REDS - Projet Boson de Higgs

Rapport

Kim-Anh Laura Nguyen
Keyvan Beroukhim
Master 2 DAC
Promo 2019-2020

Enseignant : Olivier Schwander

1 Introduction

Nous souhaitons détecter la présence du boson de Higgs dans des données simulées dans le but de reproduire le comportement de l'expérience ATLAS. Il s'agit d'un problème de détection d'évènement, ou classification binaire, dans lequel les deux classes sont :

— *background*

— *tau tau decay of a Higgs boson*

Les données sont issues du projet Kaggle *ATLAS Higgs Boson Machine Learning Challenge 2014*.

2 Analyse préliminaire des données

La base de données est constituée de 818238 évènements simulés. Chaque évènement est défini par 30 attributs numériques et un label à prédire.

Les données sont composées à **34% de labels positifs** (les *signaux*) et à 66% de labels négatifs (le *background*). Les classes ne sont pas suffisamment déséquilibrées pour gêner l'entraînement. Par ailleurs, la métrique propre à cette tâche, nommée *AMS*, nous est imposée. Le choix habituel de la mesure d'évaluation à utiliser pour prendre en compte ce déséquilibre ne se pose donc pas.

2.1 Distribution des variables

La figure 1 contient les histogrammes de répartition des valeurs de quelques variables, en omettant les valeurs manquantes. Nous constatons que **toutes les variables ne suivent pas le même type de distribution**. Or, les algorithmes d'apprentissage se comportent généralement mieux avec une distribution des données équilibrée. Dans ce dataset, de nombreuses variables ont une distribution exponentielle décroissante (e.g `DER_MASS_VIS` sur la figure 1).

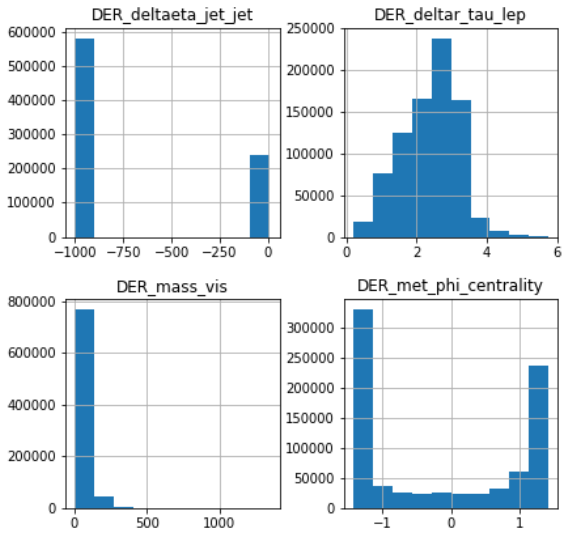
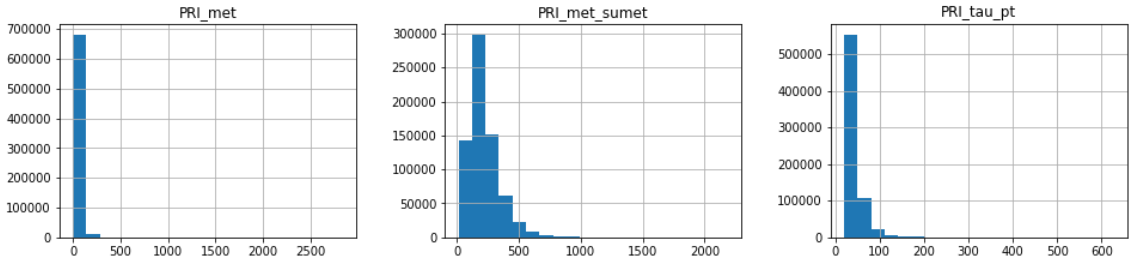
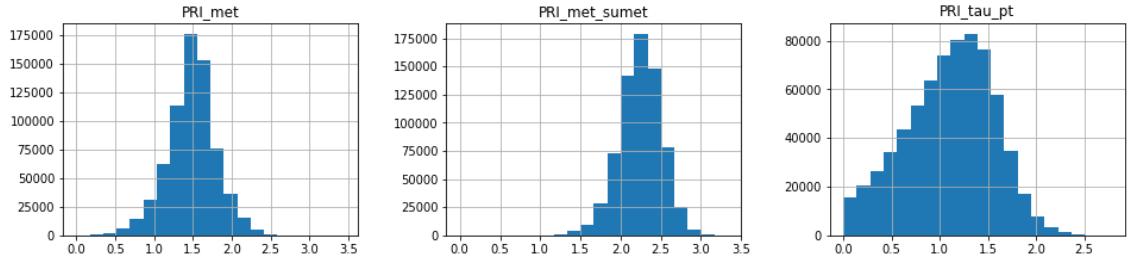


FIGURE 1 – Histogrammes de répartition des valeurs de quelques variables

Afin d'obtenir des **distributions normales** et de **stabiliser la variance**, nous effectuons une **log-transformation** de manière indépendante sur les colonnes. Nous commençons par ajouter une constante à chaque colonne à transformer pour que la valeur minimale de sur cette colonne soit 1 (comme le log ne prend que des valeurs strictement positives). Nous appliquons ensuite la fonction log sur les colonnes. Avec ce dataset et en utilisant cette méthode, nous obtenons toujours des distribution normales. Finalement nous **centrons et réduisons** chaque variable de manière indépendante car cela facilite généralement l'apprentissage des modèles.



(a) Avant log-transformation



(b) Après log-transformation

FIGURE 2 – Histogrammes de répartition des valeurs de quelques variables avant et après log-transformation

2.2 Valeurs manquantes

Le dataset contient **21% de valeurs manquantes** (indiquées par la valeur -999). D'une manière générale, les modèles d'apprentissage statistique ne gèrent pas automatiquement les valeurs manquantes. Il faut donc traiter ces données avant de les présenter à nos modèles. Nous considérons trois façons de pallier à ce problème.

2.2.1 Omission

En supprimant les évènements dont au moins un attribut n'est pas valide, nous retrouvons avec seulement 223574 exemples d'apprentissage : **restreindre le dataset aux données complètes fait perdre 75% des évènements**. Par ailleurs, la base restreinte contient 47% de labels positifs, soit 13% de plus que le dataset original. Cela met en évidence le fait que les données ne sont pas manquantes de manière indépendante : l'absence des données est liée à leur valeur (nous sommes dans un contexte *Missing Not At Random*). Un modèle entraîné sur le dataset restreint en faisant l'hypothèse "i.i.d." serait donc biaisé.

2.2.2 Omission d'attributs

Une autre méthode consiste à **supprimer les features dont le pourcentage de valeurs manquantes dépasse un certain seuil**. On **retire ensuite les évènements à valeurs manquantes**. Le dataset final ne contient donc plus aucune donnée manquante. En fixant le seuil à 40%, notre jeu de données passe de 818238 à 693636 échantillons, soit 85% des données initiales, et de 35 à 20 attributs.

2.2.3 Conservation

Les ***Random Forest*** font partie des algorithmes permettant de **gérer les données manquantes**. En effet, les arbres de décision peuvent **reconnaître une donnée manquante par un simple test de valeur**.

2.2.4 Affectation (*Imputing*)

Une autre méthode consiste à **compléter les données manquantes** de la manière la plus "intelligente" possible. Une fois les données complétées, cela permet d'**utiliser tous les modèles de classification à notre disposition**.

Nous nous servons de l'algorithme `IterativeImputing` de scikit-learn, qui complète de manière itérative les données manquantes en utilisant des modèles de régression prédisant la valeur d'une variable à partir de la valeur des autres variables.

En pratique et comme nous souhaitons conserver le maximum d'information, **nous ne supprimerons ni d'évènements ni d'attributs à valeurs manquantes**. Nous testerons, d'un côté, des **algorithmes permettant de gérer les données manquantes** et, de l'autre, la méthode de **complétion des données**. Pour cette dernière, nous appliquons d'abord les **log-transformations** puis nous réalisons l'**imputing** avant de finir par **scaler** les données. Ce choix se justifie premièrement par l'intuition que l'imputing fonctionnera mieux sur des données log-transformées et, deuxièmement, par le fait qu'estimer la moyenne et la variance en ignorant les données manquantes serait biaisé.

2.3 Sélection de features

Les données sont représentées par **30 variables explicatives**. Ce nombre n'étant **pas très élevé** (par exemple comparé au nombre de pixels dans une image), il n'y a donc **pas de grand risque de sur-apprentissage** pour les modèles. Ainsi, la sélection de caractéristique ne semble pas nécessaire. Cependant, comme le montre la figure 3, certaines variables sont **fortement corrélées** (e.g. `DER_mass_MMC` et `DER_mass_transverse_met_lep`), on comparera donc les performances avec ou sans l'utilisation d'une ***Analyse en Composantes Principales* (PCA)** afin de réduire le nombre de données explicatives.

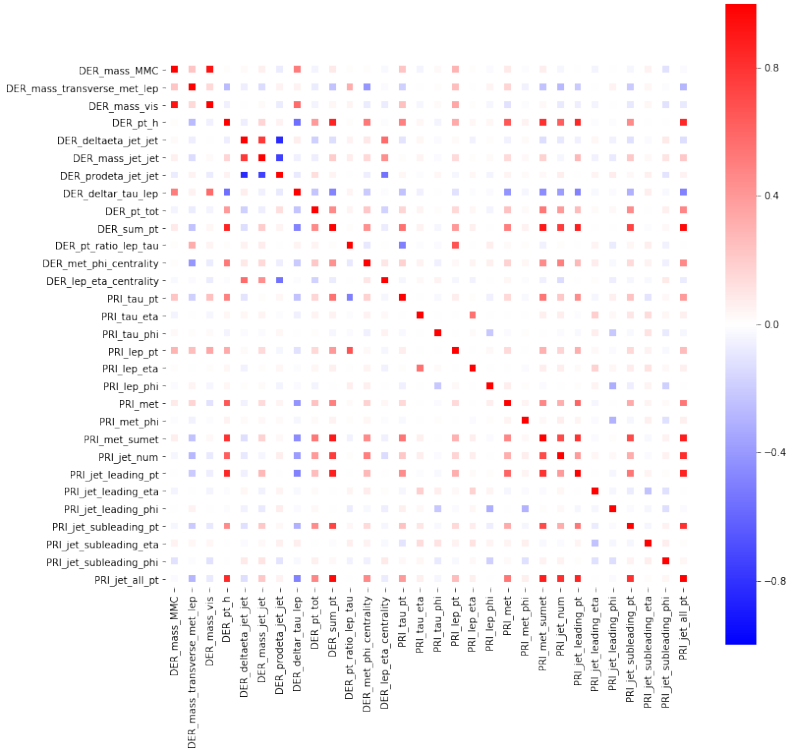


FIGURE 3 – Matrice de corrélation entre chaque attribut

3 Classification des évènements

3.1 Modèles utilisés

Nous considérons plusieurs modèles d'apprentissage pour classifier les évènements :

— SVM

— méthodes ensemblistes : Bagging de Perceptron, Random Forest, AdaBoost

3.1.1 Support Vector Machines

Les *Support Vector Machines* (SVMs) nous fournissent une ***baseline***. Généralisation des classifieurs linéaires, ils nécessitent un **faible nombre d'hyperpararamètres**, ont des **garanties théoriques** et donnent de **bons résultats** en pratique.

3.1.2 Méthodes d'ensemble

Afin de **réduire de réduire le biais et la variance** de nos modèles et ainsi améliorer les performances, nous utilisons des **méthodes d'ensemble**. Ces algorithmes consistent à **combiner plusieurs classifieurs faibles** pour obtenir un modèle **plus stable, plus robuste et ayant une meilleure capacité de généralisation**.

Bootstrap AGGregatING (Bagging). Le *Bootstrap Aggregating* est une méthode pour **réduire la variance par moyennage**. Plusieurs ensembles d'apprentissage sont simulés par ***bootstrap* (tirage avec remise)** puis **sur chaque sous ensemble, on entraîne un classifieur faible**. Dans notre contexte de classification binaire, la prédiction du modèle est définie par le **vote majoritaire des modèles simples (*aggregating*)** .

Comme chaque classifieur faible est entraîné sur des données légèrement différentes, le modèle d'ensemble est capable de **capturer de petites variations dans les données** et donc de mieux **généraliser**.

Nous choisissons, uniquement pour le Bagging, le **Perceptron** comme classifieur faible. Il s'agit d'un **modèle instable**, ce qui permet d'obtenir un **ensemble de classifieurs suffisamment différents**.

Random Forest. Avec la technique du Bagging, même si chaque classifieur est appris sur un jeu de données légèrement différent, tous les modèles se servent des mêmes attributs pour partitionner les données, les arbres générés risquent donc d'être **trop similaires**. Pour pallier à ce problème, les arbres des **forêts aléatoires (*random forests*)**, également appris sur des bootstrap de l'ensemble original, **sélectionnent pour chacun de leurs noeuds un échantillon aléatoire de features**, ce qui **force les modèles à être différents** et empêche le modèle global de se concentrer sur des attributs particuliers.

AdaBoost. Avec la technique de Bagging, les classifieurs faibles sont appris indépendamment les uns des autres. Or, si ce classifieur est, à lui seul, trop faible, **le Bagging ne permettra pas d'obtenir un meilleur biais**. Le ***Boosting*** permet de traiter ce problème par **apprentissage successif de classifieurs faibles** : en donnant plus de poids aux exemples mal classés par les modèles précédents, **chaque nouveau classifieur se focalise sur les parties de l'espace mal prédites**. Contrairement au Bagging, la contribution d'un classifieur à la prédiction du modèle d'ensemble est déterminée par sa performance.

En revanche, si le classifieur faible est trop complexe, le modèle de Boosting **risque de sur-apprendre**, auquel cas la technique de Bagging est plus adaptée.

3.2 Protocole d'expérimentation

Tout d'abord, nous commençons par **séparer le jeu de données en train et en test**. Avant d'entraîner nos modèles, nous apprenons les **paramètres de prétraitement des données** sur l'ensemble d'apprentissage, et nous appliquons les prétraitements sur l'ensemble des données. Les prétraitements et apprentissages des paramètres sont effectués dans l'ordre décrit précédemment : **log-transformation** (valeurs minimales des colonnes apprises), **imputing** (paramètres de tous les régresseurs appris) et **scaling** (moyennes et écarts-types des colonnes appris).

Pour chaque algorithme, nous procédons par ***grid search* pour trouver les hyperparamètres optimaux**. Pour chaque combinaison d'hyperparamètres, la performance du modèle produit est estimée par ***cross-validation***p sur l'ensemble d'apprentissage.

Pour chaque modèle, nous récupérons les hyperparamètres ayant produit le meilleur résultat et **calculons le score final par cross-validation sur les données de test**.

Par souci de temps de calcul, les expériences de recherche de paramètres optimaux et d'évaluation des modèles ne sont pas menées sur le dataset entier. Pour les **métriques usuelles**, **le score obtenu par un modèle est indépendant de la taille du jeu de test**, mais **ce n'est pas le cas pour la métrique AMS**. Afin d'obtenir des scores en AMS comparables, nous fixons la taille des jeux de tests à 1000 éléments.

Les scores obtenus en AMS étant difficilement interprétables, nous utilisons aussi comme métrique le **taux de bonne classification (*accuracy*)**. Pour chaque métrique, les scores présentés par la suite sont obtenus en optimisant les modèles pour cette métrique.

3.3 Résultats et analyse

	C	Noyau
Accuracy	3.16	rbf
AMS	1.78	rbf

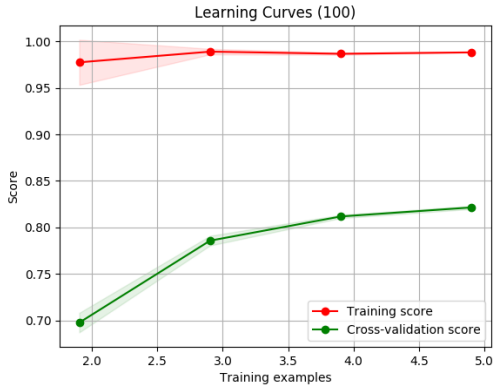
TABLEAU 1 – Paramètres optimaux pour le SVM (accuracy, $|X_{train}| = 1000$)

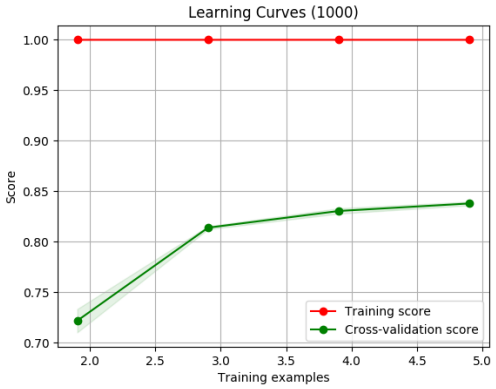
	Bagging		RF		RFNan		AdaBoost	
	Accuracy	AMS	Accuracy	AMS	Accuracy	AMS	Accuracy	AMS
Nombre d'estimateurs	500	2000	500	1000	500	500	1000	50
Profondeur max			50	50	∞	∞		

TABLEAU 2 – Paramètres optimaux pour les méthodes d'ensemble (accuracy, $|X_{train}| = 1000$)

	% Accuracy		AMS (10^{-2})	
	Apprentissage	Test	Apprentissage	Test
SVM	75.20 ± 1.55	75.30 ± 0.42	4.14 ± 0.49	3.01 ± 0.23
Bagging	73.30 ± 1.24	71.80 ± 0.71	2.90 ± 0.29	2.33 ± 0.26
RF	80.20 ± 0.50	80.70 ± 0.62	5.10 ± 0.93	3.79 ± 0.28
RFNan	81.70 ± 0.17	81.40 ± 0.82	5.14 ± 1.00	4.00 ± 0.37
AdaBoost	74.30 ± 0.79	74.50 ± 0.58	4.03 ± 0.35	3.41 ± 0.07

TABLEAU 3 – Performances obtenues avec chaque méthode $|X_{test}| = 1000$)





Courbes à commenter

Tableaux, graphiques, courbe ROC