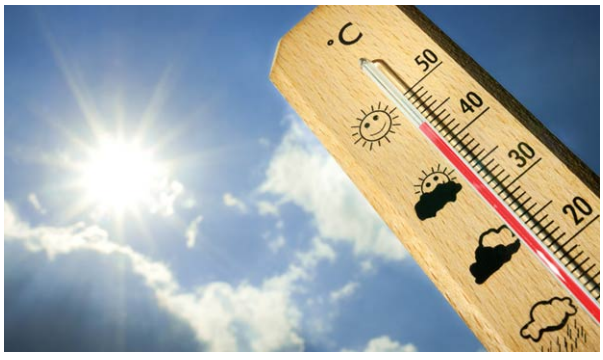
UNIVERSITÉ RENNES 2

PROJET DATA SCIENCE - MASTER 2 MAS PARCOURS SCIENCES DES DONNEES

Rapport de l'étude

Prédictions des températures de 7 stations différentes à un horizon de 36 heures

Défi Grosses Data 2018



Team : Freak-unit
Membres :

- Benjamin ALLEAU
- Abdessamad AZNAGUE
- Guillaume LE FLOCH

Organisateur : INSA de Toulouse
Encadrant : M. Romain TAVENARD

05 Octobre 2017 — 08 Janvier 2018

Plan

1 Présentation du projet

La compétition 2017-2018 est basée sur une stratégie d'adaptation statistique pour améliorer la prévision de températures à un horizon de 36 heures.

Les services du **CNRM** (Centre National de Recherches Météorologiques) construisent des grands modèles déterministes de l'atmosphère par résolution des équations de Navier et Stockes sur un maillage : 10km pour **ARPEGE**, kilométrique pour **AROME** mais limité à l'Europe. Il apparaît que ces modèles sont généralement biaisés, notamment parce qu'ils ne peuvent prendre en compte des phénomènes à petite échelle. Par exemple, un vent important (e.g. vent d'autan à Toulouse) provoque des turbulences qui, en mélangeant les couches d'air, entrainent une baisse de la température par rapport à celle prévue.

Un modèle statistique intégrant les prévisions du modèle déterministe peut contribuer à réduire significativement ces biais. En revanche, un modèle statistique seul est incapable de prévoir l'arrivée d'une perturbation océanique à partir de données locales. L'adaptation statistique est donc une mise en collaboration « optimale » des deux approches de modélisation, déterministe et statistique.

Le « Défi Grosses Data » est donc un challenge mettant en confrontation une multitude de groupes d'étudiants provenant des **INSA** de Toulouse et Rennes, ainsi que des Universités de **Bordeaux**, **Rennes 1**, **Rennes 2**, **Paris Descartes**, **Paul Sabatier (Toulouse 3)** ou encore la **Toulouse School of Economics**. Ce rapport détaillera le travail fourni par l'équipe **Freak-unit**, une des 8 équipes représentant l'**Université de Rennes 2**.

L'objectif de ce projet est simple : prédire au mieux les températures de 7 stations sur un horizon de 36 heures. Pour ce faire, 36 fichiers $train\_H.csv$ $(H = 1, .., 36)$, ont été mis à notre disposition. Ces fichiers contiennent les variables issues des modèles physiques décrits dans le préambule à un horizon de +H heures. Leur contenu est le suivant :

- les lignes correspondent à chaque jour sur une période allant du 1^{er} janvier 2014 au 30 mai 2016.
- les colonnes contiennent 30 variables (température, nébulosité, vent, humidité ..)

Nous allons détailler ces variables afin de bien comprendre quelles sont les mesures à notre disposition :

- **insee** : Facteur à 7 niveaux représentant le numéro insee des stations pour lesquelles nous cherchons à prédire les températures (Nice, Toulouse Blagnac, Bordeaux-Mérignac, Rennes, Lille Lesquin, Strasbourg Entzheim, Paris-Montsouris)
- **tH2_obs** : Observation de la température à 2 mètres *in situ* au point station (prédictant). **C'est la variable réponse que l'on cherche à prédire**
- **ech** : Facteur à 36 niveaux correspondant à l'échéance de validité (H)
- **capeinsSOL0** : Energie potentielle convective
- **ciwcH20** : Fraction de glace nuageuse à 20 mètres
- **clwcH20** : Fraction d'eau nuageuse à 20 mètres
- **nH20** : Fraction nuageuse à 20 mètres
- **pMER0** : Pression au niveau de la mer
- **rr1SOL0** : Précipitation horaire au niveau du sol
- **rrH20** : Précipitation horaire à 20 mètres
- **tpwHPA850** : Température potentielle au niveau 850 hPa
- **ux1H10** : Rafale 1 minute du vent à 10 mètres composante zonale
- **vapcSOL0** : Colonne de vapeur d'eau
- **vx1H10** : Rafale 1 minute du vent à 10 mètres composante verticale
- **ddH10_rose4** : Facteur à 4 niveaux indiquant la direction du vent à 10 mètres (en rose4)
- **ffH10** : Force du vent à 10 mètres en m/s
- **flir1SOL0** : Flux Infra-rouge en J/m^2
- **fllat1SOL0** : Flux de chaleur latente en J/m^2

- **flsen1SOL0** : Flux de chaleur sensible en J/m^2
- **flvis1SOL0** : Flux visible en J/m^2
- **hcoulimSOL0** : Hauteur de la couche limite en mètres
- **huH2** : Humidité 2mètres en %
- **iwcSOL0** : Réservoir neige kg/m^2 (équivalent en eau liquide des chutes de neige
- **nbSOL0_HMoy** : Nébulosité basse (moyenne sur les 6 points de grille autour de la station) (fraction en octat du ciel occulté)
- **ntSOL0_HMoy** : Nébulosité totale (moyenne sur les 6 points de grille autour de la station)
- **tH2** : Température à 2 mètres du modèle **AROME**
- **tH2_VGrad_2.100** : Gradient vertical de température entre 2 mètres et 100 mètres
- **tH2_XGrad** : Gradient zonal de température à 2 mètres
- **tH2_YGrad** : Gradient méridien de température à 2 mètres
- **mois** : Facteur à 12 niveaux représentant le mois

Il est à noter que ces 36 fichiers *train* contiennent des valeurs manquantes, le sujet sera approfondi un peu plus tard dans l'étude.

L'objectif de ce projet est le suivant : à partir des différents fichiers *train* nous devons réaliser un apprentissage (supervisé dans notre cas) afin de pouvoir appliquer par la suite notre modèle au fichier *test.csv* qui contient les mêmes variables, à l'exception de **tH2_obs** évidemment puisqu'il faut la prédire. Les lignes du fichier *test.csv* contiennent les 6 périodes de 15 jours suivantes :

- du 20/06/2016 au 02/07/2016
- du 01/08/2016 au 14/08/2016
- du 12/09/2016 au 25/09/2016
- du 24/10/2016 au 06/11/2016
- du 05/12/2016 au 18/12/2016
- du 08/05/2017 au 21/05/2017

Une fois les prédictions effectuées, il faut soumettre les résultats sous forme d'un fichier au format *csv* toujours, sur le site internet du challenge. Au bout d'exactement 1 heure après la soumission, nous pouvons visualiser notre score. La métrique utilisée pour évaluer le score est le **RMSE** (Root Mean Square Error). Si l'on note $\hat{y}$ notre vecteur de prédictions et y_{obs} les valeurs des températures réellement observées, le **RMSE** associé se note :

$$RMSE_{\hat{y}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_{obs_i})^2}$$

En d'autres termes, cette métrique permet de mesurer d'une certaine façon l'écart entre nos prédictions et la réalité. Il est à noter que le challenge fournit un score de 1.30755 appelé **Baseline** qui correspond à un score de référence, associé à « un modèle élémentaire de prévision sans effort particulier d'optimisation. C'est un objectif a minima à améliorer dans ce concours ». Le décor est planté, désormais il ne reste plus qu'à se lancer dans le challenge.

2 Remerciements

Avant de détailler les différentes étapes de ce projet, les 3 membres de la **Freak-unit** team souhaitent remercier tout particulièrement leur professeur encadrant, **M. Romain TAVENARD** pour son accompagnement tout au long du projet et pour l'aide apportée. Sa connaissance du *Deep Learning* aura notamment été un facteur déterminant dans la réalisation de ce projet.

3 Différentes stratégies envisagées

Afin d'effectuer l'analyse la plus fine et pertinente possible, il est primordial de bien définir la stratégie d'étude à adopter. Par « stratégie », nous parlons ici de la façon dont nous allons structurer les données et des techniques que nous allons employer pour optimiser la performance des algorithmes. Cela peut être vu comme un problème d'optimisation sous contrainte : notre objectif est de réaliser la meilleure performance sous des contraintes de temps et de matériel. Il faut donc réussir à combiner au mieux la théorie mathématique avec la réalisation informatique.

3.1 Structuration des données

Un premier travail a consisté à agréger les données des 36 fichiers $train\_H.csv$ $(H = 1, .., 36)$ dans un seul jeu de données : cela aboutit à une matrice de 189281 lignes et 31 colonnes. Dans un deuxième temps, nous avons divisé ce fichier en 7 sous-fichiers : un fichier regroupant les prédictions aux 36 échéances de chaque station. Ce choix a été guidé par 3 éléments :

- L'intuition : sachant qu'il existe plusieurs types de climats (océaniques, continental, méditerranéen) et qu'on les retrouve à travers les stations présentes dans les données, il ne paraît pas anormal d'essayer de prédire chaque station indépendamment des autres. Ne perdons pas de vue que les chiffres sont là pour quantifier la réalité, l'objectif est donc de les utiliser pour représenter au mieux les phénomènes réels.
- Un indice lâché par les organisateurs du concours qui ont laissé entendre qu'il pouvait exister un biais lié à chaque station.
- L'analyse du comportement des indicateurs par station.

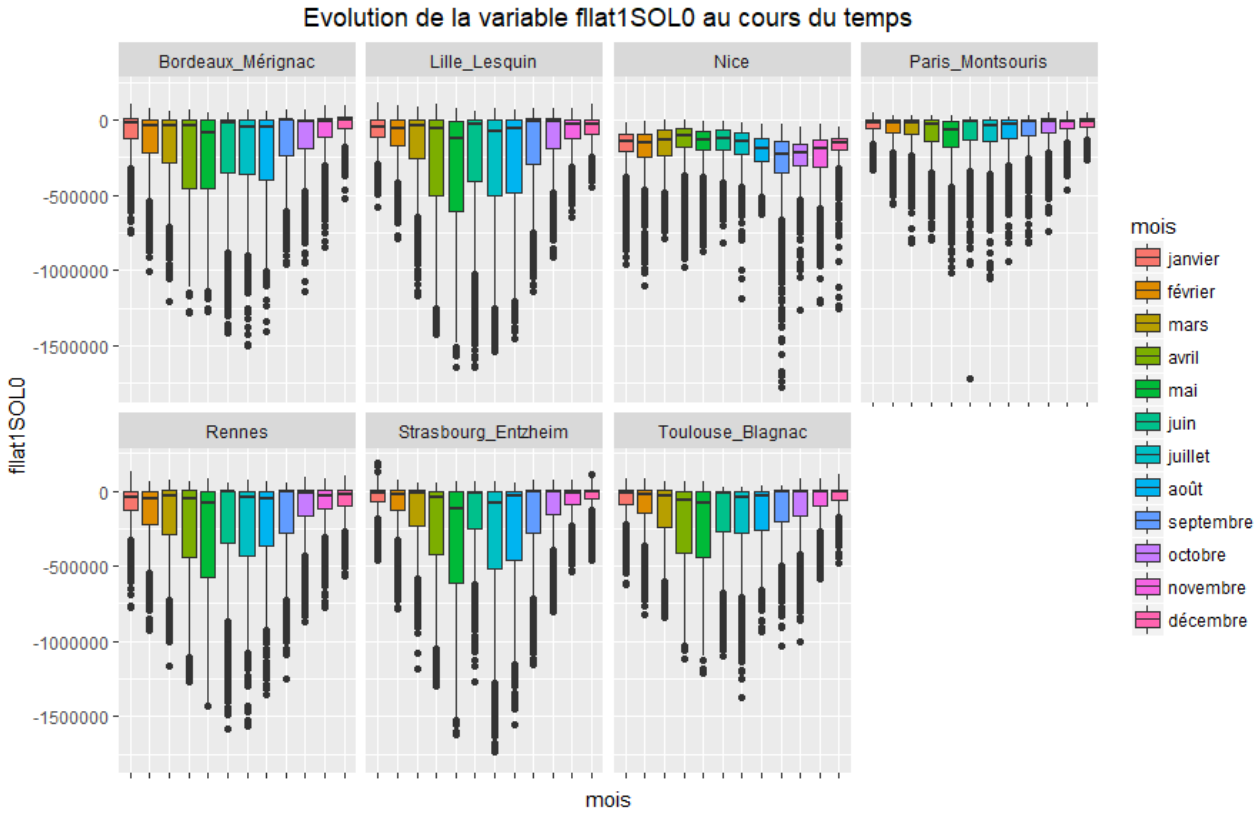


FIGURE 1 – Exemple du flux de chaleur latente dont le comportement diffère d'une station à une autre

Dans la suite, nous verrons que le gros jeu de données (189281 lignes) ou les fichiers par stations seront utilisés en fonction de l'algorithme choisi. Une autre option aurait pu être de découper les données par station et par échéance. Cependant en procédant de la sorte, on se retrouverait avec 252 modèles à estimer, et l'on s'exposerait également au risque de sur-apprentissage (*overfitting* en anglais). La dimension des données va également impacter le choix de la méthode d'estimation des paramètres et d'estimation de la performance de l'algorithme : la **validation croisée**. Nous allons donc effectuer quelques rappels sur ces différents concepts.

3.2 Notions d'overfitting/underfitting

Comme introduit brièvement un peu plus haut dans ce rapport, l'un des enjeux de ce projet dans le but d'obtenir les meilleures prédictions possibles, selon le critère du **RMSE**, et donc d'éviter de se retrouver en situation sur-apprentissage ou de sous-apprentissage (respectivement *overfitting* et *underfitting*). En ce sens, on va chercher à construire un estimateur (une fonction) optimal, robuste qui conservera un pouvoir de prédiction de qualité sur un nouvel échantillon. Prenons le dessin suivant qui sera un peu plus parlant.

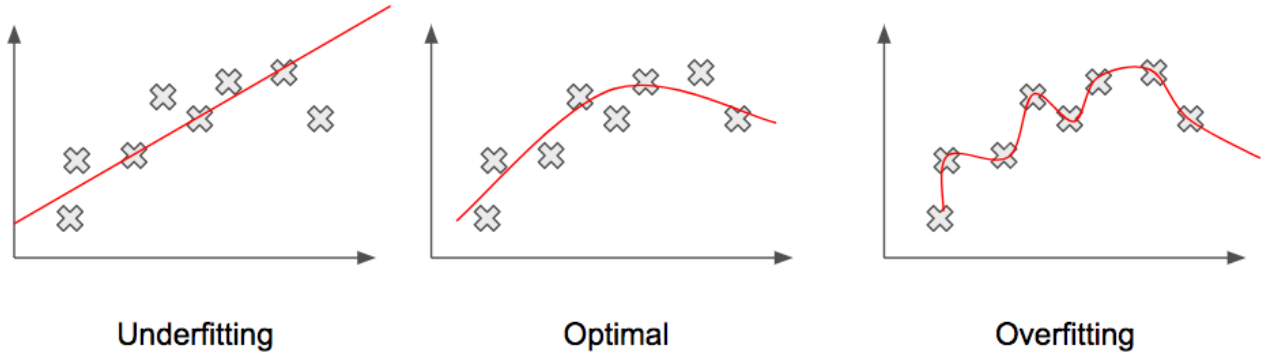


FIGURE 2 – Illustration de ces notions à partir d'un exemple simple dans $\mathbb{R}^2$

On s'aperçoit que sur le graphe de gauche, l'estimateur est trop rigide, il ne « colle » pas assez aux données. Sur le graphe de droite c'est le contraire, l'estimateur est beaucoup trop sensible aux valeurs, et lorsque les données changeront, les prédictions ne seront donc plus aussi précises. Nous souhaitons donc trouver une fonction ayant une allure similaire à celle du graphe du milieu, qui sera plus « robuste » que les deux autres.

Nous verrons dans la suite que pour remédier au problème de l'overfitting, nous pouvons utiliser des formes de **régularisation**, encore appelées pénalités et qui se traduiront par des contraintes dans le problème d'optimisation d'une certaine **fonction de perte** que l'on va choisir.

Une façon de vérifier que le modèle est robuste peut-être de procéder à de la **validation croisée**, c'est donc le prochain point que nous allons développer.

3.3 Validation croisée

Comme énoncé précédemment, nous allons utiliser des méthodes de validation croisée à la fois pour contrôler la performance et la robustesse de l'algorithme mis en œuvre, mais également pour calibrer des hyper-paramètres dans le cas de modèles complexes. Nous allons donc dans la suite effectuer quelques rappels théoriques afin de bien comprendre pourquoi nous utilisons ces méthodes ainsi qu'une présentation des deux types de validation croisée que nous avons choisi d'employer.

3.3.1 Contexte théorique

Soit d_1^n l'observation d'un n-échantillon $D_1^n = \{(X_1, Y_1), ..., (X_n, Y_n)\}$ d'une loi conjointe P sur $\mathcal{X}$ x $\mathcal{Y}$ qui est inconnue. On cherche à estimer cette loi P à l'aide des données disponibles, à savoir d_1^n. En théorie, il faudrait minimiser la quantité appelée **risque** (ou encore **erreur de généralisation**) d'une règle de prédiction que l'on va noter f. Ainsi de façon mathématique, on définit le **risque** de la manière suivante :

$\mathcal{R}_P(f) = \mathbb{E}_{(X,Y)}[l(Y, f(X))]$ avec l que l'on appelle **fonction de perte** et qui a pour expression :

- $l(y, y') = |y - y'|^q$ en régression réelle (on prendra souvent la perte *quadratique*, c'est-à-dire lorsque $q = 2$)
- $l(y, y') = \mathbb{1}_{y \neq y'}$ en discrimination binaire.

Cependant en pratique, il nous est impossible de calculer le **risque**, P étant inconnue. Il faut donc l'estimer et chercher à minimiser cette quantité estimée par la suite. Une première approche naturelle serait d'utiliser le risque empirique, c'est-à-dire d'estimer l'espérance de la fonction de perte par la moyenne empirique de cette même fonction. Le **risque empirique** d'un algorithme de prédiction $f_{D_1^n}$ construit sur D_1^n se note de la façon suivante :

$$\widehat{\mathcal{R}}_n(f_{D_1^n}) = \frac{1}{n}\sum_{i=1}^{n} l(Y_i, f_{D_1^n}(X_i))$$

Le problème de cet estimateur est l'absence d'indépendance : en effet, pour effectuer ses prédictions il se sert des mêmes données qui ont servi à faire l'algorithme d'apprentissage. En ce sens, notre algorithme risque de trop « coller » aux données d'apprentissage, puisqu'il aura acquis l'information à partir de ces données il sera capable de bien prédire la variable réponse, mais risque de ne pas être aussi performant à l'avenir sur un nouveau jeu de données. C'est ce qu'on nous avons défini comme étant l'*overfitting*. C'est donc pour corriger ce problème que nous allons avoir recours à d'autres méthodes d'estimation du **risque**, qui sont meilleures : les méthodes de **validation croisée**.

3.3.2 La validation croisée Hold-Out

Le concept de base pour la validation croisée **Hold-Out** consiste à séparer nos données d_1^n en deux sous-échantillons :

- $\mathcal{A}$: l'échantillon d'apprentissage
- $\mathcal{V}$: l'échantillon de validation

Cela implique d'avoir à la base un jeu de données suffisamment « grand » pour pouvoir appliquer une telle méthode, ce qui n'est pas un problème dans notre cas. Comment séparer les données ? En pratique on va diviser notre échantillon en deux parties égales, si l'on dispose d'un échantillon de taille n, $\mathcal{A}$ et $\mathcal{V}$ seront donc de taille $\frac{n}{2}$. Si les données sont classées, on procèdera à un tirage aléatoire sans remise de $\frac{n}{2}$ éléments dans $\{1, ..., n\}$. Ensuite on « nourrit » notre algorithme à partir des données contenues dans $\mathcal{A}$ et on teste sa performance sur les données de $\mathcal{V}$: ainsi le problème d'indépendance est réglé. Illustrons cette méthode avec un schéma.

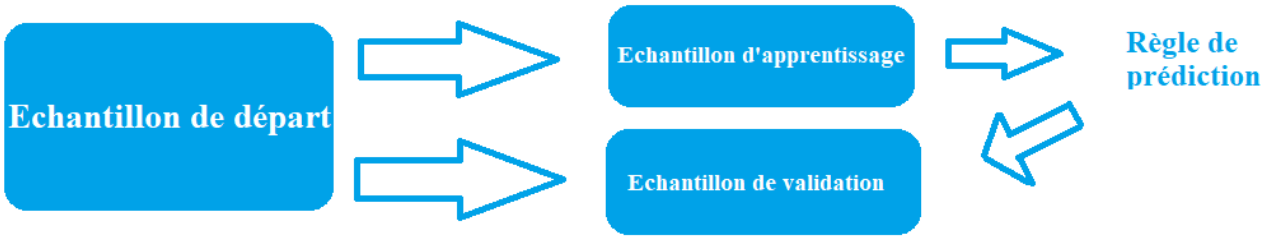


FIGURE 3 – Fonctionnement de la validation croisée Hold-Out

De façon mathématique, cet estimateur du risque se définit de la façon suivante :

$$\widehat{\mathcal{R}}(f_{D_1^n}) = \frac{1}{|\mathcal{V}|}\sum_{(x_i,y_i)\in\mathcal{V}} l(y_i, f_{\mathcal{A}}(x_i))$$

avec $|\mathcal{V}|$ le cardinal de $\mathcal{V}$ (autrement dit, le nombre de lignes de l'échantillon de validation) et $f_{\mathcal{A}}(x_i)$ les prédictions de l'algorithme f sur les observations $x_i \in \mathcal{A}$, l'échantillon d'apprentissage.

Dans le cadre de nos travaux, l'utilisation de cette méthode va nous permettre de gagner du temps si l'on utilise la matrice de travail contenant toutes les stations. La méthode suivante que nous allons présenter prendra plus de temps mais peut apporter plus de stabilité dans les résultats.

3.3.3 La validation croisée K-Fold

La validation croisée **K-Fold** est une variante de la méthode vue précédemment, qui on le rappelle était adaptée aux jeux de données de taille suffisamment grande. Cependant sur des jeux de données plus petits, comme c'est le cas de nos fichiers par station qui font environ 27000 lignes chacun,ce type de validation croisée peut également s'avérer efficace. Le fonctionnement de cette méthode itérative est le suivant :

- On procède à une partition de notre échantillon en K blocs équilibrés : $B_1, ..., B_K$ auxquels sont associés des tailles $n_1, ..., n_K$. (En pratique on prend $K = 10$ quand la taille des données le permet, sinon on se contente de 5 blocs)
- On note $d_{B_i} = \{(x_j, y_j), j \in B_i\}$ le i^{eme} bloc de l'échantillon d_1^n. Pour chaque itération $i = 1, ..., K$ on laisse de côté les données de d_{B_i} pour faire notre apprentissage sur les K-1 blocs restants
- On teste la performance de notre règle de prédiction sur le bloc B_i pour obtenir une estimation du risque à l'itération i que l'on note :

$$\widehat{\mathcal{R}}_i = \frac{1}{n_i} \sum_{j \in B_i} l(y_j, f_{d_1^n \setminus d_{B_i}}(x_j))$$

  avec $f_{d_1^n \setminus d_{B_i}}(x_j)$ l'algorithme de prédiction f appliqué aux observations x_j telles que x_j n'appartiennent pas à d_{B_i}
- Enfin, après avoir procédé à toutes les itérations, on calcule la véritable estimation du risque en faisant la moyenne des estimations du risque calculées à chaque itération :

$$\widehat{\mathcal{R}}(f_{D_1^n}) = \frac{1}{K} \sum_{i=1}^{K} \widehat{\mathcal{R}}_i$$

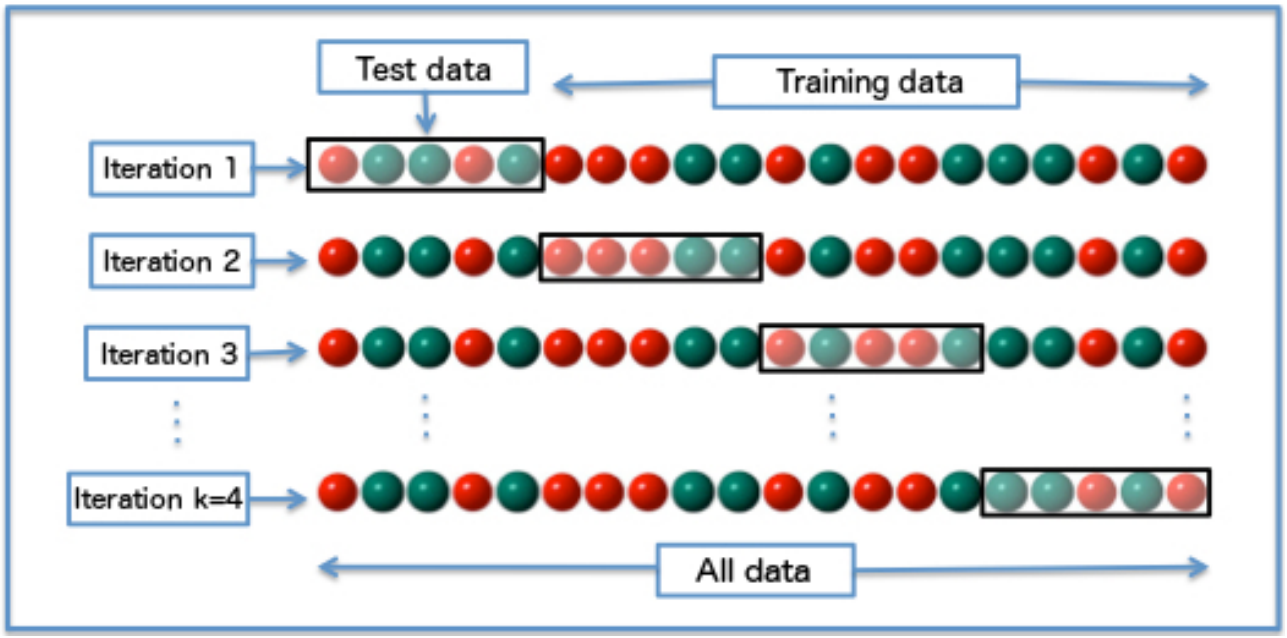


FIGURE 4 – Exemple de fonctionnement pour K=4

Il s'agit désormais d'utiliser la méthode de notre choix dans le but de trouver le meilleur compromis entre précision de l'algorithme et temps de calcul. Le domaine de la *Data Science* fait interagir la théorie mathématique avec la réalisation informatique, il est donc important de prendre en compte ces deux aspects et d'optimiser leur utilisation conjointe au mieux.

4 Analyse des corrélations entre variables et visualisation

4.1 Analyse en Composante Principale

Dans un premier temps nous nous sommes intéressé aux liens existant entre nos variable quantitatives ainsi qu'à leurs importances. Pour cela nous avons réalisé une **ACP** (Analyse en Composante Principale). Cette technique repose sur la construction d'une représentation à une dimension réduite de nos variables. Le but est donc d'avoir un résumé le plus précis possible, sans modifier les distances entres individus et variables. Pour cela nous projetons notre problème sur des axes représentant une certaine partie de l'inertie de nos données.

5 Travail des données

Avant de lancer différents algorithmes sur nos données, il est important de travailler sur des données « propres ». Nous allons donc dans la suite détailler les différentes étapes de nettoyage et de tranformation de certaines données.

5.1 Gestion et remplacement des valeurs manquantes

Une partie clé de cette étude aura été la gestion des valeurs manquantes. En effet, tous les fichiers $train\_H.csv$ $(H = 1, .., 36)$, en contiennent, ce qui nous a donc amené à nous poser les questions suivantes : faut-il éliminer les lignes contenant des *NA* ? Faut-il remplacer ces valeurs manquantes ? Si oui, pour quelles variables, de quelle façon et à quel coût ? Nous allons détailler l'analyse qui nous a permis de répondre à ces différentes interrogations et de prendre nos décisions.

Premièrement nous avons regardé s'il était possible de remplacer les valeurs sur toutes les variables qui en contiennent dans nos données. Toutes les variables numériques (*i.e* qui ne sont pas des facteurs) présentent des valeurs manquantes. Le graphe suivant nous montre l'erreur moyenne pour chaque variable après des tentatives d'imputation par **moyenne**, **médiane**, **10-NN**, **Random Forest** et **décalage temporel** (par exemple s'il manque une valeur pour la variable *ux1H10* à Bordeaux le 20/06/2015 à l'échéance 1, on va remplacer cette valeur par celle de la même date et de la même station à l'échéance 2).

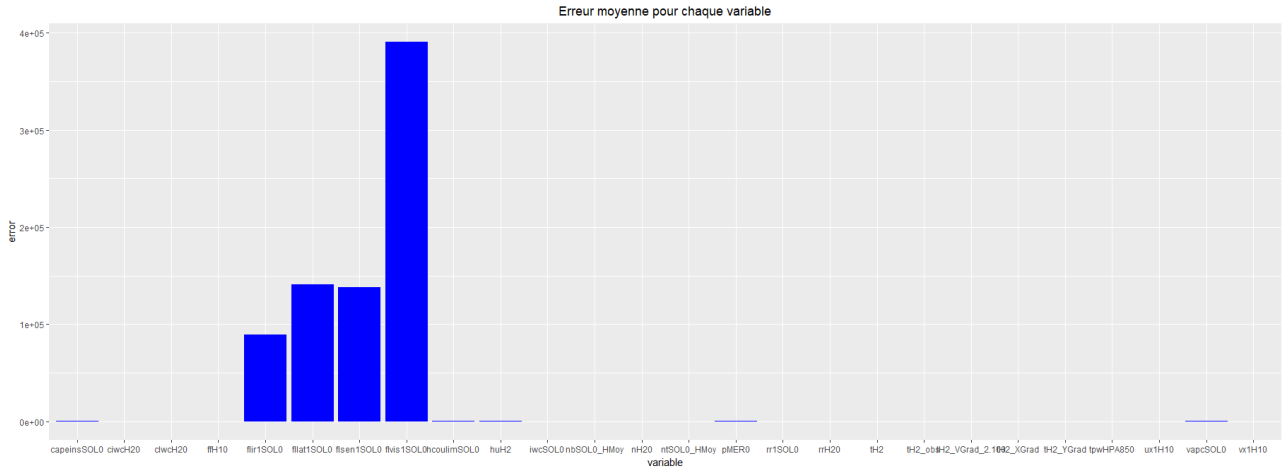


FIGURE 5 – L'imputation de certaines variables semble problématique

Clairement, ce graphique fait ressortir un problème pour les variables **flir1SOL0**, **fllat1SOL0**, **flsen1SOL0**, **flvis1SOL0** et dans une moindre mesure pour les variables **hcoulimSOL0** et **capeinsSOL0**. Nous avons donc décider de ne pas remplacer les valeurs manquantes pour ces variables, puisque les remplacer pourrait potentiellement fausser la qualité de notre analyse.

Suite à ce constat, nous allons donc nous pencher plus en détail sur les variables restantes en testant plusieurs méthodes d'imputation de valeurs. Pour réaliser nos tests, nous avons sélectionné les lignes complètes du fichier regroupant les observations pour la station de Toulouse, puis nous avons tiré 2000 lignes de façon aléatoire et sans remise pour former notre échantillon de test. Ensuite, nous avons créé une copie de ce sous-échantillon dans lequel nous avons remplacé 50% des valeurs par des valeurs manquantes, encore une fois de façon aléatoire. Il ne reste plus qu'à tester différentes méthodes et les comparer.

Parmi les méthodes d'imputation essayées, nous retrouvons :

- L'imputation **LOCF** (Last Observation Carry Forward) qui est similaire à notre décalage temporel sauf que l'on considère la dernière observation pour effectuer le remplacement
- L'imputation par la **moyenne**
- L'imputation par la **médiane**
- L'imputation par les **k-plus-proches-voisins** ($k = 5$ ici)
- L'imputation **LOESS** (**LO**cal regr**ESS**ion) dont le principe se base sur l'ajustement d'un polynôme de degré faible autour de la donnée manquante par moindres carrés pondérés, en donnant plus de poids aux valeurs proches de la donnée manquante
- L'imputation par l'algorithme **SVD** (décomposition en valeurs singulières)
- L'imputation **missForest** basée sur les forêts aléatoires, dont le fonctionnement sera détaillé plus loin dans ce rapport
- L'imputation par l'algorithme **AMELIA II** : Pour chaque tirage, les données sont estimées par bootstrap pour simuler l'incertitude puis l'algorithme EM est exécuté pour trouver l'estimateur a posteriori Θ_{MAP} pour les données bootstrap.

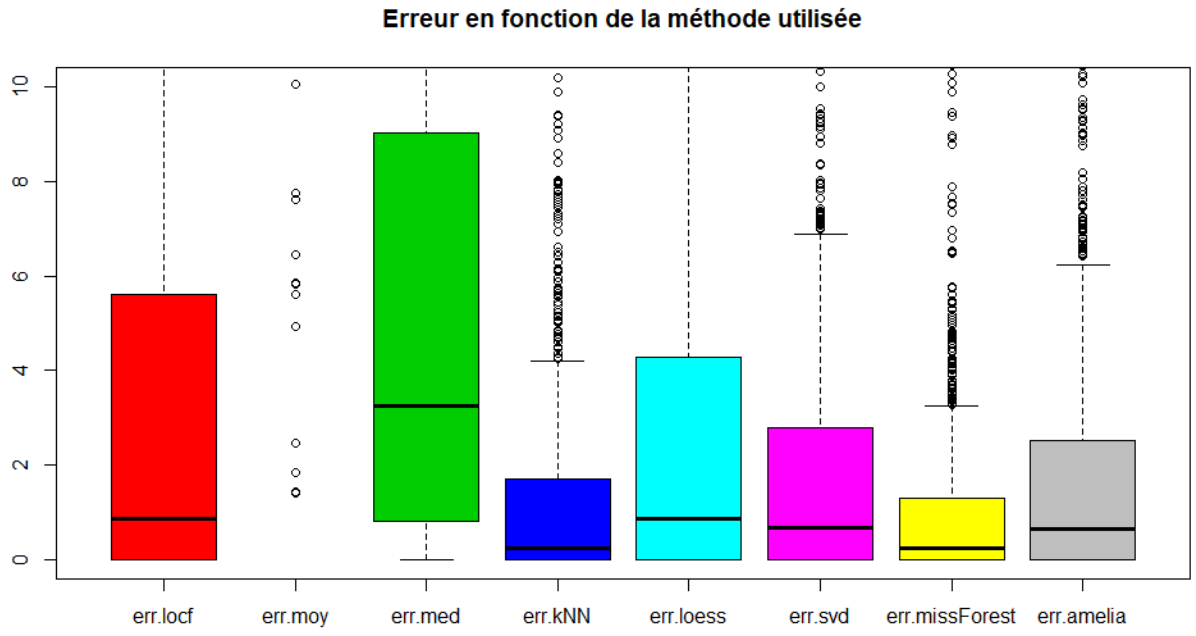


FIGURE 6 – Taux d'erreur des méthodes d'imputation testées

Le graphe précédent nous permet d'observer que c'est la méthode **missForest** qui nous procure le moins d'erreur, c'est donc celle-là que nous avons choisi. L'imputation s'est faite par station, pour les raisons exposées dans la partie « **Structuration des données** ».

5.2 Transformation de la date

Si l'on demandait à une personne de prédire la température, intuitivement la date va forcément faire partie des variables déterminantes dans sa décision. Il paraît évident que l'on ne va pas prédire la même température pour un 15 janvier que pour un 15 juillet. En revanche, on pourrait être tenté de prédire environ la même température pour un 15 avril et pour un 15 octobre, le printemps et l'automne étant assez similaires selon ce critère de température. En revanche, l'été et l'hiver s'opposent. On saisit donc que la variable date va avoir son importance dans l'explication de la température, seulement cette information n'est pas exploitable de façon brute : il va falloir la transformer. Comment choisir cette transformation ? Les critères suivants peuvent nous aider :

- La transformation doit être capable de représenter l'opposition entre l'hiver et l'été, tout en tenant compte du caractère similaire de l'automne et du printemps
- Elle doit également prendre en compte le fait que le 31 décembre d'une année et le 1^{er} janvier de l'année suivante sont 2 jours similaires
- La saisonnalité est un phénomène périodique, il faut donc représenter cet aspect dans la transformation

Après avoir énuméré ces différents critères, on peut avoir l'intuition que cette transformation à une forme sinusoïdale, plus particulièrement on peut choisir le **cosinus**. En effet, avec une telle représentation on pourrait respecter tous les critères énoncés précédemment. L'objectif va être d'associer à chaque date une coordonnée qui représente au mieux sa position dans l'année, par exemple quand on se situe au cœur de l'été on aura une coordonnée proche de 1, et inversement au cœur de l'hiver avec -1. Pour effectuer un tel codage, on procède de la façon suivante :

- on fixe le point de départ au 07/08/2013 : point central de l'été 2013. On attribue 0 comme coordonnée à ce point puisque $cos(0) = 1$
- la coordonnée de la date suivante est incrémentée d'un pas régulier tel que :

$$t_{i+1} = t_i + \frac{2\pi}{N}$$

  où N correspond au nombre de jours que contient chaque année (cette étude contient une année bissextile dont il a fallu prendre compte)

- enfin, il ne reste plus qu'à prendre le cosinus de chaque coordonnée et l'on obtient la représentation qui suit :

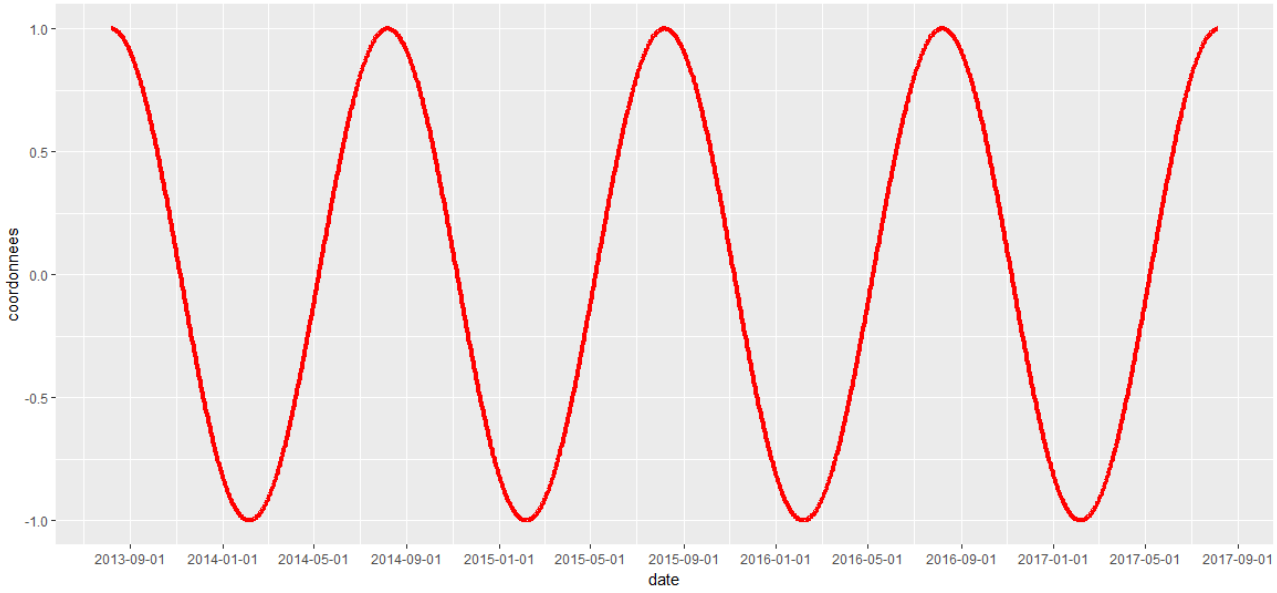


FIGURE 7 – A chaque date correspond une coordonnée indiquant sa position dans la saison

Les transformations et ajustements étant faits, nous pouvons à présent démarrer la partie de modélisation dans le but de prédire les températures.

6 Choix et application d'algorithmes d'apprentissage supervisé

Dans cette section, nous allons détailler les différents algorithmes que nous avons utilisé pour modéliser les données et effectuer nos prédictions. Etant donné qu'il n'existe pas une méthode universelle assurant qu'un algorithme est meilleur que les autres pour un type de problème donné, il faut en tester plusieurs (« No free lunch theorems »). Toutefois, nous pouvons d'ores et déjà enlever les **SVM** de la liste des algorithmes qui seront testés. Nous avons fait ce choix pour des raisons techniques : l'algorithme utilise la matrice de Gram définie de la façon suivante :

$$G = \begin{pmatrix} < x_1, x_1 > & \cdots & < x_1, x_n > \\ \vdots & \ddots & \vdots \\ < x_n, x_1 > & \cdots & < x_n, x_n > \end{pmatrix}$$

Cette matrice est de taille n x n, ce qui dans le cas de nos données peut poser problème en temps de calcul. Sachant que d'autres méthodes fonctionnant en général très bien et de façon plus rapide sont à notre disposition, nous avons donc décidé de ne pas utiliser l'algorithme **SVM**. Pour les méthodes testées, le schéma sera le suivant :

- Présentation du fonctionnement de l'algorithme et de ses propriétés théoriques
- Présentation des meilleurs résultats obtenus avec l'algorithme et des hyper-paramètres correspondants

Nous rappelons également au passage que pour optimiser la performance, le critère à minimiser sera l'erreur quadratique moyenne de notre estimateur que l'on peut exprimer de la façon suivante, si l'on note $\hat{\Theta}$ notre estimateur du paramètre Θ (la température **tH2_obs**) :

$$MSE(\hat{\Theta}) = \mathbb{E}[(\hat{\Theta} - \Theta)^2]$$

Que l'on peut encore décomposer comme étant :

$$MSE(\hat{\Theta}) = Biais(\hat{\Theta})^2 + Var(\hat{\Theta})$$

C'est donc sur ces deux aspects que nous allons tenter de jouer par la suite. Afin d'obtenir la « meilleure » performance possible, il faudra trouver le « meilleur » compromis biais/variance.

6.1 Premiers algorithmes : la régression linéaire et ses variantes

6.2 Forêt aléatoire : l'algorithme Random Forest

6.2.1 Rappels sur le fonctionnement de la méthode CART

Une façon simple de modéliser des données peut être d'utiliser un arbre de décision **CART** (**C**lassification **A**nd **R**egression **T**ree). C'est un algorithme de prédiction par moyenne locale, par partition, dont la partition est construite par divisions successives au moyen d'hyperplans orthogonaux aux axes du nœud racine $\mathfrak{X} = \mathbb{R}^p$, dépendant des (X_i, Y_i). Chaque division est effectuée à partir du choix conjoint d'une variable explicative et d'une valeur seuil pour cette variable, de sorte que la découpe effectuée minimise la variance en régression ou l'impureté dans le nœud en discrimination.

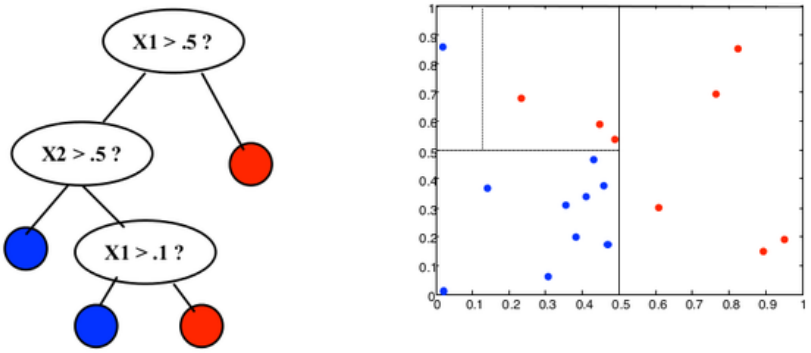


FIGURE 8 – Illustration de l'algorithme CART pour un problème de discrimination binaire dans $\mathbb{R}^2$

Néanmoins cet algorithme pose le problème majeur d'être très instable (fléau de la dimension et sensibilité à l'échantillon). C'est donc pour palier à ce problème que nous allons utiliser l'algorithme de forêt aléatoire **Random Input**. Comme son nom l'indique, cette méthode consiste en une agrégation d'arbres : c'est un algorithme de **Bagging** (**B**ootstrap **agg**regat**ing**).

6.2.2 Le bagging

Le **bagging** s'applique à des algorithmes instables, de variance forte, ce qui est le cas des arbres **CART**. En utilisant un algorithme de **bagging**, le but va justement être de réduire la variance (quitte à introduire un peu de biais). En régression, on va agréger **B** algorithmes $\hat{\eta_1}, ..., \hat{\eta_B}$ sous la forme :

$$\hat{\eta}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{\eta_b}(x)$$

Cependant en pratique ces algorithmes ne peuvent pas être i.i.d puisqu'ils sont construits sur le même échantillon D_1^n, et si l'on note $\rho(x)$ le coefficient de corrélation entre $\hat{\eta_b}$ et $\hat{\eta_{b'}}$ on a la relation suivante :

$$Var(\hat{\eta}(x)) = \rho(x)Var(\hat{\eta_b}(x)) + \frac{1-\rho(x)}{B}Var(\hat{\eta_b}(x))$$
$$\text{d'où il vient : } \lim_{B\to\infty} Var(\hat{\eta}(x)) = \rho(x)Var(\hat{\eta_b}(x))$$

Cela illustre donc le fait que le bagging va réduire la variance de l'estimateur. En revanche, on introduit un terme de corrélation : la solution de **Breiman** pour construire des algorithmes peu corrélés entre eux consiste à utiliser non pas l'échantillon D_1^n mais plutôt des échantillons **bootstrap** de D_1^n, c'est-à-dire que l'on va tirer de façon aléatoire et avec remise des lignes parmi l'échantillon D_1^n.

6.2.3 L'algorithme Random Forest RI

L'algorithme de forêt aléatoire Random Input est donc une variante de la méthode CART qui s'appuie sur les techniques de bagging. Son fonctionnement est le suivant :

Algorithm 1 Random Forest RI

Entrées :
x : l'entrée dont on veut prédire la sortie
d_1^n : l'échantillon observé
m : le nombre de variables explicatives sélectionnées à chaque nœud
B : le nombre d'itérations
Début
Pour b variant de 1 à B
Tirer un échantillon bootstrap d^{*b} de d_1^n
Construire un arbre maximal $\hat{\eta_b}$ sur l'échantillon bootstrap d^{*b} par la variante de CART
Pour chaque nœud variant de 1 à N_b
Tirer un sous-échantillon de m variables explicatives
Partitionner le nœud à partir de la « meilleure » de ces m variables
Fin pour
Fin pour
Retourner $\frac{1}{B} \sum_{b=1}^{B} \hat{\eta_b}(x)$
Fin

Il faudra donc calibrer plusieurs hyper-paramètres pour cette méthode : le nombre d'itérations **B** qui comme on l'a vu précédemment permet de réduire la variance lorsqu'il est grand, et le nombre m de variables explicatives sélectionnées à chaque nœud. Pour cela, nous pouvons utiliser plusieurs méthodes de validation croisée présentées auparavant ou bien encore la méthode **Out Of Bag** qui consiste à tester l'algorithme sur les observations n'appartenant pas à l'échantillon bootstrap d^{*b} ayant servi à construire l'algorithme de prédiction. En régression, l'erreur Out Of Bag d'une forêt aléatoire RI est définie par :

$$Error_{OOB} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \sum_{b=1}^{B} I_i^b \hat{\eta_b}(x)}{\sum_{b=1}^{B} I_i^b}\right)^2 \text{ où } I_i^b = 1 \text{ si l'observation } i \notin d^{*b}, \text{ 0 sinon}$$

6.2.4 Résultats obtenus

Les meilleurs résultats pour cette étude ont été obtenus par validation croisée 5-blocs. Le meilleur estimateur pour Toulouse, Nice, Strasbourg et Paris est le suivant :

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=11, max_leaf_nodes=None, min_impurity_split=1e-07,
           min\_samples_leaf=1, min\_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=800, n_jobs=1,
           oob_score=False, random_state=None, verbose=0, warm_start=False)
```

Pour les autres stations (Bordeaux, Rennes et Lille) le seul hyper-paramètre qui change est le nombre de variables explicatives sélectionnées à chaque nœud que l'on a appelé m (*max_features* dans la fonction **RandomForestRegressor** de la librairie *scikit-learn* en Python) qui vaut 13. Le nombre d'itérations optimal B (ou *n_estimators* ici) est de 800. Les résultats procurés par cet estimateur sont les suivants :

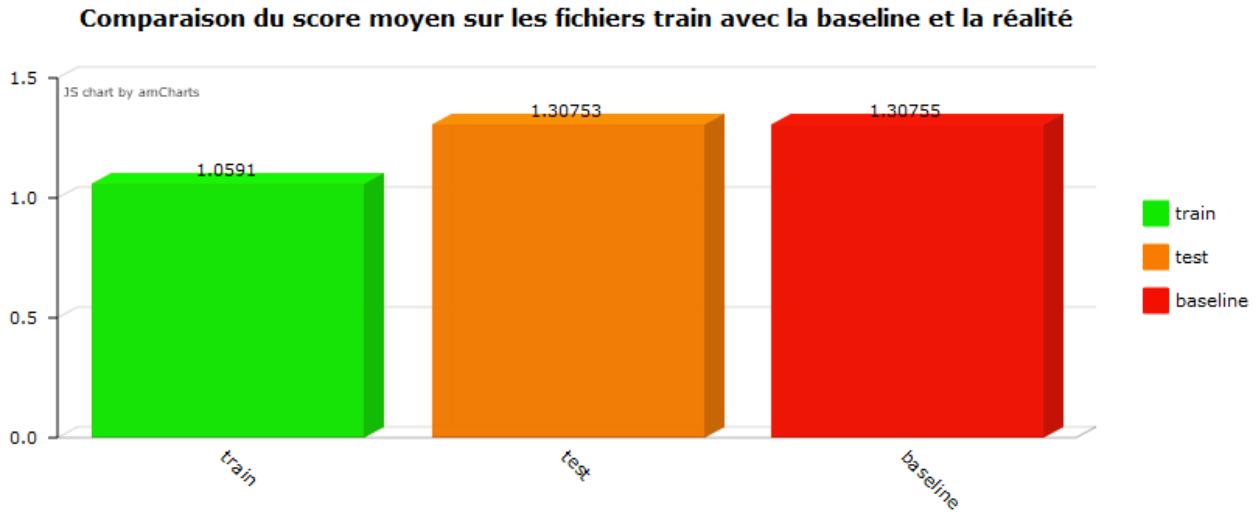


FIGURE 9 – Le score de cet algorithme est légèrement meilleur que la baseline

Cet algorithme permet donc de franchir la **baseline**, cependant on s'aperçoit encore d'un écart important entre le score moyen d'entraînement sur les 7 stations et le score sur le fichier de test. Nous allons maintenant essayer d'autres algorithmes dont le fonctionnement est différent et regarder s'ils nous permettent de gagner en performance.

6.3 Boosting : l'algorithme XGBoost

6.3.1 Rappels sur le fonctionnement du boosting

Le **boosting**, par opposition au bagging, s'applique à des algorithmes fortements biaisés, mais de faible variance. Le principe est donc bien différent de celui d'une forêt aléatoire où l'on va agréger les prédictions de plusieurs arbres pour en faire la moyenne.

Ici on va combiner dans un algorithme itératif les prédictions de *weak learners*, c'est-à-dire des classifieurs ou des régresseurs faibles, selon la nature du problème. Ces prédicteurs faibles sont encore une fois des arbres avec peu de nœuds (pour rester faibles et éviter le sur-apprentissage).

Si l'on se place dans un problème de discrimination, à chaque itérations on va attribuer un poids au classifieur en fonction de sa qualité d'estimation (selon le critère du taux de bon classement par exemple). Les observations qui ont été mal classées se verront attribuer un poids plus fort afin d'être mieux prédites lors de la prochaine itération. La figure

suivante résume le fonctionnement de ce type d'algorithmes : chaque *Box* correspond à une itération, et les prédicteurs utilisés sont des arbres à 2 nœuds (encore appelés *stumps*).

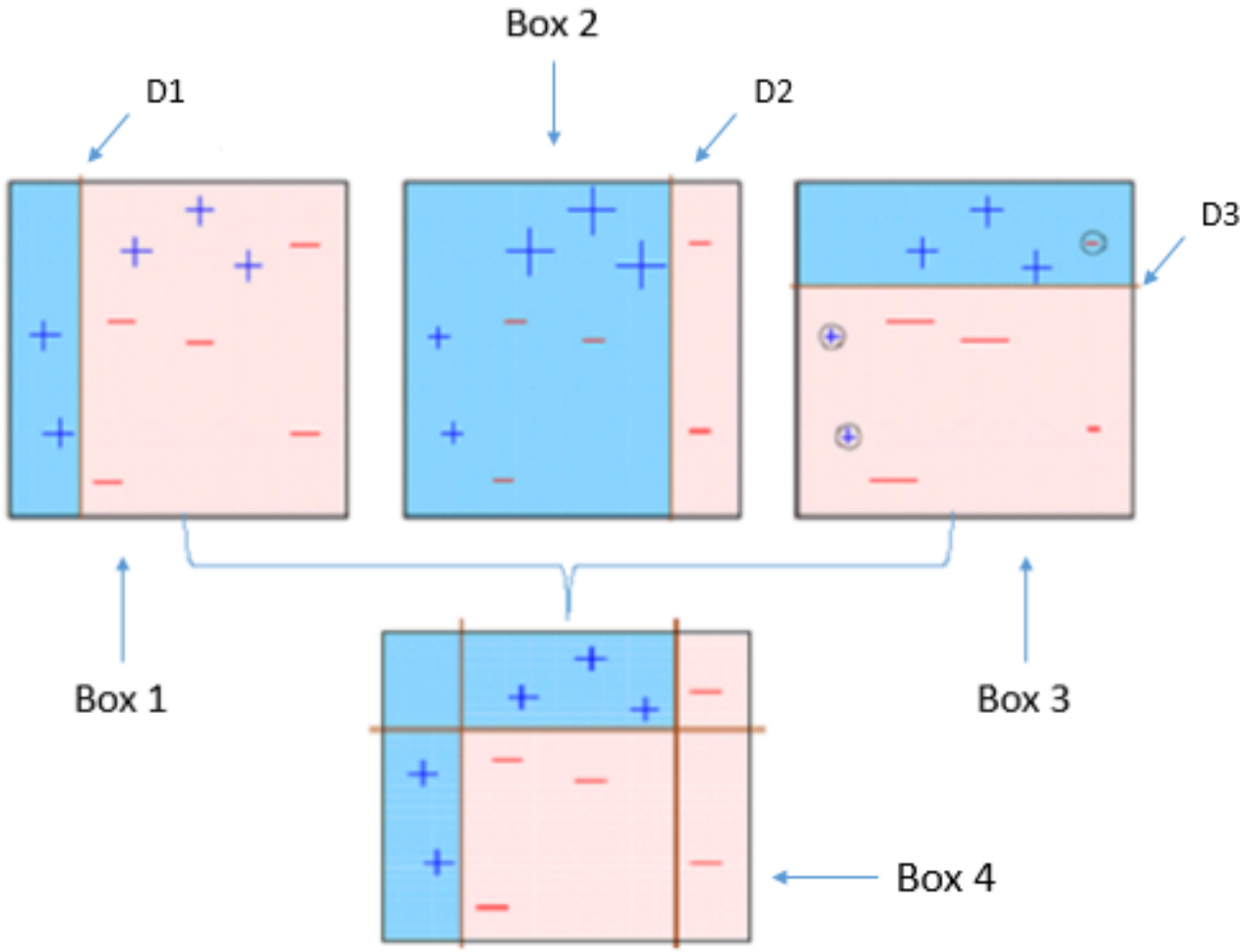


FIGURE 10 – Illustration du boosting pour un problème de discrimination binaire dans $\mathbb{R}^2$

Il existe énormément de déclinaisons du boosting, dans cet étude nous allons nous focaliser sur un algorithme d'**eXtreme Gradient Boosting** (XGBoost) régularisé qui s'appuie sur un algorithme de minimisation du risque empirique (fonction de perte l) par descente de gradient. Nous avons choisi cet algorithme par rapport à sa popularité que ce soit au niveau des compétitions de type *Kaggle* ou en entreprise (plusieurs soutenances de stage d'élèves de Master 2 parlaient de cette méthode en 2016), mais également pour sa rapidité à être implémenté. La version régularisée doit nous permettre d'éviter, ou en tout cas de limiter le sur-apprentissage.

6.3.2 Descente de gradient stochastique

Avant d'expliquer le fonctionnement de l'algorithme **XGBoost** il est primordial d'effectuer un rappel sur l'algorithme d'optimisation qui en est la base : la descente de gradient. Nous avons même fait le choix d'utiliser une descente de gradient stochastique (on ne regarde qu'une partie des observations pour effectuer la mise à jour du gradient). Cela a pour propriété de réduire la variance moyennant une augmentation du biais, mais les éléments de littérature laissent à penser que la réduction de variance est plus importante que l'introduction de biais, et que cela permet donc d'améliorer la performance de l'algorithme de prédiction. Dans le cas simple où l'on cherche à minimiser une fonction convexe, on se fixe un point de départ et on « descend » le long de cette fonction en la différenciant jusqu'à atteindre le minimum global.

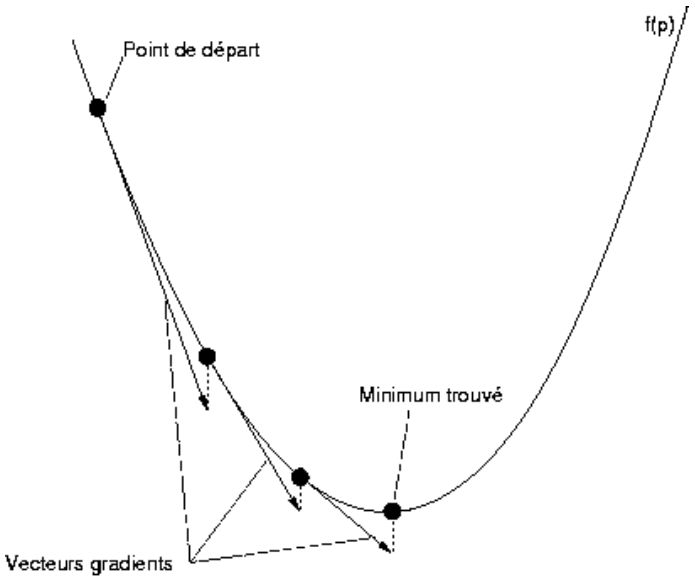


FIGURE 11 – Descente de gradient sur une fonction convexe simple dans le plan

Cependant, un paramètre essentiel à choisir dans cette méthode va être le **pas de descente**. Un pas trop grand va mener à une irrégularité et à des résultats très mauvais car le minimum ne sera jamais atteint.
Au contraire, un pas trop petit va premièrement avoir pour effet de ralentir le temps de calcul de l'algorithme, et deuxièmement nous exposer au risque de ne jamais atteindre le minimum si l'on s'arrête trop tôt au niveau du nombre d'itérations de l'algorithme.

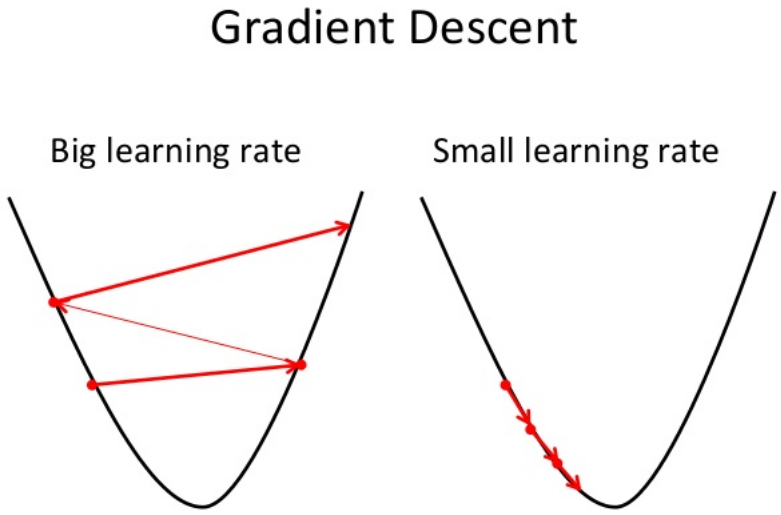


FIGURE 12 – Problème : le choix du pas est crucial

Cela nous permet déjà de nous rendre compte que le nombre d'itérations de l'algorithme et le pas (learning rate) sont deux paramètres qui vont être liés et qui seront cruciaux dans la quête du minimum global de la fonction de perte qu'on cherche à atteindre, et donc pour la qualité d'estimation. En revanche, les cas présentés ci-dessus sont des cas simples, où la fonction est convexe. Une difficulté supplémentaire en pratique va souvent être que la fonction de perte que l'on cherche à minimiser ne sera pas convexe.

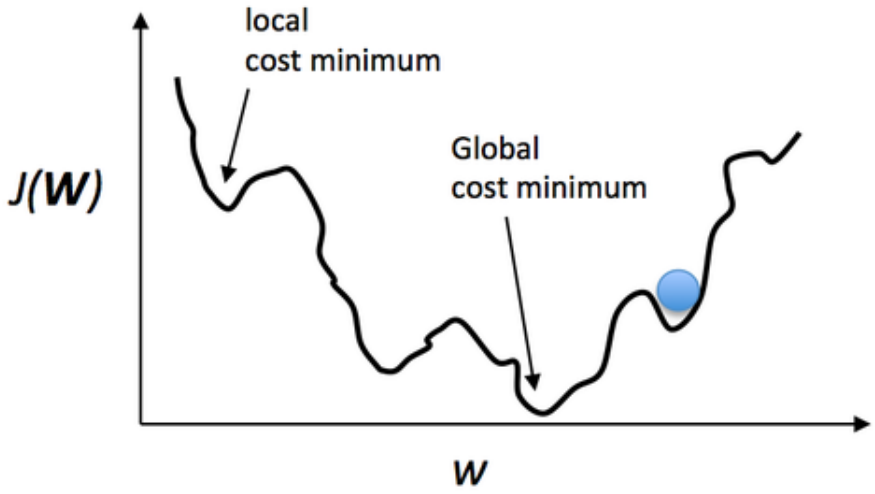


FIGURE 13 – Dans la pratique on se retrouve plus souvent confronté à ce type de fonctions

L'enjeu va donc être de choisir un nombre d'itérations et un pas optimal afin de ne pas se retrouver « coincé » dans un minimum local et de pouvoir atteindre le minimum global de la fonction de perte qu'on souhaite minimiser. Une autre solution plus complexe pourrait être de chercher un majorant convexe de la fonction de perte et de chercher à le minimiser. Ce principe est par exemple utilisé dans le cadre d'une SVM où l'on procède à une minimisation du risque empirique convexifié régularisé.

6.3.3 Fonctionnement du Gradient Boosting régularisé

Dans cette partie, nous allons exprimer le fonctionnement de cette méthode de façon algorithmique puis en détailler les étapes afin de bien comprendre le traitement effectué par cette opération. L'algorithme de gradient boosting régularisé en régression peut s'écrire de la façon suivante :

Algorithm 2 Gradient Boosting régularisé

Entrées :
ϕ_γ : algorithme faible de paramètre γ à ajuster
$\lambda \in]0, 1]$: paramètre de régularisation
Début
Initialiser $\hat{\eta}^0(x) = \underset{c}{argmin} \sum_{i=1}^{n} l(y_i, c)$
Pour b variant de 1 à B
Calculer $z_i^b = -\frac{\partial}{\partial_{\eta(x_i)}} l(y_i, \eta(x_i))|_{\eta(x_i)=\hat{\eta}^{b-1}(x_i)}$
Ajuster $\hat{\gamma}^b = \underset{\gamma}{argmin} \sum_{i=1}^{n} (z_i^b - \phi_\gamma(x_i))^2$
Calculer $\hat{\beta}_b = \underset{\beta}{argmin} \sum_{i=1}^{n} l(y_i, \hat{\eta}^{b-1}(x_i) + \beta\phi_{\hat{\gamma}^b}(x_i))$
Mettre à jour $\hat{\eta}^b(x) = \hat{\eta}^{b-1}(x) + \lambda\beta\phi_{\hat{\gamma}^b}(x)$
Fin pour
Retourner $\hat{\eta}^B$
Fin

Nous allons à présent reprendre ces écritures afin de mieux comprendre les tâches effectuées par l'algorithme. Premièrement décrivons les entrées de cet algorithme :

- ϕ_γ va correspondre dans notre étude à un arbre. γ sera par exemple le nombre de nœuds de cet arbre que l'on ajustera par validation croisée 5-blocs encore une fois

- λ est l'hyper-paramètre qui va nous permettre d'effectuer de la régularisation. On l'appelle également le *learning rate*, c'est ce que nous avons décrit précédemment comme étant le pas de descente de gradient. Puisque $\lambda \in]0, 1]$ on voit donc qu'il réduit/limite l'apprentissage de chaque itération, ce qui doit nous aider à nous prémunir contre le risque de sur-apprentissage

On entre ensuite au cœur du problème d'optimisation, qui dans le cas de la méthode **XGBoost** s'appuie sur une minimisation du risque empirique par descente de gradient. Cela implique également le choix d'une fonction de perte l. Comme expliqué précédemment :

- on choisit un point de départ lors de l'initialisation, puis on procède à la descente de gradient (dans la direction de plus forte descente correspondant à z_i^b) dans le but de minimiser l
- on met à jour les poids en fonction de la qualité de prédiction du « prédicteur faible »
- on actualise la prédiction $\hat{\eta}^b(x)$ de l'itération en fonction de la prédiction de l'itération précédente et de l'apprentissage effectué au cours de l'itération actuelle
- A la fin des **B** itérations que l'on s'est fixé, on renvoie la dernière prédiction.

Comme introduit précédemment, le nombre d'itérations **B** et le learning rate λ sont deux hyper-paramètres qui interagissent et qu'il va donc falloir calibrer de façon simultanée. Cet ajustement va se faire par validation croisée 5-blocs et on va fixer l'un des paramètres en faisant varier l'autre. En ce qui nous concerne, le nombre d'itérations sera fixé et c'est le learning rate qui va varier.

6.3.4 Résultats obtenus

La sortie en Python correspondant au meilleur estimateur, que nous allons détailler, est la suivante :

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
             learning_rate=0.05, loss='ls', max_depth=8, max_features=None,
             max_leaf_nodes=None, min_impurity_split=1e-07,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=1200,
             presort='auto', random_state=None, subsample=0.9, verbose=0,
             warm_start=False)
```

Les hyper-paramètres optimaux sont les mêmes pour toutes les stations, parmi ceux qui ont été choisis et ajustés (*i.e* pas laissés par défaut), on retrouve :

- un nombre d'itérations (*n_estimators*) de 1200 associé à un pas de descente (*learning_rate*) de 0.05
- la fonction de perte (*loss*) classique, à savoir « ls » pour *least squares regression*. La fonction **Huber** réputée plus robuste dans certains articles n'apportait pas plus de performance
- une profondeur d'arbre (*max_depth*) de 8. On souhaite garder un arbre peu profond afin qu'il reste faible et qu'on évite le sur-apprentissage, cependant les *stumps* (arbres à 2 nœuds) procuraient de moins bons résultats finaux.
- une proportion d'observations (*subsample*) utilisées pour effectuer les mises à jour de gradient de 0.9 : c'est ce paramètre qui permet d'effectuer une descente de gradient stochastique

Les résultats obtenus grâce à cet estimateur sont les suivants :

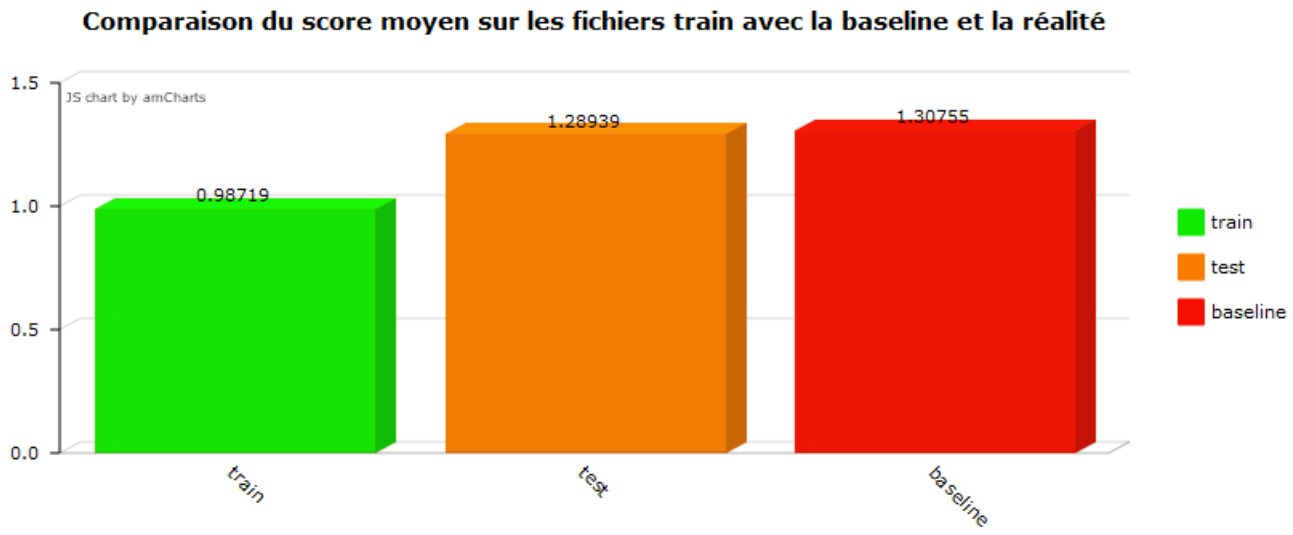


FIGURE 14 – La performance est meilleure par rapport aux algorithmes précédents

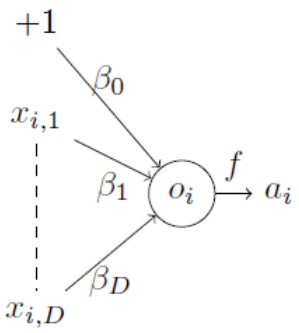
6.4 Deep Learning : l'utilisation des réseaux de neurones

6.4.1 Le contexte

Le Deep Learning permet de résoudre différents problèmes que nous pouvons rencontrer avec les réseaux de neurones comme la stabilité. Aujourd'hui il est souvent utilisé pour des problèmes divers avec des résultats parfois remarquables comme nous pouvons le voir avec le site de traduction *DeepL*, le plus abouti aujourd'hui pour le grand public ou encore pour des outils d'analyse des émotions par un visage photographié ou filmé.

6.4.2 Le principe

Le principe de base du modèle (montré dans l'image ci-dessous) est le neurone, un modèle d'inspiration biologique du neurone humain. Chez l'homme, les différents niveaux de puissance des signaux de sortie des neurones circulent le long des jonctions synaptiques et sont ensuite agrégés en entrée pour l'activation d'un neurone connecté. On appelle ce modèle le modèle **perceptron**.



Dans le modèle, la fonction f représente la fonction d'activation non linéaire utilisée dans tout le réseau et le biais β_0 représente le seuil d'activation du neurone :

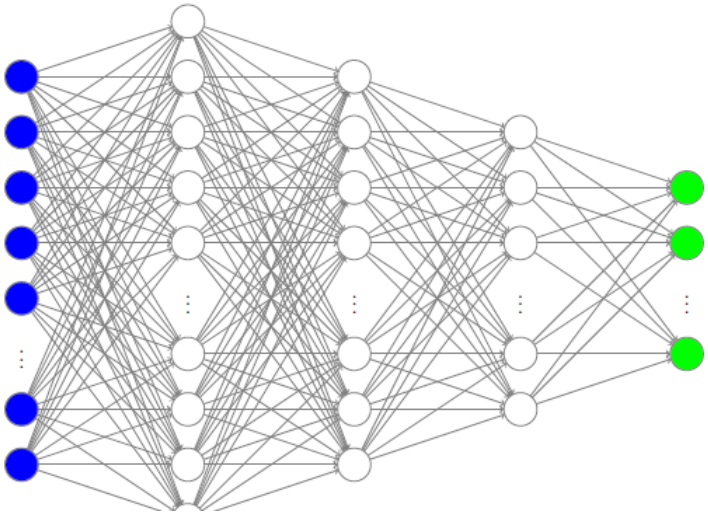
$$y_i = f(\beta_0 + \sum_j \beta_j x_{i,j}) + \varepsilon_i$$

Une fois que nous avons choisi notre fonction d'activation, il est nécessaire comme dans tout problème d'optimisation de se fixer une fonction de coût à minimiser. Il s'agit ici d'un problème de descente de gradient comme vu précédemment dans la partie sur le Boosting.

$$L(\beta) = \sum_i (y_i - a_i)^2 = \sum_i (\beta)$$

Il ne reste plus qu'à exprimer le gradient de cette fonction de coût et choisir un pas η pour effectuer les mises à jour.

Les réseaux neuronaux multicouches en aval sont constitués de plusieurs couches d'unités neuronales interconnectées (comme le montre l'image suivante), commençant par une couche d'entrée pour correspondre à l'espace d'entrée (le nombre de variables), suivie de plusieurs couches non-linéaire, et se terminant par une régression linéaire ou une couche de classification pour correspondre à l'espace de sortie (une régression linéaire dans notre cas). Les entrées et les sorties des unités du modèle suivent la logique de base du neurone décrite ci-dessus.



Les poids reliant les neurones et les biais avec d'autres neurones déterminent entièrement la sortie de l'ensemble du réseau. L'apprentissage se produit lorsque ces poids sont adaptés pour minimiser l'erreur sur les données d'entraînement marquées.

6.4.3 Notre premier réseau de neurones

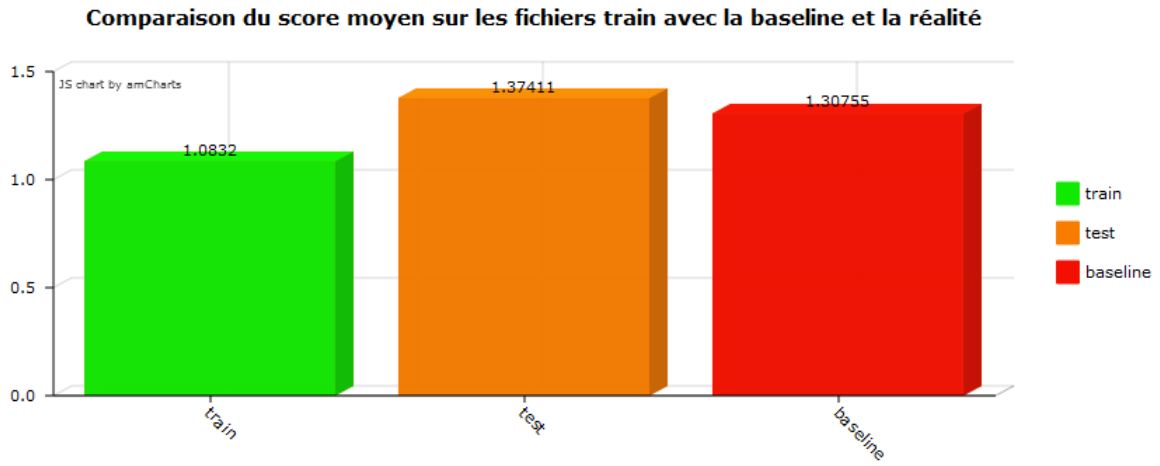
Nous avons réalisé notre premier réseau de neurones rapidement dans la compétition en parallèle des autres méthodes essayées. Pour cela nous avons utilisé le package $H2O$ sous R. Nous avons pu tester un premier algorithme de Deep Learning sans grille pour comprendre tous les paramètres et hyper-paramètres qui vont rentrer en jeu ensuite pour avoir un résultat le plus poussé possible. Avec le package $H2O$ nous dénombrons plus de 60 arguments. Quels arguments choisir et comment ?

```
model <- h2o.deeplearning(x=predictors, y=response,training_frame=train,
validation_frame=valid,stopping_metric="RMSE",
stopping_tolerance=1e-7,activation="RectifierWithDropout",hidden=c(128,128,64),
epochs=500,l1=1e-7,stopping_rounds=100,rate=0.0001,
hidden_dropout_ratios=c(0.2,0.2,0.2))
```

Pour notre premier essai sur un réseau de neurones nous avons choisi un modèle simple à 3 couches cachées. Le modèle est effectué à l'aide d'une validation croisée **Hold-Out** $80 - 20$ pourcent avec un nombre d'epochs à 500.

- Le nombre d'*epochs* spécifie le nombre d'itérations sur le jeu de données d'entraînement.
- *Stopping metric* correspond à la métrique employée pour utiliser un *early stopping*. Cela permet d'arrêter notre descente de gradient quand la tolérance relative de l'arrêt métrique pour arrêter la formation (si amélioration) est inférieure à cette valeur, ici c'est notre *stopping tolerance*.
- *Stopping Rounds*; le nombre d'itérations avant d'arrêter l'algorithme si nous avons convergence.

- *Activation* nous permet de choisir la fonction d'activation des neurones du réseau, ici c'est la fonction d'activation *ReLU* (rectified linear unit).
- *l1*, spécifie la régularisation l1 (L1 = lasso) qui contraint la valeur absolue des poids (peut ajouter de la stabilité et améliorer la généralisation, entraîne de nombreux poids à devenir 0).
- *Rate* permet de sélectionner le taux d'apprentissage α. Des valeurs plus élevées conduisent à des modèles moins stables, tandis que des valeurs plus faibles entraînent une convergence plus lente.
- *Hidden dropout ratios* va permettre de diminuer le sur-apprentissage en omettant une fraction de chaque couche cachée.



La première difficulté réside dans le fait de choisir notre réseau, un réseau avec beaucoup de couches sera-t-il meilleur pour notre problème ? Un réseau avec des couches larges est généralement meilleur pour l'apprentissage mais à partir de quel moment sommes-nous dans le *sur-apprentissage* ?

6.4.4 Optimisation du choix des paramètres pour un réseau de neurones sous H2O

Une fois notre premier modèle effectué nous avons eu pour objectif d'avoir un modèle correspondant à notre problème. Pour une raison de temps et d'efficacité nous avons décidé de fixer certains paramètres tandis que d'autres rentreront dans une grille, ils seront nos hyper-paramètres à optimiser.

```
hyper_params <- list(
hidden=list(c(32,32,32),c(64,64)),
input_dropout_ratio=c(0,0.05),
rate=c(0.01,0.02),
rate_annealing=c(1e-8,1e-7,1e-6))
```

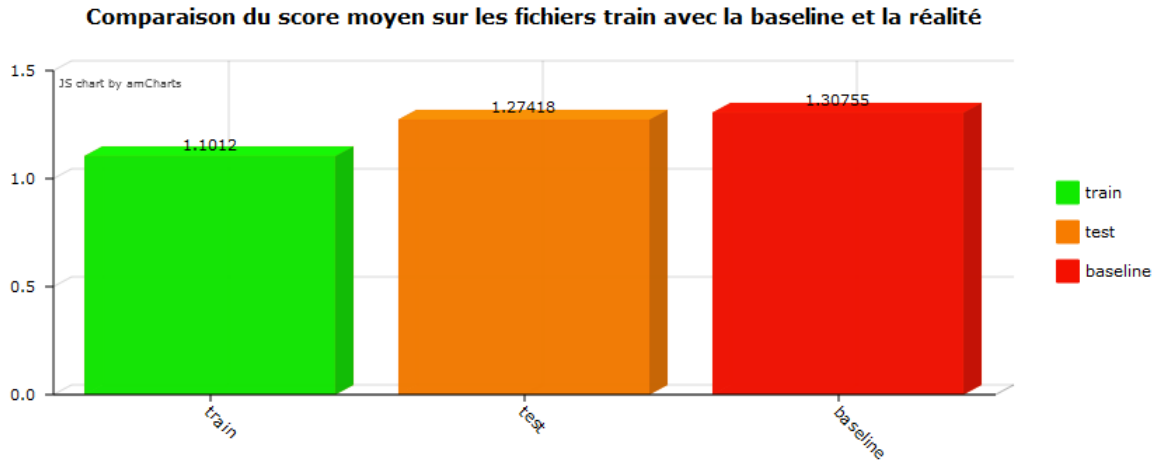
Nous utilisons les paramètres vu dans le premier modèle complété par un **rate annealing**, son but est de diminuer le taux d'apprentissage (rate) au cours de l'avancement de l'algorithme. Ce paramètre permet de gagner du temps en ayant une plus forte descente de gradient au départ afin de se rapprocher de la zone du minimum, puis ensuite en diminuant le pas au fur et à mesure de l'avancement, d'atteindre le minimum global. Cette technique suit le même principe que la dichotomie.

Pour notre modèle nous avons fixé certains paramètres :

- **epochs**, le nombre d'epochs est surtout un choix pour la rapidité du modèle, un grand nombre d'epochs sera toujours favorable.
- la régularisation **l1**.
- le paramètre **max w2**, il spécifie le maximum pour la somme des poids carrés entrants d'un neurone. C'est particulièrement utile quand nous utilisons une fonction d'activation telle que Rectifier (ReLU), cela permet d'éviter d'avoir trop de poids sur un neurone et de mieux répartir l'information.
- **Momentum start**, l'ajout du paramètre de momentum permet de stabiliser l'apprentissage, tout en augmentant la descente de la courbe de la somme des erreurs carrées (fonction de perte) selon les itérations.

Cette grille nous a permis de déterminer le modèle le plus adapté parmi ceux proposés :

```
model <- h2o.deeplearning(x=predictors,
                          y=response,
                          training_frame=train,
                          validation_frame=valid,
                          stopping_metric="RMSE",
                          stopping_tolerance=1e-3,
                          activation="RectifierWithDropout",
                          hidden=c(64,64),
                          input_dropout_ratio=0.05,
                          epochs=500, l1=1e-5,stopping_rounds=100,
                          rate=0.01,rate_annealing=1e-7)
```



Avec ce modèle nous avons atteint notre meilleur score pendant 1 mois mais là est apparu une limite, en essayant d'améliorer notre modèle nous passions à chaque fois dans de l'overfitting.

6.5 Passage sous Keras

Suite à cette stagnation, nous avons décidé d'appliquer le cours de Mr Tavenard sur les réseaux de neurones en utilisant la bibliothéque *Keras* sous python.

6.5.1 Premier modèle sous Keras

Pour réaliser notre premier réseau sous *Keras*, nous avons utilisé nos données sans les variables *capeinSOL0* et *rr1SOL0* ainsi que la transformation sinusoïdale sur les dates. Cela nous permet une fois les valeurs manquantes enlevées d'avoir 177977 individus sur 189280 ainsi nous limitons la perte de données et cela nous permet de ne pas insérer du biais en faisant une estimation sur les valeurs non présentes.

Python et les data.frame *pandas* ne prennent pas en compte les variables catégorielles de la même façon, là ou en *R* il suffit de passer la colonne en facteur, sur *Python* nous devons transformer notre data.frame en tableau disjonctif.

Exemple de la transformation :

	Repas idéal	
	boisson	aliment
ind1	Coca	Brioche
ind2	Lait	Crêpes
ind3	Coca	Cookies

	Repas idéal				
	Coca	Lait	Brioche	Cookies	Crêpes
ind1	1	0	1	0	0
ind2	0	1	0	0	1
ind3	1	0	0	1	0

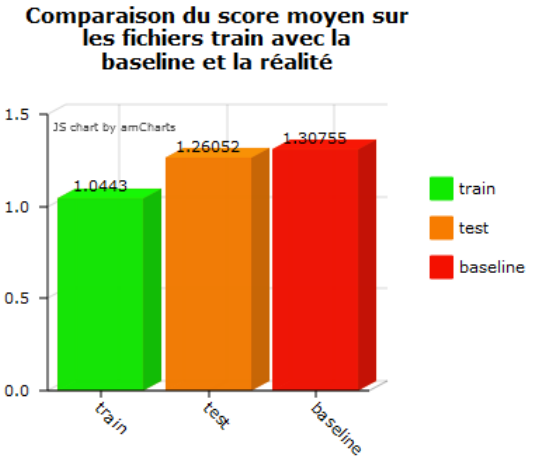
Nous nous retrouvons donc avec un data.frame au dimension suivante $1777977x72$ et notre variable cible $1777977x1$. Puis dans un second temps nous standardisons nos variables explicatives pour le jeu de données train et test, pour cela on standardise d'abord notre jeu d'apprentissage puis nous appliquons la même transformation au jeu de test.

Notre premier modèle se veut simple et compréhensible afin d'effectuer nos premiers test sous *Keras*. Pour cela nous prenons un modèle à 1 couches cachées, nous incluons aussi un DropOut de ces couches pour limiter le sur-apprentissage comme sous *H2O*, la fonction d'activation ReLU ainsi qu'un learning rate de 0.0001.

```
premiere_couche = Dense(units=100, activation="relu", input_dim=72)
couche_cachee1 = Dense(units=100, activation="relu")
couche_sortie = Dense(units=1)
opti = RMSprop(lr=0.0001)

def baseline_model():
    model = Sequential()
    model.add(premiere_couche)
    model.add(Dropout(0.02, noise_shape=None, seed=None))
    model.add(couche_cachee1)
    model.add(Dropout(0.02, noise_shape=None, seed=None))
    model.add(couche_sortie)
    model.compile(optimizer=opti, loss="mean_squared_error")
    return model

model = baseline_model()
hist = model.fit(X_train,y_train,
validation_split=0.3,epochs=500,batch_size=500,verbose=2)
```



Ce modèle nous a permis d'améliorer notre précèdent meilleur score et d'envisager des améliorations.

6.5.2 Notre meilleur score

Suite à ce modèle nous décidâmes de réutiliser le même modèle avec un data.frame enrichi. Plusieurs pistes ont été envisagées puis testées :

- Rajouter les coordonnées de l'ACP précédemment calculées,
- Rajouter les coordonnées de l'ACM sur nos jeux de données,
- Rajouter des variables par intéractions.

Le but étant de rajouter de l'information à nos jeux de données en ajoutant des variables même si elles sont corrélées aux variables initiales en prenant le principe simple de la grande dimension :

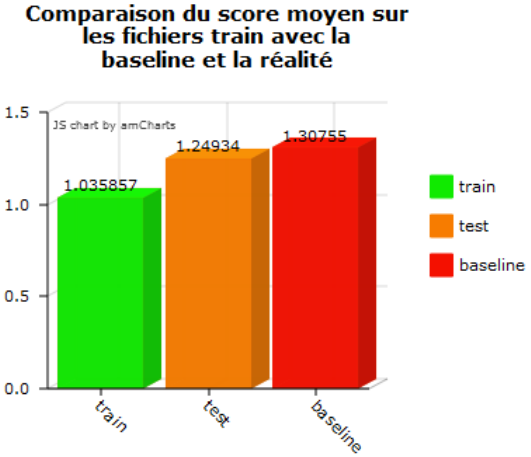
> "Rajouter de l'information ne sera jamais préjudiciable, dans le meilleur des cas nous améliorons notre score, dans le pire nous n'améliorons pas notre modèle mais nous ne le pénalisons pas non plus.

La complexité reposera principalement sur le fait de réussir à adapter notre modèle aux nouvelles variables.

Premièrement nous allons ajouter les coordonnées des individus sur les principaux axes de l'ACP. Cette manoeuvre sera composé de plusieurs étapes :

- Effectuer une ACP sur le jeu de données train sans la variable cible.
- Récupérer les coordonnées des individus sur les axes représentant 80 pourcent de l'inertie, au delà nous considérons que c'est du bruit.
- Projeter nos individus du jeu test sur nos axes calculés précédemment.

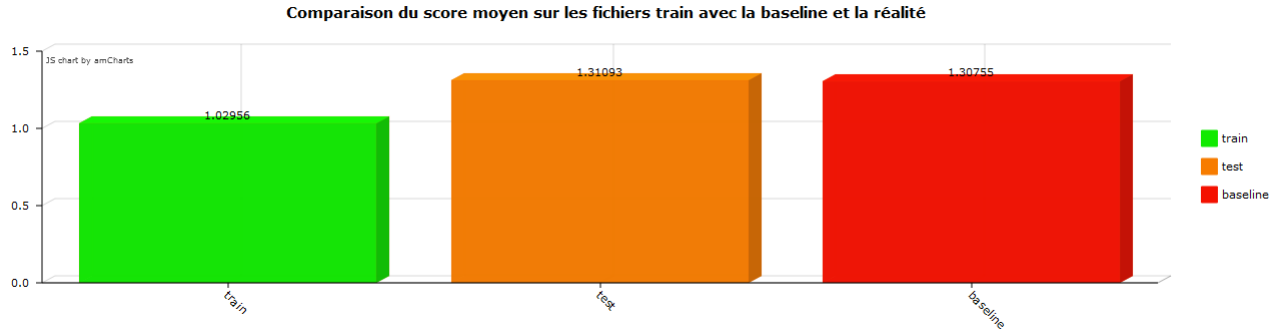
Avec ce procédé nous agrémentons nos jeux de données de 11 variables correspondant aux axes représentant 82 pourcent de l'inertie. Nous effectuons le même modèle que précédemment en adaptant la dimension d'entrée du réseau, en passant de 72 à 83. Les résultats dépassent nos premières attentes :



Deuxièmement nous allons compléter nos data.frames avec les coordonnées de l'ACM (sur les données qualitatives). Pour la réalisation de cette étape nous procédons de la même façon.

Lors de la réalisation de cette ACM, les premiers axes portent peu d'inertie ce qui implique que pour avoir 80 pourcent de l'inertie nous devons prendre les 44 premiers axes. Nous rajoutons ces 44 variables aux données déjà complété par les coordonnées de l'ACP.

En adaptant les dimensions d'entrées du réseau de neurones, lors de l'avancement du réseau nous apercevons un décrochage entre le score (RMSE) de la partie apprentissage et la partie validation. Ce décrochage est synonyme de sur-apprentissage, cela se valide sur le jeu de données test comme nous pouvons le voir ci-dessous.



Le modèle n'est plus adapté et l'information apportée par les axes de l'ACM est trop faible. Par conséquent on ne retient pas cette méthode.

Troisièmement nous procédons à la création de nouvelles variables par interaction en réalisant le produit de chaque variable quantitative et explicative avec les autres.

La création de nouvelles variable peut permettre d'utiliser le lien qu'il existe entre les variables. Nous agrémentons notre jeu de données de 600 nouvelles variables.

La limite de cette méthode apparaît rapidement lors de nos premiers réseaux, on note un fort décalage entre le score sur la partie *train* et la partie *valid*. Nous sommes clairement dans de **l'overfitting**.

AJOUTER PHOTO COURBE TRAIN VALID

En conclusion nous prenons la décision de ne garder que nos data.frame enrichi des coordonnées des 11 premiers axes de l'ACP.

6.5.3 Optimisation de notre meilleur modèle

Le modèle ayant obtenu le meilleur score possède des paramètres choisis arbitrairement. Nous allons donc essayer de faire varier différents paramètre en les incluant dans une grille. Pour effectuer le choix nous utiliserons une validation croisée *5-folds*. Nous incluons aussi un *early-stopping* afin de garder en mémoire le meilleur score lors de la descente de gradient et non la dernière itération comme précédemment.

On fait varier la *longueur* et la *largeur* du réseau.

```
dict_params = {'lst_neurals' : [[100,100,64],[300,300,100],[200,200,100,50]]}
```

L'algorithme choisi finalement le réseau initial $100 - 100 - 64$ par rapport à un modèle plus large ou plus profond. Pour conclure notre meilleur modèle est donc celui-ci :

REFAIRE TOURNER UN CODE

6.5.4 Conclusion sur le Deep Learning/Réseau de neurones (à mettre dans le BILAN??)

Le principal point positif des réseaux de neurones dans notre étude est le fait d'avoir réussi à atteindre nos meilleurs scores sur des réseaux simples sans avoir à effectuer beaucoup de transformations sur nos données. La première analyse d'un réseau est simple et visuel cependant l'effet **boite noire** apparaît rapidement, nous ne pouvons aujourd'hui expliquer par quelles raisonnement ces algorithmes nous permet d'avoir de bons scores. Nous savons ce qu'il se passe à l'entrée du réseau, connaissons la sortie du réseau mais le parcours entre les deux nous est inconnue.

Cet effet boite noire ne nous permet pas d'avoir de réel argument pour choisir nos paramètres à optimiser. **L'over-fitting** est toujours très proche de ces méthodes quand nous testons des réseaux plus complexes. Un point négatif supplémentaire est le côté "chronophage" de cette méthode.

7 Bilan

8 Bibliographie et sources

- **WikiStat**, *Scénario: Imputation de données manquantes*

- **Documentation de scikit-learn**, *Librairies pour faire de l'apprentissage supervisé en Python*

- **Jason Brownlee**, *A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning*