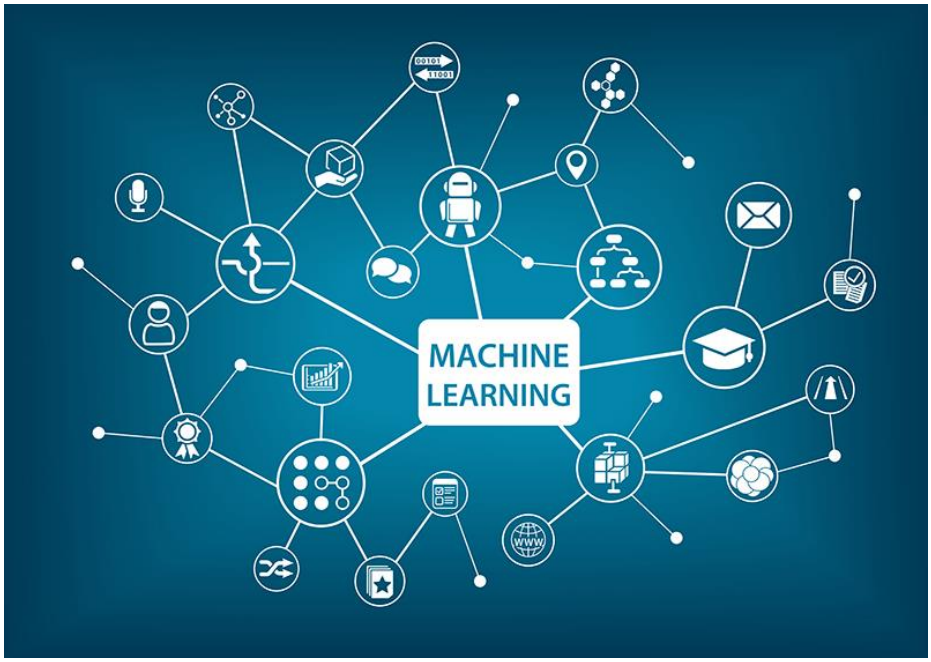
Issa KABORE
Mathias MICHEL
Teywa OZIOL
Nicolas SAVE
Alexis TOUZALIN

POLYTECH CLERMONT-FERRAND

Apprentissage Statistique

Réalisation d'une segmentation de la clientèle et création d'un score d'appétence



GMM 4A 2019/2020

Table des matières

Table des illustrations

Introduction :

Dans le cadre du cours d'apprentissage statistique, nous nous intéressons aux algorithmes de **machine learning**.

Le machine learning fait partie intégrante de **l'intelligence artificielle**. L'objectif est de permettre à l'ordinateur d'apprendre sans avoir été programmé explicitement. Le machine learning permet de répondre à de nombreuses problématiques comme la **prédiction** qu'un client va quitter sa banque ou celui qui va changer d'assurance vie. En plus du domaine du marketing, on retrouve le machine learning dans le domaine de la santé, de la finance ou encore du sport. Pour pouvoir apprendre par lui-même, il faut que l'on fournisse à la machine de nombreuses données issues de bases de données volumineuses. Avant que l'ordinateur ait la capacité d'analyser ces données, il faut au préalable qu'elles soient traitées et nettoyées afin que la machine soit en mesure de les comprendre, c'est le **preprocessing**. Pour les analyser, la machine s'appuie sur des **algorithmes**. Il existe de nombreux algorithmes et il faut choisir les bons en fonction des données initiales et des résultats que l'on souhaite obtenir.

Ici, nous verrons les **techniques de preprocessing** ainsi que plusieurs algorithmes utilisés dans le domaine du machine learning. Nous présenterons ces algorithmes et en choisirons certains que nous programmerons en Python, en fonction des résultats que l'on souhaite obtenir. Nous aborderons les deux grandes techniques de l'apprentissage statistique : **l'apprentissage supervisé et non supervisé**.

Pour utiliser tous ces algorithmes, nous travaillerons sur un **dataset** qui a été utilisé lors d'un concours de datamining au début du XX^{ème} siècle. Ce dataset est celui d'une compagnie d'assurance Hollandaise et comporte plus de **5800 individus** et plus de **80 variables** réparties en variables démographiques et en produits d'assurance.

I – Réalisation d'une segmentation

A. Présentation de la segmentation et des techniques utilisées

Dans un premier temps, nous allons réaliser une **segmentation**. La segmentation est un procédé permettant de créer des **groupes homogènes** d'individus présentant des caractéristiques communes. L'objectif d'une segmentation est d'être **plus performant** et **pertinent** en s'adaptant à chaque groupe.

Pour réaliser la segmentation, nous avons le choix entre de l'apprentissage supervisé et de l'apprentissage non supervisé :

Dans le premier cas (**l'apprentissage supervisé**), il faut **guider l'algorithme** pour qu'il apprenne par lui-même en lui fournissant des exemples qu'il estime probants après les avoir préalablement étiquetés des résultats attendus. L'intelligence artificielle apprend alors de chaque exemple en ajustant ses paramètres de façon à **diminuer l'écart entre le résultat obtenu et le résultat attendu**. La marge d'erreur se réduit ainsi au fil des entraînements, avec pour but, d'être capable de généraliser son apprentissage à de nouveaux cas.

Dans le cas de **l'apprentissage non supervisé**, l'apprentissage par la machine se fait de façon **totalement autonome**. Des données sont alors communiquées à la machine sans lui fournir les exemples de résultats attendus en sortie.
Si cette solution semble idéale sur le papier car elle **ne nécessite pas de grands jeux de données étiquetés** (dont les résultats attendus sont connus et communiqués à l'algorithme), il est important de comprendre que ces deux types d'apprentissages ne sont, par nature, **pas adaptés aux mêmes types de situations**.

C'est pour cela que **dans notre cas**, nous partons vers un **apprentissage non supervisé** car nous ne prenons pas en compte les étiquettes présentes dans notre dataset.

On choisit **dans notre cas** d'étudier **les variables socio-démographiques** car nous pensons qu'il sera plus simple de faire ressortir des similarités entre les variables.

Au cours de nos recherches, nous avons trouvé **trois algorithmes** pour réaliser une segmentation : **CAH, K-means et T-SNE**.

1. CAH

Les étapes de cet algorithme sont :

- D'abord on commence par **calculer la différence** entre les N objets.
- Ensuite on **regroupe deux objets dont le regroupement minimise un critère d'agrégation** donné, créant ainsi une classe comprenant ces deux objets. On continue jusqu'à ce que tous les objets soient regroupés.
- Puis on obtient **un arbre binaire de classification (dendrogramme)** qui représente une hiérarchie de partitions.
- Enfin on peut alors **choisir une partition en tronquant l'arbre à un niveau donné**, celui-ci dépendant soit des contraintes de l'utilisateur (l'utilisateur sait combien de classes il veut obtenir), soit de critères plus objectifs.

Les avantages de la CAH sont :

- La possibilité de travailler à partir des différences entre les objets que l'on veut regrouper. On peut donc **choisir un type de différence adapté** au sujet étudié et à la nature des données.
- La possibilité d'avoir **un dendrogramme**, qui permet de **visualiser le regroupement progressif des données**. On peut alors se faire une idée d'un nombre adéquat de classes dans lesquelles les données peuvent être regroupées. Par exemple le dendrogramme ci-dessous illustre la présence de 5 groupes.

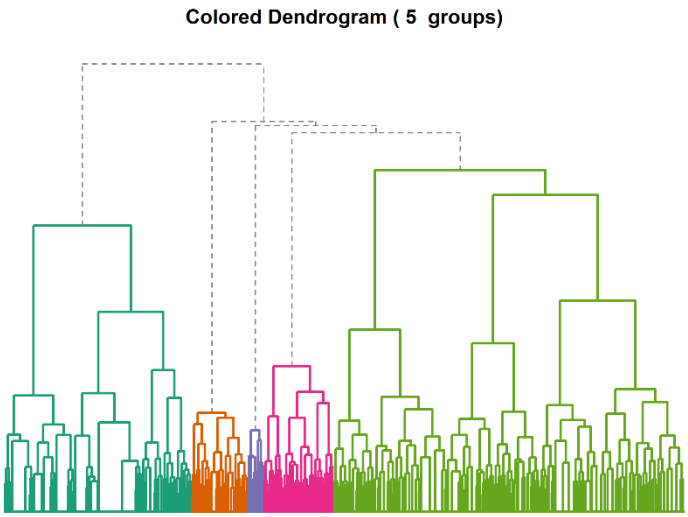


Figure 1 : Exemple d'un dendrogramme à 5 classes

D'un autre côté, **la taille du dataset constitue un inconvénient** pour cet algorithme. En effet, **la CAH ne convient pas pour un grand dataset** (nombre d'individus supérieur à 1000).

2. K-means

Il s'agit d'un **algorithme non supervisé de clustering non hiérarchique** qui permet de regrouper en clusters distincts les observations d'un jeu de données. Les données « similaires » se retrouveront dans un même cluster.

Les **avantages** de la K-means sont :

- Faible coût de calcul
- Simple à comprendre
- Correspond à un grand jeu de données
- Clusters proches (Clusters plus serrés par rapport aux algorithmes hiérarchiques)

Les **inconvénients** :

- Problèmes de prédiction de la valeur K
- Fonctionne sous certaines conditions (condition de clusters sphériques)

Le déroulement de l'algorithme est :

Etape 1 : On choisit au hasard K clients

Etape 2 : On affecte chaque individu au centre le plus proche

Figure 2 : Explication Etape 2 K-means

Etape 3 : On recalcule le centre de gravité du groupe et on arrête lorsque les individus ne changent plus de groupe

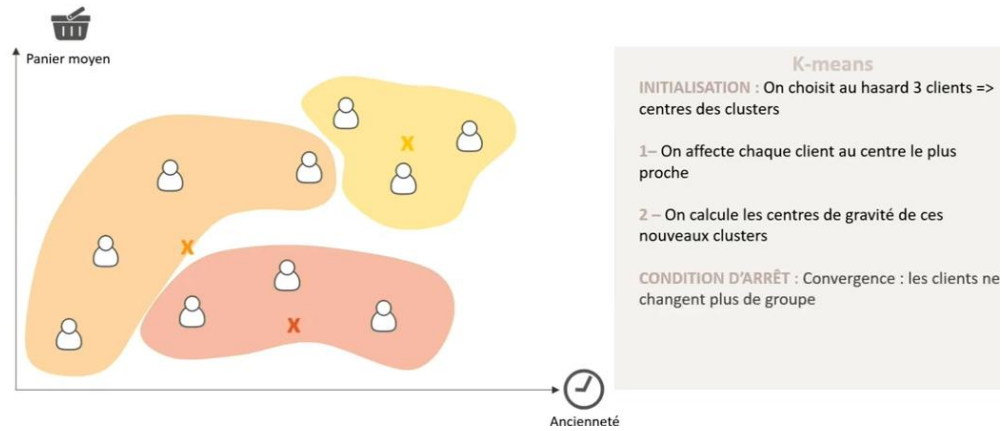


Figure 3 : Explication Etape 3 K-means

3. T-SNE

L'algorithme T-distributed Stochastic Neighbor Embedding (t-SNE) est **un algorithme non linéaire non supervisé**. Il permet de trouver une représentation fidèle de points issus d'un espace de grande dimension dans un espace de plus petite dimension.

L'algorithme commence par convertir les distances euclidiennes de grande dimension entre les points de données en probabilités conditionnelles. Le T-SNE est **basé sur une vision probabiliste des distances** entre les observations c'est à dire : si dans l'espace de base les données sont proches alors dans le nouvel espace il y aura une probabilité élevée d'avoir des représentations proches. À l'inverse si les données sont éloignées, la probabilité sera faible.

Les **avantages** de cet algorithme :

- Le T-SNE peut s'avérer salutaire pour distinguer des patterns pouvant être dissimulés par des caractéristiques particulières de densité et de dispersions d'un dataset
- La possibilité de créer une distribution qui respecte la proximité entre les objets les plus proches

Les **inconvénients** :

- Gourmand en temps de calcul
- Possède un paramètre clé : la perplexité qui représente le nombre optimal de voisins

B. Présentation des méthodes retenues et choix réalisés

Nous avons déjà vu que nous réalisons un **algorithme d'apprentissage non supervisé** pour réaliser la segmentation et que nous réaliserons **la segmentation sur les variables socio-démographiques**.

Nous avons décidé d'éliminer dans un premier temps la classification ascendante hiérarchique (CAH) car le nombre de données est trop important (+ 5000 données)

L'algorithme T-SNE semble être complexe à mettre en œuvre. Nous réaliserons donc l'algorithme des K-means.

Une fois l'algorithme réalisé, nous chercherons à caractériser les groupes obtenus.

Nous réaliserons ensuite une **analyse factorielle discriminante** afin de vérifier si **les groupes sont bien distincts et pour identifier quelles sont les caractéristiques des groupes** sur la base des variables socio-démographiques.

On a d'abord décidé d'effectuer **une K-means en sélectionnant 4 groupes**. On a d'abord modifié la valeur de K et on s'est arrêté sur 4 car cela nous semble le mieux pour la segmentation.

Une fois chaque individu relié à un groupe, on ajoute la variable groupe à notre base de données. Nous isolons chaque groupe dans une base de données différente avec la commande b0 = B [B ["gr"] == 0] en changeant l'égalité.

Nous calculons ensuite **la moyenne de chaque variable pour toutes les bases de données**, celles-ci correspondantes aux différents groupes. Une fois ces moyennes calculées, nous réunissons toutes les moyennes de chaque groupe dans **un seul Dataframe**.

Afin d'enlever certaines variables nous avons fait le choix de seulement garder les variables dont **la variance était supérieure à 1.5** car c'est avec cette valeur, nous gardons une dizaine de variables ce qui correspond environ aux nombres de variables à garder avec l'histogramme.

Une fois toutes ces variables éliminées, **nous avons pu dessiner un radar**.

Nous effectuons ensuite l'**AFD** afin de pouvoir **calculer les coefficients de corrélation** entre les trois nouveaux axes créés et les variables d'origine. Ceci nous permet de voir quelles sont **les variables qui caractérisent les trois axes** et ainsi voir quelles sont les variables qui sont déterminantes et celle qui le sont moins pour définir les individus. Nous pourrons ensuite comparer si nous trouvons les mêmes variables que celles du radar.

C. Résultats obtenus

Au niveau des résultats obtenus, **chaque individu est placé dans un des quatre groupes** grâce à l'algorithme de segmentation des K-means. Nous avons ensuite :

- **Groupe 1 :** C'est un type de client entre 5 et 6, entre 40 et 50% touchent le revenu_1, ce groupe a plus tendance à souscrire à une assurance santé public, sont locataires, sont peu mariés et environ 45% d'entre eux ont un niveau d'étude bas, ils ont un pouvoir d'achat assez faible
- **Groupe 2 :** C'est un type de client entre 2 et 3, moins de 20% touchent le revenu 1, 50% d'entre eux souscrivent à une assurance santé publique, sont en général propriétaires, mariés et très peu d'entre eux ont un niveau d'étude bas, ils ont un pouvoir d'achat assez élevé
- **Groupe 3 :** C'est un type de client entre 6 et 7, très peu d'entre eux touchent le revenu 1, ont tendance à souscrire une assurance de santé privé, sont plus souvent propriétaires, 45% d'entre eux ont un niveau d'étude bas, environ 70% d'entre eux sont mariés et possèdent un pouvoir d'achat moyen
- **Groupe 4 :** Type de client entre 7 et 8, ont un pouvoir d'achat moyen, la plupart d'entre eux souscrivent à une assurance santé public, sont en général locataires, plus de 60% d'entre eux ont un niveau d'étude bas, sont plus souvent mariés

Avec l'AFD, on voit que **les variables les plus déterminantes dans la création des axes sont celles que l'on retrouve dans la construction du radar et la caractérisation des groupes**. Cependant, le revenu_moyen est important dans l'AFD et on ne le retrouve pas dans le radar.

II – Transformation des données et preprocessing

Le **preprocesssing** consiste à **traiter au préalable les données pour améliorer les performances de nos algorithmes**. Le but est de transformer les données en un format compréhensible pour la machine et pour qu'elle n'effectue pas de contre sens.

Le **preprocessing** est effectué juste après la phase de nettoyage des données et avant la phase de traitement des données

A. Présentation des méthodes de preprocessing

Il existe **différentes méthodes** pour réaliser l'étape du preprocessing. Nous allons présenter les plus couramment utilisées.

Tout d'abord, il faut **traiter les valeurs manquantes**. Des traitements différents sont nécessaires selon le type des variables. Pour les **variables numériques**, on peut les remplacer par la moyenne, la médiane ou encore un tirage aléatoire dans la distribution empirique (discrète). Quant aux **variables catégorielles** on peut les remplacer par le mode, la médiane (uniquement s'il existe une relation d'ordre) ou un tirage aléatoire dans la distribution empirique (multinomiale). Mais il existe un autre traitement qui consisterait à faire de la **« prédiction inversée »** des variables manquantes par les autres variables. Celui-ci **nécessite plus de temps de calcul** et ne sera donc pas utilisé.

Ensuite, il y a **l'encodage** qui consiste à **présenter à la machine des valeurs numériques adaptées**. Pour cela, il faut donc transformer des données si elles sont dans le mauvais format. Il existe deux types d'encodage :

- **L'encodage ordinal** qui vise à associer à chaque classe d'une variable une valeur unique. **La fonction en Python** réalisant cette opération est une fonction de la bibliothèque Scikit-Sklearn : LabelEncoder(). Mais l'encodage ordinal a un **principal inconvénient** : il affecte un nombre à chaque modalité et crée une variable, par exemple sur l'exemple ci-dessous, le chien codé « 1 » serait considéré par l'algorithme comme « supérieur » au chat codé « 0 ». Pour résoudre ce problème, il existe l'encodage One Hot.

Figure 4 : Encodage ordinal

- **L'encodage One Hot** consiste à créer n variables binaires avec n qui est le nombre de modalités de la variable de base (cf. figure ci-dessous). **Les fonctions en Python** permettant de réaliser cet encodage sont deux fonctions de la bibliothèque Scikit-Sklearn : LabelBinarizer(), OneHotEncoder(). Mais cet encodage présente aussi un **inconvénient**, en effet, s'il y a trop de modalités, cela conduira à la création d'un grand nombre de nouvelles variables. Mais si on analyse ces variables, il n'y aura qu'un seul « 1 » par ligne, et donc beaucoup plus de « 0 ». Pour contourner ce problème il existe **l'encodage sparse** qui consiste à coder uniquement les « 1 » présents dans la matrice. **Il existe une fonction Python** réalisant cet encodage, sklearn.decomposition.sparse_encode.

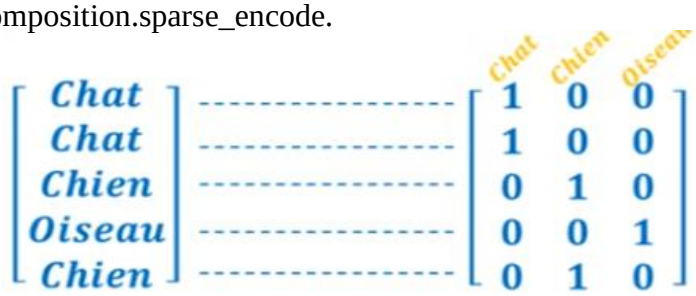


Figure 5 : Encodage One Hot

Un autre type de preprocessing est **la normalisation**. Cette méthode consiste à mettre sur une même échelle les données quantitatives pour faciliter l'entraînement du modèle (Exemple : algo de descente de gradient plus performant). Il faut **transformer chaque variable de telle sorte qu'elle soit comprise entre 0 et 1**. Le principal avantage est qu'on conserve le rapport, les distances entre les variables, et nous n'avons aucune perte d'information. **La fonction en Python** réalisant cette normalisation est une fonction Scikit-Sklearn : MinMaxScaler(). Le gros problème intervient si le max ou le min sont des valeurs absurdes, l'analyse s'en retrouve faussée.

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Il existe également **la technique de la standardisation**. Il existe **deux types** de standardisation. La première consiste à **mettre sur une même échelle les données quantitatives** pour faciliter l'entraînement du modèle. Il faut transformer chaque variable de telle sorte que **la moyenne soit nulle et l'écart type égal à 1, c'est un centrage-réduction**. Le principal **avantage** est qu'on conserve le rapport, les distances entre les variables, et nous n'avons aucune perte d'information. **La fonction en Python** réalisant cette normalisation est une fonction Scikit-Sklearn : StandartScaler(). **L'inconvénient** est que s'il y a des valeurs absurdes, l'analyse est faussée car la moyenne et l'écart-type faussés.

$$X_{scaled} = \frac{X - \mu_X}{\sigma_X}$$

Le deuxième type de standardisation a le même objectif mais cette technique est très peu sensible aux outliers, les valeurs extrêmes. **La fonction en Python** réalisant cette normalisation est une fonction Scikit-Sklearn : RobustScaler(). Le **principal avantage** est que la méthode est peu sensible aux valeurs extrêmes puisque la médiane et les quartiles sont peu sensibles.

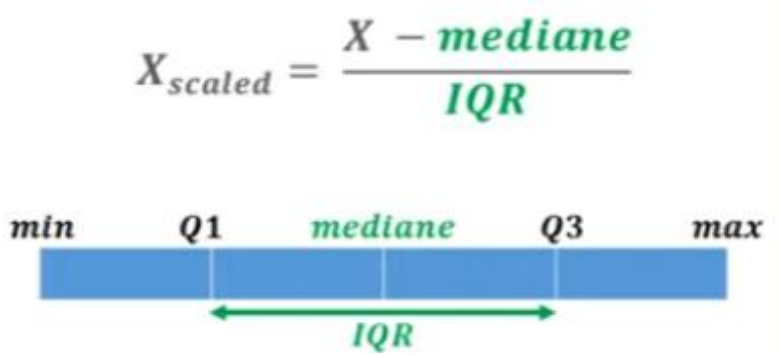


Figure 6 : Schéma deuxième type de stantardisation

Une autre méthode s'appelle **la régression polynomiale**. La méthode "classique" pour procéder à une régression polynomiale consiste à créer un tableau dont chaque colonne va correspondre à un degré polynomial. Puis procéder à une régression pour chaque colonne. Une méthode plus rapide consiste à utiliser **le module sklearn.preprocessing.PolynomialFeatures** qui va générer automatiquement les données nécessaires : polyn = PolynomialFeatures(degree=d). Ensuite, toujours avec **scikit-learn**, il suffit de créer et **fit le modèle de régression** : clf = linear_model.LinearRegression(). Pour finir, il faut réduire les coefficients par la méthode **« Ridge »** (qui permet de réduire les coefficients sans les annuler, grâce à une régularisation de type L2) ou la méthode **« Lasso »** (couramment utilisée pour effectuer une sélection de variables : plus le coefficient Lasso est élevé et moins il restera de variables dans le modèle).

Il existe également **deux méthodes de transformations non linéaires : la transformation de puissance et la normalisation des quantiles**. En statistique, **une transformation de puissance** est une famille de fonctions qui sont appliquées pour **créer une transformation monotone des données à l'aide de fonctions de puissance**. C'est une technique de transformation de données utile utilisée pour stabiliser la variance, rendre les données plus proches de la distribution normale, améliorer la validité des mesures d'association telles que la corrélation de Pearson entre les variables et pour d'autres procédures de stabilisation des données. **La fonction Python** associée est : sklearn.preprocessing.PowerTransformer.

En statistique, **la normalisation des quantiles** est une technique permettant de **rendre deux distributions identiques en termes de propriétés statistiques**. Pour normaliser par quantile une distribution de test par rapport à une distribution de référence de même longueur, il faut trier la distribution de test et la distribution de référence. L'entrée la plus élevée dans la distribution du test prend alors la valeur de l'entrée la plus élevée dans la distribution de référence, l'entrée suivante la plus élevée dans la distribution de référence, et ainsi de suite, jusqu'à ce que la distribution du test soit une perturbation de la distribution de référence. En général, une distribution de référence sera une des distributions statistiques standard telles que la distribution gaussienne ou la distribution de Poisson. **En Python**, la fonction associée est : sklearn.preprocessing.quantile_transform.

Enfin il reste **la discrétisation**, discrétiser une variable quantitative c'est, mathématiquement**, transformer un vecteur de nombres réels en un vecteur de nombres entiers** nommés "indices de classe". C'est pourquoi cette effectuer cette transformation se dit en langage courant "réaliser un découpage en classes". En statistiques, discrétiser c'est à la fois réaliser cette transformation mathématique, nommer et justifier les classes. Pour réaliser une discrétisation, il faut choisir **le nombre de classes et les bornes de classe**. Pour réaliser une bonne discrétisation, il faut justifier à la fois le nombre de classes et les bornes de classe, le terme "bonne" faisant référence à des critères explicitement définis. Intuitivement, un bon découpage correspond à des classes homogènes et séparées, ce qui correspond respectivement aux notions statistiques de faible variance intraclasse et de forte variance interclasse. Mais d'autres critères sont possibles, comme l'équirépartition, le respect d'un nombre minimal de données par classe etc.

Il existe quelques formules "toute faites" pour **déterminer à l'aveugle le nombre n de classes à partir du nombre N de données** :

- Brooks-Carruthers 5*log(N,base=10)
- Huntsberger 1 + 3,332*log(N,base=10)
- Sturges log(N+1,base=2)

Deux autres formules, censées être plus précises, mettent en jeu le minimum a des données et le maximum b et utilisent aussi d'autres paramètres de la dispersion : sig, l'écart-type et eiq l'écart inter-quartiles :

- Scott (b-a)/(3.5*sig*N**(-1/3))
- Freedman-Diaconis (b-a)/(2*eiq*N**(-1/3))

Il faut ensuite **choisir les bornes des classes**, il existe différentes méthodes. **La méthode des quantiles** (le critère visé est l'équirépartition, c'est à dire le même nombre de données par classe), **la méthode des amplitudes** (on garantit ici que le critère d'égalité d'amplitude de classe est respecté, l'amplitude étant la différence entre la plus grande valeur et la plus petite valeur), **la méthode des moyennes emboitées** (le nombre de classes est ici une puissance de deux. On sépare l'intervalle de départ en deux en prenant comme valeur de séparation la moyenne globales des valeurs) et **la méthode des grandes différences relatives** (on trie les valeurs par ordre croissant puis on calcule les différences relatives successives entre une valeur et sa suivante. On change de classe lorsque la différence relative est supérieure à un seuil arbitraire, classiquement 50 %). **Les fonctions Python** sont : sklearn.preprocessing.Binarizer.

B. Sélection de variables

La sélection de variables est une technique qui consiste à **choisir les variables dans nos données qui contribuent le plus à la variable cible**. En d'autres termes, nous choisissons les meilleurs prédicteurs pour la variable cible.

Les classes du module **sklearn.feature_selection** peuvent être utilisées pour la sélection de variables/réduction de la dimensionnalité sur des ensembles d'échantillons, soit pour améliorer les scores de précision des estimateurs, soit pour stimuler leurs performances sur des ensembles de données à très haute dimension.

Les variables numériques et catégorielles doivent être traitées différemment.

La sélection peut être effectuée de plusieurs façons, mais il existe globalement **trois stratégies de base** :

- Les statistiques univariées
- La sélection basée sur le modèle
- La sélection itérative

1. Les statistiques univariées

La sélection univariée de variables fonctionne en **sélectionnant les meilleures caractéristiques sur la base de tests statistiques univariés**. Elle peut être considérée comme une étape de prétraitement pour un estimateur. Le module **Scikit-learn** en Python propose des fonctions de sélection de variables comme :

SelectKBest qui supprime toutes les caractéristiques sauf celles qui obtiennent les meilleurs scores

SelectPercentile qui supprime toutes les caractéristiques, sauf un pourcentage de points le plus élevé spécifié par l'utilisateur en utilisant des tests statistiques univariés communs pour

chaque variable : taux de faux positifs **SelectFpr** ou encore taux de fausses découvertes **SelectFdr** par exemple.

GenericUnivariateSelect qui permet d'effectuer une sélection univariée des caractéristiques à l'aide d'une stratégie configurable. Cela permet de sélectionner la meilleure stratégie de sélection univariée avec un estimateur de recherche hyperparamétrique.

2. La sélection basée sur le modèle

Cette stratégie **utilise un modèle d'apprentissage supervisé** pour juger de l'importance de chaque variable, pas forcément le même modèle que celui utilisé lors de la modélisation finale. Elle doit fournir une mesure de l'importance de chaque variable pour pouvoir effectuer un classement en fonction de cette mesure. Nous avons comme méthodes utilisables : **les arbres de décision, les modèles de décision arborescents et les modèles linéaires**.

Le principe de **l'arbre de décision** est de choisir pour chaque nœud la variable qui, par ses valeurs, sépare le mieux les individus en fonction des catégories de la variable cible. On itère le processus jusqu'à ce que la scission ne soit plus possible ou plus souhaitable. **Les principaux algorithmes de construction** de l'arbre de décision sont :

- **L'algorithme de Hunt**, dont les autres algorithmes sont inspirés
- **CHAID** : uniquement pour des variables explicatives qualitatives ou discrètes, basé sur le critère du $\chi 2$
- **CART** (Classification And Regression Tree), sépare chaque nœud en deux
- **ID3** (Iterative Dichotomizer 3)
- **C4.5, C5** versions améliorées respectivement de ID3 et C4.5
- **SLIQ** (Supervised Learning In Quest), SPRINT, pour les grandes quantités de données

Les modèles linéaires pénalisés par la norme L1 ont des solutions « sparses » : beaucoup de leurs coefficients estimés sont nuls. Lorsque l'objectif est de réduire la dimension des données pour les utiliser avec un autre classificateur, ils peuvent être utilisés en même temps que feature_selection.SelectFromModel pour sélectionner les coefficients non nuls. En particulier, les estimateurs rares utiles à cette fin sont **le modèle linéaire Lasso** pour la régression, **le modèle linéaire LogisticRegression** et **le svm LinearSVC** pour la classification :

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> X_new.shape
(150, 3)
```

Figure 7 : Exemple de programme utilisant LinearSVC

Avec les SVM et la régression logistique, le paramètre C contrôle la rareté : plus le C est petit, moins il y a de caractéristiques sélectionnées. Avec le lasso, plus le paramètre alpha est élevé, moins il y a de caractéristiques sélectionnées.

Les estimateurs basés sur **les arbres** (voir le module **sklearn.tree** et la forêt d'arbres dans le module **sklearn.ensemble**) peuvent être utilisés pour **calculer l'importance des variables**, qui à leur tour peuvent être utilisées pour éliminer les caractéristiques non pertinentes.

3. La sélection itérative de caractéristiques

Il y a **deux méthodes de base** : soit **commencer sans variable**, les ajouter une à une jusqu'à ce qu'un critère d'arrêt soit atteint, soit **commencer avec toutes les variables**, les supprimer une à une jusqu'à ce qu'un critère d'arrêt soit atteint.

L'élimination récursive des caractéristiques (RFE) consiste à sélectionner des caractéristiques en considérant de manière récursive des ensembles de caractéristiques de plus en plus petits. Tout d'abord, l'estimateur est formé sur l'ensemble initial de caractéristiques et l'importance de chaque caractéristique est obtenue soit par **un attribut coef_** soit par **un attribut feature_importances_**. Cette procédure est répétée de manière récursive sur l'ensemble jusqu'à ce que le nombre souhaité de caractéristiques à sélectionner soit finalement atteint.

4. Autres méthodes

Nous avons **les méthodes de filtrage** qui sélectionnent les variables indépendamment du modèle. Comme son nom l'indique, dans cette méthode, vous filtrez et ne prenez que le sous-ensemble des variables pertinentes. Le modèle est construit après avoir sélectionné les caractéristiques. Le filtrage est ici effectué à l'aide d'une matrice de corrélation et il est le plus souvent effectué à l'aide de la **corrélation de Pearson**.

Le coefficient de corrélation a des valeurs comprises entre -1 et 1 :

- Une valeur plus proche de 0 implique une corrélation plus faible (0 exact impliquant une absence de corrélation)
- Une valeur plus proche de 1 implique une corrélation positive plus forte
- Une valeur plus proche de -1 implique une corrélation négative plus forte

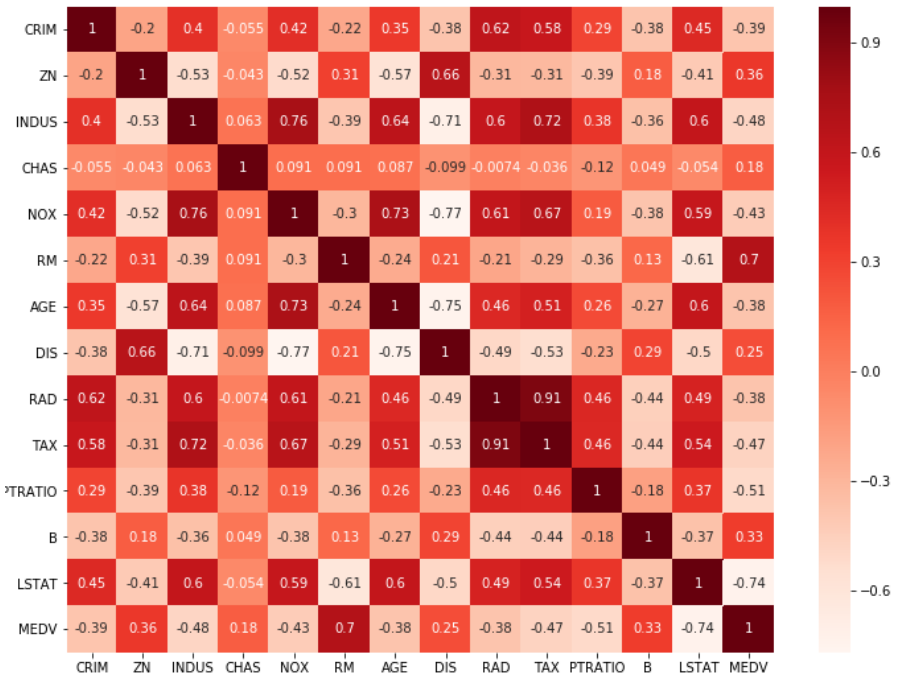


Figure 8 : Exemple d'une matrice de corrélation

Une autre méthode serait la **VarianceThreshold**. C'est une approche de base simple pour la sélection de variables. Elle permet de **supprimer toutes les caractéristiques dont la variance n'atteint pas un certain seuil**. Par défaut, elle supprime toutes les caractéristiques à variance nulle, c'est-à-dire les caractéristiques qui ont la même valeur dans tous les échantillons.

À titre d'exemple, supposons que nous ayons un ensemble de données avec des caractéristiques booléennes, et que nous voulions supprimer toutes les caractéristiques qui sont soit un soit zéro (activées ou désactivées) dans plus de 80% des échantillons. Les caractéristiques booléennes sont des variables aléatoires de Bernoulli, et la variance de ces variables est donnée par :

$$\mathrm{Var}[X] = p(1-p)$$

Donc nous pouvons sélectionner en utilisant le seuil 0,8 * (1 - 0,8).

C. Nos choix pour la préparation de notre dataset

Pour la préparation de notre dataset, nous avons utilisé plusieurs méthodes parmi celle ci-dessus.

Pour notre jeu de 45 variables :

On a utilisé **des arbres de régression** pour pouvoir supprimer des variables. En effet, les variables qui ne donnaient pas de feuilles sont supprimées. Une feuille est créée s'il y a au moins 20 individus sur la feuille. On a supprimé 4 variables, ce qui signifie que 4 arbres sur les 80 variables ont une seule feuille.

On a ensuite effectué **un second tri en traçant les histogrammes** pour chaque variable. S'il y a très peu de valeurs pour une modalité et qu'il y a que deux modalités (notamment les produits d'assurance), on supprime la variable.

Ensuite, **on regarde la corrélation entre les variables deux à deux**. Pour cela on a tracé deux graphiques différents. Un premier permettant de visualiser où sont approximativement les corrélations. C'est très utile pour avoir une idée globale. On a pu ainsi voir que les dernières variables sont très corrélées. Le deuxième graphique affiche à la fois la couleur et la valeur de la corrélation. On peut plus vite les repérer. On a ainsi supprimé un certain nombre de variables jusqu'à arriver à 45. Parmi deux variables corrélées, on n'a pas choisi au hasard laquelle supprimer mais on ne s'appuie pas sur une technique. On a sélectionné celle qu'on pense qui sera le mieux. La limite de corrélation est 0,74.

Sur ces 45 variables on procède à un **regroupement de modalités** pour que l'algorithme soit plus efficace. Si les queues de distribution ne sont pas « jolies », on fait un arbre sur la variable et on procède à un regroupement des modalités.

La dernière opération que l'on fait, pour préparer les données pour les algorithmes est **une standardisation de l'ensemble des variables restantes**, puisque les deux algorithmes que l'on souhaite appliquer (SVM et Régression Logistique) ont besoin de variables standardisées.

Voici tout d'abord la liste des 45 variables obtenues :

mt_auto
nb_bateau
mt_RC
pouvoir_achat
mt_incendie
mt_bateau
niv_etud_bas
revenu_moyen
niv_etude_haut
revenu1
proprietaire
auto0
auto1
type_client
nb_securite_soc
PCStop
revenu4
revenu3
assur_sante_prive
PCSouvr
PCSagri
celibataire
PCSinter
mt_cyclomoteur
PCSouvr_quali
sans_religion
nbpers_au_foyer
nb_accident_famil
concubin
mt_velo
nb_invalidite
PCScadre
nb_RC_agri
mt_assur_vie
nb_MRH
mt_remorque
mt_tracteur
nbmaisons
sans_enfant
auto2
catholique
mt_moto
autre_religion
revenu5
revenu2
nb_RC_entrepris

A partir de cette liste, nous créons d'autres jeux de variables avec une nouvelle méthode. Cette méthode consiste à regarder la corrélation entre les variables explicatives et la variable cible.

On peut donc en déduire un jeu de 10 variables :

mt_auto
nb_bateau
mt_RC
pouvoir_achat
mt_incendie
mt_bateau
niv_etud_bas
revenu_moyen
niv_etude_haut
revenu1

Un jeu de 20 variables :

mt_auto
nb_bateau
mt_RC
pouvoir_achat
mt_incendie
mt_bateau
niv_etud_bas
revenu_moyen
niv_etude_haut
revenu1
proprietaire
auto0
auto1
type_client
nb_securite_soc
PCStop
revenu4
revenu3
assur_sante_prive
PCSouvr

Et enfin un jeu de 30 variables :

mt_auto	proprietaire	PCSagri
nb_bateau	auto0	celibataire
mt_RC	auto1	PCSinter
pouvoir_achat	type_client	mt_cyclomoteur
mt_incendie	nb_securite_soc	PCSouvr_quali
mt_bateau	PCStop	sans_religion
niv_etud_bas	revenu4	nbpers_au_foyer
revenu_moyen	revenu3	nb_accident_famil
niv_etude_haut	assur_sante_prive	concubin
revenu1	PCSouvr	mt_velo

Ces trois jeux sont les 10, 20 ou 30 premières variables les plus corrélées avec la variable cible parmi les 45 variables restantes.

Pour finir, nous pensons que c'est avec le jeu de 10 variables ou de 20 variables que nous obtiendrons les meilleurs résultats. Le jeu avec 30 variables comporte peut-être trop de variables et **un phénomène de sur-apprentissage peut apparaître**. Nous pensons que les méthodes que nous avons utilisées précédemment pour la sélection de variables sont moins performantes que celle-ci.

III – Les classifieurs et score

Le rôle d'un **classifieur** est de **classer dans des groupes (des classes) les échantillons qui ont des propriétés similaires**, mesurées sur des observations.

L'objectif d'un **score** est **d'attribuer une note variante entre 0 et 1 à un individu**. Cette note représente une probabilité. En fonction des résultats, une entreprise appliquera une certaine stratégie pour vendre le produit à une certaine catégorie de la population. Dans notre cas on s'intéresse à la souscription ou non d'une assurance caravane.

Ici, nous allons tout d'abord **créer un modèle à partir d'une table d'entraînement**. Une fois le modèle créé on le fait **fonctionner avec de nouveau individus** (individus « test »). On obtient des scores pour chaque individu entre 0 et 1. Enfin, **on cherche un seuil** pour pouvoir classer entre 0 et 1 nos individus. On pourra comparer le résultat avec leurs étiquettes. On pourra chercher à optimiser le modèle pour mieux prédire les individus.

A. Arbres de décision

Un **arbre de décision** est un outil d'aide à la décision représentant un ensemble de choix sous la forme graphique d'un arbre. **Les différentes décisions possibles sont situées aux extrémités des branches** (les « feuilles » de l'arbre), et sont atteintes en fonction de décisions prises à chaque étape. Il a l'avantage d'être **lisible** et **rapide à exécuter**. Il s'agit de plus d'une représentation calculable automatiquement par des algorithmes d'apprentissage supervisé.

Un avantage majeur des arbres de décision est qu'ils peuvent être calculés automatiquement à partir de bases de données par des algorithmes d'apprentissage supervisé. Ces algorithmes sélectionnent automatiquement les variables discriminantes à partir de données non-structurées et potentiellement volumineuses.

Ils permettent de **traiter des données hétérogènes**, éventuellement manquantes, sans hypothèses de distribution. Ils permettent aussi de détecter des interactions et des phénomènes non linéaires.

En Python : clf = tree.DecisionTreeClassifier()

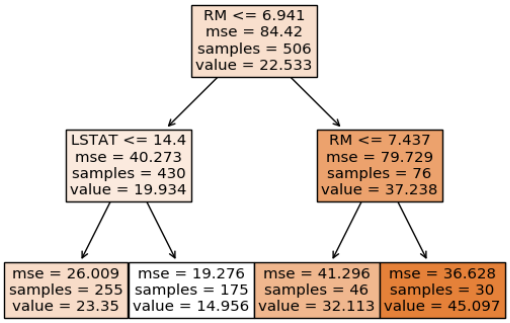


Figure 9 : Exemple d'un arbre de décision en Python

B. Méthode des K plus proches voisins (KNN)

C'est **une méthode d'apprentissage supervisé les plus simples de Machine Learning**. Dans ce cadre, on dispose d'une base de données d'apprentissage constituée de *N* couples « entrée-sortie ».

Pour estimer la sortie associée à une nouvelle entrée *x*, la méthode des *k* plus proches voisins consiste à prendre en compte (de façon identique) les *k* échantillons d'apprentissage dont l'entrée est la plus proche de la nouvelle entrée *x*, selon une distance à définir. Si ce sont des **données quantitatives (continues) : distance euclidienne**. Si ce sont des **données quantitatives : distance de Manhattan**, qui converge plus rapidement en grande dimension et n'accorde pas trop de crédits aux valeurs extrêmes. Et **si ce sont des données catégorielles : similarité de Jacquard**.

Il s'agit de **classer l'entrée dans la catégorie à laquelle appartient les k plus proches voisins** dans l'espace des caractéristiques identifiées par apprentissage.

Avantages :

- Algorithme très simple
- Ne nécessite pas de construire de modèle, d'ajuster des paramètres ou de faire des hypothèses supplémentaires
- Algorithme polyvalent : utilisable pour la classification, la régression et la recherche d'informations
- 2 paramètres : k et la distance

Inconvénients :

- Algorithme de plus en plus long quand le nombre de variables augmente
- Nécessite le stockage de la base d'apprentissage
- Mauvais dans les cas de « Sparse datasets », jeux de données majoritairement vides

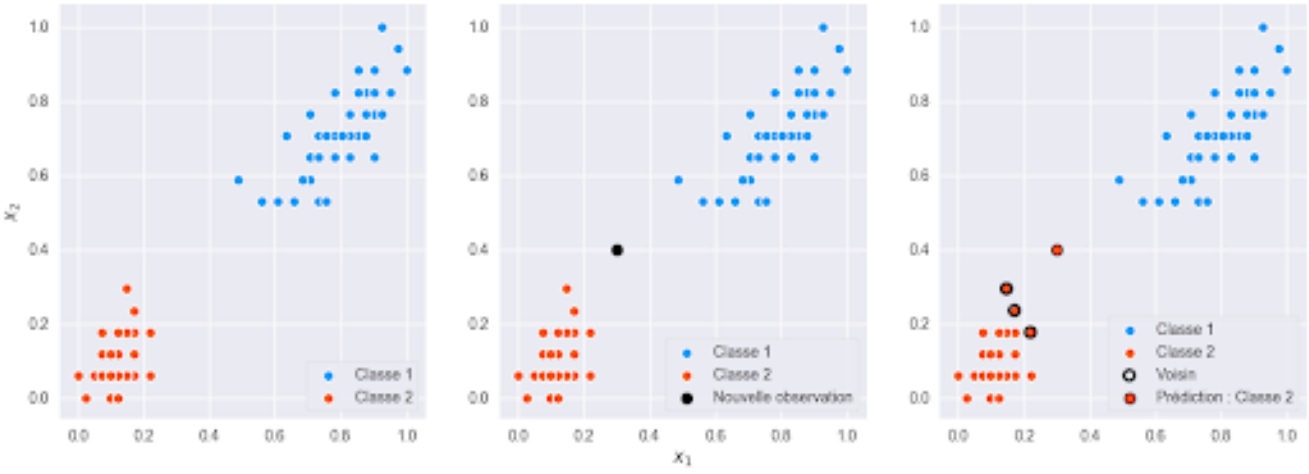


Figure 10 : Schéma de fonctionnement de la méthode KNN

C. SVM (Machines à vecteurs de supports)

C'est un algorithme **d'apprentissage supervisé très utilisé dans tous les domaines**. Il peut fonctionner et donner des résultats très cohérents avec moins de données que certains autres algorithmes.

Le principe est le suivant : **on dispose de deux groupes**, cependant, la frontière entre ces deux régions n'est pas connue. On n'arrive donc pas à bien les différencier. On souhaite **présenter un nouveau point dont on connait que la position dans le plan** et l'algorithme de classification sera capable de prédire si ce nouveau point est du groupe 1 ou du groupe 2.

Il faut donc **trouver la frontière** pour catégoriser le point. Pour que la méthode SVM puisse trouver cette frontière, **il faut lui donner des données d'entraînement**. En l'occurrence, on donne au SVM un ensemble de points, dont on sait déjà si ce sont des groupe 1 ou groupe 2.

Une fois la frontière trouvée, on peut affecter à un groupe, la nouvelle valeur.

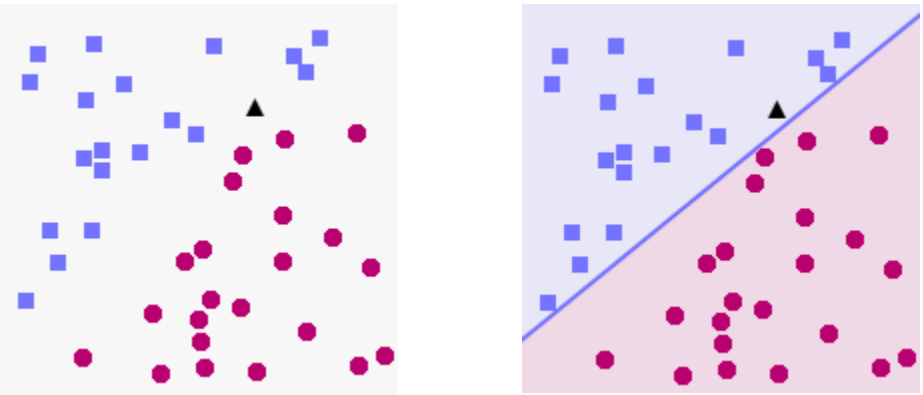


Figure 11 : Schéma expliquant la SVM

La principale **difficulté** pour l'algorithme est donc de trouver la frontière, comme la SVM est un classificateur linéaire, il cherche une droite pour séparer les points. La SVM va **placer la frontière aussi loin que possible des carrés bleus mais également aussi loin que possible des ronds roses**.

Figure 12 : Frontière optimale (à gauche) et non optimale (à droite)

Ci-dessus, à gauche, la frontière est optimale, à droite, elle ne l'est pas du tout. La frontière gauche à **une « meilleur capacité de généralisation ».** La frontière est dite optimale.

Si jamais il n'est pas possible de trouver un séparateur linéaire entre les données, l'algorithme va transformer les données de sorte que l'on puisse trouver une séparation linéaire. **On parle alors de kernel trick**.

Jusqu'ici nous avons pris des exemples sur un plan pour pouvoir comprendre la démarche. En réalité, on travaille souvent en dimension quelconque > 2. Le principe reste le même à part que **l'on ne cherche pas à tracer une droite comme frontière mais un hyperplan**. On obtiendra une équation h(x) de cet hyperplan. Pour chaque nouvelle donnée, on remplacera les coordonnées de la donnée xk et si on trouve que h(xk) > 0, on affecte la donnée à la première catégorie. Si h(xk) < 0, on affecte la donnée à la seconde catégorie.

Pour les cas multiclasses : on regarde à quelle catégorie le nouveau point à le plus de chance d'appartenir.

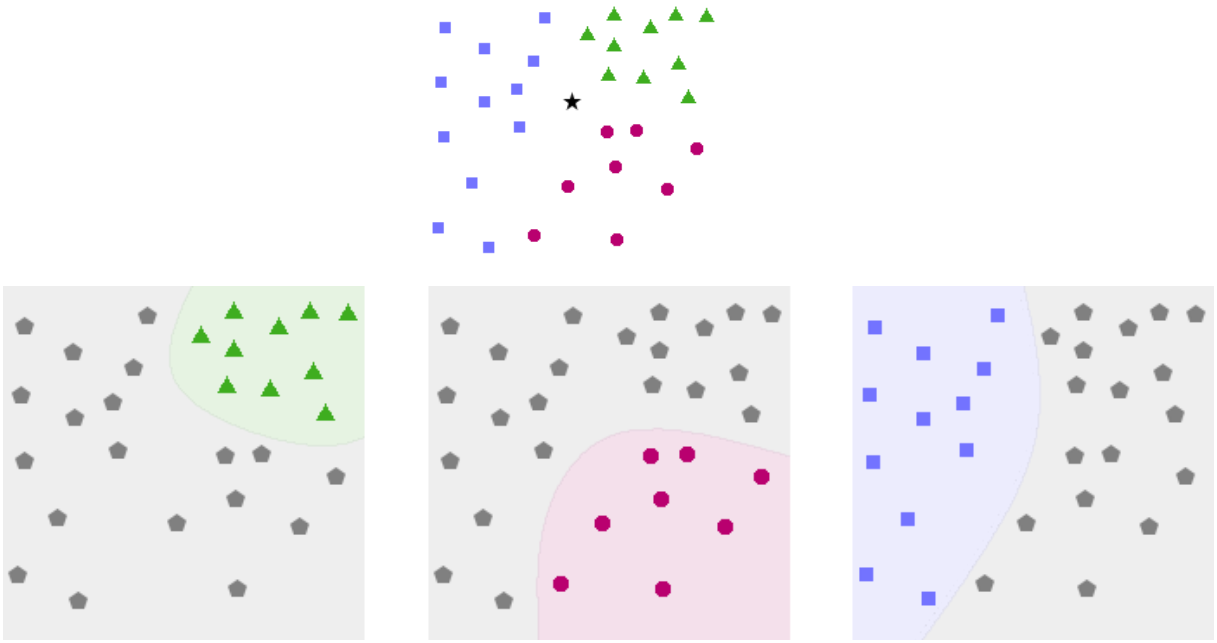


Figure 13 : Schéma présentant SVM en multiclasses

Sous **Python**, il existe une fonction qui est la suivante :

sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ties=False, random_state=None)

puis :

fit(self, X, y, sampl

e_weight=None)

Avantages :

- Très efficace en dimension élevée
- Efficace si la dimension de l'espace est plus grande que le nombre d'échantillons d'apprentissage
- Demande moins de mémoire que d'autres méthodes

Inconvénients :

- Si le nombre d'attributs est beaucoup plus grand que le nombre d'échantillons, les performances sont moins bonnes
- Comme il s'agit de méthodes de discrimination entre les classes, elles ne fournissent pas d'estimations de probabilités

D. Régression logistique

C'est **l'une des méthodes les plus utilisées en scoring**, les variables sont explicatives quantitatives ou qualitatives. La variable Y à prévoir est, elle, binaire. Le but est de **montrer une relation de dépendance** entre une variable à expliquer et une série de variables explicatives. Il s'agit d'une régression logistique car la loi de probabilité est modélisée à partir d'une loi logistique.

Sous Python : demo_clf(LogisticRegression(), X, y, proba=True)

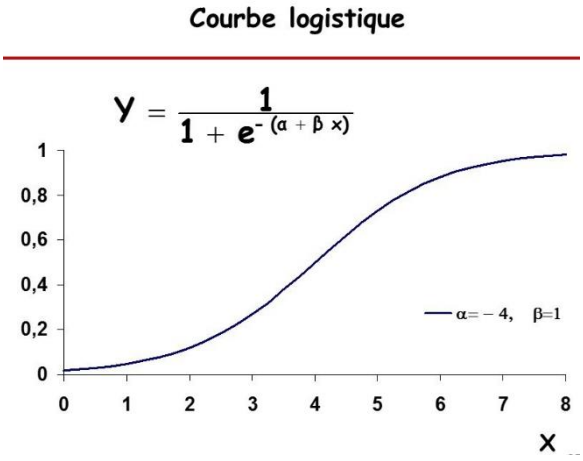


Figure 14 : Courbe de la fonction logistique utilisée pour cette régression

E. Bayésien naïf

C'est **un algorithme très populaire en Machine Learning**. Il est très utilisé dans les problématiques de classification de texte. Un exemple d'utilisation est celui du filtre anti-spam. Le classifieur naïf bayésien se base sur **le théorème de Bayes** qui est fondé sur les probabilités conditionnelles.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Cet algorithme est dit naïf car :

- Les features ou les propriétés sont indépendantes les unes des autres. Or c'est rarement le cas, si on prend les classifications de textes la présence d'un mot peut être lié à la présence d'un autre mot !
- On fait aussi le postulat que les features sont d'importances équivalentes dans la tâche de classification, ce qui n'est pas vraiment le cas. Pour certains textes la présence d'un mot ou d'un groupe de mot est très signifiant dans leur classe.

Il existe trois types de classifieurs naïfs bayésiens :

- **Naïf bayésien multinomial** : principalement utilisé pour les problèmes de classification des documents, c'est-à-dire si un document appartient à la catégorie des sports, de la politique, de la technologie, etc. Les caractéristiques / prédicteurs utilisés par le classificateur sont la fréquence des mots présents dans le document.
- **Naïf bayésien Bernoulli :** similaire aux bayes naïfs multinomiaux mais les prédicteurs sont des variables booléennes. Les paramètres que nous utilisons pour prédire la variable de classe ne prennent que des valeurs oui ou non, par exemple si un mot apparaît ou non dans le texte.
- **Naïf bayésien gaussien :** les prédicteurs prennent une valeur continue et ne sont pas discrets, nous supposons que ces valeurs sont échantillonnées à partir d'une distribution gaussienne. Étant donné que la façon dont les valeurs sont présentes dans l'ensemble de données change, la formule de probabilité conditionnelle devient :

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$$

La méthode est la suivante : tout d'abord déterminer un ensemble d'apprentissage, ensuite, déterminer les probabilités à priori de chaque classe, on applique la règle de Bayes et on choisit la classe la plus probable, c'est à dire celle avec la plus grande probabilité.

Avantages :

- Requiert relativement peu de données d'entraînement pour estimer les paramètres nécessaires à la classification (grâce à l'hypothèse d'indépendance, on se contente des variances)
- Eviter les problèmes lors de l'augmentation de la taille des données de départ
- Peu de sur-apprentissage

Inconvénients :

- Parfois difficile à intégrer directement dans notre application
- Compliqué à personnaliser pour des besoins spécifiques

Sous Python: naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)

alpha : réel qui correspond au lissage de Laplace/Linstone.

fit_prior : booléen qui permet de dire si on calcule la probabilité à priori ou non. Si ce n'est pas le cas, on considère ces probabilités de classes uniformes.

class_prior : liste des probabilités apriori prédéfinies.

IV – Mesures de performance

A. Matrice de confusion

Le but de **la matrice de confusion** est **d'évaluer la qualité prédictive du modèle**, c'est un résumé des résultats de prédictions sur un problème de classification. Un des intérêts de la matrice de confusion est qu'**elle montre rapidement si un système de classification parvient à classifier correctement**. Elle se présente sous la forme ci-dessous :

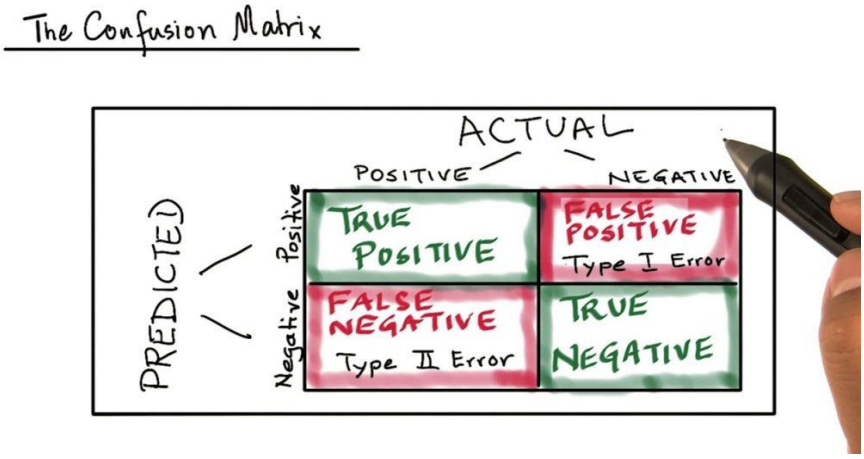


Figure 15 : Schéma d'une matrice de confusion

Par exemple, la case « true positive », qui est « vrais positifs » correspond aux cas où la prédiction est positive et où la valeur réelle est effectivement positive.

Il existe deux techniques pour calculer la performance des modèles de classification et pour trouver le seuil C du score pour obtenir la règle suivante : le modèle calcule un score *c* qui est comparé au seuil *S* pour prédire la classe

$$S(X) \leq c \Rightarrow \text{l'individu est affecté au groupe 1}$$

$$S(X) > c \Rightarrow \text{l'individu est affecté au groupe 2}$$

B. Courbe ROC

Elle correspond à **la représentation graphique du taux de vrais positifs en fonction du taux de faux positifs.** Chaque valeur de *S* fournira un point de la courbe ROC, qui ira de (0, 0) à (1, 1). Un classificateur aléatoire tracera une droite allant de (0, 0) à (1, 1).

Au point (0, 0) le classificateur déclare toujours 'négatif' : il n'y a aucun faux positif, mais également aucun vrai positif. Les proportions de vrais et faux négatifs dépendent de la population sous-jacente.

Au point (1, 1) le classificateur déclare toujours 'positif' : il n'y a aucun vrai négatif, mais également aucun faux négatif. Les proportions de vrais et faux positifs dépendent de la population sous-jacente.

Au point (0, 1) le classificateur n'a aucun faux positif ni aucun faux négatif, et est par conséquent parfaitement exact, ne se trompant jamais.

Au point (1, 0) le classificateur n'a aucun vrai négatif ni aucun vrai positif, et est par conséquent parfaitement inexact, se trompant toujours. Il suffit d'inverser sa prédiction pour en faire un classificateur parfaitement exact.

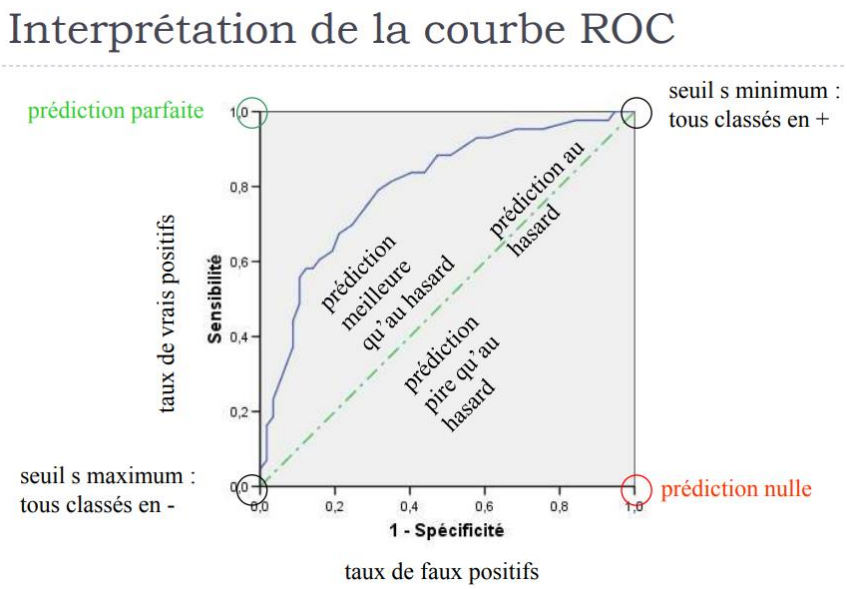


Figure 16 : Explication d'une courbe ROC

C. Courbe LIFT

Correspond à la représentation de la proportion y de vrais positifs en fonction de la proportion x d'individus sélectionnés.

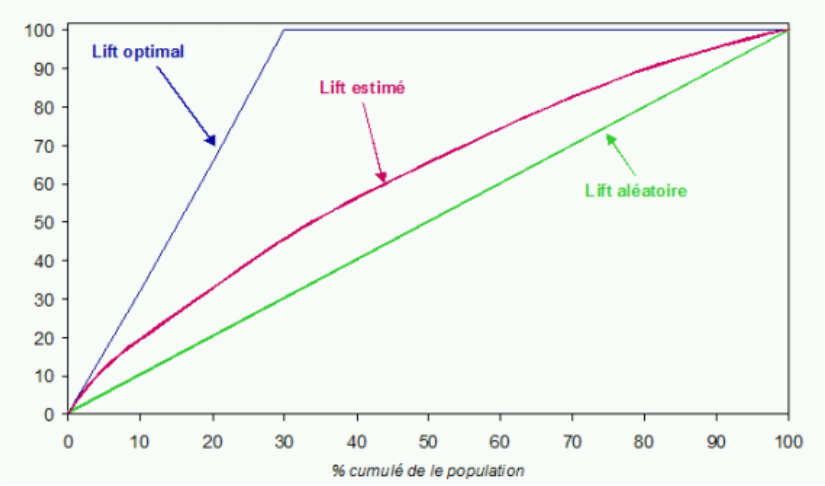


Figure 17 : Schéma d'une courbe LIFT

V – Scoring avec la liste des 10 variables

Dans cette partie, nous allons expliquer notre fichier avec les 10 variables sélectionnées. Ce fichier regroupe **les différentes méthodes de scoring** que nous avons utilisées pour ce jeu de données.

Préliminaires

Tout d'abord voici les bibliothèques que nous avons utilisé pour notre scoring.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
from sklearn import model_selection
from sklearn.neighbors import KNeighborsClassifier
from sklearn import neighbors, metrics
from sklearn import preprocessing
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
from matplotlib import pyplot
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn import model_selection
from sklearn import preprocessing
```

Figure 18 : Liste des bibliothèques utilisées

Notre trame de données est la variable **B**. Après avoir importé nos données, nous avons associé à chaque individu une clé.

```
B = pd.read_csv('10var.csv', delimiter= ',',header=0)
B.rename(columns={'Unnamed: 0':'cle'}, inplace=True)
```

Nous avons également créé une variable **varexpliquee** qui contient la variable à expliquer.

```
varexpliquee = B['assur_caravane']
```

Pour commencer chaque méthode de scoring, nous avons effectué une technique permettant d'ajuster la distribution entre les différentes catégories de données. Il s'agit de **l'Over / Under sampling** sur nos données.

```
#Resultat incroyable
from imblearn.combine import SMOTEENN
smote_enn = SMOTEENN(random_state=0)
X_resampled, y_resampled = smote_enn.fit_resample(X_knn, y_knn)

#print(X_resampled)
#print(y_resampled)
```

Figure 19 : Over/Under sampling sur nos données

Ensuite, **nous avons effectué une séparation** en tables d'entraînement X_train, y_train et en tables de test x_test, y_test.

```
#Separation en table entrainement et test
X_train, x_test, y_train, y_test = model_selection.train_test_split(X_resampled, y_resampled, test_size=0.25)
```

Figure 20 : Séparation des tables

1. Méthode des K plus proches voisins (KNN)

Comme mentionné précédemment, **la méthode KNN nécessite un paramètre de complexité k**. C'est le seul hyperparamètre de ces méthodes. Nous avons donc voulu optimiser ce paramètre. Pour cela **nous créons une grille de recherche**. Le seul paramètre de cette grille est donc le nombre k de voisins. Le score que l'on cherche à optimiser est le score 'accuracy'. On cherche à **optimiser le nombre de bonnes prédictions** (à 0 et à 1). On effectue aussi une validation croisée à 5 folds pour vérifier les résultats. On entraîne le modèle et on affiche les meilleurs paramètres en fonction des résultats de la grille de recherche. Ensuite, on prédit avec x_test. On affiche aussi le recall, le score, la précision, la matrice de confusion et y_test.

La figure ci-dessous illustre la méthode.

```
# Fixer les valeurs des hyperparamètres à tester
param_grid = {'n_neighbors':[1,2,3,4,5,6,7,8,9,10, 11, 13, 15]}

# Choix du score à optimiser
score = 'accuracy'

# Créer un classifieur kNN avec recherche d'hyperparamètre par validation croisée
clf_knn = GridSearchCV(
    neighbors.KNeighborsClassifier(), # un classifieur kNN
    param_grid,     # hyperparamètres à tester
    cv=5,           # nombre de folds de validation croisée
    scoring=score   # score à optimiser
)

# Optimiser ce classifieur sur le jeu d'entraînement
clf_knn.fit(X_train, y_train)

# Afficher le(s) hyperparamètre(s) optimaux
print("Meilleur(s) hyperparamètre(s) sur le jeu d'entraînement:")
print(clf_knn.best_params_)

# Afficher les performances correspondantes
print("Résultats de la validation croisée :")
for mean, std, params in zip(
        clf_knn.cv_results_['mean_test_score'], # score moyen
        clf_knn.cv_results_['std_test_score'],  # écart-type du score
        clf_knn.cv_results_['params']           # valeur de l'hyperparamètre
    ):
    print("{} = {:.3f} (+/-{:.03f}) for {}".format(score,mean,std*2,params))

# Meilleur choix de l'hyperparmetre k : 2

#Mesure de performance (avec k = 2)

# Prévision de l'échantillon test
pred = clf_knn.predict(x_test)

# Precision
print("\n precision")
print(precision_score(y_test,pred))

# Rappel
print("\n rappel")
print(recall_score(y_test,pred))

# F1 score
print("\n score f1")
print(f1_score(y_test,pred))

# Matrice de confusion
print(confusion_matrix(y_test,pred))
```

Figure 21 : Utilisation de la méthode KNN en Python

Nous obtenons les résultats suivants :

```
Meilleur(s) hyperparamètre(s) sur le jeu d'entraînement:
{'n_neighbors': 1}
Résultats de la validation croisée :
accuracy = 0.980 (+/-0.009) for {'n_neighbors': 1}
accuracy = 0.979 (+/-0.008) for {'n_neighbors': 2}
accuracy = 0.955 (+/-0.012) for {'n_neighbors': 3}
accuracy = 0.958 (+/-0.007) for {'n_neighbors': 4}
accuracy = 0.937 (+/-0.012) for {'n_neighbors': 5}
accuracy = 0.939 (+/-0.008) for {'n_neighbors': 6}
accuracy = 0.919 (+/-0.013) for {'n_neighbors': 7}
accuracy = 0.926 (+/-0.011) for {'n_neighbors': 8}
accuracy = 0.905 (+/-0.010) for {'n_neighbors': 9}
accuracy = 0.909 (+/-0.010) for {'n_neighbors': 10}
accuracy = 0.896 (+/-0.011) for {'n_neighbors': 11}
accuracy = 0.884 (+/-0.010) for {'n_neighbors': 13}
accuracy = 0.876 (+/-0.016) for {'n_neighbors': 15}

 precision
0.9757785467128027

 rappel
0.9982300884955753

 score f1
0.9868766404199475
[[ 894   28]
 [   2 1128]]
```

Figure 22 : Résultats de la méthode KNN

On a un modèle **très performant avec un rappel, une précision et un score proche de 1**. Le meilleur k est k = 1 ou k = 2 (ça peut changer quand on exécute étant donné que la séparation en table d'entrainement est aléatoire). On prédit 1156 personnes à 1 et 1128 sont bien prédit. On prédit 896 personnes à 0 et 894 individus sont correctement prédit. Il y a donc 30 erreurs sur plus de 2000 individus.

2. Retraitement des variables

Nous avons également remarqué que **certaines variables possédaient une mauvaise queue de distribution**. Afin de remédier à cela, nous avons procédé au **retraitement des variables correspondantes**. Les variables que nous avons retraitées sont :

- ✓ mt_auto
- ✓ nb_bateau
- ✓ mt_RC
- ✓ mt_incendie
- ✓ mt_bateau
- ✓ revenu_moyen
- ✓ niv_etude_haut
- ✓ revenu1

Voici un exemple du code pour la variable : **nb_bateau**

```
#nb_bateau

print(X['nb_bateau'].value_counts())

arbre_cla_nbb=DecisionTreeClassifier(max_depth=2,min_samples_leaf=20)
arbre_cla_nbb.fit(X_new[:,1].reshape(X_new.shape[0],1),varexpliquee)
nb_b=np.repeat(-1,X_new.shape[0])
nb_b[X_new[:,1]<=0.5]=0
nb_b[(X_new[:,1]>0.5)] = 1

print(nb_b)

for i in range(0,5822):
     X.at[i,'nb_bateau'] = nb_b[i]

fig = plt.figure(figsize=[10,10])
fig.suptitle(B.columns[0])
plot_tree(arbre_cla_nbb)

print(X['nb_bateau'].value_counts())
```

Figure 23 : Exemple de retraitement d'une variable, ici nb_bateau

Un retraitement des variables est nécessaire pour l'ensemble des méthodes suivantes. Ici, on explique notre jeu de 10 variables, nous avons retraité plus de variables pour nos jeux à 20, 30 ou 46 variables.

Nous obtenons également **l'arbre de décision** et la **nouvelle répartition des variables**, c'est à dire 5789 en modalité 0 et 33 en modalité 1.

```
0    5789
1      31
2       2
Name: nb_bateau, dtype: int64
[0 0 0 ... 0 0 0]
0    5789
1      33
Name: nb_bateau, dtype: int64
```

de

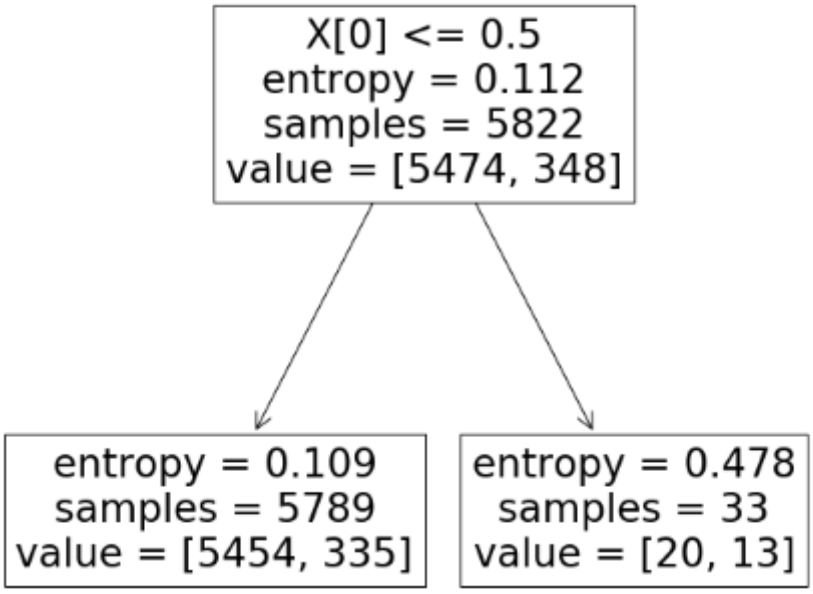


Figure 24 : Suite du retraitement de la variable nb_bateau

Une fois ces traitements réalisés, nous pouvions continuer à travailler et réaliser les autres scoring.

3. Régression logistique

Cette méthode exige une **standardisation** des variables explicatives. On standardise X_train par rapport à la moyenne et l'écart type de X_train et on standardise aussi X_test par rapport à la moyenne et l'écart type de X_train.

```
#Standardisation car Reg Log
std_scale = preprocessing.StandardScaler().fit(X_train)

X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)
```

Figure 25 : On standardise pour effectuer une régression logistique

On réalise ensuite la régression :

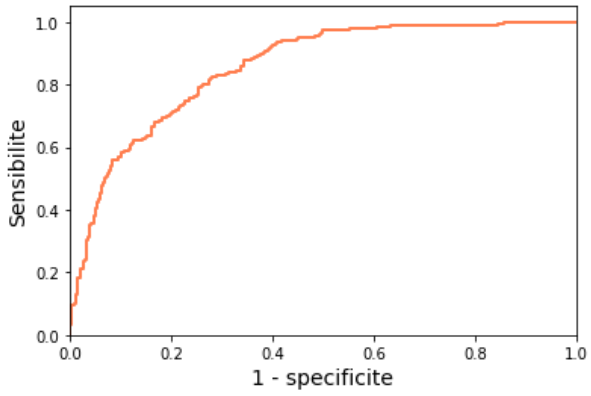
```
#Construction d'une grille
parameters = {'penalty':['l1', 'l2'],'C':np.logspace(-10,1,10),'solver':['liblinear','saga']}

model = LogisticRegression()
grid_reg_log = model_selection.GridSearchCV(model,param_grid = parameters,scoring='accuracy')
model_rl = grid_reg_log.fit(X_train_std,y_train)
print(model_rl.best_params_)
print(model_rl.best_score_) #realise automatiquement par validation croisee

# Courbe Roc
y_pred_proba = model_rl.predict_proba(X_test_std)[:, 1]
[fpr, tpr, thr] = metrics.roc_curve(y_test, y_pred_proba)
plt.plot(fpr, tpr, color='coral', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('1 - specificite', fontsize=14)
plt.ylabel('Sensibilite', fontsize=14)
plt.show()

print(metrics.auc(fpr, tpr))


# precision-recall curve and f1
# predict probabilities
lr_probs = model_rl.predict_proba(X_test)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# predict class values
yhat = model_rl.predict(X_test)
lr_precision, lr_recall, _ = precision_recall_curve(y_test, lr_probs)
lr_f1, lr_auc = f1_score(y_test, yhat), auc(lr_recall, lr_precision)
# plot the precision-recall curves
no_skill = len(y_test[y_test==1]) / len(y_test)
pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
pyplot.plot(lr_recall, lr_precision, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

Figure 26 : Réalisation de la régression logistique en Python

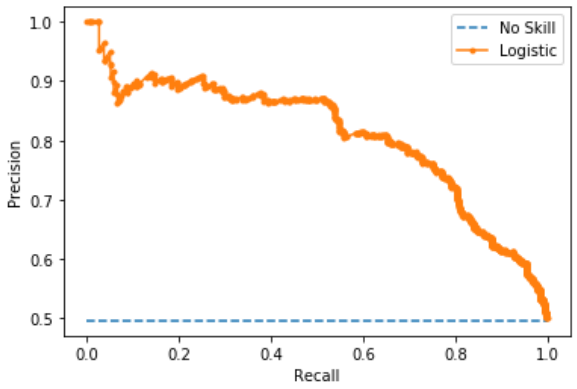
Dans cet algorithme, nous créons **une grille de recherche** où l'on teste plusieurs hyperparamètres différents. On teste deux types de pénalité différents, des valeurs de C différentes et deux solveurs différents**. Ce sont les deux solveurs les plus utilisés pour une régression logistique**. On entraîne le modèle, on affiche le meilleur choix de paramètres en fonction des résultats de la grille de recherche. On affiche le score. On affiche aussi deux courbes. La première est la spécificité en fonction de la sensibilité. La seconde est le recall en fonction de la précision. On affiche aussi la matrice de confusion.

La Précision, le Rappel et le F-score nous permettent de mesurer la qualité du classifieur. Ainsi nous obtenons les résultats suivants :

```
{'C': 0.0359381366380464, 'penalty': 'l2', 'solver': 'liblinear'}
0.7452721591843446
```




```
0.8537720678505353
```




```
[[592 174]
 [192 563]]


precision
0.7639077340569878
rappel
0.7456953642384105
score f1
0.7546916890080428
```

Figure 27 : Résultats obtenus grâce à la régression logistique

Les résultats sont bons mais très en dessous de la méthode KNN.

4. Arbres de décision

```
# définition du modèle
tree= DecisionTreeClassifier(max_depth=9)
treeC=tree.fit(X_train, y_train)

# Optimisation de la profondeur de l'arbre
param=[{"max_depth":list(range(2,10))}]
tree= GridSearchCV(DecisionTreeClassifier(),param,cv=10,n_jobs=-1)
treeOpt=tree.fit(X_train, y_train)
# paramètre optimal
print("Meilleur score = %f, Meilleur paramètre = %s" % (1. - treeOpt.best_score_,treeOpt.best_params_))

# Prévision de l'échantillon test
yChap = treeOpt.predict(X_test)
# matrice de confusion
print(confusion_matrix(y_test,yChap))
```

Figure 28 : Arbre de décision en Python

Les résultats que nous obtenons :

```
Meilleur score = 0.118237, Meilleur paramètre = {'max_depth': 9}
[[585 144]
 [ 47 745]]
```

Figure 29 : Résultats obtenus par l'arbre de décision

Nous créons de nouveau **une grille de recherche** où **l'on optimise la profondeur de l'arbre**. La meilleure profondeur est 9. On entraîne le modèle, on prédit avec l'échantillon X_test et on affiche la matrice de confusion. Les résultats sont du même ordre que la régression logistique. **Ils sont donc moins bon que la méthode KNN.**

5. Méthode SVM

Tout comme pour la régression logistique, **la méthode SVM** nécessite une **standardisation** des données (X_train et X_test par rapport à X_train).

```
# standardiser les données
std_scale = preprocessing.StandardScaler().fit(X_train)

X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)

# definir les hyperp parametres
param_grid = {'C': [0.1, 1, 10],
              'gamma': [1, 0.1, 0.01],
              'kernel': ['rbf','sigmoid']}

grid = GridSearchCV(SVC(), param_grid)

# on test le modele optimal
grille = grid.fit(X_train_std, y_train)

# afficher les paramètres optimaux
print("The optimal parameters are {} with a score of {:.2f}".format(grille.best_params_, grid.best_score_))

# prédire sur le jeu de test avec le modèle optimisé
y_pred = grille.decision_function(X_test_std)

# construire la courbe ROC du modèle optimisé
fpr_cv, tpr_cv, thr_cv = metrics.roc_curve(y_test, y_pred)

# calculer l'aire sous la courbe ROC du modèle optimisé
auc_cv = metrics.auc(fpr_cv, tpr_cv)
print(auc_cv)

# La courbe ROC
fig = plt.figure(figsize=(6, 6))
plt.plot(fpr_cv, tpr_cv, '-', lw=2, label='AUC=%.2f' % auc_cv)
plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('SVM ROC Curve', fontsize=16)
plt.legend(loc="lower right", fontsize=14)
plt.show()

print(confusion_matrix(y_test, y_pred))
```

Figure 30 : Programme de la SVM en Python

Dans ce nouveau programme, nous créons de nouveau **une grille de recherche** où l'on met différents paramètres comme le **C, le gamma et le noyau**. Nous cherchons les ordres de grandeur de C et gamma. Pour le noyau, nous avons mis dans nos paramètres **deux noyaux classiques de la SVM** pour que l'algorithme trouve lequel des deux noyaux est le meilleur dans notre étude. Le noyau va impacter la manière dont les individus sont projetés en dimension supérieure. La forme de la fonction sera différente. On entraîne le modèle et on prédit. On utilise y_test et les prédictions pour effectuer une matrice de confusion. On trace aussi une courbe ROC.

Voici les résultats que nous obtenons :

```
The optimal parameters are {'C': 10, 'gamma': 1, 'kernel': 'rbf'} with a score of 0.87
0.9182626869854149
```

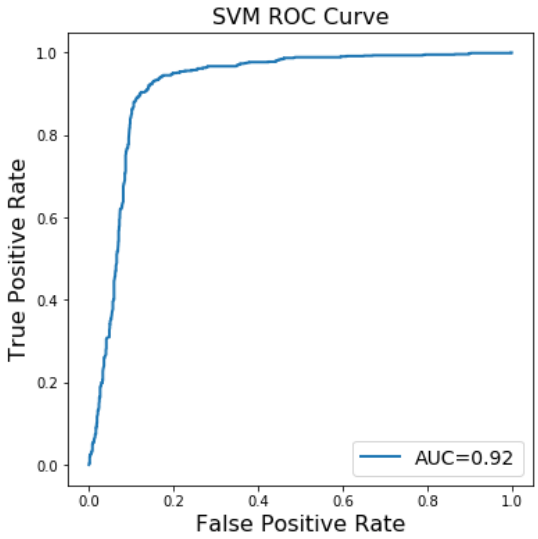


Figure 31 : Résultats obtenus grâce à la SVM

Les résultats sont très bons, avec un AUC (aire sous la courbe) très important, qui en témoigne. Les résultats sont un peu moins bons que l'algorithme KNN.

6. Bayésien naïf

Parmi les types de classifieurs naïfs bayésiens, selon notre étude, nous avons utilisé le bayésien naïf gaussien.

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train_std,y_train)
gnb.score(X_train_std,y_train)

y_pred = gnb.predict(X_test_std)
print(confusion_matrix(y_test, y_pred))

print(gnb.score(X_train_std,y_train))

# construire la courbe ROC
from sklearn import metrics
fpr, tpr, thr = metrics.roc_curve(y_test, y_pred)

# calculer l'aire sous la courbe ROC
auc = metrics.auc(fpr, tpr)

# créer une figure
from matplotlib import pyplot as plt
fig = plt.figure(figsize=(6, 6))

# afficher la courbe ROC
plt.plot(fpr, tpr, '-', lw=2, label='gamma=0.01, AUC=%.2f' % auc)

# donner un titre aux axes et au graphique
plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('Bays Naif ROC Curve', fontsize=16)

# afficher la légende
plt.legend(loc="lower right", fontsize=14)

# afficher l'image
plt.show()
```

Figure 32 : Bayésien naïf gaussien en Python

Ici, nous n'avons pas réalisé de grille de recherche. L'algorithme n'est sûrement pas optimisé. **On entraîne le modèle et on prédit avec notre jeu de test**. On affiche une courbe ROC et la matrice de confusion. **Les résultats sont bons mais moins que la méthode KNN**.

7. Réseau de neurones

Pour finir nous avons décider de mettre en place **un réseau de neurones.** Nous avons donc installé la bibliothèque Tensorflow.

Un réseau de neurones représente **un ensemble de nœuds connectés entre eux** dans lequel chaque variable correspondant à un nœud. Les étapes dans la mise en œuvre d'un réseau de neurones pour la prédiction ou le classement sont :

- identification des données en entrée et en sortie
- normalisation de ces données : on fait la standardisation des données
- constitution d'un réseau avec une topologie adaptée
- apprentissage du réseau
- test du réseau : usage de la table train
- application du modèle généré par l'apprentissage : usage de la table test

Dans notre cas, nous avons le code ci-dessous :

```
import tensorflow as tf

# Utilisation du package tensorflow à installer

#print(X)
#print(y)

from imblearn.combine import SMOTEENN
smote_enn = SMOTEENN(random_state=0)
X_rn, y_rn = smote_enn.fit_resample(X, y)


from sklearn import model_selection
X_train, X_test, y_train, y_test = model_selection.train_test_split(X_rn, y_rn, test_size=0.2,random_state = 1)

std_scale = preprocessing.StandardScaler().fit(X_train)

X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)

X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(10, activation = "relu"))
model.add(tf.keras.layers.Dense(5, activation = "relu"))
model.add(tf.keras.layers.Dense(2, activation = "softmax"))

model.compile(loss="sparse_categorical_crossentropy",optimizer="sgd",metrics=["accuracy"])

history= model.fit(X_train, y_train, epochs = 500,validation_split = 0.2)

model.summary()
```

Figure 33 : Réseau de neurones en Python

Nous créons notre modèle. **Nous ajoutons un layer initial contenant 10 neurones**. Un neurone pour chaque variable. Nous créons **des layers intermédiaires qui vont permettre de connecter les neurones** puis un layer de sortie avec deux neurones pour la prédiction (0 ou 1). On compile le modèle. On utilise le loss correspondant à la forme de nos données. On utilise l'algorithme du gradient conjugué stochastique pour optimiser l'algorithme. On optimise selon

le nombre de bonnes réponses (0 ou 1). On entraîne notre modèle et on ajoute une validation croisée. **Le nombre d'epochs est le nombre de fois que l'on entraîne le modèle**. Pour fixer ce nombre, il faut faire tourner l'algorithme plusieurs fois**. Un nombre d'epochs trop important entraîne un sur-aprentissage (overfitting).** Un nombre d'epochs trop faible va fournir un modèle qui ne sera pas bon.

```
loss_curve = history.history["loss"]
acc_curve = history.history["accuracy"]

loss_val_curve = history.history["val_loss"]
acc_val_curve = history.history["val_accuracy"]

plt.plot(loss_curve, label = "Train")
plt.plot(loss_val_curve, label="validation")
plt.title("Loss")
plt.legend()
plt.show()

plt.plot(acc_curve, label = "Train")
plt.plot(acc_val_curve, label="validation")
plt.title("Accuracy")
plt.legend()
plt.show()

model.evaluate(X_test_std,y_test)

y_pred = model.predict(X_test_std)

L = []
for i in range(1521):
    if y_pred[i][0] > y_pred[i][1]:
        L.append(0)
    else:
        L.append(1)

print(y_test.shape)
y_pred_fin = np.array(L)
print(y_pred_fin.shape)
print(confusion_matrix(y_test, y_pred_fin))
```

Figure 34 : Programme permettant l'affichage des résultats

La première partie de ce code sert à créer deux courbes permettant de visualiser un overfitting. En voici un exemple ci-dessous. Un nombre d'epochs trop important entraîne un écart trop important entre les deux courbes. Il faut donc changer le nombre d'epochs.

Voici aussi les deux courbes que nous obtenons :

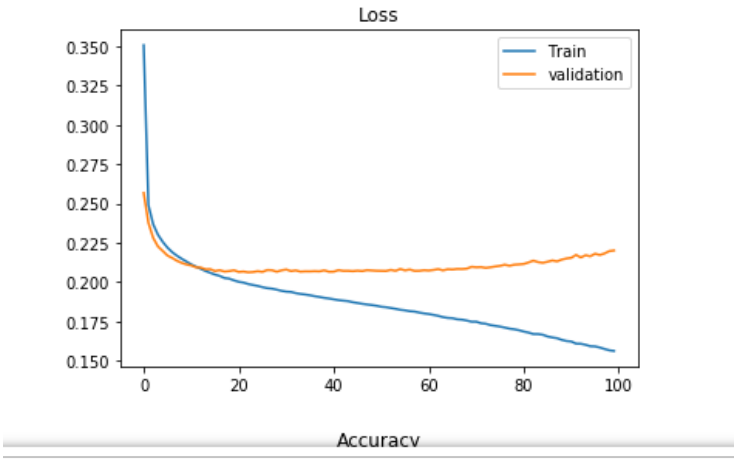


Figure 35 : Courbes montrant un overfitting

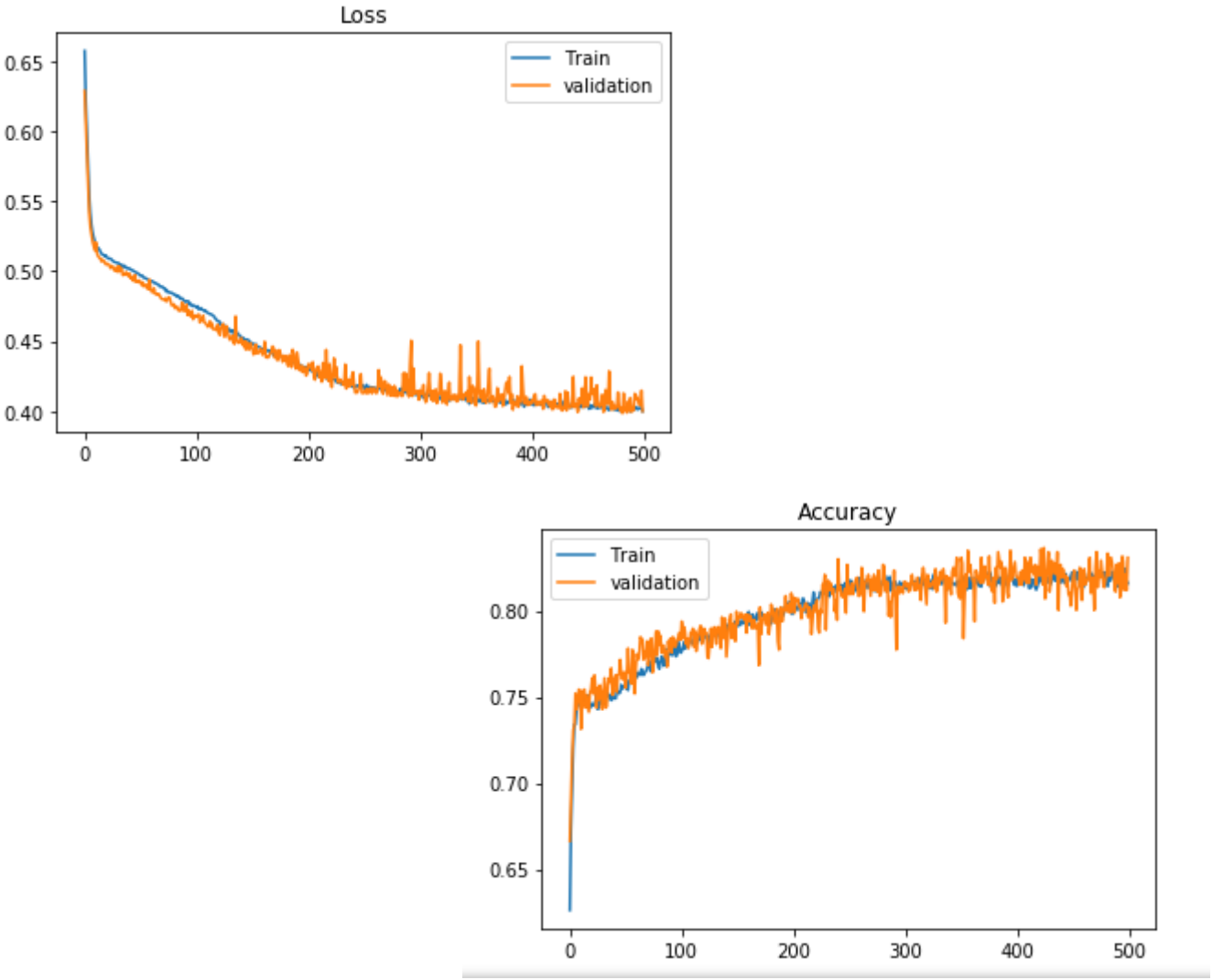


Figure 36 : Courbes de perte et de précision obtenues

Le reste du code ci-dessus sert à **tester le modèle qui prédit des probabilités pour chaque modalité (0 ou 1)**. La somme des probabilités pour chaque individu est donc égale à 1. On fait une boucle pour avoir y_pred avec la prédiction à 0 ou 1. On affiche la matrice de confusion.

On obtient la matrice de confusion suivante :

```
[[141 588]
 [ 25 767]]
```

Figure 37 : Matrice de confusion obtenue

Il y a beaucoup d'erreurs, notamment des individus qu'on prédit à 1 alors qu'ils sont à 0

VI – Généralisation des résultats.

Nous avons effectué les mêmes méthodes pour nos jeux de 10,20,30 et 46 variables.

- **Pour 10 variables**, la meilleure méthode est **la méthode KNN.**
- **Pour 20 variables**, la meilleure méthode est **la méthode KNN.**
- **Pour 30 variables**, les méthodes **KNN et réseau de neurones** sont efficaces.
- **Pour 46 variables**, les méthodes **KNN et réseau de neurones** sont efficaces.

Les méthodes **SVM** appliquées sur les différents jeux sont aussi efficaces. Dans l'ensemble toutes les méthodes adoptées ont fourni de bons résultats. **Les meilleurs résultats sont obtenus avec les méthodes KNN avec 20 variables et les réseaux de neurones pour 30 et 46 variables.**

Nous avons d'abord transformé le jeu de données de test pour avoir 30 variables retraités. Nous avons appliqué les mêmes opérations que lors de la phase de préparation des algorithmes mais nous l'avons appliqué au jeu de données de test. Pour générer les trois fichiers demandés, **nous avons utilisés la méthode du réseau de neurones avec 30 variables.**

Pour générer le fichier avec la prévision à 0 ou 1 pour chaque individu, nous avons utilisé la méthode « predict » et nous avons mis notre nouveau jeu de test en paramètre.

Pour générer le fichier contenant les 1000 individus les plus susceptibles d'être à 1, nous avons affiché non pas la prédiction mais la probabilité d'appartenance à chaque modalité. Nous avons ensuite trié le résultat par ordre décroissant de probabilité d'appartenance à la modalité 1. Nous avons choisi les 1000 premiers.

Pour générer le fichier avec n individus les plus susceptibles d'appartenir à la classe 1, nous avons fixé une probabilité d'appartenance à la classe (0.9). Si la probabilité qu'un individu appartient à la classe 1 est supérieur à 90%, nous sélectionnons l'individu.