Ensembles classifiers in Class Imbalance Learning

Rebecca Leygonie et Nemanja Kostadinovic

Master 2 Big Data et Fouille de Données
Université Paris 8
2020-2021

1 Introduction

De nos jours, les entreprises collectent des données sous toutes formes et demandent ensuite à les utiliser.
Chaque cas d'usage est unique en son genre, les données sont d'une telle variété... Des données d'un même type ne doivent pas obligatoirement suivre le même prétraitement.
Bien qu'en réalité, il est difficile de trouver une solution parfaitement adaptée à un cas d'usage précis, il existe des méthodes répondant à des thématiques.
Aujourd'hui, nous parlons d'un problème très courant : le déséquilibre des données.
Le cas le plus fréquent pour être confronté à des données déséquilibrées est la détection d'anomalies. Ce cas d'usage n'est pas anodin car une des premières raisons pour lesquelles les entreprises s'intéressent à l'intelligence artificielle est à des fins de sécurisation de leurs systèmes par une surveillance quasi continue. Autrement-dit, la génération de données en continu et la mise à disposition de celles-ci permet de surveiller les fluxs et donc de détecter des anomalies qui pourraient coûter cher à l'entreprise.
Ainsi, le problème des données déséquilibrées est courant et très problématique et c'est pourquoi il nous a paru intéressant d'en discuter.
À titre indicatif, il n'y a pas de délimitation stricte de la répartition du travail de chacun.

2 Le problème des données déséquilibrées

Les données déséquilibrées font généralement référence à un problème de classification où les classes ne sont pas représentées de manière égale.
Par exemple, on peut avoir un problème de classification à 2 classes (binaire) avec 100 instances (lignes). Au total, 80 instances sont étiquetées comme étant de classe 1 et les 20 instances restantes sont étiquetées comme étant de classe 2. Il s'agit alors d'un ensemble de données déséquilibré et le rapport entre les instances de classe 1 et de classe 2 est de 80 : 20 ou, plus précisément, de 4 : 1.

Le problème du déséquilibre de classe intervient sur les problèmes de classification à deux classes ainsi que sur les problèmes de classification à plusieurs classes. La plupart des techniques que nous présentons par la suite peuvent être utilisées dans les deux cas mais nous nous concentrons sur le cas d'une classification binaire.

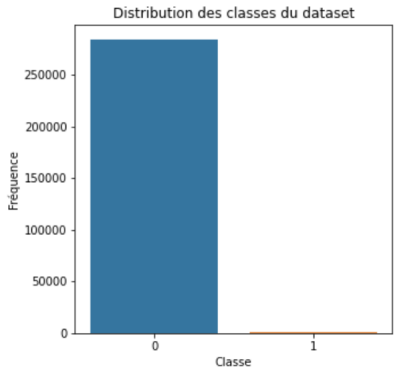
La plupart des ensembles de données de classification n'ont pas exactement le même nombre d'instances dans chaque classe, mais une petite différence n'a souvent pas d'importance.
Il existe des problèmes où un déséquilibre entre les classes n'est pas seulement courant, il est attendu. Par exemple, dans les ensembles de données comme ceux qui caractérisent les transactions frauduleuses, il y a un déséquilibre. La

grande majorité des transactions seront dans la classe *"non frauduleuse"* et une très petite minorité sera dans la classe *"frauduleuse"*. C'est le cas d'usage que nous utilisons par la suite.
Les données que nous utilisons pour illustrer les méthodes/modèles que nous détaillons par la suite sont des données bancaires. Plus précisément, des transactions bancaires avec un label indiquant si elles sont frauduleuses ou non.

:

	Time	V2	V3	V4	V5	V6	V7	V8	Amount	Class
0	0.0	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	149.62	0
1	0.0	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	2.69	0
2	1.0	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	378.66	0
3	1.0	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	123.50	0
4	2.0	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	69.99	0
...	...	...	...	...	...	...	...	...	...	...
284802	172786.0	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	0.77	0
284803	172787.0	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	24.79	0
284804	172788.0	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	67.88	0
284805	172788.0	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	10.00	0
284806	172792.0	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	217.00	0

La répartition des classes est très déséquilibrée, en effet, la classe 0 : "transaction non frauduleuse" contient 284 315 cas contre 492 pour la classe 1 : "transaction frauduleuse".
Plus formellement, le jeu de données est représenté par 99,8% de cas 0 et 0,2% de cas 1.



Lorsque l'on applique un modèle de classification sur des données, la mesure d'évaluation du modèle la plus courante est l'*accuracy*, qui estime le taux de données bien classées.
Dans le cas d'une classification binaire (0 et 1), on appelle "vrais positifs", les données dont le label prédit par le modèle est 1 lorsque leur label est vraiment 1 et "vrais négatifs" les données dont le label prédit par le modèle est 0 lorsque

leur label est vraiment 0.
La valeur de l'*accuracy* est alors donnée par :

$$accuracy = \frac{vrais\_positifs + vrais\_négatifs}{vrais\_positifs + faux\_positifs + vrais\_négatifs + faux\_négatifs}$$

Le paradoxe avec cette mesure d'évaluation, c'est que lorsque les données sont déséquilibrées, les mesures d'*accuracy* indiquent que le modèle est très performant (par exemple $accuracy = 90\%$), alors que cette valeur ne reflète que le taux de bien classés pour la classe sur-représentée.

Lorsqu'on entraîne des modèles sur un ensemble de données déséquilibré, s'ils obtiennent des valeurs d'*accuracy* excellentes, (98% par exemple, 98% des cas étant de la classe 0 ("non frauduleuse")), c'est parce qu'ils examinent les données et décident intelligemment que la meilleure chose à faire est de toujours prévoir la "classe 0" et d'obtenir une *accuracy* élevée. Ainsi, le modèle fait ce qu'on appelle du sur-apprentissage, *over-fitting* en anglais. Autrement-dit, le modèle, ayant appris sur trop de données dont le label est 0, il n'est pas capable de généraliser son apprentissage et classifie toutes (ou presque) les nouvelles données à 0.

3 Re échantillonage

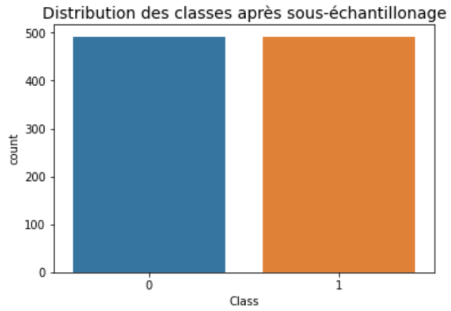
La toute première réaction possible face à un ensemble de données déséquilibré est de considérer que les données ne sont pas représentatives de la réalité : si tel est le cas, nous supposons que les données réelles sont presque équilibrées mais qu'il existe un biais de proportion dans les données recueillies. Dans ce cas, il est presque obligatoire d'essayer de collecter des données plus représentatives.
Dans le cas où l'ensemble des données est déséquilibré parce que la réalité est ainsi, il existe plusieurs méthodes pour équilibrer l'ensemble de données avant d'y appliquer un classificateur. Nous utilisons ici deux méthodes :

- le sous-échantillonnage, qui consiste à échantillonner à partir de la classe majoritaire afin de ne garder qu'une partie de ces points. l'inconvénient du sous-échantillonnage est qu'il supprime des données, donc des informations qui peuvent être précieuses.
- le sur-échantillonnage, qui consiste à reproduire certains points de la classe minoritaire afin d'accroître sa cardinalité.

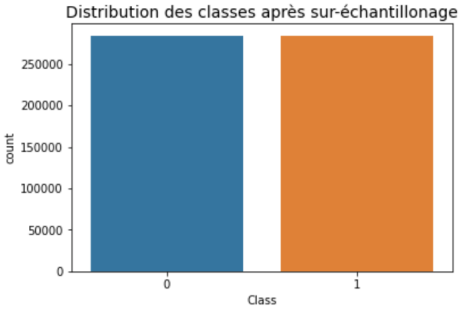
Le fait d'équilibrer les données est une astuce pour améliorer les performances réelles des modèles de classification. Cependant, ce n'est pas la seule solution ou du moins, elle peut être couplée à d'autres, comme le fait d'utiliser des méthodes ensemblistes par exemple.

Dans notre cas d'usage, afin de déterminer la meilleure méthode où le meilleur couplage de méthode, nous testons les méthodes ensemblistes sur les données d'origine mais aussi sur des données sous échantillonnées et sur-échantillonnées.

Pour sous-échantillonner nos données, nous utilisons la méthode de la bibliothèque *imblearn* : imblearn.under_sampling.RandomUnderSampler()
Après avoir sous-échantilloné les données, les classes 0 et 1 sont toutes les deux représentés pars 492 données.



Pour sur-échantillonner nos données, nous utilisons la méthode imblearn.over_sampling.RandomOverSampler()
Après avoir sur-échantillonné les données, les classes 0 et 1 sont toutes les deux représentées par 284 315 données.



Par la suite, nous détaillons le fonctionnement des modèles utilisés et montrons, pour chaque modèle, son application sur les trois jeux de données par une matrice de confusion, qui permet de visualiser les données mal classées pour chaque label.
Par la suite, une section est dédiée à la comparaison et à la discussion des différentes méthodes utilisées.

4 Application des méthodes ensemblistes

Maintenant que nous avons vu les problèmes qu'engendre un déséquilibre des données dans l'apprentissage d'un modèle, nous allons voir différentes techniques pour pallier ce problème.

Dans le cadre de notre problématique de détection d'opérations frauduleuses pour la banque, il est plus important de réduire le nombre de faux négatifs, c'est-à-dire le nombre d'opérations bancaires frauduleuses non détectées comme tel. Chaque modèle va donc afficher une matrice de confusion pour pouvoir comparer son résultat aux autres.

4.1 Les méthodes de bagging

4.1.1 Bagging Classifier

Le *Bagging Classifier* [12] peut être appelé méta-estimateur d'ensemble qui est créé en ajustant plusieurs versions de l'estimateur de base, formé avec un ensemble de données de formation et modifié en utilisant une technique d'échantillonnage par sac (données échantillonnées par remplacement) ou autres. La technique d'échantillonnage par sac peut aboutir à un ensemble de données d'entraînement constitué d'un double de l'ensemble de données ou d'un ensemble de données unique. Cette technique d'échantillonnage est également appelée agrégation bootstrap. Le prédicteur final (également appelé *Bagging* classificateur) combine les prédictions faites par chaque estimateur/classificateur par vote. En créant chacun des estimateurs individuels, on peut configurer le nombre d'échantillons et/ou les caractéristiques qui doivent être prises en compte lors de l'ajustement.

Le classificateur aide à réduire la variance des estimateurs individuels en introduisant la randomisation dans l'étape de formation de chacun des estimateurs et en formant un ensemble à partir de tous les estimateurs. Si la variance est élevée, cela signifie que la modification de l'ensemble de données de formation entraîne une modification importante de l'estimateur construit ou formé.

Le classificateur de formation peut être appelé de l'une des façons suivantes, en fonction de la technique d'échantillonnage utilisée pour créer les échantillons de formation :

- **Échantillonnage par collage :** Lorsque les sous-ensembles de données aléatoires sont prélevés de manière aléatoire sans remplacement (bootstrap = Faux), l'algorithme peut être appelé *Pasting*
- **Mise en sac de l'échantillon :** Lorsque les sous-ensembles aléatoires de données sont tirés avec remplacement (bootstrap = Vrai), l'algorithme peut être appelé *Bagging*. Il est également appelé agrégation bootstrap.

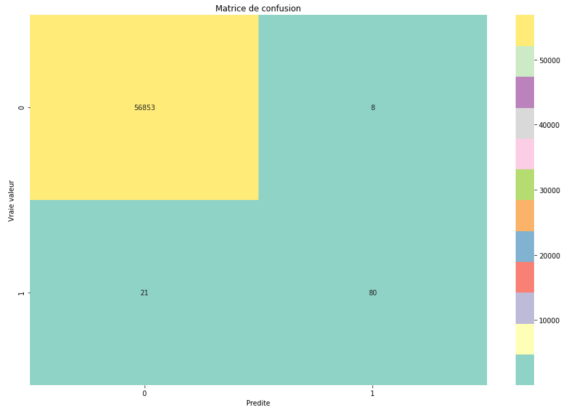
- **Subspace aléatoire :** Lorsque les sous-ensembles aléatoires de données sont tirés au sort, l'algorithme peut être appelé "sous-espace aléatoire".
- **Patches aléatoires :** Lorsque les sous-ensembles d'échantillons et de caractéristiques sont tirés au sort, l'algorithme peut être appelé "patchs aléatoires".

Dans ce projet, le *Bagging Classifier* est créé en utilisant Sklearn.ensemble.BaggingClassifier() avec les paramètres par défaut. Soit nombre d'estimateurs fixé à 10 , max_features fixé à 1.0, et max_samples fixé à 1.0 et la technique d'échantillonnage utilisée est par défaut (bagging). La méthode appliquée est celle des patchs aléatoires car les échantillons et les caractéristiques sont tirés de manière aléatoire.

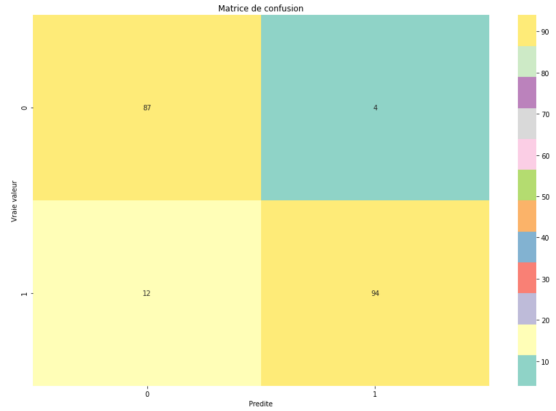
Le script pour appeler le modèle, l'entraîner sur les données *train* et le tester sur les données *test* et récupérer les scores est :

```
clf = BaggingClassifier()
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

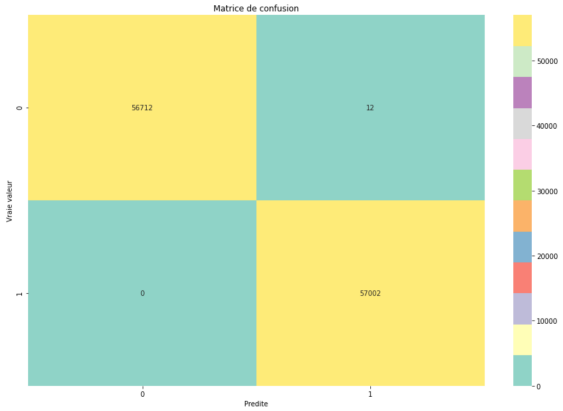
Nous appliquons premièrement le modèle sur les données d'origine :
L'*accuracy* obtenue avec ce modèle sur ces données est **0.9994** et la matrice de confusion :



Sur les données sous-échantillonnées, l'*accuracy* est **0.9368** et la matrice de confusion :



Sur les données sur-échantillonnées, l'*accuracy* est **0.9998** et la matrice de confusion :



4.1.2 Utilisation du *bagging* avec des algorithmes linéaires

L'utilisation des méthodes linéaires produit des résultats médiocres, au point que nous choisissons de ne pas vous les présenter ici. A la place, nous avons eu l'idée d'utiliser la puissance du *bagging* afin de faire collaborer plusieurs modèles simples de séparation linéaire.
Les algorithmes choisis sont la régression logistique, la classification naïve bayésienne et le *OneClass SVM*.

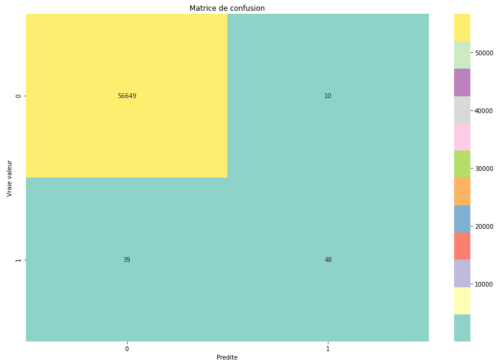
La régression logistique est adaptée à notre problématique car nous devons classer nos données en deux catégories, opération bancaire frauduleuse ou non. Il s'agit d'un algorithme de classification, qui est utilisé lorsque la variable à prédire est catégorielle. L'idée de la régression logistique est de trouver une relation entre les caractéristiques et la probabilité d'un résultat particulier. Dans la régression logistique, nous n'ajustons pas directement une ligne droite à nos

données comme dans la régression linéaire. Au lieu de cela, nous ajustons une courbe en forme de S, appelée *Sigmoïde*, à nos observations. Nous utilisons 400 estimateurs de régression logistique.

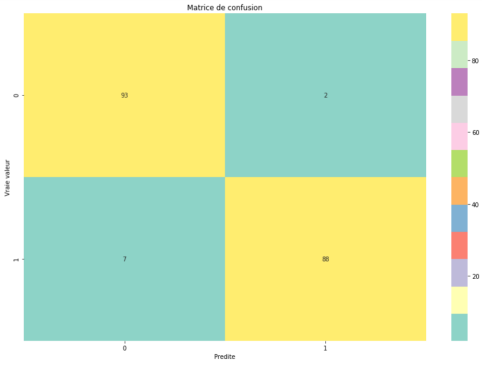
Le script pour appeler le modèle, l'entraîner sur les données *train*,le tester sur les données *test* et récupérer les scores est :

```
clf = BaggingClassifier(LogisticRegression(random_state=0, solver='lbfgs'), n_estimators
= 400, oob_score = True, random_state = 90)
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

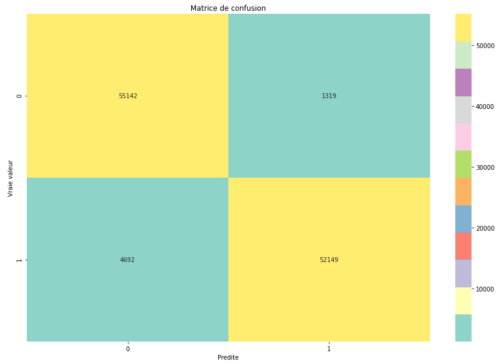
Nous appliquons premièrement ce modèle sur nos données d'origine :
L'*accuracy* obtenue est **0.9991** et la matrice de confusion :



Sur les données sous-échantillonnées, l'*accuracy* est **0.9368** et la matrice de confusion :



Sur les données sur-échantillonnés l'*accuracy* est **0.9993** et la matrice de confusion :

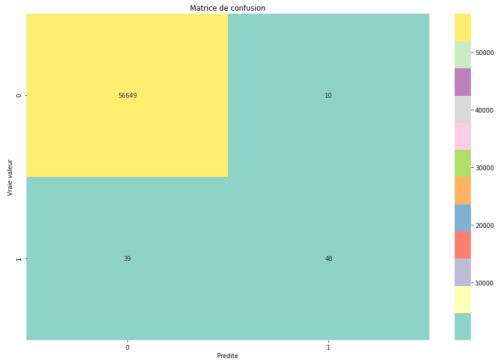


Le second modèle est la classification naïve bayésienne [14], qui calcule la probabilité qu'un échantillon soit d'une certaine catégorie, sur la base de connaissances préalables.
Il utilise le théorème naïf de Bayes, qui suppose que l'effet d'une certaine caractéristique d'un échantillon est indépendant des autres caractéristiques. Cela signifie que chaque caractère d'un échantillon contribue indépendamment à déterminer la probabilité de classification de cet échantillon, en produisant la catégorie de la plus forte probabilité de l'échantillon.

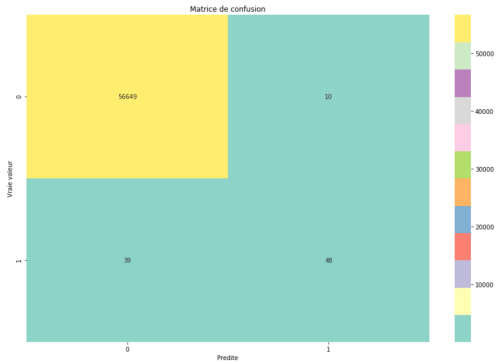
Nous avons fixé le nombre d'estimateurs à 400 car c'était un bon compromis entre le temps d'exécution et le résultat obtenu.
Le script pour appeler le modèle, l'entraîner sur les données *train*, le tester sur les données *test* et récupérer les scores est :

```
clf = BaggingClassifier(GaussianNB(), n_estimators = 400, oob_score = True, random_state = 90))
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

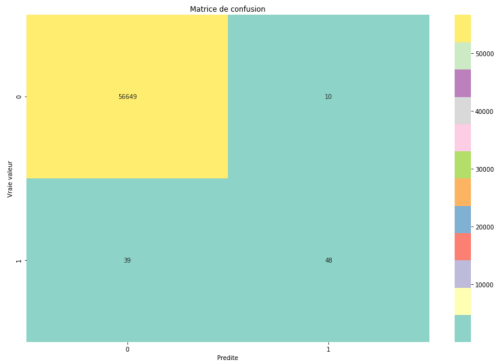
Nous appliquons premièrement ce modèle sur les données d'origines, l'*accuracy* obtenue est **0.9991** et la matrice de confusion :



Sur les données sous-échantillonnées l'*accuracy* est **0.9210** et la matrice de confusion :



Sur les données sur-échantillonnées l'*accuracy* est **0.9470** et la matrice de confusion :



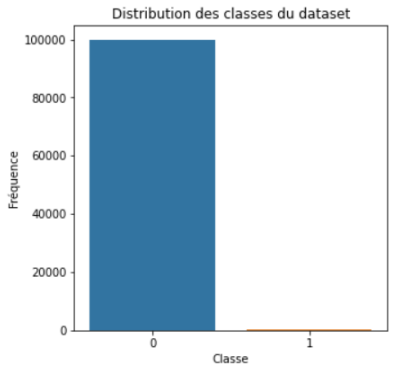
Le dernier modèle est le *OneClass SVM* [13]. Le problème abordé par ce modèle comme le dit la documentation, est la détection de l'anomalie dans les données. L'idée de la détection d'observations anormales est de détecter des événements rares, c'est-à-dire des événements qui se produisent rarement, et donc dont vous avez très peu d'échantillons ce qui rejoint notre problème de départ. Alors, comment décider de ce qu'est une anomalie ?
De nombreuses approches sont basées sur l'estimation de la densité de probabilité des données. L'anomalie correspond aux échantillons où la densité de probabilité est "très faible". Le degré de rareté dépend de l'application.
Or, l'algorithme utilise les *SVM* qui sont des méthodes à marge maximale, c'est-à-dire qu'ils ne modélisent pas une distribution de probabilité. L'idée est ici de trouver une fonction qui soit positive pour les régions à forte densité de points, et négative pour les petites densités. Les anomalies sont ici les opérations frauduleuses qui apparaissent donc très rarement dans nos données, ce sont des anomalies.

L'utilisation de cet algorithme est très coûteuse en ressource système, c'est pourquoi nous avons échantillonné nos données en considérant seulement un ensemble de 100000 échantillons des données d'origine pour l'apprentissage. De plus, nous avons réduit le nombre d'estimateurs à cinq.

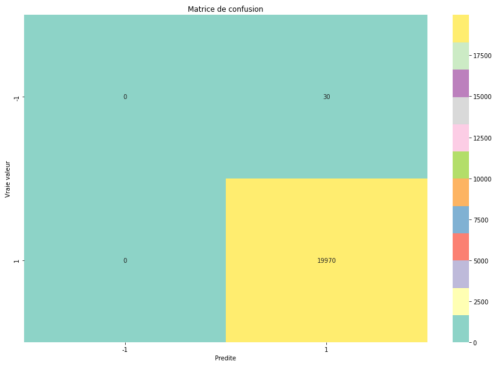
Le script pour appeler le modèle, l'entraîner sur les données *train*, le tester sur les données *test* et récupérer les scores liés aux prédictions est:
clf = BaggingClassifier(OneClassSVM(verbose=True), n_estimators = 5, oob_score = True, random_state = 90)
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
Une particularité dans l'utilisation du modèle *OneClassSVM* est que les deux classes doivent avoir comme label -1 et 1. Pour palier ce problème, nous transformons la classe 0 en 1 et 1 en -1.

Nous appliquons premièrement ce modèle sur les données d'origines, ré-échantillonnées aléatoirement car nous voulons avoir moins de données afin de permettre à l'algorithme de se terminer en un temps raisonnable.

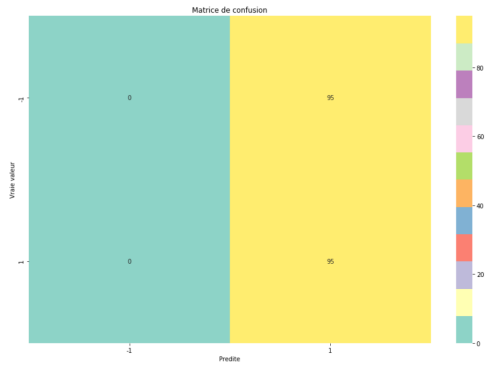
Voici la distribution des données après ré-échantillonage aléatoire :



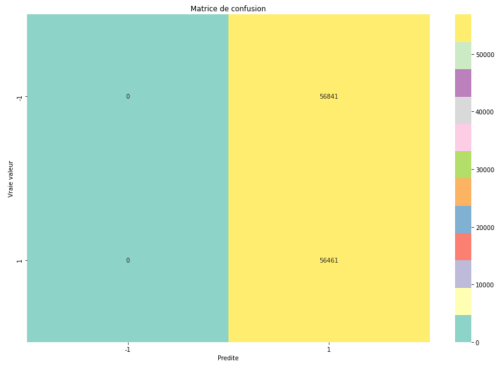
L'*accuracy* obtenue avec ce modèle sur ces données est **0.9985** et la matrice de confusion est :



Sur les données sous-échantillonnées, *accuracy* est **0.5** et la matrice de confusion :



Sur les données sur-échantillonnées, *accuracy* est **0.4983** et la matrice de confusion :



Le bagging permet donc d'agréger les résultats des modèles faibles (linéaires) et de les utiliser comme un seul modèle de classification.

4.1.3 Forêts d'arbres décisionnels (Random Forest Classifier)

L'algorithme *Random Forest Classifier* (forêts aléatoires) [9] [10], est constitué d'un grand nombre d'arbres de décision individuels qui fonctionnent comme un ensemble. Chaque arbre individuel de la forêt aléatoire produit une prédiction de classe. La classe ayant le plus de votes devient la prédiction de notre modèle. Le concept fondamental derrière l'algorithme *Random Forest Classifier* est simple mais puissant.
Dans le domaine de la science des données, la raison pour laquelle le modèle de la forêt aléatoire fonctionne si bien est la suivante : un grand nombre de modèles relativement peu corrélés (arbres de décision) fonctionnant ensemble sont plus performants que n'importe lequel des modèles constitutifs individuels.
La faible corrélation entre les modèles est la clé. Les modèles non corrélés peuvent produire des prévisions d'ensemble plus précises que n'importe lequel des modèles individuels. La raison de cet effet est que les arbres se protègent mutuellement de leurs erreurs individuelles (tant qu'ils ne s'égarent pas constamment tous dans la même direction). Si certains arbres peuvent se tromper, les autres auront raison, de sorte qu'en tant que groupe, les arbres sont capables de se guider dans la bonne direction (vers la bonne décision de classe).

Les conditions préalables à la bonne performance des forêts aléatoires sont les suivantes :

1. Il doit y avoir un signal réel dans nos caractéristiques afin que les modèles construits à l'aide de ces caractéristiques fonctionnent mieux que les suppositions aléatoires.

2. Les prédictions (et donc les erreurs) faites par les arbres individuels doivent avoir de faibles corrélations les unes avec les autres.

Alors comment la forêt aléatoire fait-elle en sorte que le comportement de chaque arbre individuel ne soit pas corrélé avec le comportement des autres arbres du modèle ?
Elle utilise les deux méthodes suivantes :

- Bagging (bootstrap)
- Choix aléatoire des caractéristiques

Bagging (bootstrap)
Les arbres de décision sont très sensibles aux données sur lesquelles ils ont effectués leur apprentissage, de petites modifications de l'ensemble d'apprentissage peuvent entraîner des structures d'arbre sensiblement différentes. La forêt aléatoire en tire profit en permettant à chaque arbre individuel de prélever au hasard un échantillon de l'ensemble de données avec remplacement, ce qui donne des arbres différents. Ce processus est connu sous le nom de *bagging*.
Avec le *bagging*, nous ne remplaçons pas les données d'apprentissage par des

morceaux plus petits et nous apprenons chaque arbre sur un morceau différent. Au contraire, si nous avons un échantillon de taille N, nous continuons à alimenter chaque arbre avec un ensemble de données d'apprentissage de taille N (sauf indication contraire). Mais au lieu des données d'entraînement originales, nous prenons un échantillon aléatoire de taille N avec remplacement.

Choix aléatoire des caractéristiques
Dans un arbre de décision seul, lorsqu'il est temps de diviser un nœud, nous considérons toutes les caractéristiques possibles et choisissons celle qui produit le plus de séparation entre les observations du nœud gauche et celles du nœud droit. En revanche, chaque arbre d'une forêt aléatoire ne peut choisir que parmi un sous-ensemble aléatoire de caractéristiques. Cela oblige à une variation encore plus grande entre les arbres du modèle et, en fin de compte, entraîne une corrélation plus faible entre les arbres et une plus grande diversification du modèle d'ensemble.

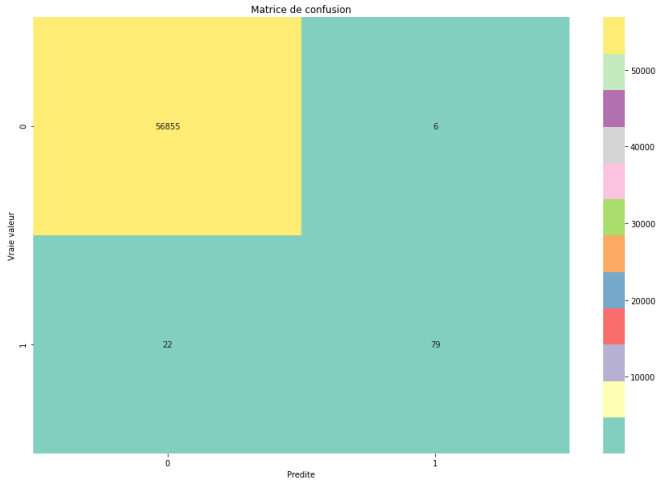
Les forêts d'arbres décisionnels sont très utilisées car elles sont beaucoup moins sensibles aux observations aberrantes présentes dans nos données. Elles ne font pas non plus d'hypothèses fortes sur la distribution sous-jacente des données et peuvent implicitement gérer la colinéarité des caractéristiques, car si vous avez deux caractéristiques très similaires, le gain d'information résultant du fractionnement sur l'une des caractéristiques épuisera également le pouvoir prédictif de l'autre caractéristique. Mais le facteur le plus déterminant pour décider d'utiliser une forêt d'arbres décisionnels qu'un autre algorithme est probablement la possibilité avec cet algorithme de mieux comprendre la relation que les caractéristiques ont avec la cible et le degré d'influence qu'elles ont.

Pour appliquer ce modèle sur nos données, nous utilisons la méthode
sklearn.ensemble.RandomForestClassifier()

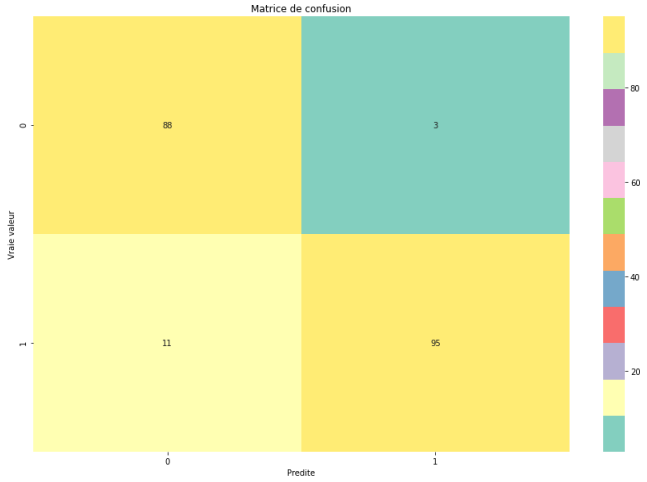
Le script pour initialiser le modèle, l'entraîner et le tester est :

```
clf=RandomForestClassifier()
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

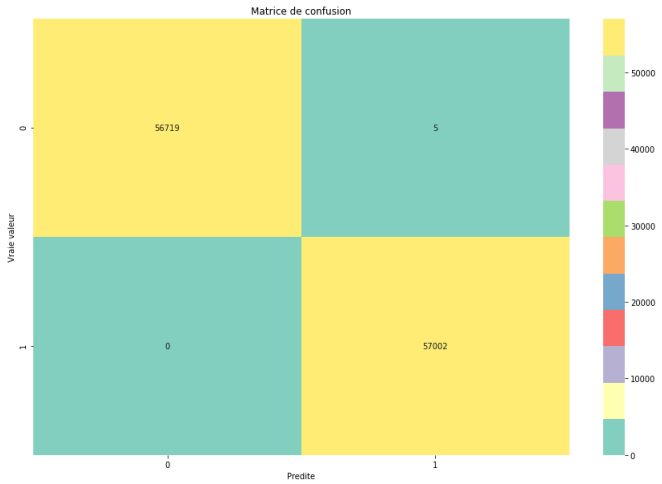
L'application du modèle sur les données de transactions bancaires d'origine obtient une *accuracy* de **0.9995** et la matrice de confusion suivante :



Sur les données de transactions bancaires sous-échantillonnées, l'*accuracy* est **0.9315** et la matrice de confusion :



Sur les données de transactions bancaires sur-échantillonnées, l'*accuracy* est **0.9999** et la matrice de confusion est la suivante :



4.1.4 ExtraTrees Classifier

Le modèle *Extremely Randomized Trees* [8], aussi appelé *ExtraTrees Classifier* est semblable à un classificateur *Random Forest* mais la façon dont sont construits les arbres est différente.
Chaque souche de décision sera construite selon les critères suivants :

1. Toutes les données disponibles dans l'ensemble de formation sont utilisées pour construire chaque souche.
2. Pour former le nœud racine ou n'importe quel nœud, la meilleure division est déterminée en recherchant dans un sous-ensemble de caractéristiques choisies au hasard de taille $\sqrt{nombredecaractéristiques}$. La répartition de chaque caractéristique sélectionnée est choisie au hasard.
3. La profondeur maximale de la souche de décision est de 1.

Dans un classificateur *Extra Trees*, les caractéristiques et les divisions sont choisies au hasard, d'où l'expression "arbre extrêmement aléatoire". Comme les divisions sont choisies au hasard pour chaque caractéristique, il est moins coûteux en termes de calcul qu'une forêt aléatoire.
Le classificateur *Extra Trees* fonctionne de la même manière que le *Random Forest*. Cependant, il y a des différences de performance. À savoir : Les arbres de décision présentent une variance élevée, les forêts aléatoires une variance moyenne, et les arbres supplémentaires une faible variance.

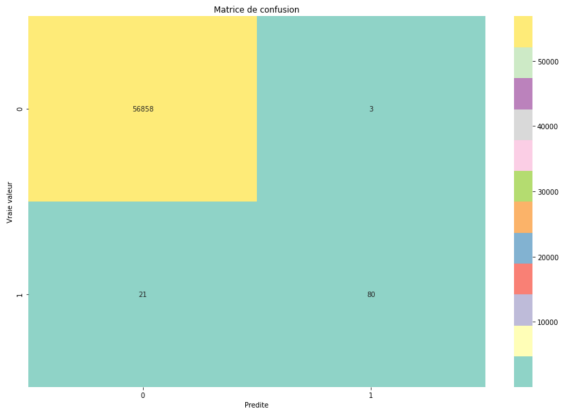
Pour l'application du modèle, nous avons utilisé la librarie *Sklearn*, plus précisément, la méthode *Sklearn.ensemble.ExtraTreesClassifier()*.

Le script pour initialiser le modèle, l'entraîner sur les données train, le tester

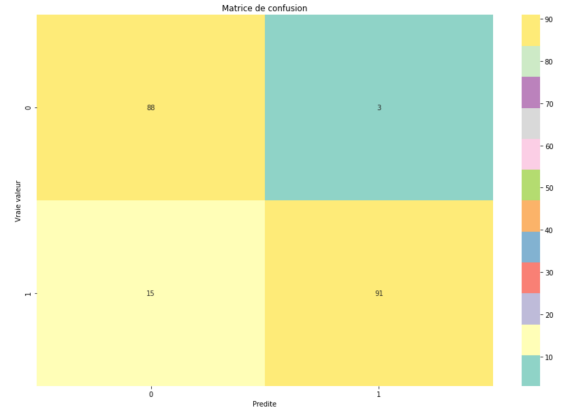
sur les données test et récupérer les prédictions/scores est :

```
clf=ExtraTreesClassifier()
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

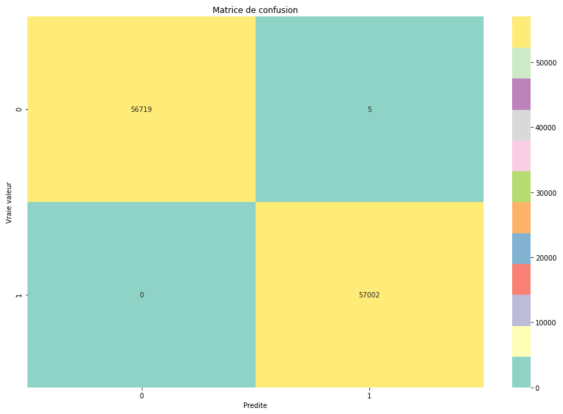
L'application du modèle sur les données d'origine obtient une *accuracy* de **0.9995** et la matrice de confusion suivante :



Sur les données sous-échantillonnées, l'*accuracy* est **0.9210** et la matrice de confusion est la suivante :



Sur les données sur-échantillonnées, l'*accuracy* est **0.9999** et la matrice de confusion est la suivante :



4.2 Les méthodes de boosting

4.2.1 AdaBoost Classifier

AdaBoost (Adaptive Boosting) [5] est une technique de *boosting* très populaire qui vise à combiner plusieurs classificateurs faibles pour construire un classificateur performant.

Un seul classificateur peut ne pas être capable de prédire avec précision la classe d'un objet, mais lorsque nous regroupons plusieurs classificateurs faibles, chacun apprenant progressivement des objets mal classés des autres, nous pouvons construire un modèle performant. Le classificateur mentionné ici peut être n'importe lequel des classificateurs de base, des arbres de décision (par défaut) à la régression logistique, etc.
Nous désignons par classificateur "faible" les classificateurs qui fonctionnent mieux que les suppositions aléatoires mais qui ne sont pas performants pour attribuer des classes à des objets.

Plutôt que d'être un modèle en soi, *AdaBoost* peut être appliqué sur n'importe quel classificateur pour tirer les leçons de ses défauts et proposer un modèle plus précis.
Comprenons le fonctionnement, d'*AdaBoost*. Les noeuds de décision sont comme les arbres d'une forêt aléatoire, mais pas "pleinement développés". Ils ont un nœud et deux feuilles. *AdaBoost* utilise une forêt de ces noeuds de décision plutôt que des arbres. Les noeuds de décision divisent les exemples en deux sous-ensembles basés sur la valeur d'une caractéristique. Chaque noeud choisit une caractéristique et un seuil puis divise les exemples en deux groupes de part et d'autre du seuil.
Les noeuds sont pas un bon moyen de prendre des décisions. Un arbre combine les décisions de toutes les variables pour prédire la valeur cible. Une noeud, par contre, ne peut utiliser qu'une seule variable pour prendre une décision.
Pour trouver le noeud de décision qui correspond le mieux aux exemples, nous

pouvons essayer toutes les caractéristiques de l'entrée ainsi que tous les seuils possibles et voir lequel donne la meilleure précision.
Bien qu'il semble naïvement qu'il y ait un nombre infini de choix pour le seuil, deux seuils différents ne sont significativement différents que s'ils placent certains exemples de part et d'autre de la frontière de séparation. Pour essayer toutes les possibilités, il est possible de trier les exemples en fonction de la caractéristique en question et essayer un seuil se situant entre chaque paire d'exemples adjacents.

Pour mieux comprendre son fonctionnement, commençons par considérer un ensemble de données avec N points, ou lignes, dans notre ensemble de données.

$$x_i \in R^n, y_i \in -1, 1$$

- n est la dimension des nombres réels, ou le nombre d'attributs dans notre ensemble de donnée
- x est l'ensemble des points de données
- y est la variable cible

Nous calculons les échantillons pondérés pour chaque point de données. *AdaBoost* attribue un poids à chaque exemple d'apprentissage afin de déterminer sa signification dans l'ensemble de données d'apprentissage. Lorsque les poids attribués sont élevés, cet ensemble de points de données d'apprentissage est susceptible d'avoir une plus grande influence sur l'ensemble d'apprentissage. De même, lorsque les poids attribués sont faibles, ils ont une influence minimale sur l'ensemble des données d'apprentissage.

Au départ, tous les points de données auront le même échantillon pondéré w :

$$w = 1/N \in 0, 1$$

où N est le nombre total de points de données.

La somme des échantillons pondérés est toujours égale à 1, de sorte que la valeur de chaque poids individuel se situe toujours entre 0 et 1. Ensuite, nous calculons l'influence réelle de ce classificateur sur la classification des points de données à l'aide de la formule :

$$\alpha^t = 1/2ln(1 - totalError)/totalError$$

Alpha est le degré d'influence que ce noeud aura dans la classification finale. L'erreur totale (totalError) n'est rien d'autre que le nombre total de classifications erronées pour cet ensemble d'apprentissage divisé par la taille de l'ensemble des données.

Remarquons d'après l'équation précédente que lorsqu'un noeud de décision fonctionne bien, ou n'a pas d'erreurs de classification, il en résulte un taux d'erreur de 0 et une valeur alpha positive relativement importante.

Si le noeud se contente de classer la moitié correctement et l'autre moitié incorrectement (un taux d'erreur de 0,5, ce qui n'est pas mieux qu'une estimation aléatoire !), la valeur alpha sera de 0. Enfin, si le noeud ne cesse de donner des résultats mal classés, la valeur alpha sera alors négative et importante.

Après avoir récupérer les valeurs réelles de l'erreur totale pour chaque noeud, il est temps de mettre à jour les poids des échantillons que nous avions initialement pris (1/N) pour chaque point de données. La formule est la suivante :

$$w_i = w_{i-1} * e^{\alpha}$$

En d'autres termes, le nouveau poids de l'échantillon sera égal à l'ancien poids de l'échantillon multiplié par le nombre d'Euler, porté à plus ou moins alpha (que nous venons de calculer à l'étape précédente).

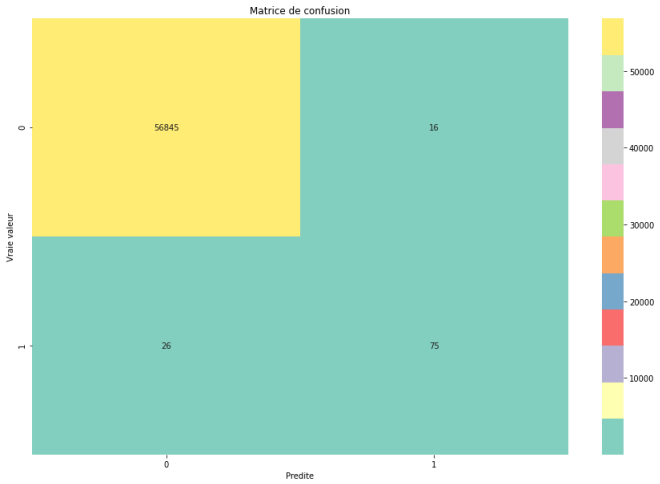
Les deux cas pour l'alpha (positif ou négatif) indiquent :

- Alpha est positif lorsque la sortie prévue et la sortie réelle concordent (l'échantillon a été classé correctement). Dans ce cas, nous diminuons le poids de l'échantillon par rapport à ce qu'il était auparavant, puisque nous obtenons déjà de bons résultats.
- Alpha est négatif lorsque la sortie prévue ne correspond pas à la classe réelle (c'est-à-dire que l'échantillon est mal classé). Dans ce cas, nous devons augmenter le poids de l'échantillon afin que la même erreur de classification ne se répète pas dans les noeuds suivante. C'est ainsi que les noeuds sont dépendantes de leurs prédécesseurs.

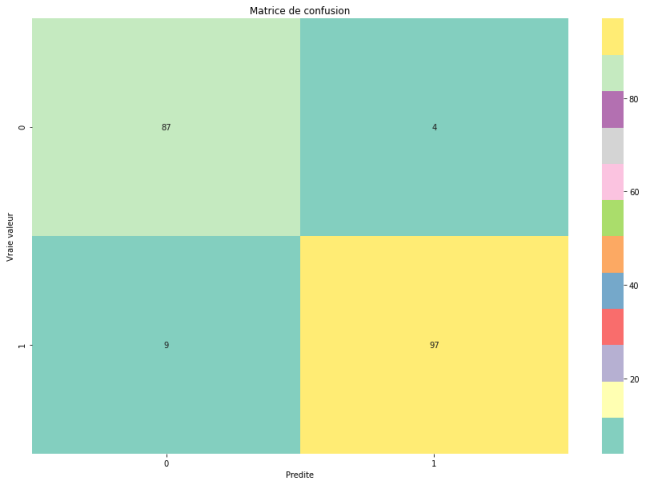
Pour appliquer ce modèle sur nos données, nous utilisons la méthode *sklearn.ensemble.AdaBoostClassifier()*.
Pour initialiser le modèle, l'entraîner et le tester, le script est le suivant :

```
clf = AdaBoostClassifier()
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

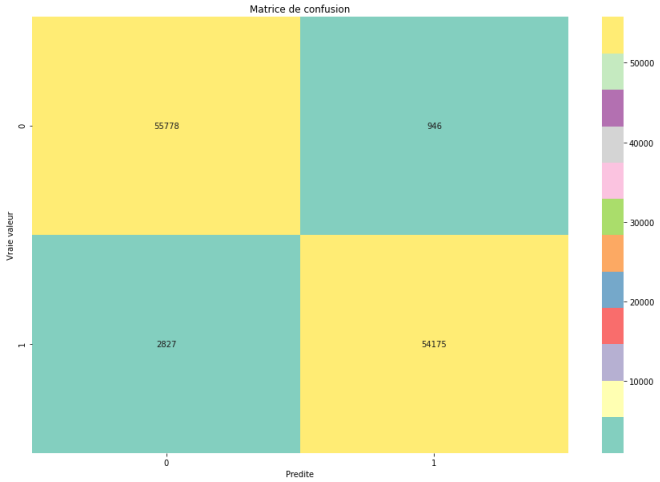
L'application du modèle sur les données de transactions bancaires d'origine obtient une *accuracy* de **0.9991** et la matrice de confusion suivante :



Sur les données de transactions bancaires sous-échantillonnées, l'*accuracy* est **0.9315** et la matrice de confusion est :



Sur les données de transactions bancaires sur-échantillonnées, l'*accuracy* est **0.9679** et la matrice de confusion est :



4.2.2 Gradient Boosting Classifier

Le *Boosting* est un type spécial de technique d'apprentissage d'ensemble qui fonctionne en combinant plusieurs apprenants faibles (prédicteurs avec une faible précision) en un apprenant fort (un modèle avec une forte précision). Chaque modèle fonctionne en prêtant attention aux erreurs de son prédécesseur. Dans le cas du classificateur *Gradient Boosting* [6], chaque prédicteur essaie de s'améliorer par rapport à son prédécesseur en réduisant les erreurs. Mais l'idée fascinante derrière le *Gradient Boosting* est qu'au lieu d'ajuster un prédicteur sur les données à chaque itération, il ajuste en fait un nouveau prédicteur aux erreurs résiduelles faites par le prédicteur précédent.
Nous détaillons ici le fonctionnement de la classification *Gradient Boosting*.
Afin de faire des prédictions initiales sur les données, l'algorithme obtient le *log of the odds* de la caractéristique cible. Il s'agit généralement du nombre de valeurs vraies (valeurs égales à 1) divisé par le nombre de valeurs fausses (valeurs égales à 0).
Une fois qu'il a le *log(odds)*, il convertit cette valeur en une probabilité en utilisant une fonction logistique afin de faire des prédictions. La formule de conversion du *log(odds)* en une probabilité est la suivante :

$$\frac{e * log(odds)}{(1 + e * log(odds))}$$

Pour chaque instance de l'ensemble de formation, il calcule les résidus pour cette instance, ou, en d'autres termes, la valeur observée moins la valeur prédite.
Une fois cette opération effectuée, il construit un nouvel arbre de décision qui tente réellement de prédire les résidus qui ont été calculés précédemment.
Cependant, c'est là que le processus devient légèrement délicat : lors de la construction d'un arbre de décision, un certain nombre de feuilles est autorisé (ce nombre peut être défini comme un paramètre par un utilisateur, et il est

généralement compris entre 8 et 32.). Cela conduit à deux des résultats possibles :

- Plusieurs instances tombent dans la même feuille
- Une seule instance a sa propre feuille

Contrairement au *Gradient Boosting* pour la régression, où nous pourrions simplement faire la moyenne des valeurs des instances pour obtenir une valeur de sortie, et laisser l'instance unique comme une feuille à part, il est nécessaire de transformer ces valeurs à l'aide d'une formule :

$$\frac{\sum Residual}{\sum[PreviousProb * (1 - PreviousProb)]}$$

Cette transformation est appliquée pour chaque feuille de l'arbre car l'estimateur de base est un *log(odds)* et l'arbre a été construit sur une probabilité, il n'est donc pas possible de simplement les additionner parce qu'ils proviennent de deux sources différentes.
A présent, pour faire de nouvelles prédictions, le modèle fait deux choses:

- obtention de la prédiction du *log(odds)* pour chaque instance de l'ensemble de formation
- conversion de cette prédiction en une probabilité

Pour chaque cas de l'ensemble de formation, la formule pour faire des prédictions est la suivante :

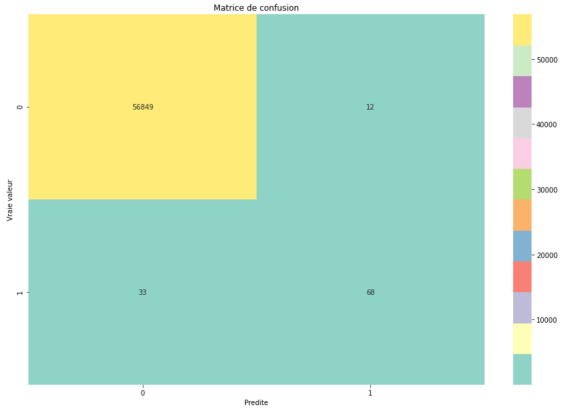
$$base_{l}og_{o}dds + (tauxd'apprentissage * valeurrésiduelleprédite)$$

Le taux d'apprentissage est un hyper-paramètre qui est utilisé pour mettre à l'échelle la contribution de chaque arbre, en sacrifiant les biais pour une meilleure variance. En d'autres termes, ce nombre est multiplié par la valeur prédite afin de ne pas surcharger les données.
Une fois la prévision *log(odds)* calculée, elle doit être convertie en une probabilité en utilisant la formule précédente.
Après avoir effectué ce processus, les nouveaux résidus de l'arbre sont calculés et un nouvel arbre qui correspond aux nouveaux résidus est crée.
Une fois de plus, le processus est répété jusqu'à ce qu'un certain seuil prédéfini soit atteint, ou que les résidus soient négligeables.

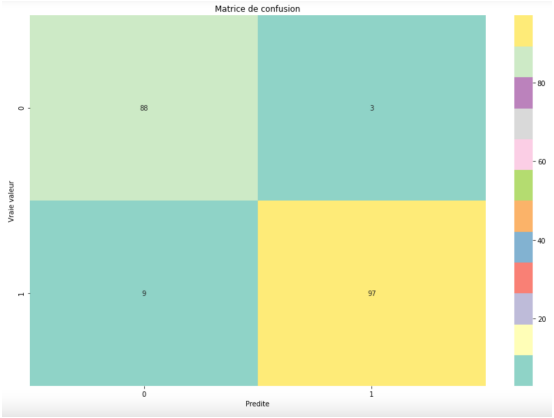
Pour appliquer ce modèle, nous utilisons la librairie *Sklearn*, plus précisément la méthode *Sklearn.ensemble.GradientBoostingClassifier()*.
Le script pour initialiser le modèle, l'entraîner sur les données *train*, le tester sur les données *test* et récupérer les prédictions et le score est :

```
clf = GradientBoostingClassifier()
clf.fit(X_train, y_train.values.ravel())
predicted = clf.predict(X_test)
score = accuracy_score(y_test, predicted)
```

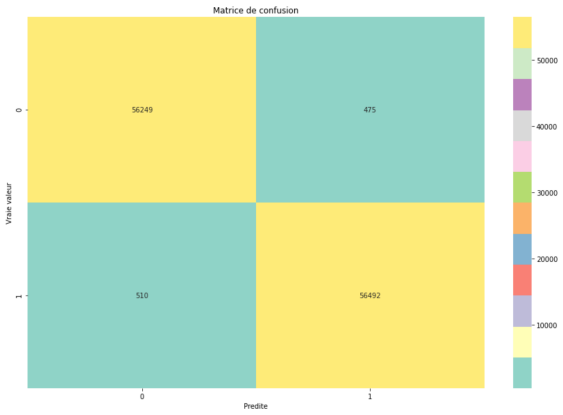
L'application du modèle *Gradient Boosting Classifier* sur les données d'origines obtient une *accuracy* de **0.9992** et la matrice de confusion suivante :



Sur les données sous-échantillonnées, l'*accuracy* est **0.9421** et la matrice de confusion :



Sur les données sur-échantillonnées, l'*accuracy* est **0.9938** et la matrice de confusion :



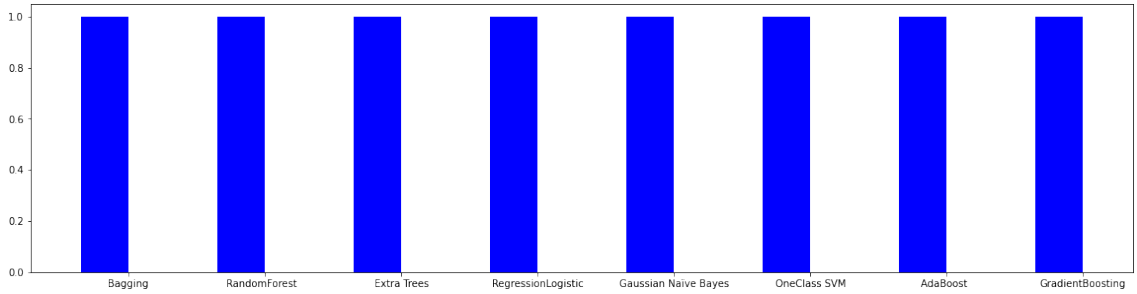
5 Comparaison

Si nous récapitulons, nous avons testés huit modèles :

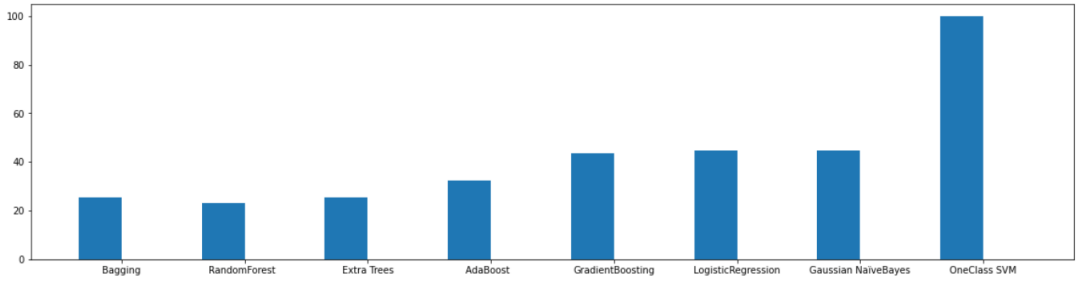
- Bagging Classifier
- Random Forest Classifier
- Extra Trees Classifier
- Logistic Regression
- Gaussian Naïve Bayes
- One Class SVM
- Ada Boost Classifier
- Gradient Boosting Classifier

Chaque modèle a été appliqué aux données d'origine, qui sont déséquilibrées, puis sur les données sous-échantillonnées et enfin sur les données sur-échantillonnées. Nous avons récolté pour chaque application de modèle, l'*accuracy* c'est à dire le taux de données bien classées par le modèle dans le dataset de test. Comme cette mesure n'est pas révélatrice dans le cas des données déséquilibrées, nous avons aussi affiché les matrice de confusion associées aux tests des modèles. En effet, la mesure qui nous intéresse est celle du nombre de cas frauduleux classés non frauduleux par le modèle. Si on rappelle qu'un cas frauduleux est associé au label 1, nous cherchons à minimiser le nombre de faux négatifs.

A présent, nous allons comparer les modèles pour chaque ensemble de données. Lorsque l'on applique les modèles sur les données d'origine, les mesures d'*accuracy* sont très élevées et ne permettent pas de conclure sur la différence de performance des modèles.

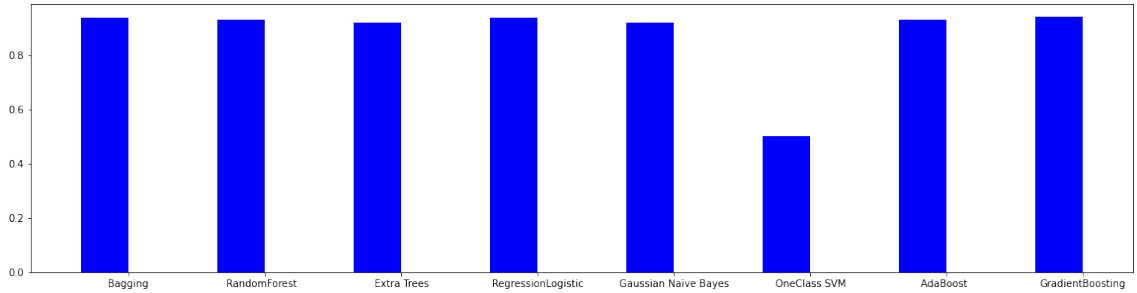


Observons le ratio faux négatifs vs faux positifs :

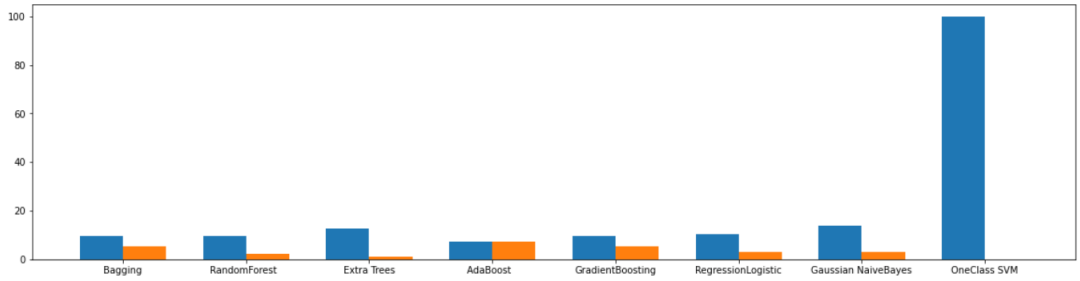


On observe que les modèles ont un taux de prédiction de faux négatifs beaucoup plus important que celui des faux positifs. Les modèles font ce qu'on appelle du sur-apprentissage. Parmi les différents modèles, *One Class SVM* est le moins performant, avec un taux de faux négatifs très important, dépassant 90%.
On observe aussi que les modèles *Random Forest* et *Extra Trees* paraissent être les plus performants avec un taux de faux négatifs moins élevé. Cette hypothèse reste à confirmer avec l'application des modèles sur les données sous et sur échantillonnées.

Les mesures d'*accuracy* des applications des modèles sur les données sous-échantillonnées sont très élevées. Bien qu'elles le soient moins que sur les données d'origine, elles ne sont toujours pas interprétables.

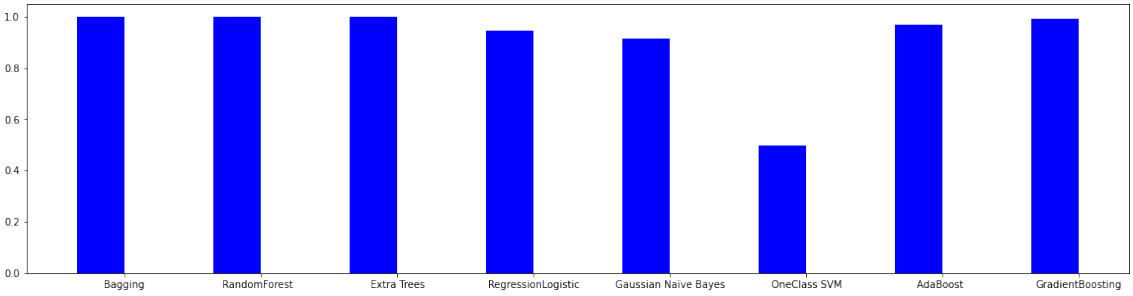


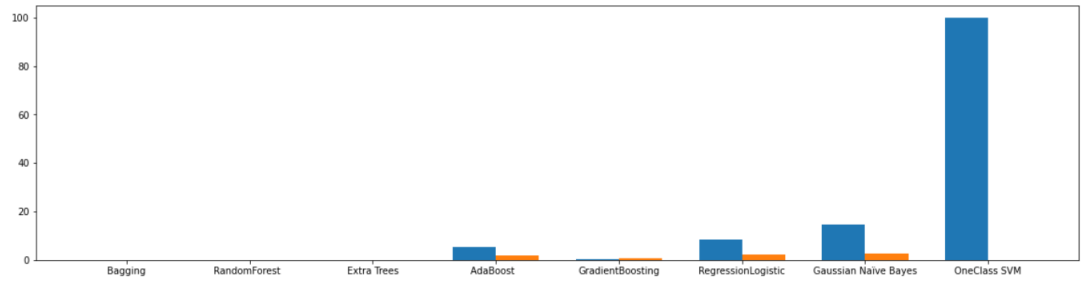
On observe cependant que le modèle *One Class SVM* a une *accuracy* bien inférieure à celle des autres Lorsque l'on se penche sur le ratio faux positifs vs faux négatifs, on observe que les taux de faux négatifs sont bien moins important que sur les données d'origine :



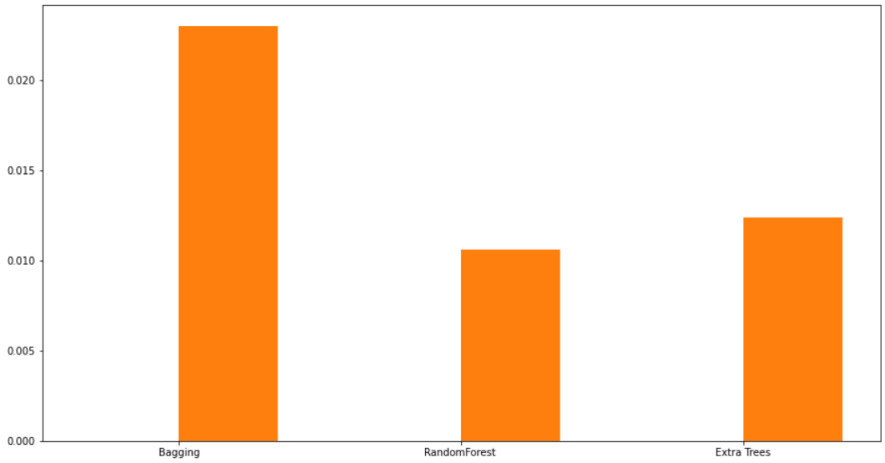
On observe que le modèle *One Class SVM* a un taux très important de faux négatifs contrairement aux autres modèles. De plus, les modèles *Random Forest* et *Extra Trees*, qui paraissaient être les plus performants sur les données d'origine, sont visiblement les moins performants parmi les modèles ayant un taux de faux-négatifs raisonnable.
On observe aussi que le taux de faux positifs, bien que moins important, est maintenant comparable à celui des faux négatifs. La technique de sous-échantillonage est donc performante, voyons le cas du sur-échantillonage.

Encore une fois, hormis pour le modèle *One Classe SVM*, les *acccuracy* des modèles sont très élevées et difficilement interprétables.





L'histogramme de comparaisons des taux de faux négatifs et faux positifs par modèle montre que les modèles *One Class SVM*, *Gaussian Naïve Bayes*, *Regression Logistic*, *Gradient Boosting* et *AdaBoost* ne sont pas performants. Il est préférable de les enlever. Voici ce que nous observons sans eux:



On conclut donc que les modèles ne sont pas du tout performants sur les données d'origine bien que *Random Forest* et *Extra Trees* se distinguent légèrement, les taux de faux négatifs ne sont jamais inférieurs à 20%, ce qui est trop élevé pour pouvoir valider un modèle.
Pour les modèles appliqués sur les données sous-échantillonnées, si l'on ne considère pas le modèle *One Class SVM*, les taux de faux négatifs sont toujours inférieurs à 10% ce qui est une amélioration.
Enfin, sur les données sur-échantillonnées, les modèles *Random Forest*, *Bagging Classifier* et *Extra Trees* ont un taux de faux négatifs et de faux positifs inférieurs à 1%, ce qui est très encourageant.

Nous avons à la fois comparé des méthodes d'échantillonage et des modèles. Pour les méthodes d'échantillonage, celle qui donne de meilleurs résultats est le sur-échantillonage. Cela paraît logique car avec cette technique, on rajoute des données d'apprentissage, donc de l'information au modèle.

Concernant les modèles, ceux les plus performants sont *Random Forest* et *Extra Trees*. Le moins performant étant *One Class SVM*.
On suppose que la non performance du modèle *One Class SVM* est due au fait qu'il n'a que 5 estimateurs (par contrainte de temps) contrairement à 400 pour les autres.

6 Conclusion

Le déséquilibre des données n'est pas toujours une mauvaise chose et dans les ensembles de données courantes, il y a toujours un certain degré de déséquilibre. Cela dit, il ne devrait pas y avoir d'impact important sur les performances de votre modèle si le niveau de déséquilibre est relativement faible.

Cependant, dans certains domaines tels que la détection des fraudes, le diagnostic médical et la gestion des risques, la distribution des classes est souvent très déséquilibrée et constitue donc un problème préoccupant. Par exemple, dans notre projet, nous utilisons la data science pour aider les banques et les institutions financières à réduire les taux d'opérations frauduleuses, et nous sommes confrontés à des ensembles de données fortement déséquilibrés où la classe minoritaire représente moins 1% de l'ensemble des données.

Il y a trois problèmes principaux auxquels nous sommes confrontés :

1. Le problème technique : les algorithmes d'apprentissage machine (ML) sont conçus pour minimiser les erreurs.
   Comme la probabilité que des observations appartiennent à la classe majoritaire est significativement élevée dans un ensemble de données déséquilibrées, les algorithmes sont beaucoup plus susceptibles de classer les nouvelles observations dans la classe majoritaire. Dans notre exemple, avec un taux d'opérations frauduleuses inférieur 1%, l'algorithme est incité à classer les nouvelles opérations dans la classe non frauduleuse, car il serait correct dans 99% des cas.

2. Le problème intrinsèque : dans la vie réelle, le coût d'un faux négatif est généralement beaucoup plus élevé que celui d'un faux positif, mais les algorithmes de ML pénalisent les deux à un poids similaire.
   Prenons l'exemple d'une banque : si votre modèle prévoit qu'une opération est frauduleuse mais que ce n'est pas le cas, la perte maximale à laquelle vous êtes soumis est le temps de vérification manuelle. En revanche, si votre modèle classe une opération bancaire par défaut comme étant sûr, le coût est alors sensiblement plus élevé car c'est le montant de l'opération qui sera perdu.

3. Le problème humain : la population et l'importante variété d'opérations bancaires ne permet pas la vérification de toutes les opérations et leur

classement manuel. Il faut rechercher à réduire le nombre de faux positifs afin de réduire le temps de vérification manuelle.

Deux approches différentes ont été adoptées dans notre projet pour remédier au déséquilibre des données : l'utilisation de différents algorithmes mais aussi des traitements sur les données.

Premièrement l'utilisation des algorithmes d'ensemble qui combine les prédictions de différents classificateurs. Chaque classificateur individuel est formé avec un sous-ensemble aléatoire. Les méthodes de ré-échantillonnage utilisées sont principalement au nombre de deux, le *bagging* et le *boosting*.

- La méthode de *bagging* entraîne tous les classificateurs avec différents *bootstraps* de l'ensemble de données. Un *bootstrap* est un sous-ensemble aléatoire de N échantillons qui sont remplacés plusieurs fois. Une fois que les modèles sont formés, le résultat final est le vote majoritaire des classificateurs.
- Le *bootstrap* utilise la classe la plus difficile à prévoir, pour entraîner les modèles de classification. Le premier classificateur est formé avec un échantillon aléatoire de l'ensemble de données. La classe qui a le plus d'observations mal classifiées sera la classe majoritaire dans l'échantillon suivant. Et ainsi de suite.

La méthode du *boosting*, en général, atteint de meilleures performances que l'échantillonnage *bootstrap*.

Deuxièmement nous avons procédé à des traitements sur les données. Cela consiste à ré-échantillonner les données afin d'atténuer l'effet causé par le déséquilibre des classes. Les deux techniques les plus courantes sont le sur-échantillonnage et le sous-échantillonnage.

- Dans l'ensemble d'apprentissage, l'avantage du sur-échantillonnage est qu'aucune information de l'ensemble d'apprentissage initiale n'est perdue, car toutes les observations des classes minoritaires et majoritaires sont conservées.
- Le sous-échantillonnage, vise à réduire le nombre d'échantillons majoritaires pour équilibrer la répartition des classes. Comme il supprime des observations de l'ensemble de données initial, il risque d'écarter des informations utiles.

Nous avons exposé dans la section précédente les résultats obtenus ainsi que nos interprétations sur les modèles testés. Finalement nous avons aussi testé d'autres modèles adaptés aux données mal distribuées issues de la librairie *imblearn* comme *BalancedRandomForestClassifier()*, *BalancedBaggingClassifier()*,

RUSBoostClassifier() et *EasyEnsembleClassifier()*.

L'échantillonnage et la classification effectué par ces algorithmes a été très performant mais le fonctionnement approfondi de ces algorithmes n'a pas été décris dans le projet qui s'est concentré sur les modèles ensemblistes traditionnels et sert ici comme une référence.
D'autres approches peuvent être étudiées comme la sélection des caractéristiques, car la haute dimensionnalité d'un ensemble de données peut avoir un effet négatif sur les performances du modèle. Il peut être intéressant de sélectionner les caractéristiques les plus précieuses et d'exclure celles qui apportent le moins d'informations ou alors d'appliquer des modèles de réduction de dimension pour palier ce problème.

References

[1] Sukarna Barua et al. "MWMOTE–majority weighted minority oversampling technique for imbalanced data set learning". In: *IEEE Transactions on Knowledge and Data Engineering* 26.2 (2012), pp. 405–425.

[2] Eric Bauer and Ron Kohavi. "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants". In: *Machine learning* 36.1-2 (1999), pp. 105–139.

[3] Leo Breiman. "Bagging predictors". In: *Machine learning* 24.2 (1996), pp. 123–140.

[4] Leo Breiman. "Pasting small votes for classification in large databases and on-line". In: *Machine learning* 36.1-2 (1999), pp. 85–103.

[5] Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139.

[6] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[7] Mikel Galar et al. "A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.4 (2011), pp. 463–484.

[8] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine learning* 63.1 (2006), pp. 3–42.

[9] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282.

[10] Tin Kam Ho. "The random subspace method for constructing decision forests". In: *IEEE transactions on pattern analysis and machine intelligence* 20.8 (1998), pp. 832–844.

[11] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. "Exploratory undersampling for class-imbalance learning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39.2 (2008), pp. 539–550.

[12] D. Opitz and R. Maclin. "Popular Ensemble Methods: An Empirical Study". In: *Journal of Artificial Intelligence Research* 11 (Aug. 1999), pp. 169–198. ISSN: 1076-9757. DOI: 10.1613/jair.614. URL: http://dx.doi.org/10.1613/jair.614.

[13] Bernhard Schölkopf et al. "Support Vector Method for Novelty Detection". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 2000, pp. 582–588. URL: https://proceedings.neurips.cc/paper/1999/file/8725fb777f25776ffa9076e44fcfd776-Paper.pdf.

[14] Harry Zhang. "The Optimality of Naive Bayes". In: *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*. May 17-19, 2004 (Miami Beach, Florida, USA). Ed. by Valerie Barr and Zdravko Markov. AAAI Press, 2004.