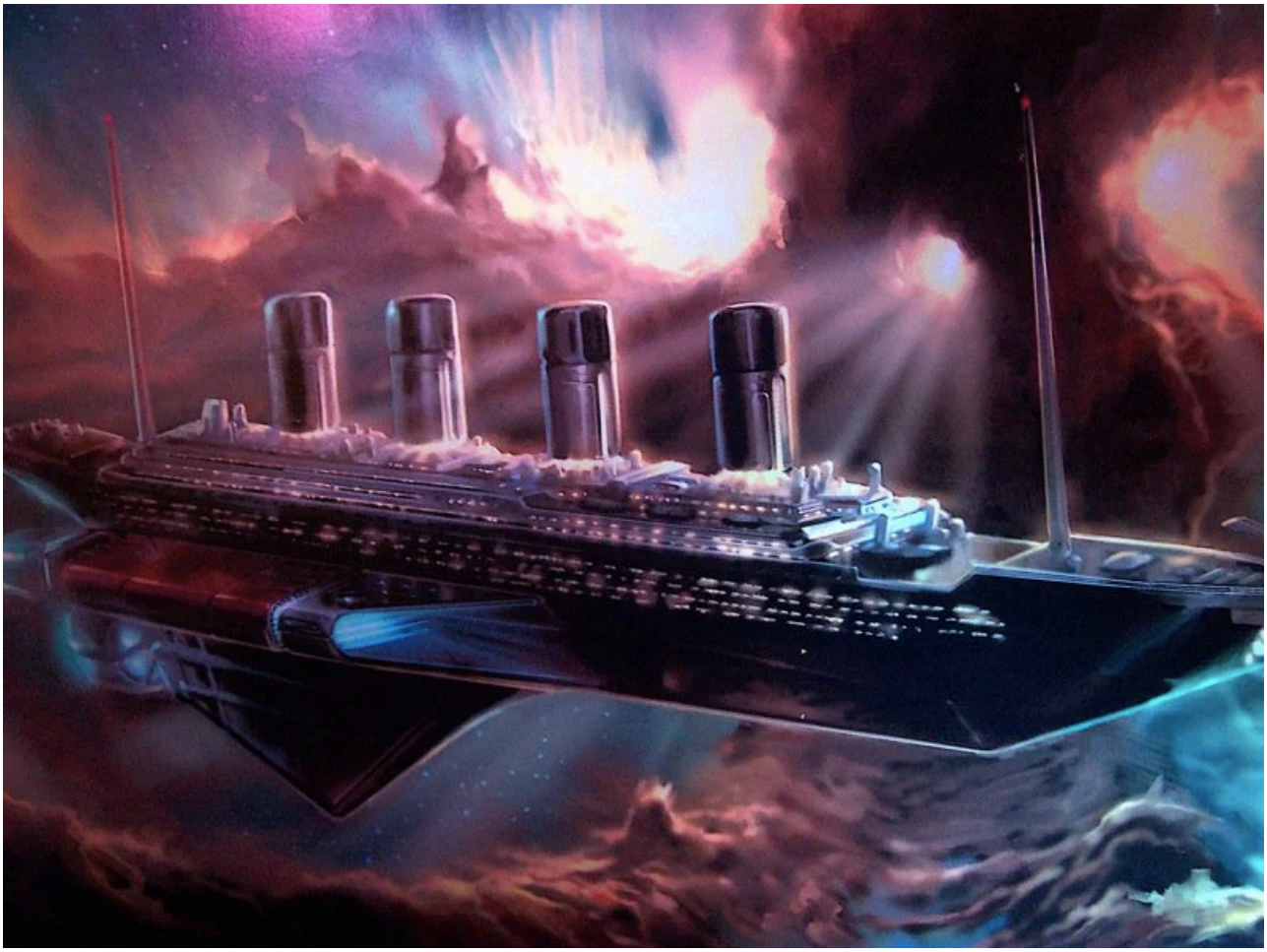
Spaceship Titanic.

Rapport - Projet n°8 : « Participez à une compétition Kaggle ! »

Luke Duthoit - Juillet 2022



(Illustration d'un épisode de « Doctor Who », trouvée sur Reddit)

Table des matières

0 - Introduction

Ce rapport récapitule l'ensemble du travail accompli durant deux semaines et demi, dans le cadre du projet n°8 du parcours étudiant « Ingénieur *Machine Learning* » proposé par le site de formation en ligne OpenClassrooms.

Si son intitulé « Participez à une compétition Kaggle ! » est assez explicite quant à la nature du travail demandé, il ne laisse en revanche pas forcément présager que c'était à l'étudiant de choisir librement la compétition en question. Mon choix s'est donc porté sur la compétition « *Spaceship* Titanic » (*https://www.kaggle.com/competitions/spaceship-titanic*) pour plusieurs raisons. La première d'entre elle était que la simplicité apparente de cette compétition rendait crédible la perspective de réaliser ce projet en relativement peu de temps, ce qui était un critère essentiel à mes yeux, ne disposant plus que de 4 semaines pour accomplir les projets n°7 et 8. La seconde était que ce problème de classification et sa thématique sympathique de science-fiction permettaient également de « souffler » après trois mois intenses à travailler sur les projets n°5 et 6 très complexes. Ce ne sont pas forcément les raisons les plus honorables, mais ce choix m'a effectivement permis de travailler relativement vite et bien, et donc de tenir mon planning de travail.

Lors des pages qui vont suivre, nous allons prendre le temps de rappeler brièvement le cadre de cette compétition, puis nous détaillerons les étapes successives selon lesquelles se projet a été construit. En particulier, nous traiterons de l'analyse exploratoire des données, des choix des opérations de nettoyage et de *features engineering*, ainsi que les essais et optimisations de différents modèles. L'ensemble des lignes de code associées à ces descriptions sont accessibles sous forme de *notebook* Jupyter :

- sur mon espace personnel Kaggle (*https://www.kaggle.com/code/lukeduthoit/eda-featureengineering-noscaling-mlpclassifier*),
- sur mon *drive* Google (*https://drive.google.com/drive/folders/13fdKGQRnlFh358dauKTs3vjOpY2nd-1P*),
- ainsi que sur un répertoire en ligne GitHub (*https://github.com/LukDuth/OC_Student_ML_P8_Titanic_Spaceship*).

Ces deux derniers présentent l'avantage (le 1er grâce au répertoire ***/Notebooks/Archives/***, le 2nd via ses différentes branches explicitement nommées), de pouvoir visualiser l'ensemble des étapes successives de rédaction des *notebooks*, la version finale soumise à la compétition étant relativement épurée de plusieurs phases d'essais afin d'en réduire le temps d'exécution. Pour y avoir accès, n'hésitez pas à m'écrire à *luke.duthoit@gmail.com*.

I - Préambule au code

I.A - Description et objectifs de la compétitions

La compétition « *Spaceship* Titanic » est largement inspirée d'une autre compétition basée sur le célèbre jeu de données des passagers du Titanic. Tel son illustre prédécesseur, le vaisseau spatial futuriste Titanic a percuté un obstacle lors d'un voyage interstellaire, cet obstacle étant cette fois-ci une « anomalie spatio-temporelle », ayant conduit à la transportation d'une partie des passagers vers une autre dimension !

Sur la base des données concernant les passager, il nous faut donc prédire pour chaque passager dont l'information est manquante si il a été transporté - ou non - dans la dimension alternative. Le jeu de données est fourni d'emblée divisé entre jeu d'entraînement et jeu de test, sus formes de deux matrice. Chaque ligne représente un passager, et chaque colonne un paramètre, parmi lesquels une dizaine de caractéristiques potentielle ainsi qu'une étiquette (cette étiquette étant constante - donc inconnue - pour le jeu de test). Nous avons donc affaire à un problème de classification binaire.

L'objectif officiel est de trouver un algorithme maximisant un score de précision, et de s'en servir pour renseigner les étiquettes du jeu de test. Bien que cela ne soit pas explicitement écrit à ma connaissance, le score de précision doit être calculé à partir du jeu d'entraînement car nous n'avons pas accès aux vraies étiquettes du jeu de test (cette compétition n'ayant pas de limite temporelle). Afin de participer honnêtement, il convient donc de calculer un score moyen à l'issue d'une validation croisée sur le jeu d'entraînement. Un fichier contenant la prédiction pour chacun des passager du jeu de test est également à soumettre pour acter sa participation. La hauteur du score de précision détermine le classement dans la compétition.

I.B - Description sommaire du jeu de données

Le jeu de donnée est fourni sous la forme de deux fichiers :

- **train.csv** : C'est le jeu d'entraînement, il contient les informations personnelles d'environ deux tiers des passagers (~8700 lignes).
- **test.csv** : C'est le jeu test, il contient les informations personnelles du tiers approximatif restant des passagers (~4300 lignes).

Les paramètres (colonnes) de ces deux jeux sont presque les mêmes :

- **PassengerId [string]** : Un identifiant unique pour chaque passager. Chaque identifiant s'écrit sous la forme *gggg_pp* où *gggg* représente le groupe de passagers au sein duquel le passager courant appartient, et *pp* est son numéro au sein de ce groupe. Des passagers du même groupe sont souvent des membres d'une même famille, mais pas nécessairement.

- **HomePlanet [string]** : La planète où le passager a embarqué, souvent sa planète de résidence.
- **CryoSleep [bool]** : Booléen indiquant si le passager a été mis en état de sommeil cryogénique le temps du voyage. Si c'est le cas, le passager est confiné dans sa cabine.
- **Cabin [string]** : Le numéro de cabine où réside le passager, qui s'écrit sous la forme suivante *d*/*n*/*s*, où *d* représente le pont, *n* le numéro proprement dit, et *s* le côté du pont [bâbord ou tribord] où se situe la cabine.
- **Destination [string]** : La planète où le passager souhaite être débarqué.
- **Age [int]** : L'âge du passager.
- **VIP [bool]** : Booléen indiquant si le passager a payé pour avoir accès aux services spéciaux pour *VIP* durant le voyage.
- **RoomService**, **FoodCourt**, **ShoppingMall**, **Spa**, **VRDeck [float]** : Cinq montants correspondants à des dépenses « de luxes » potentielles faites par le passager.
- **Name [string]** : Prénom et nom de famille du passager.
- **Transported [bool]** : Booléen indiquant si le passager a été - ou non - transporté dans la dimension parallèle. IL S'AGIT DE NOTRE ETIQUETTE. A la différence des autres colonnes, elle n'est présente qu'au sein du jeu d'entraînement.

Enfin, le fichier **sample_submission.csv** est également fourni, et à soumettre complété à l'issue de notre travail et de la prédiction du paramètre *Transported* sur le jeu de test. Il est constitué de deux colonnes : *PassengerId* et *Transported*. La première suit les valeurs prise par son homologue au sein du jeu de test, quand la seconde est initialisée systématiquement à *False*.

I.C - Chargement et organisation des données

Les trois fichiers précédents ont donc été téléchargés depuis la page web de la compétitions sur mon disque dur local afin de pouvoir développer mon code sous différentes versions, et sans avoir besoin de dépendre formellement d'une connexion internet pour cela. En effet, il était possible d'écrire un *notebook* en ligne directement depuis mon espace personnel Kaggle, mais j'ai choisi de n'en rien faire, seule la version finale ayant été chargée et exécutée en ligne afin que la performance de mon code puisse être officiellement évaluée.

I.D - Gestion des versions du code

Comme précisé plus haut, chaque version du code a été écrite et programmée en locale sur mon ordinateur. La gestion des versions du code a été faite de deux manières en parallèle :

- Soit via un répertoire en ligne GitHub, grâce au système de branches, chaque branches ayant servi lors d'un nombre précis d'étapes/d'essais.
- Soit en créant un nouveau notebook en local et en archivant les anciens à mesure que l'on progressait dans les code en ne gardant que les choix concluant à chaque étapes.

Nous récapitulons dans le tableau ci-dessous la correspondances entre les différentes étapes/essais auxquels nous avons procédés, et les noms des branches GitHub/des versions archivées en local du notebook :

Etapes/phases d'essais contenues dans les différentes versions du notebook	Noms des branches du répertoire GitHub correspondantes	Noms de sauvegarde des versions archivées en local correspondantes
Chargement des données + EDA, *feature engineering* + une 1ère tentative de remplissage des *NaN*.	Chargée initialement dans la branche ***main***.	`Duthoit_Luke_1_Code_072022_01`
Nouvelles manières plus malignes de remplir les *NaN*.	Développée dans la branche ***filling_missing_values***, puis versée dans ***main*** (version précédent écrasée).	`Duthoit_Luke_1_Code_072022_02`
***Encoding* des *features* qualitatives + comparaisons de différents types de *scaling* au travers de l'optimisation/ validation croisée d'un *classifier* simple.**	Développée dans la branche ***encoding_and_testing_scalings***, puis versée dans ***main*** (version précédent écrasée).	`Duthoit_Luke_1_Code_072022_03`
A partir du meilleur scaling : recherche du meilleure modèle de ML en comparant différents classifiers (optimisation des hyper-paramètres et validation croisée)	Développée dans la branche ***testing_classifiers***, puis versée dans ***main*** (version précédent écrasée).	`Duthoit_Luke_1_Code_072022_04`
A partir du meilleur modèle : prédictions sur le jeu de test et soumission des résultats.	Développée dans la branche ***predictions_and_results_submission***, puis versée dans ***main*** (version précédent écrasée).	`Duthoit_Luke_1_Code_072022_05`

II - Analyse exploratoire des données (*EDA*)

II.A - Découverte du jeux de données

II.A.1) Importance des valeurs manquantes

On a commencé par mesurer l'importance des valeurs manquantes (*NaN*) au sein des colonnes des deux jeux de données, ce qui a conduit à la production de la figure ci-dessous [Fig1].

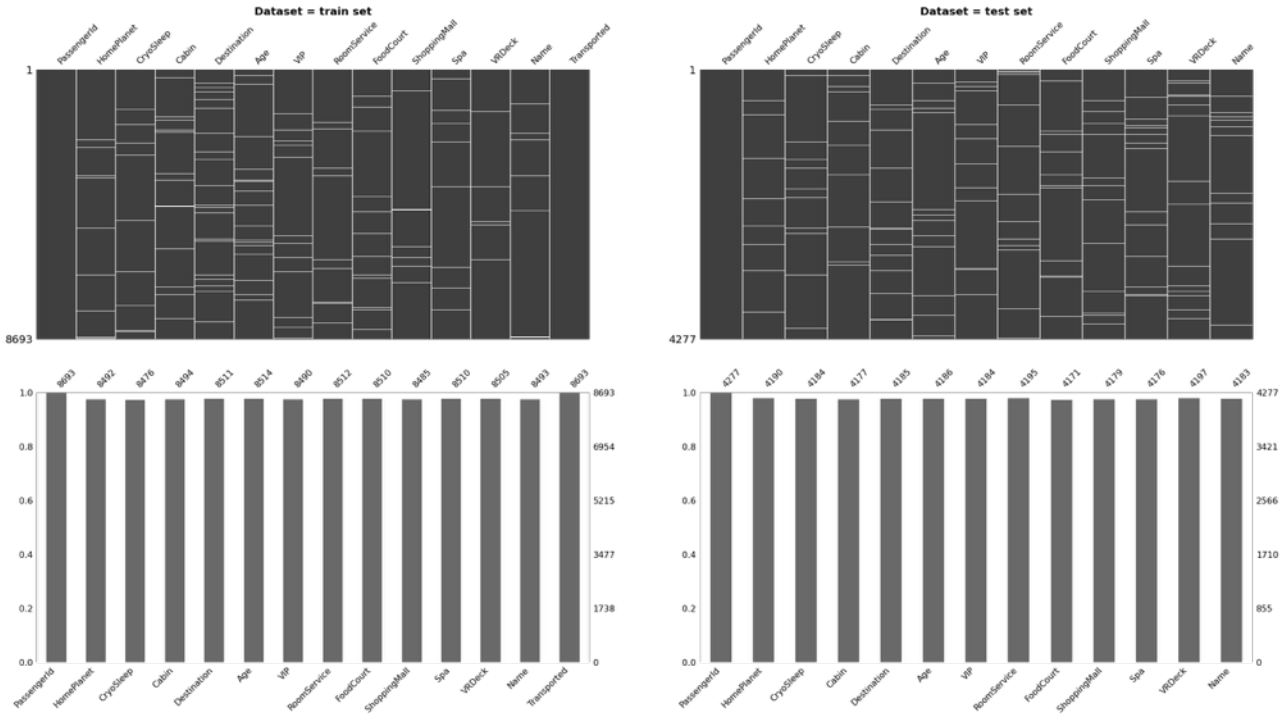


Fig1 - Positions des NaN au sein des colonnes (en haut), et taux de remplissages des colonnes (en bas).

On y voit que les *Nan*, au sein d'une même colonne, semblent rarement correspondre aux même lignes. Malgré de très bon taux de remplissage (chaque colonne ayant près de 97% de ses valeurs bien définies), cette désynchronisation fait que, d'après nos calculs, « seulement » 76-77% des lignes de chaque jeu est totalement remplie. Nous avons donc décidé en regard de ce chiffre relativement important de ne pas supprimer les lignes contenant des NaN, ce qui nous a obligé par la suite à developper des stratégies adaptées de remplissages des valeurs manquantes.

II.A.2) Concaténation des deux jeux

Les deux jeux de données, transformés en objets Python de type *pandas.Dataframe* lors de leur lecture par le notebook, ont été concaténés par nos soins en un seul et même *pandas.Dataframe*, afin de faciliter mon travail lors de la phase d'analyse exploratoire des données, où il était important de regarder les distributions des différents paramètres aussi bien selon leur appartenance à l'un de ces deux jeux qu'indépendamment à leur

appartenance. Afin de pouvoir facilement remonter à leur provenance, une colonne supplémentaire de booléens nommée **TrainOrTest** a été créée ; elle prend *True* si le passager vient du jeu d'entraînement, *False* si c'est du jeu de test. La colonne *Transformed*, initialement non présente dans le *pandas.Dataframe* correspondant au jeu de test, a donc pris des valeurs non définies *NaN* aux lignes correspondantes.

II.A.3) Recherche d'éventuelles lignes dupliquées

Stricto sensu, il n'y a pas de lignes dupliquées dans le jeu de données, du fait de l'unicité de chaque identifiant des passager, la colonne *PassengerId* ayant donc autant de valeurs uniques que de lignes.
Néanmoins, comme l'a fait remarqué l'internaute *@CaruzzoC* lors d'un discussion sur la cardinalité des/la corrélation entre *features*, (*https://www.kaggle.com/competitions/spaceship-titanic/discussion/309513*), il existe une centaine de voyageurs ayant exactement le même nom, ce qui nous a poussé à recherché d'éventuels passagers dupliqués « cachés » derrières des identifiants différents.
En réalité, comme montré dans le *notebook* (en section **I.B.3**, cellules **17-19**), il n'en est probablement rien, les passagers homonymes ne siégeant jamais dans les mêmes cabines, et voyageant parfois vers des planètes différentes. Leur nom en commun est probablement dû au hasard de la création du jeu de donnée.

II.A.4) *Feature engineering* (1) : création de nouvelles *features*

Nous avons décidé de procéder à une première vague de *feature engineering* en créant de nouvelles colonnes, la plupart du temps sur la foi de ce qu'on a pu lire au sein de plusieurs discussions relatives à cette compétition :

- Grâce à une discussion mentionnant explicitement l'intérêt de ne pas se débarrasser du paramètre *NumCabin* (*https://www.kaggle.com/competitions/spaceship-titanic/discussion/309693*), nous avons décidé de « découper » ce paramètre en trois selon les *slashs* qui séparent les différentes informations, créant ainsi trois nouveaux paramètres remplaçant l'ancien :
  - **Deck** [qui prend une valeur parmi {A, B, C, D, E, F, G, T}] ;
  - **Num** [le numéro de cabine converti explicitement en entier] ;
  - **Side** [qui vaut soit S soit P].
- Au sein de cette même discussion, l'internaute *@bcruise* a fait remarquer qu'il pouvait être utile de séparer *PassengerId* en deux selon *l'underscore*, le nombre à deux chiffres constituant la seconde partie étant important (car lié à la taille du groupe de passager). Nous avons donc suivi cet avis, et remplacé *PassengerId* par :
  - **PdI1** [le nombre à 4 chiffres à gauche de *l'underscore*, converti en entier] ;
  - **PdI2** [le nombre à 2 chiffres à droite de *l'underscore*, converti explicitement en entier].
- Grâce à une discussion sur les différentes techniques d'encodages des paramètres qualitatifs (*https://www.kaggle.com/competitions/spaceship-titanic/discussion/309803*), nous avons repris l'idée de sommer les cinq paramètres qui concernent les dépenses des passager en un seul, créant ainsi (tout en conservant les cinq paramètres originaux) le paramètre **TotalExpenses**.

II.A.5) Une première sélection de *features*

Afin de faire une première sélection au sein des colonnes pour savoir lesquelles pouvaient servir de caractéristiques en entrée de l'algorithme, nous nous sommes en partie basés sur la cardinalité de chaque colonne [le nombre de valeurs uniques prises], qui est affichée dans la capture d'écran ci-contre [Fig2].

Column	Number of unique values
PId1	9280
PId2	8
HomePlanet	3
CryoSleep	2
Deck	8
NumCabin	1894
Side	2
Destination	3
Age	80
VIP	2
RoomService	1578
FoodCourt	1953
ShoppingMall	1367
Spa	1679
VRDeck	1642
TotalExpenses	2980
FirstName	2883
LastName	2406
Transported	2
TrainOrTest	2

Fig2 - Cardinalité des colonnes.

En effet, un trop grand nombre de valeurs uniques par rapport au nombre de ligne, comme c'est le cas pour *PId1*, est caractéristique d'un paramètre dont le rôle est d'identifier les éléments du jeu de donné (ce qui est le cas car *PId1* provient de *PassengerId*). *PId1* ne peut donc servir de caractéristique. *A contrario*, *PId2*, pourtant issue de *PassengerId* elle aussi, possède très peu de valeurs uniques, et donc peut éventuellement servir de caractéristique pour l'algorithme. Pour ce qui est de *FirstName* et *LastName*, ils présentent certes une cardinalité bien moins grande que celle de *PId1*, mais ils sont issues de l'ancienne colonne *Name*, qui était quasiment constituée de valeurs uniques car servait également d'identifiant. Nous ne les avons donc pas considérées comme des *features* potentielles.
De plus, les colonnes prenant de valeurs numériques (*int*, *float*, etc) mais prenant peu de valeurs uniques différentes peuvent être considérées d'avantage comme des *features* qualitatives (ou catégorielles) que quantitatives (ou ordinales). C'est ce que nous avons considérés pour *PId2*, par exemple.
Enfin, TrainOrTest n'a évidemment pas vocation à être une caractéristique de l'algorithme.
Nous récapitulons donc les différentes listes de colonnes selon leur assignation (non features, features catégorielles, features ordinales) dans le tableau ci-dessous :

Colonnes écartées de la liste des *features* potentielles : (4)	**Colonnes considérées comme potentielles *features* catégorielles : (7)**	**Colonnes considérées comme potentielles *features* ordinales : (8)**
PId1, FirstName, LastName, TrainOrTest	PId2, HomePlanet, CryoSleep, Deck, Side, Destination, VIP	NumCabin, Age, RoomService, FoodCourt, ShoppingMail, Spa, VRDeck, TotalExpenses

Notons bien que cette sélection n'est qu'une première approche, et qu'elle ne constitue pas notre sélection finale de *features* pour l'algorithme.

Nous pouvons désormais passer à l'analyse exploratoire uni-variée de ces *features* potentielles.

II.B - EDA uni-variée

II.B.1) Potentielles *features* ordinales.

On représente ci-dessous [Fig3] les distributions individuelles de chacune de ces huit colonnes (voir tableau précédent), sous formes d'histogrammes uni-variés. Notons que le nombre d'intervalles est calculé automatiquement lors du tracé graphique (au moyen d'une fonction de a librairie *seaborn*), que les descriptions statistiques de chacune de ces distributions sont écrites en légende de chaque sous-figure, et que le nombre d'apparitions [*count*, sur l'axe des ordonnées] est en échelle logarithmique.

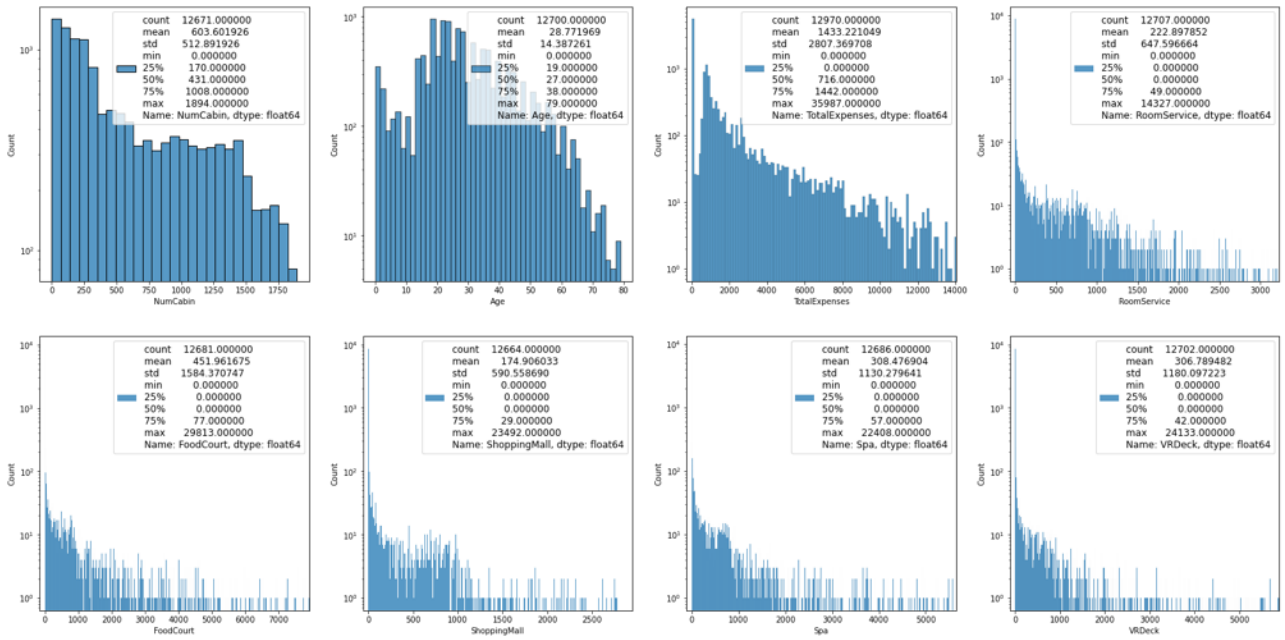


Fig3 - Distributions individuelles (uni-variée) des potentielles features *ordinales.*

Ces distributions présentent un mode toujours parmi les valeurs prises les plus basses (pour *NumCabin* et *Age*, dont le mode est moins facilement lisible que pour les autres colonnes, il prend respectivement la valeur de 82 et 18), et sa fréquence d'apparition écrase souvent le reste de la distribution. Ceci est particulièrement vrai pour les six paramètres qui concernent des dépenses luxueuses, pour lesquelles la valeur nulle représente plus de 50% des valeurs prises sur l'ensemble des passagers. Cela conduit d'ailleurs à ce que leur écart-type soit plusieurs fois supérieur à leur moyenne (seules *NumCabin* et *Age* y échappent). Notons que la création de *TotalExpenses* a permis d'atténuer un peu ce phénomène (sa médiane étant non nulle, et son écart n'étant « que » environ deux fois supérieur à sa moyenne), ce qui signifie que les cinq paramètres précédents sont pas systématiquement simultanément nuls.
Il fallait donc s'attendre à ce qu'il y ait d'éventuelles corrélations non linéaires entre ces potentielles *features* et notre *target Transported*. C'est en autre pour cela que nous avons fait le choix par la suite de calculer des coefficients de corrélation dite « **φ**K » (voir section II.C. 1, en page 13).
Passons désormais aux distributions uni-variées des sept *features* catégorielles potentielles.

II.B.2) Potentielles *features* catégorielles.

On représente ci-dessous [Fig4] les distributions individuelles des sept colonnes concernées, où le code couleur permet de distinguer la fréquence d'apparition de chacune des catégories constitutives de la colonne représentée. Notons qu'une brève description statistique de chacune de ces distributions est écrite en légende de chaque sous-figure.

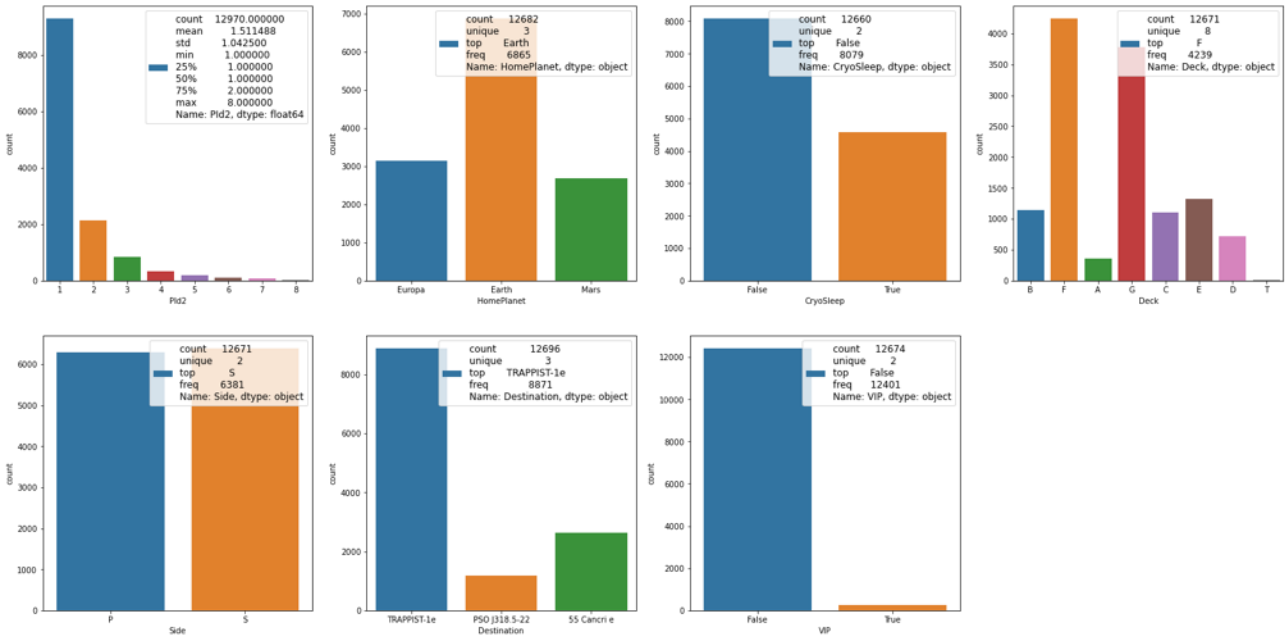


Fig4 - Distributions individuelles (uni-variée) des potentielles features *catégorielles.*

L'analyse de ces distributions est plus compliquée que précédemment, en raison de différences importantes entre paramètres, ainsi qu'à l'intérieur même des distributions individuelles (c'est à dire d'une catégorie à l'autre). Pour résumer ce que nous avons déjà écrit dans le notebook (section **I.C.3**, cellules **37-38**) :

- Les paramètres *Side*, *HomePlanet* et *CryoSleep* présentent des catégories assez équilibrées (en particulier *Side*), au sens où les catégories les moins représentées le sont néanmoins de manière significative. Ils pourraient constituer de bons *features* pour l'algorithme.
- Les paramètres *PId2*, *Deck* et *Destination* sont assez déséquilibrés, car une ou deux catégories représentent près ou plus de deux tiers des occurrences, alors qu'aucune autres des catégories significativement minoritaires ne représente plus de 15-20% des apparitions. Ces *features* potentielles pourraient nécessiter un peu de *feature engineering* pour faciliter la tâche de l'algorithme (par exemple, en fusionnant les catégories significativement minoritaires en une seule catégorie, afin d'avoir plus de poids statistique).
- Le paramètre VIP est quasi-inutile en l'état, car il est en réalité quasiment constamment égal à une seule de ces deux catégories.

II.B.3) Recherche d'écarts statistiques éventuels entre jeux de test et d'entraînement.

Enfin, toutes ces distributions ayant été tracées sans tenir compte de connaître l'appartenance des passagers à l'un des deux jeux de données, il nous faut vérifier si les données conserveront à peu près les mêmes propriétés entre la phase d'entraînement et la phase de test de l'algorithme
Pour les paramètres ordinaux, nous avons fait calculer quelques propriétés statistiques (médiane, écart-type) de leurs distributions individuelles en regroupant les passagers selon les valeurs prises par la colonne *TrainOrTest*. Le résultat est présenté dans la capture d'écran ci-dessous [Fig5].

	Age		NumCabin		TotalExpenses		RoomService		FoodCourt		ShoppingMall		Spa		VRDeck	
	std	50%	std	50%	std	50%	std	50%	std	50%	std	50%	std	50%	std	50%
TrainOrTest																
Test	14.179072	26.0	514.968131	442.0	2816.402100	714.0	607.011289	0.0	1527.663045	0.0	560.821123	0.0	1117.186015	0.0	1246.994742	0.0
Train	14.489021	27.0	511.867226	427.0	2803.045694	716.0	666.717663	0.0	1611.489240	0.0	604.696458	0.0	1136.705535	0.0	1145.717189	0.0

Fig5 - Propriétés statistiques des paramètres précédents selon l'appartenance au train/test set.

On y voit que ces deux propriétés statistiques varient très peu en terme d'écart-relatif d'un jeu de données à l'autre, pour chaque paramètre.
Pour les paramètres catégoriels, on a repris la représentation graphique des occurrences de chaque catégorie, mais en utilisant le code couleur cette fois-ci pour distinguer l'appartenance à l'un des deux jeux d'entraînement ou de test [ci-dessous, Fig6].

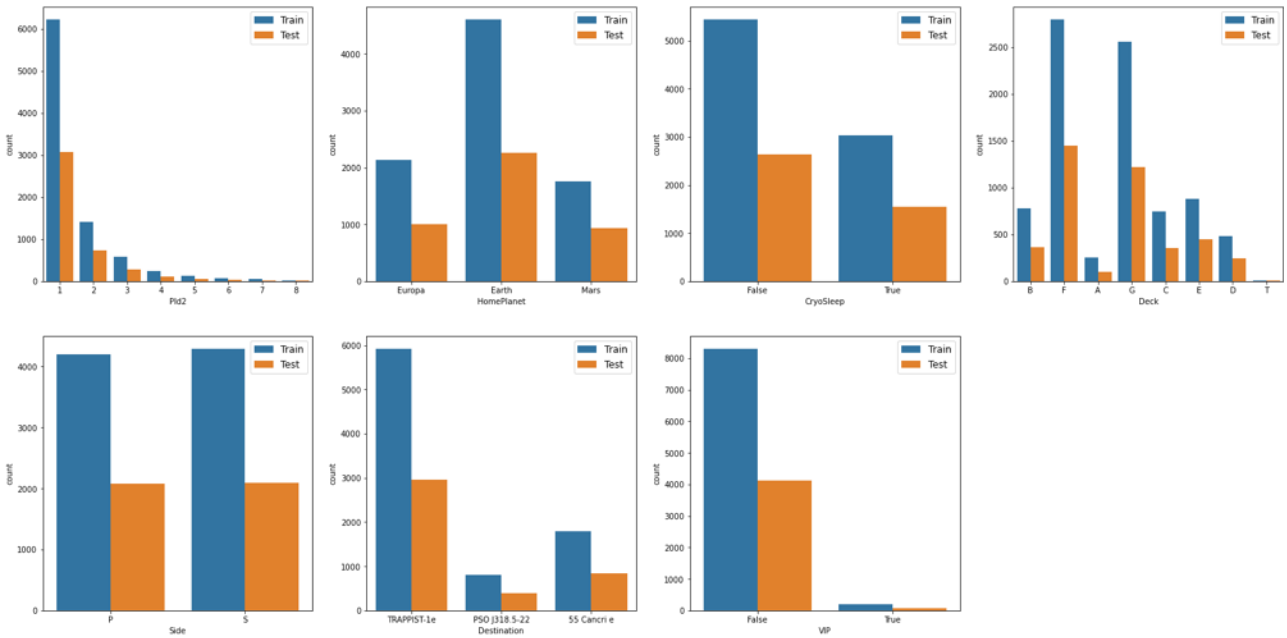


Fig6 - Distributions individuelles (uni-variée) des potentielles features *catégorielles.*

Les hiérarchies entre catégories qu'on voyait en figure Fig4 semblent être globalement respectées d'un type de jeu à l'autre.
Nous pouvons conclure que les jeux d'entraînement et de test respectent à peu près les mêmes propriétés statistiques. C'est un soulagement, nous pouvons passer à la suite.

II.C - EDA multi-variée

II.C.1) Calcul de la matrice des coefficients de corrélation de type φK.

Nous avons commencé la partie multi-variée de l'EDA par le calcul des coefficients de corrélation entre les paramètres étudiés jusqu'à présents. Or, comme nous l'avons fait remarquer un peu plus haut (section II.B.1, page 10), nous ne souhaitons pas utiliser les coefficients de corrélation linéaire dit « de Pearson » car nous pouvions nous attendre à d'éventuelles relations de type non linéaires entre paramètres. De plus, les coefficients de Pearson ne peuvent être calculés qu'entre paramètres quantitatifs, ce qui exclurait de l'analyse les nombreux paramètres catégoriels (sauf *PId2*), sauf à leur faire subir d'entrée de jeu du *one hot/label encoding* pour transformer leurs chaînes de caractères en nombre. Une solution élégante à ces deux problèmes nous est apparue grâce à une discussion au sujet des corrélations entre les paramètres du jeu de donnée (*https://www.kaggle.com/competitions/spaceship-titanic/discussion/309513*). L'auteur *@rishirajacharya* y présentait de nombreux type de coefficients de corrélations, dont un « nouveau type » : la corrélation dite « **φ**K ». D'après un article sur de blog à ce sujet (*https://towardsdatascience.com/phik-k-get-familiar-with-the-latest-correlation-coefficient-9ba0032b37e7*), ce nouveau type de coefficient permet de révéler efficacement des corrélations - en particulier non linéaires - entre tout type de paramètre. Son principal défaut est de fournir des coefficients en valeur absolue, ne donnant donc pas accès à la « direction » de la corrélation. Nous avons jugé cet inconvénient mineur au regard des avantages de cette technique. La matrice des coefficients de corrélation **φ**K entre les *features* potentielles ET la *target Transported* pour le jeu d'entraînement est affichée ci-contre [Fig7].

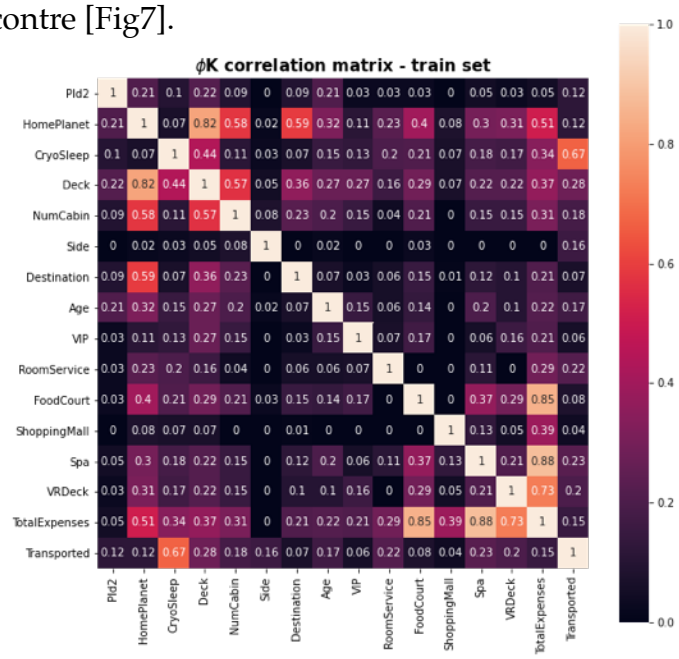


*Fig7 - Matrice des coefficients de corrélation **φ**K.*

Cette matrice révèle des corrélations entre paramètres issu d'une même colonne originelle (entre *TotalExpenses* et les cinq types de dépenses de luxes, entre *Deck* et *NumCabin*, etc) ce à quoi on pouvait s'attendre. En revanche, elle livre néanmoins quelques informations plus intéressantes :

- presque aucune corrélation entre Side et *Deck*/*NumCabin* ;
- forte corrélation entre *HomePlanet* et *Deck* (et donc *NumCabin*), mais également avec *Destination* ;
- assez forte corrélation entre *Deck* et *CryoSleep* ;
- les dépenses de luxes font des corrélations non négligeables, voir assez fortes avec *HomePlanet*, *Deck*, *NumCabin* et *CryoSleep* ; *TotalExpenses* en particulier.

Enfin, la dernière ligne de cette matrice permet un éclairage particulièrement intéressant sur le fait que le paramètre *CryoSleep* est considérablement plus corrélé que n'importe quel

autre avec l'étiquette *Transported*. Il va donc de soit que ce paramètre sera absolument à mettre en entrée de l'algorithme de prédiction. A contrario, quatre paramètres (ShoppingMail, VIP, Destination et FoodCourt) ne font même pas 0.1 de corrélation **φ**K avec l'étiquette. Il sera donc légitime de se poser la question de les conserver ou pas.

II.C.2) Distributions des *features* potentielles selon la valeur de l'étiquette.

Dans cette optique de chercher à déceler les relations éventuelles entre les distributions des *fetaures* potentielles et de notre *target*, nous avons re-tracé ci-dessous [Fig8] leurs distributions individuelles en utilisant un code couleur pour séparer les distributions selon la valeur prise par Transported. Notons que par soucis de place, nous n'avons représenté que la distribution de *TotalExpenses* parmi les six paramètres de dépenses luxueuses.

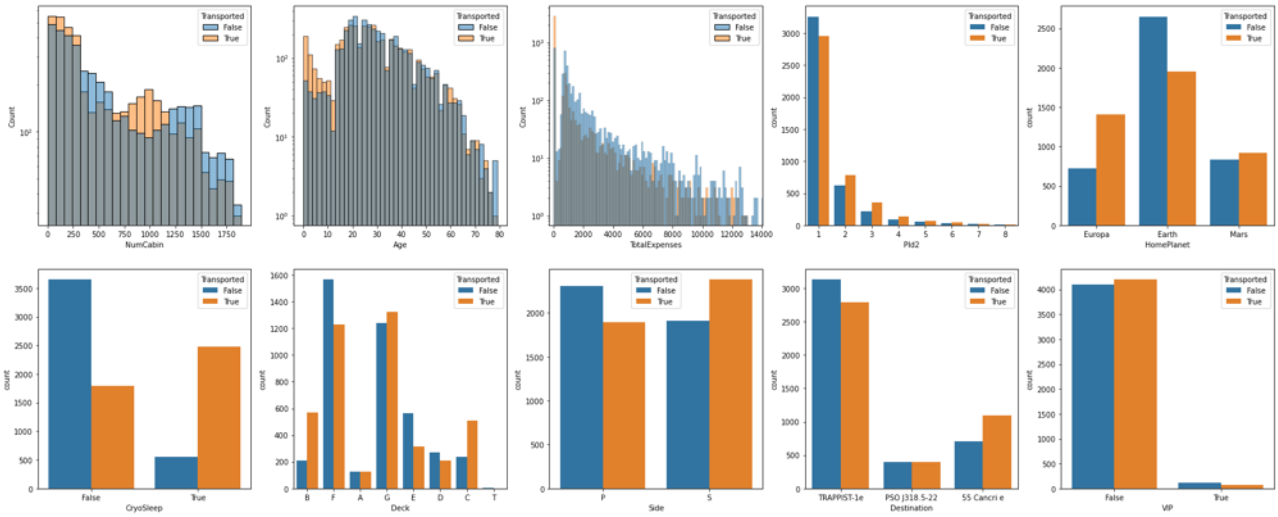


Fig8 - Distributions des features *potentielles en fonction de la valeur prise par* Transported.

Nous comprenons mieux la très forte corrélation entre *CryoSleep* et *Transported*, puisque être en sommeil cryogénique conduit quasi-certainement à la transportation vers une dimension parallèle, alors qu'être éveillé conduit bien plus souvent à l'inverse. De fait, pour *TotalExpenses*, la transportation est sur représentée pour les dépenses nulles (car être en sommeil cryogénique conduit à des dépenses nulles - ce qu'on confirmera graphiquement en section X) et inversement.
Pour les autres paramètres, de telles relations ne sont largement pas si évidentes :

- Il semble y avoir une sur-représentation de la transportation parmi les passagers de moins de dix ans, alors que les proportions sont très équilibrés pour toutes les autres classes d'âge.
- Les écarts au sein de *NumCabin* sont difficilement interprétables en l'état.
- Certaines planètes d'origines, certains ponts (*HomePlanet*/*Deck*) semblent favoriser la transportation (*Earth*/*C,B*) ou la non-transportation (*Europa*/*F,E*), mais dans des écarts moins disproportionnés que pour *CryoSleep* et *TotalExpenses*.
- Pour *VIP*, ou encore *Destination*, on est quasiment à l'équilibre (d'où leurs *coefficients* avec *Transported* très faible).

II.C.3) Distributions des *features* quantitatives en fonction des *features* qualitatives.

Passons maintenant aux relations entres paramètres, en particulier, entre paramètres quantitatifs et qualitatifs. On représente ainsi, ci-dessous, quelques exemples des distributions des features quantitatives (réduits à *NumCabin*, *Age*, et *TotalExpenses*), en fonction des valeurs prises par différentes features qualitatives (en l'occurence, *Deck* et *HomePlanet*), symbolisées grâce au code couleur [Fig9]. On fera remarquer au lecteur que par soucis d'économie d'écriture (ce rapport étant déjà très long), on ne représente ici que les relations les plus significatives (au sens des coefficients de corrélations). Des représentations bien plus exhaustives ont été faites dans le notebook (voir section **I.D.2**, cellules **43-44**).

Les informations représentées corroborent les coefficients de corrélations calculées précédemment. Ainsi :

- On voit que la planète d'origine conditionne l'âge moyen (plus élevé pour les habitants d'Europa et Mars), mais aussi le numéro de cabine (les bas numéro sont surtout occupés par les habitants d'Europa), et le montant total dépensé (hors dépenses nulles, chaque planète d'origine domine une portion du spectre des dépenses).
- De même, il y a des répartitions claires de numéro de cabine et de dépenses en fonction du pont d'appartenance des passager.

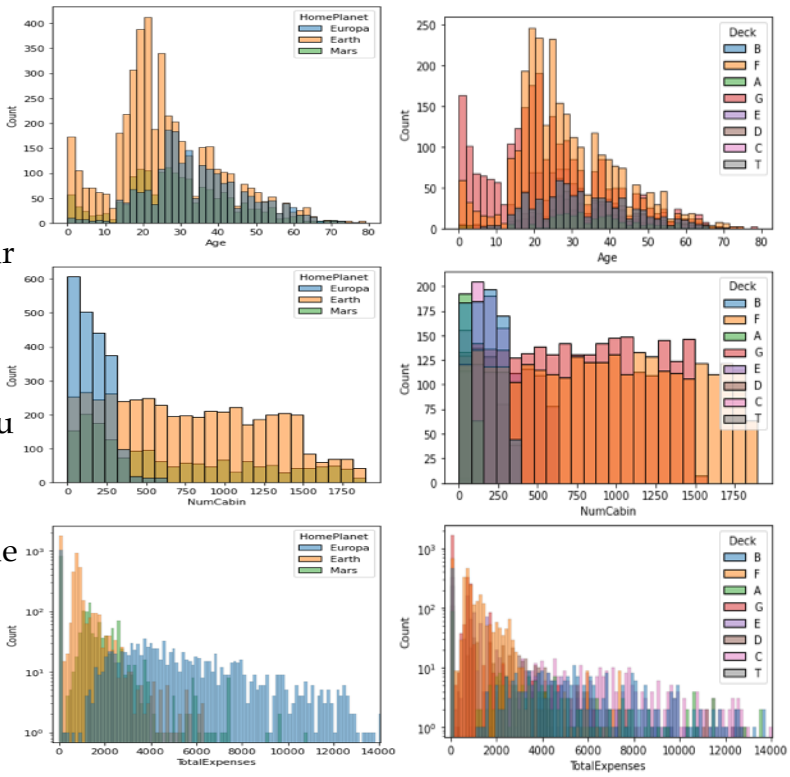


Fig9 - Distributions des features *quantitatives en fonction de catégories prises par des* features *qualitatives.*

Ce sont là parmi relations les plus importantes entre *features*, et nous les avons utilisées dans la mesure du possible lors de la phase de remplissage des valeurs manquantes (voir en section III.A, page 17 de ce rapport). C'est également au sein de ces figures qu'on montre

II.C.4) Distributions bi-variées des *features* qualitatives entre elles.

Nous ne représenterons pas la figure correspondant aux trente-six distributions bi-variées formées par chaque couple de *features* qualitatives (section **I.D.3** du *notebook*, cellules **45**) par soucis de lisibilité d'une figure d'une telle taille au sein de ce rapport. Néanmoins, nous pouvons mentionner que ces distributions, à l'image de celles présentées en Fig9, permettent de mieux comprendre les valeurs des coefficients de corrélations **φ**K entre certaines features, en particulier entre *HomePlanet*, *Deck* et *CryoSleep*, par exemple.

II.C.5) Importance de paramètres non considérés comme des *features* potentielles.

Enfin, nous aurons un mot pour l'importance de paramètres tels que *PId1* ou encore *LastName*, pour lesquels on a déjà justifié pourquoi ils ne seraient pas retenus comme des features. Ces paramètres permettent néanmoins de recouper des informations importantes entre elles. On a montré dans le *notebook* (section **I.D.4**, cellules **47-50**) les que les passagers prenant la même valeur pour *PId1* avait souvent le même nom de famille, et partageait souvent la même cabine (même pont, même numéro, même côté). En effet, il s'agit souvent de passagers de la même famille. Cette relation est à mémoriser, car nous l'avons utilisée lors du remplissage des valeurs manquantes.

III - Pré-traitement des données, *feature engineering* (2)

III.A - Remplissage des valeurs manquantes.

Tout d'abord, précisions que nous avons fait le choix de remplir les valeurs manquantes en tenant compte des relations existantes entre les différents paramètres du jeu de données, au lieu de choisir la simplicité (en imposant un remplissage constant, égal à la valeur la plus fréquente par paramètre, par exemple).
Nous avons procédé aux remplissages des valeurs manquantes en suivant globalement l'ordre décroissant des coefficients de corrélation **φ**K faits avec l'étiquette *Transported*. En effet, puisque nous avons utilisé de nombreuses relations entre *features* pour ce faire, nous savions que plus nous remplissions de valeurs manquantes pour les premiers paramètres, plus nous allions utiliser des valeurs « supposées » [*i.e.* d'anciennes valeurs manquantes] pour remplir celles des paramètres suivants, risquant ainsi de propager et d'amplifier de plus en plus d'erreurs à mesure que le remplissage progressait. En remplissant les paramètres selon cet ordre, on a sans doute minimisé les erreurs de prédictions induites par ce remplissage de valeurs.
L'ensemble des travaux réalisés à cet effet sont décrits en section II.A (cellules 51-88). Nous nous contentons ici de décrire le remplissage de quelques paramètres seulement, afin de donner une idée au lecteur de la relative complexité de la tâche entreprise.

III.A.1) Pour le paramètre *CryoSleep*.

D'après la matrice en Fig7, c'est *CryoSleep* qui est - et de loin - le plus fortement corrélé à *Transported*. C'est donc par lui que nous avons commencé, et nous avons procédé en deux temps :

- Comme montré dans le *notebook* (section **II.A.1**), on a rempli la quasi-totalité de ses valeurs manquantes grâce à la relation entre *CryoSleep* et les paramètres de dépenses : si toutes les dépenses sont nulles, nous avons passé *CryoSleep* = *True*, et inversement, si la moindre dépense est non nulle, alors nous passons *CryoSleep* = *False*.
- Pour les quelques valeurs restantes (une dizaine), nous avons regroupé les passagers par valeur prise pour *Deck*, et nous avons attribué la valeur la plus fréquente que prenait *CryoSleep* aux passagers pour lesquels ce paramètre n'était pas défini.

III.A.2) Pour les paramètres *Deck*, *NumCabin*, et *Side*.

C'est sans doute les paramètres pour lesquels cette tâche de remplissage a été la plus longue (section **II.A.2**, cellules **57-68** du *notebook*). Elle a été réalisée en trois temps :

- Tout d'abord, nous avons utilisé le fait que des passagers prenant la même valeur de *PId1* partageait souvent LA même cabine (car souvent issues de la même famille). Nous avons donc cherché, pour les passagers sans cabine définie, s'il y avait d'autres personnes au sein du même groupe, et le cas échéant, nous leur avons attribué les mêmes valeurs pour *Deck*, *NumCabin*, et *Side*. Cette phase nous a permis de remplir environ la moitié des valeurs manquantes.

- Deuxième phase, qui fut la plus complexe, de loin. Nous avons d'abord recensé l'ensemble des cabines libres, c'est à dire l'ensemble des valeurs encore jamais prises simultanément par *Deck*, *NumCabin*, et *Side (*dans la limite imposée par la valeur maximale de *NumCabin* en fonction la valeur prise par *Deck*, voir Fig9*)*. Toutes ces valeurs « possibles mais fictives » de cabines ont été répertoriées dans un dictionnaire, avec comme clef le pont correspondant. Nous avons ensuite déterminé quels étaient les ponts possibles pour les passagers sans cabine en fonction de la valeur prise par HomePlanet. Puis, nous avons calculé les valeurs médianes que prenait *TotalExpenses* pour chaque groupe de passagers déterminés par (*HomePlanet*, *Deck*). Grâce à cela, on regardait quel groupe prenait la valeur médiane la plus proche de la valeur prise par *TotalExpenses* pour chaque passager sans cabine. Ainsi, on pouvait déterminer une valeur pour *Deck*, et grâce au dictionnaire précédent, attribuer l'une des cabines libres correspondant à ce pont. Grâce à cette méthode, nous avons rempli quasiment l'autre moitié restante des valeurs manquantes de *Deck*, *NumCabin*, et *Side*.
- Troisième phase : Pour les derniers passagers sans cabine (moins d'une dizaine), la valeur de HomePlanet était également inconnue. Donc on a repris l'idée précédente, mais en calculant les médianes de *TotalExpenses* par valeur prise par *Deck* seulement.

Notons que cette méthode n'a été conçue qu'à partir de la 2nde version du notebook (de suffixe *_02.ipynb*, présente dans la branche *filling_missing_values* de GitHub).

III.A.3) Pour les autres paramètres.

Pour les autres paramètres, nous avons souvent eu recourt à des méthodes similaires, d'abord en exploitant les relations qu'on a mises en évidence précédemment (entre *CryoSleep* et *TotalExpenses*, entre *HomePlanet* et *Deck*, etc), puis en se basant sur des calcules de valeurs médianes par passagers groupé par couple de paramètres. Une exception notable : *Age* (section **II.A.5**, cellules **83-85** du *notebook*) dont les valeurs manquantes ont été remplies en utilisant la moyenne de ses 10 plus proches voisins (la proximité étant calculée par la distance euclidienne, et dans l'espace déterminé par les six paramètres de dépenses).

III.B - Sélection finale des *features* pertinents

Comme montré dans le notebook (section **II.B.1**, cellules **89-91**), le remplissage des valeurs manquantes a peu modifié la matrice des coefficients de corrélation **φ**K, en particulier ceux formés entre les différentes paramètres et l'étiquette *Transported*.
Nous avons donc maintenu notre intuition initiale (voir section II.C.1, page 14 de ce rapport), et conservé comme caractéristiques pour notre algorithme les paramètres ayant un coefficient de corrélation **φ**K supérieur ou égal à 0.1 (c'est à dire qu'on a laissé tombé *Destination, VIP, FoodCourt* et *ShoppingMail*).
Grâce au paramètre TrainOrTest, nous avons donc pu constituer nos matrices de *features* et nos vecteurs *target* pour les jeux d'entraînement et de test, notés classiquement (et respectivement) *X_train*, *y_train*, *X_test* et *y_test*. On donne un aperçu de X_train ci-dessous [Fig10] :

	PId2	HomePlanet	CryoSleep	Deck	NumCabin	Side	Age	RoomService	Spa	VRDeck	TotalExpenses
0	1	Europa	False	B	0.0	P	39.0	0.0	0.0	0.0	0.0
1	1	Earth	False	F	0.0	S	24.0	109.0	549.0	44.0	736.0
2	1	Europa	False	A	0.0	S	58.0	43.0	6715.0	49.0	10383.0
3	2	Europa	False	A	0.0	S	33.0	0.0	3329.0	193.0	5176.0
4	1	Earth	False	F	1.0	S	16.0	303.0	565.0	2.0	1091.0
...	...	...	...	...	...	...	...	...	...	...	...
8688	1	Europa	False	A	98.0	P	41.0	0.0	1643.0	74.0	8536.0
8689	1	Earth	True	G	1499.0	S	18.0	0.0	0.0	0.0	0.0
8690	1	Earth	False	G	1500.0	S	26.0	0.0	1.0	0.0	1873.0
8691	1	Europa	False	E	608.0	S	32.0	0.0	353.0	3235.0	4637.0
8692	2	Europa	False	E	608.0	S	44.0	126.0	0.0	12.0	4826.0

8693 rows × 11 columns

Fig10 - Allure de la matrice des features *sur le jeu d'entraînement.*

III.C Encodage de *features*

Comme on le voit grâce à la figure Fig10, après avoir rempli les valeurs manquantes, il nous faut désormais convertir les *features* qui contiennent des chaînes de caractères et des booléens en nombre pour pouvoir être mis en entrée des modèles de *machine learning* qu'on va tester. Nous avons essentiellement le choix entre deux type d'encodage : le *one hot encoding*, ou le *label enconding* ; le premier type devant s'appliquer lorsque les valeurs prises par une feature ne présentent pas d'ordre hiérarchique évident, le deuxième lorsqu'au contraire il existe de tels types de relations entre les valeurs prises.
Nous avons donc appliqué le *one hot encoding* à *HomePlanet*, *CryoSleep*, et *Side*, créant ainsi sept nouvelles colonnes de 0 et de 1, en remplacement des trois features précédentes, et explicitement nommées en fonction des catégories correspondantes.
Au contraire, *Deck* a subi du *label encoding* pour deux raisons :

- En nous basant sur la sociologie des passagers du Titanic de 1912, il existe une corrélation entre niveau de richesse et pont de résidence (le pont A accueillant les clients les plus fortunés, et au contraire le pont G accueillant les plus pauvre, le pont T étant réservé visiblement à des membres d'équipages). Cette corrélation est confirmé en figure Fig9, pour la distribution de *TotalExpenses* en fonction de *Deck.*
- En nous basant sur les plans du Titanic, nous constatons que plus la lettre est élevée, plus le pont est situé dans les tréfonds du navire. Il existe donc une hiérarchie topologique entres les différents ponts, et cela a pu avoir son importance au moment d'accéder aux embarcations.

Deck conserve son intitulé mais voit donc ses valeurs modifiées, selon le schéma suivant : plus la lettre est haute, plus le chiffre est élevé.
On donne un aperçu de X_train après encodage ci-dessous [Fig11].

	PId2	x0_Mars	x0_Europa	x0_Earth	x0_S	x0_P	x0_True	x0_False	Deck	NumCabin	Age	RoomService	Spa	VRDeck	TotalExpenses
0	1	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1	0.0	39.0	0.0	0.0	0.0	0.0
1	1	0.0	0.0	1.0	1.0	0.0	0.0	1.0	5	0.0	24.0	109.0	549.0	44.0	736.0
2	1	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0	0.0	58.0	43.0	6715.0	49.0	10383.0
3	2	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0	0.0	33.0	0.0	3329.0	193.0	5176.0
4	1	0.0	0.0	1.0	1.0	0.0	0.0	1.0	5	1.0	16.0	303.0	565.0	2.0	1091.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8688	1	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0	98.0	41.0	0.0	1643.0	74.0	8536.0
8689	1	0.0	0.0	1.0	1.0	0.0	1.0	0.0	6	1499.0	18.0	0.0	0.0	0.0	0.0
8690	1	0.0	0.0	1.0	1.0	0.0	0.0	1.0	6	1500.0	26.0	0.0	1.0	0.0	1873.0
8691	1	0.0	1.0	0.0	1.0	0.0	0.0	1.0	4	608.0	32.0	0.0	353.0	3235.0	4637.0
8692	2	0.0	1.0	0.0	1.0	0.0	0.0	1.0	4	608.0	44.0	126.0	0.0	12.0	4826.0

8693 rows × 15 columns

Fig11 - Allure de la matrice des features *sur le jeu d'entraînement après encodages.*

On note que suite au *one hot encoding* avec remplacement des colonnes ayant subi l'encodage, on passe de onze à quinze colonnes :

III.D Chercher à optimiser le *scaling* des données encodées

Après l'encodage, on s'est penché sur la problématique des différences d'échelles (*scaling*) entres les distributions des différentes *features*. On représente ainsi la description statistique de chaque *feature* post-encodage dans la capture d'écran ci-dessous [Fig12] :

	PId2	x0_Mars	x0_Europa	x0_Earth	x0_S	x0_P	x0_True	x0_False	Deck	NumCabin	Age	RoomService	Spa	VRDeck	TotalExpenses
count	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.000000	8693.00000	8693.000000	8693.000000	8693.000000	8693.000000
mean	1.517773	0.206488	0.250086	0.543426	0.501438	0.498562	0.360175	0.639825	4.295180	600.178419	28.82242	204.462211	284.843725	267.697400	1334.809272
std	1.054241	0.404808	0.433087	0.498139	0.500027	0.500027	0.480079	0.480079	1.782546	512.615884	14.35309	640.842415	1101.239148	1050.323772	2729.003490
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
50%	1.000000	0.000000	0.000000	1.000000	1.000000	0.000000	0.000000	1.000000	5.000000	425.000000	27.00000	0.000000	0.000000	0.000000	646.000000
max	8.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	7.000000	1894.000000	79.00000	14327.000000	22408.000000	20336.000000	35987.000000

Fig12 - Allure de la matrice des features *sur le jeu d'entraînement après encodages.*

On y a constaté des différences parfois importantes en terme d'écart-types et de valeurs maximales. Or, il est connu que certain algorithmes sont sensibles à ce genre de différences de *scaling*, et pourrait baser leur prédictions uniquement sur certaines features. Nous avons donc décider de mettre à l'épreuve différents types de *scaling*, en comparant les taux de précision d'un algorithme simple sur des jeux de données passés par différents *scalings*.
Notons que ce travail, en apparence absent du notebook, a en réalité été conduit lors de la 3ème version de celui-ci (celle de suffixe *_03.ipynb*, présente dans la branche *encoding_and_testing_scalings* du répertoire GitHub en ligne).
L'algorithme simple était une régression logistique, dont nous avons préalablement optimisé plusieurs de ses hyper-paramètres, et dont le taux de précision est calculé en moyenne lors d'une validation croisée à cinq plis sur le jeu d'entraînement.

Nous avons confronté trois types de *scalings* :

- Le 1er était le *scaling* en l'état du jeu de donnée après l'encodage.
- Le 2ème consistait à passer les six *features* de dépenses luxueuse, ainsi que *Deck* et *NumCabin*, par la fonction « log+1 » afin de réduire l'importance de leurs valeurs maximales, puis de normaliser l'écart-type de tous les *features* à 1.
- Le 3ème consistait à passer tous les *features* par un *QuantileTransformer*, de sorte à lisser leurs distributions sans chercher forcément la normalisation.

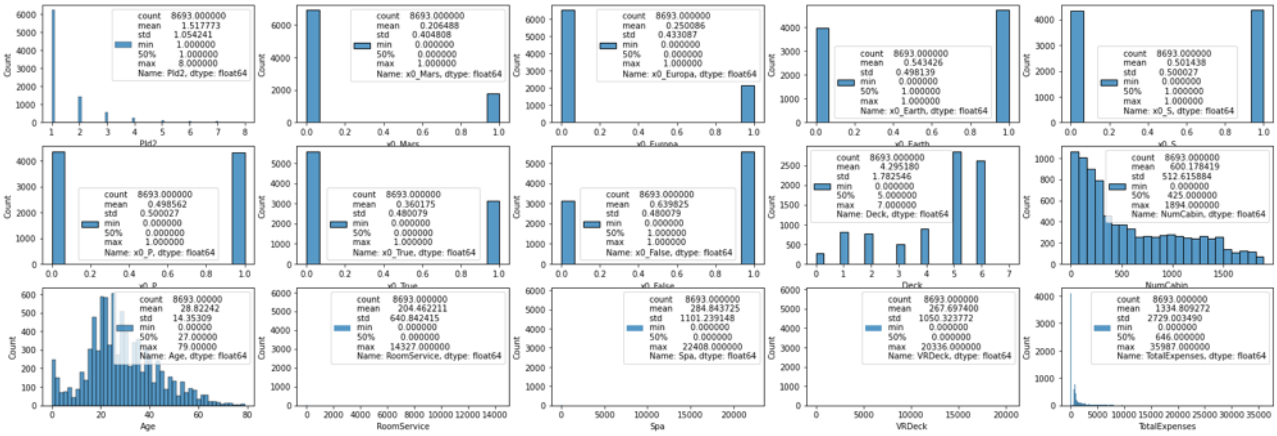


Fig13 - Features *post-encodage sans rescaling.*

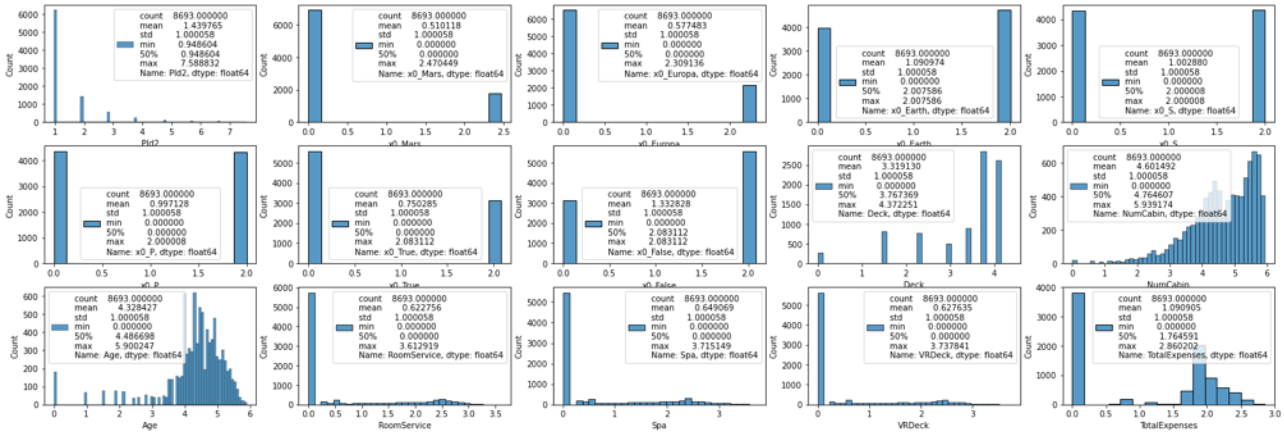


Fig14 - Features *post-encodage après passage au log+1 et normalisation des écart-types.*

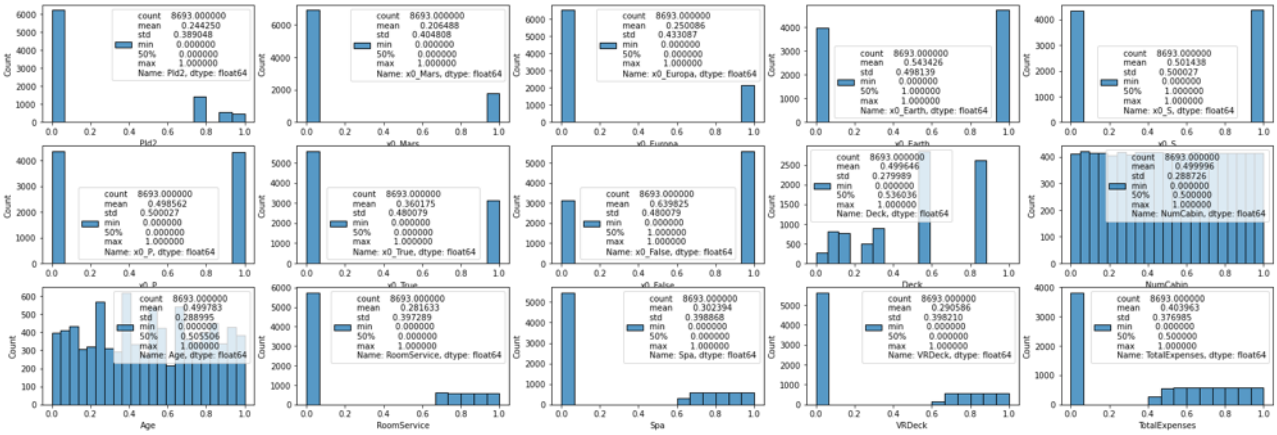


Fig15 - Features *post-encodage après passage par un* QuantileTransformer.

Les distributions individuelles des *features* selon ces trois types d'échelles sont représentées ci-dessus [dans le même ordre, respectivement Fig13, Fig14, Fig15], afin de pouvoir comparer l'effet des ces différentes techniques de *scaling* appliquées aux données. On notera que pour les *features* ayant subi le *one hot encoding*, les effets sont minimes, du fait de la nature de leur spectre de valeurs possible [0 ou 1].
Les résultats de la validation croisée d'une régression logistique aux hyper-paramètres optimisés sur ces jeux sont récapitulés dans le tableau suivant :

Type de scaling :	Valeurs des hyper-paramètres optimisés :	Meilleur taux de précision (en moyenne sur 5 plis) :
Laissé en l'état à la sortie de l'encodage	{C : 0.01 ; penalty : L1 ; solver : liblinear}	0.7799
Passage au log+1 et normalisation de l'écart-type	{C : 0.1 ; penalty : L1 ; solver : liblinear}	0.7636
Passage par un *QuantileTransformer*	{C : 0.01 ; penalty : L2 ; solver : sage}	0.7373

Ce travail salutaire a révélé que le meilleur modèle était de ne pas faire de *rescaling*, bien que l'écart avec le 2nd type de *rescaling* soit faible. Le passage par un *QuantileTransformer* est clairement contre-productif.
On a donc laissé l'échelle des données dans l'état où elles trouvent à l'issue de la phase d'encodage, et à partir des la 4ème version du *notebook* (consacrée à la comparaison des performances d'algorithmes), ce travail sur les différentes échelles a été supprimé, demeurant seulement une trace écrite explicative et un affichage de la 1ère ligne du tableau précédent, en guise de témoignage du travail accompli (celle de suffixe *_03.ipynb*, présente dans la branche *encoding_and_testing_scalings* du répertoire GitHub en ligne).

IV - Comparaisons d'algorithmes de *machine learning*

IV.A - Justifications de nos choix d'algorithmes

Nous allons enfin décrire le coeur même du sujet, à savoir les performances de différents modèles de machine learning, afin d'en sélectionner explicitement un, en vue de faire une prédiction sur le jeu de test et de pouvoir soumettre ce résultat sur Kaggle. Il fallait donc commencer par choisir les algorithmes de classification à comparer. Or, lors de notre découverte de cette compétition, nous sommes assez vite tombés sur un *kernel* Kaggle assez élégant et exhaustif créé par *@odinsOn* (*https://www.kaggle.com/code/odins0n/spaceship-titanic-eda-27-different-models*). Ce travail se caractérisait notamment par la comparaison de 27 modèles de classifications « d'un seul coup » grâce à une librairie python nommée *lasyclassifier*. Plutôt que de nous lancer dans une comparaison d'un grand nombre de modèles (ce qui n'aurait par ailleurs rien apporté de neuf par rapport au travail accompli par cet internaute), nous avons choisi de nous concentrer plutôt sur une poignée modèles de classification, et de privilégier l'optimisation de leurs hyper-paramètres (chose qui n'est pas faite dans le projet *https://www.kaggle.com/code/odins0n/spaceship-titanic-eda-27-different-models*).
Notre choix s'est donc porté sur deux types d'algorithmes :

- 1) **AdaBoostClassifier** : Déjà utilisé dans le travail de *@odinsOn*, c'est un des meilleurs algorithmes en termes de taux de précision, de f1-score, et de temps de calcul. Par ailleurs, c'est un des meilleurs algorithmes testés par la fonction *lasyclassifier* qui soit directement accessible via la librairie *sklearn*, déjà installée dans l'environnement virtuel de mon ordinateur.
- 2) **MLPClassifier** : Un modèle de classification basée sur des réseaux de neurones, accessible via *sklearn*, et non testé dans le notebook de *@odinsOn*.

On notera que ce travail que ce travail d'optimisation d'hyper-paramètres et de comparaison de performances a été réalisé lors de l'écriture de la 4ème version du notebook (celle de suffixe *_04.ipynb* sur le drive, et présente sans suffixe dans la branche *testing_classifiers* du répertoire GitHub en ligne). L'hyper-optimisation (et la validation croisée allant de paire à effectuer sur chaque combinaison possible d'hyper-paramètres) a été effectué grâce à un objet python de type **GridSearchCV**.

IV.B - AdaBoostClassifer

IV.B.1) Première phase d'optimisation.

De tous les hyper-paramètres possibles, nous n'avons tenté de n'en optimiser que deux, que nous comprenions le mieux la portée : le nombre d'estimateurs (*n_estimators*) (l'estimateur de base de cet algorithme ensembliste étant l'arbre de décision), et donc le taux d'apprentissage (*learning_rate*). Pour un premier essai d'hyper-optimisation, nous avons testé :

- $n\_estimators \in \{10; 20; 50; 75; 100\}$,

- *learning_rate* ∈ {0.1; 1; 2; 5; 10}.

Le meilleur résultat a été obtenu pour **(*n_estimators*, *learning_rate*)=(50, 1)**, avec une précision sur jeu de validation en moyenne sur cinq plis de **0,7795** (on a tronqué délibérément la valeur à quatre décimales).

IV.B.2) Seconde phase d'optimisation.

On a cherché à voir si, en affinant ces valeurs d'hyper-paramètres, on pouvait améliorer significativement la précision. Nous avons donc testé :

- *n_estimators* ∈ [25 : 75 : 5] (lire « de 25 à 75 - exclu - par pas de 5 »),
- *learning_rate* ∈ [0.2 : 2 : 0.1].

Le meilleur résultat a été obtenu pour **(*n_estimators*, *learning_rate*)=(45, 1.1)**, avec une précision sur jeu de validation en moyenne sur cinq plis de **0,7828**. On a donc une amélioration infime du résultat en regard des efforts supplémentaires déployés. Nous n'avons donc pas cherché à raffiner davantage les hyper-paramètres.

IV.B.3) *Features importance.*

On a en revanche profité des propriétés de ce modèle pour tracer, à partir du meilleur modèle optimisé précédemment, l'importance « Gini » des différentes *features* lors des chemins de décisions crées par l'ensemble des estimateurs du classeur AdaBoost (représenté ci-dessous [Fig16]).

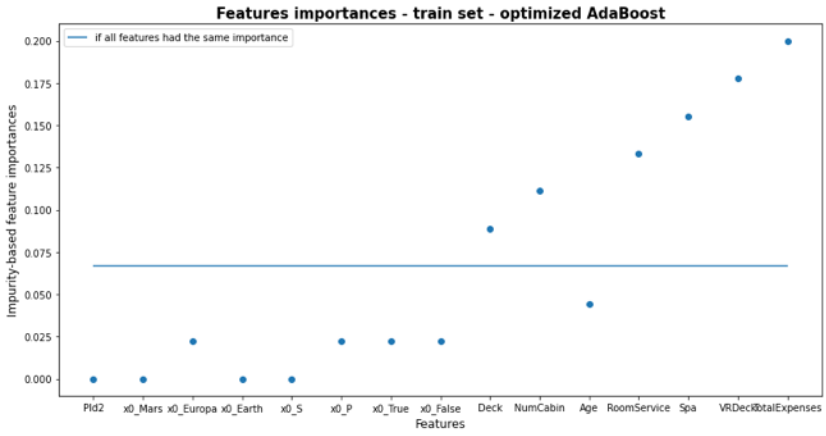


Fig16 - Importance des features pour le meilleur modèle issu de l'optimisation d'*AdaBoostClassifier.*

Plusieurs *features*, en particuliers celles issues du *one hot encoding*, ont une importance nulle. Seuls six *features* sur quinze ont une importance supérieure à l'importance moyenne. Cela remet peut être en cause nos choix de *features* et/ou d'encodage.

IV.C - MLPClassifier

IV.C.1) Première phase d'optimisation avec *solver='adam'*.

Cette fois ci, nous avons tenté d'optimiser un peu plus d'hyper-paramètres. Or les hyper-paramètres disponibles dépendaient en partie de la valeur de l'hyper-paramètre **solver**,

qui détermine l'algorithme de descente de gradient. Nous avons donc commencé par fixé *solver* sur l'algorithme « Adam », puis nous avons testé :

- *learning_rate* ∈ {'invscaling'; 'constant'},
- *hidden_layer_sizes* ∈ {(100,); (125,); (150,); (175,); (200,)},
- *activation* ∈ {'logistic'; 'tanh'; 'relu'},
- *learning_rate_init* ∈ numpy.logspace(-5, 0, 6),
- *beta_1* ∈ numpy.logspace(-4, numpy.log$_{10}$(0.9), 4),
- *beta_2* ∈ numpy.logspace(-4, numpy.log$_{10}$(0.99), 3).

Le meilleur résultat a été obtenu pour :

- ***activation = 'logistic',***
- ***beta_1 = 1e-4,***
- ***beta_2 = 9.9499e-3,***
- ***hidden_layer_sizes = (100,),***
- ***learning_rate = 'invscaling',***
- ***learning_rate_init = 1e-4,***

avec une précision sur jeu de validation en moyenne sur cinq plis de **0,7898** (on a tronqué de nouveau la valeur à quatre décimales).
Notons que préalablement, nous avions imposé comme hyper-paramètres non soumis à l'optimisation de procéder à l'apprentissage sur 30 itérations au maximum, et en dotant le réseaux de neurones d'un objet de type **early_stopping** afin d'éviter de pousser jusqu'à 30 itérations en cas de stagnation de l'apprentissage sur plus de 10 itérations. Ces précautions ont systématiquement été reproduites par la suite.

IV.C.2) Seconde phase d'optimisation avec *solver='adam'*.

Deux hyper-paramètres ayant pris comme valeur optimale leur borne inférieure de plage de variations possibles (en l'occurence *beta_1* et *hidden_layer_sizes*), nous décidons de leur refaire subir une optimisation (tous les autres hyper-paramètres étants bloqués sur les valeurs optimales précédentes). On a donc tenté les combinaisons suivantes :

- *hidden_layer_sizes* ∈ {(88,); (100,); (112,)},
- *beta_1* ∈ numpy.logspace(-4.5, -3.5, 5).

Le meilleur résultat a été obtenu de nouveau pour les mêmes valeurs que précédemment, et donc sans amélioration de la précision moyenne sur jeu de validation. Nous n'avons pas cherché à affiner d'avantage ces hyper-paramètres.

IV.C.3) Première phase d'optimisation avec *solver='SGD'*.

Comme annoncé plus haut, nous devons également tester un autre algorithme de descente de gradient, et donc les nouveaux hyper-paramètres qui en découle. On a donc testé l'algorithme « SGD », avec les combinaisons suivantes d'hyper-paramètres :

- *learning_rate* ∈ {'invscaling'; 'constant'},
- *hidden_layer_sizes* ∈ {(100,); (125,); (150,); (175,); (200,)},
- *activation* ∈ {'logistic'; 'tanh'; 'relu'},

- *learning_rate_init* ∈ numpy.logspace(-5, 0, 6),
- *momentum* ∈ numpy.logspace(-4, -1, 4),
- *nesterovs_momentum* ∈ {True, False}.

Le meilleur résultat a été obtenu pour :
- ***activation = 'relu',***
- ***momentum = 1e-2,***
- ***nesterovs_momentum = False,***
- ***hidden_layer_sizes = (125,),***
- ***learning_rate = 'invscaling',***
- ***learning_rate_init = 1e-3,***

avec une précision sur jeu de validation en moyenne sur cinq plis de **0,7766**.
Le score est moins bon que le meilleur modèle avec l'algorithme « Adam », on décide donc de ne pas poursuivre nos efforts d'optimisation.

IV.D - Comparaison des meilleurs modèles après optimisation des hyper-paramètres

Les performances des trois meilleurs modèles - issus de l'optimisation/validation croisée d'*AdaBoostClassifier*, de *MLPClassifier*('*Adam*') et de *MLPClassifier*('*SGD*') - sont présentées ci-dessous [Fig17]. A gauche, nous représentons les précisions moyennes sur 5 plis (pour la combinaison optimale d'hyper-paramètres) et à droite le temps de prédiction sur le jeu de validation, toujours en moyenne sur 5 plis. Les barres d'erreurs représentent les écarts-types correspondant.

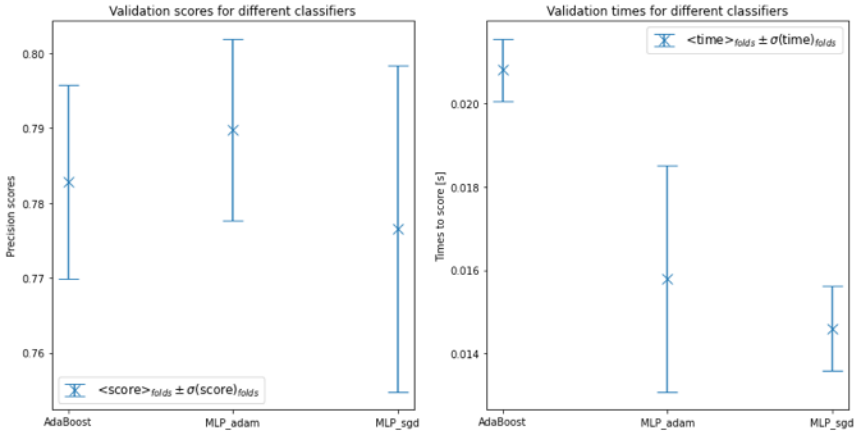


Fig17 - Performances en précision et en temps de prédiction, en moyenne sur 5 plus, sur le jeu de validation.

Le meilleur modèle en terme de précision est donc celui issu de l'optimisation de *MLPClassifier*('*Adam*'). Il n'est pas aussi bon en terme de performances temporelles que lorsque l'algorithme de descente de gradient est SGD, mais puisque le critère d'évaluation de cette compétition est la précision, et non la durée de prédiction, on s'en contentera.

Les performances du meilleur modèle, celui retenu pour la prédiction, sont récapitulée (avec l'affichage des hyper-paramètres optimisés) dans la capture d'écran ci-dessous [Fig18] :

	mean_test_score	std_test_score	mean_fit_time [s]	std_fit_time [s]
MLPClassifier(activation='logistic', beta_1=0.0001, beta_2=0.009949874371066206,\n early_stopping=True, learning_rate='invscaling', max_iter=40,\n random_state=420)	0.790524	0.013322	1.588758	0.172974

Fig18 - Performances du meilleures modèle.

On notera que la précision moyenne finale peut légèrement varier d'une exécution du notebook à une autre (passant de 0,7905 à 0,7898, par exemple). Et ceux, alors qu'on a bien veillé à bloquer le hasard en passant toujours la même constante à l'hyper-paramètre *random_state*. Notre expérience récente des réseaux de neurones (issue du projet n°6 du même parcours) nous ferait pencher pour une petite part de hasard persistante lors de la division des tâches à accomplir entre les différents coeur de processeurs de notre machine locale. C'est en tout cas la seule explication relativement crédible qu'on puisse trouver pour le moment (en tant qu'étudiant novice).

V - Prédictions sur le jeu de test, conclusions et perspectives.

On a donc utilisé ce modèle optimisé pour réaliser la prédiction sur le jeu de test requise par les organisateurs de la compétition. Les résultats (nombre de True et de False parmi les prédictions sur le jeu de test) sont restitué sous la forme d'un diagramme circulaire ci-dessous [Fig19]. Notons que par soucis de comparaison, nous avons également restitué (sur la gauche de la figure) les nombre d'occurence de chaque valeur de Transported au sein du jeu d'entraînement.

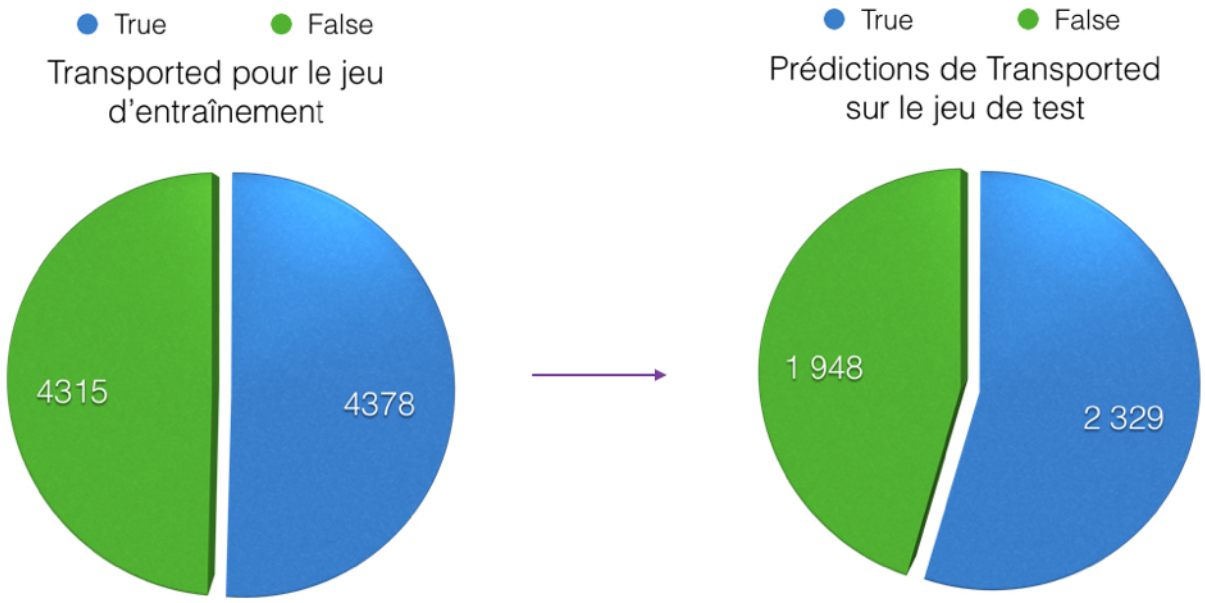


Fig19 - Proportions True/False lors des prédictions sur le jeu de test, et proportions de référence sur le jeu d'entraînement.

On peut y constater que les proportions de la prédiction sur le jeu de test (~55%/45% de True/False) sont assez proches de celles - authentiques - du jeu d'entraînement (quasiment du 50%/50%). Sachant qu'il y avait très peu de différences de distributions individuelles des features entre les deux types de jeux (voir plus haut les figures Fig5 et Fig6), ce petit écart n'est pas choquant.

En conclusions, ces résultats nous on permit , lorsque nous avons soumis à Kaggle une version quasi-finale de notre code (seule de la mise en forme demeurait à faire) d'être classé 1555/2428, soit dans le top 65%. Même si les scores sont très serrés sur des centaines de participants, c'est néanmoins un peu décevant, le temps nous aillant pousser à chercher à coder relativement vite.
Si c'était à refaire, je passerais certainement plus de temps à travailler mes *features* et à complexifier le réseau de neurones final (notamment en mettant plus de couches cachées).