Institut de Statistique de l'Université de Paris

Projet Python

Sujet 1

Juliette Darsonval

16 janvier 2018

P. Saurel

Table des matières

CHAPITRE

1 INTRODUCTION

1.1 Objectif

Python est un langage de programmation orienté objet qui peut s'utiliser dans de nombreux contextes grâce à des librairies spécialisées. L'objectif de ce projet est d'étudier une base de données libre de droit, de traiter ces données au moyen d'outils statistiques et de restituer visuellement les analyses et résultats obtenus. Dans ce projet, on utilisera les librairies suivantes :

- Pandas pour l'importation des données et la gestion des dataframes
- Numpy et Scipy pour les analyses et calculs
- Matplotlib et Seaborn pour la visualisation des résultats
- Sklearn pour la partie apprentissage statistique

1.2 Présentation du jeu de données

Le jeu de données utilisé est extrait du site UCI Machine Learning Repository et téléchargeable en open-source à l'adresse suivante : https://archive.ics.uci.edu/ml/datasets/adult. Le dataset est nommé "Adult" ou "Census Income" est a été extrait de la base de données Census de 1994 par Barry Becker et est principalement utilisé dans un objectif de classification.

Il est composé de 14 variables explicatives (6 quantitatives, 8 qualitatives) et de 32 561 observations. Le but est de prédire la variable réponse "income" qui détermine si une personne gagne plus de 50.000$ par an.

CHAPITRE

2

ANALYSE EXPLORATOIRE

Dans cette partie, nous réalisons une analyse exploratoire des données. Les packages matplotlib et seaborn ont été utilisé pour réaliser les graphiques. Certaines fonctions graphiques sont aussi disponibles dans la configuration classique de Python.

2.1 Variables quantitatives

Nos variables quantitatives sont réparties de la façon suivante :

	age	education_num	capital_gain	capital_loss	hours_per_week
count	32561.000000	32561.000000	32561.000000	32561.000000	32561.000000
mean	38.581647	10.080679	1077.648844	87.303830	40.437456
std	13.640433	2.572720	7385.292085	402.960219	12.347429
min	17.000000	1.000000	0.000000	0.000000	1.000000
25%	28.000000	9.000000	0.000000	0.000000	40.000000
50%	37.000000	10.000000	0.000000	0.000000	40.000000
75%	48.000000	12.000000	0.000000	0.000000	45.000000
max	90.000000	16.000000	99999.000000	4356.000000	99.000000

FIGURE 2.1 – Description des variables quantitatives

La moyenne d'âge de notre base de données est de 38 ans, la durée moyenne des études est de 10 ans et la durée de travail est de 40 heures par semaine.

Nous affichons quelques graphiques descriptifs de nos variables quantitatives :

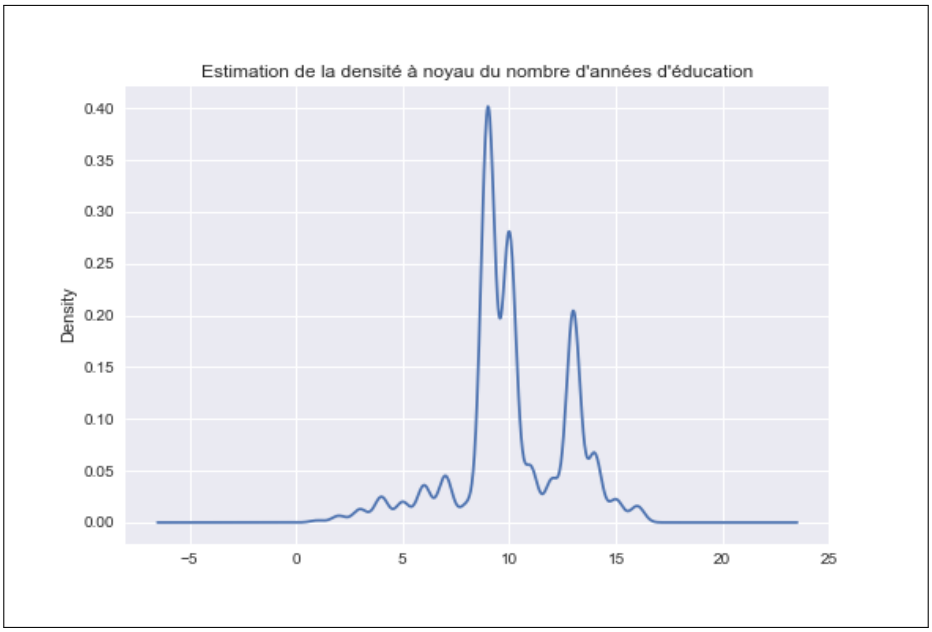

FIGURE 2.2 – Densité du nombre d'années d'éducation

L'estimateur à noyau du nombre d'années peut être considéré comme un lissage de l'histogramme de cette variable.

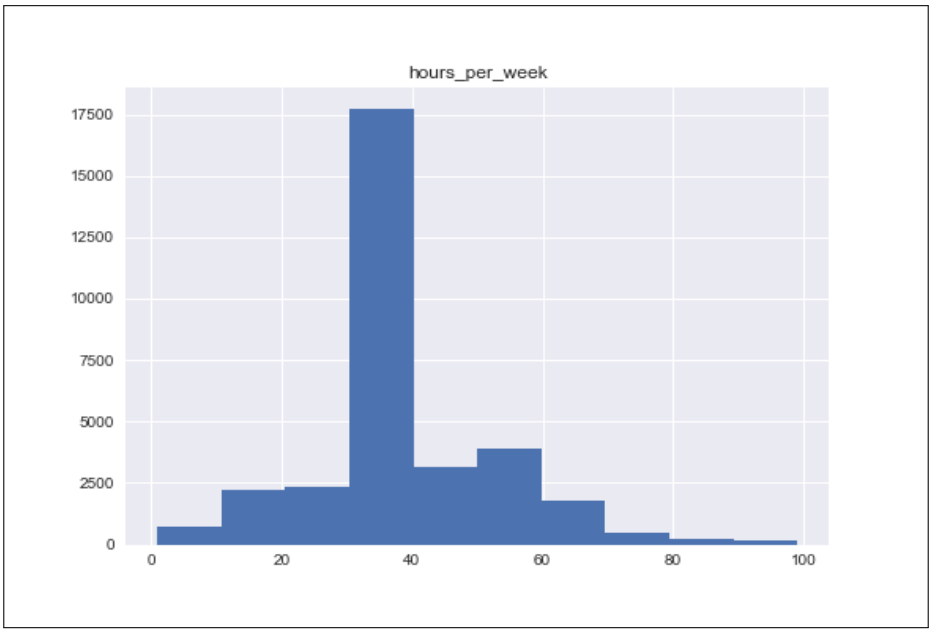

FIGURE 2.3 – Histogramme du nombre d'heures de travail par semaine

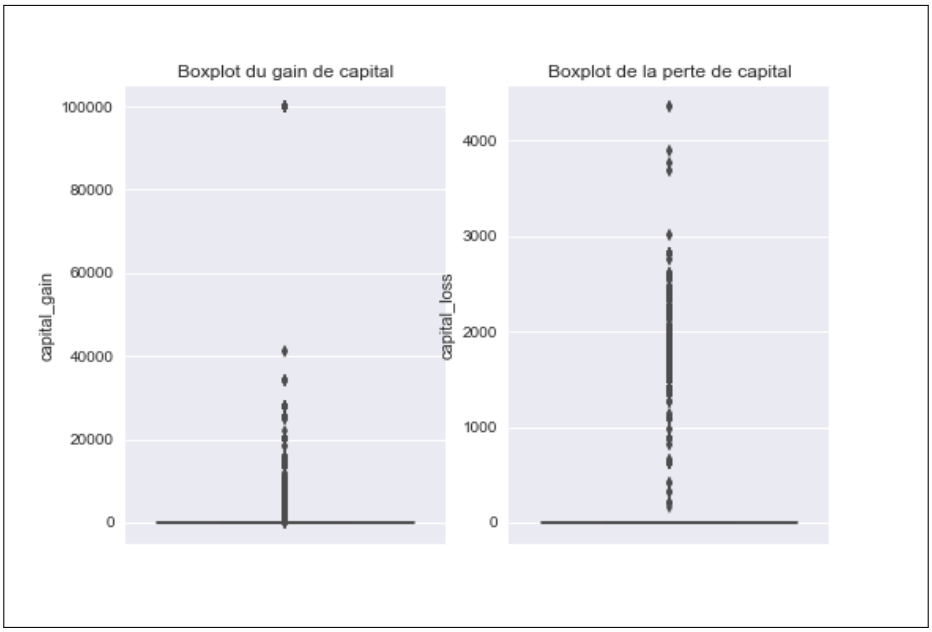

FIGURE 2.4 – Boxplot du capital

Le boxplot du gain et de la perte de capital nous montre que les valeurs sont en majorités nulles.

2.2 Variables qualitatives

Les variables qualitatives sont réparties de la façon suivante :

- workclass a 9 modalités, la plus fréquente étant Private avec 22 696 occurrences
- education a 16 modalités, la plus fréquente étant HS-grad avec 10 501 occurrences
- marital_status a 7 modalités, la plus fréquente étant Married-civ-spouse avec 14 976 occurrences
- occupation a 15 modalités, la plus fréquente étant Prof-specialty avec 4 140 occurrences
- relationship a 6 modalités, la plus fréquente étant husband avec 13 193 occurrences
- race a 5 modalités, la plus fréquente étant white avec 27 816 occurrences
- sex a 2 modalités, la plus fréquente étant male avec 21 790 occurrences
- native_country a 42 modalités, la plus fréquente étant united states avec 29 170 occurrences
- income a 2 modalités, la plus fréquente étant <=50K avec 24 720 occurrences

Plusieurs types de graphiques sont disponibles pour décrire visuellement nos variables. Par exemples des diagrammes en bâtons :

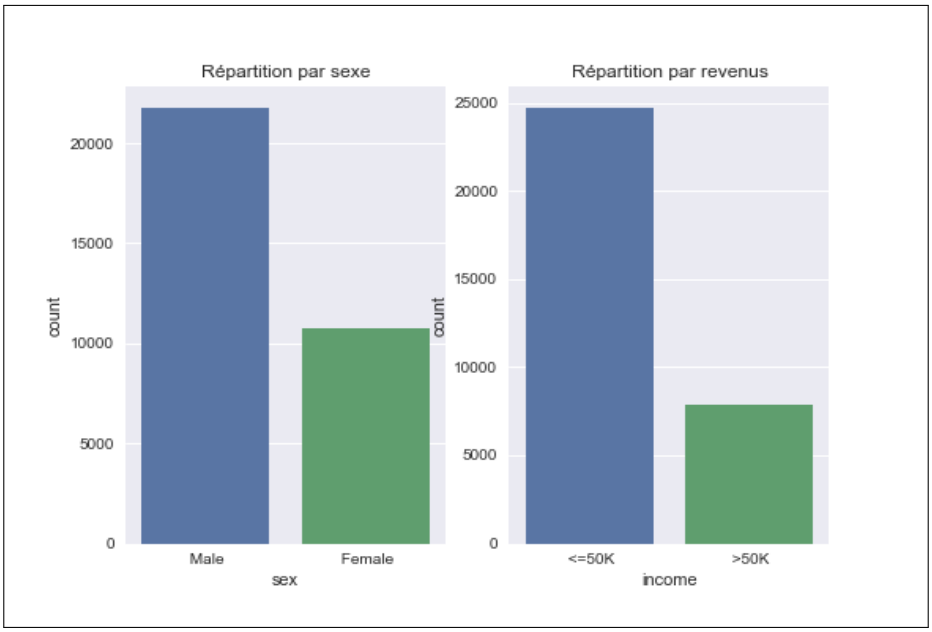

FIGURE 2.5 – Répartition des variables sexes et revenus

Comme notre description de la base de données nous a montré, la modalité male représente les 2/3 de la variable sexe. De la même façon, la modalité <=50K est aussi majoritaire pour la variable income.

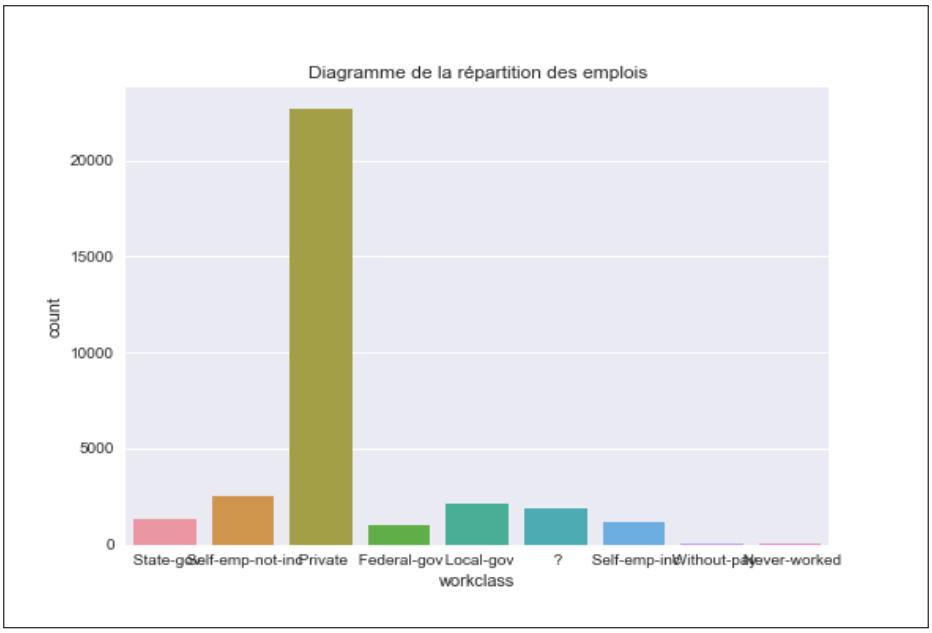

FIGURE 2.6 – Répartition des emplois

La modalité Private est largement plus représentée que les autres au sien de la variable workclass.

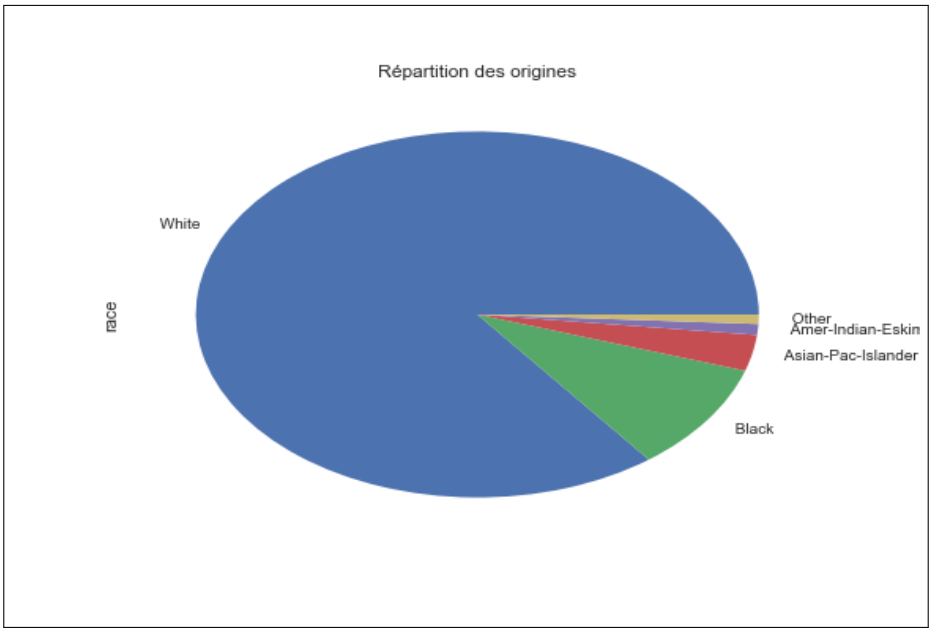

FIGURE 2.7 – Répartition des origines

De le même façon, la modalité white est aussi sur-représentée au sein de la variable race.

2.3 Etude bivariée

Il est aussi intéressant de réaliser une étude des variables 2 à 2. Pour les variables quantitatives, on commence par trouver la matrice des corrélations qui nous indique les corrélations entre variable.

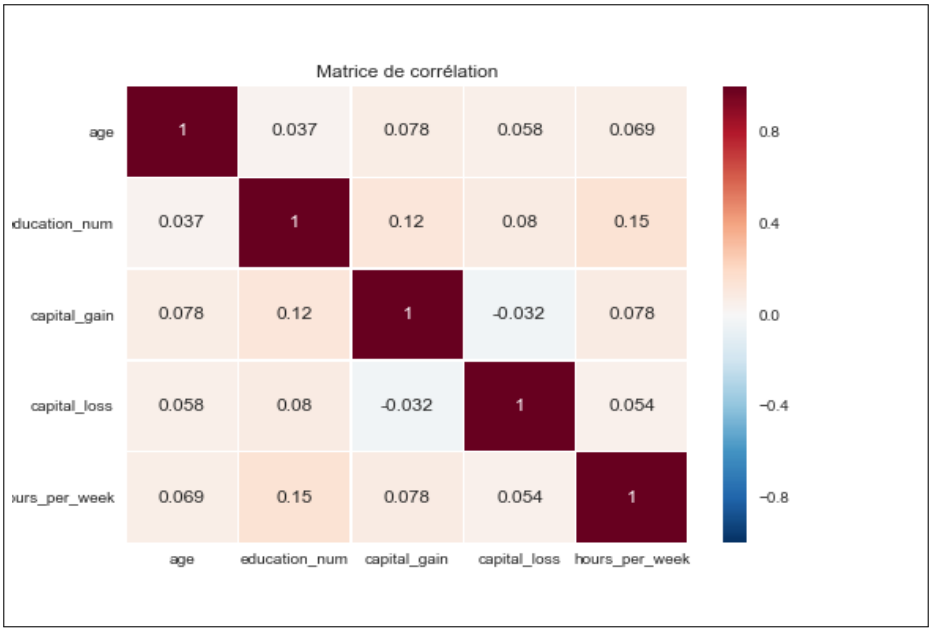

FIGURE 2.8 – Heatmap des corrélations

Les corrélations négatives sont en bleu dans l'image. Plus la corrélation est élevée, plus la couleur rose est foncée.

On peut aussi réaliser un nuage de points de toutes les variables quantitatives 2 à 2.

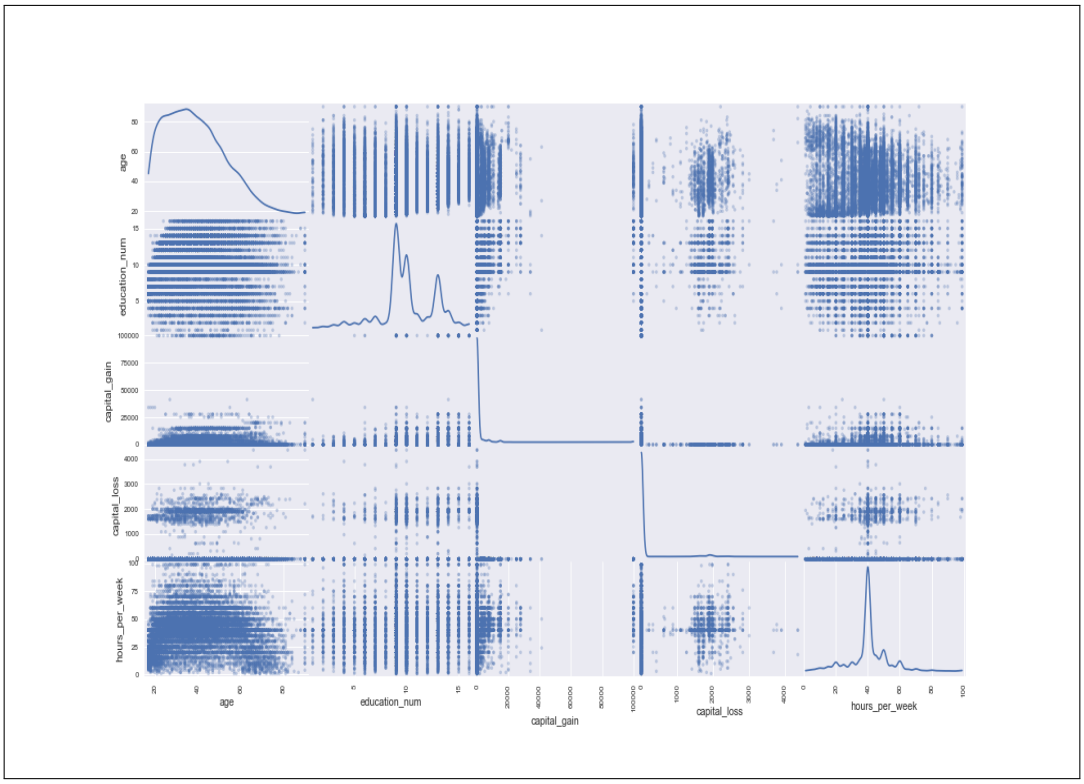

FIGURE 2.9 – Pairplot des variables quantitatives

Pour étudier l'impact de certaines variables sur la variable réponse "income", nous utilisons des factorplots.

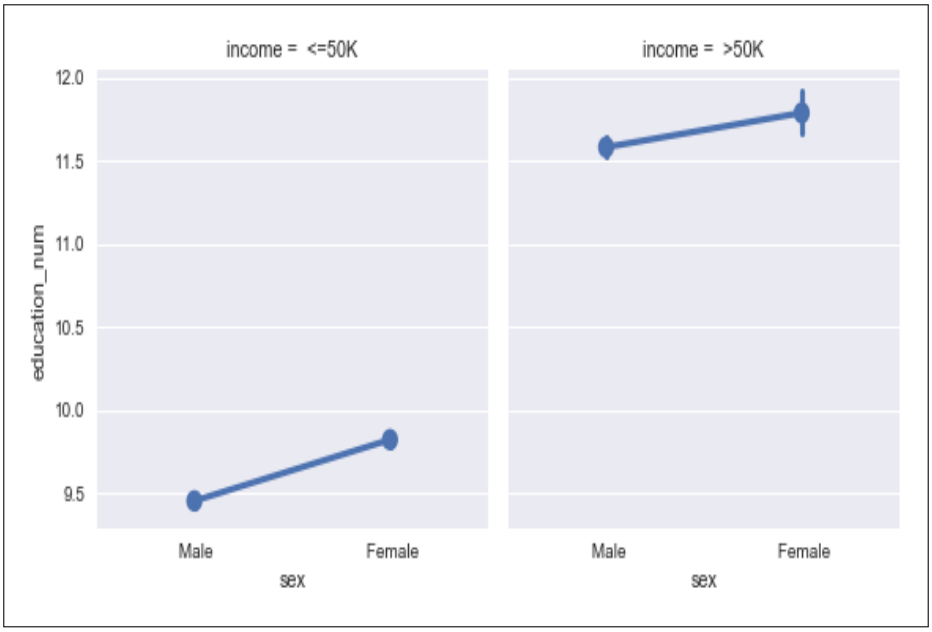

FIGURE 2.10 – Nombre d'années d'éducation en fonction du sexe et des revenus

On constate que quelque soit le salaire, les femmes effectuent en moyenne plus d'années d'étude. En moyenne, il y a un écart de 3 ans d'années d'étude entre la population gagant plus de 50 000$ et celle gagant moins de 50 000$.

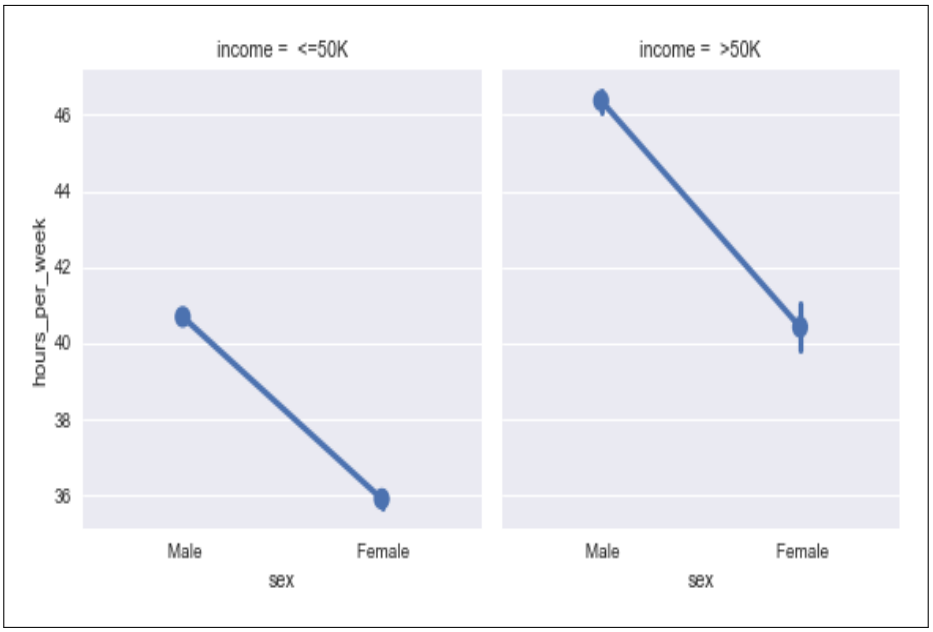

FIGURE 2.11 – Nombre d'heures de travail par semaine en fonction du sexe et des revenus

De la même façon, ceux gagant plus de 50 000$ effectuent plus d'heures par semaine que ceux gagant moins.

2.4 Classe

Le langage Python est un langage objet. On peut donc créer des classes avec des attributs mais aussi des objets.

Dans ce projet, on crée une classe Adult qui nous permet d'obtenir une description sommaire d'un objet de cette classe, mais aussi de vérifier si l'objet appartient bien cette classe. On peut aussi par exemple comparer l'âge entre deux personnes appartenant à cette classe.

CHAPITRE

3 TRAITEMENT

3.1 Nettoyage et recodage des variables

La plupart des estimateurs de la bibliothèque scikit-learn admettent seulement en paramètres des variables numériques et sans valeurs manquantes. Pour cela, nous devons donc procéder à deux étapes essentielles avant de pouvoir faire tourner nos modèles :

- Traiter les observations à valeurs manquantes : plusieurs solutions s'offrent à nous. Nous pouvons choisir de supprimer simplement l'observation, ou alors de remplacer la valeur manquante par la moyenne, la médiane ou l'observation la plus fréquente par exemple.
- Recoder les variables catégorielles : en effet, nous avons besoin de les transformer en variables numériques afin de pouvoir les utiliser dans nos modèles.

La méthode automatique sous Numpy nous permet de déterminer que notre dataset ne contient pas de valeurs manquantes. Il ne nous reste plus qu'à encoder nos variables catégorielles.

Cet encodage peut se réaliser de plusieurs façons : une possibilité est de recoder nos variables en {0,1,...} suivant le nombre de modalités en utilisant la méthode de "Label Encoding". Toutefois, cette solution ne paraît pas entièrement adaptée à toutes nos variables. En effet, elle est particulièrement efficace pour des variables ordinales. Dans notre cas, la plupart des variables ne sont pas ordonnées et utiliser cette méthode pourrait biaiser l'analyse. Nous choisissons d'utiliser la méthode "One Hot Encoding".

Avec cette approche, chaque catégorie sera recodée dans une nouvelle colonne avec la valeur 0 ou 1. Cela nous évite donc de donner trop d'importance à certaines catégories mais nous rajoute un nombre important de colonnes dans notre base de données.

Pour cela, après plusieurs tests, nous décidons donc d'abandonner certaines variables comme les variables "native country", "education" (la présence de la variable "education_num" nous permet de conserver de l'information), "occupation" (cette fois, la présence de la variable "workclass" nous permet de garder de l'information) et "relationship".

De la même façon, nous recodons notre variable réponse "income" en {0,1} ainsi que la variable "sex" avec la méthode "Label Encoding". Pour la variable "sex", la modalité Female prend la valeur 0 et la modalité Male prend la valeur 1. Pour la variable "income", la modalité <=50K prend la valeur 0 et la modalité >50K prend la valeur 1.

3.2 Séparation training set / test set

Par la suite, nous allons utiliser des méthodes de classification supervisée. Nous devons donc séparer notre base de données en deux sets :

- Le training set : Ici, on prend 75% de notre jeu de données original. Il s'agit de la base sur laquelle on entraînera nos modèles à correctement classifier nos observations.
- Le test set : Il reste donc 25% de la base de donnée. Il s'agit du set avec lequel nous pourrons comparer nos prévisions afin d'établir la qualité de prédiction de nos différents modèles.

CHAPITRE

4 CLASSIFICATION

Dans cette partie, nous utiliserons la librairie Scikit-learn afin de fitter les 4 modèles suivants :

- K-plus-proches-voisins
- Régression logistique
- Arbre de décision
- Random Forest

Il s'agit de 4 méthodes de classification. Notre but sera donc de déterminer la meilleure méthode pour prédire si une personne de notre data set gagne plus de 50.000$ par an ou pas.

Pour tous les modèles nous utiliserons les étapes suivantes :

- Un appel du modèle avec ses paramètres, par exemple : knn = neighbors.KNeighborsClassifier(...)
- Le fitting du modèle sur nos bases de train : knn.fit(X_{train}, y_{train})
- La prédiction à partir de notre modèle fitté : $y_{predict}$ = knn.predict(X_{test})
- Le calcul de notre score de précision : accuracy_score(y_{test}, $y_{predict}$)

4.1 K-plus-proches-voisins

On implémente les k-plus-proches-voisins avec la méthode *KNeighborsClassifier* de la classe *sklearn.neighbors*. A partir de la documentation, on obtient l'implémentation par défaut :

KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs)

ainsi que la description des options disponibles. Ci-dessous, nous détaillons quelques unes des options nous intéressant :

- n_neighbors est le nombre de voisins que l'on recherche. Pour cet algorithme, il s'agit du paramètre à optimiser pour gérer la complexité du modèle.
- weights est le poids accordé aux observations. Plusieurs options sont disponibles :
  - uniform : par défaut, tous les points ont le même poids.
  - distance : les points ont comme poids l'inverse de leur distance au point étudié. Plus ils sont éloignés, moins on leur accorde de l'importance.
  - callable : on peut définir nous même une fonction de la distance.

- algorithm est l'algorithme utilisé pour implémenter les plus-proches-voisins. Comme précédemment, on a plusieurs options :

  — ball_tree : à l'initialisation, deux clusters sphériques sont créés et chaque observation appartient à l'un ou à l'autre. Au fur et à mesure de l'algorithme, les clusters sont subdivisés en d'autres clusters toujours sphériques. Cet algorithme est basé sur un arbre binaire.

  — kd_tree : aussi basé sur un arbre binaire. Les données sont séparées en deux selon la médiane de la variable ayant la plus grande variance et ainsi de suite.

  — brute : l'algorithme va chercher les plus-proches-voisins de manière basique.

  — auto : choisit un algorithme ci-dessus selon les valeurs passées à la méthode fit.

- leaf_size : valeurs utilisées pour les algorithmes BallTree et KDTree. Il s'agit du nombre de feuilles à partir duquel l'algorithme commencera à chercher de manière brute. Modifier cette valeur ne changera pas le résultat mais peut impacter de manière significative la vitesse ainsi que la mémoire utilisée par les algorithmes.

- p : la puissance utilisée pour la métrique.

- metric : la métrique utilisée pour l'arbre. La métrique par défaut 'minkowski' associée à p=2 revient à utiliser la distance euclidienne pour la recherche des plus proches voisins.

On commence par implémenter la version par défaut des k-plus-proches-voisins qui nous donne une précision de 0.841. Afin de comparer ce résultat, on fait tourner le modèle en changeant d'abord la méthode des poids utilisée puis l'algorithme.

Ces tests faits "à la main" avec le nombre de voisins par défaut n=5 nous permet de déterminer que la meilleure combinaison semble être la méthode des poids uniformes associée à l'algorithme BallTree. Par la suite, on fait tourner l'algorithme avec différents nombres de voisins et on décide de choisir le nombre de voisins n=10.

Ce choix n'est peut-être pas le plus optimal car l'algorithme a seulement été testé sur quelques valeurs à la main. Une solution pour obtenir la valeur optimale du nombre de voisins serait d'effectuer une validation croisée sur une grille de valeurs. Ces méthodes sont implémentées sous Scikit-Learn mais nous décidons de ne pas les utiliser ici car notre but est simplement de comparer les différentes méthodes de classification entre elles, pas de toutes les optimiser.

Nous avons plusieurs indicateurs pour estimer la performance de notre modèle. Tout d'abord, nous avons une précision de prédiction sur le set de test de **85,2%**. Ensuite, nous affichons ci-dessous la matrice de confusion ainsi que la courbe ROC qui permettent de visualiser les vrais positifs et les vrais négatifs. Finalement, l'AUC (Area Under the Curve) nous donne aussi une indication sur la performance du modèle.

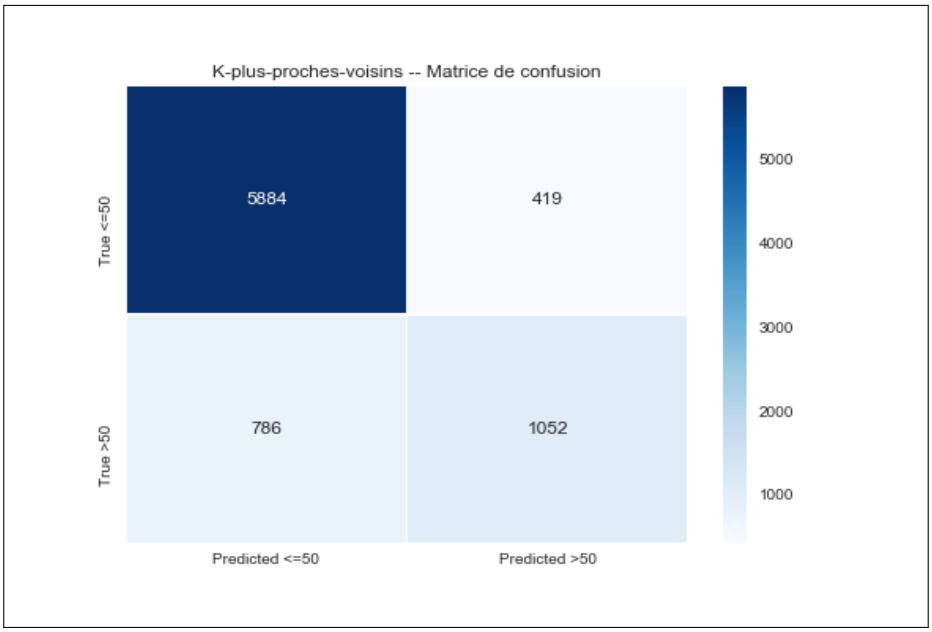

FIGURE 4.1 – Matrice de confusion – K-plus-proches-voisins

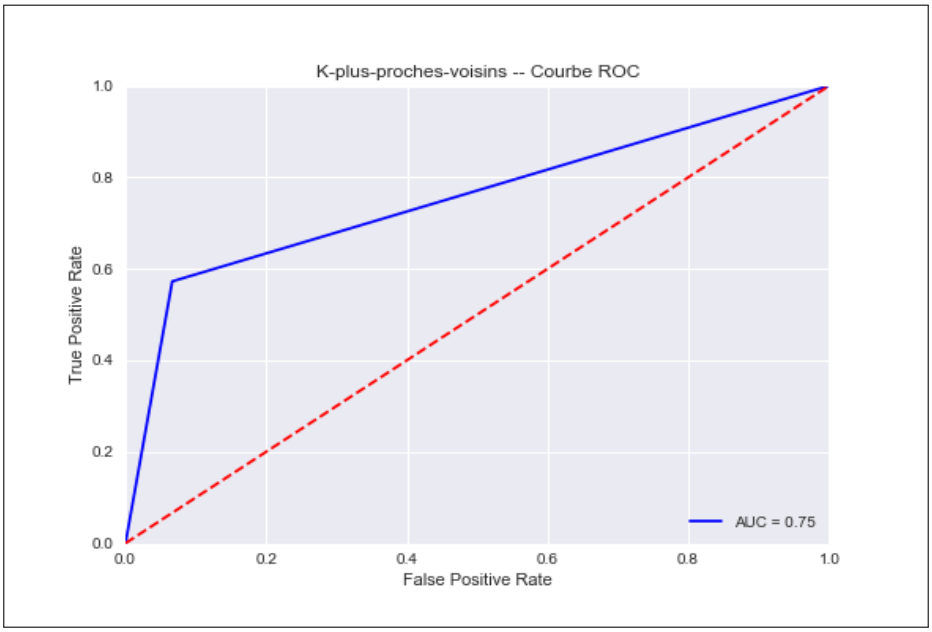

FIGURE 4.2 – Courbe ROC – K-plus-proches-voisins

4.2 Régression Logistique

On implémente la régression logistique avec la méthode *LogisticRegression* de la classe *sklearn.linear_model*. A partir de la documentation, on obtient l'implémentation par défaut :

LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1)

ainsi que la description des options disponibles. Ci-dessous, nous détaillons quelques unes des options nous intéressant :

- penalty : la pénalité utilisée, L1 ou L2 (par défaut).
- class_weight : permet d'attribuer un poids aux classes. On peut choisir de balancer automatiquement les classes, ou alors d'entrer nous même un dictionnaire des poids. Par défaut, chaque classe a automatiquement un poids de 1.
- solver : algorithme à utiliser pour l'optimisation.
  - newton-cg : Convient aux problèmes multilinéaires. Pénalité L2 seulement.
  - lbfgs : Convient aux problèmes multilinéaires. Pénalité L2 seulement.
  - liblinear : par déaut. Utile pour les petits datasets. Limité aux problèmes bilinéaires. Pénalité L1 et L2.
  - sag : Plus rapide sur un grand dataset. Convient aux problèmes multilinéaires. Pénalité L2 seulement.
  - saga : Plus rapide sur un grand dataset. Convient aux problèmes multilinéaires. Pénalité L1 et L2.
- max_iter : nombre maximum d'itérations du solver pour converger. Seulement utile pour newton-cg, sag et lbfgs.
- multi_class : ovr ou multinomial. Par défaut, ovr fitte un problème binaire.
- warm_start : quand fixé à True, la solution de l'appel prédédent est réutilisée pour l'optimisation. Pas utile pour liblinear.

On commence par implémenter la version par défaut de la régression logistique qui nous donne une précision de 84,7%. Afin de comparer ce résultat, on fait tourner le modèle en changeant modifiant le solver utilisé.

On observe que le solver "newton-cg" est celui ayant la précision la plus proche du solver "liblinear" mais il est toujours moins précis. Les solvers "lbfgs" et "sag" n'atteignent eux que 81% de précision. De plus, le solver "sag" demande un nombre d'itérations beaucoup plus important pour une plus faible précision de prédiction. A noter que le solver "saga" n'était pas disponible dans la version de scikit-learn utilisée.

On teste maintenant différentes options en conservant le solver "liblinear". La norme L1 nous donne une précision de prédiction de 84,76% contre 84,78% pour la norme L2. Utiliser l'option "balanced" pour le paramètre class_weight fait baisser notre taux de précision à 77%

On choisit donc de conserver le modèle par défaut avec un taux de précision de **84,7%** pour notre régression logistique. Comme précédemment, nous construisons la matrice de confusion ainsi que la courbe ROC.

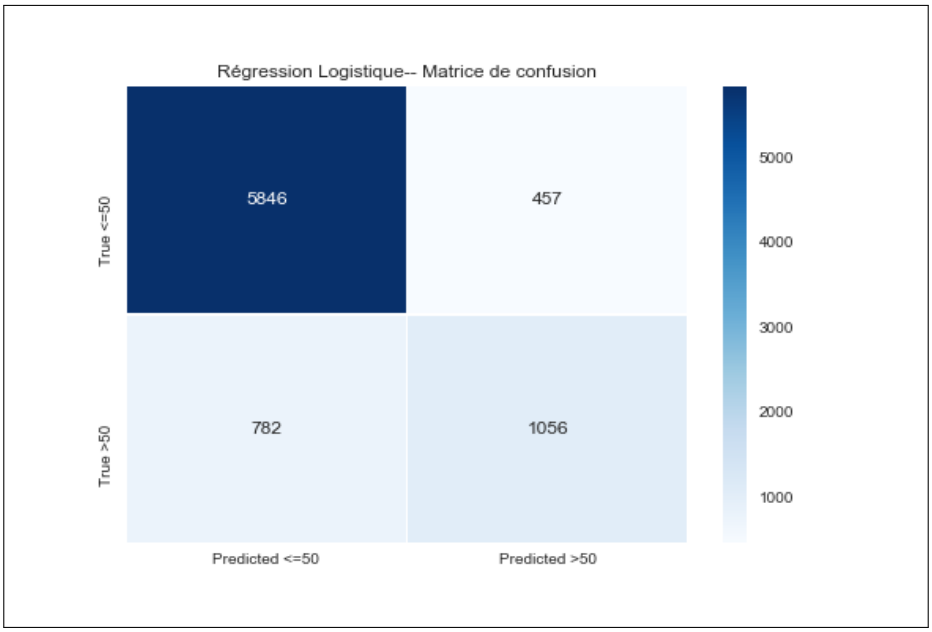

FIGURE 4.3 – Matrice de confusion – Régression Logistique

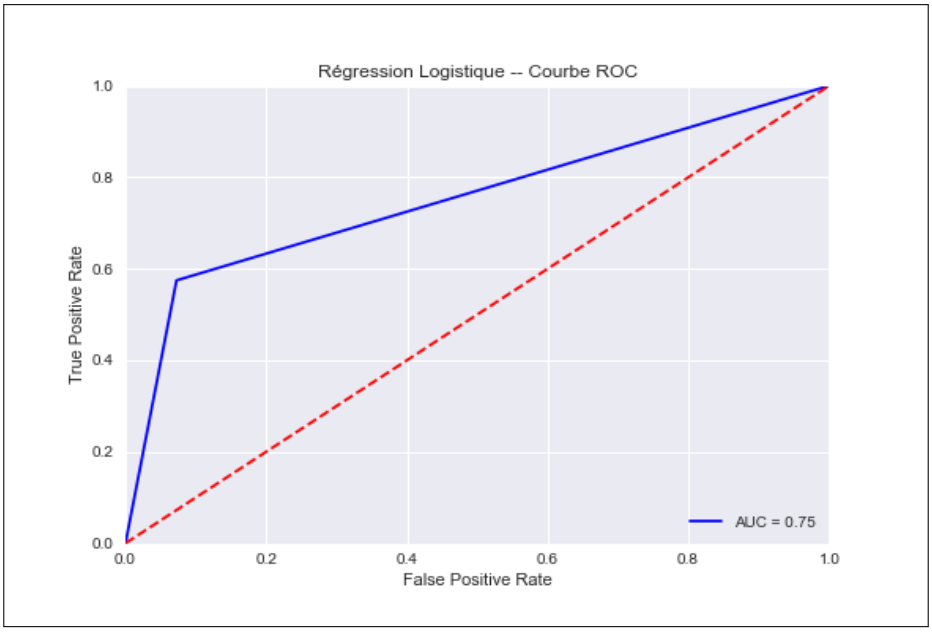

FIGURE 4.4 – Courbe ROC – Régression Logistique

4.3 Arbre de décision

On implémente un arbre de décision avec la méthode *DecisionTreeClassifier* de la classe *sklearn.tree*. A partir de la documentation, on obtient l'implémentation par défaut :

DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort=False)

ainsi que la description des options disponibles. Ci-dessous, nous détaillons quelques unes des options nous intéressant :

- criterion : le critère utilisé pour mesurer la qualité d'un split. Le coefficient de Gini d'impureté d'un noeud est utilisé par défaut mais on peut aussi utiliser l'entropie qui mesure le gain d'information.
- splitter : stratégie à utiliser pour choisir le split à chaque noeud. Par défaut la stratégie "best" choisit le meilleur split. La stratégie "random" choisit le meilleur split au hasard.
- max_depth : la profondeur maximale de l'arbre. Si None, l'arbre continue de grandir jusqu'à ce que toutes les feuilles soient pure ou jusqu'à ce que toutes les feuilles contiennent moins d'échantillons que min_samples_split.
- min_samples_split : nombre minimum d'échantillons requis pour splitter un noeud.
- min_samples_leaf : nombre minimum d'échantillons requis pour être une feuille.
- max_features : le nombre de variables à considérer pour opérer le split.
- max_leaf_nodes : nombre maximum de feuilles. Les meilleurs noeuds sont sélectionnés par leur réduction relative d'impureté.

On commence par implémenter la version par défaut de l'arbre de décision qui nous donne une précision de 82,9%. Afin de comparer ce résultat, on fait tourner le modèle en testant les différents paramètres.

Le critère de sélection basé sur l'entropie nous donne une précision de 83%. On peut voir que l'arbre implémenté par défaut a une profondeur de 45 et l'arbre implémenté avec le critère d'entropie une profondeur de 48. On décide donc d'essayer de limiter la taille de l'arbre dans les deux cas. En effet, un arbre trop profond aura tendance à over-fitter notre modèle et aura donc une moins bonne tendance à la généralisation.

Après plusieurs tests "à la main", il semble qu'une profondeur de 10 noeuds nous apporte le meilleur critère de précision pour les deux arbres.

On choisit donc de conserver le modèle avec le critère de sélection de Gini et une profondeur de 10 pour un taux de précision de **86,3%**. Comme précédemment, nous construisons la matrice de confusion ainsi que la courbe ROC.

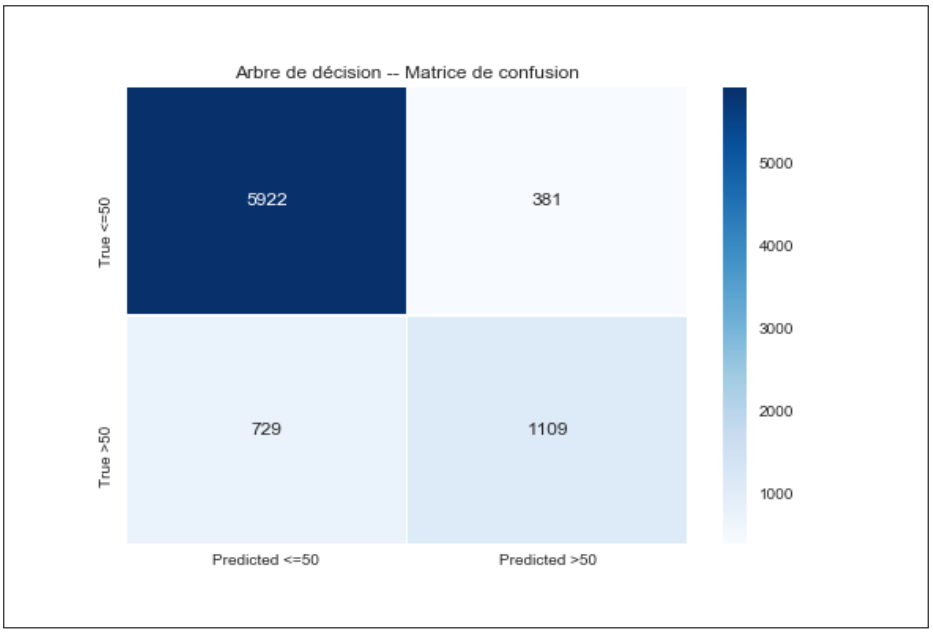

FIGURE 4.5 – Matrice de confusion – Arbre de décision

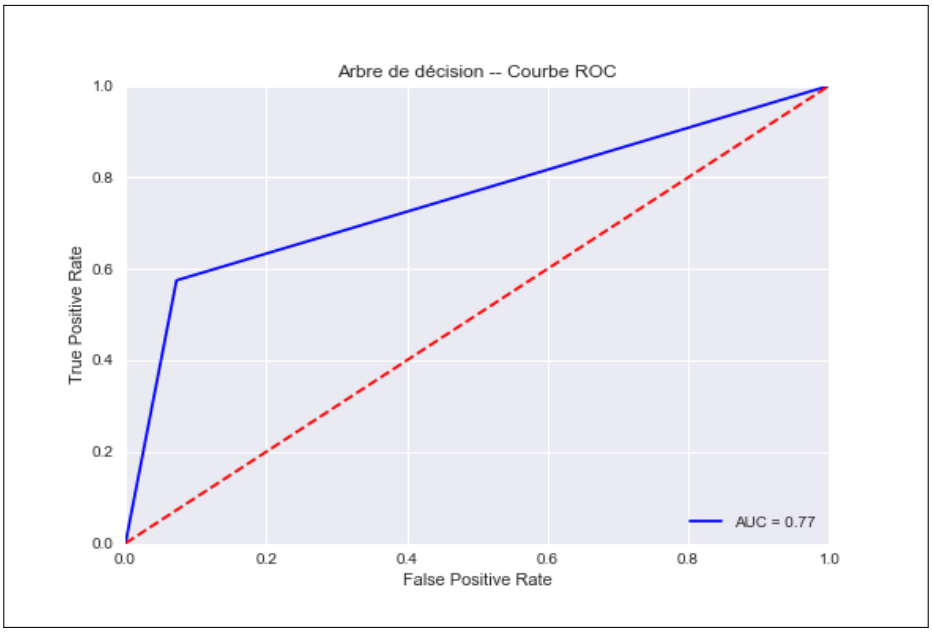

FIGURE 4.6 – Courbe ROC – Arbre de décision

4.4 Random Forest

On implémente une forêt aléatoire avec la méthode *RandomForestClassifier* de la classe *sklearn.ensemble*. A partir de la documentation, on obtient l'implémentation par défaut :

RandomForestClassifier(n_estimators=10, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False, class_weight=None)

ainsi que la description des options disponibles. Ci-dessous, nous détaillons quelques unes des options nous intéressant :

- n_estimators : nombre d'arbres à utiliser.
- bootstrap : utilisation ou non d'échantillonage lors de la construction des arbres.
- oob_score : utilisation de l'erreur "out-of-bag" lors du calcul de la précision généralisée.

On commence par implémenter la version par défaut de la random forest qui nous donne une précision de 84,58%. Afin de comparer ce résultat, on fait tourner le modèle en testant les différents paramètres.

Après plusieurs tests "à la main", il semble qu'augmenter le nombre d'arbres augmente la précision de notre modèle. Toutefois, cela augmente aussi le temps de calcul.

On choisit donc de conserver un modèle à 350 arbres pour un taux de précision de **84,9%**. Comme précédemment, nous construisons la matrice de confusion ainsi que la courbe ROC.

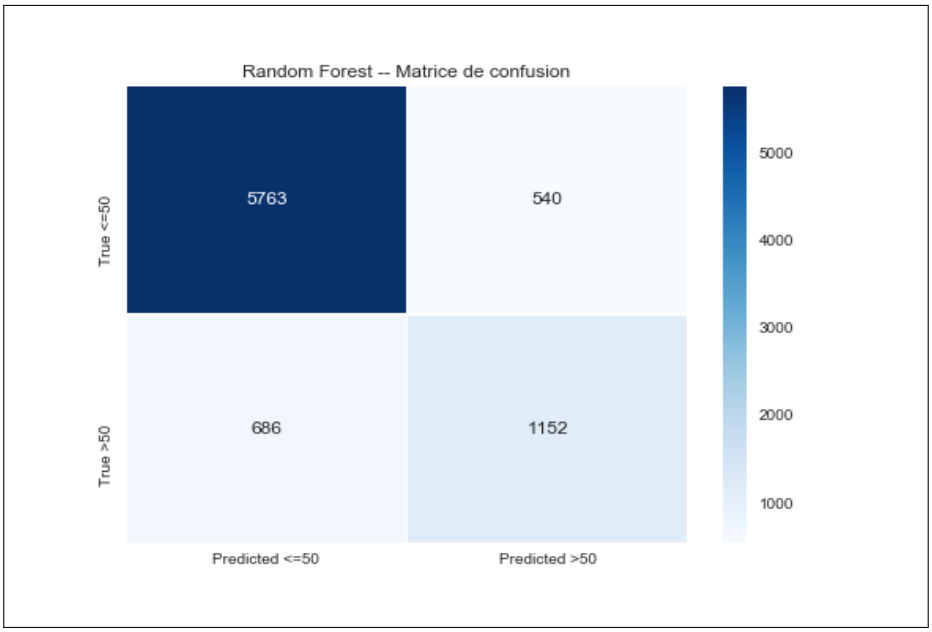

FIGURE 4.7 – Matrice de confusion – Random Forest

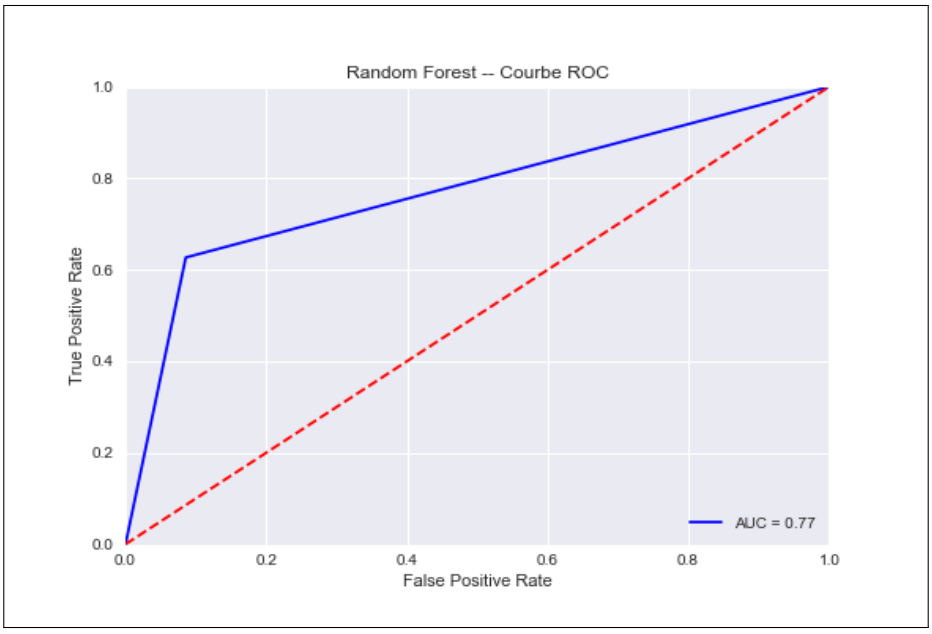

FIGURE 4.8 – Courbe ROC – Random Forest

4.5 Comparaison

Nous avons donc fait tourner 4 modèles de classification. Le tableau ci-dessous récapitule les performances de nos modèles.

Modèles	Précision	AUC
K-plus-proches-voisins	85.2%	0.75
Régression Logistique	84.7%	0.75
Arbre de décision	86.3%	0.77
Random Forest	84.9%	0.77

Il semble que le modèle le plus performant soit l'arbre de décision. Cela paraît logique car il s'agit d'une simple classification binaire et la base de données n'est pas extrêmement grande. Le modèles des K-plus-proches-voisins utilisés avec l'algorithme BallTree est aussi performant.

A l'inverse, il faut un grand nombre d'arbres pour construire une random forest performante. cela peut-être du à la complexité du modèle par rapport à la simplicité des données.