# Higgs Boson Machine Learning Challenge

Adrian Ahne - Paul-Alexis Dray

February 2, 2018

Vous trouverez tous les fichiers cités dans ce document ainsi qu'un fichier **README.md** sur *Github* à l'adresse suivante :

https://github.com/padipadou/higgs_challenge

## 1 Introduction

### 1.1 Contexte

On s'intéresse ici à la bien connue détection du boson de Higgs dans des données simulées de façon à reproduire le comportement de l'expérience ATLAS. C'est donc un problème de classification binaire, avec deux classes que sont :

- *background*
- *tau tau decay of a Higgs boson*

Nous allons présenter dans ce rapport les différentes étapes de ce projet.

### 1.2 Description des fichiers fournis

- *training.csv* - Jeu de données d'entraînement de 250000 événements, dont une colonne ID, 30 colonnes de variables, une colonne de poids (ou d'importance) et une colonne d'étiquettes.
- *test.csv* - Jeu de données de test de 550000 événements avec une colonne ID et 30 colonnes de variables.
- *HiggsBosonCompetition_AMSMetric.py* - Script Python pour calculer la métrique d'évaluation de la compétition.

### 1.3 Détails généraux à propos des variables

- Toutes les variables sont en format nombre flottant, sauf *PRI_jet_num* qui est un nombre entier.
- Les variables préfixées avec **PRI** (pour **PRI**mitives) sont des quantités "brutes" concernant la collision de la particule mesurée par le détecteur.
- Les variables préfixées avec **DER** (pour **DER**ived) sont des quantités calculées à partir des entités primitives vues précédement, celle-ci ont donc été sélectionnées par les physiciens d'ATLAS. Ainsi une partie de la création de nouvelles variables, soit le *features engineering*, est réalisée en amont par des physiciens, plus à même de comprendre les phénomènes complexes du LHC (cf partie 2.4).
- Il peut arriver que, pour certaines entrées, certaines variables ne signifient rien ou ne puissent pas être calculées; dans ce cas, leur valeur est $-999$, ce qui est en dehors de la plage normale de toutes les variables(cf partie 2.3.1).
- Enfin, chaque variable est expliquée de manière succinte à la rubrique *Dataset Semantics* de la page http://opendata.cern.ch/record/328
- Il est conseillé de ne pas utiliser la variable *Weight* dans https://higgsml.lal.in2p3.fr/files/2014/04/documentation_v1.8.pdf p. 16

# 2 Partie 1: Exploration / Préparation

## 2.1 Chargement des données

Pour le chargement de données nous avons utilisé la bibliothèque *pandas*. Cette bibliothèque nous permet de faire facilement un prétraitement des données et des premières analyses rapides.

## 2.2 Analyse descriptive

Pour mieux comprendre les données nous avons fait une brève analyse descriptive dans le fichier **descriptive_analysis.py**. D'abord nous avons regardé le "sommaire" des variable (max, min, moyenne, etc.) pour obtenir une première information sur les données. Dans la figure 1, nous avons montré une petite partie de cette description. Par exemple, on observe qu'il y a souvent la valeur $-999$ dans plusieurs variables, ce qui signifie très probablement une valeur manquante.

| | EventId | DER_mass_MMC | DER_mass_transverse_met_lep | DER_mass_vis | DER_pt_h | DER_deltaeta_jet_jet | DER_mass_jet_jet | DER_prodeta_jet_j |
|---|---|---|---|---|---|---|---|---|
| count | 250000.000000 | 250000.000000 | 250000.000000 | 250000.000000 | 250000.000000 | 250000.000000 | 250000.000000 | 250000.0000( |
| mean | 224999.500000 | -49.023079 | 49.239819 | 81.181982 | 57.895962 | -708.420675 | -601.237051 | -709.3566( |
| std | 72168.927986 | 406.345647 | 35.344886 | 40.828691 | 63.655682 | 454.480565 | 657.972302 | 453.0198 |
| min | 100000.000000 | -999.000000 | 0.000000 | 6.329000 | 0.000000 | -999.000000 | -999.000000 | -999.0000( |
| 25% | 162499.750000 | 78.100750 | 19.241000 | 59.388750 | 14.068750 | -999.000000 | -999.000000 | -999.0000( |
| 50% | 224999.500000 | 105.012000 | 46.524000 | 73.752000 | 38.467500 | -999.000000 | -999.000000 | -999.0000( |
| 75% | 287499.250000 | 130.606250 | 73.598000 | 92.259000 | 79.169000 | 0.490000 | 83.446000 | -4.5930( |
| max | 349999.000000 | 1192.026000 | 690.075000 | 1349.351000 | 2834.999000 | 8.503000 | 4974.979000 | 16.6900( |

8 rows × 32 columns

Figure 1: Brève description de certaines valeurs grâce à *pandas*

Après cela, nous avons aussi tracé les histogrammes des distributions des variables dans la figure 2. On peut déjà voir que probablement la variable *PRI_jet_num* est catégorielle ou encore que l'on peut supprimer la variable *EventId*. On remarque aussi qu'il y a plusieurs variables avec des valeurs à $-999$, ce qui signifie valeurs manquantes. En plus, on peut voir que plusieurs variables ont une distribution gaussienne et donc nous pouvons les normaliser.
En considérant la variable *Label*, nous remarquons que les deux étiquettes sont à peu près équilibrées avec 79197 observations de la classe $b$ et 58899 de la classe $s$.

Pour mettre en évidence les relations entre les variables (en terme de covariance) nous avons utilisé la bibliothèque *seaborn*, voir figure 3. On peut observer que, par exemple, la variable *Label* (l'étiquette) est une variable qui est corrélée linéairement à d'autres variables, ce qui va constituer une bonne base pour nos futurs modèles. Par ailleurs, on constate que le dataset est plutôt peu corrélé en général, ainsi une ACP ne serait pas forcément très adaptée.

## 2.3 Préparation des données

Nous allons décrire brièvement les traitements que nous avons effectué sur les données. Tout d'abord nous avons transformé le type de la variable *PRI_jet_num* en *categorical*, comme elle n'est constituée que des valeurs {0, 1, 2, 3}. Pour la variable *Label* toutes les valeurs $b$ sont remplacées par 0 et $s$ par 1.
Nous avons utilisé par ailleurs un *pipeline* pour connecter toutes les étapes de prétraitement des données et le modèle (cf partie 3.1).

### 2.3.1 Traitement des valeurs manquantes

La première étape du *pipeline* traite les valeurs manquantes (classe : *Missing_values()*) avec les étapes suivantes :

- Remplacer les valeurs manquantes $-999$ par $np.NaN$
- Enlever les variables où l'on a plus de 70% de valeurs manquantes (177457 sur 250000 lignes au total) : *DER_deltaeta_jet_jet, DER_mass_jet_jet, DER_prodeta_jet_jet, DER_lep_eta_centrality, PRI_jet_subleading_pt, PRI_jet_subleading_eta, PRI_jet_subleading_phi*

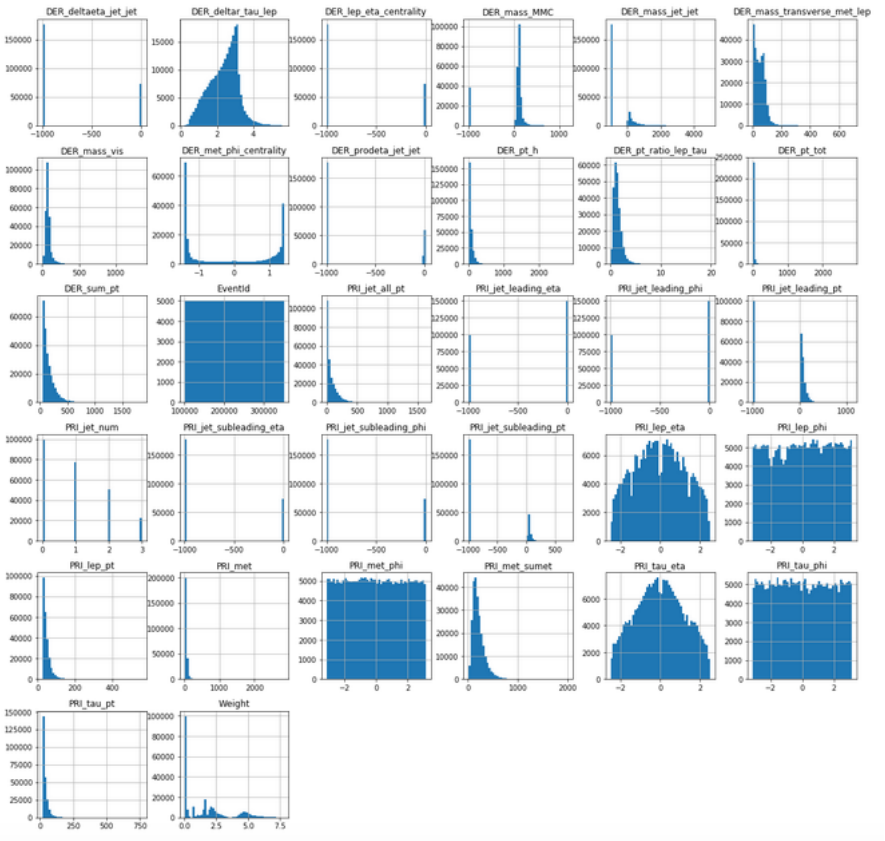


Figure 2: Histogrammes de distribution des variables

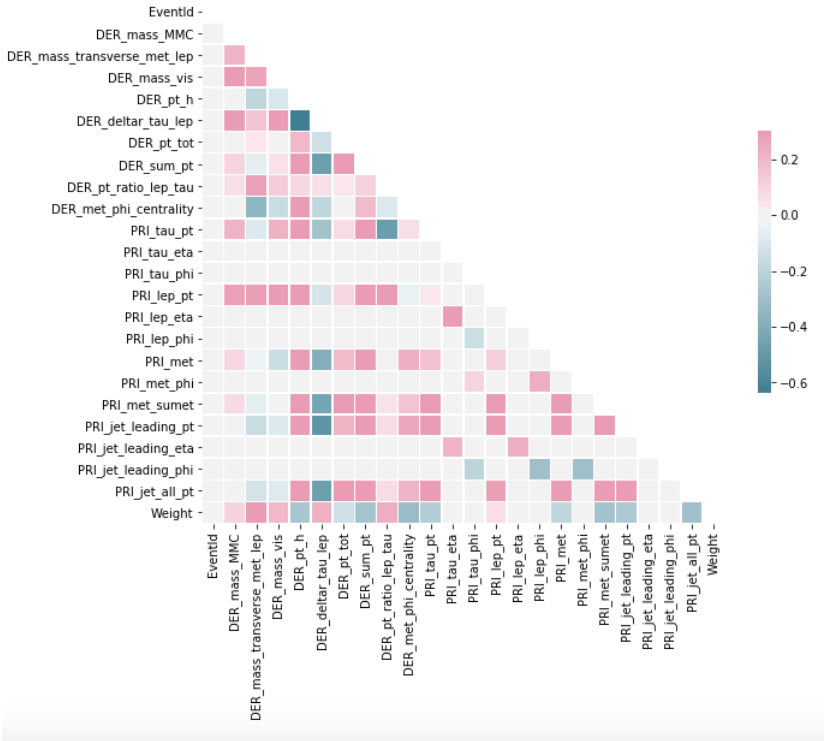


Figure 3: Matrice de covariance

- Remplacer les valeurs manquantes $np.NaN$ du reste des features par les moyennes (ici d'autres options auraient été possibles, comme remplacer par la médiane ou remplacer par la valeur la plus fréquente mais nous ne les avons pas exporées)

Ces étapes sont dans le fichier d'aide **utility.py**.

### 2.3.2 Traitement par type de variable

La prochaine étape du *pipeline* consiste en un traitement par type de variable. La fonction *Feature-Union* est utilisée pour traiter en parallèle des variables bouléennes, numériques et catégorielles où chaque type exploite un propre *pipeline*. Une classe *TypeSelector(type)* nous aide à extraire seulement les variable du type qui nous intéresse. Les traitements pour chaque type sont les suivants:

- **Boolean**: *TypeSelector(boolean)* est le seul traitement, car il n'y a pas des variable bouléenne dans les données. Nous l'avons ajouté pour des raisons de consistance.
- **Numeric**: *TypeSelecter(np.number)* pour choisir les variable flottantes, les variables vont alors être normalisées avec le *StandardScaler()* suivi d'une selection des meilleures variables par rapport à leur score *SelectKBest()*.
- **Categorical**: *TypeSelector(Categorical)* pour choisir la variable *PRI_jet_num*. Sur cette variable nous appliquons un "One-hot-encoding". Ainsi *pandas* encode les valeurs manquantes avec -1, mais la fonction *OneHotEncoder()* de *sklearn* n'accepte seulement que des valeurs positives. Nous avons ajouté, au cas où, une classe *StringIndexer()*qui remplace ces valeurs -1, par les valeurs de la variable *categorical*.

## 2.4 Features engineering

Après lecture de la documentation, nous avons compris que beaucoup de travail avait été fourni en amont par les scientifiques en charge du projet, sur la création de variables. Ainsi les variables préfixées avec *DER* correspondent à des variables construites par des physiciens plus à même de comprendre les phénomènes complexes du LHC. En effet, il nous paraît difficile de créer des variables en tant que novice en la matière, aussi facilement que lorque l'on aurait pu le faire pour d'autres projets tels que l'étude des survivants du Titanic par exemple.

# 3 Partie 2: Classification et méthodes utilisées

La conception efficace d'un modèle repose sur des principes tels que :

- réglage fin des hyperparamètres pour nos modèles
- test de nombreux modèles différents
- test de nombreuses représentations pour nos données

Ainsi nous devons également considérer :

- des milliers de configurations d'hyperparamètres possibles pour chaque modèle
- des dizaines de modèles
- des dizaines de méthodes de prétraitement

Apparait alors le *pipeline* de *Scikit-learn* qui permet d'industrialiser de nombreux processus sur les données avant de les faire entrer dane le modèle.

## 3.1 Utilisation du pipeline de Scikit-learn

Le *pipeline* nous a permis d'appliquer séquentiellement une liste de transformations ainsi qu'un modèle après ces transformations. Lors de l'utilisation, nous avons pu le paramétrer très simplement et de manière efficace en utilisant le nom de l'étape et le nom du paramètre séparé par un "__".

Nous verrons comment nous avons utilisé le *pipeline* dans la partie consacrée à l'outil *GridSearch* (cf partie 3.3). Nous allons présenter d'abord *TPOT*, un outil Python open source qui va optimiser automatiquement une série de fonctions et de modèles qui maximisent la précision de la validation croisée sur l'ensemble de données.

## 3.2 Utilisation de la bibliothèque TPOT

*TPOT* est encore en développement et repose sur des méthodes dites évolutives ou génétiques qui utilisent ainsi des algorithmes évolutifs pour rechercher dans l'espace des hyperparamètres pour un modèle donné. L'optimisation hyperparamétrique évolutive suit un processus inspiré du concept biologique de l'évolution :

- Création d'une population initiale de solutions aléatoires (i.e. génération aléatoire de tuples d'hyperparamètres)
- Evaluation des tuples d'hyperparamètres (par exemple, validation croisée de 10 fois du modèle avec ces hyperparamètres)
- Classement des tuples hyperparamétriques
- Remplacement des tuples hyperparamétriques les moins performants par de nouveaux tuples hyperparamétriques générés par croisement et mutation des autres
- Répétition des 3 dernières étapes jusqu'à ce que les performances du modèle soient satisfaisantes ou que les performances du modèle ne s'améliorent plus

L'utilisation et le fonctionnement de *TPOT* nous ont paru intéressants car nous utilisons de nombreux modèles où le réglage des hyperparamètres, qui est effectué avant l'apprentissage, est très important. Hors, il n'y a jamais de règles claires pour définir ces hyperparamètres.

L'inspiration de cette idée vient de la nature: le succès d'un individu n'est pas seulement déterminé par ses connaissances et ses compétences qu'il a acquises à travers l'expérience (soit l'apprentissage pour un modèle par exemple), ce succès dépend aussi de son patrimoine génétique (qui correspondrait aux hyperparamètres du modèle) amélioré ici par l'algorithme génétique.

Après avoir découvert cette librairie, et quelques utilisations plus tard nous nous sommes rendus compte qu'elle était un peu trop "boîte noire" à nos yeux et difficilement paramétrable. Ceci étant dû à sa relative jeunesse. Ainsi nous sommes tournés vers des outils plus conventionnels de type *GridSearch*. Cependant *TPOT* nous a quand même orienté vers des méthodes de type **arbres de décision**.
Un des meilleurs modèles proposés fût, par exemple, le *DecisionTreeClassifier* avec une précision de 81.7% et et la configuration proposée suivante de criterion="gini", max_depth=10, min_samples_leaf=19, min_samples_split=20.
Cependant, il faut préciser que nous avons ajouté un paramètre faisant en sorte de n'évaluer chaque *pipeline* qu'au maximum pendant 30 secondes. Comme nous n'avions pas assez de puissance de calcul, nous avons été obligés d'appliquer cette contrainte. Par conséquent, *TPOT* n'a peut-être pas trouvé le meilleur modèle possible parce qu'il n'avait tout simplement pas assez de temps. Vous trouverez les exemples dans **tpot_test.py**.

## 3.3 Utilisation de GridSearch

La méthode *GridSearch* correspond à une recherche exhaustive à travers un sous-ensemble spécifié manuellement de l'espace hyperparamétrique d'un ou plusieurs modèles. Un algorithme de *GridSearch* doit être guidé par une mesure de performance. Notre *GridSearch* prend en paramètre

notre *pipeline*, qui décrit le préprocessing et le modèle, et la validation croisée sur l'ensemble d'apprentissage (ici: K-fold). Pour chacun des algorithmes nous avons sélectionné des valeurs d'hyperparamètres à explorer.

### 3.3.1 Résultats

Nous avons testé plusieurs classifieurs avec *GridSearch* pour chaque modèle. Le choix des hyperparamètres pour chaque modèle est décrit dans le fichier du code. Comme nous avons installé un *pipeline* général, nous avons pu tester facilement ces différents modèles. Voici donc les meilleures configurations renvoyées par *GridSearch* pour 4 modèles différents :

| Model | Accuracy en % | Paramètres | Temps d'execution (en s) |
|---|---|---|---|
| **Logistic regr.** | 0.74 | selectKbestFeatures = 15, tol = 1e-2, penalty = 12, solver = 'lbfgs' | 23 |
| **LDA** | 0.736 | selectKbestFeatures = 15, tol=1e-2, n_components=1, solver='lsqr', shrinkage='auto' | 19 |
| **Decision tree** | 0.76 | selectKbestFeatures = 'all', splitter = 'best', presort = True, criterion = 'entropy' | 3617 |
| **Random forest** | 0.824 | selectKbestFeatures = 'all', oob_score = False, n_estimators = 15, criterion = 'entropy', max_feautres = 'auto' | 301 |

*GridSearch* a été réalisée dans le fichier **grid_search_pipeline.py** et la mesure des temps d'exécution des meilleurs modèles dans le fichier **time_measurement_best_models.py**. Le temps d'exécution signifie le temps d'apprentissage des données sur une validation croisée (*StratifiedKFold* avec 10 split et shuffle True).
Le meilleur résultat en terme d'*accuracy* est le "random forest" avec 82% suivi par le "decision tree" avec 76%. Cependant, le temps d'exécution est significativement plus long que pour la régression logistique et l'analyse discriminante linéaire. Ainsi, si l'on a assez de puissance de calcul, on peut bien utiliser le *decision tree* ou *random forest* pour obtenir une bonne performance. A l'inverse, si la puissance n'est pas exceptionnelle, il faut plutôt choisir les deux modèles plus simples, que sont la régression logistique et l'analyse discriminante linéaire, qui donnent aussi une *accuracy* largement acceptable.

## 3.4 Utilisation de Keras

*Keras* est une API dite de haut niveau qui permet de construire des réseaux de neurones, écrite en Python et capable de fonctionner sur *TensorFlow*, *CNTK* ou *Theano*. Cette librairie a été développée dans le but de permettre une expérimentation rapide. Pouvoir passer de l'idée au résultat avec le moins de délai possible apparaît comme la clé pour avancer en recherche.

Nous avons tout d'abord tenter de construire une métrique adaptée correspondant à l'AMS (cf partie 3.5). Cependant cette mesure fait intervenir les poids ("importance") de chaque échantillon (cf description des fichiers partie 1.2). Nous n'avons pas réussi à injecter les poids de ces lignes dans la métrique.
Nous avons donc décidé de construire cette métrique sans les poids (ce qui fait évidemment perdre du sens de celle-ci). L'utilisation de la fonction *custom metric* s'est encore révélée compliquée car elle implique une compilation en amont (comme prévue dans le fonctionnement de *TensorFlow*) et donc une utlisation des fonction de *keras.backend* et/ou de celles de *TensorFlow*. On arrive ici aux limites de l'utilisation dite "facile" prévue par *Keras*. Nous avons donc abandonné cette voie pour utiliser la simple précision, nous avons cependant ajouté une implémentation de l'aire sous la courbe ROC pour avoir une information supplémentaire.

Nous avons décidé d'utiliser donc l'apprentissage sans utiliser les poids relatifs à l'importance de chaque ligne. En effet, lorsque nous utilisions ces poids lors de l'apprentissage, sans utiliser de métrique les prenant en compte, nous avancions "à l'aveugle" sans pouvoir voir une évolution

intéressante puisque la précision et l'aire sous la courbe vue précedemment ne prenaient pas en compte ces mêmes poids.

Nous avons donc testé plusieurs architectures de réseaux et plusieurs fonctions d'activation en utilisant le modèle *Sequential* proposé par *Keras*.

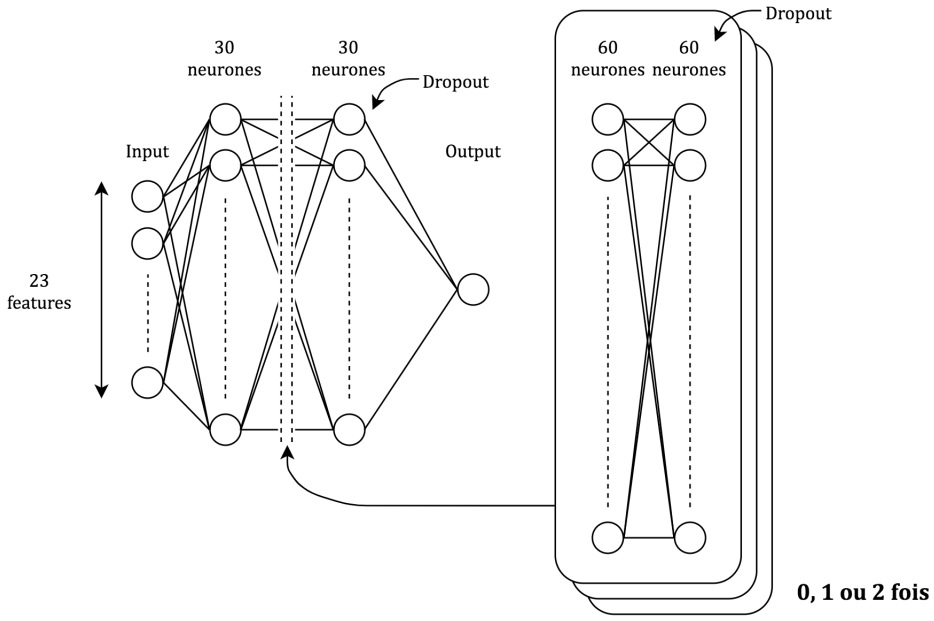


Figure 4: Les différentes architectures testées pour notre problème

1. Fonction sigmoïde :
    - 30 - 30 - 1 (1681 paramètres: figure 5)
    - 30 - 60 - 60 - 30 - 1 (8101 paramètres: figure 6)
    - 30 - 60 - 60 - 60 - 60 - 30 - 1 (15421 paramètres: figure 7)
2. Fonction Relu :
    - 30 - 30 - 1 (1681 paramètres: figure 8)
    - 30 - 60 - 60 - 30 - 1 (8101 paramètres: figure 9)
    - 30 - 60 - 60 - 60 - 60 - 30 - 1 (15421 paramètres: figure 10)
    - 30 - 60 - 60 - 60 - 60 - 60 - 60 - 30 - 1 (22741 paramètres: figure 11)

Nos architectures sont constituées d'une couche d'entrée de 30 neurones et d'une couche avant la sortie constituée, elle aussi, de 30 neurones. Nous y avons interposé 0, 1 ou 2 (voir même 3) blocs de 2*60 neurones. Nous avons utilisé soit uniquement des fonctions sigmoïde soit uniquement des fonctions Relu sauf pour le dernier neurone de sortie, toujours en sigmoïde naturellement. Nous avons ajouté des options de *Dropout* avant la sortie et à la fin de chaque bloc pour minimiser l'*overfitting*. Cette architecture est détaillée de façon visuelle dans la figure 4.

Chaque architecture nous a permis de produire des résultats que vous trouverez dans les figures 5 à 11. Ces modèles ont été générés grâce au fichier **keras_test.py**, les poids calculés sont sauvegardés dans le répertoire **models**, ainsi on peut tout à fait imaginer les entraîner davantage si besoin. Sur notre serveur, chaque batch tournait en 40-50 secondes, donc 40 *epochs* en 30 minutes environ.

Comme résumé dans la figure 12, nous avons décidé de garder le modèle intitulé "nn_relu_30_2x60_30_1_40epochs" pour sa simplicité et ses résultats plus que corrects. Celui ci ne contient donc que des fonctions Relu (qui produisent de meilleurs résultats que les sigmoïdes ici) et un bloc de 2*60 couches cachées.

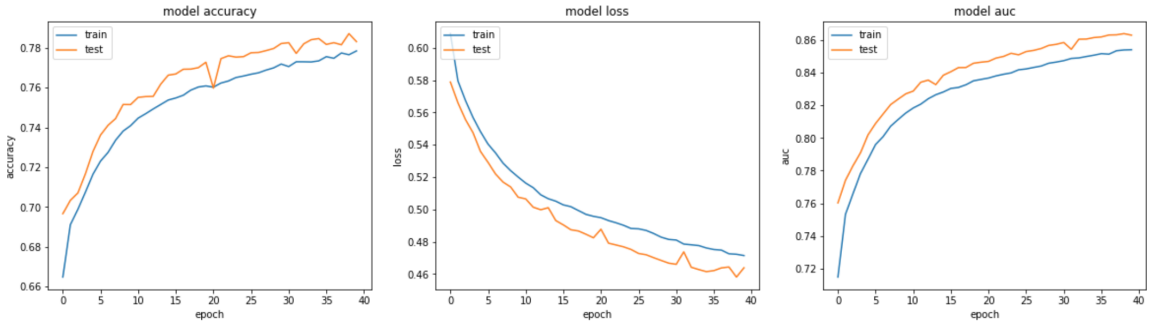


Figure 5: Fonction sigmoïde : 30 - 30 - 1

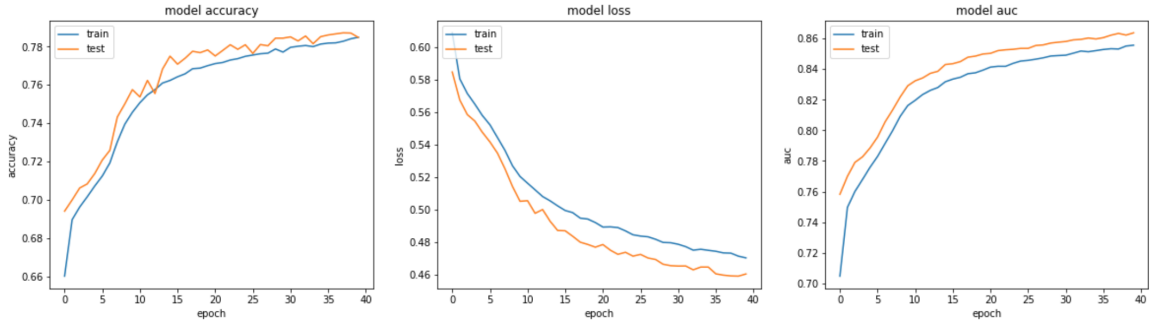


Figure 6: Fonction sigmoïde : 30 - 60 - 60 - 30 - 1

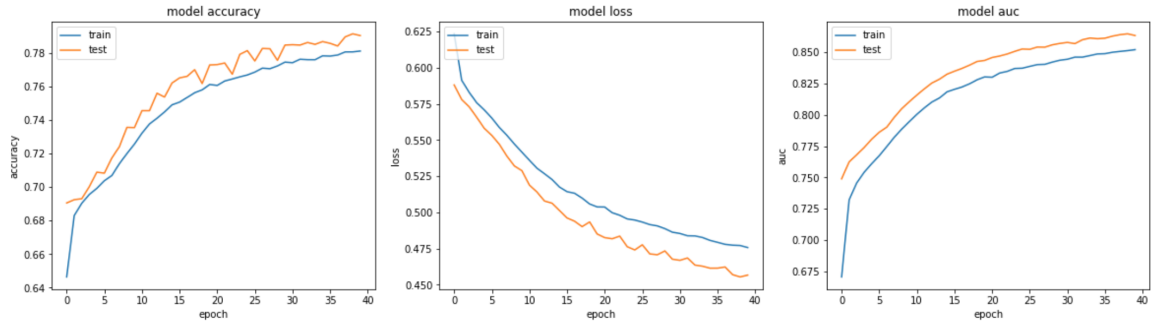


Figure 7: Fonction sigmoïde : 30 - 60 - 60 - 60 - 60 - 30 - 1

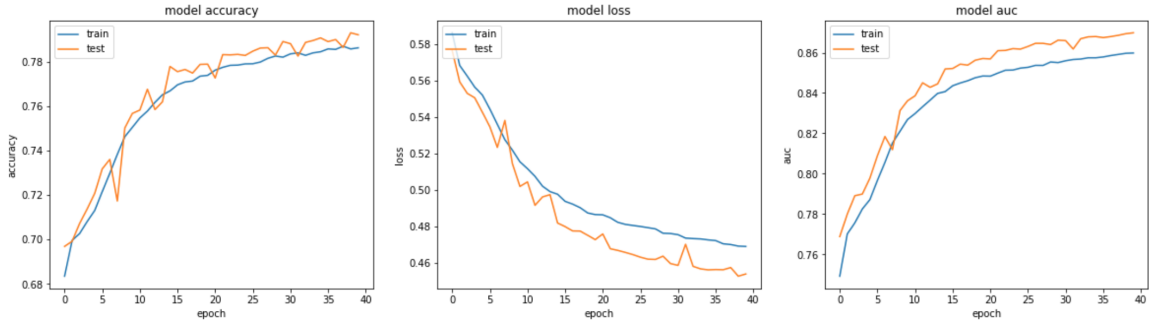


Figure 8: Fonction relu : 30 - 30 - 1

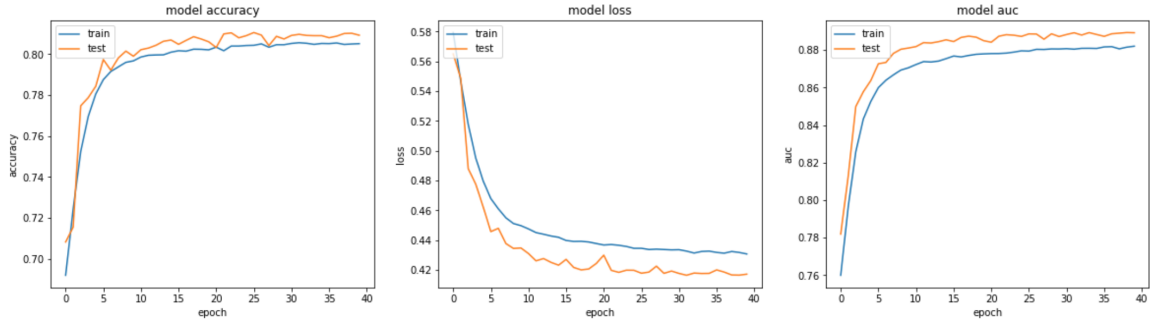


Figure 9: Fonction relu : 30 - 60 - 60 - 30 - 1

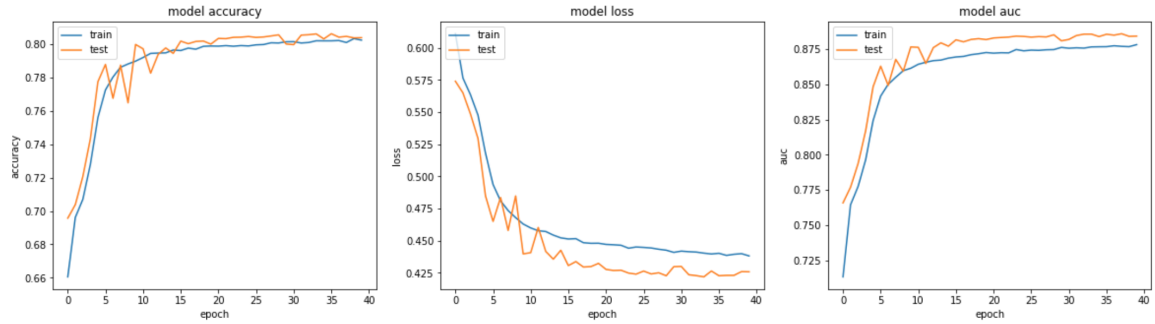


Figure 10: Fonction relu : 30 - 60 - 60 - 60 - 60 - 30 - 1

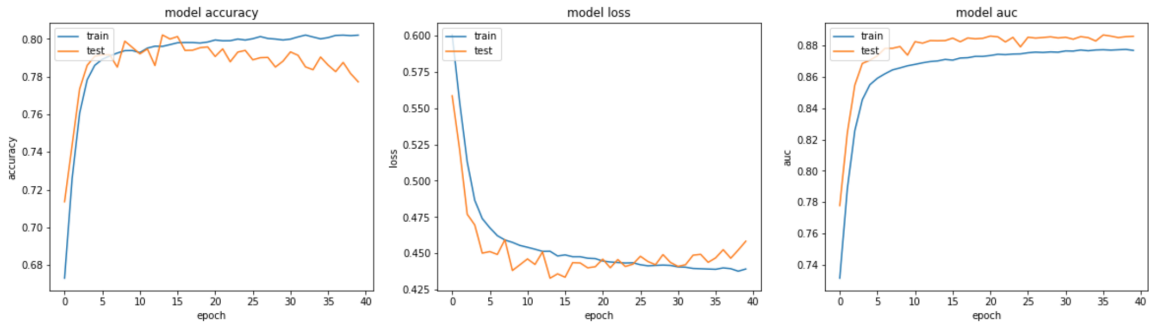


Figure 11: Fonction relu : 30 - 60 - 60 - 60 - 60 - 60 - 60 - 30 - 1 overfitting avéré !

| | Sigmoid_30_30_1 | Sigmoid_30_60x2_30_1 | Sigmoid_30_60x2_60x2_30_1 | Relu_30_30_1 | **Relu_30_60x2_30_1** | Relu_30_60x2_60x2_30_1 | Relu_30_60x2_60x2_60x2_30_1 |
|---|---|---|---|---|---|---|---|
| Auc | 0,861 | 0,865 | 0,859 | 0,863 | 0,888 | 0,876 | 0,884 |
| Loss | 0,466 | 0,461 | 0,458 | 0,457 | 0,421 | 0,426 | 0,459 |
| Accuracy | 0,779 | 0,781 | 0,789 | 0,787 | 0,809 | 0,802 | 0,778 |

Figure 12: Tableau contenant les résultats des architectures testées

### 3.5 Métrique du challenge

La métrique d'évaluation du challenge correspond à l'*approximate median significance* (AMS) :

AMS $= \sqrt{2\left((s+b+b_r)\log\left(1+\frac{s}{s+b_r}\right)-s\right)}$ avec

- $s, b$ : taux de vrais positifs et de faux négatifs respectivement
- $br = 10$ est la constante de régularisation,

$s = \sum_{i=1}^{n} w_i 1\{y_i = \mathrm{s}\}1\{\hat{y}_i = \mathrm{s}\}$

$b = \sum_{i=1}^{n} w_i 1\{y_i = \mathrm{b}\}1\{\hat{y}_i = \mathrm{s}\}$

où la fonction indicatrice $1\{\mathrm{A}\}$ vaut 1 si son argument A est vrai 0 sinon

Plus précisément $(y_1, ..., y_n) \in \{b, s\}^n$ correspond au vecteur des vraies étiquettes.
$(\hat{y}_1, ..., \hat{y}_n) \in \{b, s\}^n$ sera le vecteur des étiquettes prédites.
$(w_1, ..., w_n) \in \mathbb{R}^{+n}$ sera le vecteur des poids (importance) de chaque ligne.

Il nous a été difficile d'implémenter cette métrique dans nos classifieurs. Nous avons tenter notamment d'utiliser la fonction *make_scorer* de *Scikit-learn* pour créer cette métrique. Celle-ci fonctionne bien mais le problème est de l'utiliser en ajoutant les poids necessaires à son calcul, ce qui est plus complexe et que nous n'avons pas réussi à faire.

## 4 Conclusion

### 4.1 discussion / critiques

La possibilité de pouvoir jouer librement et experimenter sur les données a créé des moments de réflexion intense très appréciables. Après quelques temps d'essais plus ou moins fructueux, nous avons pleinement compris le fonctionnement de l'outil *pipeline* et *GridSearch* qui sont finalement extrêmement utiles pour la construction et l'évaluation de nos modèles.

Malheureusement nous n'avons pas eu le temps de vraiment mettre en oeuvre les étapes du *features engineering*, c.a.d. tester de façon plus détailée l'influence de la transformation des variables, comme le logarithme ou des fonctions inverses, sur le modèle ou encore la création de nouvelles variables par la combinaison des variables existantes.

La découverte de la bibliothèque *TPOT* a été aussi une expérience très enrichissante. C'est en outil extrêmement pratique qui fait gagner un temps non négligable au data scientist en lui permettant de trouver les bons hyperparamètres rapidement. Il faut cependant beaucoup de puissance de calcul : nous avons du restreindre notre recherche sur *TPOT* pour finalement être capable de trouver un bon modèle.

Nous regrettons aussi de ne pas avoir pu utiliser les poids ainsi que l'AMS, pour obtenir un score correct dans le classement du concours Kaggle. C'est ce problème de poids qui nous fait prendre conscience des limites du haut niveau de l'API *Keras*.

Un dernier point, et non des moindres, consisterait à investir très rapidement dans un ordinateur possédant au moins une carte graphique digne de ce nom. Pour ce projet nous avons en effet utilisé un ordinateur i5 dual core, 2.53GHz, 8Go RAM, (carte graphique NVIDIA GeForce GT 220 non utilisée car obsolète pour *Tensorflow*).