FOUILLES DE DONNEES MASSIVES

Détection de fraudes bancaires

Frintz Elisa, Madi Corodji Jacky, Rives Alexandre

16/01/2022 – Master 2 SISE

Lien Git-Hub : https://github.com/Elisa-Frintz/Projet_Fraude_bancaire

Table des matières

Introduction

Dans le cadre de notre cours de fouille de données massives, nous avons eu l'occasion de travailler sur des données réelles de fraudes issues d'une enseigne de la grande distribution ainsi que de certains organismes bancaires (FNCI et Banque de France).
L'objectif de notre projet est de proposer la meilleure solution possible pour détecter des fraudes de chèques bancaires dans un contexte déséquilibré de données massives. Pour cela, nous avons choisi le langage de programmation python qui est l'un des plus utilisés en science des données.
Tout au long de ce projet, nous avons essayé de faire très attention aux temps de calcul, à la reproductibilité et à la compréhension métier des résultats obtenus.
Afin de présenter au mieux notre travail, nous allons tout d'abord présenter le contexte et les données utilisées. Ensuite, nous détaillerons l'organisation de notre code ainsi que notre stratégie pour travailler efficacement avec une telle volumétrie de données. Enfin, nous expliquerons chaque méthode utilisée avant de comparer leurs performances. Pour conclure, nous discuterons des meilleures méthodes et de leurs perspectives d'évolution.

I. Contexte et présentation des données

Notre fichier contient la liste de l'ensemble des transactions de chèques bancaires qui ont eu lieu dans une enseigne de la grande distribution entre le 1er février et le 30 novembre 2018. Ces données proviennent de deux organismes bancaires différents (FNCI et Banque de France). Nous avons ainsi remarqué que la ligne n°1956361 est une duplication du nom des variables qui est sûrement dû à la fusion des deux fichiers issus des deux organismes. Il faut donc en tenir compte pour l'importation des données.
Notre fichier contient 4 646 773 observations et 23 variables et sa taille est de 1 514 848 142 octets.

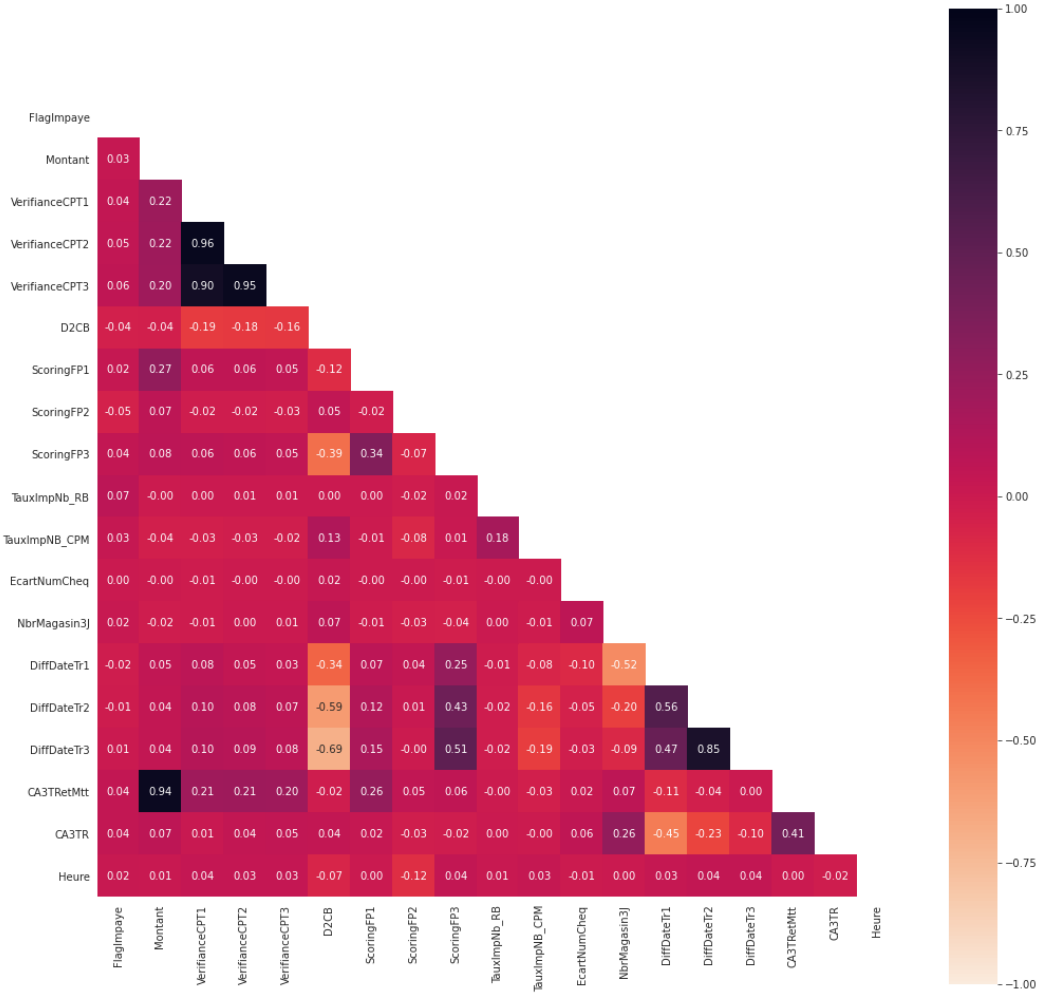
L'importation de ce fichier à l'aide la librairie pandas étant longue et coûteuse, notre stratégie est de l'importer une première fois afin de réaliser tous les traitements nécessaires (suppression de variables, changement de type etc..). Après le nettoyage de nos données, nous avons transformé ce dataframe propre en format dask qui permet une meilleure gestion des fichiers de grande dimension.

Afin de gagner un maximum de temps et d'espace mémoire, nous avons exporter ces dataframes dask propres aux format parquet à l'aide la librairie fastparquet. A partir de là, nous pouvons réimporter ces fichiers directement au format dask très rapidement.

Nous avons donc exporté les fichiers suivants :

- Le dataframe propre d'apprentissage,
- Le dataframe propre de test,
- Le dataframe propre d'apprentissage avec sur-échantillonnage,
- Le dataframe propre d'apprentissage avec sous-échantillonnage.

Matrice de corrélation :



Les changements apportés à notre jeu de données

- Variable "Heure" → Suppression parce qu'elle fait doublon avec la variable "DateTransaction".
- Variable "DateTransaction" → Séparation en deux variables "Date" et "Heure".
- Variables "ZIBZIN" et "IDAvisAutorisationCheque" → Suppression parce que ce sont des variables d'identifiants, elles n'apportent donc pas d'information pertinente.
- Variable "CodeDecision" → En observant le dataframe, nous avons remarqué que la variable qualitative "CodeDecision" possède une valeur à 4 alors que dans la description des données, cette variable ne peut prendre que les valeurs 0, 1, 2 ou 3. Nous faisons donc le choix de retirer l'observation du dataframe.
- Variables "VerifianceCPT1", "VerifianceCPT2" et "VerifianceCPT3" → Nous supprimons ces trois variables parce qu'elles sont constantes à 0 dans le dataframe de test.
- Variable "CA3TRetMtt" → Suppression dans 97,5 % des cas, il s'agit de la somme de "Montant" et de "CA3TR".

<u>Transformation des données : Centrage-réduction</u>

La méthode de transformation fit_transform() est utilisé sur les données d'entraînement afin que nous puissions mettre à l'échelle les données d'entraînement et également apprendre les paramètres de mise à

l'échelle de ces données. En utilisant la méthode transform(), nous pouvons utiliser la même moyenne et la même variance que celles calculées à partir de nos données d'entraînement pour transformer nos données de test. Ainsi, les paramètres appris par notre modèle à l'aide des données d'apprentissage nous aideront à transformer nos données de test.
Dans notre cas, nous utilisons la méthode fit_transform() sur nos données d'entraînement initiales, puis nous utilisons la méthode transform() pour appliquer ce même centrage-réduction sur les données de test mais également sur les données d'apprentissage qui ont été sur-échantillonnées. En effet, le sur-échantillonnage de nos données aurait de grandes chances de faire varier les paramètres de centrage-réduction par rapport aux données d'entraînement initiales.

II. Organisation du code et stratégie pour gérer la volumétrie

Afin de rendre gérer au mieux la volumétrie de ce projet et de faciliter la reproductibilité, nous avons décidé d'organiser notre code en trois parties de la manière suivante :

A) Exploration et nettoyage des données

Le premier notebook jupyter contient les statistiques descriptives nécessaires à la compréhension de notre jeu de données. C'est ici que nous l'avons nettoyé afin qu'il corresponde au mieux à notre utilisation (suppression de variables, recodage des variables qualitatives, sur-échantillonnage, sous-échantillonnage, etc.). C'est également ici que nous avons exporté nos jeux de données propres au format dask et parquet afin qu'il soit plus facilement ré-importable par la suite (gain de temps et de lignes de codes).

B) Création des modèles

Le second notebook jupyter contient la création des modèles de machine learning que nous avons tenté d'appliquer à notre jeu de données. Ici, nous importons directement les jeux de données propres construits à l'étape précédente. Nous n'avons donc plus qu'à réaliser quelques quelques traitements rapides tels que séparer la variable cible des variables explicatives et centrer-réduire les données d'apprentissage.
Pour construire les modèles, nous utilisons la librairie dask_ml qui nous permet d'entraîner nos modèles de manière parallèle, ce qui nous fait gagner énormément de temps. Toujours dans un souci de gain de temps et de reproductibilité des calculs, nous utilisons la librairie pickle afin d'exporter nos modèles entraînés au format modele.sav.

Calcul du chiffre d'affaires

Afin de déployer nos modèles entraînés sur nos données de test, nous avons créé une fonction déploiement() qui elle-même fait appel à une autre fonction Calcul_CA().
- Calcul_CA(Montant, yReel, yPred) est une fonction qui répond à la question bonus de notre projet :
- Lorsque nous prédisons une fraude et que ça en est bien une (Vrai positif), le chiffre d'affaires est de 0.
- Lorsque nous prédisons une bonne transaction et que ça en est bien une (Vrai négatif), le chiffre d'affaires est égal au montant de la transaction. Tout s'est bien passé.
- Lorsque nous prédisons une fraude et alors que ça n'en est pas (Faux positif), le chiffre d'affaires est de 80 % du montant de la transaction. En effet, dans ce cas l'enseigne perd la confiance de son client qui ne reviendra sûrement pas. Les faux positifs sont donc très importants à minimiser.

- Lorsque nous prédisons une bonne transaction alors qu'il s'agit d'une fraude, le chiffre d'affaires est égal à 1 – exp(1/montant). Cela fait donc perdre du chiffre d'affaires à l'entreprise mais cela reste très faible comme perte.

Grâce à cette analyse, nous comprenons beaucoup mieux les enjeux de la détection de fraude et cela nous donne une information cruciale à utiliser dans le paramétrage de nos algorithmes : L'enseigne a besoin de détecter un maximum de fraude. Pourtant, il est plus avantageux pour elle de laisser passer certaines fraudes plutôt que de prédire une fraude alors que ça n'en est pas une. Afin de maximiser le chiffre d'affaires, il faut donc minimiser le taux de faux négatif et le taux de faux positif, mais il est quand même beaucoup plus important de minimiser le taux de faux positif quitte même à ce que le taux de faux négatif augmente légèrement.

Déploiement du modèle sur les données de test

Afin de faciliter le déploiement des modèles sur les données de test et de réduire le nombre de lignes de code, nous avons créé la fonction deploiement(modele, XTest, yTest, scale, features).
Celle-ci prend en entrée :
- modele : le nom/chemin du modèle à déployer, les données
- XTest: les données explicatives de test
- yTest : la variable cible à prédire, cela nous permet de calculées les performances du modèle.
- X_ca (pour les algorithmes non supervisés) : données non centrée-réduites pour le calcul du CA.
- scale : un booléen qui permet à l'utilisateur de dire si les données du modèle sont centrer-réduire ou pas.
- features : un booléen qui permet d'afficher ou non un graphique d'importance des variables (nécessité de connaître le modèle car tous ne peuvent pas afficher ce graphique).

Cette fonction fonctionne de la manière suivante :
Si l'option scale = True, nous centrons et réduisons les données XTest à l'aide du scaler déjà entraîné plus tôt. Cela permet d'appliquer le modèle sur le même type de données à partir duquel il a été construit.
Ensuite, nous importons le modèle déjà entraîné et nous l'appliquons à nos données de test pour calculer les valeurs prédites et leur score.
A partir de là et avec l'aide des données réelles de l'échantillon de test yTest, nous calculons et affichons les performances du modèle qui sont : la matrice de confusion, le rappel, la précision, l'aire sous la courbe ROC : AUC (uniquement pour les algorithmes supervisés) et le chiffre d'affaires généré. Ces données de performance sont retournées par la fonction afin qu'elles puissent être stockées et comparées entre elles.

C) Déploiement et comparaison des modèles

Le troisième et dernier notebook jupyter contient uniquement le déploiement de tous les modèles. Il est assez rapide d'exécution puisqu'il fait à appel aux modèles précédemment entraînés. Suite à cela, nous pouvons comparer les différents modèles à l'aide de graphiques comparatifs basés sur différents indicateurs de performance tels que le chiffre d'affaires généré, l'aire sous la courbe ROC (AUC), le rappel, la précision et l'accuracy.

III. Méthodes utilisées

A) Méthodes d'apprentissage supervisées

Régression logistique

Nous avons décidé d'utiliser la régression logistique car elle possède de nombreux hyperparamètres qui nous permettent d'essayer plusieurs modèles en même temps. En effet, entre les solveurs, les pénalisations, les degrés des pénalités et les stratégies d'échantillonnage, nous avons pu tester de nombreuses situations.

Régression logistique par défaut :
Nous avons dans un premier temps commencé par une régression logistique par défaut. De ce fait, le solveur utilisé est le 'liblinear' qui n'est pas conseillé pour les grands jeux de données. Les résultats s'avèrent en effet médiocre.

Régression logistique solveur SAGA :
La documentation de scikit-learn préconise d'utiliser le solveur SAGA pour les grands jeux de données. Les résultats sont sans appel, l'AUC augmente et le nombre de fraude détectées est multiplié par 3. En revanche nous augmentons également le nombre de faux positifs et nous avons pu voir précédemment que cela n'était pas une bonne chose.

Régression logistique avec différentes pénalisations :
Nous avons ensuite pensé à essayer sur les différentes pénalisations. Les résultats n'ont que très peu bougé.

Régression logistique avec sur-échantillonnage SMOTE :
La méthode d'échantillonnage SMOTE apporte un avantage sur la reconnaissance de fraudeurs mais augmente considérablement le nombre de bons payeurs reconnus comme fraudeurs, et nous avons vu que cela n'était pas une bonne chose. Même si l'AUC est plus élevée, dénoncer une plus grande quantité des personnes non-fraudeuses comme fraudeuses n'est pas ce qui est recherché.

Pour conclure, il semblerait que la régression logistique ne soit pas l'algorithme le plus adapté pour ce genre d'analyse.

Méthodes basées sur les arbres de décision

Après de nombreuses recherches, nous avons remarqué que les algorithmes basés sur des arbres de décision étaient préconisés pour de la détection de fraudes bancaires.

Arbre de décision

Nous avons donc commencé par entraîner un arbre de décision simple qui est très sensible au sur-apprentissage. Celui-ci donne des résultats mitigés et bien moins bon que les méthodes de régression logistique vu précédemment.

Forêt aléatoire

La forêt aléatoire est une méthode plus robuste et donc moins sujette au sur-apprentissage que qu'un arbre de décision seul. Plus coûteuses en calcul, dans notre cas nous avons utilisé 10 arbres de décision. L'algorithme apprend alors sur 10 arbres de décision différents et retourne 10 prédictions. La prédiction retenue est celle qui a été le plus souvent prédite. Nous remarquons que les résultats sont excellents, le nombre de faux positifs est très réduit.

Adaboost avec Arbre de décision

Adaboost est une méthode d'apprentissage basée sur les apprenants faibles. C'est-à-dire qu'il va se servir de ses erreurs pour progresser petit à petit. Il est également moins sujet au sur-apprentissage que d'autres algorithmes. Dans notre cas, il est basé sur un arbre de décision de profondeur 1 et on a fixé le nombre maximal d'estimateurs à 10. Les résultats de cette méthode sont encore meilleurs que la forêt aléatoire précédente. L'aire sous la courbe ROC est légèrement plus faible les performances globales sont bien meilleures. En effet, il ne prédit aucun faux positif et c'est une excellente nouvelle ! Le nombre de faux négatifs est légèrement plus haut mais cela nous permet de maximiser le chiffre d'affaires. Nous rappelons qu'il est beaucoup plus avantageux de minimiser au maximum le nombre de bons clients identifiés comme fraudeurs quitte même à augmenter légèrement le nombre de fraudeurs non détectés.

B) Méthodes d'apprentissage non supervisées et semi-supervisées

Nous avons voulu tester des algorithmes semi et non supervisés afin de les comparer aux supervisés. Le postulat dans ce contexte est de considérer les fraudes, fortement minoritaires, comme des données aberrantes, ou « outliers ». Les fraudeurs ont des comportements inhabituels et cela se retrouvera dans le dataset où les individus fraudeurs auront des caractéristiques sensiblement différentes de ceux en règle.

Les trois algorithmes sélectionnés sont : One Class SVM, Isolation Forest et Autoencoder.

Pour le premier et le troisième, nous partons sur deux approches :

- Une approche non supervisée appelée « Outliers Detection » qui se base sur la détection de observations anormales, qui n'appartiennent pas aux clusters de données concentrées.

- Une approche semi-supervisée appelée « Novelty Detection ». Pour cette méthode, l'algorithme n'est entraîné que sur une classe (la classe majoritaire), et, lorsqu'on lui présente de nouvelles données, il classera comme « novelty », toute observation aberrante.

One Class SVM

Nous avons choisi cet algorithme car il est adapté à la problématique qui est de détecter une très faible proportion d'une classe en la séparant des observations non frauduleuses.

Parce que le jeu de données est de grande taille, nous avons utilisé SGD One Class SVM du module sci-kit learn, associé à une approximation de kernel afin de reproduire les résultats de la version « de base » tout en

augmentant fortement les performances de temps de traitement (environ 30 secondes pour entraîner un modèle sur un peu moins de 4 millions de lignes).

One class SVM par défaut

Nous testons dans un premier temps l'algorithme sans kernel en spécifiant l'hyperparamètre « nu ». Il correspond à la proportion d'outliers dans le jeu de données. Dans notre cas il est connu donc nous pouvons le fixer.

Les résultats sont loin de donner satisfaction.

One class SVM avec kernel

Nous répétons l'étape précédente avec cette fois-ci l'association d'une approximation de kernel. On choisit un kernel polynomial de degré 5. Les résultats, sans être exceptionnels, sont nettement supérieurs.

On crée une fonction « Model_OneClassSVM » qui va permettre d'entraîner et évaluer différentes combinaisons de paramètres pour différent kernel afin de trouver le meilleur.

Cette fonction prend en entrée : X_train, X_test, yTest, kernel, degree = None et, pour différentes valeurs de nu, de gamma et de n_components (nombre de dimensions de l'espace dans lequel les observations sont projetées par le kernel), retourne dans un dataframe différentes métriques d'évaluation (précision, rappel, accuracy, f1_score, auc, …). Pour chaque résultat, la fonction retourne les paramètres, ce qui nous permettra d'exporter le meilleur modèle.

Trois kernels sont sélectionnés : RBF, Polynomial et Sigmoid, pour lesquels nous appliquons la fonction.

Les dataframes obtenus sont exportés au format .csv.

One Classe SVM + kernel : approche Outliers Detection

Les résultats obtenus montrent que le kernel RBF est moins performant que les deux autres qui eux, donnent des résultats quasiment similaires.

De manière générale, les résultats bien que largement au-dessus de la moyenne, sont assez faibles au niveau de la précision, avec un taux de faux positifs élevé dans le contexte bancaire.

One Classe SVM + kernel : approche Novelty Detection

Nous appliquons de nouveau la fonction « Model_OneClassSVM » en entraînant les modèles sur la classe non frauduleuse uniquement. L'intuition laisserait penser que de cette façon, les modèles auront plus de faciliter à détecter les fraudes.

Cependant les résultats sont quasiment similaires pour les kernels Polynomial et Sigmoid. Pour le kernel RBF, le rappel est plus élevé, mais la précision a fortement diminué par rapport à l'approche non-supervisée.

Finalement, c'est en utilisant le kernel Sigmoid en outlier detection qu'on obtient les meilleures performances. Il y a cependant très peu d'écart avec le kernel Polymial et non n'observons pas de différence entre les deux approches semi et non-supervisées.

A noter que le kernel sigmoid est assez similaire à la fonction sigmoid de la régression logistique.

Isolation Forest

L'algorithme Isolation Forest est semblable à Random Forest dans le sens où il est basé sur un ensemble d'arbres de décision, à ceci près qu'il se concentre sur la détection d'outliers.

Pour un échantillon défini et à chaque itération, il sélectionne au hasard une variable sur laquelle les observations sont comparées et classées, selon une valeur discriminante, prise également aléatoirement.

Les outliers sont des individus qui se distinguent fortement de l'ensemble de l'échantillon, ils sont donc détectés très rapidement. Ainsi, le nombre de branches pour atteindre une feuille (noeud terminal) sera sensiblement plus petit que ceux de la classe dominante.

Isolation Forest par défaut

Un premier entraînement avec les paramètres par défaut, sauf pour le paramètre « contamination » qui représente aussi la part d'outliers dans le jeu de données, donne des résultats très mauvais.

Isolation Forest avec paramètres optimisés

Comme pour One Class SVM, on crée une fonction permettant de trouver la meilleure combinaison de paramètres parmi les suivants :

- Contamination
- Max_samples : la taille du sous-échantillons sur lequel discriminer les observations
- N_estimators : le nombre d'arbres
- Max_features : le nombre maximal de variables explicatives sur lesquels comparer les observations

La meilleure combinaison trouvée fournit des performances très faibles, ce qui est surprenant. Une hypothèse serait que certains fraudeurs n'ont pas de comportement sensiblement différent des personnes en règle, ce qui rend difficilement la discrimination. D'autant que la valeur de discrimination est choisie de manière aléatoire.

Autoencoder

L'autoencoder est un algorithme non supervisé qui utilise un réseau de neurones pour reconstruire un jeu de données de grande dimension. Il identifie les caractéristiques discriminantes et constitue donc une solution potentiellement intéressante. Il réduit la dimensionnalité de l'échantillon dans les couches cachées, lors de l'apprentissage, puis reconstruit le jeu de données en essayant de prédire la variable cible en couche de sortie.

Ici, les deux approches de détection ont été utilisées, de nombreux modèles ont été entraînés en essayant différentes combinaisons de paramètres tels que : le nombre d'epochs, le nombre de neurones de la couche cachée, les fonctions d'activation cachée et de sortie, le taux de drop_out ou encore la part donnée à la norme L2.

Quelque soit l'approche et quelque soient les paramètres, cet algorithme n'arrive pas à faire mieux que des performances légèrement au-dessus de la moyenne. Il n'est pas adapté à notre problématique. Ici aussi, peut-être qu'il a du mal à différencier des individus des deux classes aux caractéristiques proches.

IV. Comparaison des performances de chaque modèle

Voici un récapitulatif de tous les modèles d'apprentissage supervisé que nous avons testés :

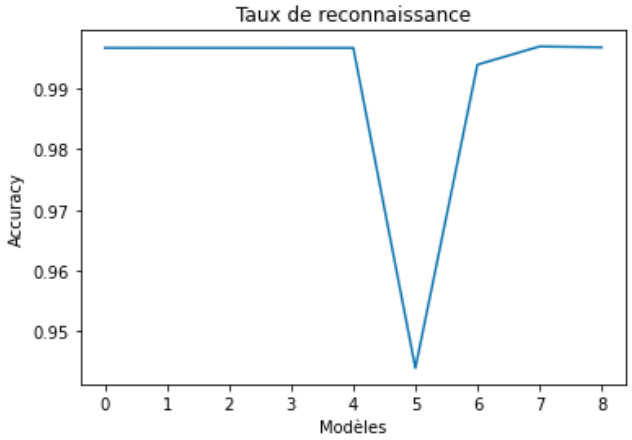
	Modèle	Données	CA	AUC	Rappel	Précision	Accuracy
modele1.sav	LogisticRegression(random_state=0)	Initiales + réduction Centrage	-45836958.17	0.8327	0.6657	0.9449	0.9967
modele2.sav	LogisticRegression(random_state=0, solver='saga')	Initiales + réduction Centrage	-45841059.47	0.8309	0.6622	0.9487	0.9967
modele3.sav	LogisticRegression(C=0.1, penalty='l1', random_state=0, solver='saga')	Initiales + réduction Centrage	-45844719.47	0.8309	0.6622	0.9489	0.9967
modele4.sav	LogisticRegression(C=0.1, random_state=0, solver='saga')	Initiales + réduction Centrage	-45841059.47	0.8309	0.6622	0.9487	0.9967
modele5.sav	LogisticRegression(C=0.01, l1_ratio=0.5, penalty='elasticnet', random_state=0, solver='saga')	Initiales + réduction Centrage	-45844667.27	0.8309	0.6622	0.9487	0.9967
modele6.sav	LogisticRegression(random_state=0, solver='saga')	SMOTE + réduction Centrage	-44935012.82	0.8822	0.8194	0.1168	0.9439
modele7.sav	DecisionTreeClassifier(random_state=0)	Initiales	45838330.04	0.8381	0.6795	0.6492	0.9939
modele8.sav	RandomForestClassifier(n_estimators=10, random_state=0)	Initiales	45884533.32	0.8345	0.6692	0.9773	0.9969
modele9.sav	AdaBoostClassifier(base_estimator= DecisionTreeClassifier(max_depth=1, random_state=0), n_estimators=10, random_state=0)	Initiales	45889118.39	0.8183	0.6366	1.0	0.9968

Voici un récapitulatif de tous les modèles d'apprentissage semi et non-supervisé que nous avons testés, précision et rappel pour la classe frauduleuse :

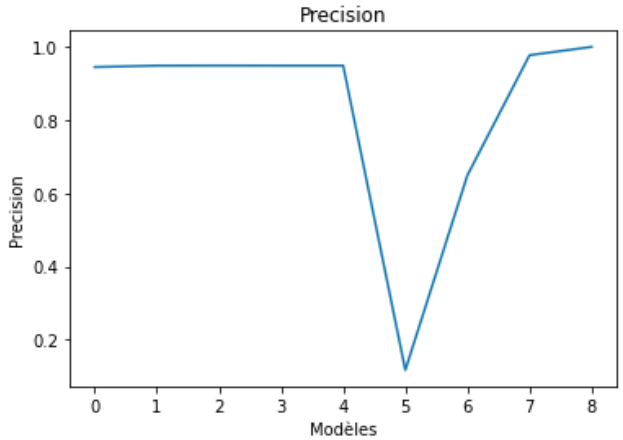
	Modèle	Données	CA	AUC	Rappel	Précision	Accuracy
modele8_rbf.sav	Pipeline(steps=[('nystroem', Nystroem(gamma=0.0001, n_components=15, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.0065, random_state=2, tol=1e-07))])	Initiales + Centrage réduction	45521104.44	0.8202	0.6441	0.6032	0.9931
modele8_poly.sav	Pipeline(steps=[('nystroem', Nystroem(degree=5, gamma=0.0001, kernel='poly', n_components=25, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.015, random_state=2, tol=1e-07))])	Initiales + Centrage-réduction	45863615.79	0.8254	0.6528	0.7417	0.9949
modele8_sigmoid.sav	Pipeline(steps=[('nystroem', Nystroem(gamma=0.0001, kernel='sigmoid', n_components=25, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.015, random_state=2, tol=1e-07))])	Initiales + Centrage-réduction	45864667.39	0.8252	0.6523	0.7535	0.9951
modele8_rbf_nov.sav	Pipeline(steps=[('nystroem', Nystroem(gamma=0.05, n_components=5, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.0065, random_state=2, tol=1e-07))])	Fit sur classe 0 + Centrage-réduction	45431727.72	0.8448	0.6963	0.4817	0.9907
modele8_poly_nov.sav	Pipeline(steps=[('nystroem', Nystroem(degree=5, gamma=0.0001, kernel='poly', n_components=25, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.0065, random_state=2, tol=1e-07))])	Fit sur classe 0 + Centrage-réduction	45863504.36	0.8151	0.6323	0.7411	0.9948
modele8_sigmoid_nov.sav	Pipeline(steps=[('nystroem', Nystroem(gamma=0.0001, kernel='sigmoid', n_components=25, n_jobs=-2, random_state=2)), ('sgdoneclasssvm', SGDOneClassSVM(max_iter=2000, nu=0.015, random_state=2, tol=1e-07))])	Fit sur classe 0 + Centrage-réduction	45857845.51	0.8272	0.6563	0.7438	0.9949
modele9.sav	IsolationForest(contamination=0.015, max_features=16, max_samples=300000, n_estimators=50, n_jobs=-2, random_state=2)	Initiales	45428474.29	0.8210	0.6552	0.3072	0.9839
Auto_outl	auto_outl = AutoEncoder(epochs=15, contamination=0.0065, hidden_neurons =[64, 32, 16, 8, 16, 32, 64], hidden_activation = 'sigmoid', batch_size = 768, dropout_rate = 0.4, random_state = 2, l2_regularizer = 0.4, loss = 'BinaryCrossentropy', output_activation = 'sigmoid', preprocessing = False)	Initiales + centrage réduction	45884533.32	0.7828	0.57	0.54	0.99
Auto_nov	Identiques à Auto_outl	initales + centrage réduction	45889118.39	0.8468	0.70	0.43	0.99

Comparaison de la performance des modèles supervisés

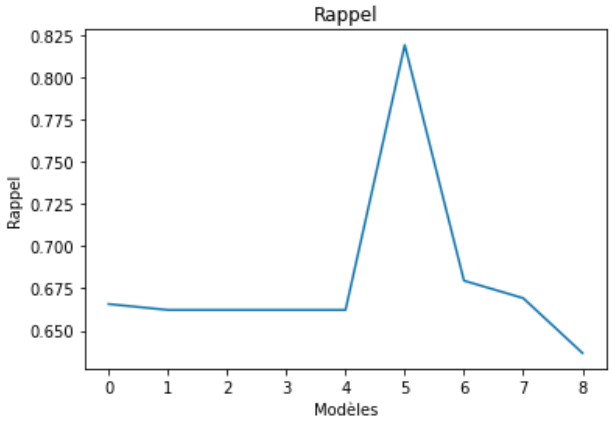
Taux de reconnaissance



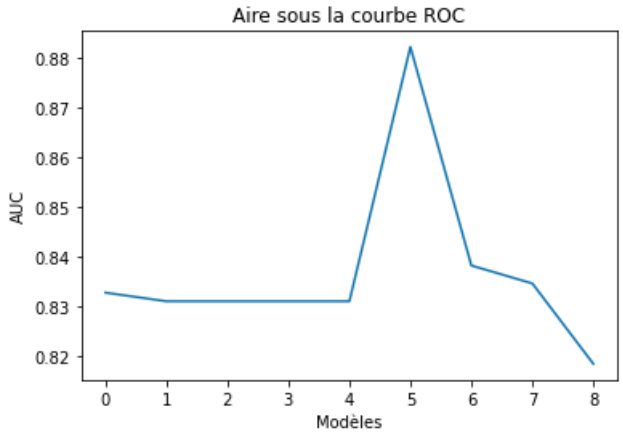
Précision



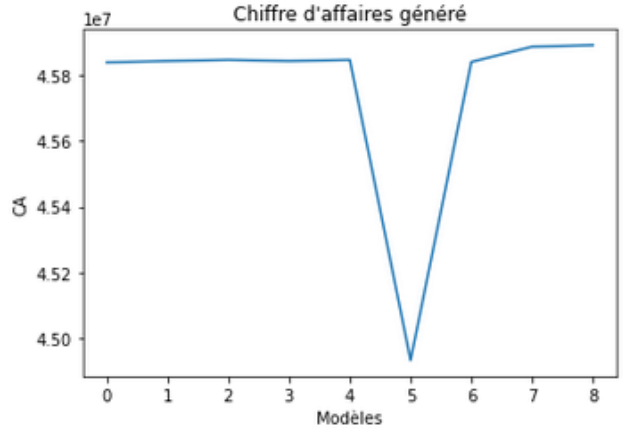
Rappel



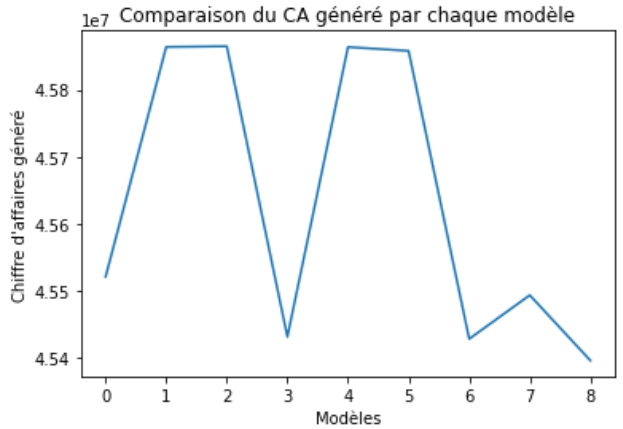
Aire sous la courbe ROC



Chiffre d'affaires généré pour les modèles supervisés



Chiffre d'affaires généré pour les modèles semi et non-supervisés



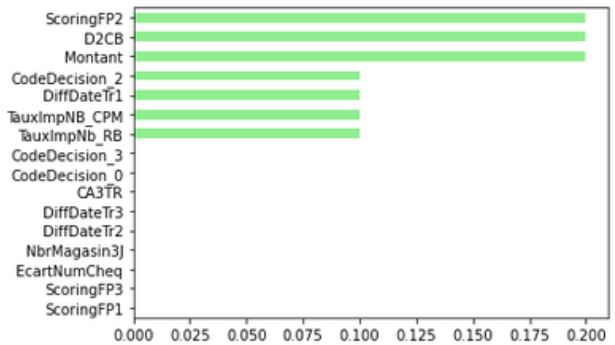
Pour réaliser ce projet de détection de fraudes bancaires, nous avons choisi de comparer nos modèles à l'aide de quatre éléments différents : le chiffre d'affaires généré, l'aire sous la courbe ROC (auc), le rappel, la précision et l'accuracy. Ce choix est arbitraire, il existe bien d'autres métrique d'évaluation.

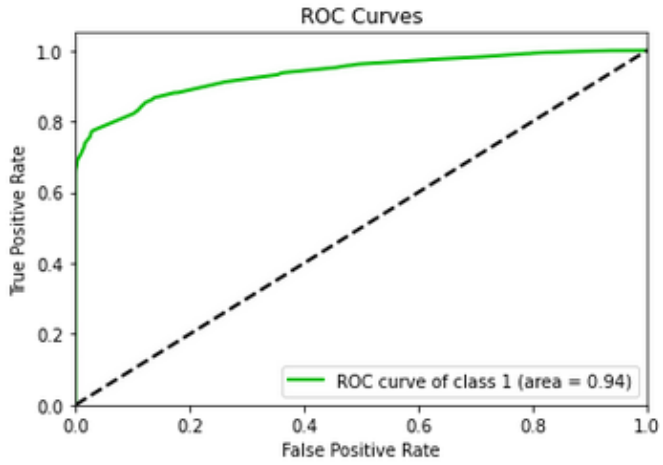
Selon la nature de notre problème à résoudre et les besoins métiers, nous allons privilégier certaines métriques au lieu d'autres. Dans notre cas, des experts métiers ont pu estimer le chiffre d'affaires qui seraient généré dans chacune des situations (Vrai positif / Faux positif / Vrai négatif / Faux négatif). Cela nous donne donc un indicateur de performance personnalisé bien plus robuste que les autres.
Il s'agit donc d'une question de point de vue :
- Si l'on regarde le chiffre d'affaires généré ou la précision, le meilleur modèle est le modèle 9.
- Si l'on regarde l'aire sous la courbe ROC ou le taux de rappel, le meilleur modèle est le modèle 6.
- Si l'on regarde l'accuracy, le meilleur modèle est le modèle 8.

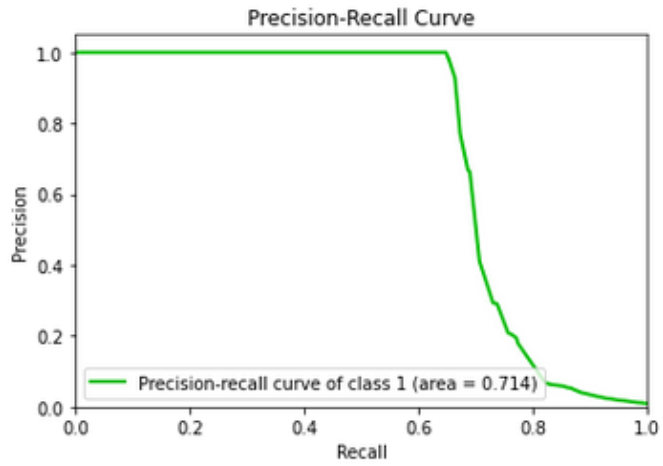
Dans notre cas de détection de fraudes bancaires, la priorité est de maximiser le chiffre d'affaires généré tout en minimisant au maximum le nombre de bons clients considérés comme fraudeurs (précision). Nous pouvons donc conclure qu'ici, le meilleur modèle est le modèle 9. Les algorithmes semi et non-supervisés sont en deçà des algorithmes supervisés en termes de performance.

Détail du meilleur algorithme : le modèle 9

```
----------------------------------------------------------
Modèle :
 AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1,
                                                          random_state=0),
                    n_estimators=10, random_state=0)
----------------------------------------------------------
Estimateurs :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    740837
           1       1.00      0.64      0.78      6573

    accuracy                           1.00    747410
   macro avg       1.00      0.82      0.89    747410
weighted avg       1.00      1.00      1.00    747410

----------------------------------------------------------
Matrice de confusion :
 [[740837      0]
 [  2388   4185]]
----------------------------------------------------------
Auc Score :
 0.8183477863989046
----------------------------------------------------------
Chiffre d'affaire = 45889118.4 euros
----------------------------------------------------------
Importance des variables :
```







V. Conclusion

Pour conclure, nous pouvons dire que nous avons beaucoup apprécié ce projet. En effet, cela nous a permis de nous familiariser avec de nombreuses méthodes et concepts tels que : la gestion des données très volumineuses, l'optimisation des temps de calcul, la reproductibilité, la sauvegarde et le déploiement de modèles déjà entraîner etc...

De plus, ce projet lié à des données réelles de fraude nous a permis de comprendre beaucoup de choses notamment liées l'évaluation des modèles qui dépend énormément du contexte. En effet, ici le but est de détecter le maximum de fraude mais pas à tout prix. En effet, les indicateurs de performance classique (AUC, accuracy, etc.) sont importants et peuvent nous donner de bons résultats. Cependant, dans notre contexte, nous avons développé un indicateur de performance personnalisé (le chiffre d'affaires) et celui-ci est bien meilleur que les indicateurs classiques dans ce cas-là. En effet, au-delà de la reconnaissance des fraudes, il est préférable pour l'enseigne de laisser passer certains fraudeurs plutôt que de considérer des bons clients comme fraudeurs. Cela a un meilleur impact sur le chiffre d'affaires. Le contexte est ici très important car sans cela, nous aurions peut-être choisi un modèle moins performant.

Le meilleur modèle que nous avons réussi à produire est un modèle Adaboost avec Arbre de décision. En effet, c'est celui qui maximise le plus le chiffre d'affaires généré et la précision. De plus, nous avons également remarqué que ce modèle minimise le plus possible le nombre de faux positifs qui tombe même à 0 sur nos données de test. Après avoir cru à du sur-apprentissage, nous nous sommes finalement dit que cela correspondait plutôt bien à la réalité.

VI. Perspectives d'évolution

Une des perspectives d'évolution de notre projet consisterait à détecter automatiquement si les modèles déployés s'utilisent sur des données centrées-réduites et si l'on peut afficher un graphique d'importance des variables. Actuellement, la fonction de déploiement nécessite que l'utilisateur connaissent le modèle construit afin de mettre les bons paramètres dans fonctions de déploiement.
Une autre perspective d'évolution de notre projet serait serait de créer une interface graphique utilisable par un utilisateur lambda.