ESIEE PARIS

Apprentissage automatique 2

Rapport de compétition

Yanis PERRIN

Table des matières

Analyse des données

L'analyse des données est l'une des parties, si ce n'est la partie la plus importante de notre projet. Elle nous permet de comprendre quelles sont les données disponibles dans le dataset. En comprenant ces données, nous pourrons ensuite effectuer de la features selection ou même du features engineering afin d'obtenir de meilleures performances.

Le dataset

Le dataset "train" récupéré est sous format "csv" et représente l'ensemble d'entraînement, c'est avec ce dataset que l'on entraînera nos différents modèles. Ce dataset d'entraînement contient 10906 observations (une observation correspondant à un individu) et 36 variables. Parmi ces 36 variables, 2 représentent nos variables à prédire.

Les variables à prédire

Dans cette compétition, notre objectif est de prédire l'appétence pour une assurance vie et un plan d'épargne retraite. Dans notre dataset, ces deux variables se caractérisent par le label "O" si la réponse est positive et par le label "N" si la réponse est négative. Nos deux variables sont de types "str", on utilisera des modèles types classifier pour effectuer les prédictions.

Les variables à notre disposition

En plus des deux variables à prédire, nous disposons de 34 variables différentes qui auront chacune un impact différent sur nos prédictions. Parmi les 34 variables disponibles, seulement 2 sont numériques (l'âge et le nombre d'enfants de l'individu). Les autres variables sont catégorielles (nationalité, niveau de diplôme, niveau d'urbanisation…etc).

Les valeurs manquantes

Parmi les 10906 observations disponibles, certaines contiennent des valeurs manquantes pour certaines variables. Parmi les 36 variables disponibles, 8 d'entre elles présentent des valeurs manquantes. La proportion en moyenne des ces valeurs manquantes parmi ces 8 variables est de 10.3 %, elle peut aller jusqu'à 20% pour la variable "work". La présence de ces valeurs manquantes n'est donc pas à négliger car si ces valeurs ne sont pas remplacées cela peut impacter la précision de nos modèles, une étape de pré-processing est alors nécessaire pour remplacer ces valeurs.

Preprocessing

L’étape de pre-processing nous permet de préparer nos données avant de les injecter dans nos différents modèles.

Standardisation des données numériques

La première étape du pre-processing consiste à standardiser les données numériques, cette étape nous permet de modifier les valeurs des colonnes numériques pour utiliser une échelle commune, sans que les différences de plages de valeurs ne soient faussées et sans perte d'informations. Cette mise à l'échelle nous permet de réduire leur variance ou leur valeur absolue. Pour cela, nous utilisons la méthode StandardScarler() de la librairie Sklearn sur les variables “Âge” et “Nbenf”.

Encodage des variables catégorielles

La deuxième étape du pre-processing consiste à encoder nos variables catégorielles. En effet, les variables catégorielles non encodées ne pourront pas être utilisées pendant l’entraînement de nos différents modèles. L’encodage consiste à changer les valeurs des différentes classes de la variable par une valeur entre 0 et le nombre de classe -1. Chaque valeur entre 0 et le nombre de classe -1 définit alors une valeur initiale d’une classe (exemple de la variable “Sexe” : Cette variable présente deux valeurs différentes “H” pour sexe masculin et “F” pour sexe féminin, en encodant cette variable la valeur “F” deviendra 0 et la valeur “H” deviendra 1). Ainsi, nos variables catégorielles pourront être comprises par nos différents modèles. Pour cela, nous utilisons la méthode LabelEncoder() de la librairie Sklearn, cette méthode appliquée à nos données nous permet d’effectuer le cheminement expliquées précédemment.

À savoir : Une fois, nos variables catégorielles encodées celles-ci présentent des valeurs comprises en 0 et le nombre de classe -1, nos différents modèles peuvent alors déterminer un ordre d’importance des classes (exemple de la variable “Sexe” : le modèle peut penser que la valeur “H” devenue 1 est supérieur à la variable “F” devenue 0). Pour remédier à ce problème, on pourrait utiliser la méthode get_dummies() de la librairie Pandas qui permet de créer une colonne contenant uniquement les valeurs 0 ou 1 pour chaque classe de la variable (exemple de la variable “Sexe” : une colone “Sexe_H” pourrait être créer en indiquant par un 1 si l’individu est de sexe masculin et 0 si il ne l’est pas et une colonne “Sexe_F” qui pourrait indiquer par un 1 si l’individu est de sexe féminin et 0 si il ne l’est pas). Ainsi, on pourrait éviter ce problème d’ordre défini par le modèle. Cependant, après plusieurs essais effectués, les résultats obtenus par les différents modèles n’étaient pas si différents des résultats sans cette méthode. Pour éviter de complexifier les données et de créer plus de 80 variables différentes, on décide de ne garder notre méthode initiale.

Remplacement des valeurs manquantes

La troisième et dernière étape du pré-processing consiste à remplacer les valeurs manquantes présentent dans le dataset. Pour cela, on utilise la méthode IterativeImputer() de la libraire Sklearn. Cette méthode permet, à l'aide d'un estimateur (ici un GradientBoostingClassifier), de remplacer les valeurs manquantes en appliquant la stratégie "most_frequent" c'est-à-dire en remplaçant les valeurs manquantes par la valeur la plus fréquente dans l'ensemble du dataset. Cependant, même si cette méthode est similaire à la méthode SimpleImputer() de Sklearn, elle ajoute une étape lors de son imputation. En plus de déterminer les valeurs manquantes par la stratégie choisie, elle va utiliser l'estimateur passé en paramètre comme un prédicteur afin de déterminer les valeurs manquantes d'une variable en fonction des autres variables. Elle répète ce processus jusqu'à ce que le nombre d'itérations ou la tolérance d'arrêt choisi est atteinte.

Une fois les valeurs manquantes remplacées, les données sont prêtes à être utilisées par nos différents modèles. Cependant, il serait intéressant d'effectuer une sélection des variables les plus impactantes pour notre modèle. Cela nous permettra de retirer les variables non impactantes et ainsi éviter de complexifier l'entraînement de nos modèles tout en augmentant leur précision.

Split & Features Selection

Split

Avant d'effectuer la sélection des variables, on divise le jeu de données en données d'entraînement et de test afin de pouvoir entraîner nos modèles sur le jeu de données d'entraînement et de le tester sur le jeu de données de test. Ainsi, nous pourrons étudier le score de nos modèles sur des données encore jamais vues par les modèles. Le jeu de test représente 33% de nos données. Bien sûr, on drop les variables à prédire de nos variables explicatives. Dans notre cas, on prédira les variables indépendantes l'une de l'autre mais les étapes qui vont suivre suivent le même procédé pour les deux variables.

La méthode RFECV

Pour effectuer la sélection de variables, on utilisera la méthode RFECV de la librairie Sklearn. Cette méthode permet, à l'aide d'un estimateur (ici GradientBoostingClassifier) et de la cross validation d'éliminer les variables les moins impactantes de notre dataset. Pour une meilleure précision, on décide de faire appel à la méthode GridSearchCv() qui permet de déterminer les meilleurs paramètres de l'estimateur entré en paramètre en fonction de la métrique à optimiser (ici le roc_auc). On utilise alors GridSearchCv() pour déterminer les meilleurs paramètres de notre RFECV. Une fois les paramètres connus, nous appliquons la méthode sur notre jeu de données. Nous obtenons alors un jeu de données qui contient uniquement les variables les plus impactantes. Nous pouvons donc passer au choix des différents algorithmes et modèles que nous utiliserons pour la prédiction de nos deux variables.

Choix des algorithmes

Les algorithmes choisis

En qui concerne le choix des algorithmes, nos variables à prédire étant catégorielles, nous nous focaliserons bien entendu sur les différents classifier possibles. Parmi les différents classifiers possibles, six ont été retenus pour le projet :

- La régression logistique (LogisticRegression de la bibliothèque Sklearn)
- La méthode des K plus proches voisins (KNeighborsClassifier de la bibliothèque Sklearn)
- Les forêts aléatoires (RandomForestClassifier de la bibliothèque Sklearn)
- Le gradient boosting (GradientBoostingClassifier de la bibliothèque Sklearn)
- L'extrême gradient boosting (XGBClassifier de la bibliothèque XGBoost)
- Le light gradient boosting (LGBMClassifier de la bibliothèque LightGBM)

Ces modèles ont été choisis car ce sont les classifiers les plus performants à l'heure actuelle.

À noter qu'un algorithme SVM (support vector machine) aurait aussi pu être utilisé. Parmi les modèles choisis, certains ont déjà été abordés lors des différents TP's (RandomForestClassifier, GradientBoostingClassifier, LogisticRegression).

Les différentes étapes

- En premier lieu, nous devons déterminer les paramètres qui maximisent le score AUC obtenus de chaque modèle.
- Une fois les paramètres choisis, nous entraînons chacun de nos modèles séparément sur notre jeu de données d'entraînement.
- Puis, nous testerons sa précision avec le jeu de données de test.
- Une fois les modèles testés séparément, nous effectuerons un modèle d'ensemble dit "Stacking" afin de combiner nos différents modèles.
- Nous choisirons à la fin les trois modèles qui présenteront le meilleur score sur le jeu de données "test.csv".

Choix des paramètres

Approche générale

Afin de déterminer les meilleurs paramètres de nos modèles, nous utilisons une nouvelle fois l'algorithme de grid search via la méthode GridSearchCv de Sklearn. Ainsi, pour chaque modèle nous définissons un ensemble de paramètres à tester. Les paramètres qui optimisent le mieux le score AUC seront retournés par l'algorithme GridSearchCv. Nous pourrons alors tester le modèle en fonction des paramètres retournés.

Les paramètres à tester

Parmi les 6 modèles, on définit les différents paramètres à tester :

LogisticRegression :

- C : Inverse de la force de régularisation
- penalty : Norme choisie
- solver : Algorithme utilisé

KNeighborsClassifier :

- n_neighbors : Nombre de voisins
- weights : Fonction de poids utilisée pour la prédiction

RandomForestClassifier :

- criterion : Fonction utilisée pour mesurer la qualité d'une division.
- max_depth : Profondeur maximale de chaque arbre
- max_features : Le nombre de fonctionnalités à prendre en compte lors de la recherche de la meilleure répartition.
- n_estimators : Nombre d'arbres de la forêt
- random_state : Contrôle à la fois le caractère aléatoire du bootstrap des échantillons utilisés lors de la construction des arbres et l'échantillonnage des fonctionnalités à prendre en compte lors de la recherche de la meilleure répartition à chaque nœud

GradientBoostingClassifier :

- subsample : La fraction d'échantillons à utiliser pour ajuster les arbres individuels
- n_iter_no_change : Le nombre d'époques sur lesquelles aucune amélioration ne doit être observée pour s'arrêter.
- learning_rate : taux d'apprentissage, met à l'échelle l'effet de chaque arbre sur la prédiction globale.
- max_features (même chose que RandomForestClassifier)
- random_state (même chose que RandomForestClassifier)
- n_estimators (même chose que RandomForestClassifier)

- max_depth (même chose que RandomForestClassifier)

XGBClassifier :

- colsample_bytree: Sous-échantillonnage des colonnes
- gamma : Réduction minimale des pertes requise pour créer une partition supplémentaire sur un nœud feuille de l'arbre
- reg_alpha : Terme de régularisation L1 sur les poids.
- min_child_weight : Somme minimale de poids d'instance dans un child.
- n_estimators (même chose que RandomForestClassifier)
- max_depth(même chose que RandomForestClassifier)
- learning_rate (même chose que GradientBoostingClassifier)
- subsample (même chose que GradientBoostingClassifier)

LGBMClassifier :

- subsample_freq : Fréquence des fractions d'échantillons
- reg_lambda : Terme de régularisation L2 sur les poids
- num_leaves : Feuilles d'arbre maximales pour les bases runners
- min_split_gain : Réduction minimale des pertes requise pour créer une partition supplémentaire sur un nœud feuille de l'arbre.
- n_estimators (même chose que RandomForestClassifier)
- max_depth(même chose que RandomForestClassifier)
- learning_rate (même chose que GradientBoostingClassifier)
- colsample_bytree: (même chose que XGBClassifier)
- subsample (même chose que GradientBoostingClassifier)
- reg_alpha : (même chose que XGBClassifier)

Les paramètres retenus pour chaque modèles

Après une succession de test voici les paramètres retenus pour chaque modèles en fonction de la variable à prédire :

Assurance vie :

LogisticRegression : {'C': 0.1, 'max_iter': 1000, 'penalty': 'l2', 'solver': 'lbfgs'}

KNeighborsClassifier : {'n_neighbors': 50, 'weights': 'distance'}

RandomForestClassifier : {'criterion': 'gini', 'max_depth': 12, 'max_features': 2, 'n_estimators': 500, 'random_state': 46}

GradientBoostingClassifier : {‘max_features’: 0.3,’ n_iter_no_change’: 26, ‘random_state’: 12, ‘subsample’: 0.5, ‘n_estimators’: 177, ‘max_depth’: 2, ‘learning_rate’:0.1}

XGBClassifier : {'max_depth': 4, 'learning_rate': 0.1, 'colsample_bytree': 0.6, 'random_state': 29, 'subsample': 0.7, 'n_estimators': 90, 'min_child_weight': 8, 'gamma': 0.1}

LGBMClassifier : {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 4, 'min_split_gain': 0.3, 'n_estimators': 100, 'num_leaves': 10, 'reg_alpha': 1.2, 'reg_lambda': 1.2, 'subsample': 0.8, 'subsample_freq': 10, 'random_state': 9}

Retraite :

LogisticRegression : {'C': 1.0, 'max_iter': 70, 'penalty': 'l2', 'solver': 'liblinear'}

KNeighborsClassifier : {'n_neighbors': 71, 'weights': 'distance'}

RandomForestClassifier : {'criterion': 'entropy', 'max_depth': 8, 'max_features': 4, 'n_estimators': 200, 'random_state': 42}

GradientBoostingClassifier : {'max_features': 0.3, 'n_iter_no_change': 24, 'learning_rate': 0.1, 'random_state': 163, 'subsample': 0.5, 'n_estimators': 500, 'max_depth': 2}

XGBClassifier : {'max_depth': 2, 'learning_rate': 0.05, 'colsample_bytree': 0.6, 'random_state': 124, 'subsample': 0.9, 'n_estimators': 500, 'min_child_weight': 4, 'gamma': 0.1, 'reg_alpha': 1}

LGBMClassifier : {'colsample_bytree': 0.7, 'learning_rate': 0.1, 'max_depth': 4, 'min_split_gain': 0.3, 'n_estimators': 500, 'num_leaves': 20, 'reg_alpha': 1.2, 'reg_lambda': 1.2, 'subsample': 0.8, 'subsample_freq': 20, 'random_state': 47}

Validation

Résultats Jupyter Notebook

Une fois les meilleurs paramètres sélectionnés, on teste les modèles sur le jeu de données de test. Voici les score (roc_auc) obtenus pour les différents modèles (y compris le stacking) en fonctions des deux variables à prédire :

Assurance vie :

Modèles	Score
LogisticRegression	0.7397
KNeighborsClassifier	0.7231
RandomForestClassifier	0.7536
GradientBoostingClassifier	0.7529
XGBClassifier	0.7552
LGBMClassifier	0.7532
StackingClassifier	0.7542

Retraite :

Modèles	Score
LogisticRegression	0.8139
KNeighborsClassifier	0.7957
RandomForestClassifier	0.8390
GradientBoostingClassifier	0.8434
XGBClassifier	0.8387
LGBMClassifier	0.8377
StackingClassifier	0.8377

Observations Jupyter Notebook

Assurance vie :

Le modèle qui présente le score le plus élevé pour l'assurance vie est le XGBClassifier, cela semble logique car c'est en théorie le modèle le plus performant et celui qui performe le mieux sur ce genre données. On distingue clairement la différence entre les modèles simples (LogisticRegression et KNeighborsClassifier) et les modèles d'ensemble plus complexes (RandomForestClassifier, GradientBoostingClassifier, XGBClassifier, LGBMClassifier, StackingClassifier).

Retraite :

Le modèle qui présente le score le plus élevé pour la retraite est le GradientBoostingClassifier, cela semble aussi logique car c'est en théorie l'un des modèles les plus performants sur ce genre de données. Étonnamment, le XGBClassifier présente un score moins élevé, peut-être que le choix des paramètres doit être réétudié. Encore une fois, on distingue clairement la différence entre les modèles simples et les modèles d'ensemble plus complexes.

Résultats Kaggle

Modèles	Score sur 50% des données	Score sur l'ensemble des données
StackingClassifier (Assvie et Retraite)	0.7809	0.77585
GradientBoostingClassifier (Retraite) et XGBClassifier(Assvie)	0.78307	0.78031
GradientBoostingClassifier (Retraite) et XGBClassifier avec early stopping (Assvie)	0.78283	0.78036

Observations Kaggle

Le modèle qui a performé le mieux sur 50% des données est celui du GradientBoostingClassifier pour prédire la variable Retraite et XGBClassifier pour prédire la variable Assurance vie.

Lors de mes différents essais pour prédire la variable Assvie, j'avais fait le choix de ne pas ajouter d'early stopping pour le XGBClassifier car si celui-ci était ajouté dans les paramètres je ne pouvais pas effectuer le StackingClassifier. Après plusieurs tentatives, je me suis rendu compte que l'ajout de l'early stopping augmentait nettement le score du XGBClassifier seul, passant de 0.7552 à 0.7577. J'ai donc effectué une dernière soumission avec ce dernier dépassant de peu l'ancien modèle sur l'ensemble des données.

Améliorations possibles

Smote

Lors de la phase d'analyse des données, je me suis vite rendu compte que les données des variables à prédire sont inéquitablement répartis parmi les deux classes 'O' et 'N'. La variable Assvie présentant 8510 observations pour la classe 'N' et 2396 pour la classe 'O'. La variable Retraite, quant à elle, présente 10128 observations pour la classe 'N' et 778 pour la classe 'O'. Ce manque d'information pour la classe 'O' impacte clairement l'apprentissage du modèle et donc son efficacité. Le modèle est plus apte à prédire la classe majoritaire que la classe minoritaire. Cela se remarque clairement dans la matrice de confusion :

col_0	False	True
Assvie		
False	2715	85
True	666	133

Figure 1 : Matrice de confusion effectué après la prédiction de la variable Assvie via un GradientBoostingClassifier

On remarque clairement que la majorité du score de ce GradientBoostingClassifier provient de la bonne prédiction de la classe majoritaire qui est prédite correctement à 97%. La classe minoritaire, quant à elle, est prédite correctement à seulement 17%.

Pour régler ce problème, il existe une bibliothèque Python nommée Imbalanced-learn qui permet à l'aide de la méthode SMOTE de créer de nouvelles observations de la classe minoritaire en s'appuyant sur la technique des k plus proches voisins. Après avoir étudié son fonctionnement, j'ai testé la méthode SMOTENC (méthode smote meilleure sur les données catégorielle) sur nos données. Le résultat obtenu était très satisfaisant, voire trop satisfaisant, en fait le modèle faisait preuve d'overfitting et ne performait pas sur les données inconnues. Malheureusement, je n'ai pas eu le temps de régler ce problème d'overfitting. La méthode du smote reste donc une piste à étudier pour de futures améliorations.

Features Engineering

L’une des pistes d’améliorations possibles est la création de nouvelles variables à partir des variables existantes. La création de nouvelles variables peut permettre au modèle d’effectuer de meilleures prédictions. Malheureusement, nos variables sont dans la majorité catégorielles, je n’ai donc pas trouvé comment créer de nouvelles variables à partir des variables existantes (impossibilité d'additionner deux variables …etc). Cela reste néanmoins une piste à étudier.