d'autres termes, s'est embrouillé). Scikit-Learn nous permet de créer une matrice de confusion en utilisant confusion_matrix() et en lui passant les données originales et les données prédites.

Entrée [48]:
```
#Matrice de confusion

#Afficher la mtrice de confusion
print(confusion_matrix(y_test, y_preds))
```

```
[[25  4]
 [ 3 29]]
```

Entrée [49]:
```
#Importer Seaborn
import seaborn as sns
sns.set(font_scale=1.5) #Augmenter le font size

def plot_conf_mat(y_test, y_preds) :
    """
    Affiche une matrice de confusion en utilisant la seaborn heatmap().
    """
    fig, ax = plt.subplots(figsize=(3,3))
    ax = sns.heatmap(confusion_matrix(y_test, y_preds),
                     annot=True,
                     cbar=False)
    plt.xlabel("Le Vrai Label")
    plt.ylabel("Le Label Prédit")

plot_conf_mat(y_test, y_preds)
```



On peut voir que le modèle s'embrouille (prédit mal) de manière relativement identique dans les deux classes. En fait, il y a 4 cas où le modèle a prédit 0 alors qu'il aurait dû être 1 (faux négatif) et 3 cas où le modèle a prédit 1 au lieu de 0 (faux positif).

5.14.3. Classification report :

```
Entrée [50]: #Le rapport de Classification
             print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.89      0.86      0.88        29
           1       0.88      0.91      0.89        32

    accuracy                           0.89        61
   macro avg       0.89      0.88      0.88        61
weighted avg       0.89      0.89      0.89        61
```

Nous prendrons le meilleur modèle avec les meilleurs hyperparamètres et utiliserons cross_val_score() avec différentes valeurs de score. cross_val_score() fonctionne en prenant un estimateur (Machine Learning Model) avec des données originales et des données prédites. Il évalue ensuite le modèle Machine Learning sur les données originales et les données prédites en utilisant cross-validation et un paramètre de score défini. Rappelons-nous les meilleurs hyperparamètres ;

```
Entrée [51]: #Verifier les meilleurs hyperparametres
             gs_log_reg.best_params_
```

```
Out[51]: {'C': 0.23357214690901212, 'solver': 'liblinear'}
```

```
Entrée [52]: #Importer le cross_val_score
             from sklearn.model_selection import cross_val_score

             #Instantier le meilleur modèle par les meilleurs hyperparametres (Trouvés par GridSerchCV)
             clf = LogisticRegression (C = 0.23357214690901212,
                                       solver = 'liblinear')
```

```
Entrée [53]: #Le score d'accuracy cross-validé
             cv_acc = cross_val_score(clf,
                                      X,
                                      y,
                                      cv=5,
                                      scoring='accuracy')
             cv_acc
```

```
Out[53]: array([0.81967213, 0.90163934, 0.8852459 , 0.88333333, 0.75      ])
```

Comme il y a 5 mesures, nous prendrons la moyenne ;

```
Entrée [54]: cv_acc = np.mean(cv_acc)
             cv_acc
```

```
Out[54]: 0.8479781420765027
```

Nous allons maintenant faire de même pour d'autres mesures de classification.

Entrée [55]:
```
#Le score de précision cross-validé
cv_precision = np.mean(cross_val_score(clf,
                                       X,
                                       y,
                                       cv = 5,
                                       scoring = 'precision'))
cv_precision
```

Out[55]: 0.8215873015873015

Entrée [56]:
```
#Le score du Recall cross-validé
cv_recall = np.mean(cross_val_score(clf,
                                    X,
                                    y,
                                    cv=5,
                                    scoring = 'recall'))
cv_recall
```

Out[56]: 0.9272727272727274

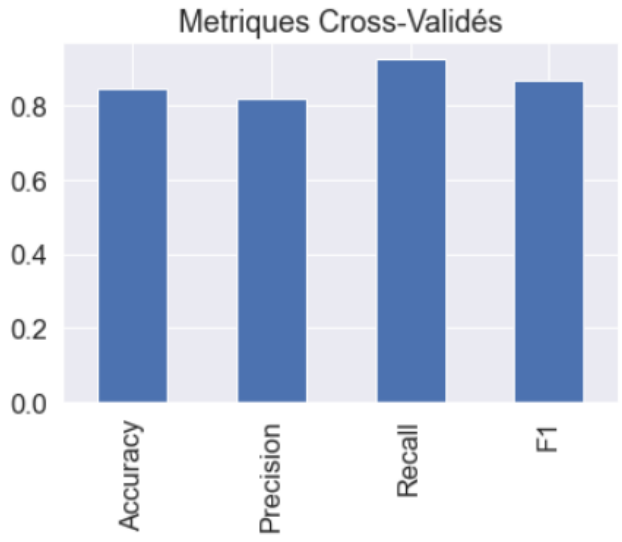
Entrée [57]:
```
#Le score F1 cross-validé
cv_f1 = np.mean(cross_val_score(clf,
                                X,
                                y,
                                cv=5,
                                scoring="f1"))
cv_f1
```

Out[57]: 0.8705403543192143

Visualisons ces mesures ;

Entrée [58]:
```
#Visuliser ces metriques
cv_metrics = pd.DataFrame({"Accuracy" : cv_acc,
                           "Precision" : cv_precision,
                           "Recall" : cv_recall,
                           "F1" : cv_f1},
                          index=[0])
cv_metrics.T.plot.bar(title="Metriques Cross-Validés",
                      legend=False)
```

Out[58]: <AxesSubplot:title={'center':'Metriques Cross-Validés'}>



5.15. L'importance des features :

Puisque nous utilisons LogisticRegression, nous utiliserons l'attribut coef_. Dans la documentation de Scikit-Learn pour LogisticRegression, l'attribut coef_ est le coefficient des features dans la fonction de décision. Nous pouvons accéder à l'attribut coef_ après avoir ajusté une instance de LogisticRegression.

```
Entrée [59]: #Ajuster une instance de LogisticRegression
             clf.fit(X_train, y_train)

 Out[59]: LogisticRegression(C=0.23357214690901212, solver='liblinear')
```

```
Entrée [60]: #Verifier coef_
             clf.coef_

 Out[60]: array([[ 0.00369922, -0.90424087,  0.67472828, -0.0116134 , -0.00170364,
                   0.04787689,  0.33490186,  0.02472938, -0.63120403, -0.57590919,
                   0.4709512 , -0.6516535 , -0.69984202]])
```

Ces valeurs indiquent combien chaque feature contribue à la façon dont un modèle décide si les tendances d'un échantillon de données sur la santé des patients sont plus favorables ou non aux maladies cardiaques.

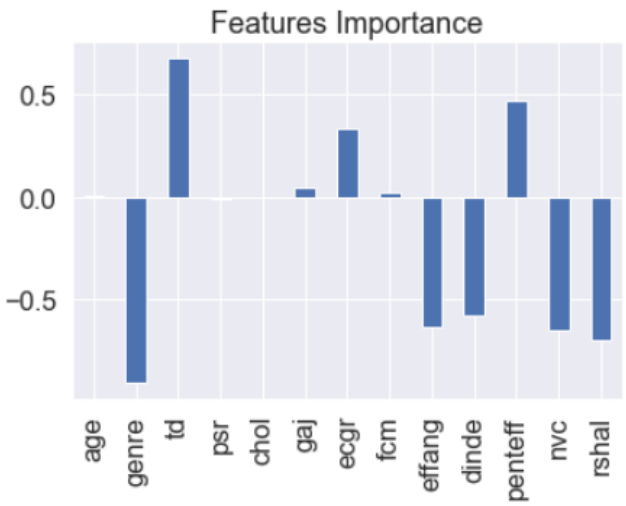
Même en sachant cela, ce coef_ array ne signifie toujours pas grand-chose. Mais il le sera si nous le combinons avec les colonnes de notre base de données.

```
Entrée [61]: #Matcher les features aux colonnes
             features_dict = dict(zip(cardio.columns, list(clf.coef_[0])))
             features_dict

 Out[61]: {'age': 0.003699219011760782,
           'genre': -0.9042408714480176,
           'td': 0.6747282766540338,
           'psr': -0.01161340334265323,
           'chol': -0.0017036446360052262,
           'gaj': 0.04787688669240361,
           'ecgr': 0.3349018562558094,
           'fcm': 0.024729384743360134,
           'effang': -0.631204028843173,
           'dinde': -0.575909185434028,
           'penteff': 0.47095119664446533,
           'nvc': -0.6516535002884537,
           'rshal': -0.6998420233395882}
```

Maintenant que nous avons fait correspondre les coefficients des features aux différentes colonnes, visualisons-les.

```
Entrée [62]: #Visualiser l'importance des features
             features_cardio = pd.DataFrame(features_dict, index=[0])
             features_cardio.T.plot.bar(title="Features Importance", legend = False)
```

```
Out[62]: <AxesSubplot:title={'center':'Features Importance'}>
```



Par exemple, l'attribut genre a une valeur négative de -0,904 (négative corrélation), ce qui signifie que lorsque la valeur du genre augmente (c'est-à-dire en allant de 0 (femme) à 1 (homme), la valeur target diminue (c'est-à-dire en allant de 1 (malade) à 0 (non malade).

6. Conclusion :

Pour conclure, à ce stade, après avoir essayé différentes mesures, nous nous demandons si nous avons atteint la métrique d'évaluation : « si nous pouvons atteindre une précision de 95 % pour prédire si un patient souffre ou non d'une maladie cardiaque pendant la phase de validation du concept, nous assurerons ce projet ».

Dans le cas présent, nous n'avons pas arriver au niveau requis. La plus grande précision que notre modèle a atteint était inférieure à 90%.