Python Data Tools

Python est un langage de programmation interprété et orienté objet de plus en plus populaire, en particulier dans le monde de la Data Science. Cet article introduit les bases de la programmation en Python pour des applications autour de la donnée ainsi que les principales librairies, NumPy, Pandas et Scikit-learn.

Yoann Benoit,
Data Scientist chez Xebia

Installer Python et ses principales librairies

Pour installer Python efficacement avec toutes les principales librairies nécessaires, le mieux est de s'orienter vers Anaconda, qui est une distribution gratuite de Python incluant près de 200 packages et contenant des installers à la fois pour Python 2.7 et 3.4.
En plus d'IDE comme Spyder ou même IntelliJ Idea, un IPython Notebook est disponible. Il permet, de manière interactive et rapide, l'exploration et la manipulation de données. Vous pouvez grâce à ce notebook combiner l'exécution de code, de textes, de formules mathématiques (LaTeX) et de graphes en un seul document. Pour le lancer, il suffit de taper les commandes suivantes dans votre console dans le bon répertoire suite à l'installation d'Anaconda :

```
>> ipython notebook
```

Le gestionnaire de notebooks devrait alors apparaître dans votre navigateur. Bien que les principales librairies soient déjà installées avec Anaconda, il est possible que vous ayez besoin d'installer un package qui ne soit pas encore présent. Pour ce faire, il faut utiliser la commande *conda* ou *pip* comme suit :

```
>> conda install myPackage
>> pip install myPackage
```

Comme expliqué en introduction, nous allons vous présenter au cours de cet article trois des librairies les plus importantes pour l'exploration, l'analyse et de traitement de données en Python. Les deux premières, que sont NumPy et Pandas, vont être utilisées principalement pour charger, explorer et préparer la donnée en vue d'analyses plus poussées futures. La librairie scikit-learn entre ensuite en jeu afin d'utiliser efficacement des algorithmes de Machine Learning sur les données préparées au préalable.

Le calcul scientifique avec NumPy

NumPy est le package essentiel pour le calcul scientifique (l'autre référence est SciPy) et l'analyse de données en Python. De nombreuses autres librairies de plus haut niveau, notamment pour le Machine Learning, sont basées sur ce package. Ses principales fonctionnalités sont les suivantes :

- Création et manipulation extrêmement performante d'arrays à n dimensions permettant des opérations arithmétiques vectorisées ;
- Opérations mathématiques rapides sur un tableau de données sans nécessité d'écrire des boucles ;
- Algèbre linéaire ;
- Possibilité d'intégration de code écrit en C, C++ et Fortran.

Le package NumPy ne fournit pas de fonctionnalités haut niveau pour l'analyse de données, celles-ci sont plutôt présentes dans les librairies qui sont basées dessus. Il est cependant fortement bénéfique de maîtriser ses concepts pour utiliser au mieux les autres outils.
Il est nécessaire d'importer au préalable la librairie pour avoir accès à ses fonctionnalités, souvent via une appellation donnée.

```
>>> import numpy as np
```

Création de tableaux multidimensionnels

L'objet ndarray (N-dimensional array) de NumPy permet d'avoir de larges tableaux de données, qui doivent être du même type, sur lesquels on peut effectuer de nombreuses opérations mathématiques. Il est très simple des les créer, tout comme d'effectuer des opérations :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]]) # Create a 2x3 Array
>>> type(a)  # ndarray
```

Pour accéder ou modifier un ou des éléments du tableau, il suffit de spécifier leur position entre crochets.

```
>>> print a[0,0]  # Print one element
>>> print a[:,1]  # Return the second column
>>> a[1,:] = [7,8,9]  # Replace second line elements by [7, 8, 9]
```

En créant un tableau de cette manière, si l'on ne spécifie pas de type pour les données, NumPy va tenter d'inférer le meilleur possible.
Il est de plus possible de créer des tableaux spéciaux : remplis de zéros, de uns ou bien de random.

```
zeros = np.zeros(5)  # One dimensional array of 0s
ones = np.ones(shape=(3, 4), dtype=np.int32) # 3x4 Array of 1s
random_array = np.random.randint(0, 10, (3, 4)) # 3x4 Array with random integers between 0 (included)
 and 10 (excluded)
```

D'autres fonctions permettent la création de ndarray, comme *arange, empty, eye* et d'autres.

Opérations sur des tableaux

L'avantage indéniable des tableaux NumPy est qu'ils permettent de faire des opérations de manière vectorisée, c'est-à-dire sans avoir à faire de boucle for.
Toutes les opérations arithmétiques sur des ndarray s'appliquent à tous les éléments.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 Array
>>> a*2 # Every element of the array is multiplied by 2
>>> a*a  # Element-wise multiplication (this is not a matricial operation)
```

De la même manière, de nombreuses fonctions mathématiques usuelles sont disponibles dans NumPy et applicables sur des ndarray.

```
>>> np.sqrt(a)  # Element-wise square root
```

```
>>> np.exp(a)  # Element-wise exponential value
```

Beaucoup d'autres fonctions sont disponibles, telles que abs, floor, cos, etc. Des fonctions statistiques sur les tableaux sont accessibles en tant que méthodes de ces derniers.

```
>>> a.sum()  # Sum of all elements of the array
>>> a.mean()  # Mean of all elements of the array
>>> a.sum(axis=0)  # Sum by column
>>> a.min(axis=1)  # Min by row
```

Si l'on veut faire de l'algèbre linéaire sur les tableaux, quelques fonctions sont disponibles.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 Array
>>> b = np.array([[1, 2], [3, 4], [5, 6]]) # 3x2 Array
>>> np.dot(a,b)  # Matrix multiplication
```

Comme expliqué précédemment, NumPy est donc une librairie extrêmement puissante pour le calcul scientifique, mais ne fournit pas de fonctionnalités haut niveau. Elle constitue le socle principal pour d'autres librairies avec une syntaxe plus simple pour la manipulation et l'exploitation de données.

La manipulation de données avec Pandas

NumPy est un outil efficace mais peu pratique pour manipuler des données hétérogènes. Dans la pratique, la librairie Pandas, basée comme beaucoup d'autres sur NumPy, est plus adaptée, car elle propose des structures de données haut niveau et de nombreux outils pour la manipulation efficace et rapide de données. Pandas possède les fonctionnalités suivantes :

- Des structures de données avec des axes labélisés ;
- Manipulation de séries temporelles ;
- Gestion efficace des données manquantes ;
- Opérations relationnelles semblables au SQL.

```
>>> import pandas as pd
```

Il existe deux principales structures de données dans Pandas, *Series* et *DataFrame,* permettant de très nombreuses manipulations de données.

Series

Une Series est un tableau à une dimension (≈ un vecteur) de variables labélisées de n'importe quel type (integers, strings, floating point numbers, Python objects, etc.). L'axe des labels d'une Series se nomme l'index. Pandas va créer un index par défaut lors de la création de la Series, bien que les labels puissent aussi être spécifiés.

```
>>> s1 = pd.Series([1,3,5,np.nan,6,8])  # A Series containing integers and one non-specified value (nan)
>>> s2 = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

Chose qui n'était pas possible de manière simple avec NumPy, les données sont ici accessibles aussi grâce à leurs labels.

```
>>> s2[['a', 'c']]  # Access values with labels 'a' and 'c'
```

DataFrame

Une DataFrame est une structure de données à 2 dimensions avec des colonnes labélisées, potentiellement de différents types. On peut voir un DataFrame comme une feuille de calcul, une table SQL, ou un dictionnaire (≈ une map) d'objets Series. Lorsque l'on manipule une DataFrame, les opérations sur les lignes et sur les colonnes sont traitées relativement de la même manière, la donnée étant structurée au sein de blocs à deux dimensions plutôt que grâce à des listes ou des dictionnaires. Il existe de nombreuses manières de créer un DataFrame. Une manière commune de le faire est à partir d'un dictionnaire de listes ou d'arrays NumPy.

```
>>> df = pd.DataFrame(np.random.randint(0, 10, (6,4)), columns=('A','B','C','D'))  # Create a DataFrame from a NumPy array
>>> df2 = pd.DataFrame({'state': ['Ohio', 'Ohio', 'Nevada', 'Nevada'],
  'year': [2000, 2002, 2001, 2002],
  'pop': [1.5, 3.6, 2.4, 2.9]})  # Create a DataFrame from a dictionary
>>> df2.dtypes  # Types of all columns
```

Il est aussi possible de créer un DataFrame directement en chargeant un fichier csv à l'aide de la méthode *pd.read_csv()*. Il faut alors éventuellement spécifier le séparateur et d'autres paramètres relatifs au fichier. D'autres formats de lecture sont aussi disponibles.
La méthode describe permet d'afficher un résumé des statistiques de vos données numériques (somme, moyenne, min, max, quartiles, etc.) :

```
>>> df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	3.666667	3.500000	4.000000	4.333333
std	2.732520	3.082207	3.898718	2.804758
min	1.000000	0.000000	1.000000	0.000000
25%	2.000000	1.250000	1.250000	2.750000
50%	2.500000	3.000000	2.000000	5.000000
75%	5.250000	5.500000	7.250000	6.500000
max	8.000000	8.000000	9.000000	7.000000

De nombreuses méthodes permettent de récupérer l'index, les colonnes ou les données sous différentes formes

```
>>> print df.index  # Return all indexes
>>> print df.columns  # Return columns names
>>> print df.values  # Values as a NumPy array
>>> df['A']  # Series corresponding to the column labeled 'A'
>>> df.A  # Same as above
>>> df[['A','C']]  # DataFrame corresponding to the columns labeled 'A' and 'C'
```

La sélection de données peut aussi se faire par label, position ou indexation booléenne.

```
>>> df.iloc[3]  # Fourth line
>>> df.iloc[1,1]  # Scalar element at line 2, column 2
>>> df[df.A > 5]  # New DataFrame generated from boolean values
```

Les fonctions NumPy comme *min*, *max* ou *abs* fonctionnent aussi sur des DataFrame. Il est, de plus, possible d'utiliser la methode *apply* pour appliquer une fonction sur des arrays à une dimension. Prenons l'exemple suivant, qui calcule pour chaque colonne une valeur correspondant à la différence entre leur max et leur min. On peut pour

cela faire appel à des fonctions lambda, qui sont très utilisées en programmation fonctionnelle:

```
>>> f = lambda x: x.max() - x.min()  # Lambda function subtracting max and min of a column
>>> df.apply(f, axis=0)  # Apply f column-wise
```

Les DataFrames permettent d'explorer les données avec une syntaxe très simple et via de nombreuses méthodes utiles. Si vous possédez par exemple une colonne avec des données catégorielles, il est possible de compter le nombre d'occurrences de chaque catégorie grâce à la méthode *value_counts()*

```
>>> letters = pd.Series(['a', 'b', 'c', 'c', 'a', 'c', 'b', 'a', 'b', 'a', 'c', 'c', 'c', 'b', 'a'])
>>> letters.value_counts()
```

Lorsque des données sont manquantes, Pandas les remplace par des NaN (Not a Number). Il est alors toujours possible de faire des calculs en omettant les NaN. On peut filtrer les lignes avec des NaN en utilisant la méthode *dropna()*, ou bien remplacer les NaN par des valeurs (0 par exemple) en utilisant *fillna(0)*.
D'autres fonctionnalités sont disponibles avec Pandas, comme l'indexing hiérarchique, le groupement de deux DataFrame à l'aide de *groupby* ou *concat* ou encore l'utilisation de méthodes pour représenter graphiquement les données (Pandas se base sur la librairie matplotlib, qui est la librairie de graphe en Python la plus utilisée).

Le Machine Learning avec Scikit-learn

Créée en 2007, Scikit-learn est l'une des librairies open source les plus populaires pour le Machine Learning en Python. La librairie est construite à partir de NumPy, Matplotlib et SciPy.
En plus de contenir et de rendre accessible facilement tous les principaux algorithmes de Machine Learning (**classification**, **clustering**, **régression**), elle contient aussi de nombreux modules pour la **réduction de dimension**, l'**extraction de features**, le **processing des données** et l'**évaluation des modèles**.

Brève introduction au Machine Learning

Le but de cet article n'est pas de présenter le Machine Learning au lecteur, mais quelques notions de base sont indispensables afin de comprendre comment et dans quelles situations utiliser Scikit-learn. *Pour approfondir le sujet, je vous invite à parcourir nos articles de blog (blog.xebia.fr) sur le sujet.*
Le Machine Learning est la branche de l'Intelligence Artificielle qui s'occupe de la construction et de l'étude de systèmes qui apprennent à partir de données. On distingue deux principales catégories d'algorithmes d'apprentissage : supervisé et non-supervisé.
Dans l'apprentissage supervisé, on a en notre possession un dataset contenant à la fois des caractéristiques et des labels. On peut ainsi voir la variable label (notre cible) comme une fonction des caractéristiques. L'objectif est alors d'apprendre cette fonction afin de pouvoir prédire le label d'un nouvel objet étant donné ses caractéristiques. On peut par exemple vouloir prédire le prix d'une maison (le label est donc le prix) en fonction de sa surface, son nombre de chambres, etc. On parle ici de régression, car la variable à prédire est continue. On parle, au contraire, de classification lorsque l'on cherche à prédire une classe, comme par exemple le fait de prédire si un mail est un spam ou non à partir de son contenu.
En ce qui concerne l'apprentissage non supervisé, nous n'avons aucune information sur une éventuelle variable à prédire. Seul un dataset de caractéristiques est à notre disposition et l'objectif est alors d'extraire une structure des données. On parle notamment de clustering lorsque l'on cherche à diviser un ensemble de données en différents groupes homogènes, sans avoir d'information préalable sur la nature de ces groupes. La démarche en Machine Learning est relativement systématique, quelle que soit l'application : L'algorithme apprend un modèle sur un dataset d'apprentissage (qui a souvent eu besoin au préalable d'être traité), et est capable de faire des prédictions pour de nouvelles données. Ainsi, si nous avons un historique de maisons avec leurs caractéristiques ainsi que leur prix de vente, il est possible de construire un modèle sur cet historique, et de l'utiliser pour prédire le prix d'une nouvelle maison à estimer.
Maintenant que nous maîtrisons un peu plus de vocabulaire relatif au Machine Learning, nous pouvons commencer à utiliser Scikit-learn et à découvrir ses différentes fonctionnalités.

Scikit-learn : une excellente documentation

Que l'on débute dans la Data Science ou bien que l'on soit un utilisateur expérimenté, la documentation attachée à Scikit-learn est d'un recours très efficace. Toutes les méthodes sont bien documentées et l'on peut trouver de très nombreux exemples d'utilisation pour tous types de tâches. La documentation fournit même des conseils quant à l'utilisation de tel ou tel modèle de Machine Learning en fonction de vos besoins et de vos données disponibles, sous forme d'un schéma global facile à utiliser, visible Fig.1.

Une utilisation simple des algorithmes grâce à une API consistante

La plupart des algorithmes de Scikit-learn utilisent des jeux de données sous forme de tableaux (ou de matrices) à deux dimensions. Ces tableaux vont être soit du type numpy.ndarray, soit de type scipy.sparse

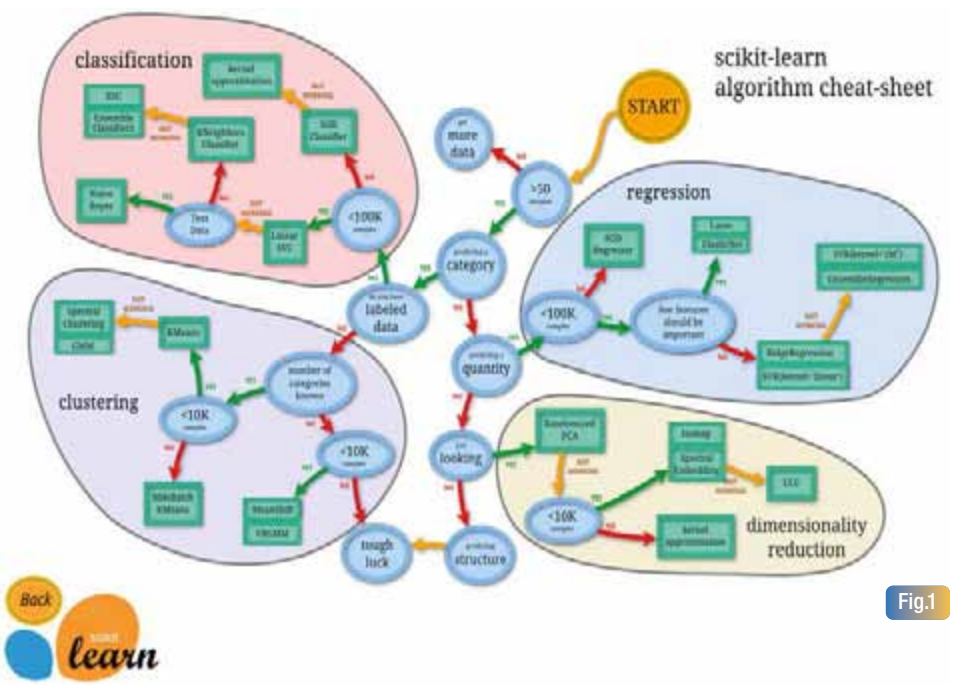

Fig.1

de la librairie SciPy. Il est aussi possible de fournir directement une DataFrame Pandas, la fonction prendra alors automatiquement son attribut *values* qui fournit les données sous forme d'array NumPy. La dimension de cette matrice sera [n_samples, n_features], avec:

- **n_samples:** le nombre d'échantillons de données ou d'observations. Un échantillon peut représenter un document, une image, un son, une vidéo, un objet, une ligne dans une base de données ou d'un fichier .csv ;
- **n_features:** le nombre de caractéristiques qui décrivent une observation de manière quantitative. Ces données doivent être des nombres (réels ou entiers).

Dans le cadre d'un apprentissage supervisé, les algorithmes nécessitent de plus qu'on leur fournisse le vecteur des labels associés à chaque donnée (un array NumPy à une dimension par exemple).
La plupart des opérations vont être réalisées à l'aide d'objets de type *estimator*. Un algorithme de Machine Learning correspond à une classe de type estimator :

```
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression(normalize=True)
>>> print model
```

scikit-learn s'efforce d'avoir une interface uniforme pour tous les algorithmes, ce qui est une de ses plus grandes forces. Étant donné un objet *estimator model*, plusieurs méthodes sont disponibles.
Pour tous les estimators :

- model.fit() : lance le mécanisme d'apprentissage.
  - model.fit(X, y) pour les algorithmes supervisés (X les observations d'entraînement, y leurs labels) ;
  - model.fit(X) pour les algorithmes non-supervisés.

Pour les estimators supervisés :

- model.predict(X_new) : prédit et retourne les labels d'un nouveau jeu de données X_new ;
- model.predict_proba(X_new) : pour les algorithmes de classification, prédit et retourne les probabilités d'appartenance à une classe d'un nouveau jeu de données X_new ;
- model.score(X_test, y_test) : retourne un score de performance du modèle entre 0 (mauvais) et 1 (idéal mais suspicieux).

Pour les estimators non-supervisés:

- model.transform(X_new) : transforme un jeu de données selon le modèle ;
- model.fit_transform(X) : apprend ses paramètres grâce à un jeu de données et transforme celui-ci.

Mise en situation

Nous allons maintenant mettre en application les notions introduites jusque là sur Pandas et Scikit-learn sur un exemple concret.
Scikit-learn permet de télécharger des jeux de données permettant de tester des modèles. Nous allons ici travailler sur le jeu de données *california housing data*, où l'objectif est de prédire le prix d'une maison à partir de caractéristiques (appelées features) telles que l'âge de la maison, la surface moyenne des chambres, la population dans la région ou encore les coordonnées géographiques.

```
>>> import pandas as pd
>>> from sklearn import datasets
>>> dataset = datasets.fetch_california_housing() # Downloads the dataset
>>> print dataset.feature_names
>>> dataset.data.shape
```

Comme on peut le constater, nous avons en notre possession des données concernant 20640 maisons, et 8 caractéristiques pour chaque maison. Le prix correspondant est stocké dans *dataset.target*. A noter que les données ont été modifiées pour ne pas correspondre aux valeurs réelles.
Avant de vouloir construire un modèle sur ces données, il est toujours important de les explorer et de comprendre leur sens. Pour cela, quoi de mieux que de les mettre sous forme d'une DataFrame Pandas et de faire appel à ses fonctionnalités.

```
>>> data_df = pd.DataFrame(data=dataset.data, columns=dataset.feature_names) # Create a Data Frame corresponding to the data and the columns names
>>> data_df.head() # Print the first lines of the dataset
>>> data_df.describe(percentiles=[0.25, 0.5, 0.75, 0.99]) # General statistics on all columns
>>> target_df = pd.DataFrame(data=dataset.target, columns=["Price"]) # DataFrame corresponding to the target
```

La méthode *describe()* permet notamment de détecter des outliers dans les données. Et comme on peut le constater, de nombreux sont présents dans ce jeu de données. Si l'on compare les valeurs entre le 99ème percentile et la valeur maximale de chaque variable, on constate parfois de très gros écarts (notamment avec AveOccup, AveRooms, AveBedrms). Ils sont d'autant plus visibles lorsque l'on dessine les distributions, chose facilement faisable en pandas avec la méthode *hist()*, qui se base sur matplotlib (Fig.2).

```
>>> %matplotlib inline # Allows to drow plots in the notebook
>>> data_df.hist(bins=50, figsize=(15,7)); # Histograms for all numerical columns of the DataFrame
```

Lorsque les distributions sont très clairement écrasées, cela signifie qu'il y a un ou plusieurs points très éloignés du reste. Ces outliers peuvent correspondre à des erreurs, ou bien peuvent représenter un sous-ensemble de la réalité qui est sous-représenté dans les données fournies. Il faut alors soit les remplacer par des valeurs adéquates, on parle alors d'imputation, soit supprimer les lignes en contenant, sans quoi nos modèles peuvent parfois être erronés s'ils sont sensibles aux outliers. Dans notre exemple, nous allons les supprimer. On ne va garder

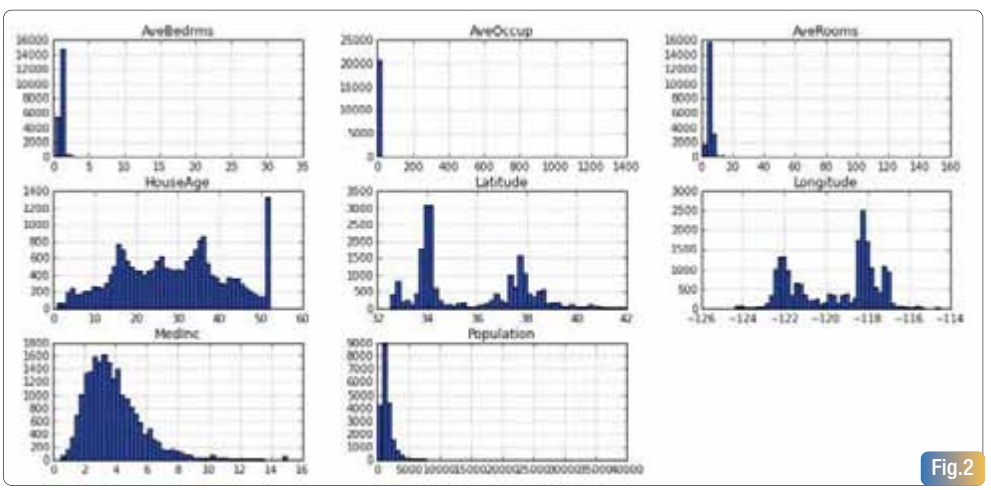

Fig.2

que les valeurs jusqu'au 99ème percentile. Ceci est très simple en Pandas, il suffit de lui fournir un vecteur de booléens contenant *False* là où il y a des outliers (Fig.3). Ne pas oublier de changer la target en conséquence.

```
>>> data_clean = data_df[((data_df.HouseAge<52) & (data_df.AveRooms<10.5) & (data_df.AveBedrms<3) &
        (data_df.Population<6000) & (data_df.AveOccup<6) & (target_df.Price<5))].copy()  # Removing outliers in DataFrame of values
>>> target_clean = target_df[((data_df.HouseAge<52) & (data_df.AveRooms<10.5) & (data_df.AveBedrms<3) &
        (data_df.Population<6000) & (data_df.AveOccup<6) & (target_df.Price<5))].copy()  # Removing outliers in DataFrame of target
>>> data_clean.hist(bins=50, figsize=(15,7));  # Histogram of the clean data
```

Maintenant que Pandas nous a permis de préparer nos données, on peut utiliser Scikit-learn pour construire un modèle dessus. Afin de tester les performances du modèle, il est important de séparer les données en un train et un test set. Le modèle va être appris sur le train set (qui représente la majorité des données). Le test set va être utilisé comme si on ne connaissait pas la target, le but sera alors de comparer la prédiction du modèle sur ce test set avec les vraies valeurs; cela permet d'évaluer le modèle, l'idée étant d'avoir une estimation de l'erreur. Cette séparation se fait très simplement avec Scikit-learn grâce à la méthode *train_test_split()*.

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(data_clean,target_clean, test_size=0.2)
# We take 20% of the data for final testing
```

Place maintenant à la construction de notre modèle de Machine Learning. Nous allons utiliser ici un modèle simple de Régression Linéaire, pour lequel le but est de trouver la meilleure combinaison possible des caractéristiques afin de minimiser la moyenne du carré des erreurs entre la prédiction et la vraie valeur pour chaque donnée d'apprentissage.
Cependant, comme on l'a vu précédemment, quel que soit l'algorithme choisi, l'utilisation de ce dernier se fait de manière identique, seuls les paramètres des modèles sont à changer.
L'utilisateur pourra alors tester d'autres modèles pour améliorer les performances que l'on obtiendra. Une fois le modèle chargé, il suffit de l'entraîner sur le train set à l'aide de la méthode *fit()* et de faire des prédictions sur le test set à l'aide de la méthode *predict()*.

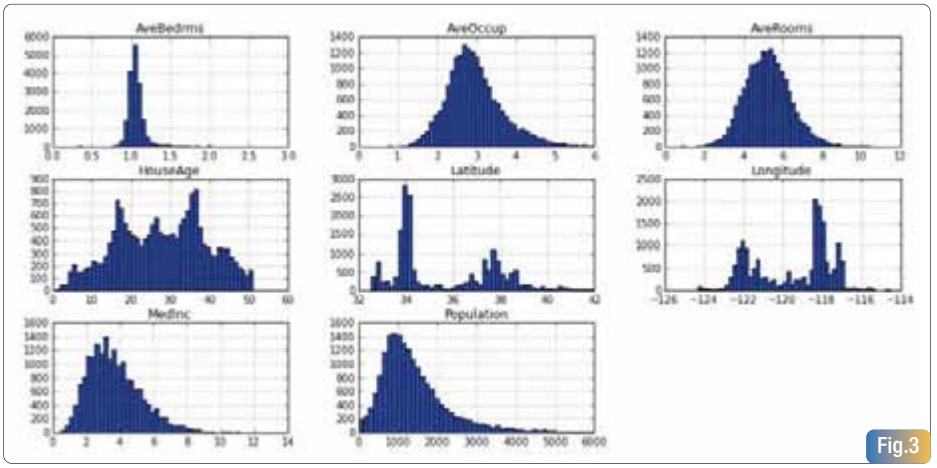

Fig.3

```
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression(fit_intercept=True, normalize=False)  # Create a Linear Regression model
>>> model.fit(X_train, y_train)  # Train the model on the train set
>>> pred = model.predict(X_test)  # Predict on the test set
```

Il est important de normaliser les données (*normalize=True*), sans quoi les caractéristiques avec les plus grandes valeurs seront prépondérantes. Maintenant que nous avons des prédictions, il faut les comparer avec les vraies valeurs du test set.

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(pred, y_test)
```

On obtient un score de 0.33 (le but étant d'être proche de zéro), ce qui permet d'avoir une première idée des scores obtenables sur ce dataset. Il est à noter que le modèle en l'état est trop simple pour caractériser le phénomène étudié. Une approche possible pour y remédier serait d'ajouter des variables d'ordres supérieurs. Une seconde approche est d'envisager des modèles plus complexes (avec moins de biais) mais il faudra alors faire attention à ne pas overfitter (coller aux données d'apprentissage). En effet l'overfitting peut être vu comme un apprentissage par coeur et le modèle serait alors incapable de généraliser et d'avoir de bonnes performances sur des données encore jamais observées. L'utilisateur curieux pourra tenter d'utiliser d'autres algorithmes réputés plus performants, tels que les RandomForestRegressor, en faisant bien attention aux paramètres requis, en effet ce type de modèle requiert la spécification de certains paramètres, appelés "hyperparamètres". On parle alors de tuning du modèle. scikit-learn dispose à cet effet de méthodes tel le *GridSearch* pour assister l'utilisateur dans la sélection du jeu de paramètres "optimal" pour le modèle, optimal au sens de la fonction à optimiser (ici le mean_squared_error) et relativement à la grille (discrète) spécifiés par l'utilisateur.

Il existe de très nombreuses autres fonctionnalités associées à la librairie Scikit-learn, on peut notamment citer la réduction de dimension (PCA).

Conclusion

De nombreuses autres librairies Python peuvent être utilisées pour travailler sur des données, on peut notamment citer **statsmodel** (modèles statistiques) ou **nltk** (traitement de données textuelles). Nous vous avons introduit ici les plus utilisées.
Python est un langage très couramment utilisé en Data Science, nous avons vu pourquoi grâce à ses différentes librairies. Il est relativement simple à prendre en main pour faire du travail sur la donnée, et tout est fait pour nous faciliter la tâche dans la réalisation d'opérations courantes qui peuvent être parfois assez complexes. Enfin, pour des problématiques de Big Data, le projet Spark (très en vogue en ce moment) propose une API en Python, appelée PySpark, montrant l'importance de ce langage dans le monde de la Data Science.

Références

Python for Data Analysis, O'Reilly
https://github.com/jakevdp/sklearn_pycon2014