

Atelier 1 : Classification

Pr. Ali Idri et Fatima Zahrae Nakach, Ph.D. student, UM6P

Master of Science in Quantitative and Financial Modeling

2022-2023

L'objectif de cet atelier est de comprendre les étapes nécessaires pour effectuer une analyse de classification binaire en utilisant plusieurs algorithmes de machine learning sur la dataset Pima Indians et de déterminer le modèle le plus efficace pour prédire le diabète.

Les étapes de cet atelier sont les suivantes:

1. **Préparation de la dataset :** charger la dataset Pima Indians, nettoyer les données et manipuler les données manquantes, normaliser les données et vérifier la balance des classes.
2. **Équilibrage de la dataset :** utiliser des techniques d'équilibrage de la dataset telles que pour équilibrer les données et réduire le biais dans les résultats.
3. **Sélection de caractéristiques (features) :** choisir les variables les plus pertinentes pour la prédiction de la variable cible dans le modèle de classification en éliminant les variables redondantes ou peu informatives.
4. **Train-Test-Split :** Diviser la dataset en données d'entraînement et de test pour évaluer les modèles de classification sur des données jamais vues auparavant.
5. **Classification binaire :** utiliser des algorithmes de classification tels que KNN, arbre de décision et SVM pour prédire si un patient a ou non un diabète.
6. **Évaluation de la performance des modèles :** utiliser des métriques telles que l'accuracy, la matrice de confusion et le rapport de classification pour évaluer la performance de chaque algorithme de classification.
7. **Cross-validation :** Utiliser la cross-validation pour évaluer la performance des modèles de classification.
8. **Balayage de grille :** utiliser la technique de balayage de grille pour trouver les meilleurs hyperparamètres pour chaque algorithme de classification.
9. **Comparaison des modèles :** comparer les performances de chaque algorithme de classification pour déterminer le modèle le plus efficace pour prédire le diabète chez les patients Pima Indians.
10. **Visualisation des résultats :** utiliser des outils de visualisation tels que Matplotlib et Seaborn pour visualiser les résultats des différents modèles et comprendre les tendances et les relations entre les variables.

N.B : Les exemples du code source présentés dans ce TP sont des propositions. pour chaque partie, vous serez amenés à rendre vos propres implémentations et interprétations.

Introduction

1. Dataset

La dataset Pima Indians est une base de données médicales comprenant des données sur des patients indiens Pima. Il contient des informations sur 8 variables numériques pour chacun des 768 patients :

- Nombre de grossesses : Nombre de fois où une femme a été enceinte
- Glucose : Taux de glucose plasmatique à 2 heures dans un test de tolérance au glucose
- Pression artérielle : pression artérielle diastolique (mm Hg)
- Épaisseur du pli cutané : épaisseur du pli cutané du triceps (mm)
- Insuline : niveau d'insuline sérique à 2 heures (mu U/ml)
- Indice de masse corporelle : indice de masse corporelle (poids en kg/(taille en m)²)
- Diabète de parent proche de fonction : Histoire familiale de diabète
- Âge : âge en années

La variable cible est si le patient a ou non un diabète, représenté par un 1 ou un 0 respectivement.

2- Google Colab



L'atelier se déroulera sur Google Colab, une plateforme en ligne qui fournit des environnements de développement interactifs pour les algorithmes de machine learning.

Lien vers le notebook :

<https://colab.research.google.com/drive/1nTKw1Wea2uPiTTZwaJELCd4nPQbZskIJ?>

Avec Colab, vous pouvez utiliser des bibliothèques Python populaires telles que TensorFlow, PyTorch et Keras, ainsi que des outils de visualisation tels que Matplotlib et Seaborn. Il dispose également d'une intégration GPU et TPU gratuite pour accélérer les calculs intensifs en ressources.

L'un des avantages de Colab est qu'il n'est pas nécessaire d'avoir une puissance de calcul locale élevée pour utiliser des algorithmes de machine learning complexes, car tout se passe dans le cloud de Google. Il est également facile à utiliser pour les débutants en raison de sa simplicité d'interface et de son intégration directe avec Google Drive, ce qui permet de stocker et de partager facilement des fichiers et des projets.

Partie 1 : Préparation de la dataset

1) Importation des librairies nécessaires

La première étape consiste à importer les librairies nécessaires pour l'atelier, principalement:

- 1- NumPy est le package fondamental pour le calcul scientifique en Python. NumPy est une bibliothèque de calcul scientifique dédié à la manipulation de matrices et de tableaux en multiple dimensions.
- 2- Pandas est un package couramment utilisé pour traiter l'analyse des données. Il simplifie le chargement de données provenant de sources externes telles que des fichiers texte et des bases de données, et fournit des moyens d'analyser et de manipuler les données.
- 3- Matplotlib est un package Python utilisé pour le plot et la visualisation de données. C'est un complément utile à Pandas, et comme Pandas, c'est une bibliothèque très riche en fonctionnalités qui peut produire une grande variété de plots, de graphiques, de cartes et d'autres visualisations.
- 4- Scikit-learn (Sklearn) est la bibliothèque la plus utile et la plus robuste pour l'apprentissage automatique (Machine learning) en Python. Elle fournit une sélection d'outils efficaces pour l'apprentissage automatique et la modélisation statistique, notamment la classification, la régression, le clustering.., via une interface cohérente en Python. Cette bibliothèque s'appuie sur NumPy, SciPy et Matplotlib.

```
#Import libraries
import pandas as pd
from sklearn.feature_selection import SelectFromModel
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn import tree
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest, mutual_info_classif, chi2, f_classif
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score,
recall_score, classification_report
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV
from sklearn.tree import export_graphviz
import graphviz
from sklearn.model_selection import LeaveOneOut
```

2) Chargement de la dataset Pima Indians

- Importation d'un fichier CSV :

L'une des fonctions très importante et mature de la lirairie Pandas est `read_csv()` qui permet de lire et manipuler n'importe quel fichier .csv. Faisons-le sur notre fichier diabetes.csv.

```
# Importer un fichier CSV
DataCSV= pd.read_csv("diabetes.csv")
DataCSV
```

Supposons que le fichier csv ne contient pas de colonnes:

```
#Ajouter des headers:
DataCSV= pd.read_csv("diabetes.csv", header = None, names = ['preg', 'plas', 'pres', 'skin',
'test', 'mass', 'pedi', 'age', 'class'])
DataCSV = DataCSV.drop( columns= 'Id', index= 0, axis=1)
DataCSV
```

- Importation d'un lien URL

```
url = https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
feature_names = names[:8]
data = pd.read_csv(url, names=names)
```

3) Analyse exploratoire des données

Le but de cette partie est de se familiariser avec l'analyse de données et d'interpréter les différents résultats.

a-Afficher la base de données chargées

	data									
	preg	plas	pres	skin	test	mass	pedi	age	class	
0	6	148	72	35	0	33.6	0.627	50	1	
1	1	85	66	29	0	26.6	0.351	31	0	
2	8	183	64	0	0	23.3	0.672	32	1	
3	1	89	66	23	94	28.1	0.167	21	0	
4	0	137	40	35	168	43.1	2.288	33	1	
...	
763	10	101	76	48	180	32.9	0.171	63	0	
764	2	122	70	27	0	36.8	0.340	27	0	

Figure 1

b-Afficher les cinq premières lignes de la base de données

```
data[:5]
```

	preg	plas	pres	skin	test	mass	pedi	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Figure 2

c-Afficher le nombre d'instances pour chaque classe

```
def show_num_inst(data):
    # Compter le nombre d'instances pour chaque classe
    class_counts = data['class'].value_counts()

    # Définir les étiquettes et les valeurs pour le graph
    labels = class_counts.index
    values = class_counts.values

    # Configurer et afficher le graph
    plt.bar(labels, values)
    plt.xlabel('Class')
    plt.ylabel('Count')
    plt.title('Number of Instances per Class')
    plt.show()
```

```
[ ] show_num_inst(data)
```

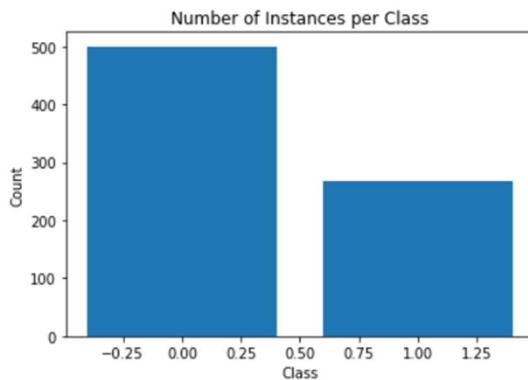


Figure 3

d-Définir les attributs (variables prédictives) et la variable cible

```
#Splitting the data into dependent and independent variables
```

```
X = data.drop("class", axis=1)
```

```
Y = data["class"]
```

Question 1 : Afficher les attributs et la variable cible

e-Que donne le code suivant ? Que peut-on remarquer de la figure 4?

```
def plot_input(X,Y):
    # Convert the data from numpy array to a pandas dataframe
    df_ros = pd.DataFrame({'Age': X["age"], 'Blood_Pressure': X["pres"], 'target': Y})
    # Plot the chart
    plt.figure(figsize=(12, 8))
    sns.scatterplot(x = 'Age', y = 'Blood_Pressure', hue = 'target', data = df_ros)
    plt.title('Random Over Sampling')
```

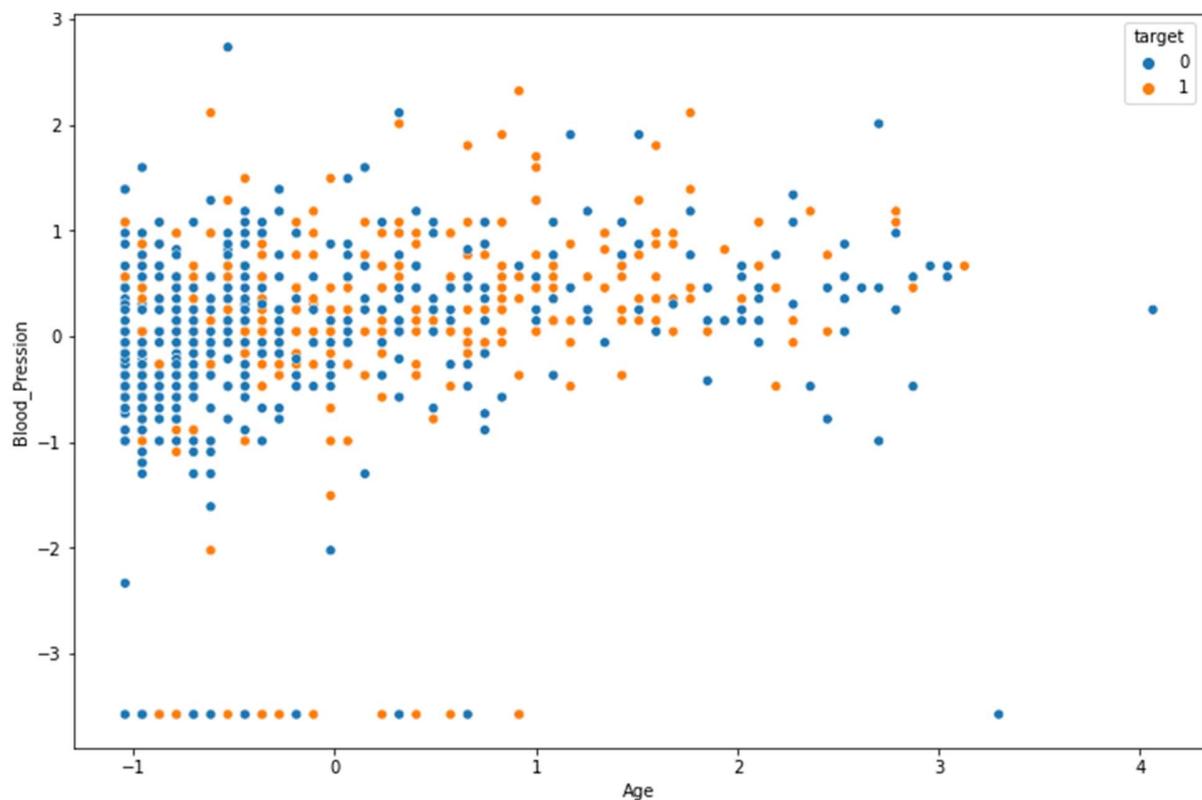


Figure 4

Question 2 : Afficher le graphe de la Figure 4 en utilisant des attributs différents

f-Quelles informations nous donne le box plot de la figure 5?

```
#histogramms
```

```
data.hist(figsize=(15,10))
plt.suptitle("histogram", fontsize=16)
plt.show()
#boxplot
data.boxplot(figsize=(8,4))
plt.title("Bar plot", fontsize=16)
plt.show()
```

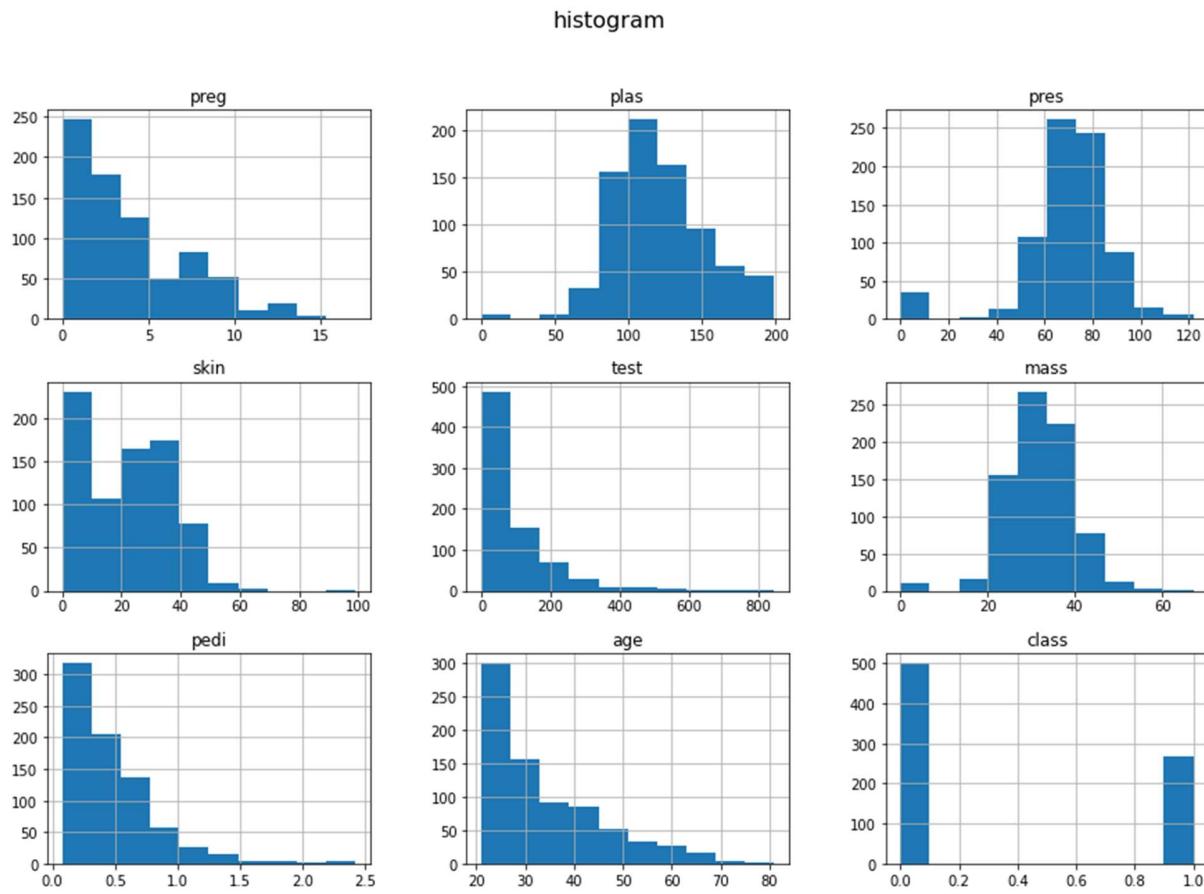


Figure 5

g-Visualiser la distribution des classes dans la base de données

```
data.groupby(by = "class").mean()
data.groupby(by="class").mean().plot(kind="bar", figsize=(15,10))
plt.title('Class vs measurements')
plt.ylabel('mean measurement (cm)')
plt.grid(True)
plt.legend(loc="upper left", bbox_to_anchor=(1,1))
```

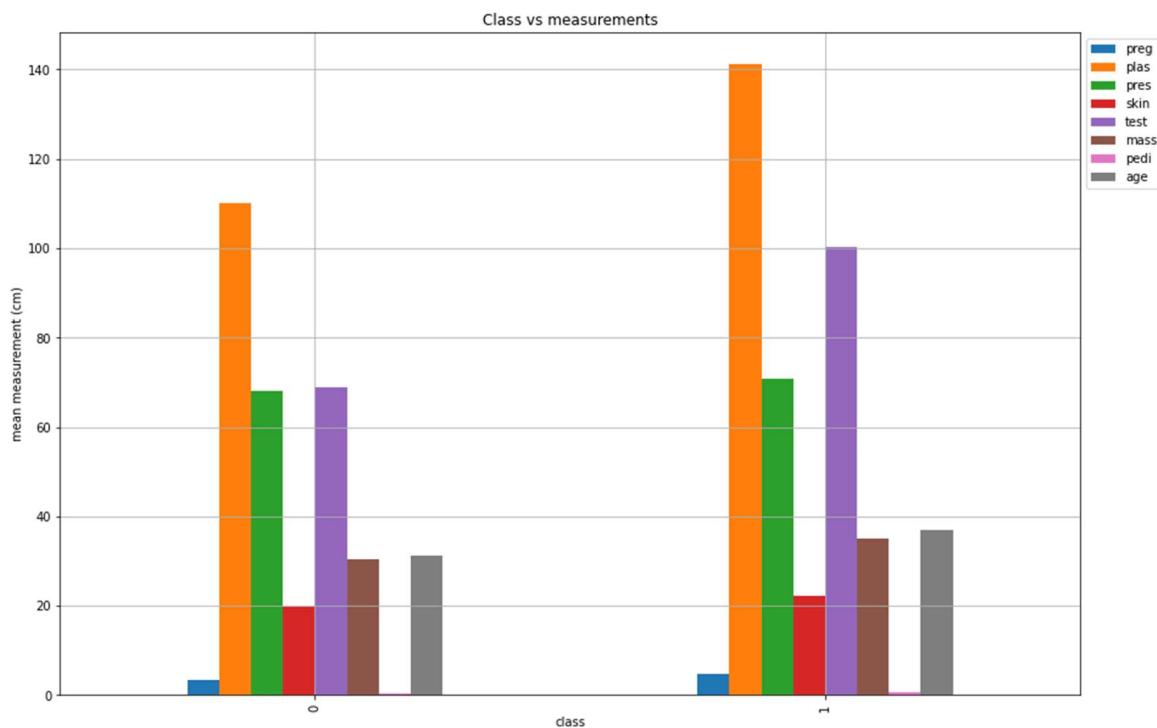


Figure 6

4) Prétraitement des données

-Normalisation de données

```
scaler = MinMaxScaler(feature_range=(0, 1))
X = scaler.fit_transform(X)
# summarize transformed data
np.set_printoptions(precision=3)
#check transformations
print(X[0:5,:])

#Save new dataset
df = pd.DataFrame(X, columns = feature_names)
X = pd.DataFrame(X, columns = feature_names)
dff["class"] = Y
```

Question 3 : Afficher le graphe de la Figure 5 et 6 après la normalisation, comparer.

5) Equilibrage de la dataset

-Méthode 1 : Under Sampling

```
from imblearn.under_sampling import RandomUnderSampler
y = dff["class"]
rus = RandomUnderSampler()
X_train, y_train = rus.fit_resample(X, Y)
```

-Méthode 2 : Over Sampling

```
from imblearn.over_sampling import RandomOverSampler
rus = RandomOverSampler()
X_test, y_test = rus.fit_resample(X, Y)
data2 = pd.DataFrame(X_test)
data2["class"] = y_test
```

-Méthode 3 : SMOTE Sampling

```
from imblearn.over_sampling import SMOTE
oversample = SMOTE(k_neighbors=3)
X3, y3 = oversample.fit_resample(X, Y)
data3 = pd.DataFrame(X3)
data3["class"] = y3
```

Question 4 : Afficher le graphe de la Figure 3 pour voir le nombre d'instances de chaque classe pour chaque méthode

-On affiche la figure 4 après avoir appliqué la méthode 2 et 3, même si le nombre d'instances est le même, on remarque plus de points sur la Figure8, Pourquoi ?

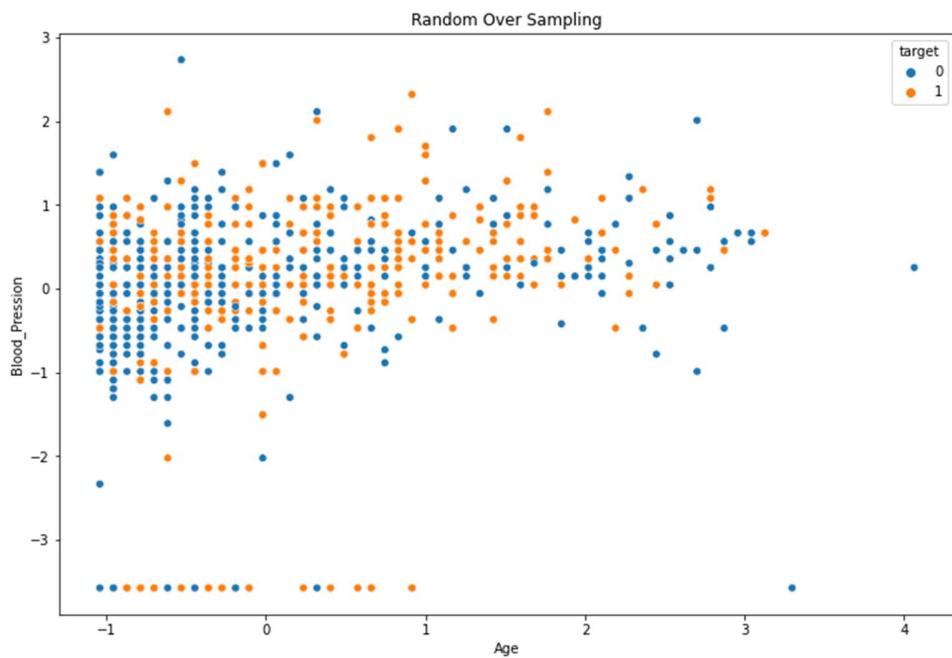


Figure 7

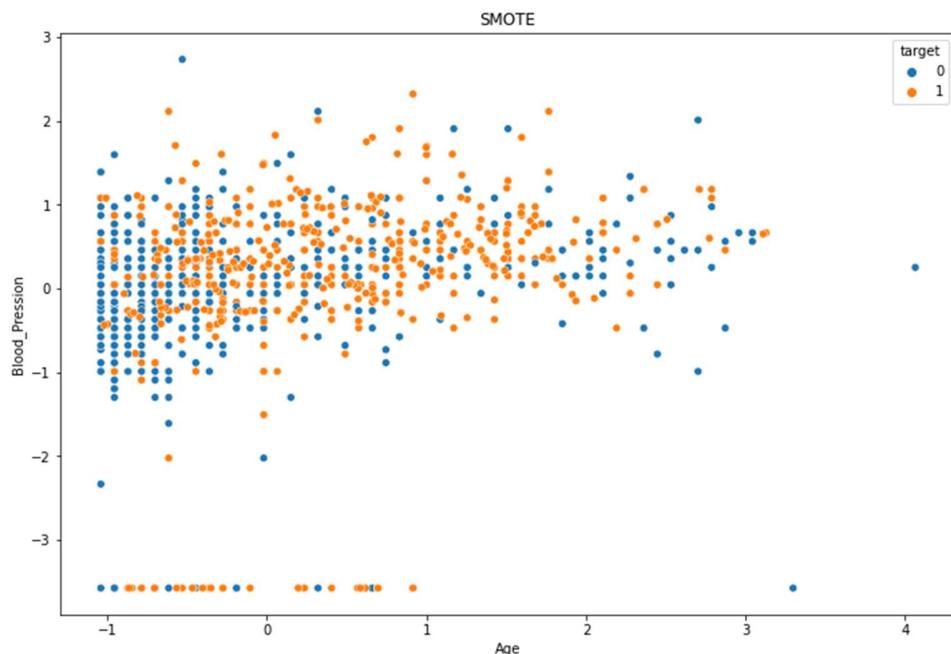


Figure 8

Partie 2 : Sélection de caractéristiques (features)

Lors de la construction d'un modèle d'apprentissage automatique, il est presque rare que toutes les variables de l'ensemble de données soient utiles pour construire le modèle. L'ajout de variables redondantes réduit la capacité de généralisation du modèle et peut également réduire la précision globale d'un classificateur. De plus, ajouter de plus en plus de variables à un modèle augmente la complexité globale du modèle. Ainsi, la sélection des caractéristiques « feature selection » devient une partie indispensable de la construction de modèles d'apprentissage automatique.

- ➔ L'objectif de la sélection de caractéristiques dans l'apprentissage automatique est de trouver le meilleur ensemble de caractéristiques qui permet de construire le modèle.

Les techniques de sélection de caractéristiques dans l'apprentissage automatique peuvent être classées dans les catégories suivantes :

1- Les filtres :

Les méthodes de filtrage prennent en compte les propriétés intrinsèques des caractéristiques mesurées par des statistiques univariées ou multivariées. Ces méthodes sont plus rapides et moins coûteuses en termes de calcul que les méthodes d'encapsulation (wrappers). Lorsque l'on traite des données à haute dimension, il est plus économique d'utiliser des méthodes de filtrage.

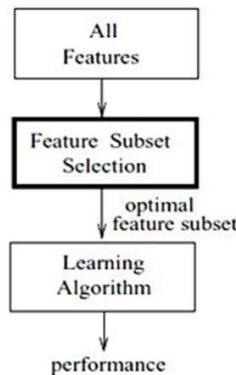


Figure 9

1.1 Information Gain

Consiste à calculer la réduction de l'entropie résultant de la transformation d'un ensemble de données. Il peut être utilisé pour la sélection des caractéristiques en évaluant le gain d'information de chaque variable dans le contexte de la variable cible.

```
from sklearn.feature_selection import mutual_info_classif
```

1.2 Chi-square Test

Utilisée pour les caractéristiques catégorielles d'un ensemble de données. Nous calculons le Chi-square entre chaque caractéristique et la cible et sélectionnons le nombre souhaité de caractéristiques ayant les meilleurs scores. Pour appliquer correctement le chi-deux afin de tester la relation entre diverses caractéristiques de l'ensemble de données et la variable cible, les conditions suivantes doivent être remplies : les variables doivent être catégoriques, échantillonées indépendamment et les valeurs doivent avoir une fréquence attendue supérieure à 5.

```
from sklearn.feature_selection import chi2
```

1.3 Variance threshold

Le seuil de variance est une approche de base simple pour la sélection des caractéristiques. Il élimine toutes les caractéristiques dont la variance n'atteint pas un certain seuil. Par défaut, il élimine toutes les caractéristiques à variance nulle, c'est-à-dire les caractéristiques qui ont la même valeur dans tous les échantillons. Nous supposons que les caractéristiques ayant une variance plus élevée peuvent contenir plus d'informations utiles.

```
from sklearn.feature_selection import VarianceThreshold
```

1.4. Méthodes disponibles sur SKLEARN :

Vous pouvez voir toutes les méthodes disponibles sur `sklearn.feature_selection`, la librairie `sklearn.feature_selection` comprend actuellement des méthodes de sélection par filtre univarié.

<code>feature_selection.GenericUnivariateSelect([...])</code>	Univariate feature selector with configurable strategy.
<code>feature_selection.SelectPercentile([...])</code>	Select features according to a percentile of the highest scores.
<code>feature_selection.SelectKBest([score_func, k])</code>	Select features according to the k highest scores.
<code>feature_selection.SelectFpr([score_func, alpha])</code>	Filter: Select the pvalues below alpha based on a FPR test.
<code>feature_selection.SelectFdr([score_func, alpha])</code>	Filter: Select the p-values for an estimated false discovery rate.
<code>feature_selection.SelectFromModel(estimator, *)</code>	Meta-transformer for selecting features based on importance weights.
<code>feature_selection.SelectFwe([score_func, alpha])</code>	Filter: Select the p-values corresponding to Family-wise error rate.
<code>feature_selection.SequentialFeatureSelector(...)</code>	Transformer that performs Sequential Feature Selection.
<code>feature_selection.RFE(estimator, *[...])</code>	Feature ranking with recursive feature elimination.
<code>feature_selection.RFECV(estimator, *[...])</code>	Recursive feature elimination with cross-validation to select the number of features.
<code>feature_selection.VarianceThreshold([threshold])</code>	Feature selector that removes all low-variance features.
<	
<code>feature_selection.chi2(X, y)</code>	Compute chi-squared stats between each non-negative feature and class.
<code>feature_selection.f_classif(X, y)</code>	Compute the ANOVA F-value for the provided sample.
<code>feature_selection.f_regression(X, y, *[center])</code>	Univariate linear regression tests returning F-statistic and p-values.
<code>feature_selection.r_regression(X, y, *[center])</code>	Compute Pearson's r for each features and the target.
<code>feature_selection.mutual_info_classif(X, y, *)</code>	Estimate mutual information for a discrete target variable.
<code>feature_selection.mutual_info_regression(X, y, *)</code>	Estimate mutual information for a continuous target variable.
<	

Figure 10

1.5. Application Information gain

Nous pouvons utiliser Scikit-learn pour calculer le gain d'information. Scikit-learn dispose de deux fonctions pour le calculer:

- `mutual_info_classif` (pour la classification)
- `mutual_info_regression` (pour la regression)

```
MI_score = mutual_info_classif(X, y, random_state=0)
for feature in zip(feature_names, MI_score):
    print(feature)
```

D'après le résultat obtenu quelle sont les features les plus importatnt pour la dataset Pima Indians ?

Visualisez les résultats :

```
plt.figure(figsize=(4,4))
plt.bar(x=feature_names, height=MI_score, color='blue')
plt.xticks(rotation='vertical')
plt.ylabel('Mutual Information Score')
plt.title('Mutual Information Score Comparison')
plt.show()
```

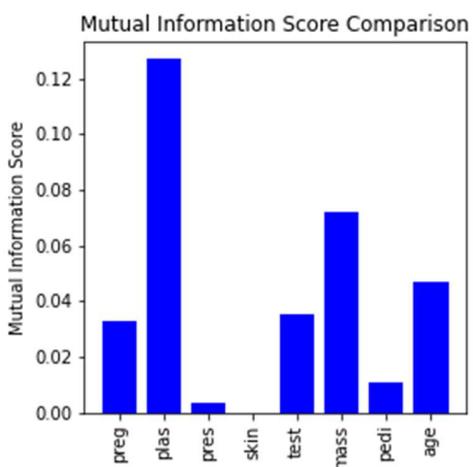


Figure 11

1.6. Application de la matrice de corrélation

```
#Using Pearson Correlation
plt.figure(figsize=(12,10))
cor = df.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.show()
```

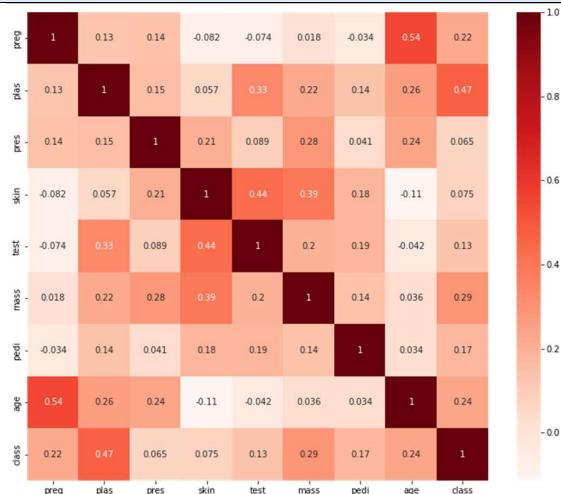


Figure 12

```
#Correlation with output variable
cor_target = abs(cor["class"])
#Selecting highly correlated features
relevant_features = cor_target[cor_target>0.25]
relevant_features
```

-On vérifie si les attributs corrélés avec la variable cible ne sont pas corrélés entre eux

```
[ ] print(df[["plas","mass"]].corr())
```

	plas	mass
plas	1.000000	0.221071
mass	0.221071	1.000000

Figure 13

1.7 Application de la méthode SelectKbest

Exécuter le code et comparer les résultats ce cette méthode avec les résultats des autres filtres.

```
# Use SelectKBest to select the best features based on univariate test results
selector = SelectKBest(k=2)
X_new = selector.fit_transform(X, y)
print(X_new.shape)
print(selector.scores_)
print(selector.feature_names_in_)
```

Question 5 : SelectKBest pour un autre K, montrer que le résultat obtenu est différent des résultats avant pour K = 2

2. Wrappers

Les wrappers nécessitent une méthode d'encapsulation pour rechercher l'espace de tous les sous-ensembles possibles de caractéristiques, en évaluant leur qualité par l'apprentissage et l'évaluation d'un classificateur avec ce sous-ensemble de caractéristiques. Le processus de sélection des caractéristiques est basé sur un algorithme d'apprentissage automatique spécifique que nous essayons d'adapter à un ensemble de données donné. Il suit une approche de recherche gloutonne en évaluant toutes les combinaisons possibles de caractéristiques par rapport au critère d'évaluation. Les méthodes wrapper donnent généralement une meilleure précision prédictive que les méthodes de filtrage.

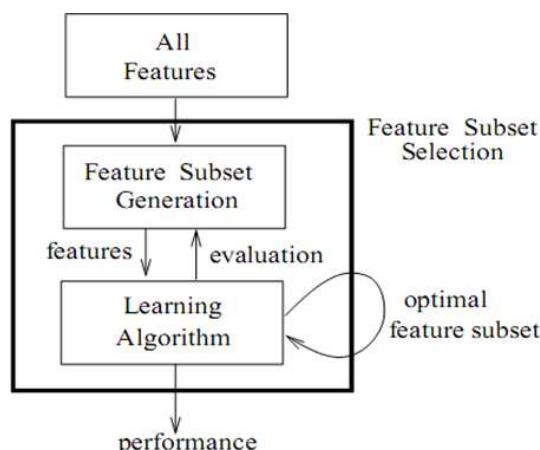


Figure 14

2.1. Forward feature selection

Il s'agit d'une méthode itérative dans laquelle nous commençons par la variable la plus performante par rapport à l'objectif. Ensuite, nous sélectionnons une autre variable qui donne la meilleure performance en combinaison avec la première variable sélectionnée. Ce processus se poursuit jusqu'à ce que le critère prédéfini soit atteint.

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

Pour importer de **mlxtend.feature_selection** il faut installez le package MLxtend.

```
!pip install mlxtend
import joblib
import sys
sys.modules['sklearn.externals.joblib'] = joblib
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
```

On applique SequentialFeatureSelector à notre dataset, l'algorithme d'apprentissage pour cet exemple est RandomForest :

```
# Initialize the Random Forest Classifier
clf = RandomForestClassifier()

# Initialize the Sequential Feature Selector
sfs = SFS(clf,
           k_features=4,
           forward=True,
           floating=False,
           scoring='accuracy',
           cv=5)

# Fit the SFS to the data
sfs = sfs.fit(X, Y, custom_feature_names=feature_names)

# Get the selected features
selected_features = list(sfs.k_feature_idx_)
print("Selected features:", selected_features)
```

```

1: {'feature_idx': (1,),
 'cv_scores': array([0.68831169, 0.68181818, 0.70779221, 0.7254902 , 0.73856209]),
 'avg_score': 0.7083948731007554,
 'feature_names': ('plas',)},
2: {'feature_idx': (1, 5),
 'cv_scores': array([0.71428571, 0.7012987 , 0.73376623, 0.74509804, 0.71895425]),
 'avg_score': 0.7226805873864698,
 'feature_names': ('plas', 'mass')},
3: {'feature_idx': (0, 1, 5),
 'cv_scores': array([0.75324675, 0.69480519, 0.75974026, 0.76470588, 0.73202614]),
 'avg_score': 0.7409048467871997,
 'feature_names': ('preg', 'plas', 'mass')},
4: {'feature_idx': (0, 1, 5, 7),
 'cv_scores': array([0.74675325, 0.74675325, 0.78571429, 0.85620915, 0.73202614]),
 'avg_score': 0.7734912146676852,
 'feature_names': ('preg', 'plas', 'mass', 'age')}}


```

Figure 15

Question 6 : Appliquer SFS pour un nombre différent de features et comparer.

2.2. Exhaustive feature selection

Il s'agit de la méthode de sélection des caractéristiques la plus robuste couverte jusqu'à présent. Il s'agit d'une évaluation par force brute de chaque sous-ensemble de caractéristiques. Cela signifie qu'elle essaie toutes les combinaisons possibles de variables et renvoie le sous-ensemble le plus performant.

```
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
```

Exécutez le code ci-dessous :

```

knn = KNeighborsClassifier()
efs = EFS(estimator=knn, # The Ml model
          min_features=1,
          max_features=6,
          scoring='accuracy', # The metric to use to evaluate the classifier is accuracy
          cv=5)
efs = efs.fit(X, Y)
print('Best accuracy score: %.2f % efs.best_score_) # best_score_ shows the best score
print('Best subset (corresponding names):', efs.best_feature_names_)
# best_feature_names_ shows the feature names that yield the best score

```

Features: 255/255Best accuracy score: 0.77

Best subset (corresponding names): ('plas', 'pres', 'skin', 'mass', 'age')

Figure 16

-On sauvegarde les nouveaux attributs sélectionnés dans la variable X_new, et on affiche le nombre d'attribut avant et après l'application de EFS.

```
# Transform the dataset
X_new = efs.transform(X)
# Print the results
print('Number of features before transformation: {}'.format(X.shape[1]))
print('Number of features after transformation: {}'.format(X_new.shape[1]))
```

Number of features before transformation: 8
 Number of features after transformation: 5

Figure 17

Visualisation des résultats:

```
# Show the performance of each subset of features
efs_results = pd.DataFrame.from_dict(efs.get_metric_dict()).T
efs_results.sort_values(by='avg_score', ascending=True, inplace=True)
efs_results
```

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err	
4	(4,)	[0.6883116883116883, 0.5454545454545454, 0.545...	0.555844		(test,)	0.089088	0.069314	0.034657
2	(2,)	[0.5974025974025974, 0.5714285714285714, 0.545...	0.561158		(pres,)	0.027762	0.0216	0.0108
3	(3,)	[0.512987012987013, 0.551948051948052, 0.61038...	0.584738		(skin,)	0.057522	0.044754	0.022377
6	(6,)	[0.564935064935065, 0.6298701298701299, 0.5389...	0.593812		(pedi,)	0.057159	0.044472	0.022236
24	(2, 6)	[0.5974025974025974, 0.6168831168831169, 0.558...	0.597683		(pres, pedi)	0.028491	0.022167	0.011083
...	
240	(1, 2, 3, 4, 5, 7)	[0.7272727272727273, 0.7337662337662337, 0.759...	0.751345	(plas, pres, skin, test, mass, age)	0.022508	0.017512	0.008756	
106	(0, 1, 6, 7)	[0.7532467532467533, 0.7207792207792207, 0.753...	0.752644	(preg, plas, pedi, age)	0.032598	0.025362	0.012681	
222	(0, 1, 2, 3, 5, 7)	[0.7922077922077922, 0.7207792207792207, 0.759...	0.753892	(preg, plas, pres, skin, mass, age)	0.038084	0.029631	0.014815	
243	(1, 2, 4, 5, 6, 7)	[0.7532467532467533, 0.7012987012987013, 0.746...	0.7566	(plas, pres, test, mass, pedi, age)	0.053268	0.041445	0.020722	
201	(1, 2, 3, 5, 7)	[0.7597402597402597, 0.7142857142857143, 0.785...	0.765674	(plas, pres, skin, mass, age)	0.036869	0.028685	0.014343	

246 rows × 7 columns

Figure 18

Question 7: Appliquer EFS pour un nombre différent de features, comprarez.

Figure 16 indique que l'accuracy de KNN avec la meilleure sélection de features est de 77%, ce résultat reflète-t-il vraiment la performance de KNN sur la dataset Pima Indians ?

Partie 3 : Classification

-On divise la base de données en deux sous-ensembles (méthode du Hold Out), le premier dit d'apprentissage et le second dit de validation ou de test. Le modèle est construit sur l'échantillon d'apprentissage et validé sur l'échantillon de test avec un score de performance de notre choix. Ce fractionnement peut être effectué à l'aide de l'utilitaire `train_test_split` de Scikit-Learn

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
```

1. Construire le modèle K-nearest neighbors

a- Entraînement de KNN

KNN (K-Nearest Neighbors) est un algorithme de classification et de régression supervisée utilisé pour prédire la classe ou la valeur d'une nouvelle observation en se basant sur les caractéristiques des observations similaires dans l'ensemble de données d'entraînement. KNN fonctionne en calculant la distance entre la nouvelle observation et chaque observation d'entraînement, puis en sélectionnant les K voisins les plus proches (les plus similaires) en fonction de cette distance. La classe ou la valeur prédictive est ensuite déterminée en prenant en compte les classes ou les valeurs des K voisins les plus proches.

On entraîne notre modèle sur `X_train`, `Y_train` et on le teste sur `X_test`. Les classes prédictives `y_pred` sont par la suite comparées avec les classes réelles `Y_test` pour calculer l'accuracy.

```
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
acc = metrics.accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
```

Exécutez ce code plusieurs fois, est ce que vous obtenez le même résultat ?Pourquoi ?

Question 8 : Appliquer KNN avec un n_neighbors différent

b- Cross-validation (Validation croisée)

Une façon de résoudre ce problème est d'utiliser la Cross-validation, c'est-à-dire d'effectuer une séquence d'ajustements où chaque sous-ensemble de données est utilisé à la fois comme ensemble d'apprentissage et comme ensemble de validation (Figure 19).

Il en résulte deux scores d'accuracy que nous pouvons combiner (en prenant la moyenne, par exemple) pour obtenir une meilleure mesure de la performance globale du modèle. Cette forme particulière de cross-validation est une 2-folds-cross-validation, c'est-à-dire une validation dans laquelle nous avons divisé les données en deux ensembles et utilisé chacun d'eux alternativement comme ensemble de validation et d'apprentissage.

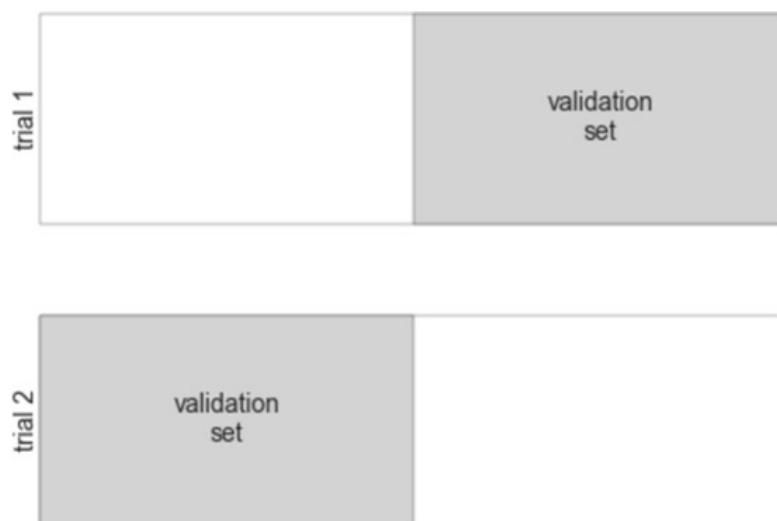


Figure 19

```
#2 fold
model=KNeighborsClassifier(n_neighbors=3)
y_test_model = model.fit(X_train,y_train).predict(X_test)
y_train_model=model.fit(X_test,y_test).predict(X_train)
accuracy_score(y_train,y_train_model), accuracy_score(y_test,y_test_model)
```

c- 5-folds cross-validation

Nous pourrions développer cette idée en utilisant encore plus de folds dans les données - par exemple, voici une représentation visuelle de la 5-folds-cross-validation :

Split 1	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Metric 1
Split 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Metric 2
Split 3	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Metric 3
Split 4	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Metric 4
Split 5	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Metric 5

Training data Test data

5-fold cross validation

Figure 20

Ici, nous divisons les données en cinq groupes et utilisons chacun d'entre eux à tour de rôle pour évaluer le modèle construit sur les autres 4/5 des données. Cette opération est implémentée par **cross_val_score** de Scikit-Learn:

```
cross_val_score(clf, X, Y, cv=5)

array([0.68181818, 0.69480519, 0.75324675, 0.75163399, 0.68627451])
```

Figure 21

La répétition de la validation sur différents sous-ensembles de données nous donne une idée encore plus précise des performances de l'algorithme.

-Exécutez le code plusieurs fois, comment vous expliquez le résultat ?

d- Leave one out cross validation (LOOCV)

Scikit-Learn met en œuvre un certain nombre de schémas de validation croisée utiles dans des situations particulières. Nous pouvons aller jusqu'au cas extrême où notre nombre de folds est égal au nombre de points de données : c'est-à-dire que nous nous entraînons sur tous les points sauf un dans chaque essai. Ce type de validation croisée est connu sous le nom de validation croisée leave-one-out, et peut être utilisé comme suit :

```
#LOOCV
scores = cross_val_score(model, X, Y, cv =LeaveOneOut() )
scores.mean()
print(scores)
print(scores.mean())
```

-Exécuter le code plusieurs fois, qu'est-ce que vous remarquez ? Comment expliquer le résultat ?

e- Application de KNN avec Feature Selection

-Exécutez le code suivant et comparer la performance de KNN avec et sans feature selection.

```
# Define the different feature selection filters
filters = [
    ("f_classif", f_classif),
    ("mutual_info_classif", mutual_info_classif),
    ("chi2", chi2)
]

# Evaluate the performance of each feature selection filter
for name, filter_func in filters:
    print("Evaluating filter:", name)
    k_best = SelectKBest(filter_func, k=4)
    X_train_fs = k_best.fit_transform(X_train, y_train)
    X_test_fs = k_best.transform(X_test)

    # Train a Random Forest classifier
    model = KNeighborsClassifier()
    model.fit(X_train_fs, y_train)

    # Predict the target variables of test set
    y_pred = model.predict(X_test_fs)
```

```
# Evaluate accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}%".format(accuracy * 100))
print("---" * 20)
```

f. Evaluation du modèle

La figure 22 présente l'application de l'algorithme KNN avec l'approche holdout sur la base de données Pima Indians.

```
model2 = KNeighborsClassifier(k=5)
y_pred=model2.fit(X_train,y_train).predict(X_test)
y_test_model=model2.predict(X_test)
cm1=confusion_matrix(y_test, y_test_model)
print("Confusion Matrix: \n", cm1)
print(accuracy_score(y_test, y_test_model))
print(precision_score(y_test, y_test_model, average='weighted'))
print(recall_score(y_test, y_test_model, average='macro'))
print("Accuracy:", accuracy_score(y_test, y_test_model))
print("Confusion matrix:", confusion_matrix(y_test, y_test_model))
print("Classification report :", classification_report(y_test, y_test_model))
```

Le résultat est affiché en termes de matrice de confusion. Aussi le rapport de métriques est généré à l'aide de la fonction *Classification_report*.

```
Confusion Matrix:
[[65 34]
 [20 35]]
0.6493506493506493
0.6727560589453173
0.6464646464646464
Accuracy: 0.6493506493506493
Confusion matrix: [[65 34]
[20 35]]
Classification report :
precision    recall   f1-score   support
          0       0.76      0.66      0.71      99
          1       0.51      0.64      0.56      55
   accuracy                           0.65      154
    macro avg       0.64      0.65      0.64      154
weighted avg       0.67      0.65      0.66      154
```

Figure 22

-Interpréter les résultats.

Travail demandé : Compléter le tableau suivant en fixant *n_neighbors*=5.

	Holdout (70%-30%)	5-folds-cross-validation	LOOCV
Accuracy			
Precision			
Recall			
F1-score			

Est-ce que le choix de la méthode de validation impacte les performances du modèle?

g- Trouver les paramètres optimaux : Grid Search

Il existe plusieurs manières de tester les paramètres d'un modèle et le Grid Search est une des méthodes les plus simples. Pour chaque paramètre, on détermine un ensemble de valeurs que l'on souhaite tester. Le Grid Search croise simplement chacune de ces hypothèses et va créer un modèle pour chaque combinaison de paramètres. Il renvoie par la suite les paramètres du modèle avec la meilleure performance.

Le Grid Search fournie par **GridSearchCV** génère de manière exhaustive des candidats à partird'une grille de valeurs de paramètres spécifiées avec le paramètre **param_grid**.

On l'applique pour KNN :

```
#creat a new KNN model
Knn2 = KNeighborsClassifier()
grid_param={'n_neighbors': range(1,31),
            'weights': ['uniform', 'distance'],
            'metric' : ['euclidean', 'manhattan', 'minkowski']}
grid = GridSearchCV(Knn2, grid_param, cv = 10, scoring = 'accuracy')
grid.fit(X,Y)
print(grid.best_score_)
print(grid.best_params_)
print(grid.best_estimator_)
```

⇨ 0.7681989063568011
 {'metric': 'euclidean', 'n_neighbors': 15, 'weights': 'distance'}
 KNeighborsClassifier(metric='euclidean', n_neighbors=15, weights='distance')

Figure 23

-Comparer les résultats obtenus avec les résultats précédent.

#Question 8 : Appliquer le grid search sur KNN avec une dataset où les meilleurs attributs sont sélectionnés.

2. Construire le modèle Decision Tree

a - Modèle d'apprentissage :

Un arbre de décision (ou decision tree en anglais) est un modèle d'apprentissage automatique utilisé pour la classification ou la régression. Il s'agit d'une structure en forme d'arbre où chaque nœud interne représente une décision basée sur une caractéristique (ou "feature") des données d'entrée, et chaque feuille représente une prédiction.

L'arbre est construit de manière récursive en sélectionnant la caractéristique qui divise le mieux les données d'entrée en sous-groupes homogènes (par exemple, en classant les exemples en fonction de leur étiquette de classe majoritaire). Cette sélection est souvent effectuée en maximisant une mesure d'impureté (par exemple, l'entropie ou l'indice de Gini) qui mesure la quantité d'incertitude dans les données d'entrée.

L'arbre de décision peut être utilisé pour la classification en prédisant la classe majoritaire correspondante à la feuille atteinte par les données d'entrée, ou pour la régression en prédisant la valeur moyenne des exemples correspondant à la feuille atteinte.

Les avantages de l'arbre de décision sont sa simplicité, son interprétabilité et sa capacité à capturer des interactions non linéaires entre les caractéristiques. Les inconvénients sont sa tendance à surapprendre sur les données d'entraînement, sa sensibilité au bruit et à la variation des données, ainsi que sa difficulté à modéliser des relations complexes entre les caractéristiques.

La base de données est déjà séparée en deux ensembles : un ensemble d'entraînement (training set) et un ensemble de test (test set). L'ensemble d'entraînement sera utilisé pour entraîner l'arbre de décision, tandis que l'ensemble de test sera utilisé pour évaluer ses performances.

b- Entrainement de Decision Tree

On Entraîne l'arbre de décision en utilisant les paramètres par défaut et on affiche l'accuracy.

```
clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
acc2 = metrics.accuracy_score(y_test, y_pred)
print("Accuracy:", acc2)
```

-Vous pouvez ajuster les paramètres de l'arbre de décision tels que la profondeur maximale ou le

nombre minimal d'exemples par feuille pour éviter la surapprentissage.

Les principaux paramètres que l'on peut régler pour l'algorithme de l'arbre de décision dans la bibliothèque Scikit-learn (sklearn) en Python :

- criterion** : mesure d'impureté utilisée pour sélectionner la meilleure caractéristique de division. Peut être "gini" pour l'indice de Gini ou "entropy" pour l'entropie (valeur par défaut).
- splitter** : stratégie de division utilisée pour choisir la caractéristique de division. Peut être "best" pour choisir la meilleure caractéristique de division ou "random" pour choisir une caractéristique de division aléatoire.
- max_depth** : profondeur maximale de l'arbre. Si None (valeur par défaut), l'arbre sera développé jusqu'à ce que toutes les feuilles soient pures ou que le nombre minimal d'exemples par feuille soit atteint.
- min_samples_split** : nombre minimal d'exemples requis pour diviser un nœud interne. Si un nœud contient moins d'exemples que cela, il ne sera pas divisé.
- max_features** : nombre maximal de caractéristiques à considérer lors de la recherche de la meilleure division. Peut être un nombre entier, une fraction (par exemple 0,5 pour la moitié des caractéristiques) ou "sqrt" pour la racine carrée du nombre de caractéristiques (valeur par défaut).
- random_state** : graine aléatoire utilisée pour l'initialisation du générateur de nombres aléatoires, pour assurer la reproductibilité des résultats.

c- Visualiser l'arbre de décision

```
from sklearn import tree
fig = plt.figure(figsize=(15,15))
_ = tree.plot_tree(clf,
                   feature_names=feature_names,
                   class_names=["0","1"],
                   filled=True)
```

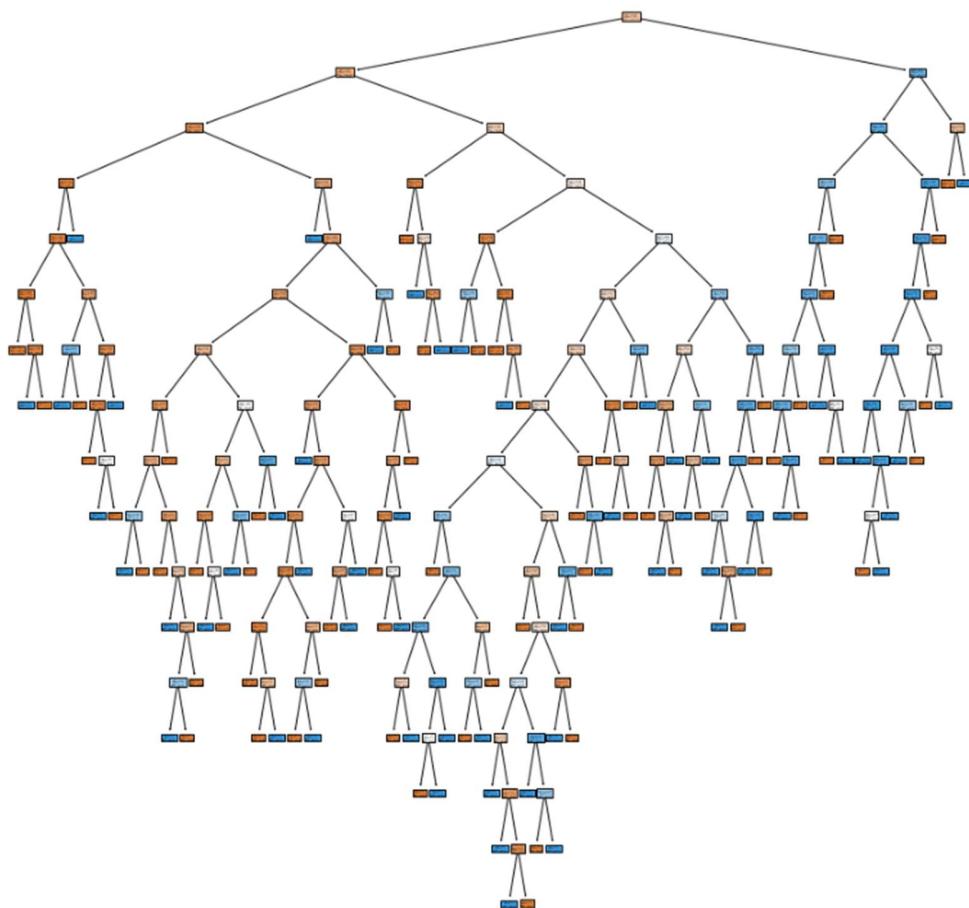


Figure 24

d- Les paramètres optimaux : Grid Search

- Comme on a fait pour KNN, on applique le Grid Search pour Decision Tree.

```
#Grid search
DT = tree.DecisionTreeClassifier()
params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [2, 4, 6, 8, 10],
    'min_samples_split': [2, 4, 6, 8, 10],
    'min_samples_leaf': [1, 2, 3, 4, 5]
}
grid = GridSearchCV(DT, params, cv = 10, scoring = 'accuracy')
grid.fit(X,y)
print(grid.best_score_)
print(grid.best_params_)
print(grid.best_estimator_ )
```

-On visualise l'arbre avec les paramètres optimaux:

```
dot_data = tree.export_graphviz(grid.best_estimator_, out_file=None,
feature_names=feature_names, class_names=['no','yes'],
filled=True, rounded=True,
special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

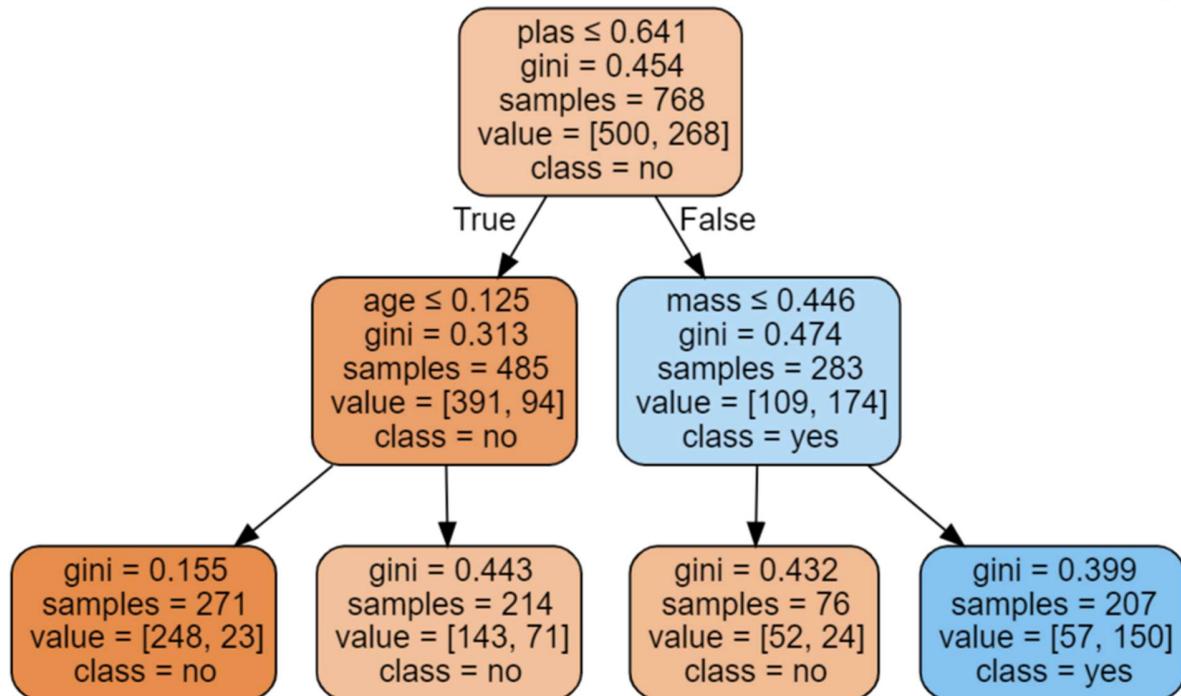


Figure 25

e- Pruning

Le "pruning" (élagage) des arbres de décision est une technique de régularisation utilisée pour éviter le surajustement (overfitting) d'un arbre de décision. Le "pruning" vise à éviter cela en supprimant des parties de l'arbre qui n'améliorent pas sa précision de manière significative. Il existe plusieurs techniques de "pruning", mais l'une des plus courantes est appelée élagage de l'arbre à coût minimal (minimal cost complexity pruning). Cette technique consiste à ajouter un paramètre de complexité (alpha) à la fonction de coût de l'arbre de décision. En ajustant ce paramètre, on peut trouver la taille optimale de l'arbre qui minimise la perte sur l'ensemble de données de validation. On l'applique comme suit :

- Utiliser la fonction **cost_complexity_pruning_path** de la bibliothèque scikit-learn pour calculer une série de valeurs de paramètre d'alpha pour la taille de l'arbre.

- Créer une série d'arbres de décision avec différentes valeurs d'alpha à l'aide de la fonction DecisionTreeClassifier de la bibliothèque scikit-learn.
- Évaluer les performances de chaque arbre de décision sur l'ensemble de test à l'aide de la fonction accuracy_score de la bibliothèque scikit-learn.
- Sélectionner le meilleur arbre de décision en fonction des performances sur l'ensemble de test.

```
# Use a pruning algorithm to prune the decision tree
clf = tree.DecisionTreeClassifier()
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path ccp_alphas[:-1]
clfs = []
for ccp_alpha in ccp_alphas:
    clf = tree.DecisionTreeClassifier(ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

# Evaluate the pruned decision tree using the testing data
acc_scores = []
for clf in clfs:
    y_pred = clf.predict(X_test)
    acc_score = accuracy_score(y_test, y_pred)
    acc_scores.append(acc_score)

# Find the best pruning parameter based on accuracy score
best_clf = clfs[acc_scores.index(max(acc_scores))]

# Evaluate the best pruned decision tree using the testing data
y_pred = best_clf.predict(X_test)
acc_score = accuracy_score(y_test, y_pred)
print("Accuracy score: {:.2f}".format(acc_score))
print("ccp_alpha: {:.3f}".format(ccp_alphas[acc_scores.index(max(acc_scores))]))
```

#Question 9 : Afficher le meilleur arbre obtenu avec la méthode de Pruning

3. Construire le modèle SVM

a - Modèle d'apprentissage

-SVM (Support Vector Machine ou Machine à vecteurs de support) : Les SVMs sont une famille d'algorithmes d'apprentissage automatique qui permettent de résoudre des problèmes tant de classification que de régression. Ils sont connus pour leurs solides garanties théoriques, leur grande flexibilité ainsi que leur simplicité d'utilisation.

-Le principe est de séparer les données en classe à l'aide d'une frontière, de telle façon que la distance entre les différents groupes de données et la frontière séparatrice soit maximale. Cette distance est appelée marge. Et les données les plus proches de la frontière sont appelées vecteurs de support.

Nous entraînons une machine à vecteur de support (SVM) avec les paramètres par défaut. En faisant appel à la méthode **score()** on calcule l'accuracy de ce modèle.

```
svm = SVC()
svm.fit(X_train, y_train)
acc3= svm.score(X_test, y_test)
svm.score(X_test, y_test)
```

b- Changement des paramètres par défaut

SVC peut prendre plusieurs paramètres optionnels, les plus importants sont : **C, kernel et gamma**.

- «**C**» est un terme de régularisation qui pondère l'attache aux données (un nombre réel strictement positif).

-«**Kernel**» spécifie le type de noyau à utiliser dans l'algorithme, et qui peut être soit « linear », « poly », « rbf », « sigmoid » ou « precomputed », par défaut il prend la valeur « rbf ».

- «**Gamma**» : correspond au coefficient de noyau pour les noyaux « rbf », « poly », et « sigmoid ». Prend comme valeur soit « scale », « auto » ou un nombre à virgule.

#Question 10 : Modifier le type de kernel pour SVM, on crée trois modèles qui ont comme kernel : « linear », « poly » et « rbf ».

Travail demandé : Compléter le tableau suivant et visualiser les résultats.

C	0.1	1	0.2	1	10	100
Kernel	rbf	rbf	poly	poly	rbf	linear
Gamma	0.1	0.1	1	0.5	10	1
Accuracy						

f- Les paramètres optimaux : Grid Search

Comme on a fait pour les 2 autres classificateurs, on applique le Grid Search pour SVM, en variant les paramètres :

- Type de kernel : {linear, poly, rbf}
- C : {0.1,0.5,1.0,2.0,10.0}

```

parametres = {"kernel":['linear','poly','rbf'], "C":[0.1,0.5,1.0,2.0]}
#classifieur à utiliser
svmc = SVC()
#instanciation de la recherche
grille = GridSearchCV(estimator=svmc,param_grid=parametres,scoring="accuracy")
#lancer l'exploration
resultats = grille.fit(X_train,y_train)
print(resultats.best_params_)
print(resultats.best_score_)
```

-Quels sont les Meilleurs paramètres obtenus?

Question 11 : Appliquer le grid search sur SVM avec une dataset où les meilleurs attributs sont sélectionnés.

4- Comparer SVM avec KNN et Decision Tree

Voici un exemple de code Python pour comparer les performances des algorithmes de classification DT, SVM et KNN, à l'aide d'un graphe :

```

# Afficher les performances des modèles
labels = ['DT', 'SVM', 'KNN']
accs = [acc2, acc3, acc]
plt.bar(labels, accs)
plt.title('Accuracy of Indian Pima dataset classification models')
plt.ylabel('Accuracy')
plt.show()
```

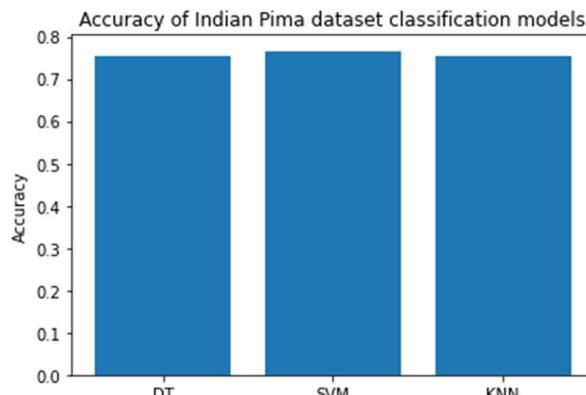


Figure 26

Quel modèle classe le mieux le jeu de données Indian Pima ?

Travail demandé : Comparer KNN, DT et SVM en utilisant les meilleurs paramètres ainsi que les meilleurs attributs de la dataset. Visualiser les résultats avec d'autres types de graphes.