جامعة سيدي محمد بن عبد الله بفاس
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵙⵉⴷⵉ ⵎⵓⵃⵎⴷ ⴱⵏ ⵄⴱⴷⵍⵍⴰⵀ ⴱ ⴼⴰⵙ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH DE FES

كلية العلوم والتقنيات فاس
ⵜⴰⵖⵡⵍⵓⵜ ⵏ ⵜⵙⵙⵏⵜⵉⵡⵉⵏ ⴷ ⵜⵙⵙⵏⵜⵉⵡⵉⵏ - ⴼⴰⵙ
FACULTÉ DES SCIENCES ET TECHNIQUES DE FÈS

Prédiction des maladies coronariennes

Master SDSI

Réalisé par :

- MOUFID ANAS
- OUAZZANI CHAHDI HAMZA

Encadré par :

- Pr. S. NAJAH

Année Universitaire 2023-2024

Exploratory_Data_Analysis

April 13, 2024

1 Idée générale

L'Organisation mondiale de la santé estime que les maladies cardiaques sont responsables de 12 millions de décès chaque année dans le monde. Aux États-Unis et dans d'autres pays développés, la moitié des décès sont attribués aux maladies cardiovasculaires. Cependant, détecter les maladies cardiaques précocement permet de prendre des mesures pour améliorer le mode de vie des patients et réduire les complications associées. Dans ce contexte, cette étude vise à identifier les facteurs de risque les plus pertinents des maladies cardiaques et à prédire le risque global à l'aide de la régression logistique et de la préparation des données avec l'outil Spark. Nous utiliserons spécifiquement Pyspark, que nous présenterons dans le paragraphe suivant, pour analyser et manipuler les données de manière efficace. L'objectif de cette étude est de prédire si un patient présente un risque de maladie coronarienne (CHD) dans les 10 prochaines années en fonction de différents facteurs de risque démographiques, comportementaux et médicaux. Ce rapport présentera en détail la méthodologie utilisée pour atteindre cet objectif ainsi que les résultats obtenus et leur interprétation.

1.1 Pourquoi Pyspark ?

PySpark est une bibliothèque de traitement de données distribuée qui est conçue pour traiter de grandes quantités de données. En effet, le choix de PySpark pour le prétraitement des données,

l'apprentissage du modèle et la prédiction dans notre mini-projet de prédiction de maladies coronariennes offre plusieurs avantages :

- **Traitement distribué et parallélisme** : PySpark utilise le parallélisme pour accélérer les opérations de traitement des données, ce qui permet d'effectuer des calculs simultanément sur plusieurs nœuds.
- **Évolutivité** : PySpark est très évolutif et peut gérer des volumes de données de plus en plus importants sans compromettre les performances.
- **Flexibilité** : PySpark prend en charge une grande variété de sources de données, telles que les fichiers CSV, les bases de données, les fichiers JSON, etc. Ce qui le rend très flexible pour traiter différents types de données.
- **Intégration avec les outils Big Data** : PySpark est conçu pour fonctionner avec les outils de Big Data tels que Hadoop, HBase, Cassandra, etc. Il peut également être utilisé avec d'autres outils de traitement de données tels que Pandas et Scikit-learn.
- **Prise en charge du traitement en temps réel** : PySpark prend également en charge le traitement en temps réel des données à l'aide de la bibliothèque Spark Streaming, ce qui permet de traiter les données en temps réel, ce qui est essentiel pour les applications telles que l'analyse de données en temps réel et la surveillance de l'état du système.

2 I. Analyse et exploration des données:

2.1 1. Importation des bibliothèques et du jeu de données

[64]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession
```

[65]:
```
# Initialisation de la session Spark // Port : 4040
# Une fois la seesion crée, on peut interagir avec les données stockées dans un␣
 ↪environnement distribué.
spark = SparkSession.builder.appName('HeartDiseasesPredicion').config("spark.ui.
 ↪port", "4041").getOrCreate()
```

[66]:
```
# Importing data
path = 'cardiovascular_risk.csv'
# Chargement du jeu de données
data = spark.read.csv('dataset/train.csv', header=True, inferSchema=True)

df = pd.read_csv(path,index_col='id')
```

[4]:
```
df[df['TenYearCHD'] == 1].count()
```

[4]:
```
age               511
education         498
sex               511
```

```
is_smoking          511
cigsPerDay          510
BPMeds              504
prevalentStroke     511
prevalentHyp        511
diabetes            511
totChol             504
sysBP               511
diaBP               511
BMI                 504
heartRate           510
glucose             472
TenYearCHD          511
dtype: int64
```

2.2 2. Compréhension des données

[ ]:

[67]:
```
# First 10 rows of the dataset
df.head()
```

[67]:

id	age	education	sex	is_smoking	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearCHD
0	64	2.0	F	YES	3.0	0.0	0	0	0	221.0	148.0	85.0	NaN	90.0	80.0	1
1	36	4.0	M	NO	0.0	0.0	0	1	0	212.0	168.0	98.0	29.77	72.0	75.0	0
2	46	1.0	F	YES	10.0	0.0	0	0	0	250.0	116.0	71.0	20.35	88.0	94.0	0
3	50	1.0	M	YES	20.0	0.0	0	1	0	233.0	158.0	88.0	28.26	68.0	94.0	1
4	64	1.0	F	YES	30.0	0.0	0	0	0	241.0	136.5	85.0	26.42	70.0	77.0	0

[68]:
```
# Dataset Rows & Columns
df.shape
```

[68]:
```
(3390, 16)
```

[69]:
```
# Dataset Rows & Columns
df.shape
# Dataset Info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3390 entries, 0 to 3389
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   age              3390 non-null   int64
 1   education        3303 non-null   float64
 2   sex              3390 non-null   object
 3   is_smoking       3390 non-null   object
 4   cigsPerDay       3368 non-null   float64
 5   BPMeds           3346 non-null   float64
 6   prevalentStroke  3390 non-null   int64
 7   prevalentHyp     3390 non-null   int64
 8   diabetes         3390 non-null   int64
 9   totChol          3352 non-null   float64
 10  sysBP            3390 non-null   float64
 11  diaBP            3390 non-null   float64
 12  BMI              3376 non-null   float64
 13  heartRate        3389 non-null   float64
 14  glucose          3086 non-null   float64
 15  TenYearCHD       3390 non-null   int64
dtypes: float64(9), int64(5), object(2)
memory usage: 450.2+ KB
```

2.2.1 2.1. Description des variables :

[70]:
```
# Dataset Columns
df.columns
```

[70]:
```
Index(['age', 'education', 'sex', 'is_smoking', 'cigsPerDay', 'BPMeds',
       'prevalentStroke', 'prevalentHyp', 'diabetes', 'totChol', 'sysBP',
       'diaBP', 'BMI', 'heartRate', 'glucose', 'TenYearCHD'],
      dtype='object')
```

Demographic: * Sex: Homme or femme (“M” or “F”) * Age: Age du patient (Continu - Bien que les âges enregistrés aient été tronqués à des nombres entiers, le concept d’âge est continu) * Education: Le niveau d’éducation du patient (valeurs catégorielles - 1, 2, 3, 4)

Behavioral: * is_smoking: Si le patient est un fumeur actuel ou non(“YES” or “NO”) * Cigs Per Day: Le nombre de cigarettes fumées en moyenne par jour par la personne. (Peut être considéré comme continu car on peut avoir n’importe quel nombre de cigarettes, même une demi-cigarette.)

Medical (history): * BP Meds: Si le patient prenait des médicaments contre l’hypertension artérielle (Nominal) * Prevalent Stroke: Si le patient avait déjà eu un AVC(Nominal) * Prevalent Hyp: Si le patient était hypertendu (Nominal) * Diabetes: Si le patient était diabétique (Nominal)

Medical (current): * Tot Chol: niveau de cholestérol total (Continu) * Sys BP: pression artérielle systolique (Continu) * Dia BP: pression artérielle diastolique (Continu) * BMI: Indice de masse corporelle (Continu) * Heart Rate: fréquence cardiaque (Continu - En recherche médicale, des variables telles que la fréquence cardiaque, bien qu’en réalité discrètes, sont considérées comme continues en raison du grand nombre de valeurs possibles.) * Glucose: niveau de glucose (Continu)

variable à predire (desired target): * 10 ans de risque de maladie coronarienne CHD (binaire: «1», signifie «Yes», «0» signifie «No»).

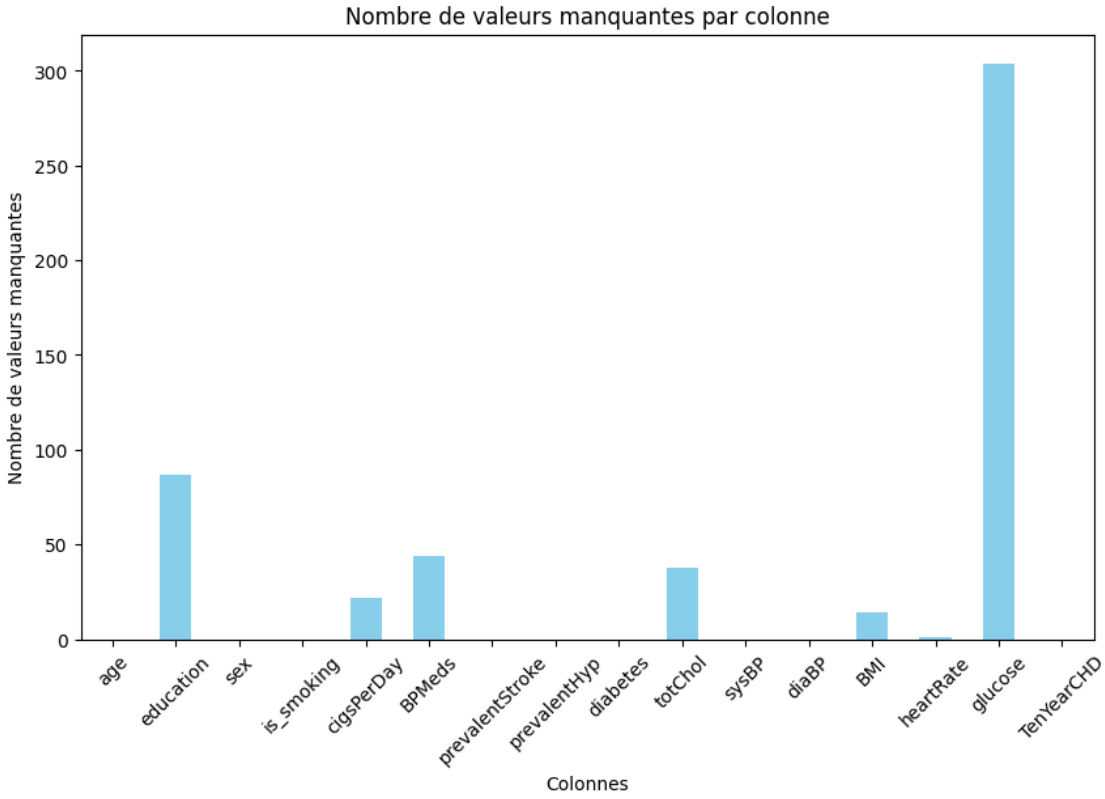
[71]:
```
# checking duplicates
len(df[df.duplicated()])
```

[71]:
```
0
```

[72]:
```
# checking null values
df.isna().sum().sum()
```

[72]:
```
510
```

[73]:
```
# Compter le nombre de valeurs manquantes dans chaque colonne
missing_values = df.isnull().sum()

# Créer le diagramme en bâtons
plt.figure(figsize=(10, 6))
missing_values.plot(kind='bar', color='skyblue')
plt.title('Nombre de valeurs manquantes par colonne')
plt.xlabel('Colonnes')
plt.ylabel('Nombre de valeurs manquantes')
plt.xticks(rotation=45)
plt.show()
```



- Le jeu de données provient d'une étude cardiovasculaire en cours sur les résidents de la ville de Framingham, Massachusetts. L'objectif de classification est de prédire si le patient présente un risque de maladie coronarienne (CHD) dans les 10 prochaines années.
- **Le jeu de données contient 3390 lignes et 16 colonnes. Il n'y a pas d'enregistrements en double et contient 510 valeurs manquantes.**

[74]:
```
# Dataset Describe
df.describe(include='all').T
```

[74]:

	count	unique	top	freq	mean	std	min	\
age	3390.0	NaN	NaN	NaN	49.542183	8.592878	32.0	
education	3303.0	NaN	NaN	NaN	1.970936	1.019081	1.0	
sex	3390	2	F	1923	NaN	NaN	NaN	
is_smoking	3390	2	NO	1703	NaN	NaN	NaN	
cigsPerDay	3368.0	NaN	NaN	NaN	9.069477	11.879078	0.0	
BPMeds	3346.0	NaN	NaN	NaN	0.029886	0.170299	0.0	
prevalentStroke	3390.0	NaN	NaN	NaN	0.00649	0.080309	0.0	
prevalentHyp	3390.0	NaN	NaN	NaN	0.315339	0.464719	0.0	
diabetes	3390.0	NaN	NaN	NaN	0.025664	0.158153	0.0	
totChol	3352.0	NaN	NaN	NaN	237.074284	45.24743	107.0	
sysBP	3390.0	NaN	NaN	NaN	132.60118	22.29203	83.5	
diaBP	3390.0	NaN	NaN	NaN	82.883038	12.023581	48.0	

```
BMI              3376.0    NaN  NaN   NaN   25.794964   4.115449  15.96
heartRate        3389.0    NaN  NaN   NaN   75.977279  11.971868   45.0
glucose          3086.0    NaN  NaN   NaN    82.08652  24.244753   40.0
TenYearCHD       3390.0    NaN  NaN   NaN    0.150737   0.357846    0.0

                    25%    50%    75%    max
age                42.0   49.0   56.0   70.0
education           1.0    2.0    3.0    4.0
sex                 NaN    NaN    NaN    NaN
is_smoking          NaN    NaN    NaN    NaN
cigsPerDay          0.0    0.0   20.0   70.0
BPMeds              0.0    0.0    0.0    1.0
prevalentStroke     0.0    0.0    0.0    1.0
prevalentHyp        0.0    0.0    1.0    1.0
diabetes            0.0    0.0    0.0    1.0
totChol           206.0  234.0  264.0  696.0
sysBP             117.0  128.5  144.0  295.0
diaBP              74.5   82.0   90.0  142.5
BMI               23.02  25.38  28.04   56.8
heartRate          68.0   75.0   83.0  143.0
glucose            71.0   78.0   87.0  394.0
TenYearCHD          0.0    0.0    0.0    1.0
```

[75]:
```
# Renaming the columns
df.rename(columns={'cigsPerDay':'cigs_per_day','BPMeds':'bp_meds',
                   'prevalentStroke':'prevalent_stroke','prevalentHyp':
 ↪'prevalent_hyp',
                   'totChol':'total_cholesterol','sysBP':'systolic_bp','diaBP':
 ↪'diastolic_bp',
                   'BMI':'bmi','heartRate':'heart_rate','TenYearCHD':
 ↪'ten_year_chd'},
          inplace = True)
```

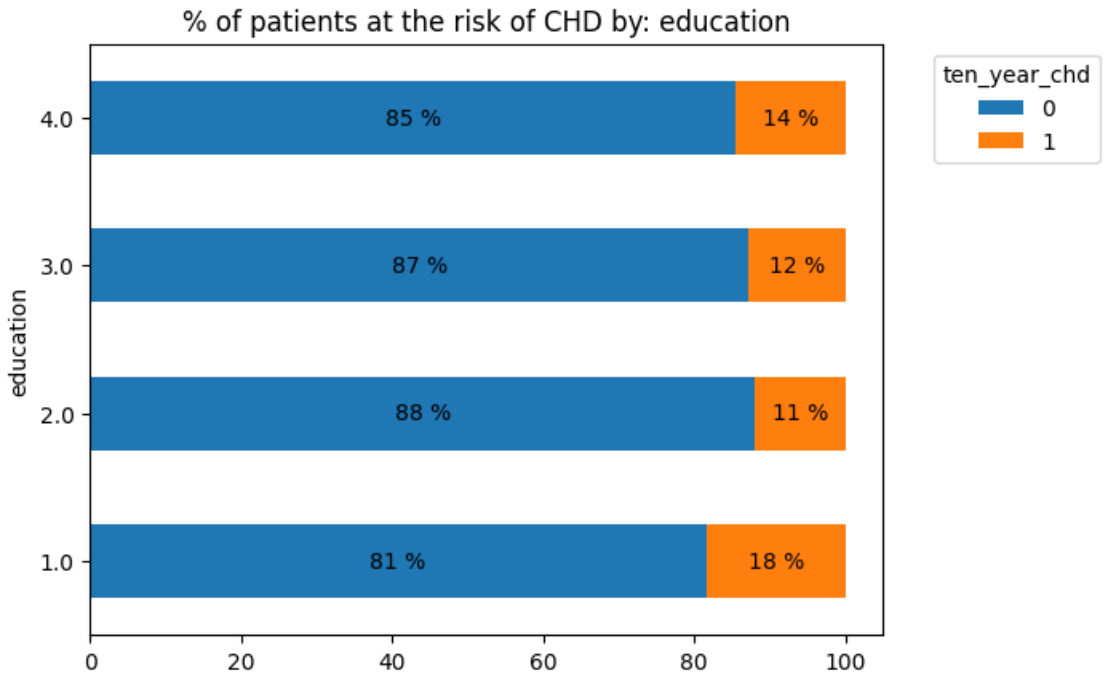
2.2.2 2.2. Analyse du distribution des données

[76]:
```
dependent_var = ['ten_year_chd']
continuous_var = ['age','cigs_per_day','total_cholesterol','systolic_bp',␣
 ↪'diastolic_bp', 'bmi', 'heart_rate', 'glucose']
categorical_var = ['education', 'sex',␣
 ↪'is_smoking','bp_meds','prevalent_stroke', 'prevalent_hyp', 'diabetes']
```

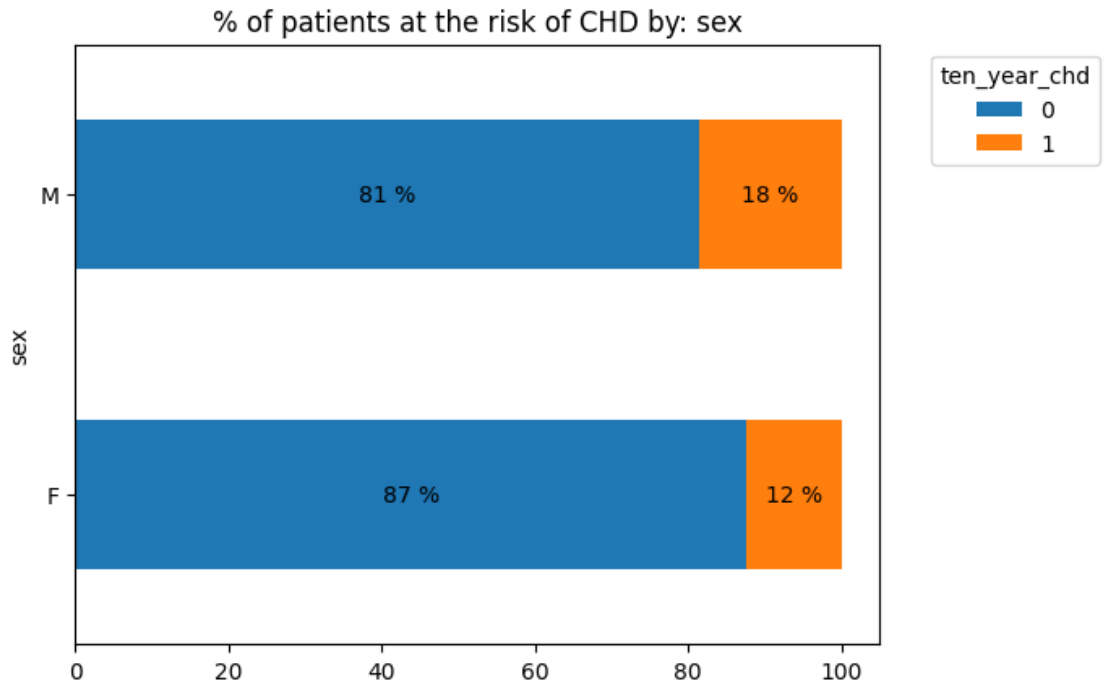
[77]:
```
# 100% stacked bar chart

for i in categorical_var:
    x_var, y_var = i, dependent_var[0]
    plt.figure(figsize=(10,5))
```

```
    df_grouped = df.groupby(x_var)[y_var].value_counts(normalize=True).
↪unstack(y_var)*100
    df_grouped.plot.barh(stacked=True)
    plt.legend(
        bbox_to_anchor=(1.05, 1),
        loc="upper left",
        title=y_var)

    plt.title("% of patients at the risk of CHD by: "+i)
    for ix, row in df_grouped.reset_index(drop=True).iterrows():
        # print(ix, row)
        cumulative = 0
        for element in row:
            if element > 0.1:
                plt.text(
                    cumulative + element / 2,
                    ix,
                    f"{int(element)} %",
                    va="center",
                    ha="center",
                )
            cumulative += element
    plt.show()
```

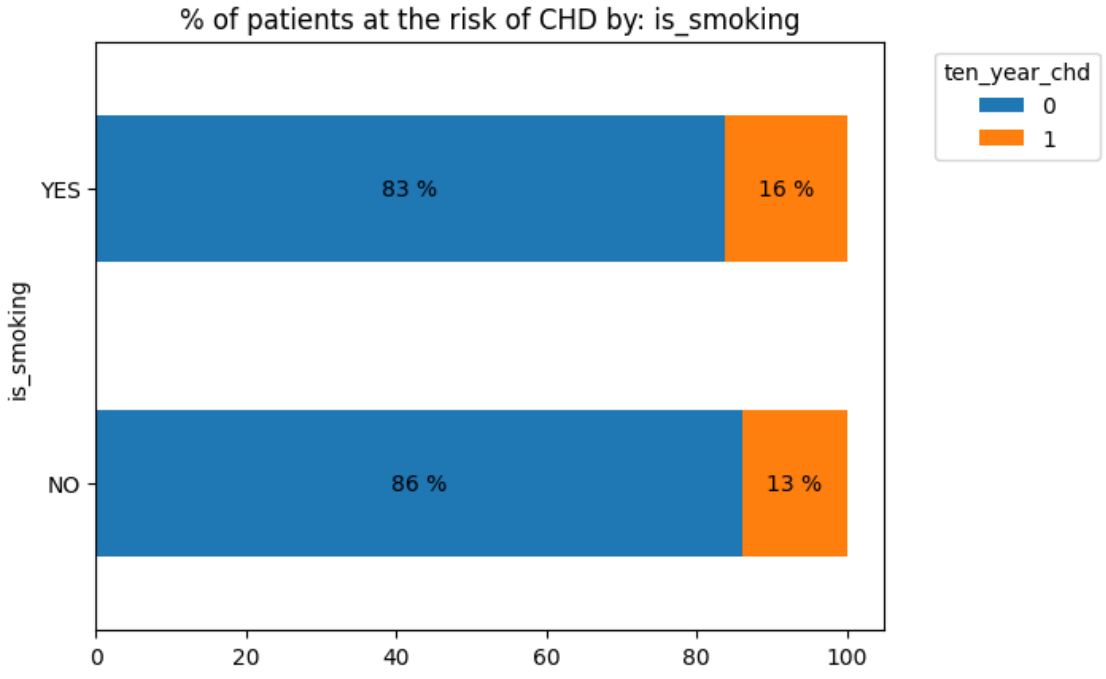
```
<Figure size 1000x500 with 0 Axes>
```



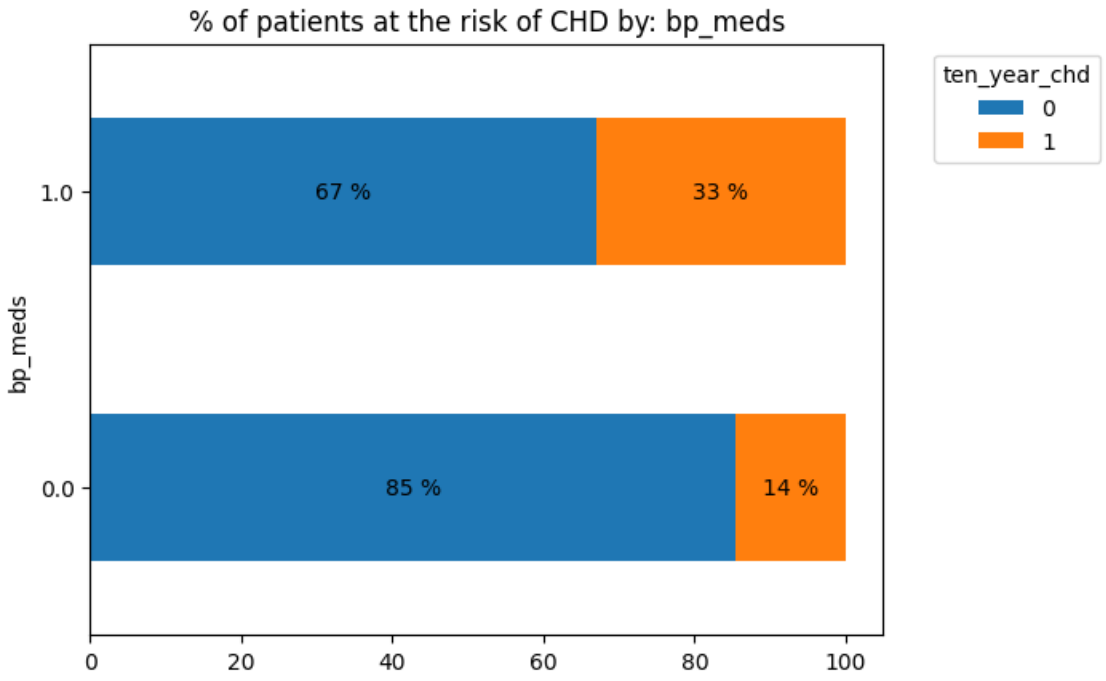
```
<Figure size 1000x500 with 0 Axes>
```



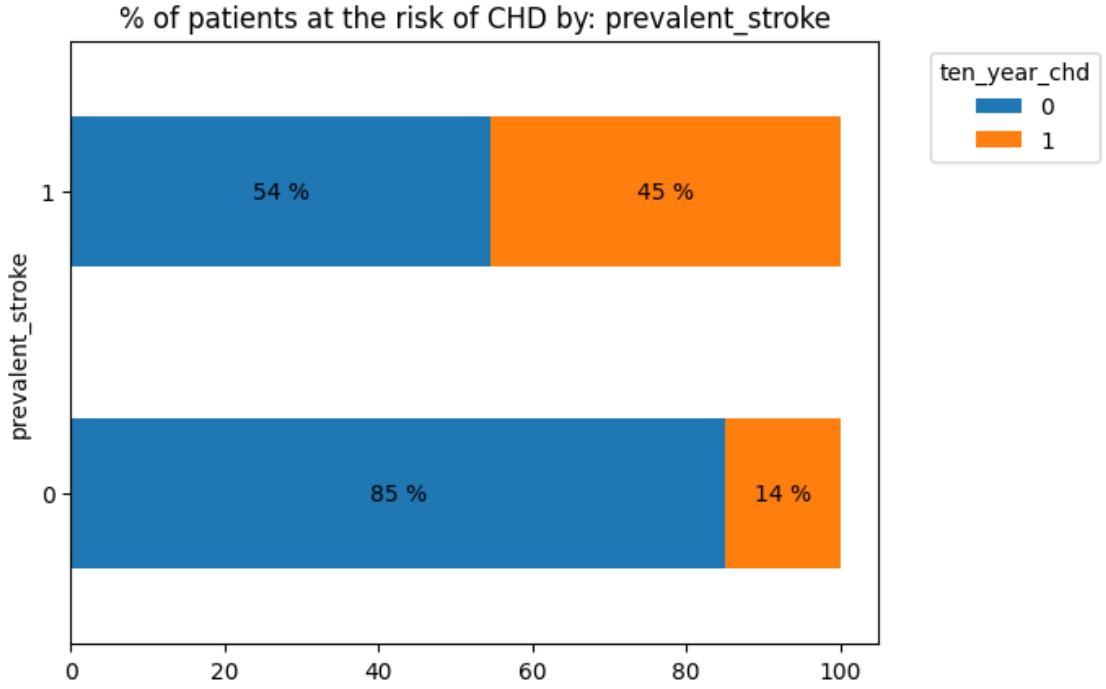

```
<Figure size 1000x500 with 0 Axes>
```



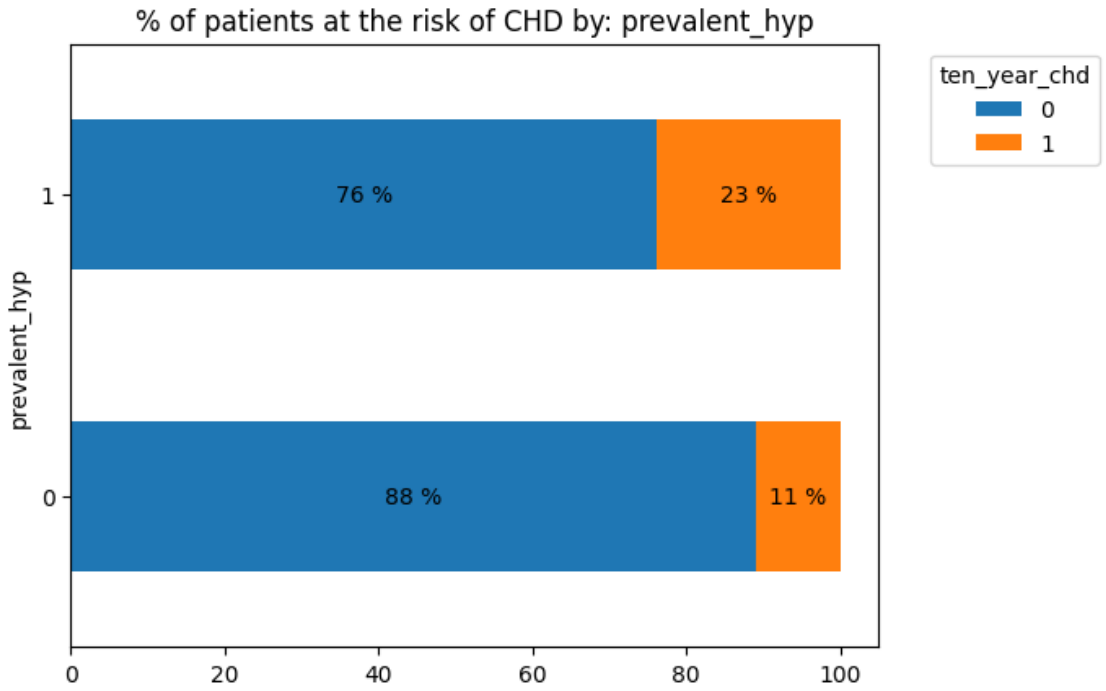
```
<Figure size 1000x500 with 0 Axes>
```



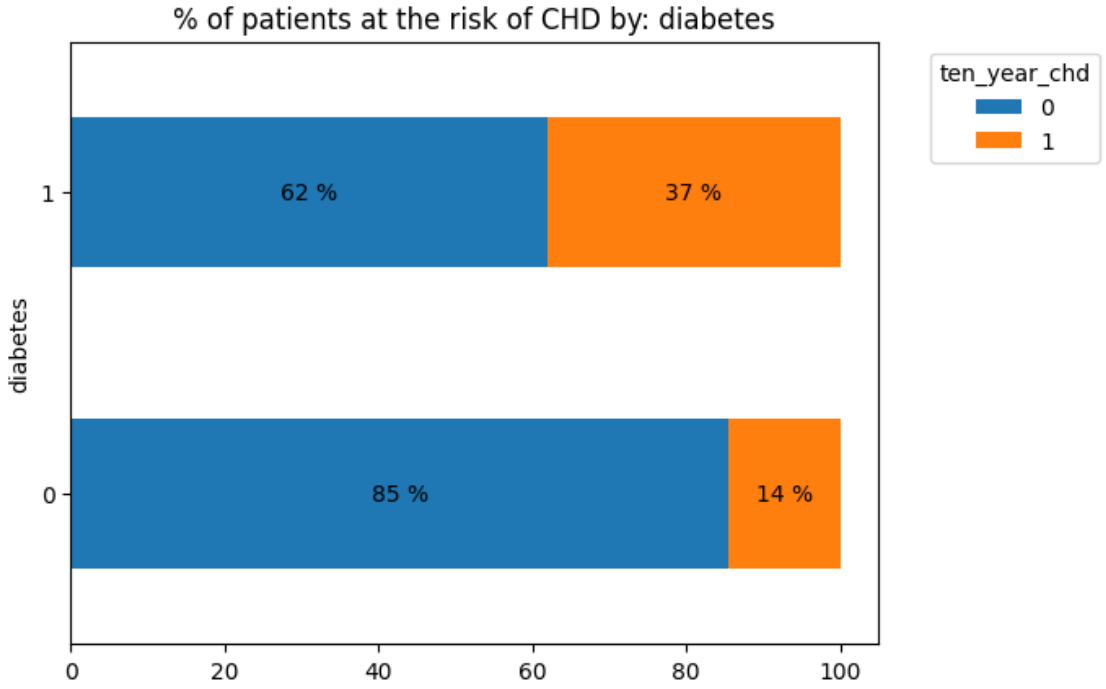

```
<Figure size 1000x500 with 0 Axes>
```



```
<Figure size 1000x500 with 0 Axes>
```




```
<Figure size 1000x500 with 0 Axes>
```



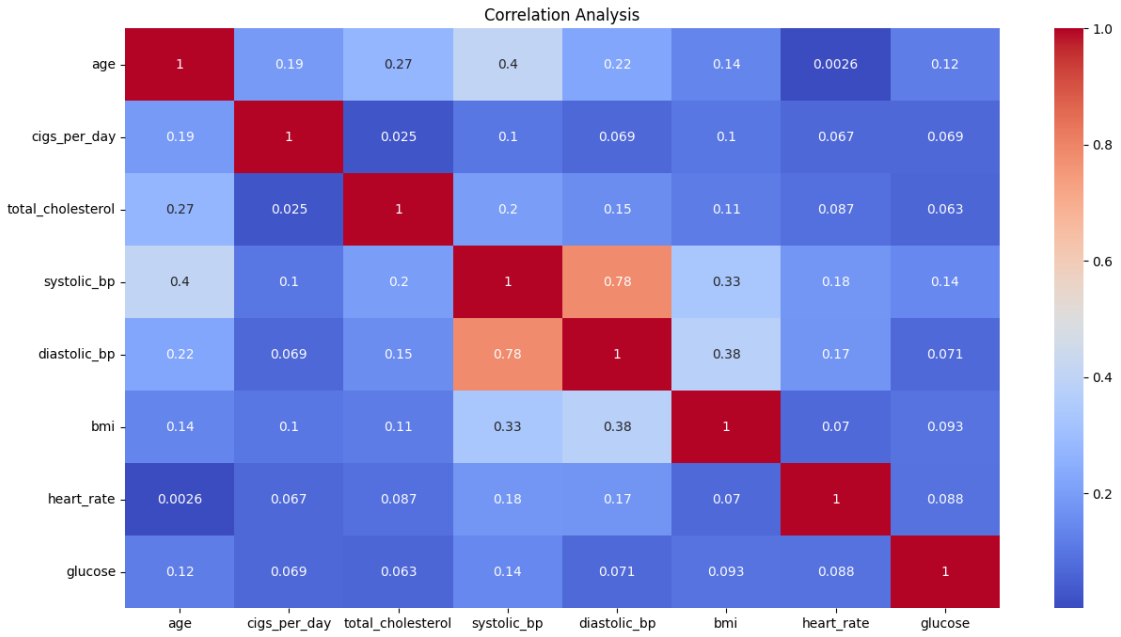
Résumé: * 18%, 11%, 12%, 14% des patients appartenant respectivement aux niveaux d'éducation 1, 2, 3, 4 ont finalement été diagnostiqués avec une MCV (maladie cardiovasculaire). * Les patients masculins présentent un risque significativement plus élevé de MCV (18%) que les patients féminins (12%).** * Les patients qui fument ont un risque significativement plus élevé de MCV (16%) que les patients non fumant (13%). * Les patients prenant des médicaments pour la tension artérielle ont un risque significativement plus élevé de MCV (33%) que les autres patients (14%). * Les patients ayant déjà eu un AVC ont un risque significativement plus élevé de MCV (45%) que les autres patients (14%). * Les patients hypertendus ont un risque significativement plus élevé de MCV (23%) que les autres patients (11%). * Les patients diabétiques ont un risque significativement plus élevé de MCV (37%) que les autres patients (14%).

[78]:
```
# Encoding the binary columns

df['sex'] = np.where(df['sex'] == 'M',1,0)
df['is_smoking'] = np.where(df['is_smoking'] == 'YES',1,0)
```

2.2.3 2.3. Analyse de corrélation:

[79]:
```
# Correlation magnitude
plt.figure(figsize=(15,8))
plt.title('Correlation Analysis')
correlation = df[continuous_var].corr()
sns.heatmap(abs(correlation), annot=True, cmap='coolwarm')
```

[79]:
```
<Axes: title={'center': 'Correlation Analysis'}>
```



- **Les variables tension artérielle systolique(systolic_bp) et diastolique(diastolic_bp) sont fortement corrélées.**

2.2.4 2.4. Handling Multicollinearity:

[80]:
```
# Range of systolic bp and diastolic bp

print(df['systolic_bp'].min(),df['systolic_bp'].max())
print(df['diastolic_bp'].min(),df['diastolic_bp'].max())
```

```
83.5 295.0
48.0 142.5
```

Pour gérer la multicollinéarité entre ces deux variables continues indépendantes, nous pouvons remplacer ces deux colonnes par une nouvelle variable 'pression pulsée', définie comme suit :

Pression Pulsée = Pression Artérielle Systolique - Pression Artérielle Diastolique

Reference

[81]:
```
# Creating a new column pulse_pressure
# and dropping systolic_bp and diastolic_bp

df['pulse_pressure'] = df['systolic_bp']-df['diastolic_bp']
df.drop('systolic_bp',axis=1,inplace=True)
df.drop('diastolic_bp',axis=1,inplace=True)
```

[82]:
```
# columns
df.columns
```

[82]:
```
Index(['age', 'education', 'sex', 'is_smoking', 'cigs_per_day', 'bp_meds',
       'prevalent_stroke', 'prevalent_hyp', 'diabetes', 'total_cholesterol',
       'bmi', 'heart_rate', 'glucose', 'ten_year_chd', 'pulse_pressure'],
      dtype='object')
```

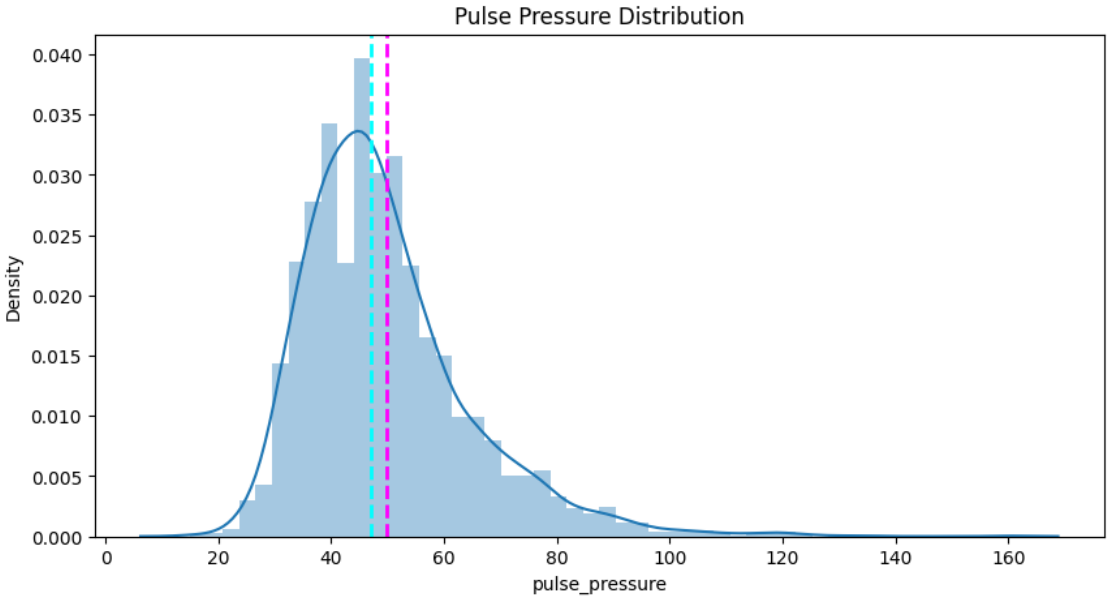
[83]:
```
# Updating the continuous_var list

continuous_var.remove('systolic_bp')
continuous_var.remove('diastolic_bp')
continuous_var.append('pulse_pressure')
```

[84]:
```
# Analyzing the distribution of pulse_pressure
plt.figure(figsize=(10,5))
sns.distplot(df['pulse_pressure'])
plt.axvline(df['pulse_pressure'].mean(), color='magenta', linestyle='dashed',␣
 ↪linewidth=2)
plt.axvline(df['pulse_pressure'].median(), color='cyan', linestyle='dashed',␣
 ↪linewidth=2)
```

```
plt.title('Pulse Pressure Distribution')
```

```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\3383542107.py:3: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df['pulse_pressure'])
```

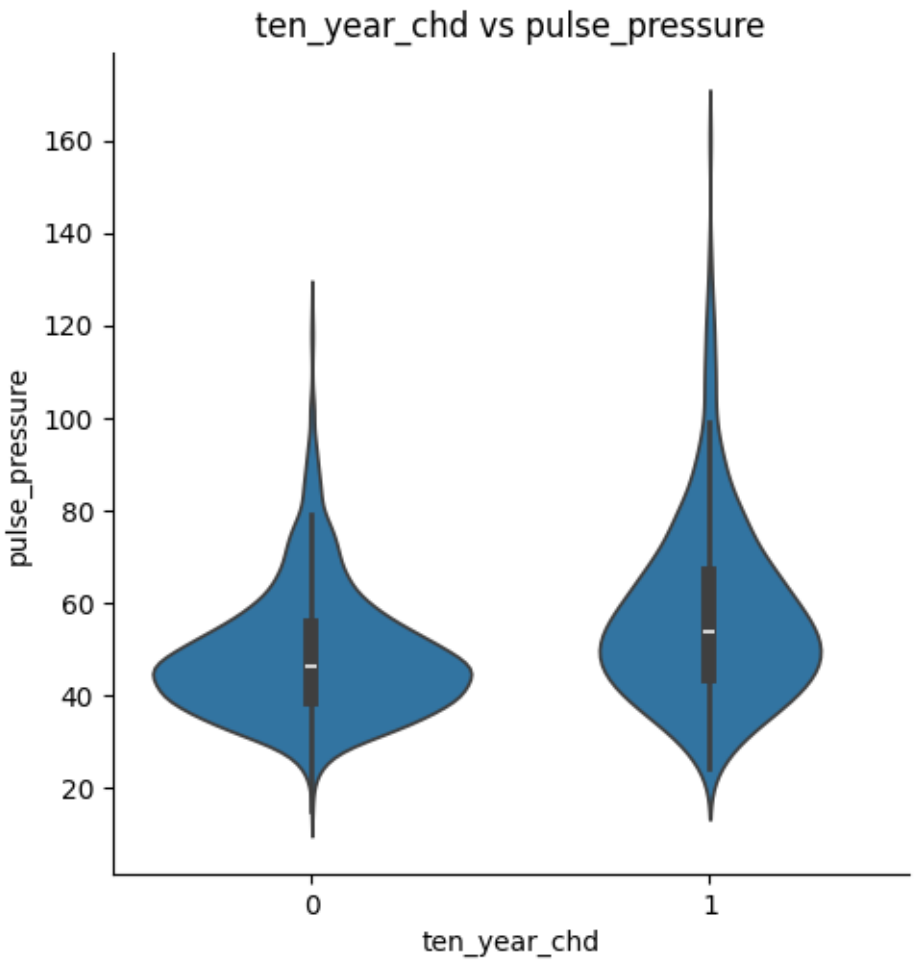
[84]: `Text(0.5, 1.0, 'Pulse Pressure Distribution')`



Les pressions pulsées présentent une asymétrie positive.

[85]:
```
# Relationship between pulse pressure with the dependent variable
plt.figure(figsize=(10,5))
sns.catplot(x=dependent_var[0],y='pulse_pressure',data=df,kind='violin')
plt.title('ten_year_chd vs pulse_pressure')
plt.show()
```

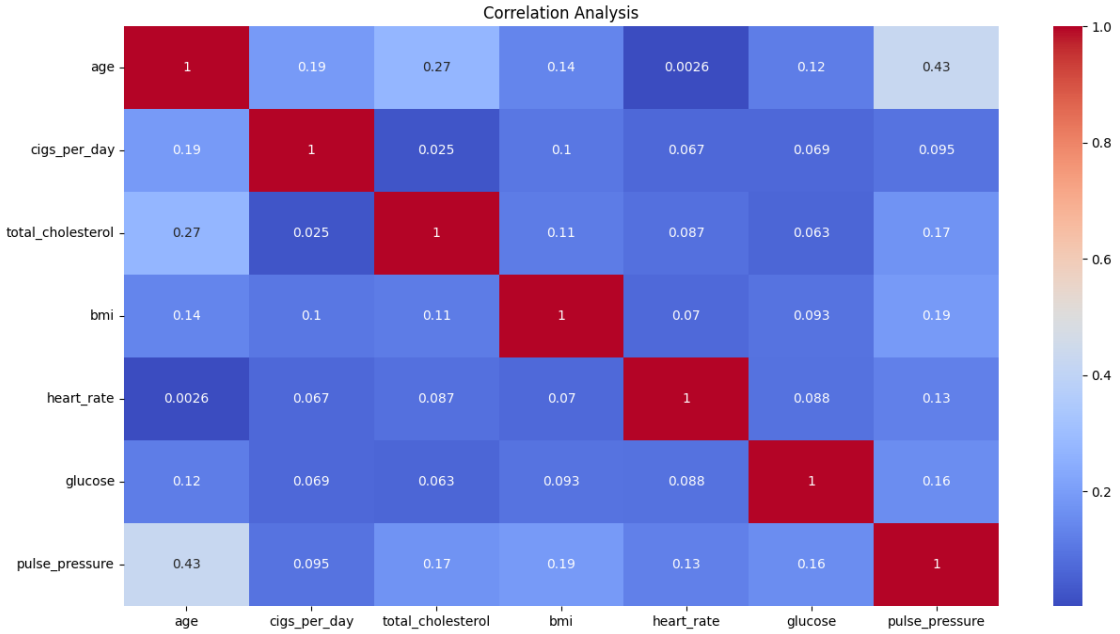
```
<Figure size 1000x500 with 0 Axes>
```



En moyenne, les patients ayant une pression pulsée plus élevée sont exposés à la maladie coronarienne sur une période de 10 ans.

[86]:
```
# Updated correlations
plt.figure(figsize=(15,8))
plt.title('Correlation Analysis')
correlation = df[continuous_var].corr()
sns.heatmap(abs(correlation), annot=True, cmap='coolwarm')
```

[86]: <Axes: title={'center': 'Correlation Analysis'}>



Nous avons réussi à gérer la multicollinéarité parmi les variables continues dans l'ensemble de données.

3 II. Préparation des données:

[87]:
```
# checking for null values
df.isna().sum()
```

[87]:
```
age                 0
education          87
sex                 0
is_smoking          0
cigs_per_day       22
bp_meds            44
prevalent_stroke    0
prevalent_hyp       0
diabetes            0
total_cholesterol  38
bmi                14
heart_rate          1
glucose           304
ten_year_chd        0
pulse_pressure      0
dtype: int64
```

[88]:
```
# total null values
df.isna().sum().sum()
```

[88]:
```
510
```

Il y a un total de 510 valeurs manquantes dans le jeu de données.

3.1 1. Remplacement des valeurs manquantes dans les colonnes catégorielles par l'entrée la plus fréquente "Le mode" :

[89]:
```
# education distribution before mode imputation
df.education.value_counts()
```

[89]:
```
education
1.0    1391
2.0     990
3.0     549
4.0     373
Name: count, dtype: int64
```

[90]:
```
# bp_meds distribution before mode imputation
df.bp_meds.value_counts()
```

[90]:
```
bp_meds
0.0    3246
1.0     100
Name: count, dtype: int64
```

[91]:
```
# Replacing the missing values in the categorical columns with its mode
df['education'] = df['education'].fillna(df['education'].mode()[0])
df['bp_meds'] = df['bp_meds'].fillna(df['bp_meds'].mode()[0])
```

[92]:
```
# education distribution after mode imputation
df.education.value_counts()
```

[92]:
```
education
1.0    1478
2.0     990
3.0     549
4.0     373
Name: count, dtype: int64
```

[36]:
```
# bp_meds distribution after mode imputation
df.bp_meds.value_counts()
```

[36]:
```
bp_meds
0.0    3290
```

```
1.0     100
Name: count, dtype: int64
```

3.2 2. cigs_per_day:

[93]:
```
# Mean and median number of cigarettes per day
df.cigs_per_day.mean().round(0),df.cigs_per_day.median()
```

[93]:
```
(9.0, 0.0)
```

[94]:
```
# All missing values in the cigs_per_day column
df[df['cigs_per_day'].isna()]
```

[94]:

id	age	education	sex	is_smoking	cigs_per_day	bp_meds	prevalent_stroke	prevalent_hyp	diabetes	total_cholesterol	bmi
422	55	1.0	0	1	NaN	0.0	0	1	0	213.0	28.66
466	45	3.0	1	1	NaN	0.0	0	1	0	170.0	26.74
469	42	1.0	1	1	NaN	0.0	0	0	0	196.0	22.06
491	61	1.0	0	1	NaN	0.0	0	1	0	356.0	27.30
538	41	1.0	0	1	NaN	0.0	0	0	0	171.0	24.35
767	54	1.0	1	1	NaN	0.0	0	0	0	219.0	26.05
1461	47	2.0	0	1	NaN	0.0	0	0	0	365.0	24.44
1467	49	1.0	0	1	NaN	0.0					
1597	57	1.0	0	1	NaN	0.0					
1986	49	2.0	0	1	NaN	0.0					
2031	58	4.0	0	1	NaN	0.0					
2087	49	4.0	1	1	NaN	0.0					
2240	47	3.0	0	1	NaN	0.0					
2284	45	2.0	1	1	NaN	0.0					
2527	58	2.0	1	1	NaN	0.0					
2638	57	1.0	1	1	NaN	0.0					
2648	39	2.0	1	1	NaN	0.0					
2723	43	4.0	1	1	NaN	0.0					
2879	42	1.0	1	1	NaN	0.0					
2945	53	2.0	1	1	NaN	0.0					
3301	55	1.0	1	1	NaN	0.0					
3378	39	3.0	0	1	NaN	0.0					

1467	0	0	0	252.0	21.45
1597	0	0	0	229.0	24.43
1986	0	1	0	233.0	25.31
2031	0	1	0	270.0	23.35
2087	0	0	0	256.0	28.21
2240	0	0	0	321.0	28.14
2284	0	0	0	248.0	27.88
2527	0	0	0	235.0	21.02
2638	0	0	0	223.0	24.74
2648	0	0	0	285.0	27.62
2723	0	0	0	222.0	25.50
2879	0	0	0	226.0	25.29
2945	0	0	0	276.0	24.21
3301	0	0	0	214.0	29.25
3378	0	0	0	197.0	19.71

id	heart_rate	glucose	ten_year_chd	pulse_pressure
422	69.0	66.0	0	72.0
466	83.0	85.0	0	46.5
469	66.0	NaN	0	50.0
491	103.0	106.0	0	70.0
538	79.0	82.0	0	52.5
767	95.0	86.0	0	38.0
1461	72.0	80.0	0	51.0
1467	72.0	89.0	0	54.0
1597	80.0	93.0	0	46.0
1986	90.0	72.0	0	56.0
2031	75.0	NaN	0	77.5
2087	93.0	85.0	1	46.0
2240	90.0	74.0	0	44.0
2284	64.0	88.0	0	49.0
2527	81.0	135.0	0	51.5
2638	62.0	103.0	0	35.0
2648	85.0	65.0	0	39.0
2723	75.0	NaN	0	40.5
2879	62.0	98.0	0	39.0
2945	58.0	82.0	0	44.0
3301	70.0	103.0	0	47.0
3378	55.0	63.0	0	50.0

D'après le tableau ci-dessus, nous constatons que pour chaque cas de valeurs manquantes dans le nombre de cigarettes par jour, les patients ont déclaré qu'ils fument.

Vérifions la moyenne et la médiane du nombre de cigarettes fumées par les patients qui ont déclaré fumer.

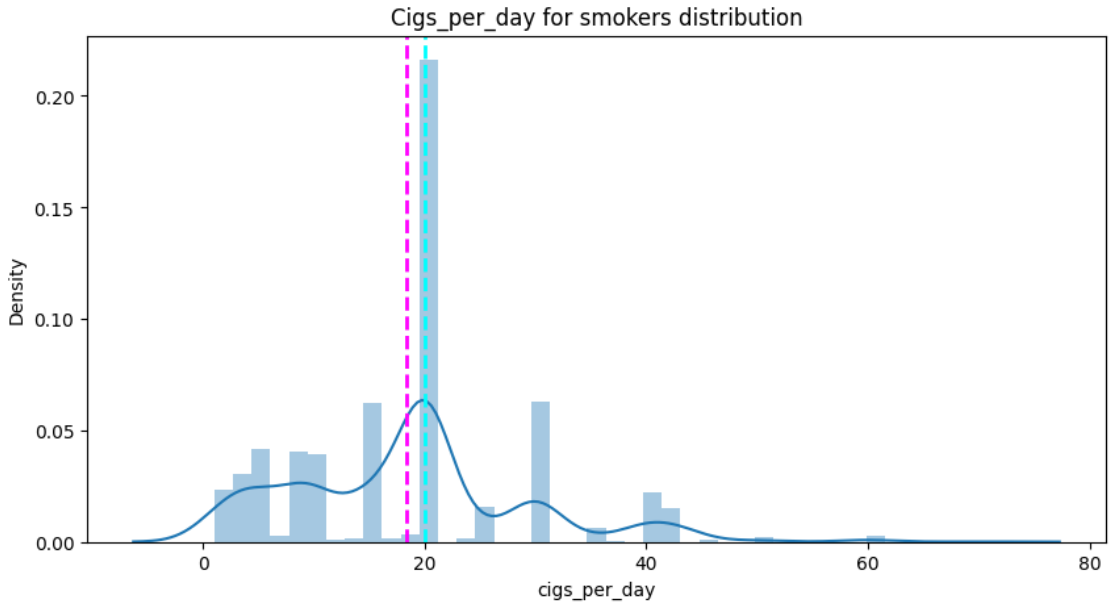
[95]:
```
# mean and median number of cigarettes per day for a smoker (excluding␣
 ↪non-smokers)
df[df['is_smoking']==1]['cigs_per_day'].
 ↪mean(),df[df['is_smoking']==1]['cigs_per_day'].median()
```

[95]:
```
(18.345945945945946, 20.0)
```

- La moyenne du nombre de cigarettes pour un fumeur est de 18,34
- la médiane du nombre de cigarettes pour un fumeur est de 20.

[96]:
```
# distribution of number of cigarettes per day for smokers (excluding␣
 ↪non-smokers)
plt.figure(figsize=(10,5))
sns.distplot(df[df['is_smoking']==1]['cigs_per_day'])
plt.axvline(df[df['is_smoking']==1]['cigs_per_day'].mean(), color='magenta',␣
 ↪linestyle='dashed', linewidth=2)
plt.axvline(df[df['is_smoking']==1]['cigs_per_day'].median(), color='cyan',␣
 ↪linestyle='dashed', linewidth=2)
plt.title('Cigs_per_day for smokers distribution')
plt.show()
```

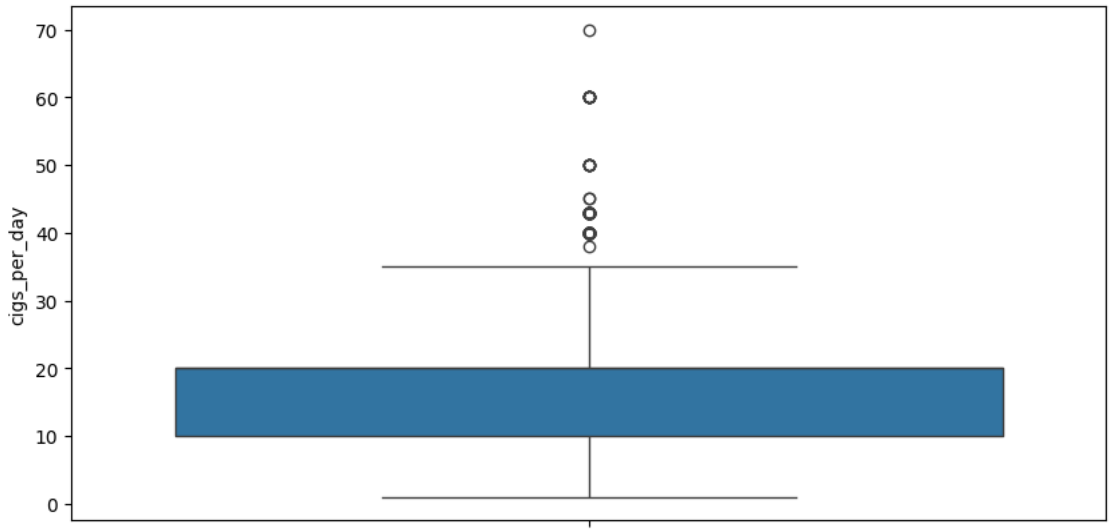
```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\2033564051.py:3: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df[df['is_smoking']==1]['cigs_per_day'])
```



On constate que la densité est maximal pour les gens fumant 20 cigarette par jours

[97]:
```
# box plot for the number of cigarettes per day for smokers (excluding␣
 ↪non-smokers)
plt.figure(figsize=(10,5))
sns.boxplot(df[df['is_smoking']==1]['cigs_per_day'])
```

[97]: <Axes: ylabel='cigs_per_day'>



Puisque le nombre de cigarettes fumées par les patients fumeurs contient des valeurs aberrantes, les valeurs manquantes dans la colonne "cigs_per_day" peuvent être imputées avec sa valeur médiane.

[98]:
```
# Imputing the missing values in the cigs_per_day
df['cigs_per_day'] = df['cigs_per_day'].
  ↪fillna(df[df['is_smoking']==1]['cigs_per_day'].median())
```

[99]:
```
# Checking for any wrong entries where the patient is not a smoker
# and cigarettes per day above 0

df[(df['is_smoking']==0) & (df['cigs_per_day']>0)]
```

[99]:
```
Empty DataFrame
Columns: [age, education, sex, is_smoking, cigs_per_day, bp_meds,
prevalent_stroke, prevalent_hyp, diabetes, total_cholesterol, bmi, heart_rate,
glucose, ten_year_chd, pulse_pressure]
Index: []
```

[100]:
```
# Checking for any wrong entries where the patient is a smoker
# and cigarettes per day is 0

df[(df['is_smoking']==1) & (df['cigs_per_day']==0)]
```

[100]:
```
Empty DataFrame
Columns: [age, education, sex, is_smoking, cigs_per_day, bp_meds,
prevalent_stroke, prevalent_hyp, diabetes, total_cholesterol, bmi, heart_rate,
glucose, ten_year_chd, pulse_pressure]
Index: []
```

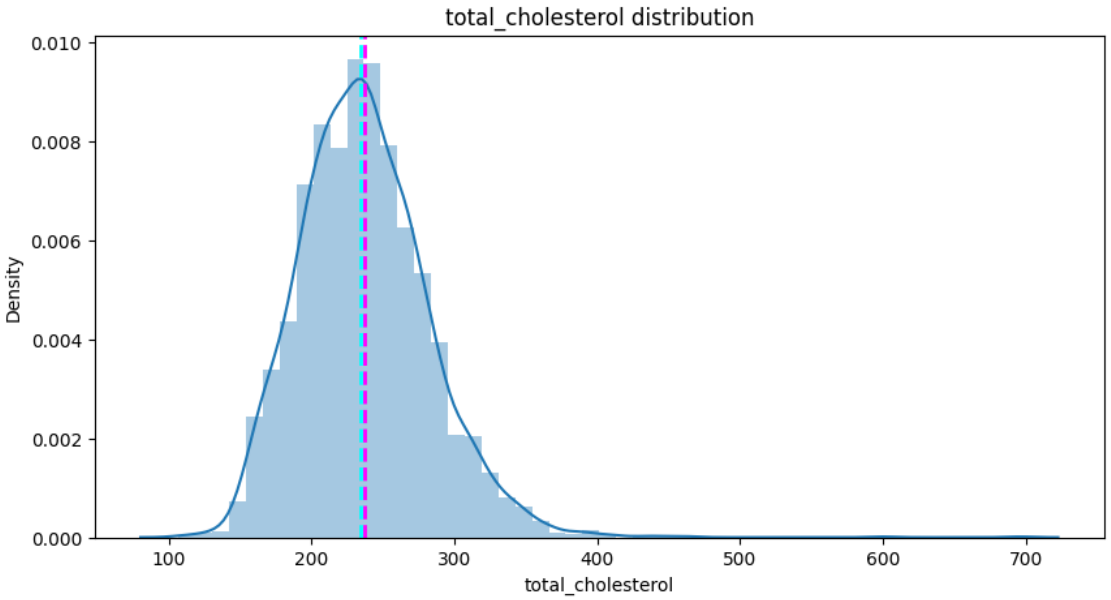
Il n'y a aucune erreur dans cette ligne.

3.3 3. total_cholestrol, bmi, heart_rate:

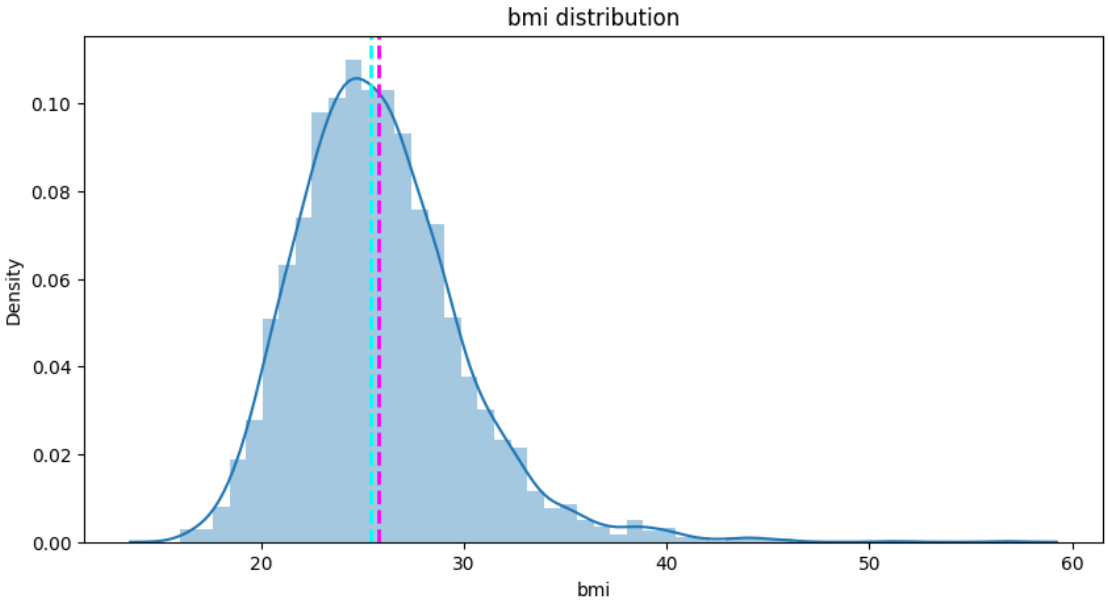
[101]:
```
# Checking the distribution of the total_cholesterol, bmi, and heart_rate
for i in ['total_cholesterol','bmi','heart_rate']:
  plt.figure(figsize=(10,5))
  sns.distplot(df[i])
  plt.axvline(df[i].mean(), color='magenta', linestyle='dashed', linewidth=2)
  plt.axvline(df[i].median(), color='cyan', linestyle='dashed', linewidth=2)
  plt.title(i+' distribution')
  plt.show()
```

```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\1580107993.py:4: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
```

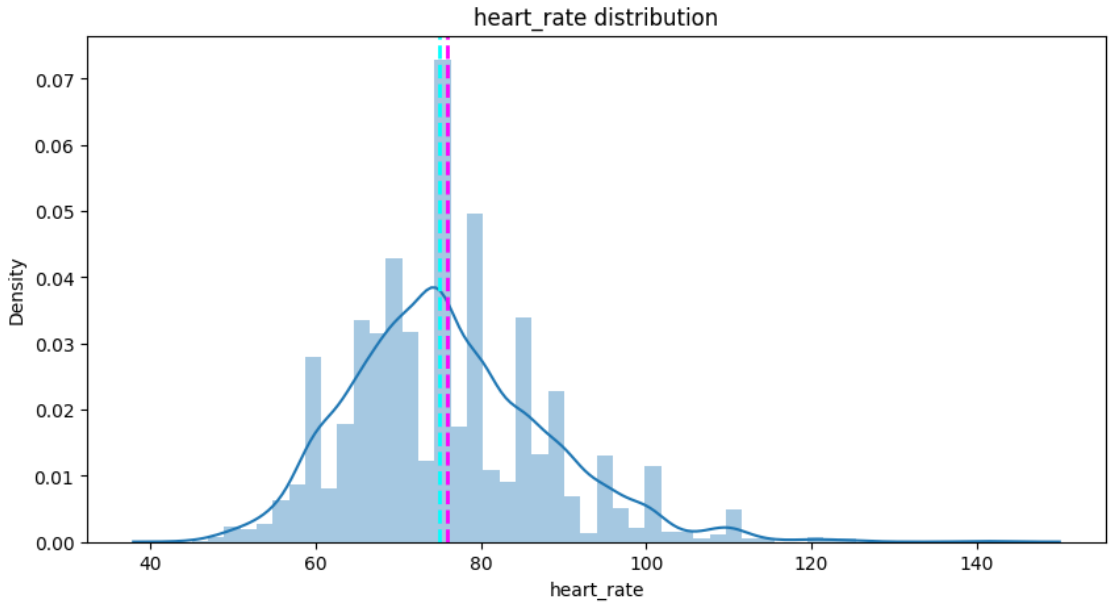
```
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df[i])
```




```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\1580107993.py:4: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df[i])
```

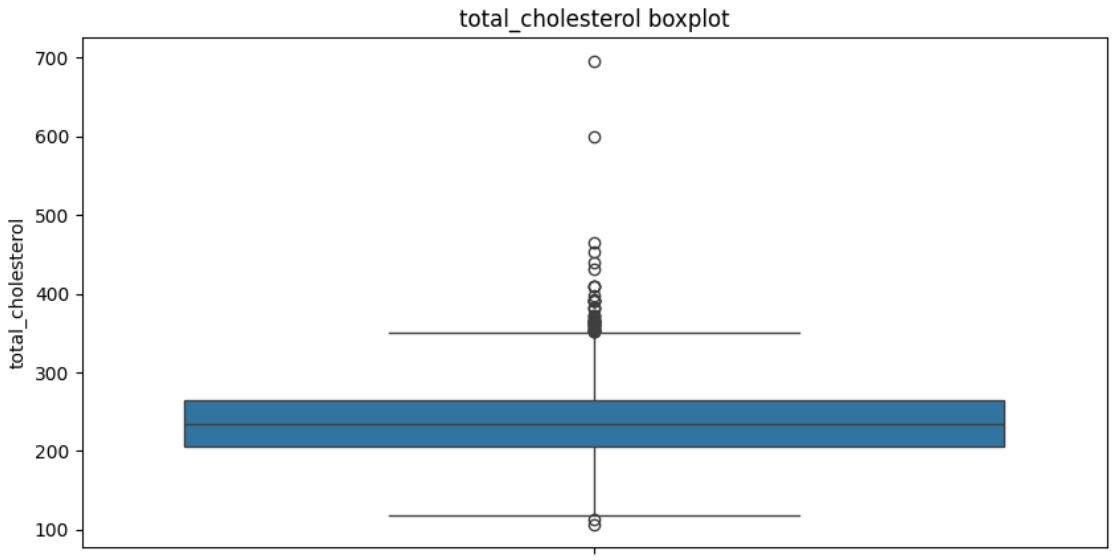


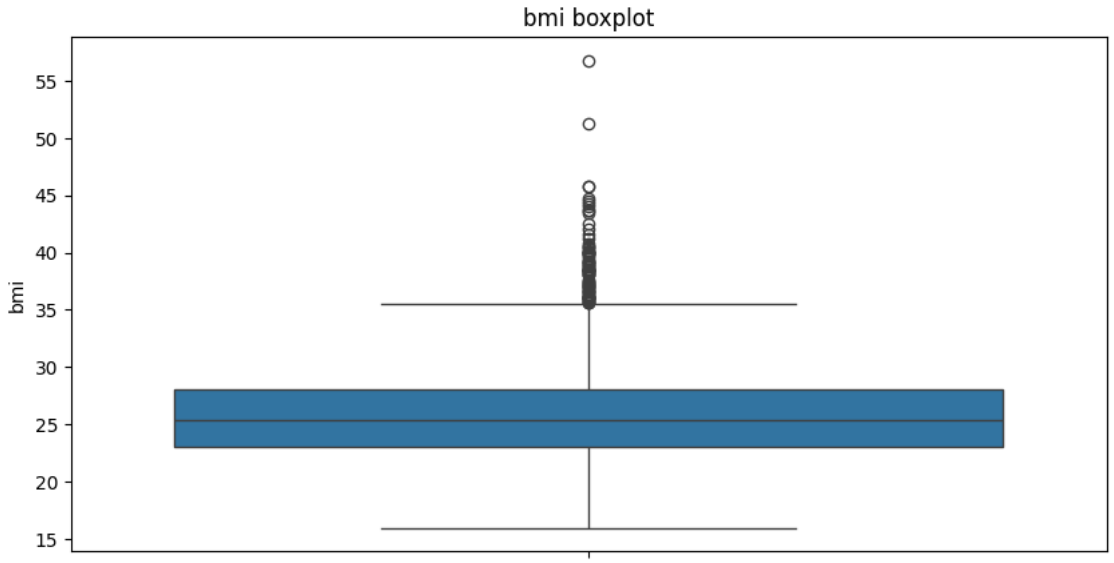

```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\1580107993.py:4: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df[i])
```

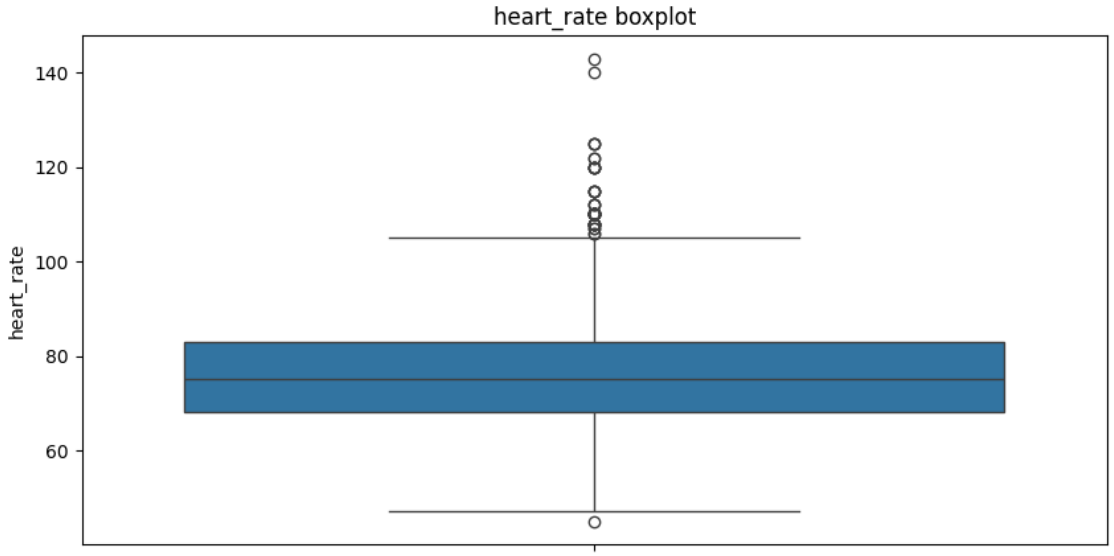


Les colonnes total_cholesterol, bmi et heart_rate présentent une asymétrie positive.

[102]:
```
# Checking outliers in total_cholesterol, bmi, heart_rate columns
for i in ['total_cholesterol','bmi','heart_rate']:
  plt.figure(figsize=(10,5))
  sns.boxplot(df[i])
  plt.title(i+' boxplot')
  plt.show()
```







les colonnes: total_cholestrol, bmi, and heart_rate contiennent des valeurs aberrantes.

[103]:
```
# Mean and median for total_cholesterol
df.total_cholesterol.mean(),df.total_cholesterol.median()
```

[103]:
```
(237.07428400954655, 234.0)
```

[104]:
```
# Mean and median for bmi
df.bmi.mean(),df.bmi.median()
```

[104]: (25.7949644549763, 25.38)

[105]:
```
# Mean and median for heart_rate
df.heart_rate.mean(),df.heart_rate.median()
```

[105]: (75.97727943346119, 75.0)

Puisque les colonnes total_cholesterol, bmi et heart_rate présentent une asymétrie positive et contiennent également des valeurs aberrantes, nous pouvons imputer les valeurs manquantes avec leur médiane.

[106]:
```
# Imputing missing values in the total_cholesterol, bmi, and heart_rate with␣
 ↪their medain values
df['total_cholesterol'] = df['total_cholesterol'].
 ↪fillna(df['total_cholesterol'].median())
df['bmi'] = df['bmi'].fillna(df['bmi'].median())
df['heart_rate'] = df['heart_rate'].fillna(df['heart_rate'].median())
```

[107]:
```
# mean and median of total_cholesterol after median imputation
df.total_cholesterol.mean(),df.total_cholesterol.median()
```

[107]: (237.03982300884957, 234.0)

[108]:
```
# mean and median of bmi after median imputation
df.bmi.mean(),df.bmi.median()
```

[108]: (25.793250737463126, 25.38)

[53]:
```
# mean and median of heart_rate after median imputation
df.heart_rate.mean(),df.heart_rate.median()
```

[53]: (75.97699115044247, 75.0)

3.4 4. glucose:

[109]:
```
# total missing values in glucose
df.glucose.isna().sum()
```

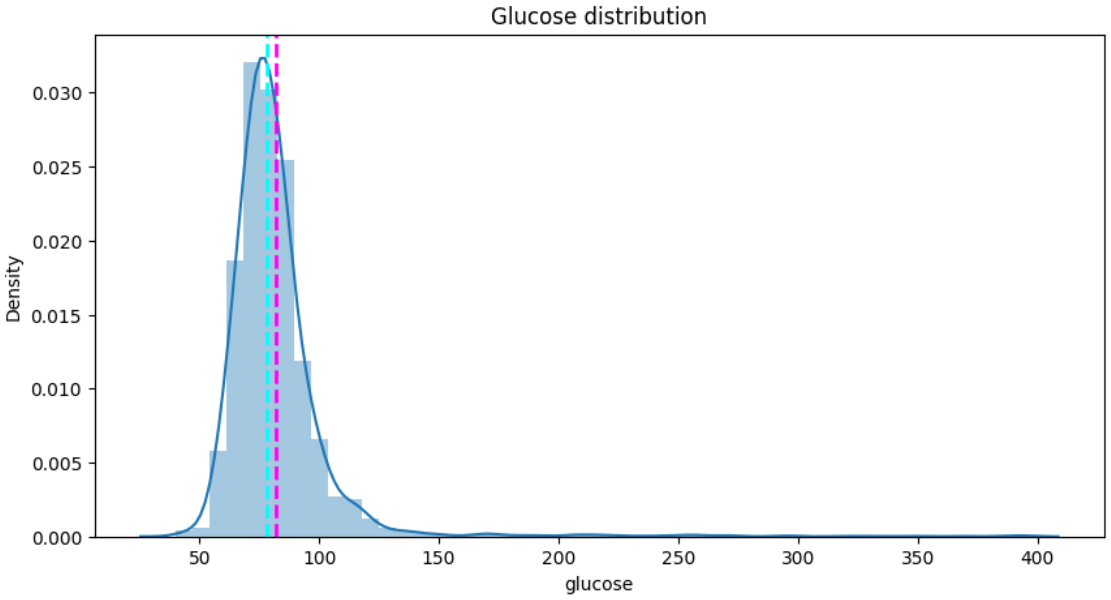
[109]: 304

La colonne "glucose" contient 304 valeurs manquantes.

[110]:
```
# distribution of glucose
plt.figure(figsize=(10,5))
sns.distplot(df['glucose'])
plt.axvline(df['glucose'].mean(), color='magenta', linestyle='dashed',␣
 ↪linewidth=2)
```

```
plt.axvline(df['glucose'].median(), color='cyan', linestyle='dashed',␣
 ↪linewidth=2)
plt.title('Glucose distribution')
plt.show()
```

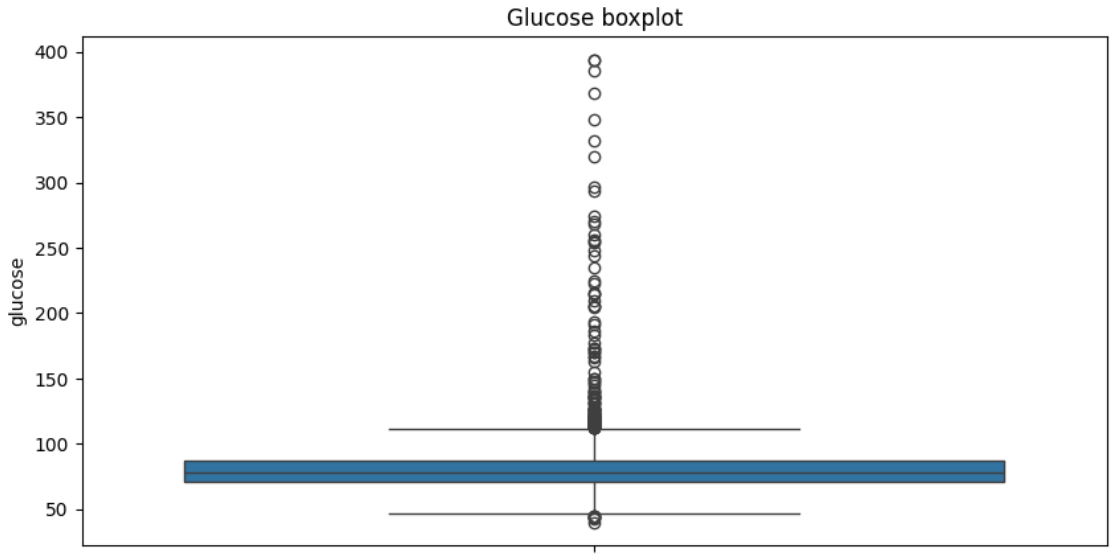
```
C:\Users\vlogi\AppData\Local\Temp\ipykernel_26096\711793079.py:3: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df['glucose'])
```



glucose présentent une asymétrie positive.

[111]:
```
# Outliers in glucose
plt.figure(figsize=(10,5))
sns.boxplot(df['glucose'])
plt.title('Glucose boxplot')
```

[111]:
```
Text(0.5, 1.0, 'Glucose boxplot')
```



la colonne "glucose" contient des valeurs aberrantes.

[112]:
```
# Mean, median, and mode for glucose
df.glucose.mean(),df.glucose.median(),df.glucose.mode()
```

[112]:
```
(82.08651976668827,
 78.0,
 0    75.0
 Name: glucose, dtype: float64)
```

- La distribution est asymétrique positive, avec des valeurs aberrantes.
- Il y a **304** valeurs manquantes dans la colonne glucose. **Si nous choisissons de les imputer avec une seule valeur de moyenne / médiane, nous introduirons un biais élevé à ce moment-là.**
- **Pour éviter cela, nous pouvons imputer les valeurs manquantes en utilisant l'imputation par les k plus proches voisins (KNN).**

[113]:
```
from sklearn.impute import KNNImputer

# Using KNN imputer with K=10
imputer = KNNImputer(n_neighbors=10)
imputed = imputer.fit_transform(df)
df = pd.DataFrame(imputed, columns=df.columns)
```

[114]:
```
# mean, median, and mode for glucose after knn imputation
df.glucose.mean(),df.glucose.median(),df.glucose.mode()
```

[114]:
```
(82.03504424778762,
 78.0,
```

```
0    75.0
Name: glucose, dtype: float64)
```

Après l'imputation KNN, il n'y a pas de changement massif dans les valeurs moyennes. Et les valeurs de médiane et de mode restent les mêmes.

[115]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3390 entries, 0 to 3389
Data columns (total 15 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   age                3390 non-null   float64
 1   education          3390 non-null   float64
 2   sex                3390 non-null   float64
 3   is_smoking         3390 non-null   float64
 4   cigs_per_day       3390 non-null   float64
 5   bp_meds            3390 non-null   float64
 6   prevalent_stroke   3390 non-null   float64
 7   prevalent_hyp      3390 non-null   float64
 8   diabetes           3390 non-null   float64
 9   total_cholesterol  3390 non-null   float64
 10  bmi                3390 non-null   float64
 11  heart_rate         3390 non-null   float64
 12  glucose            3390 non-null   float64
 13  ten_year_chd       3390 non-null   float64
 14  pulse_pressure     3390 non-null   float64
dtypes: float64(15)
memory usage: 397.4 KB
```

L'imputeur KNN a converti toutes les colonnes en type de données float64. Par conséquent, il faut changer le type de données des colonnes en fonction du type de données stockées dans chaque colonne respective.

[116]:
```
# changing datatypes
df = df.astype({'age': int, 'education':int,'sex':int,'is_smoking':
 ↪int,'cigs_per_day':int,
                'bp_meds':int,'prevalent_stroke':int,'prevalent_hyp':
 ↪int,'diabetes':int,
                'total_cholesterol':float,'bmi':float,'heart_rate':
 ↪float,'glucose':float,
                'ten_year_chd':int})
```

[117]:
```
# checking for missing values
df.isna().sum()
```

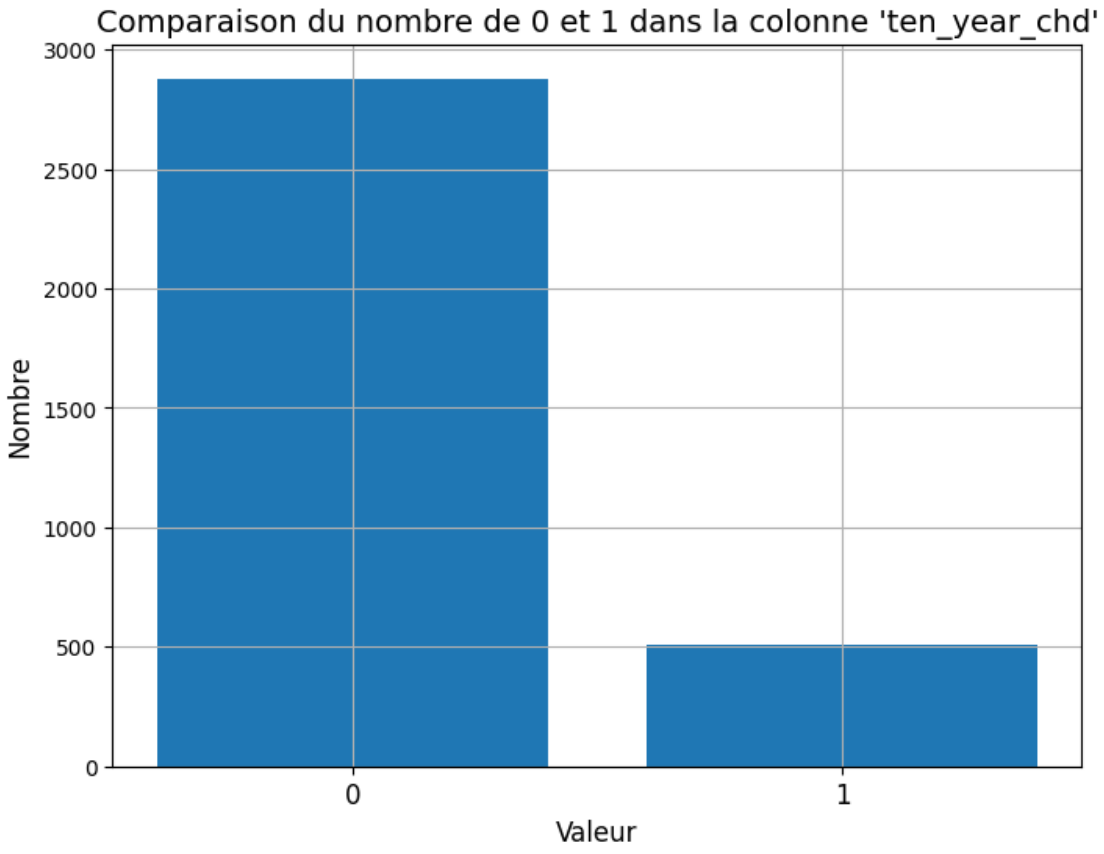
[117]:
```
age                 0
education           0
sex                 0
is_smoking          0
cigs_per_day        0
bp_meds             0
prevalent_stroke    0
prevalent_hyp       0
diabetes            0
total_cholesterol   0
bmi                 0
heart_rate          0
glucose             0
ten_year_chd        0
pulse_pressure      0
dtype: int64
```

Nous avons réussi à gérer toutes les valeurs manquantes dans l'ensemble de données.

3.5 5. Traitement du déséquilibre des données :

[119]:
```
def afficher_comparaison_0_1(dataframe, nom_colonne_cible):
    """
    Crée un graphique à barres comparant le nombre de 0 et de 1 dans une colonne␣
↪cible d'un dataframe.

    Args:
        dataframe (pd.DataFrame): Le DataFrame contenant la colonne cible.
        nom_colonne_cible (str): Le nom de la colonne cible contenant les valeurs␣
↪0 et 1.
    """
    try:
        # Compter les occurrences de 0 et 1
        compte_0 = (dataframe[nom_colonne_cible] == 0).sum()
        compte_1 = (dataframe[nom_colonne_cible] == 1).sum()

        # Créer les données pour le graphique
        donnees = pd.DataFrame({'Valeur': [0, 1], 'Nombre': [compte_0, compte_1]})

        # Créer le graphique à barres
        plt.figure(figsize=(8, 6))  # Ajuster la taille du graphique si nécessaire
        plt.bar(donnees['Valeur'], donnees['Nombre'])
        plt.xlabel("Valeur", fontsize=12)
        plt.ylabel("Nombre", fontsize=12)
        plt.title("Comparaison du nombre de 0 et 1 dans la colonne '{}'".
↪format(nom_colonne_cible), fontsize=14)
```

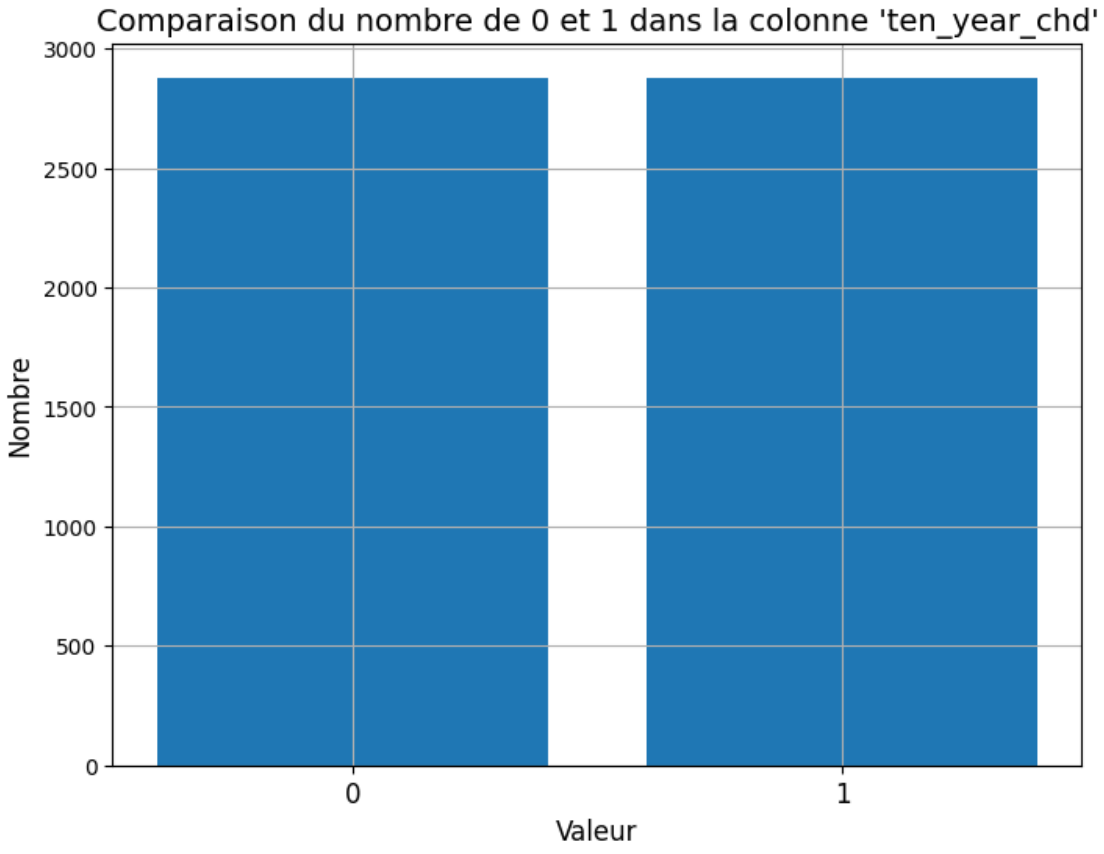
```
    plt.xticks([0, 1], ['0', '1'], fontsize=12)  # Afficher les étiquettes 0 et␣
 ↪1 sur l'axe des x
    plt.grid(True)
    plt.show()

  except KeyError:
    print("Erreur : La colonne '{}' n'existe pas dans le DataFrame.".
 ↪format(nom_colonne_cible))
  except Exception as e:
    print("Erreur inattendue :", e)

# Exemple d'utilisation (remplacer par votre DataFrame et le nom de la colonne)
# dataframe = pd.DataFrame({'target': [0, 1, 1, 0, 1, 0, 0, 1]})
afficher_comparaison_0_1(df, 'ten_year_chd')
```



Il est évident que les classes sont déséquilibrées, ce qui ferait que nos modèles seront d'avantage biaisé en faveur de la classe « 0 » (personnes sans risque de maladie coronarienne) car seulement environ 15 % des patients ont reçu un diagnostic de maladie coronarienne. nous suréchantillonnons l'ensemble de données du train à l'aide de SMOTE (Synthetic Minority Oversampling Technique). Cela garantit que nos modèles seront entraîné de manière égale sur tous les types de résultats et

qu'il ne seront pas biaisé en faveur d'un résultat particulier.

[120]:
```
# Defining dependent and independent variables
x = df.drop('ten_year_chd',axis=1)
y = df[dependent_var]

from imblearn.over_sampling import SMOTE

# Fitting the data
smote = SMOTE(sampling_strategy='minority')
x_sm, y_sm = smote.fit_resample(x, y)

df = pd.concat([x_sm, y_sm], axis=1)
afficher_comparaison_0_1(df, 'ten_year_chd')
```



Nous avons réussi à suréchantillonner la classe minoritaire en utilisant SMOTE. Désormais, les modèles que nous allons constre pourront apprendre des deux classes sans aucun parti pris.

3.6 6. Sauvegarde des données préparées:

[122]:
```
df.to_csv('clean_data.csv', index=False)
```

Models_Training

April 13, 2024

1 III. Entrainement des Models

1.1 1. Importation des bibliothèques et du jeu de données (préparé préalablement)

Nous commençons par l'importation des bibliothèques nécessaires telles que pandas, numpy et pyspark. Ces bibliothèques sont essentielles pour le traitement des données, la manipulation des tableaux et la mise en œuvre des algorithmes d'apprentissage automatique à grande échelle.

[4]:
```
import pandas as pd
import numpy as np
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression,␣
 ↪RandomForestClassifier, DecisionTreeClassifier
from pyspark.ml.evaluation import␣
 ↪BinaryClassificationEvaluator,MulticlassClassificationEvaluator
from pyspark.ml.classification import LinearSVC
from sklearn.metrics import classification_report
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
from prettytable import PrettyTable
```

En utilisant SparkSession de pyspark.sql, nous initialisons une session Spark, ce qui nous permettra de manipuler des données distribuées de manière efficace.

[5]:
```
# Initialisation de la session Spark // Port : 4040
# Une fois la seesion crée, on peut interagir avec les données stockées dans un␣
 ↪environnement distribué.
spark = SparkSession.builder.appName('HeartDiseasesPredicion').config("spark.ui.
 ↪port", "4041").config("spark.driver.maxResultSize", "4g").getOrCreate()
```

Nous procédons ensuite à l'importation des données à partir du fichier CSV 'clean_data.csv' contenant les données traitées précédemment à l'aide de la méthode spark.read.csv() de Spark

[6]:
```
# Importing data
path = 'clean_data.csv'
data = spark.read.csv(path, header=True, inferSchema=True)
```

Ensuite, nous assemblons les caractéristiques pertinentes en un vecteur à l'aide de VectorAssembler de pyspark.ml.feature. Nous définissons une liste de colonnes 'feature_columns' qui contient les noms des caractéristiques que nous voulons inclure dans notre vecteur de caractéristiques.

[7]:
```
# Assembler les caractéristiques en un vecteur
feature_columns = ['age', 'education', 'sex', 'is_smoking', 'cigs_per_day',␣
 ↪'bp_meds', 'prevalent_stroke', 'prevalent_hyp', 'diabetes',␣
 ↪'total_cholesterol', 'bmi', 'heart_rate', 'glucose', 'pulse_pressure']
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(data)
```

Ensuite, nous divisons nos données en ensembles d'entrainement et de test à l'aide de la méthode randomSplit de Spark.

[8]:
```
# Diviser les données en ensembles de formation et de test
(trainingData, testData) = data.randomSplit([0.7, 0.3], seed=42)
```

Ensuite, nous configurons un évaluateur pour évaluer les performances de nos modèles sur les données de test. Dans ce cas, nous utilisons BinaryClassificationEvaluator de pyspark.ml.evaluation, spécifiant la colonne contenant les étiquettes de nos données (labelCol="ten_year_chd").

[9]:
```
# Évaluer les modèles sur les données de test
evaluator = BinaryClassificationEvaluator(labelCol="ten_year_chd")
```

Ensuite, nous créons un évaluateur spécifiquement pour mesurer le rappel (recall) des modèles sur les données de test. Pour cela, nous utilisons MulticlassClassificationEvaluator de pyspark.ml.evaluation, en spécifiant la colonne contenant les étiquettes réelles (labelCol="ten_year_chd"), la colonne contenant les prédictions des modèles, et le nom de la métrique que nous voulons calculer (metricName="weightedRecall").

[19]:
```
# Créer un évaluateur pour le recall
evaluator_recall = MulticlassClassificationEvaluator(labelCol="ten_year_chd",␣
 ↪predictionCol="prediction", metricName="weightedRecall")
```

1.2 2. Logistic Regression

Dans cette section, nous abordons la première technique de modélisation, à savoir la régression logistique. Nous utilisons la bibliothèque pyspark.ml.classification pour mettre en œuvre ce modèle. Nous instancions un objet LogisticRegression en spécifiant les colonnes contenant les étiquettes (labelCol="ten_year_chd") et les caractéristiques (featuresCol="features") dans nos données.

[10]:
```
# Entraîner plusieurs modèles de classification
log_reg = LogisticRegression(labelCol="ten_year_chd", featuresCol="features")
```

Nous définissons ici une grille de paramètres à tester pour le modèle de régression logistique. Nous utilisons ParamGridBuilder de pyspark.ml.tuning pour spécifier les différentes combinaisons de paramètres que nous voulons évaluer lors de l'entraînement du modèle.

[11]:
```
# Paramètres à tester pour chaque modèle
param_grid_log_reg = ParamGridBuilder() \
    .addGrid(log_reg.regParam, [0.01, 0.1, 1.0]) \
    .addGrid(log_reg.elasticNetParam, [0.0, 0.5, 1.0]) \
    .build()
```

Nous créons ici un validateur croisé (CrossValidator) pour le modèle de régression logistique. Le validateur croisé exécutera la validation croisée sur différentes combinaisons de paramètres, sélectionnant finalement la combinaison optimale qui maximise les performances du modèle selon les critères définis par l'évaluateur.

[12]:
```
# Instancier les validateurs pour chaque modèle
cv_log_reg = CrossValidator(estimator=log_reg,
                            estimatorParamMaps=param_grid_log_reg,
                            evaluator=evaluator,
                            numFolds=5)
```

Nous procédons ici à l'entraînement du validateur croisé pour le modèle de régression logistique.

[13]:
```
# Entraîner les validateurs
cv_log_reg_model = cv_log_reg.fit(trainingData)
```

Nous sélectionnons ici le meilleur modèle de régression logistique parmi ceux entraînés par le validateur croisé.

[14]:
```
# Sélectionner les meilleurs modèles
best_log_reg_model = cv_log_reg_model.bestModel
```

Nous évaluons ici les performances du meilleur modèle de régression logistique sur l'ensemble de test. En utilisant l'évaluateur que nous avons précédemment défini

[15]:
```
# Évaluer les meilleurs modèles
log_reg_test_results = evaluator.evaluate(best_log_reg_model.
  ↪transform(testData))
print("Logistic Regression Test Area Under ROC:", log_reg_test_results)
```

```
Logistic Regression Test Area Under ROC: 0.733519537361439
```

le résultat obtenu pour l'AUC de la régression logistique sur l'ensemble de test est d'environ 0.734, ce qui indique que le modèle a une capacité modérée à bien discriminer entre les individus à risque de développer une maladie cardiaque et ceux qui ne le sont pas.

[16]:
```
# Afficher les meilleurs paramètres pour Logistic Regression
best_log_reg_params = cv_log_reg_model.bestModel.extractParamMap()
print("Meilleurs paramètres pour Logistic Regression:")
for param in best_log_reg_params:
```

```
    print(param.name, ":", best_log_reg_params[param])
```

```
Meilleurs paramètres pour Logistic Regression:
aggregationDepth : 2
elasticNetParam : 0.0
family : auto
featuresCol : features
fitIntercept : True
labelCol : ten_year_chd
maxBlockSizeInMB : 0.0
maxIter : 100
predictionCol : prediction
probabilityCol : probability
rawPredictionCol : rawPrediction
regParam : 0.01
standardization : True
threshold : 0.5
tol : 1e-06
```

Nous générons ici un rapport de classification pour évaluer plus en détail les performances du modèle de régression logistique sur l'ensemble de test.

[17]:
```
log_reg_predictions_pd = best_log_reg_model.transform(testData).
 ↪select("prediction", "ten_year_chd").toPandas()
log_reg_metrics = classification_report(log_reg_predictions_pd["ten_year_chd"],␣
 ↪log_reg_predictions_pd["prediction"], zero_division=0)
# Afficher les rapports de classification
print("Logistic Regression Classification Report:")
print(log_reg_metrics)
```

```
Logistic Regression Classification Report:
              precision    recall  f1-score   support

           0       0.68      0.63      0.66       831
           1       0.65      0.70      0.67       813

    accuracy                           0.67      1644
   macro avg       0.67      0.67      0.67      1644
weighted avg       0.67      0.67      0.67      1644
```

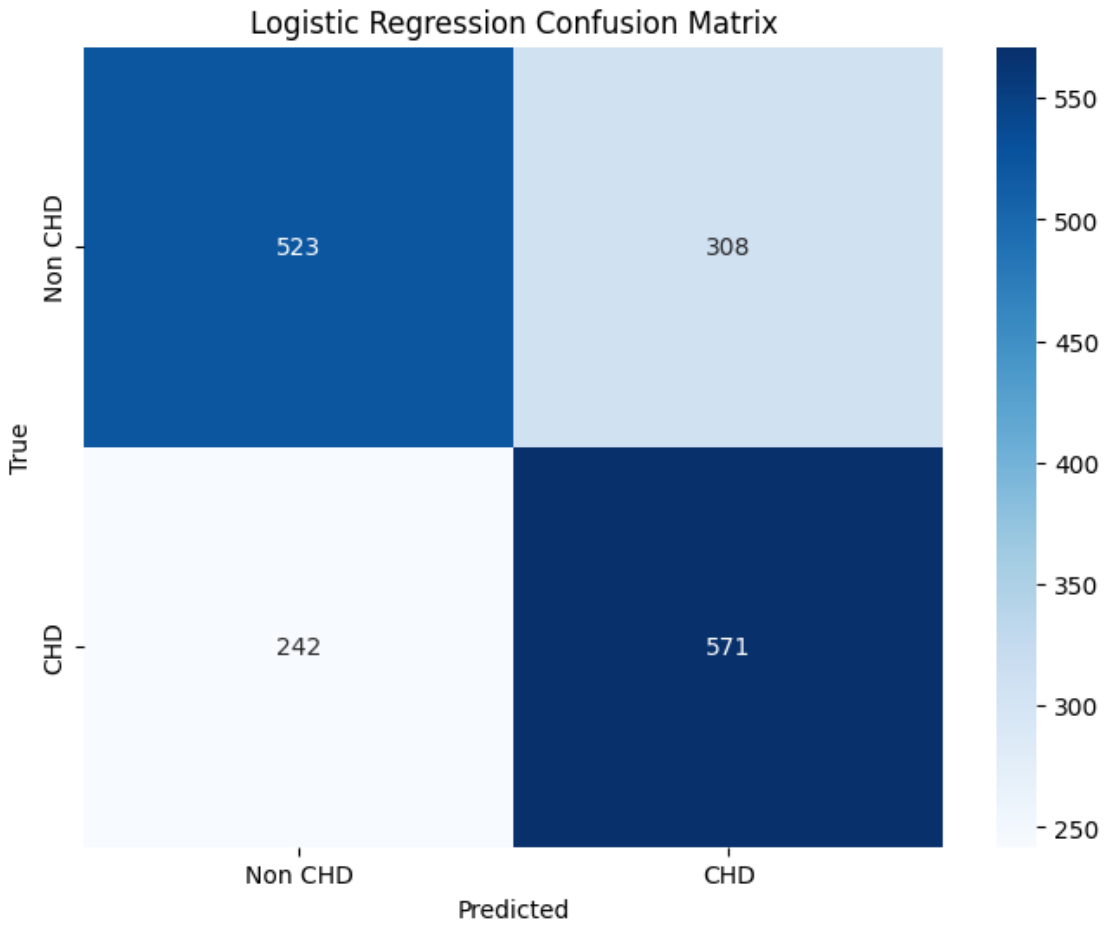
Pour la classe 0 (pas de développement de maladie cardiaque), le modèle a une précision de 0.68, ce qui signifie que parmi toutes les prédictions de cette classe, 68% sont correctes. Le rappel pour cette classe est de 0.63, indiquant que le modèle identifie correctement 63% de tous les exemples réels de cette classe. Pour la classe 1 (développement de maladie cardiaque), le modèle a une précision de 0.65, ce qui signifie que parmi toutes les prédictions de cette classe, 65% sont correctes. Le rappel pour cette classe est de 0.70, indiquant que le modèle identifie correctement 70% de tous les exemples réels de cette classe.

Nous évaluons ici le rappel (recall) du meilleur modèle de régression logistique sur les ensembles d'entraînement et de test. Pour cela, nous utilisons l'évaluateur spécifique au rappel (evaluator_recall) que nous avons précédemment défini.

[20]:
```
# Calculer le recall pour l'ensemble de test
log_reg_test_recall = evaluator_recall.evaluate(best_log_reg_model.
 ↪transform(testData))

# Calculer le recall pour l'ensemble d'entraînement
log_reg_train_recall = evaluator_recall.evaluate(best_log_reg_model.
 ↪transform(trainingData))

print("Logistic Regression Test Recall:", log_reg_test_recall)
print("Logistic Regression Train Recall:", log_reg_train_recall)
```

```
Logistic Regression Test Recall: 0.6654501216545012
Logistic Regression Train Recall: 0.670393777345649
```

[21]:
```
# Définition d'une fonction pour afficher la matrice de confusion
def plot_confusion_matrix(cm, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Non CHD',␣
 ↪'CHD'], yticklabels=['Non CHD', 'CHD'])
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title(title)
    plt.show()
# Convertir les prédictions de Spark DataFrame en Pandas DataFrame
log_reg_cm = best_log_reg_model.transform(testData).select("prediction",␣
 ↪"ten_year_chd").toPandas()
# Calculer la matrice de confusion pour chaque modèle
log_reg_cm = confusion_matrix(log_reg_cm["ten_year_chd"],␣
 ↪log_reg_cm["prediction"])
# Afficher les matrices de confusion
plot_confusion_matrix(log_reg_cm, 'Logistic Regression Confusion Matrix')
```



La matrice de confusion que nous avons affichée met en évidence la performance du modèle de régression logistique dans la prédiction des cas positifs et négatifs de développement de maladie cardiaque sur l'ensemble de test. Cependant, il est important de noter que le taux de faux négatifs est élevé, ce qui signifie que le modèle a du mal à détecter correctement les cas réels de développement de maladie cardiaque parmi ceux qui sont prédits comme non affectés. Cette situation est particulièrement préoccupante dans le contexte médical, car un taux élevé de faux négatifs peut signifier que des individus à risque de développer une maladie cardiaque pourraient ne pas être identifiés

1.3 3. Decision Tree

Pour l'entraînement du modèle d'arbre de décision, nous avons suivi un processus similaire à celui de la régression logistique.

[22]:
```
dt = DecisionTreeClassifier(labelCol="ten_year_chd", featuresCol="features")
```

[23]:
```
param_grid_dt = ParamGridBuilder() \
    .addGrid(dt.maxDepth, [5, 10, 15]) \
```

```
    .addGrid(dt.maxBins, [16, 32, 64]) \
    .build()
```

[24]:
```
cv_dt = CrossValidator(estimator=dt,
                       estimatorParamMaps=param_grid_dt,
                       evaluator=evaluator,
                       numFolds=5)
```

[25]:
```
cv_dt_model = cv_dt.fit(trainingData)
```

[26]:
```
best_dt_model = cv_dt_model.bestModel
```

[27]:
```
dt_test_results = evaluator.evaluate(best_dt_model.transform(testData))
print("Decision Tree Test Area Under ROC:", dt_test_results)
```

```
Decision Tree Test Area Under ROC: 0.8124963921119357
```

Les performances du meilleur modèle d'arbre de décision ont été évaluées sur l'ensemble de test, et l'aire sous la courbe ROC (ROC AUC) obtenue est d'environ 0.812.

[28]:
```
# Afficher les meilleurs paramètres pour Decision Tree
best_dt_params = cv_dt_model.bestModel.extractParamMap()
print("Meilleurs paramètres pour Decision Tree:")
for param in best_dt_params:
    print(param.name, ":", best_dt_params[param])
```

```
Meilleurs paramètres pour Decision Tree:
cacheNodeIds : False
checkpointInterval : 10
featuresCol : features
impurity : gini
labelCol : ten_year_chd
leafCol :
maxBins : 16
maxDepth : 15
maxMemoryInMB : 256
minInfoGain : 0.0
minInstancesPerNode : 1
minWeightFractionPerNode : 0.0
predictionCol : prediction
probabilityCol : probability
rawPredictionCol : rawPrediction
seed : -5274216314403460591
```

[29]:
```
dt_predictions_pd = best_dt_model.transform(testData).select("prediction",␣
 ↪"ten_year_chd").toPandas()
dt_metrics = classification_report(dt_predictions_pd["ten_year_chd"],␣
 ↪dt_predictions_pd["prediction"], zero_division=0)
```

```
print("Decision Tree Classification Report:")
print(dt_metrics)
```

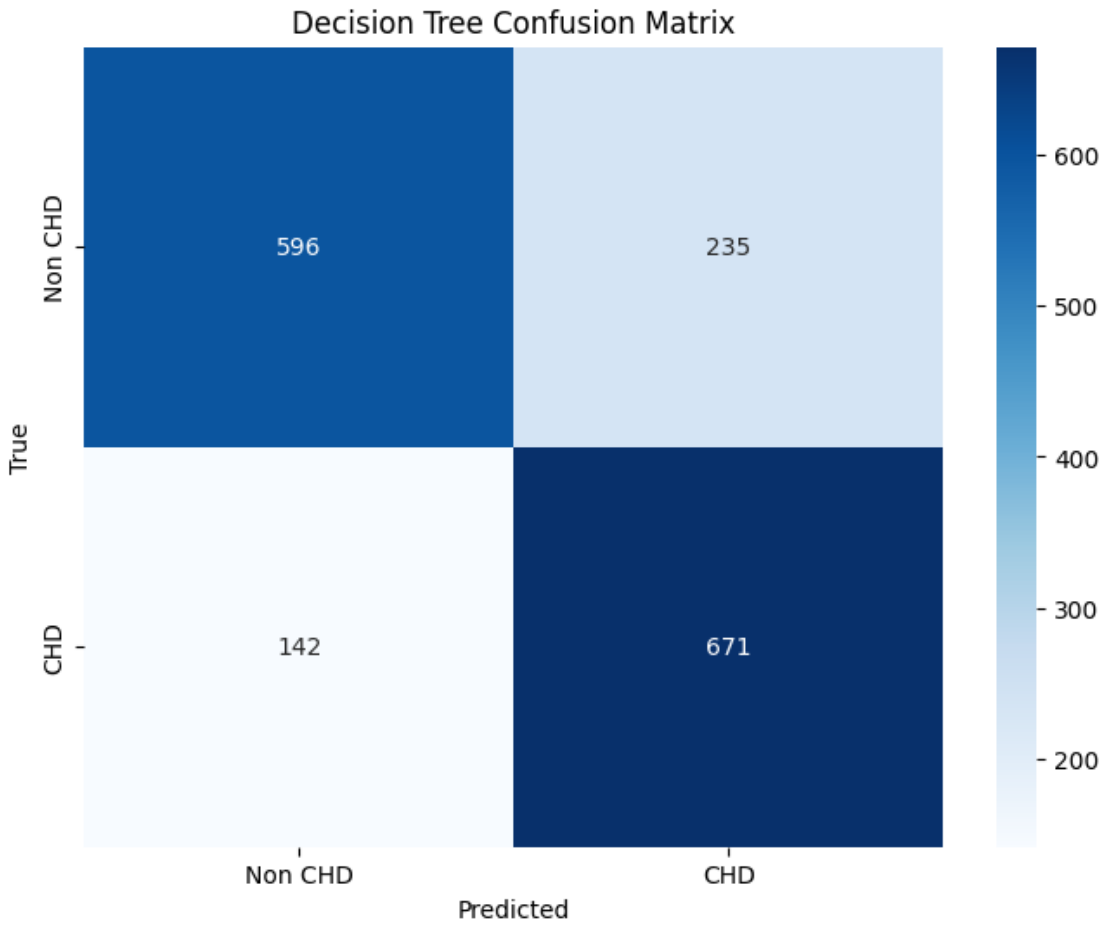
```
Decision Tree Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.72      0.76       831
           1       0.74      0.83      0.78       813

    accuracy                           0.77      1644
   macro avg       0.77      0.77      0.77      1644
weighted avg       0.77      0.77      0.77      1644
```

Globalement, le modèle d'arbre de décision semble plus performant par rapport au model de la regression logistic, avec des scores de précision, de rappel et de F1-score assez équilibrés pour les deux classes

[30]:
```
# Calculer le recall pour l'ensemble de test
dt_test_recall = evaluator_recall.evaluate(best_dt_model.transform(testData))

# Calculer le recall pour l'ensemble d'entraînement
dt_train_recall = evaluator_recall.evaluate(best_dt_model.
  ↪transform(trainingData))

print("Logistic Regression Test Recall:", dt_test_recall)
print("Logistic Regression Train Recall:", dt_train_recall)
```

```
Logistic Regression Test Recall: 0.7706812652068127
Logistic Regression Train Recall: 0.9795819154107924
```

[31]:
```
dt_cm = best_dt_model.transform(testData).select("prediction", "ten_year_chd").
  ↪toPandas()
dt_cm = confusion_matrix(dt_cm["ten_year_chd"], dt_cm["prediction"])
plot_confusion_matrix(dt_cm, 'Decision Tree Confusion Matrix')
```



avec l'arbre de décision, nous avons réussi à réduire le nombre de faux négatifs par rapport à la régression logistique. Cela indique que le modèle d'arbre de décision a une capacité modérément élevée à discriminer entre les individus à risque de développer une maladie cardiaque et ceux qui ne le sont pas, avec une performance supérieure à celle obtenue avec le modèle de régression logistique précédent. Cette amélioration pourrait indiquer que la structure d'arbre de décision est mieux adaptée à la nature des données ou qu'elle capture mieux les relations complexes entre les caractéristiques et la variable cible.

1.4 4. Random Forest

Dans ce passage vers l'utilisation du modèle de forêt aléatoire, nous tirons parti de l'observation que la structure du modèle d'arbre de décision s'est avérée être la mieux adaptée à nos données et capable de capturer les relations complexes entre les différentes caractéristiques. Sachant que la forêt aléatoire est construite à partir de plusieurs arbres de décision, il est logique de s'attendre à des performances encore meilleures.

En combinant les prédictions de multiples arbres de décision, la forêt aléatoire peut atténuer les biais individuels de chaque arbre et réduire le risque de sur-apprentissage aux données d'entraînement. En conséquence, nous anticipons des scores de performance plus élevés, reflétant une meilleure

capacité à généraliser les relations entre les caractéristiques et la variable cible sur de nouvelles données.

[32]:
```
rf = RandomForestClassifier(labelCol="ten_year_chd", featuresCol="features")
```

[33]:
```
param_grid_rf = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [5, 10, 15]) \
    .addGrid(rf.numTrees, [20, 50, 100]) \
    .build()
```

[34]:
```
cv_rf = CrossValidator(estimator=rf,
                       estimatorParamMaps=param_grid_rf,
                       evaluator=evaluator,
                       numFolds=5)
```

[35]:
```
cv_rf_model = cv_rf.fit(trainingData)
```

[36]:
```
best_rf_model = cv_rf_model.bestModel
```

[37]:
```
rf_test_results = evaluator.evaluate(best_rf_model.transform(testData))
print("Random Forest Test Area Under ROC:", rf_test_results)
```

```
Random Forest Test Area Under ROC: 0.9310082992526778
```

Le score d'aire sous la courbe ROC (ROC AUC) obtenu pour le modèle de forêt aléatoire est remarquablement élevé, atteignant environ 0.934. Cela indique une excellente capacité du modèle à discriminer entre les individus à risque de développer une maladie cardiaque et ceux qui ne le sont pas.

[38]:
```
# Afficher les meilleurs paramètres pour Random Forest
best_rf_params = cv_rf_model.bestModel.extractParamMap()
print("Meilleurs paramètres pour Random Forest:")
for param in best_rf_params:
    print(param.name, ":", best_rf_params[param])
```

```
Meilleurs paramètres pour Random Forest:
bootstrap : True
cacheNodeIds : False
checkpointInterval : 10
featureSubsetStrategy : auto
featuresCol : features
impurity : gini
labelCol : ten_year_chd
leafCol :
maxBins : 32
maxDepth : 15
maxMemoryInMB : 256
minInfoGain : 0.0
minInstancesPerNode : 1
```

```
minWeightFractionPerNode : 0.0
numTrees : 100
predictionCol : prediction
probabilityCol : probability
rawPredictionCol : rawPrediction
seed : -7033929482271434104
subsamplingRate : 1.0
```

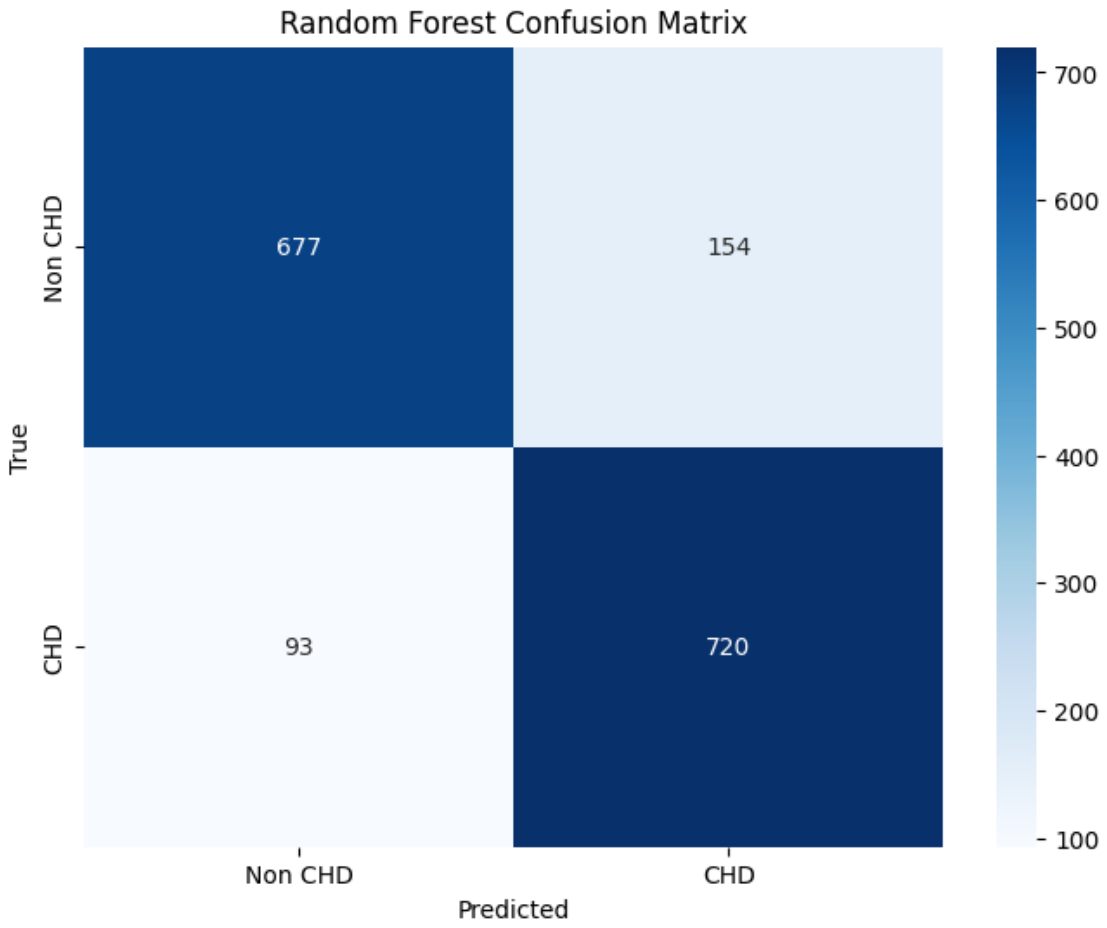
[39]:
```
rf_predictions_pd = best_rf_model.transform(testData).select("prediction",␣
 ↪"ten_year_chd").toPandas()
rf_metrics = classification_report(rf_predictions_pd["ten_year_chd"],␣
 ↪rf_predictions_pd["prediction"], zero_division=0)
print("Random Forest Classification Report:")
print(rf_metrics)
```

```
Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.81      0.85       831
           1       0.82      0.89      0.85       813

    accuracy                           0.85      1644
   macro avg       0.85      0.85      0.85      1644
weighted avg       0.85      0.85      0.85      1644
```

[40]:
```
# Calculer le recall pour l'ensemble de test
rf_test_recall = evaluator_recall.evaluate(best_rf_model.transform(testData))

# Calculer le recall pour l'ensemble d'entraînement
rf_train_recall = evaluator_recall.evaluate(best_rf_model.
 ↪transform(trainingData))

print("Logistic Regression Test Recall:",rf_test_recall)
print("Logistic Regression Train Recall:", rf_train_recall)
```

```
Logistic Regression Test Recall: 0.8497566909975669
Logistic Regression Train Recall: 0.9934370442391833
```

[41]:
```
rf_cm = best_rf_model.transform(testData).select("prediction", "ten_year_chd").
 ↪toPandas()
rf_cm = confusion_matrix(rf_cm["ten_year_chd"], rf_cm["prediction"])
plot_confusion_matrix(rf_cm, 'Random Forest Confusion Matrix')
```



En examinant la matrice de confusion, nous observons une nette amélioration par rapport aux modèles précédents, en particulier en ce qui concerne le taux de faux négatifs (FN). Cela confirme notre hypothèse initiale selon laquelle la forêt aléatoire a la capacité de mieux séparer les deux classes par rapport aux modèles individuels tels que la régression logistique et l'arbre de décision.

1.5 5. Support Vector Machine

Dans notre démarche pour choisir le modèle le plus adapté, nous avons décidé de tester le modèle de Support Vector Machine (SVM) en raison de sa structure différente par rapport aux arbres de décision.

[42]:
```
svm = LinearSVC(labelCol="ten_year_chd", featuresCol="features")
```

[43]:
```
# Construire une grille de recherche de paramètres pour chaque modèle
paramGridSVM = ParamGridBuilder() \
    .addGrid(svm.maxIter, [10, 20, 30]) \
    .addGrid(svm.regParam, [0.1, 0.01]) \
    .build()
```

[44]:
```
# Initialiser les validateurs pour SVM
cvSVM = CrossValidator(estimator=svm,
                       estimatorParamMaps=paramGridSVM,
                       evaluator=evaluator,
                       numFolds=5)
```

[45]:
```
# Entraîner les validateurs
cvSVM_model = cvSVM.fit(trainingData)
```

[46]:
```
# Sélectionner les meilleurs modèles
best_svm_model = cvSVM_model.bestModel
```

[47]:
```
svm_test_results = evaluator.evaluate(best_svm_model.transform(testData))
print("Support Vector Machine Test Area Under ROC:", svm_test_results)
```

```
Support Vector Machine Test Area Under ROC: 0.7322776837876702
```

Le score d'aire sous la courbe ROC (ROC AUC) obtenu pour le modèle de Support Vector Machine (SVM) est d'environ 0.732

[48]:
```
# Afficher les meilleurs paramètres pour SVM.
print("Meilleurs paramètres pour SVM:")
print("MaxIter:", best_svm_model.getMaxIter())
print("RegParam:", best_svm_model.getRegParam())
```

```
Meilleurs paramètres pour SVM:
MaxIter: 30
RegParam: 0.01
```

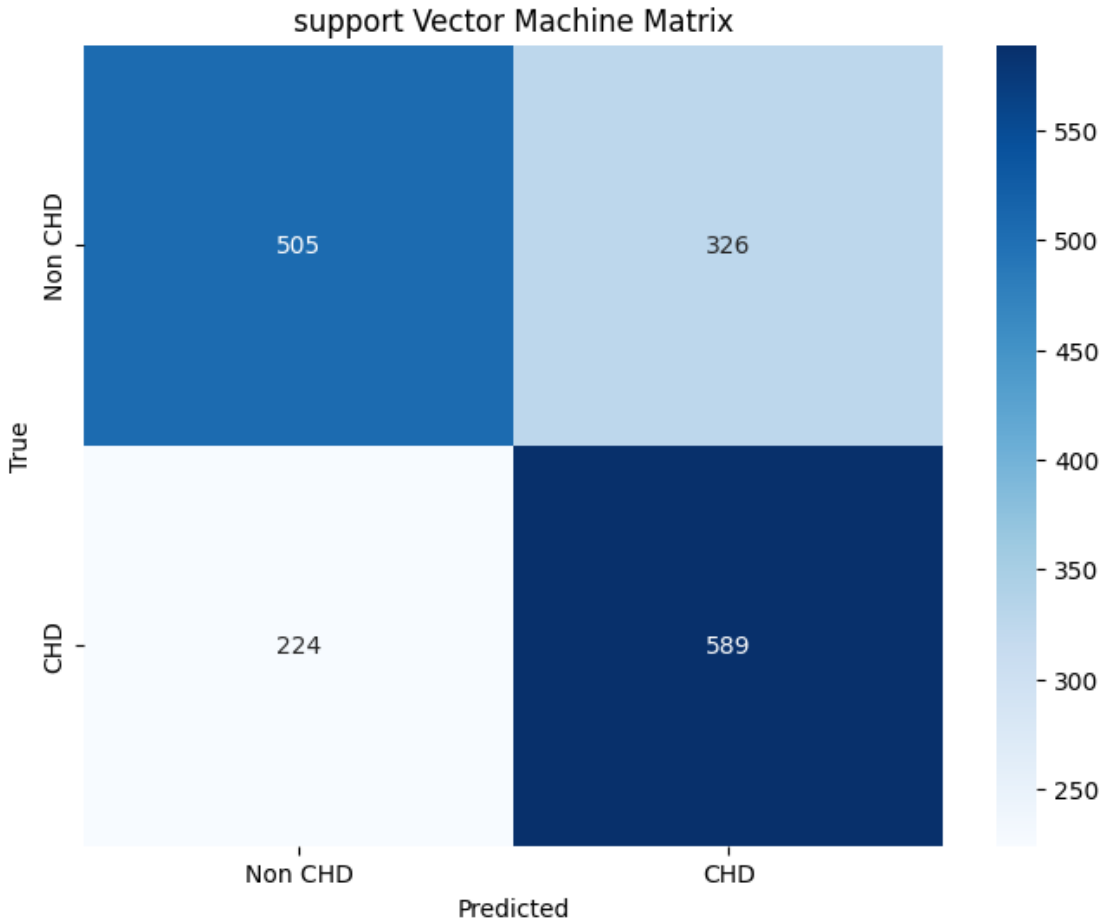
[49]:
```
# Convertir les prédictions de Spark DataFrame en Pandas DataFrame
svm_predictions_pd = best_svm_model.transform(testData).select("prediction",␣
 ↪"ten_year_chd").toPandas()
# Utiliser classification_report de scikit-learn sur les données Pandas avec␣
 ↪zero_division=0
svm_metrics = classification_report(svm_predictions_pd["ten_year_chd"],␣
 ↪svm_predictions_pd["prediction"], zero_division=0)
print("support vector machine Classification Report:")
print(svm_metrics)
```

```
support vector machine Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.61      0.65       831
           1       0.64      0.72      0.68       813

    accuracy                           0.67      1644
   macro avg       0.67      0.67      0.66      1644
weighted avg       0.67      0.67      0.66      1644
```

[50]:
```
# Calculer le recall pour l'ensemble de test
svm_test_recall = evaluator_recall.evaluate(best_svm_model.transform(testData))

# Calculer le recall pour l'ensemble d'entraînement
svm_train_recall = evaluator_recall.evaluate(best_svm_model.
 ↪transform(trainingData))

print("Logistic Regression Test Recall:", svm_test_recall)
print("Logistic Regression Train Recall:", svm_train_recall)
```

```
Logistic Regression Test Recall: 0.6654501216545012
Logistic Regression Train Recall: 0.6745260087506076
```

[51]:
```
svm_cm = best_svm_model.transform(testData).select("prediction",␣
 ↪"ten_year_chd").toPandas()
svm_cm = confusion_matrix(svm_cm["ten_year_chd"], svm_cm["prediction"])
plot_confusion_matrix(svm_cm, 'support Vector Machine Matrix')
```



Nous avons obtenu un taux de faux négatifs élevé par rapport au modèle de forets aléatoires, ce qui souligne une limitation significative du modèle de Support Vector Machine (SVM) dans la détection des cas réels de développement de maladie cardiaque parmi ceux qui sont prédits comme n'en ayant pas.

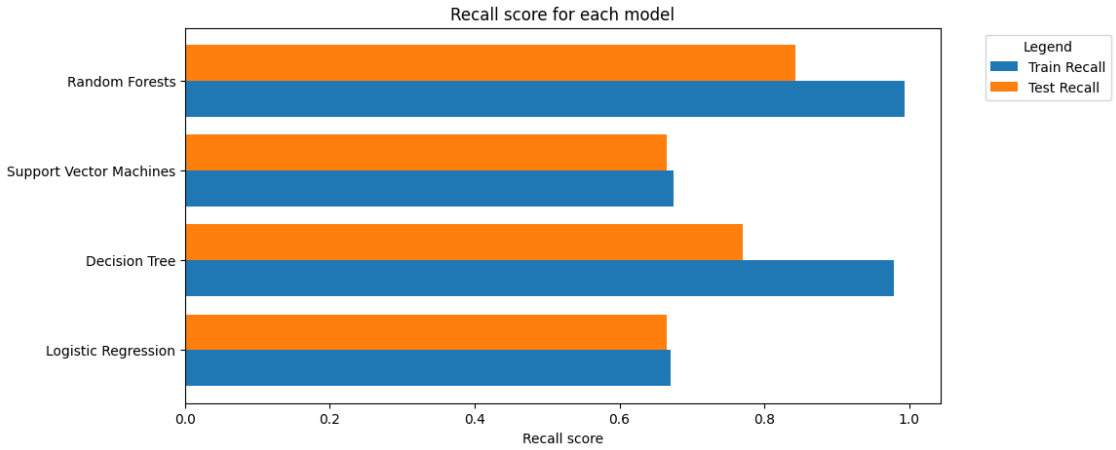
1.6 6. Resultats

[119]:
```
# Summarizing the results obtained
test = PrettyTable(['Sl. No.','Classification Model', 'Train Recall (%)','Test␣
 ↪Recall (%)'])
test.add_row(['1','Logistic␣
 ↪Regression',log_reg_train_recall*100,log_reg_test_recall*100])
test.add_row(['2','Decision Tree',dt_train_recall*100,dt_test_recall*100])
test.add_row(['3','Support Vector␣
 ↪Machines',svm_train_recall*100,svm_test_recall*100])
test.add_row(['4','Random Forests',rf_train_recall*100,rf_test_recall*100])
print(test)
```

```
+---------+-------------------------+--------------------+--------------------+
| Sl. No. |   Classification Model  |  Train Recall (%)  |  Test Recall (%)   |
+---------+-------------------------+--------------------+--------------------+
|    1    |   Logistic Regression   |  67.0393777345649  | 66.54501216545012  |
|    2    |      Decision Tree      | 97.95819154107924  | 77.06812652068126  |
|    3    | Support Vector Machines | 67.45260087506077  | 66.54501216545012  |
|    4    |      Random Forests     | 99.39231891103549  | 84.36739659367396  |
+---------+-------------------------+--------------------+--------------------+
```

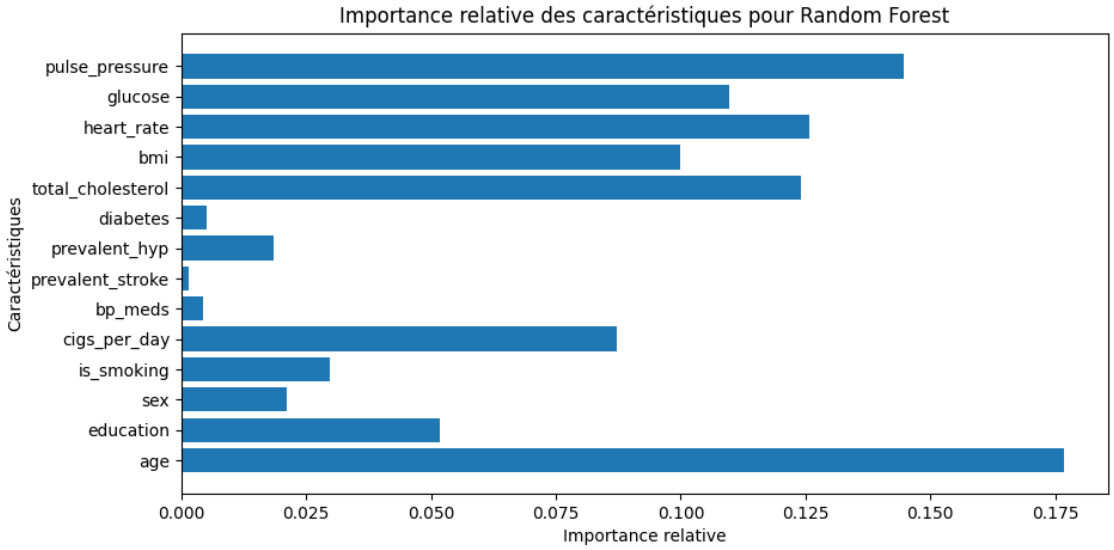
[67]:
```
# Plotting Recall scores

ML_models = ['Logistic Regression','Decision Tree','Support Vector␣
 ↪Machines','Random Forests']
train_recalls =␣
 ↪[log_reg_train_recall,dt_train_recall,svm_train_recall,rf_train_recall]
test_recalls =␣
 ↪[log_reg_test_recall,dt_test_recall,svm_test_recall,rf_test_recall]

X_axis = np.arange(len(ML_models))

plt.figure(figsize=(10,5))
plt.barh(X_axis - 0.2, train_recalls, 0.4, label = 'Train Recall')
plt.barh(X_axis + 0.2, test_recalls, 0.4, label = 'Test Recall')

plt.yticks(X_axis,ML_models)
plt.xlabel("Recall score")
plt.title("Recall score for each model")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left',title='Legend')
```

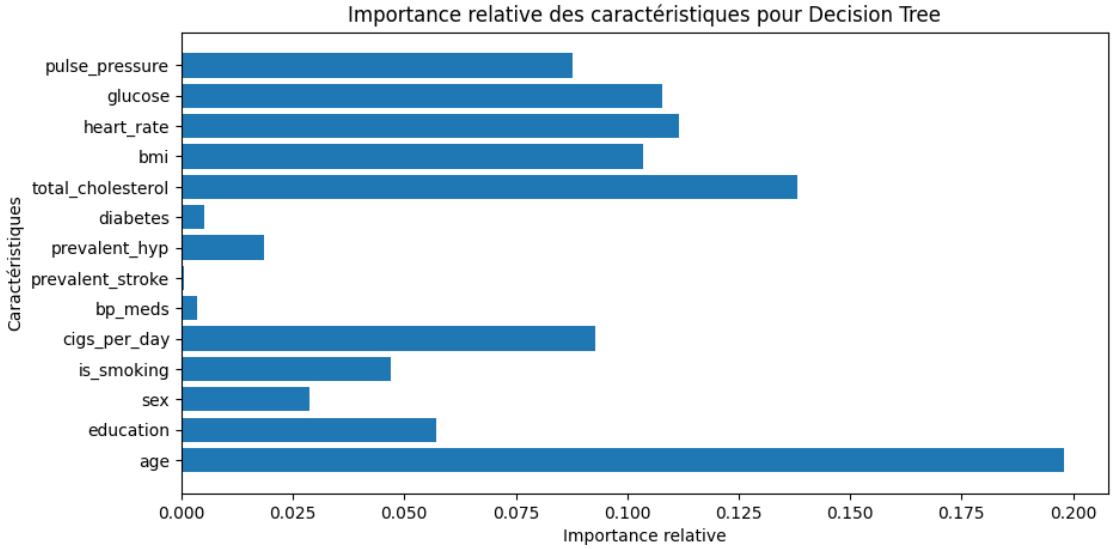
```
plt.show()
```



En conclusion, notre étude comparative des différents modèles pour la prédiction des maladies cardiaques a révélé des performances variables. Tout d'abord, nous avons constaté que les modèles d'arbre de décision et de forêt aléatoire ont montré des performances globalement meilleures que la régression logistique et le modèle de Support Vector Machine (SVM). En particulier, la forêt aléatoire a présenté les performances les plus élevées, avec une ROC AUC d'environ 0.934, ce qui indique une capacité exceptionnelle à discriminer entre les individus à risque de développer une maladie cardiaque et ceux qui ne le sont pas, tout en réduisant le taux de faux négatifs.

[70]:
```
import matplotlib.pyplot as plt

# Random Forest
rf_feature_importance = best_rf_model.featureImportances
plt.figure(figsize=(10, 5))
plt.barh(feature_columns, rf_feature_importance.toArray())
plt.xlabel('Importance relative')
plt.ylabel('Caractéristiques')
plt.title('Importance relative des caractéristiques pour Random Forest')
plt.show()

# Decision Tree
dt_feature_importance = best_dt_model.featureImportances
plt.figure(figsize=(10, 5))
plt.barh(feature_columns, dt_feature_importance.toArray())
plt.xlabel('Importance relative')
plt.ylabel('Caractéristiques')
plt.title('Importance relative des caractéristiques pour Decision Tree')
plt.show()
```





En analysant l'importance des caractéristiques dans les deux meilleurs modèles, nous constatons que l'âge est identifié comme la caractéristique la plus importante. Cette observation suggère que l'âge peut jouer un rôle significatif dans le risque de développer des maladies coronariennes.

1.7 7. Meuilleur modèle (Random Forest)

[54]:
```
# Convertir les prédictions Spark DataFrame en Pandas DataFrame
```

```
rf_predictions_pd = best_rf_model.transform(testData).select("age",␣
 ↪"education", "sex", "is_smoking", "cigs_per_day", "bp_meds",␣
 ↪"prevalent_stroke", "prevalent_hyp", "diabetes", "total_cholesterol", "bmi",␣
 ↪"heart_rate", "glucose", "pulse_pressure", "ten_year_chd","prediction").
 ↪toPandas()
# Enregistrer les prédictions dans des fichiers CSV
rf_predictions_pd.to_csv('random_forest_predictions.csv', index=False)
```

[55]:
```
rf_predictions_pd.head(20)
```

[55]:

	age	education	sex	is_smoking	cigs_per_day	bp_meds	prevalent_stroke	\
0	33	2	1	0	0	0	0	
1	34	1	0	1	10	0	0	
2	34	1	1	1	5	0	0	
3	34	1	1	1	10	0	0	
4	34	2	0	1	20	0	0	
5	34	2	0	1	20	0	0	
6	34	2	1	1	25	0	0	
7	34	3	1	1	20	0	0	
8	35	1	0	0	0	0	0	
9	35	1	1	0	0	0	0	
10	35	1	1	1	15	0	0	
11	35	2	0	0	0	0	0	
12	35	2	0	0	0	0	0	
13	35	2	0	0	0	0	0	
14	35	2	0	1	5	0	0	
15	35	2	0	1	20	0	0	
16	35	2	0	1	20	0	0	
17	35	2	1	0	0	0	0	
18	35	2	1	1	20	0	0	
19	35	2	1	1	20	0	0	

	prevalent_hyp	diabetes	total_cholesterol	bmi	heart_rate	\
0	1	0	165.000000	26.740000	54.000000	
1	0	0	185.000000	19.680000	75.000000	
2	0	0	185.000000	24.420000	70.000000	
3	0	0	210.000000	24.390000	68.000000	
4	0	0	180.000000	21.510000	91.000000	
5	0	0	220.000000	20.790000	63.000000	
6	0	0	250.000000	29.040000	63.000000	
7	0	0	155.000000	23.510000	85.000000	
8	0	0	170.000000	23.480000	75.000000	
9	0	0	275.000000	34.040000	75.000000	
10	0	0	210.000000	22.390000	57.000000	
11	0	0	208.000000	22.000000	65.000000	
12	0	0	216.000000	25.940000	75.000000	
13	0	0	248.000000	20.640000	90.000000	

14	0	0	165.000000	19.140000	68.000000
15	0	0	167.904007	16.731694	78.923206
16	0	0	168.000000	16.710000	79.000000
17	1	0	245.000000	26.230000	110.000000
18	0	0	223.000000	19.990000	80.000000
19	0	0	227.000000	29.270000	70.000000

	glucose	pulse_pressure	ten_year_chd	prediction
0	77.000000	46.500000	0	0.0
1	79.700000	22.500000	0	0.0
2	115.000000	34.500000	0	0.0
3	70.700000	35.000000	0	0.0
4	78.000000	55.000000	0	0.0
5	86.000000	50.000000	0	0.0
6	80.000000	42.000000	0	0.0
7	65.000000	45.000000	0	0.0
8	83.000000	41.000000	0	0.0
9	80.000000	53.500000	0	0.0
10	75.000000	32.000000	0	0.0
11	81.900000	50.000000	0	0.0
12	90.000000	62.000000	0	0.0
13	80.000000	34.000000	0	0.0
14	70.000000	42.000000	0	0.0
15	63.153589	28.711185	1	1.0
16	63.000000	28.500000	1	1.0
17	82.800000	63.500000	0	0.0
18	67.000000	46.000000	0	0.0
19	79.000000	33.000000	0	0.0