Vos premiers pas en Machine Learning

“ The goal is to turn data into information, and information into insight. ” Carly Fiorina

Thomas OUNNAS,
Data scientist chez Xebia

Yoann BENOIT,
Data scientist chez Xebia

Au travers de cet article, nous vous proposons une initiation au Machine Learning, qui est un sous-ensemble de l'Intelligence Artificielle. Cette discipline permet l'analyse et la construction d'algorithmes capables d'apprendre à partir de données d'entrée. Grâce à deux Use Cases simples représentant deux cas de figure classiques, nous allons vous décrire la démarche de résolution d'un problème dans ce domaine via deux branches principales :

- L'apprentissage supervise,
- L'apprentissage non-supervise.

Sachant le résultat d'un phénomène, l'apprentissage supervisé permet de comprendre son mécanisme afin d'être capable de bien prédire les résultats à venir et de prendre des décisions pertinentes.
Par phénomène, on peut envisager plusieurs choses, par exemple si le fait qu'un e-mail soit un spam ou non, le prix du mètre carré dans une ville donnée, l'identification d'un code postal, la probabilité qu'un client achète tel ou tel produit, etc.
Derrière cette idée, on identifie une variable Y (prix, label spam/non-spam, etc.) représentant le résultat que l'on cherche à prédire et que l'on appelle la cible. On dispose aussi d'un ensemble de variables dites features, qui permet de capter et décrire le phénomène étudié.
L'apprentissage non-supervisé, quant à lui, ne suppose pas la connaissance d'une variable cible. L'idée n'est donc pas de comprendre ou prédire une cible précédemment identifiée mais plutôt de trouver une structure, un pattern dans les données. On parle alors de clustering : l'idée est de former des groupes de points homogènes, sachant qu'un point est défini par un ensemble de features. La manière de regrouper ces points reste à être définie et correspond au choix d'une métrique. Il est courant d'utiliser la distance euclidienne, mais il existe une multitude de choix possibles, souvent associés à des cas d'études précis.

A - Clustering pour la compression d'image

Le clustering est un type d'apprentissage non-supervisé permettant de trouver des patterns dans les données en les segmentant en plusieurs groupes homogènes. Nous avons donc à notre disposition un dataset composé de plusieurs features, sans target. Il s'agit alors, dans la plupart des algorithmes de clustering, de choisir le nombre de groupes K que l'on veut créer, et de rassembler les données en K groupes distincts. Les données au sein d'un même groupe doivent être proches, en relation avec la métrique choisie.
Un exemple d'utilisation du clustering consiste à compresser la taille d'une image, composée de plusieurs milliers de couleurs, en réduisant leur nombre à travers des regroupements. Typiquement, il peut y avoir plusieurs tons de rouge dans une image, le but est alors de repérer ces différents tons et de les regrouper en seulement quelques groupes de rouges (voir un seul groupe).
Attention, nous sommes loin ici des techniques de compression classiques utilisées dans le traitement d'image. Nous cherchons simplement à diminuer le nombre de couleurs dans une image, tout en gardant un aspect visuel correct, pour diminuer sa taille en mémoire.

Le Clustering en pratique :

Chargement d'une image

Chargeons maintenant une image pour mieux se rendre compte du problème.

```
from PIL import Image # Image loading
im = Image.open('perroquet.jpeg', 'r')
width, height = im.size
print width, height
```

L'image a pour dimensions 1024x768 pixels, elle comporte 238135 couleurs distinctes et sa taille est de 122 Ko au format .jpeg Fig.1.

Fig.1

Création du dataset en lien avec l'image

Pour réduire le nombre de couleurs, regroupons celles qui se ressemblent dans les 238135 présentes. Pour ce faire, il nous faut donc construire un dataset qui va représenter la couleur de chaque pixel. En utilisant la codification RGB, il nous faut donc trois colonnes dans notre dataset et autant de lignes qu'il y a de pixel, soit 1024*768 = 786432 lignes. Ce dataset se construit rapidement à partir de l'image chargée en appelant la fonction getdata() :

```
import numpy as np # Manipulate arrays
image_array = np.array(im.getdata()) / 255.
```

On obtient un tableau de la forme suivante :

R	G	B
0.266667	0.364706	0.756863
0.262745	0.360784	0.752941
0.282353	0.380392	0.772549

Notez que l'on a divisé les données du tableau par 255 simplement pour avoir des données comprises entre 0 et 1 (on parle alors de rescaling), car c'est un pré-requis de la fonction qui va permettre d'afficher les images par la suite.

Une méthode de clustering : K-Means

Nous allons maintenant pouvoir regrouper les couleurs proches grâce à une méthode de clustering appelée K-Means.
Avant de tenter de former des groupes, il est nécessaire de définir la notion de proximité entre des points, c'est-à-dire de se fixer une distance adaptée au sujet. Dans notre cas, la distance la plus intuitive à utiliser est la distance euclidienne. Deux points vont donc être considérés comme proches lorsque leur distance sera faible. En considérant deux points A(xA, yA, zA) et B(xB, yB, zB), elle est définie de la manière suivante :
L'algorithme du K-Means va chercher à construire des centroïdes pour chaque groupe déterminé. Ces centroïdes ne sont pas des points appartenant au dataset, mais des points fictifs représentant les barycentres de chaque groupe de points formé.
Le K-Means est un algorithme itératif, c'est à dire qu'en partant de centroïdes initialisés aléatoirement, on va mettre à jour à chaque itération leurs

coordonnées, comme représentées dans l'image ci-dessous : Fig.2.
L'image (a) montre les données de départ qui sont visiblement séparables en deux groupes distincts. Deux centroïdes vont alors être initialisés de façon aléatoire (figure (b)). Les points sont alors affectés à son centroïde le plus proche, formant des clusters (figure (c)). Les centroïdes se replacent en calculant la distance moyenne de tous les points qui lui sont affectés (figure (d)). Au fil des mises à jour, certains points seront certainement plus proches d'autres centroïdes, ce qui modifiera leur appartenance aux clusters (figure (e)).. Le processus de mise à jour des centroïdes et d'affectation des points dans les clusters est itéré plusieurs fois jusqu'à ce que les clusters ne changent plus (ou très peu). On obtient alors nos groupes de points finaux, comme montré sur la figure (f).

Clustering des couleurs de l'image

Maintenant que nous savons comment fonctionne l'algorithme du K-Means, appliquonsle sur le dataset représentant notre image initiale. Comme beaucoup d'autres algorithmes de Machine Learning, le K-Means est implémenté dans scikit-learn, et son utilisation devient très simple. Un seul paramètre reste à définir : le nombre de groupes souhaité, c'est-à-dire le nombre de couleurs que l'on va utiliser dans l'image.
L'implémentation se fait alors de la manière suivante :

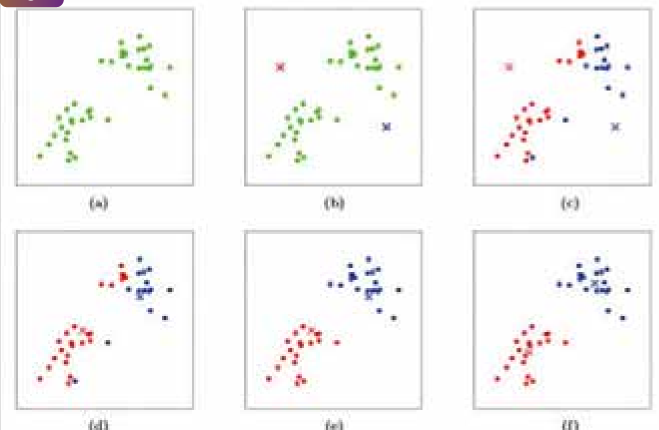
```
# Selection of the desired number of colors
n_colors = 64
# Taking a random sample of points of the data
image_array_sample = shuffle(image_array, random_state=0)[:10000]
# Fitting the K-Means
kmeans = KMeans(n_clusters=n_colors,random_state=0).fit(image_array_sample)
# Affecting each pixel to a cluster
labels = kmeans.predict(image_array)
```

Le paramètre random_state=0 n'est pas obligatoire et permet de régler l'initialisation aléatoire.

Reconstruction de l'image compressée

Nous avons alors à disposition les coordonnées RGB de chaque centroïde (présentes dans l'attribut *kmeans.cluster_centers_*), ainsi que le label du groupe dans lequel chaque pixel est affecté (présent dans labels).
A présent, nous devons reconstruire un tableau de même dimension que l'image originale, car pour le moment nous avons simplement un tableau de trois colonnes et d'autant de lignes que de pixels. Nous allons donc construire une fonction permettant de reconstruire l'image avec ces différentes contraintes. Elle a donc pour paramètres les centroïdes, les labels de chaque pixel, ainsi que la largeur et la hauteur de l'image.

Fig.2




```
def recreate_image(cluster_centers, labels, width, height):
  # Dimension of a center (3 in our case)
  d = cluster_centers.shape[1]
  # Initialization of the compressed image
  image_comp = np.zeros((height, width, d))
  # Initialization of a counter on the pixels' indexes
  label_idx = 0
  # Reconstruction of the image, pixel by pixel
  for i in range(height):
    for j in range(width):
      image_comp[i][j] = cluster_centers[labels[label_idx]]
      label_idx += 1
  return image_comp
```

Tout est maintenant à notre disposition pour reconstruire l'image avec un nombre de couleurs réduit.

```
# Recreation of the array with the initial image's dimensions
image_comp = recreate_image(kmeans.cluster_centers_, labels, width, height)

# Plotting the image
plt.figure(2)
plt.clf()
ax = plt.axes([0, 0, 1, 1])
plt.axis('off')
plt.title('Compressed Image (K=64)')
plt.imshow(image_comp);
```

Coco le perroquet, 64 couleurs

Fig.3

Voici le résultat obtenu en se restreignant à 64 couleurs sur l'image : Fig.3.
En se restreignant à 64 couleurs, l'image garde donc une bonne qualité. Le perroquet est bien reproduit grâce à des couleurs très marquées, à la différence du fond (mer et nuages) du fait de l'importance de la continuité des couleurs (du bleu clair au bleu foncé). De ce fait, en se restreignant à un plus petit nombre de couleurs, l'impression de continuité entre les couleurs est plus difficilement réalisable, d'où ces "coupures" entre chaque couleur.
Si on compare les tailles des images, l'image originale faisait 122 Ko, alors que l'image compressée réduite à 64 couleurs ne fait plus que 52 Ko.
Voici les résultats obtenus pour plusieurs K. On peut voir que notre oeil a rapidement l'impression que l'image est l'originale (à partir de 64 ou 128), alors que le nombre de couleurs est beaucoup plus faible Fig.4.
Nous avons ici illustré un algorithme de clustering dans le cadre d'une réduction de nombre de couleurs dans une image. Il y a cependant de nombreuses autres applications au clustering, comme par exemple chercher à regrouper des utilisateurs d'un site Internet selon leur comportement sur celui-ci, ou bien créer des zones pour optimiser les circuits de distribution d'une entreprise. Pour plus d'informations, se référer au tutoriel sur le même sujet sur le site de scikit-learn.

B - Régression pour la prédiction de location de vélos

Une application typique dans un contexte d'apprentissage supervisé consiste à faire de la prédiction (de ventes, de trafic, de pannes, etc.). Nous

vous proposons ici d'appliquer un modèle de régression pour prédire la demande de location de vélos dans une ville. Le but est alors de faire émerger une variable cible (notre target), qui n'est autre que le nombre de vélos loués dans une journée, en prenant en compte différentes variables (jour de la semaine, température, météo, etc.). Pour cela, nous allons appliquer une démarche systématique en Data Science :

- Exploration des données : compréhension et connaissance des données à disposition.
- Feature Engineering : création des features qui vont servir de base pour nos modèles.
- Splitting : séparation des données en trois jeux différents, un pour l'apprentissage (training set), un pour le tunning des paramètres (validation set) et le dernier pour l'évaluation du modèle (test set).
- Modelling: création et paramétrage d'un modèle de Machine Learning pour pouvoir réaliser la prédiction.

Chargement des données

Avant toute chose, nous allons charger les données en allant les chercher à leur source.

```
import pandas as pd
from StringIO import StringIO
from zipfile import ZipFile
from urllib import urlopen

url = urlopen("https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing-Dataset.zip")
zipfile = ZipFile(StringIO(url.read()))
data = pd.read_csv(zipfile.open("day.csv"), parse_dates = ['dteday'])
```

Voyons à quoi ressemblent les données pour mieux comprendre comment les traiter par la suite :

```
data.head()
```

Fig.4

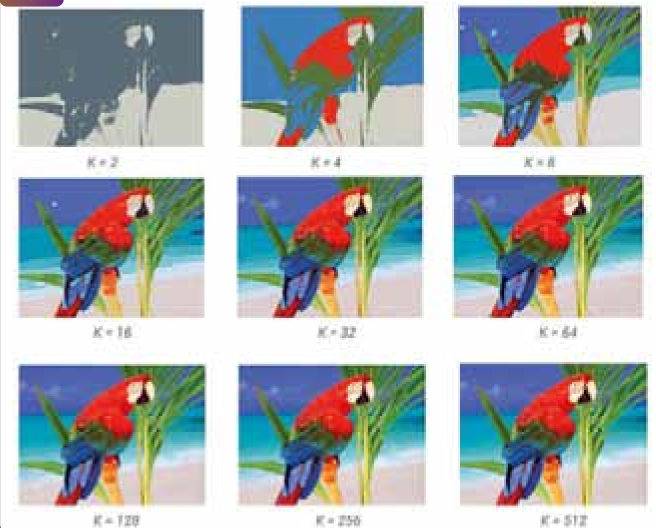


Fig.5. Plusieurs features sont à notre disposition pour l'analyse : certaines correspondent à la date (saison, année, mois, etc.), d'autres au temps (météo, température, humidité, etc.). Les trois dernières colonnes sont des variables cibles . "Casual" correspond au nombre de locations par des utilisateurs occasionnels, "Registered" au nombre de locations par des utilisateurs enregistrés et "Count" au nombre total de locations. Pour la suite de notre étude, nous allons nous concentrer sur les features season, *mnth, holiday, weekday, workingday, weathersit, temp, hum* et *windspeed*, et tenter de prédire la target casual. Restreignons donc notre dataset :

```
target = data.casual
list_features = ['season','mnth','holiday','weekday','workingday','weathersit','temp','hum','windspeed']
X = data[list_features]
```

Exploration des données et Feature Engineering

Deux types de variables sont à notre disposition :

- Des variables catégorielles : elles représentent un nombre fini de cas possibles, sans qu'il n'y ait forcément de relation d'ordre entre les états (typiquement les jours de la semaine)
- Des variables continues : le nombre de possibilités est infini et il existe une relation d'ordre entre les valeurs observées (typiquement la température)

Variables catégorielles

Observons à présent les différentes modalités des variables catégorielles à notre disposition.

```
print "Nombre de modalités de la variable %s: %d" % ('season',len(X['season'].unique()))
print "Nombre de modalités de la variable %s: %d" % ('mnth',len(X['mnth'].unique()))
print "Nombre de modalités de la variable %s: %d" % ('holiday',len(X['holiday'].unique()))
print "Nombre de modalités de la variable %s: %d" % ('weekday',len(X['weekday'].unique()))
print "Nombre de modalités de la variable %s: %d" % ('workingday',len(X['workingday'].unique()))
print "Nombre de modalités de la variable %s: %d" % ('weathersit',len(X['weathersit'].unique()))

Nombre de modalités de la variable season: 4
Nombre de modalités de la variable mnth: 12
Nombre de modalités de la variable holiday: 2
Nombre de modalités de la variable weekday: 7
Nombre de modalités de la variable workingday: 2
Nombre de modalités de la variable weathersit: 3
```

Comme on peut le constater, les variables holiday et workingday ne possèdent que deux modalités : c'est ce que l'on appelle des **dummies** (ou variables binaires). Ainsi, la variable workingday va prendre la valeur 1 s'il s'agit d'un jour ouvré en dehors des vacances, sinon la valeur 0.
Les autres variables possèdent plusieurs modalités, sans relation d'ordre entre elles : la valeur 1 de weekday n'est en aucun cas inférieure à la valeur 5. Il est donc nécessaire d'encoder ces variables de manière différente afin d'éviter une relation hiérarchique implicite entre les modalités. On parle

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331	654	985
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131	670	801
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120	1229	1349
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	0.212122	0.590435	0.160296	108	1454	1562
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	0.229270	0.436957	0.186900	82	1518	1600

Fig.5

alors de **dummisation** (ou d'encodage disjonctif complet) : c'est le procédé consistant à réécrire une variable catégorielle en plusieurs variables binaires (dummies). Par exemple, la variable season possède 4 modalités. Nous allons donc créer 4 nouvelles variables *season_1, season_2, season_3 et season_4* qui vaudront 0 ou 1 suivant la valeur considérée de *season*. Appliquons maintenant le processus de dummisation à toutes les variables catégorielles.

```
list_dummies = ['season', 'mnth', 'weekday', 'weathersit']
for v in list_dummies:
    dummies = pd.get_dummies(X[v], prefix=v+"_")
    X = pd.concat([X, dummies], axis=1)
    del X[v]
pd.set_option('max_columns', None)
X.head()
```

Fig.7.

	holiday	workingday	temp	hum	windspeed	season_1	season_2	season_3	season_4	mnth_1	mnth_2	m
0	0	0	0.344167	0.805833	0.160446	1	0	0	0	1	0	0
1	0	0	0.363478	0.696087	0.248539	1	0	0	0	1	0	0
2	0	1	0.196364	0.437273	0.248309	1	0	0	0	1	0	0
3	0	1	0.200000	0.590435	0.160296	1	0	0	0	1	0	0
4	0	1	0.226957	0.436957	0.186900	1	0	0	0	1	0	0

Variables continues

```
X[['temp','hum','windspeed']].describe()
```

Fig.8.

	temp	hum	windspeed
count	731.000000	731.000000	731.000000
mean	0.495385	0.627894	0.190486
std	0.183051	0.142429	0.077498
min	0.059130	0.000000	0.022392
25%	0.337083	0.520000	0.134950
50%	0.498333	0.626667	0.180975
75%	0.655417	0.730209	0.233214
max	0.861667	0.972500	0.507463

Regarder la description statistique globale des variables continues dans notre dataset est une étape importante car elle permet de l'explorer ou de le modifier lorsqu'il contient des valeurs manquantes ou des valeurs aberrantes. Cela permet aussi d'avoir une idée des ordres de grandeur des variables ainsi que de leur distribution.

Splitting

Ce concept consiste à séparer les données en au moins deux parties : une base d'apprentissage nommée **training set** et une deuxième qui servira à tester notre apprentissage sur des données nouvelles encore jamais observées, on parle de **test set**.
Il est aussi recommandé d'ajouter une troisième partie, appelée **validation set**; il va être utilisé pour valider un modèle et ses paramètres lors de sa construction (à noter qu'il existe d'autres méthodes d'évaluation qui ne nécessitent pas la présence de ce validation set, parmi lesquelles la cross-validation ou encore le principe du Bootstrap, mais dont l'explication sortirait du cadre de cet article). Le test set servira alors pour la validation finale du modèle, nous donnant ainsi une estimation fiable de notre erreur.
Construisons maintenant nos training, validation et test set :

```
import numpy as np
np.random.seed(seed=1234)
ind_train = np.random.choice(len(X),0.5*len(X),replace=False)
ind_val_test = list(set(range(len(X))) - set(ind_train))
ind_val = np.random.choice(ind_val_test,0.7*len(ind_val_test),replace=False)
ind_test = list(set(ind_val_test) - set(ind_val))

X_train, X_val, X_test, target_train, target_val, target_test = X.iloc[ind_train], X.iloc
[ind_val], X.iloc[ind_test], target.iloc[ind_train], target.iloc[ind_val], target.iloc[ind_test]
```

Modelling

Cette phase permet de construire un modèle de régression et de prédire la demande de vélos en fonction des features présentées précédemment. Le modèle étudié ici sera un arbre de décision, dont le principe est relativement intuitif et qui permet une bonne interprétation des résultats. Il se construit au fur et à mesure en se définissant des règles, qui vont soit nous donner un résultat, soit nous mener à une autre règle. Prenons un exemple : je prends mon lait le matin. Première chose, je regarde sa date de validité, si elle est bonne je peux le boire, sinon, je regarde s'il est périmé de moins d'une semaine. Si oui, je peux le boire, sinon, je l'ouvre et regarde son aspect pour savoir si je peux le boire ou non.
L'un des gros avantages de la librairie Scikit-Learn de Python est la simplicité d'utilisation des différentes méthodes de Machine Learning. Ainsi, une fois que les différents datasets sont construits, il devient relativement simple de passer à la phase d'apprentissage et de prédiction.

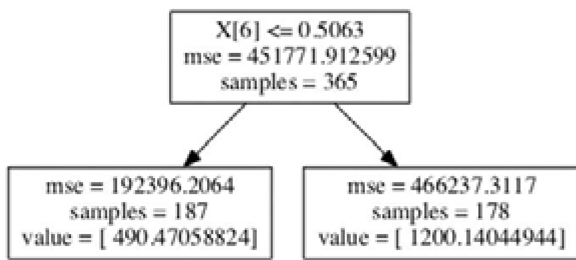
```
from sklearn.tree import DecisionTreeRegressor
model = DecisionTreeRegressor()
model.fit(X_train, target_train) # On fit le modèle sur le training set
prediction = model.predict(X_test) # On effectue les prédictions sur le test set
```

Une fois que les prédictions sont faites, il est important de pouvoir mesurer leur qualité. Pour cela, il faut avant tout se fixer une mesure de l'erreur. Dans notre cas, nous allons utiliser le **Root Mean Squared Error**. C'est une mesure classique de l'erreur faite par un modèle de régression. Nous allons donc préalablement définir la fonction de calcul de l'erreur pour l'appliquer ensuite aux prédictions faites sur le training set, ainsi que sur le test set.

```
def rmse(obs, pred):
    return np.sqrt(np.mean((obs-pred)**2))

rmse_train = rmse(target_train, model.predict(X_train))
rmse_test = rmse(target_test, prediction)

print "Train Error: %.2f" % rmse_train
print "Test Error: %.2f" % rmse_test
Train Error: 0.00
Test Error: 501.31
```

Remarque : *il existe déjà une implémentation du mse (sans la racine) dans sklearn mais nous avons pris le parti de le réécrire dans un but didactique.*

Avec un Train Error faible et un Test Error élevé, nous nous retrouvons dans une situation délicate appelée l'**overfitting**. Du fait qu'il n'existe aucune erreur sur le training set, l'overfitting peut être vu comme de l'apprentissage par coeur. L'arbre de décision, auquel nous n'avons donné aucune restriction dans les paramètres, a réussi à trouver des combinaisons de caractéristiques lui permettant à tous les coups de prédire exactement la bonne valeur sur les données qu'il a apprises. Le problème est que dès que de nouvelles données, encore jamais observées, sont soumises au modèle, il va très souvent faire de fortes erreurs, comme ont peut le constater sur le rmse du test set. Afin d'éviter ce phénomène, il faut faire ce que l'on appelle du **tunning** : trouver les paramètres à donner au modèle qui vont minimiser l'erreur sur des données non observées. Pour notre arbre de décision, l'un des paramètres sur lesquels il est possible de jouer est sa profondeur maximale, soit son nombre de règles. En effet, plus un arbre est profond, moins il y a de données dans ses feuilles et plus il y a de chance de faire de l'overfitting. En fixant par exemple la profondeur maximale à 1, nous obtenons l'arbre suivant :



Nous n'avons autorisé ici qu'une profondeur de 1 : l'arbre a donc été contraint de trouver la meilleure caractéristique et son seuil, ceci pour séparer en deux les données puis estimer la valeur à donner.

Appliquons maintenant une démarche complète de tunning qui va nous permettre de déterminer quelle est la profondeur optimale de l'arbre nous permettant de minimiser l'erreur sur le validation set.

```
scores = []
for max_depth in [1, 2, 3, 4, 5]:
    model = DecisionTreeRegressor(max_depth=max_depth)
    model.fit(X_train, target_train)
    pred = model.predict(X_val) # Prediction on the validation set
    scores.append((max_depth, rmse(target_val, pred)))
tab_scores = pd.DataFrame(scores, columns=['max_depth', 'rmse']).sort('rmse')
tab_scores
```

	max_depth	rmse
3	4	405.900402
2	3	414.233804
4	5	418.921402
1	2	465.677455
0	1	613.281642

Remarque: *encore une fois, il existe une fonction (gridSearch) déjà implémentée dans sklearn qui peut, à l'aide du paramètre n_jobs, paralléliser le tunning. Cependant nous avons pris le parti d'en écrire une version simplifiée afin de mieux illustrer le principe sous-jacent.*

Une fois le tunning fait, nous pouvons reconstruire un arbre de décision plus stable.

```
model = DecisionTreeRegressor(max_depth=tab_scores.max_depth.iloc[0])
model.fit(pd.concat([X_train, X_val]), pd.concat([target_train,target_val]))
prediction = model.predict(X_test) # Prediction on the test set

rmse_train = rmse(pd.concat([target_train,target_val]), model.predict(pd.concat([X_train, X_val])))
rmse_test = rmse(target_test, prediction)

print "Train Error: %.2f" % rmse_train
print "Test Error: %.2f" % rmse_test

Train Error: 319.07
Test Error: 383.05
```

Comme on peut le constater, l'erreur sur le train set a considérablement augmenté en comparaison avec la prédiction sans tunning. En revanche, l'erreur constatée sur le test set est largement inférieure, démontrant ainsi la plus grande capacité de généralisation de notre nouvel arbre. Il faut en effet accepter de ne pas pouvoir prédire à 100% les données sur lesquelles on a appris le modèle pour pouvoir avoir un score correct sur de nouvelles données. Nous sommes dorénavant capables grâce à ce modèle de faire des prédictions correctes sur le nombre de vélos qui vont être loués.

Pour aller plus loin

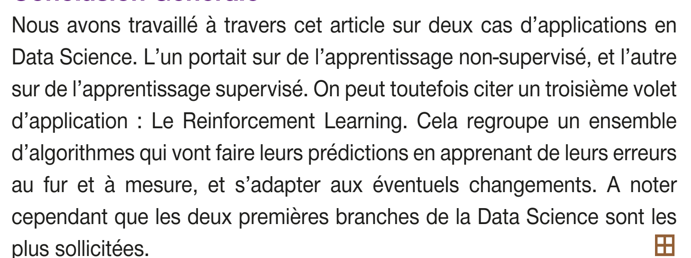
Maintenant que la démarche globale en Data Science est bien comprise, il est possible d'utiliser des modèles plus complexes afin d'améliorer les scores de prédiction sur le test set. Une évolution classique lorsque l'on utilise un arbre de décision est la **Random Forest**. Une Random Forest consiste à modéliser plusieurs arbres et à moyenner leurs résultats pour être plus robuste qu'un simple arbre de décision. Pour chaque arbre de la Random Forest, seulement une partie des variables vont lui être proposées, et les données sont sélectionnées aléatoirement avec remise. Cela permet de rajouter plus d'aléatoire dans les données et dans la sélection des features, ce qui est statistiquement plus viable dans le but d'avoir une meilleure capacité de généralisation. Encore une fois, il est nécessaire de faire un bon tunning pour éviter l'overfitting. Pour une Random Forest, il est classique d'optimiser la profondeur maximale de chaque arbre, ainsi que le nombre minimal d'éléments par feuille. C'est ce que nous allons faire maintenant avec une Random Forest composée de 2000 arbres.

```
from sklearn.ensemble import RandomForestRegressor
scores = []
for max_depth in [2, 4, 6]:
    for min_samples_leaf in [3, 5, 7]:
        model = RandomForestRegressor(n_estimators=2000, max_depth=max_depth, min_samples_leaf=min_samples_leaf)
        model.fit(X_train, target_train)
        pred = model.predict(X_val)
        scores.append((max_depth, min_samples_leaf, rmse(target_val, pred)))
tab_scores = pd.DataFrame(scores, columns=['max_depth','min_samples_leaf', 'rmse']).sort('rmse')
tab_scores
```

Une fois les meilleurs paramètres déterminés, nous pouvons finalement les utiliser pour construire notre modèle et faire nos prédictions. Maintenant que nous n'avons plus besoin de tester plusieurs jeux de paramètres, nous pouvons encore plus augmenter le nombre d'arbres, ce qui permet en général d'obtenir de meilleurs résultats.

```
max_depth_best = tab_scores.max_depth.iloc[0]
min_samples_leaf_best = tab_scores.min_samples_leaf.iloc[0]

model = RandomForestRegressor(n_estimators=5000, max_depth=max_depth_best, min_samples_leaf=min_samples_leaf_best)
model.fit(pd.concat([X_train, X_val]), pd.concat([target_train, target_val]))
prediction = model.predict(X_test)

rmse_train = rmse(pd.concat([target_train, target_val]), model.predict(pd.concat([X_train, X_val])))
rmse_test = rmse(target_test, prediction)

print "Train Error: %.2f" % rmse_train
print "Test Error: %.2f" % rmse_test

Train Error: 235.60
Test Error: 340.87
```

Comme on peut le voir, l'utilisation de modèles plus complexes tels qu'une Random Forest permet d'améliorer les scores de prédiction, à la fois sur le training set et sur le test set.



Conclusion Générale

Nous avons travaillé à travers cet article sur deux cas d'applications en Data Science. L'un portait sur de l'apprentissage non-supervisé, et l'autre sur de l'apprentissage supervisé. On peut toutefois citer un troisième volet d'application : Le Reinforcement Learning. Cela regroupe un ensemble d'algorithmes qui vont faire leurs prédictions en apprenant de leurs erreurs au fur et à mesure, et s'adapter aux éventuels changements. A noter cependant que les deux premières branches de la Data Science sont les plus sollicitées.