Master2 Probabilité et Statistiques des nouvelles données

cours de Machine Learning

Rapport de projet de Méthode statistiques pour megadonnées : compétition Kaggle "Bike Sharing Demand"

Alexandre CHEVAUX
Matthieu GARRIGUE

Enseignant :
Romuald ELIE

Année 2018-2019

Table des matières

Introduction **2**

I Présentation de la base de données **3**

La compétition Kaggle **3**

Les données **3**

Première analyse des données **4**

Feature Engineering **9**

II Description des méthodes d'apprentissage automatique utilisées **12**

L'algorithme des forêts aléatoires en régression **12**
Arbre de classification 12
Bagging (bootstrap aggregating) 12
Les forêts aléatoires 13

Modèle de mélange et Mélange Gaussien **14**
Modèle de mélange 14
Mélange Gaussien 14

Les réseaux de neurones **15**
Introduction 15
Vision globale du modèle 15
Travail en amont 16
Fonction d'activation 16
Explication du modèle 16
Exécution et optimisation 17
Paramètres 17
Optimiseur, métrique et fonction de perte 17
Entrainement du modèle 17
Analyse du modèle et création de prédiction 18
Analyse du modèle 18
Création de prédiction 18

III Résultats **19**

Outils pour présenter les résultats **19**

Validation croisée **20**

Score **20**

Kaggle **21**

Conclusion **23**

Bibliographie **24**

Introduction

Dans le cadre du cours d'apprentissage automatique, nous allons participer à la compétition Kaggle : "Forecast use a city bikeshare system". Celle-ci consiste à prédire le nombre de vélos qui seront loués à chaque heure et chaque jour en fonction de différents paramètres que nous préciserons ensuite. Ces vélos sont en libre service (même fonctionnement que les "Vélib" à Paris).

Dans ce rapport, nous étudierons dans un premier temps la base de données. Puis, nous effectuerons un premier traitement sur les données. Nous mettrons enfin en place 3 méthodes d'apprentissage automatique (que nous expliciterons) avant de présenter nos résultats.

Première partie

Présentation de la base de données

La compétition Kaggle

(Pour rappel le site Kaggle est une plateforme web organisant des compétitions en science des données. Sur cette plateforme, les entreprises proposent des problèmes en science des données. Les participants expérimentent avec différentes techniques et s'affrontent pour produire le meilleur modèle. Dans la plupart des compétitions, les observations sont notées immédiatement (la note se base sur leur valeur prédictive par rapport à un fichier de solution cachée) qui donne un classement en direct. (https ://fr.wikipedia.org/wiki/Kaggle))

Le but de cette compétition est de prédire le nombre de vélos qui seront loués en fonction de différents critères. C'est donc un problème de régression (différent des problèmes de classification ou de clustering). La compétition date de 2015 et était donc terminée quand nous y avons participé. Celle-ci fut lancée par la ville de Washington, D.C. C'est une compétition de type "Knowledge", c'est à dire qu'elle ne mettait pas en jeu de l'argent mais sert d'entrainement sur des problèmes de Machine Learning.

Nous sommes évalués ici en utilisant la formule de la "Root Mean Squared Logarithmic Error" :

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(p_i+1)-log(a_i+1))^2}$$

où a_i sont les vraies valeurs et p_i sont les valeurs prédites

Pour obtenir notre score, nous devons utiliser le fichier de test fourni qui sera comparé aux vraies valeurs connues par Kaggle.

Les données

Trois fichiers sont fournis par Kaggle : -un fichier d'exemple pour soumettre nos données

-un fichier de test

-un fichier d'apprentissage

Le fichier d'apprentissage comprend 12 colonnes.

L'index est composé de dates entre le 1 Janvier 2011 à minuit et le 19/12/2012 à 23h. Nous avons les données pour chaque jour de cette période à chaque heure de la journée. Les autres "features" :

1. **season** : la saison - 1 = printemps, 2 = été, 3 = automne, 4 = hiver (integer)
2. **holiday :** si c'est un jour de congé (booléen)
3. **workingday :** si c'est un jour travaillé (booléen)
4. **weather :** le type de météo (ensoleillé, pluvieux, nuageux...)
5. **temp :** températures (flottant)
6. **atemp :** températures ressenties (floattant)
7. **humidity :** pourcentage d'humidité (integer)
8. **windspeed :** vitesse du vent (flottant)
9. **casual :** nombre de vélos loués mais non réservés
10. **registered :** nombre de vélos loués et réservés
11. **count :** -nombre de vélos loués

Les colonnes casual et registered ne sont pas présentes dans le ficher de test (étant donné qu'elles sont directement liées aux nombre de vélos loués).

Le fichier de train possède 10886 entrées contre 6693 pour le fichier de test (soit 38% de données en test et 62% de données en apprentissage).

Les fichiers ne possèdent pas de données manquantes.

La variable à prédire est le nombre de vélos loués.

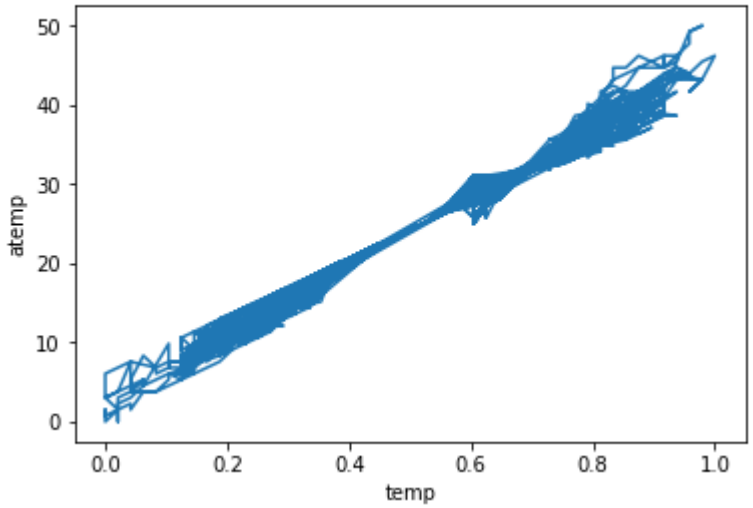
Première analyse des données

On étudie ici les liens entre les variables. Dans un 1er temps, nous allons chercher des corrélations entre les variables.

Sans grande surprise, on obtient une corrélation très proche de 1 entre les colonnes temp et atemp :

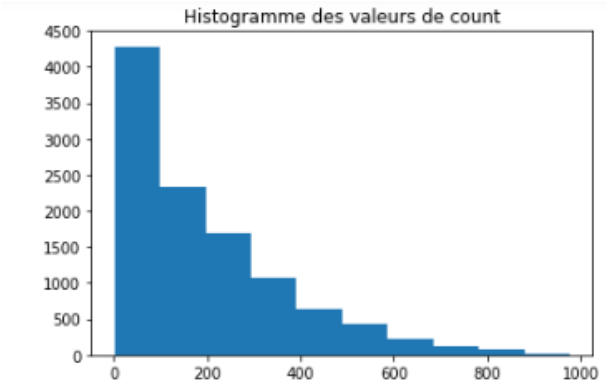
```
sns.pairplot(df_bike_train,vars=["temp", "atemp"])
df_bike_sharing["temp"].corr(df_bike_sharing["atemp"])
```

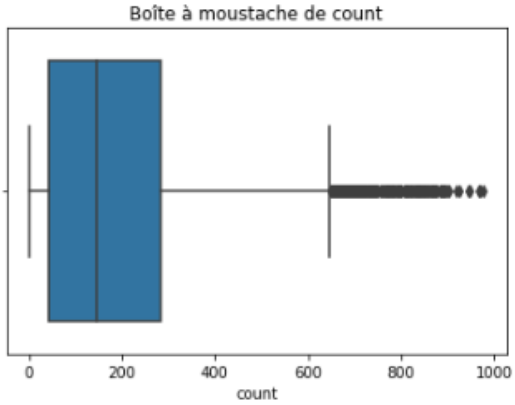
```
Corrélations entre temp et atemp:0.99
```



On peut donc supprimer une des deux variables comme elle n'apporte rien.

On effectue maintenant une analyse de la variable à prédire :
Lorsque l'on analyse cette variable, on constate que de grandes valeurs ont un fort impact sur le modèle.





Il est donc nécessaire d'effectuer une transformation des données. Ici, nous allons effectuer une transformation logarithmique afin de diminuer l'importance des données trop excentrées par rapport à la moyenne.

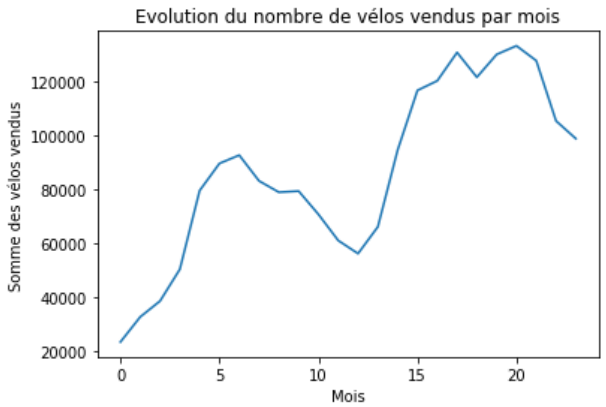
Code pour cela :

```
df_bike_sharing['count']=df_bike_sharing['count'].apply(lambda x:np.log(x))
```

Maintenant nous allons regarder si il y a une saisonnalité dans les données : Code :

```
df_bike_train['Year'] = pd.DatetimeIndex(df_bike_train["datetime"]).year
df_bike_train['Month'] = pd.DatetimeIndex(df_bike_train["datetime"]).month
x=df_bike_train.groupby(['Year','Month']).sum()
lst=[]
for i in x["count"]:
    lst.append(i)
```

Résultat :



On constate bien une forte saisonnalité (il y a beaucoup de locations les mois d'été comparé au mois d'hiver). Mais on constate aussi une forte augmentation des locations d'une année sur l'autre. Il faudra donc supprimer ce phénomène afin d'être plus précis.

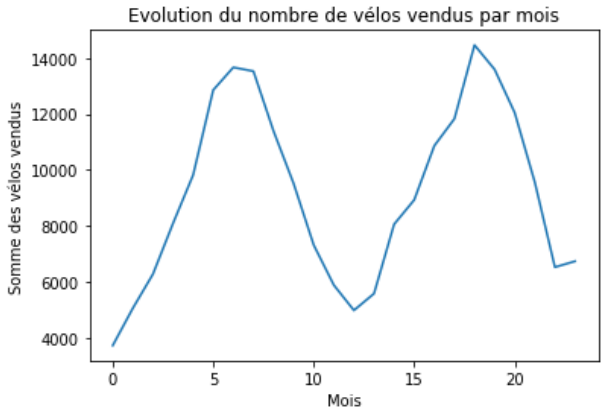
Ce phénomène se retrouve dans les températures qui sont par définition un phénomène saisonnier.

Nous allons donc enlever la saisonnalité dans cette variable.

On constate bien ici une saisonnalité annuelle sur les températures :

Code :

```
df_bike_train["nbr_day"]=1
df_bike_train['Year'] = pd.DatetimeIndex(df_bike_train["datetime"]).year
df_bike_train['Month'] = pd.DatetimeIndex(df_bike_train["datetime"]).month
x=df_bike_train.groupby(['Year','Month']).sum()
lst=[]
for i in x["temp"]:
    lst.append(i)
```



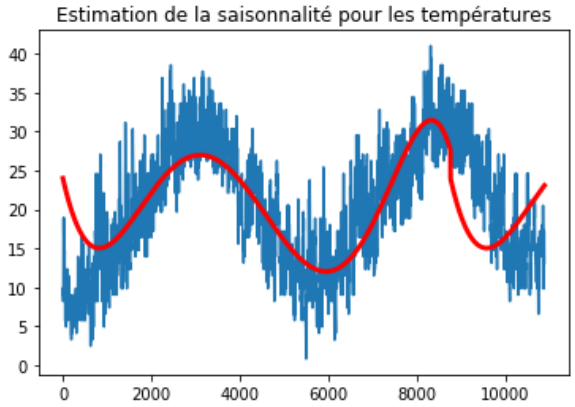
Pour éliminer, nous allons nous baser sur un modèle polynomial de Numpy (polyfit). Ici nous allons estimer nos valeurs avec un modèle polynomiale de type : $y = x^6 * b1 + x^5 * b2 + x^4 * b3 + x^3 * b4 + x^2 * b5 + x^1 * b6$ avec ici une équation de degré 6.

Ici on retrouve un polynôme de degré pair. Après quelques tests, nous allons prendre un polynôme de degré 6 pour estimer notre modèle.

Code :

```
from pandas import Series
from matplotlib import pyplot
from numpy import polyfit
series = Series(df_bike_train["temp"])
# fit polynomial: x^2*b1 + x*b2 + ... + bn
X = [i%(365*24) for i in range(0, len(series))]
y = series.values
degree = 6
coef = polyfit(X, y, degree)
print('Coefficients: %s' % coef)
# create curve
curve = list()
for i in range(len(X)):
    value = coef[-1]
    for d in range(degree):
        value += X[i]**(degree-d) * coef[d]
```

```
    curve.append(value)
```

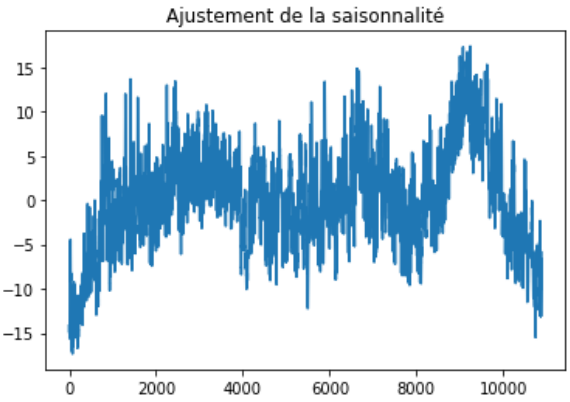


Résultat :

On va donc utiliser ce modèle pour créer une variable ajustée par rapport à la saisonnalité et remplacer la variable de temps par celle-ci.

Code :

```
series = Series(df_bike_train["temp"])
# fit polynomial: x^2*b1 + x*b2 + ... + bn
X = [i%(365*24) for i in range(0, len(series))]
y = series.values
degree = 6
coef = polyfit(X, y, degree)
print('Coefficients: %s' % coef)
# create curve
curve = list()
for i in range(len(X)):
        value = coef[-1]
        for d in range(degree):
                value += X[i]**(degree-d) * coef[d]
        curve.append(value)
# create seasonally adjusted
values = series.values
diff = list()
for i in range(len(values)):
        value = values[i] - curve[i]
        diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

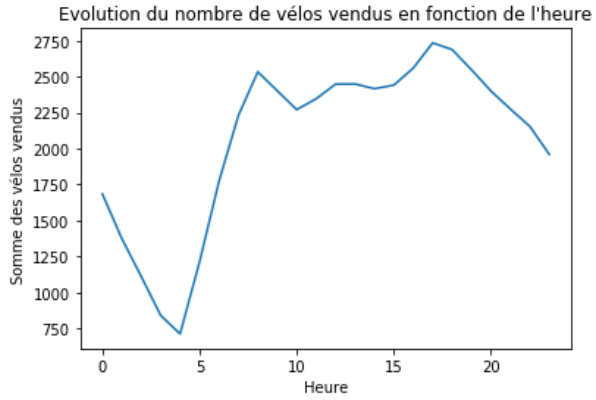


On constate tout de même un pic pour l'été 2011.

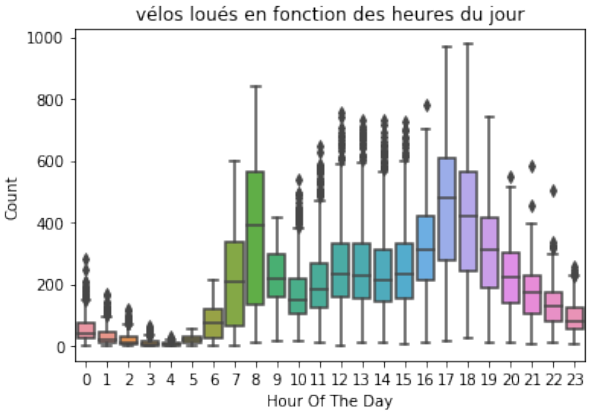
Nous allons aussi regarder l'évolution des locations en fonction de l'heure de la journée :

Code :

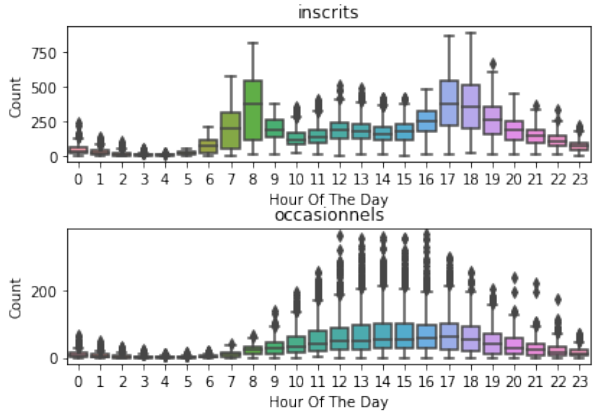
```
df_bike_train["nbr_day"]=1
df_bike_train['Hour'] = pd.DatetimeIndex(df_bike_train["datetime"]).hour
x=df_bike_train.groupby(['Hour']).sum()
lst=[]
for i in x["count"]:
    lst.append(i)
```



Nous pouvons également représenter cette évolution par un diagramme en moustache, en fonction des heures :



On constate de légers pics vers 8h et 17h, cependant pas de très fortes irrégularités pendant la journée ; on note en revanche une claire différence entre la journée et la nuit.
Observons par ailleurs, s'il est possible de distinguer une tendance selon le statut de l'individu, autrement dit s'il est enregistré ou si ce n'est qu'un usager occasionnel :



On constate que les utilisateurs occasionnels ont un usage réparti de façon plus uniforme au fil de la journée tandis que les utilisateurs enregistrés ont un usage plus irrégulier se concentrant sur les heures de forte affluence correspondant aux débuts et sorties de travail.

Ainsi, sachant que $registered + casual = count$ avec $count$ notre $target$, on peut envisager de diviser en deux la prédiction avec la partie $registered$ et la partie $casual$ pour obtenir un résultat plus précis avec nos modèles et donc adapter nos $features$ en fonction du statut du locataire. En effet, en plus de la répartition des heures de location qui n'est pas la même, des variables en liaison avec la météo n'auront également pas la même importance selon le statut du l'individu.

Feature Engineering

Afin de minimiser l'erreur de nos modèles, il est nécessaire de faire un travail préparatoire sur nos données. En effet en rajoutant des $features$, nos modèles seront plus précis (mais évidemment seulement si les $features$ rajoutés sont utiles).

Dans un premier temps, nous pouvons extraire de la date l'heure, le jour, le mois, l'année mais aussi le jour de la semaine. Ceci est fait avec les lignes suivantes en Python :

```
df_bike_sharing['Year'] = pd.DatetimeIndex(df_bike_sharing["datetime"]).
    year
df_bike_sharing['Month'] = pd.DatetimeIndex(df_bike_sharing["datetime"]).
    month
df_bike_sharing['Wday'] = pd.DatetimeIndex(df_bike_sharing["datetime"]).
    weekday
df_bike_sharing['Week'] = pd.DatetimeIndex(df_bike_sharing["datetime"]).
    week
df_bike_sharing['Day'] = pd.DatetimeIndex(df_bike_sharing["datetime"]).day
df_bike_sharing["Hour"]= pd.DatetimeIndex(df_bike_sharing["datetime"]).hour
```

Ensuite, on remplace les variable $wheater$, $holiday$ et $season$ par des indicatrices. On obtient donc 4 colonnes composées d'indicatrices. Cette méthode permet en effet d'améliorer le modèle.

Code :

```
df_bike_sharing = pd.get_dummies(df_bike_sharing, columns=['weather'])
df_bike_sharing = pd.get_dummies(df_bike_sharing, columns=['holiday'])
df_bike_sharing = pd.get_dummies(df_bike_sharing, columns=['season'])
```

Ensuite, nous normalisons nos données de température et de température ressentie par une méthode de normalisation $Min - Max$. Elle applique aux données la transformation suivante :

$Data'_{ik} = \frac{Data_{ik} - min(Data_i)}{min(Data_i) - max(Data_i)}$

avec $Data'_{ik}$ la donnée normalisée et $Data_i$ la ième colonne

De plus, comme vu précédemment, nous avons une corrélation entre les variables de température et de météo. Nous allons donc créer une nouvelle colonne qui sera le produit des deux :
Code :

```
df_bike_sharing['temp_weath_1'] = df_bike_sharing['temp'] * df_bike_sharing
    ['weather_1']
df_bike_sharing['temp_weath_2'] = df_bike_sharing['temp'] * df_bike_sharing
    ['weather_2']
df_bike_sharing['temp_weath_3'] = df_bike_sharing['temp'] * df_bike_sharing
    ['weather_3']
```

Par ailleurs, on se basant sur les observations faites plus haut par rapport à la fréquence d'utilisation des personnes enregistrées et des usagers occasionnels, nous avons également créé des variables `daypart`, l'une pour les individus enregistrés `daypart_registered` et l'autre pour les usagers occasionnels `daypart_casual`.
Code reprenant l'organisation du découpage des individus enregistrés :

```
#Pour les individus registered
 df_bike_sharing.loc[(df_bike_sharing['Hour']>=22) | (df_bike_sharing['
    Hour']==0),'daypart_registered']=0 #faible affluence
    df_bike_sharing.loc[(df_bike_sharing['Hour']>= 1) & (df_bike_sharing
        ['Hour']<=5),'daypart_registered']=1 #nuit, tres faible
        affluence
 df_bike_sharing.loc[(df_bike_sharing['Hour']==6)
                  | ((df_bike_sharing['Hour']>=10) & (df_bike_sharing['
                     Hour']<=15))
                  | (df_bike_sharing['Hour']==20)
                  | (df_bike_sharing['Hour']==21)
                 ,'daypart_registered']=2 # affluence moyenne
 df_bike_sharing.loc[(df_bike_sharing['Hour']==7),'daypart_registered'
    ]=3 #commencement rush
 df_bike_sharing.loc[(df_bike_sharing['Hour']==8),'daypart_registered'
    ]=4 #rush
 df_bike_sharing.loc[(df_bike_sharing['Hour']==17) | (df_bike_sharing['
    Hour']==18),'daypart_registered']=5 #heures de rush
 df_bike_sharing.loc[(df_bike_sharing['Hour']==9)
                  |(df_bike_sharing['Hour']==16)
                  | (df_bike_sharing['Hour']==19)
                  ,'daypart_registered']=6 #affulence importante
```

Code reprenant l'organisation du découpage des individus occasionnels :

```
# Pour les occasionnels
df_bike_sharing.loc[(df_bike_sharing['Hour']>=23) | (df_bike_sharing['
    Hour']<=7),'daypart_casual']=0 #nuit, affluence tres faible
df_bike_sharing.loc[(df_bike_sharing['Hour']== 8)
                    | (df_bike_sharing['Hour']==9)
                    | (df_bike_sharing['Hour']==21)
                    | (df_bike_sharing['Hour']==22)
                ,'daypart_casual']=1 #affluence faible
df_bike_sharing.loc[(df_bike_sharing['Hour']>=10) & (df_bike_sharing['
    Hour']<=20),'daypart_casual']=2 #affluence plus forte et reguliere
    sur le jour
```

A noter que nous utiliserons ces features seulement dans le cas où nous décomposerons la prédiction en deux parties avec deux `target` : *registered* et *casual* tel que $count = registered + casual$.

Deuxième partie

Description des méthodes d'apprentissage automatique utilisées

L'algorithme des forêts aléatoires en régression

Arbre de classification

C'est une méthode non paramétrique, qui est assez facilement interprétable car propice à la décision. L'idée est de construire un arbre binaire de classification. Il est défini par découpages récursifs des régions rectangulaires. On obtient R_n régions qui correspondent aux feuilles de l'arbre. La règle en régression est la suivante :

la moyenne des éléments de l'ensemble R_i auquel appartient notre donnée à prédire est

$y_n = \sum_{i=1}^{n} \frac{1}{Card(R_i)} y_i * \mathbb{1}_{(x_i \in R_i)}$

Pour la construction de l'arbre deux questions se posent :

-comment couper ?

-à quelle profondeur s'arrêter ?

En effet, un arbre profond peut sembler bien adapté aux domaines à disposition mais peut être trop complexe (problème de variance). A l'inverse un arbre très peu profond peut être trop grossier et aboutir à une régression non concluante.

On utilise alors l'algorithme *CART* (*classification and regression trees*) qui choisit à chaque niveau, en commençant par la racine, les variables pour lesquelles le partitionnement a lieu. A chaque étape, la division choisie est celle qui minimise l'erreur. La stratégie pour la profondeur consiste à construire d'abord un grand arbre puis à l'élaguer de façon à tenir compte du fait qu'une division paraisse plutôt mauvaise à priori mais à une étape suivante peut devenir intéressante.

Bagging (bootstrap aggregating)

En construisant un arbre de classification ,par essence, la variance est élevée. En effet, chaque division continue à exercer son influence à tous les niveaux suivants et donc une petite perturbation des données entraine une série de divisions assez différentes et donc une régression différente. Il est donc nécessaire de réduire cette variance.

La technique du bootstrap est un outil de rééchantillonnage. Un échantillon bootstrap de taille m des observations initiales est un ensemble de données $D_m = \{(X_1^{'}, Y_1^{'}), ..., (X_m^{'}, Y_m^{'})\}$
où les $(X_i^{'}, Y_i^{'})$ sont obtenus par tirage aléatoire uniforme avec remise à partir de $D_m = \{(X_1, Y_1), ..., (X_m, Y_m)\}$

Le principe du bagging est une moyenne des prévisions d'arbres de classification sur une collection d'échantillons bootstrap.

Plus précisément, nous prenons B échantillons bootstrap D_m^b avec $b = 1...B$ et on construit un arbre $\hat{f}_b$ associé à chacun.

Pour une nouvelle entrée x, on considère la moyenne des arbres. Pour une régression bagging, on prend la moyenne de chaque échantillon bootstrap soit :

$\hat{f}^{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x)$

Les forêts aléatoires

L'algorithme des forêts aléatoires est une version modifiée du bagging avec des arbres moins corrélés. L'idée est d'améliorer la réduction de variance en réduisant la corrélation entre les arbres. Quand on construit un arbre, on sélectionne aléatoirement un sous-ensemble de p variables parmi les d variables d'entrées qui deviennent candidates pour une division.

On va donc ,dans un 1er temps, tirer aléatoirement m-échantillon bootstrap D_m^b à partir de D_n. Ensuite on construit un arbre $\hat{f}^b sur D_m^b$. Cette étape est faite en choisissant p variables parmi d au hasard. Puis on sélectionne la meilleure division parmi ces p variables. Et on itère B fois cette procédure.

Enfin on moyenne ces B arbres afin d'obtenir une bonne régression. (Si nous étions dans un problème de classification, on procéderait par vote à la majorité.)

L'inconvénient est une perte d'interprétabilité.

Pour l'implémentation du code, nous avons utilisé la library sklearn.

Code :

```
import sklearn
from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics
RFR= RandomForestRegressor()
RFR.fit(X_train,Y_train)
score=cross_val_score(RFR, X, Y, cv=5, scoring='mean_squared_error')
```

(Nous détaillerons ensuite la partie de cross validation.)

Modèle de mélange et Mélange Gaussien

Modèle de mélange

A la différence des forêts aléatoires, c'est une méthode paramétrique. Soit $X_1, ..., X_n$ copies indépendantes d'une v.a. X à valeurs dans $\mathbb{R}$ de densité f (qui suit un modèle de mélange fini).

En fait, un modèle de mélange fini de la loi de probabilité consiste à modéliser des données provenant de plusieurs sous-populations. La population totale étant un "mélange" de ces sous-populations.

C'est à la fois un outil de classification/régression mais aussi d'estimation de densité.

Un modèle de mélange s'écrit : $f(x, \eta) = \sum_{k=1}^{K} p_k f_k(x)$ où les f_k sont les densités des différentes composantes du mélange. Les p_k sont les proportions associées (les poids).

Sur les p_k, on a : $p_i > 0$ et $\sum_{k=1}^{K} p_k = 1$

La loi mélange peut être vu comme la loi marginale de X obtenue à partir de (X,Z) où Z suit une loi multinomiale de paramètres $(1, p_1, ..., p_k)$ et X|Z=z suit la loi de densité fk : $z_k = 1$ avec $z_1, ..., z_k$ un vecteur contenant un 1 et des 0. Cette interprétation permet de générer un échantillon $X_1, ...X_n$ de la loi mélange.

Mélange gaussien

On considère ici que les densités cherchées sont Gaussiennes : $f_k = \Phi(x, \mu_k, \Sigma_k)$, densité d'une loi gaussienne d-dimensionnelle de moyenne μ_k
et de covariance Σ_k.

Ainsi $f(x, \eta) = \sum_{k=1}^{K} p_k \phi(x, \mu_k, \Sigma_k)$, avec Φ densité d'une Gaussienne de moyenne μ et de covariance Σ

$$\Phi(x, \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} (\det(\Sigma)^{\frac{1}{2}}} \times \exp\left[\frac{-1}{2}(x-\mu)\Sigma^{-1}(x-\mu)\right]$$

Le vecteur des paramètres est alors : $\Theta = (p_1, ..., p_{k-1}, \mu_1, ..., \mu_k, \Sigma_1, ..., \Sigma_k$

De plus, Σ_k est symétrique réelle donc diagonalisable dans une base orthonormée. Elle peut s'écrire $\Sigma_k = v_k p_k D_k^t p_k$, où $v_k = det(\Sigma_k)^{\frac{1}{d}}$ D_k est la matrice diagonalisable des valeurs propres normalisées de Σ_k rangées par ordre décroissant avec $det(D_k) = 1$, et p_k est une matrice orthogonale contenant les vecteurs propres de Σ_k.

Ces éléments permettent de caractériser les propriétés géométriques des composantes :

-le volume est donné par v_k

-l'orientation par p_k

-la forme par D_k

Pour se donner un modèle de mélange, il suffit de préciser pour chacune des caractéristiques si elle est supposée identique pour toutes les composantes du mélange ou est autorisée à varier entre les différentes composantes.

Là encore, pour la vise en place de l'algorithme, nous allons utiliser la bibliothèque sklearn :

```
from sklearn.mixture import GaussianMixture
GaussianMixture = GaussianMixture()
GaussianMixture .fit(X_train[features],Y_train)
pred = lR.predict(X_test[features])
```

Les réseaux de neurones

Introduction

Le modèle de réseau de neurones est un modèle prédictif très en vogue actuellement et qui doit entre autres sa popularité à l'émergence du big data ces dernières années, qui offre des infrastructures permettant d'exploiter leur potentiel. Leur système de fonctionnement s'inspire de celui du cerveau humain (d'où l'origine de leur nom). De façon succincte, ils s'organisent sous forme de couches composées de "neurones" contenant une information que l'on pourra noter x. Chacun de ces neurones a une sortie qui va recevoir la somme de plusieurs neurones pondérés par un poids w et auxquelles on va appliquer une fonction d'activation (dont on parlera plus tard) et un biais w_0. Si une sortie est connectée par p neurones x de poids w_p, une fonction d'activation a et un biais w_0, nous avons une valeur de sortie de la forme $a\left(\sum_{i=1}^{p} w_i x_i + w_0\right)$

Il existe plusieurs bibliothèques en Python pour les implémenter dont `tensorflow` ou bien `pytorch`. Dans le cadre de ce projet, nous utiliserons la bibliothèque `Keras` qui est en réalité une "sous-bilbiothèque" de `tensorflow` et qui permet de créer des modèles de réseaux de neurones intuitivement, efficacement, de façon modulable (et relativement simple).

Vision globale du modèle

Résumé du modèle :

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 200)               4800
_________________________________________________________________
dropout_3 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_5 (Dense)              (None, 200)               40200
_________________________________________________________________
dropout_4 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 201
=================================================================
Total params: 45,201
Trainable params: 45,201
Non-trainable params: 0
_________________________________________________________________
```

Code d'implémentation du modèle :

```
from keras.models import Sequential

from keras.layers import Dense,Dropout,Activation
from keras import regularizers
# Initialisation du modele

##Define neural network
model = Sequential()
model.add(Dense(200, activation='relu',input_dim=X_trainNN.shape[1])) #
    input layer
model.add(Dropout(0.1))
model.add(Dense(200, activation='relu')) #hidden layer
model.add(Dropout(0.1))
model.add(Dense(1))#outputlayer
```

Travail en amont

Avant d'appliquer un modèle de réseau de neurones, nous avons fait une sélection des *features* qui seront utiles pour notre modèle prédictif.
Par ailleurs, nous faisons un travail de normalisation sur les données notamment pour les variables continues tel que l'on ait une moyenne nulle et une variance de 1. Cela permet un entrainement plus simple du réseau de neurones et empêche les *features* avec des valeurs trop importantes d'avoir un impact trop élevé sur le modèle.
Pour la normalisation, nous avons utilisé la méthode du *minimax* explicitée dans la première partie du rapport.

Code pour la normalisation :

```
from sklearn.preprocessing import StandardScaler

# Methode choisie pour normaliser: MinMax
scaler = preprocessing.MinMaxScaler()

# Normalise le train set
X_trainNN = scaler.fit_transform(X_trainNN)

# Normalise le validation set
X_validNN = scaler.fit_transform(X_validNN)

# Normalise le test set
X_testNN = scaler.fit_transform(X_testNN)
```

Fonction d'activation

Notre modèle de réseau étant un modèle multi-couche qui est le plus adapté dans le cadre de ce challenge. La fonction d'activation la plus régulièrement associée à ces réseaux de neurones et qui donne les résultats les plus intéressants dans notre cas est la fonction **ReLU** (Rectifier Linear Unit).
Cette dernière est particulièrement efficace dans les modèles avec des couches cachées (*hidden layers*) comme ici. Elle s'exprime de la façon suivante : ReLU(x) =max$(0, x)$ et permet ainsi de remplacer les valeurs négatives par des 0 (ce qui peut en faire aussi sa faiblesse). Mais elle permet donc de faciliter la convergence du réseau et empêcher la saturation. D'autres fonctions d'activation populaires telles que **Sigmoid**$(\sigma(x) = \frac{1}{1+e^{-x}})$ sont parfois plus adaptées mais cette dernière donne un poids important aux petites valeurs et cause la saturation de certains neurones.

Explication du modèle

On construit notre modèle sous forme de "pile linéaire de couches" avec `Sequential()`. C'est alors à nous de définir nos couches. Ici, nous avons défini 3 couches : une "input layer", une "hidden layer" et une "output layer" ceci à partir de la fonction `model.add()` et `Dense()`, ce dernier signifiant que ce sont des couches pleinement interconnectées, avec des inputs de taille 200 dans notre modèle. Cette taille a été déterminée par "*tuning*", autrement dit en essayant diverses combinaisons de paramètres et regardant celle qui nous donne la plus faible *loss*. La couche finale prenant la valeur 1, représentant le résultat renvoyé.
Par ailleurs, nous appliquons un `Dropout` sur les couches cachées avec un rate de 10%. Il s'agit d'une technique de régularisation qui consiste à ignorer une certaine proportion de neurones (ici 10%) durant une session d'entrainement de neurones signifiant que la contribution de ces

neurones est temporairement suspendue, ceci permet ainsi d'avoir une prédiction plus généraliste et éviter l'*overfitting*.

Exécution et optimisation

Paramètres :

Le choix du nombre d'époques doit se faire de sorte à avoir une prédiction assez précise et en même temps suffisamment généraliste, c'est-à-dire qui ne fit pas trop les données et provoque de l'*overfitting*.
Le choix de "BATCH_SIZE" va déterminer le nombre de samples qui vont être propagés à travers le réseau, il va donc influer sur le nombre d'itérations que la machine va faire. Le choix de ce dernier est déterminant pour la vitesse de calcul mais aussi pour la précision de calcul de gradiant.

```
# Model parameters
BATCH_SIZE = 16
EPOCHS = 50
LEARNING_RATE = 0.001
```

Optimiseur, métrique et fonction de perte :

Avant d'entrainer le modèle, nous utilisons la fonction `compile` qui va permettre de définir la fonction de perte (ici nous avons choisi la fonction "mse" ou "Mean Squared Error", définie précédemment), un "optimizer" qui va avoir une action sur les poids (définis plus haut) pour minimiser la fonction de perte et une métrique pour mesurer l'erreur du modèle lorsque l'on va entrainer ce dernier après chaque époque.
Nous avons choisi l'*optimizer* "RMSprop" qui se rapproche beaucoup de la descente de gradiant, méthode connue permettant d'optimiser un paramètre (en calculant la pente sur la courbe de perte, et ajustant ce paramètre en fonction du résultat, jusqu'à obtenir un paramètre optimal). Ce qui le distingue de la descente de gradiant est la façon dont les gradiants sont calculés :

calcul de gradiant avec RMSprop : $v_{dw} = \beta.v_{dw} + (1-\beta).dw$

calcul de gradiant avec descente de gradiant :$v_{dw} = \beta.v_{dw} + (1-\beta).dw^2$

Dans notre modèle, l'optimisation des poids était légèrement meilleure avec le calcul de gradiant de RMSprop.
code python associé :

```
model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
```

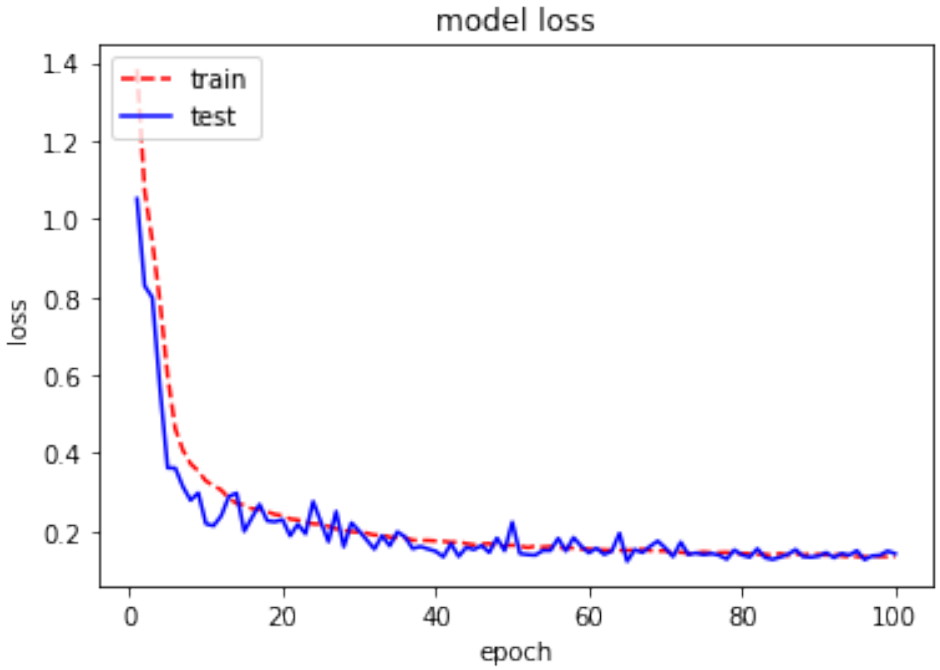
Entrainement du modèle :

Nous entrainons le modèle avec les paramètres que l'on a définis précédemment. Le réseau de neurones est entrainé en utilisant la backpropagation : pour chaque époque, un calcul de perte et d'erreur est fait (définies avec `compile`), voir section suivante pour les résultats.
Explication brève de la *backpropagation* :
Elle se base sur la décomposition de l'erreur grâce au théorème de dérivation des fonctions, on peut alors écrire la dérivée de l'erreur par rapport au poids de la façon suivante : $\frac{\partial erreur}{\partial w_{ij}} = \frac{\partial erreur}{\partial f(x)} \frac{\partial f(x)}{\partial z_i} \frac{\partial z_i}{\partial w_{ij}}$
Avec cette méthode, on a les poids définis plus haut qui sont adaptés après le calcul de gradiant dont on a également parlé précédemment.
A noter que nous entrainons notre modèle sur des données d'entrainement que nous avons splitées.

```
history = model.fit(x=X_trainNN, y=Y_trainNN, batch_size=BATCH_SIZE,
epochs=EPOCHS, verbose=1, validation_data=(X_validNN, Y_validNN),
                    shuffle=True)
```

Analyse du modèle et création de prédiction

Analyse du modèle :



On a bien une convergence de la fonction de perte (ainsi que d'autres fonctions d'erreurs non affichées ici) en fonction du nombre d'époques, ici pour 100 époques, ce qui est un des objectifs (avec aussi le fait d'éviter l'overfitting et l'underfitting).

Création de la prédiction :

Ci-dessous le code qui nous permet de générer une prédiction a partir du modèle prédéfini par tous les facteurs cités plus haut et entrainé avec ces divers facteurs. Nous appliquons ce modèle au dataset test de kaggle pour soumettre une prédiction par la suite.

```
prediction = model.predict(X_testNN, batch_size=BATCH_SIZE, verbose=1)
```

Troisième partie

Résultats

Outils pour présenter les résultats

A la différence des modèles de classification où pour évaluer un modèle on utilise entre autres les matrices de confusion, puis les courbes ROC et l'aire sous cette courbe (l'AUC ou BIC) mais ici ce c'est pas possible. En effet, nous sommes dans un modèle de régression. Nous allons donc mesurer l'erreur. Nous utiliserons des scores plutôt classiques et la RMSLE utilisée par Kaggle pour évaluer notre modèle :

-l'erreur quadratique moyenne (MSE) $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
-l'erreur quadratique moyenne absolue (MAE) $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2$
-l'erreur quadratique moyenne logarithmique (RMSLE) $\frac{1}{n}\sum_{i=1}^{n}(log(y_i+1) - log(\hat{y}_i+1)^2$

Pourquoi dans notre cas la RMSLE est-elle plus parlante que la MSE ou MAE ?

Par exemple, prenons deux exemples :

-le 08/12/2011 à 03 :00 :00 avec 1 vélo loué

-le 01/08/2012 à 18 :00 :00 avec 891 vélos loués

Calculons pour ces deux dates une erreur de prédiction de 10 vélos :

La MSE sera de 81 et de $(1-11)^2 = 100$ et de $(891-901)^2 = 100$ **alors qu'il est beaucoup plus grave dans notre cas de prévoir 11 vélos au lieu de 1 plutôt que 901 vélos au lieu de 891.**

La RMSLE permet de régler ce problème.

En effet, $(log(1+1) - log(11+1))^2 = 0.6$ et $(log(891+1) - log(901+1)^2 = 1.9 * 10^-5$

Code pour la RMSLE :

```
import math
#A function to calculate Root Mean Squared Logarithmic Error (RMSLE)
def rmsle(y, y_pred):
    assert len(y) == len(y_pred)
    terms_to_sum = [(math.log(y_pred[i] + 1) - math.log(y[i] + 1)) ** 2.0
        for i,pred in enumerate(y_pred)]
    return (sum(terms_to_sum) * (1.0/len(y))) ** 0.5
```

Code pour afficher les différents scores :

```
from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(Y_test, pred))
print('MSE:', metrics.mean_squared_error(Y_test, pred))
print('RMSE:', np.sqrt(metrics.mean_squared_error(Y_test, pred)))
print('RMSE:', np.sqrt(metrics.me(Y_test, pred)))
print('RMSLE:', rmsle(Y_test, pred))
```

Validation croisée

Si on divise nos données seulement en un échantillon d'apprentissage et de test (non utilisé pour l'apprentissage). On calibre l'erreur de la procédure sur notre jeu de données. Donc le calcul d'erreur est d'autant plus précis que l'échantillon de test est grand.

Mais dans la construction de la règle d'apprentissage, nous avons également intérêt à avoir un échantillon assez grand pour obtenir une règle performante. Ainsi le découpage échantillon test/apprentissage est intéressant seulement pour un jeu de données très grand (ce que nous n'avons pas ici).

L'idée de la validation croisée est de partager en K échantillons (K-blocs). On va alors donner le statut d'échantillon test à chaque blocs successivement (les autres devenant les bloc d'apprentissage.

Détail de la validation croisée :

1ère étape : Partition aléatoire des données en K sous ensembles $I_1, ..., I_k$ (Choix de K par l'utilisateur)

2ème étape : $\forall k = 1, .., K : D_K = \{(X_i, Y_i), i \notin I_k\}$ (ensemble d'apprentissage) $D_K = \{(X_i, Y_i), i \in I_k\}$=ensemble de validation

Soit $\{\lambda_i, ...\lambda_d\}$ un paramètre à calibrer

Pour chaque valeur de λ, on construit la méthode $\hat{f}_\lambda^k$ Et on calcule l'erreur sur D_k :

$e_k(\lambda) = \sum_{i \in I_k} \mathscr{L}(Y_i, \hat{f}_\lambda^k(X_i)$ avec $\mathscr{L}$ une fonction d'erreur (voir chapitre ci dessus)

3ème étape : $\forall \lambda$, on calcule l'erreur moyenne sur les K-blocs :

$CV(\lambda) = \frac{1}{n} \sum_{k=1}^{K} e_k(\lambda) = \frac{1}{n} \sum_{k=1}^{K} \sum_{i \in I_k} \mathscr{L}(Y_i, \hat{f}_\lambda^k(X_i)$

Code pour mettre en place une validation croisée :

```
from sklearn.model_selection import cross_val_score
score=cross_val_score(RFR, X, Y, cv=5, scoring='neg_mean_squared_error')
```

Score

Nous avons comparé différents modèles avec les MSE, MAE et surtout RMSLE. Les modèles choisis étaient : Random Forest ,AdaBoost, Machine à vecteur de support, mélange Gaussien et réseau de neurones.

Code pour obtenir les scores :

```
from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics
RFR= RandomForestRegressor()
model=RFR
#score=cross_val_score(RFR, X, Y, cv=5, scoring='mean_squared_error')
lst_mse_score.append(format_e(cross_val_score(model, X, Y, cv=5, scoring='
    neg_mean_squared_error').mean()))
lst_mae_score.append(format_e(cross_val_score(model, X, Y, cv=5, scoring='
    neg_median_absolute_error').mean()))
lst_rmsle_score.append(format_e(cross_val_score(model, X, Y, cv=5, scoring=
    'neg_mean_squared_log_error').mean()))
```

Code pour obtenir les scores des neural networks :

```
from sklearn import metrics
predic= model.predict(X_validNN, batch_size=BATCH_SIZE, verbose=1)
mseNN = metrics.mean_squared_error(Y_validNN, predic)
maeNN = metrics.mean_absolute_error(Y_validNN, predic)
rmsleNN = np.sqrt(metrics.mean_squared_log_error(Y_validNN, predic))
```

Code pour affichage du tableau :

```
from prettytable import PrettyTable
t = PrettyTable(['  ','RandomForestRegressor','SVM Regressor','
    GaussianMixture','AdaBoostRegressor', 'NeuralNetworks']])
t.add_row(lst_mse_score)
t.add_row(lst_mae_score)
t.add_row(lst_rmsle_score)
print(t)
```

Résultats :

	RandomForestRegressor	SVM Regressor	GaussianMixture	AdaBoostRegressor	NeuralNetworks
MSE	5.47E-06	1.96E+00	2.29E+01	3.52E-02	0.15
MAE	7.99E-16	8.46E-01	5.0E+00	1.21E-01	0.29
RMSLE	9.3E-08	1.21E-01	2.89E+00	2.69E-03	0.12

On constate bien que les Random Forest est l'algorithme qui minimise l'erreur, nous allons donc l'utiliser sur Kaggle.

Kaggle

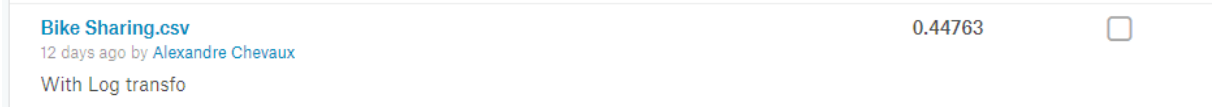
Nous avons ensuite testé nos résultats avec le dataset de test fourni par Kaggle. Celui-ci est un fichier indépendant ne comprenant pas les valeurs pour registered, casual et évidement de count. Ensuite on enregistre nos prédictions dans un CSV avec les dates et nos prédictions.

Code pour soumettre sur Kaggle :

```
RFR= RandomForestRegressor()
RFR.fit(X_train[features],Y_train)
pred = RFR.predict(X_test[features])
print(len(pred))
submission = pd.DataFrame({'datetime':df_bike_test['datetime'],'count':pred
    })
#submission["count"] = submission["count"].astype(int)
#print(submission)
submission["count"]=np.exp(submission["count"])
cpt_row=0
for row_submisson in submission["count"]:
        #print(submission.loc[[cpt_row], ['count']])
    #submission.loc[[cpt_row], ['count']]=submission.loc[[cpt_row], ['count
        ']]
    if row_submisson<0 :
            submission.loc[[cpt_row], ['count']]=0
```

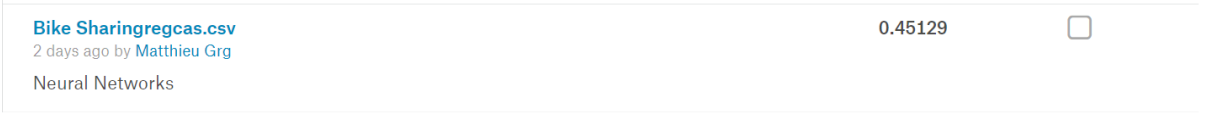
```
    cpt_row=cpt_row+1

filename = 'Bike Sharing.csv'
submission.to_csv(filename,index=False)
print('Saved file: ' + filename)
```

Nous avons obtenu une RMSLE au minimum de 12 days ago by Alexandre Chevaux 0.44763.



Pendant la compétition, cela nous aurait classés à la 844 place sur 3251 soit dans le top 26%

Score Kaggle avec réseaux de neurones :



Conclusion

Ainsi, nous avons traité le sujet Kaggle 'Bike Sharing Demand'. Celui est un sujet de régression qui consiste à prédire le nombre de vélos loués en fonction du jour et de l'heure. Dans un premier temps, nous avons fait une analyse des différentes variables afin de voir l'intérêt qu'elles présentaient et de les rendre plus pertinentes.

Ensuite, nous avons effectué un travail sur les données (Feature Engineering). Nous avons par exemple récupéré des informations de la date, groupé certaines informations, effectué une transformation logarithmique de la sortie, enlevé un paramètre de saisonnalité dans la variable de température...

Trois méthodes d'apprentissage automatique ont ensuite été présentées : les forêts aléatoires, le mélange gaussien et les réseaux de neurones. Cette présentation était à la fois théorique mais nous avons également explicité le code pour les mettre en place.

Enfin, les méthodes d'évaluation des modèles de régression ont été abordés afin de comprendre les résultats que nous avons obtenus.

Nous avons obtenu un RMSLE de 0.44763 pour le dataset de test Kaggle ce qui nous classe parmi les 26 premiers pour cent.

Bibliographie

-Cours d'Aurélie Fisher de Data Mining du master ISIFAR (aurelie.fischer-at-univ-paris-diderot.fr)

-https ://openclassrooms.com/fr/courses/4297211-evaluez-et-ameliorez-les-performances-dun-modele-de-machine-learning

-https ://machinelearningmastery.com/time-series-seasonality-with-python/

- Résumé et idées de travail sur le projet programmé en R : https ://www.analyticsvidhya.com/blog/2015/06/soluti kaggle-competition-bike-sharing-demand/

- https ://www.datacamp.com/community/tutorials/deep-learning-python

- https ://openclassrooms.com/fr/courses/4470406-utilisez-des-modeles-supervises-non-lineaires/4732186-empilez-les-perceptrons

- https ://machinelearningmastery.com/