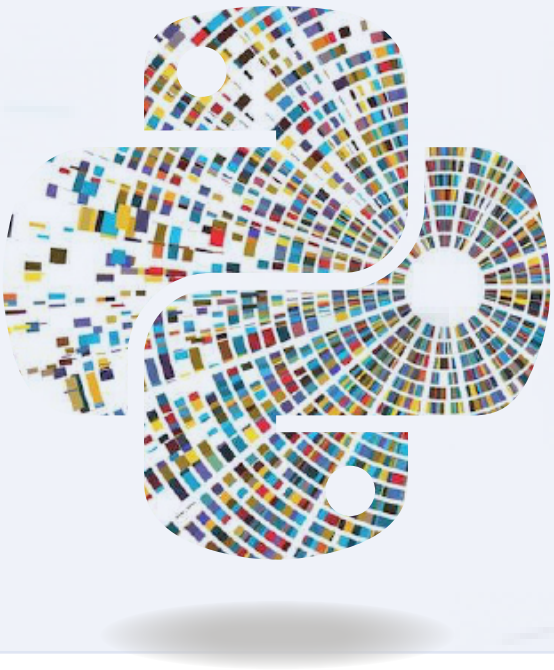

analyse de données et intelligence artificielle en Python

Interpréter les données réelles avec NumPy, pandas et le scikit-learn

Angelo Cardellicchio (Italie)

L'analyse et l'interprétation des données provenant de notre environnement est un sujet d'intérêt croissant. Ces données font désormais partie intégrante de notre vie quotidienne, qu'il s'agisse des données climatiques ou des données acquises au cours de processus de fabrication intelligents. La quantité de données nous permet, en théorie, de caractériser n'importe quel phénomène. Cependant il faut pour cela de nombreuses compétences, théoriques et pratiques. Nous examinons ici certaines de ces techniques et utilisons Python pour l'analyse de données du monde réel.

Des termes tels que *big data* (= *données massives)* et *intelligence artificielle* appartiennent au langage quotidien. Cela est principalement dû à deux facteurs. Le premier est la diffusion croissante et omniprésente des systèmes d'acquisition de données qui a permis la création de *référentiels de connaissances* pratiquement illimités. Le second est la croissance continue des capacités de calcul, grâce à l'utilisation généralisée des GPGPU (unités de traitement graphique polyvalentes) [1], qui a permis de relever des défis de calcul considérée autrefois comme impossible.

Commençons par la description d'un scénario d'application qui nous accompagnera au long de cet article. Imaginons qu'il faille surveiller une chaîne de production (peu importe le produit). Nous pouvons acquérir des données provenant d'un large éventail de sources. Par exemple, nous pouvons placer des capteurs sur toute la chaîne de production, ou utiliser des *informations contextuelles* qui indiquent l'âge et le type de chaque machine. Cet ensemble de données, ou jeu de *données*, peut être utilisé à différentes fins. Cela peut être la maintenance prédictive, pour évaluer et prévoir l'apparition de situations anormales, planifier les commandes de pièces de rechange ou entreprendre des réparations avant que les pannes ne se produisent, ce qui permet de réaliser des économies et d'accroître la productivité. En outre, la connaissance de l'historique des données nous permet de corréler les données mesurées par chaque capteur, pour mettre en évidence les éventuelles relations de cause à effet. Par exemple, si une augmentation soudaine de température et d'humidité dans la pièce a été suivie d'une diminution du nombre de pièces fabriquées, une modification pourrait consister à maintenir des conditions climatiques constantes à l'aide d'une climatisation.

La mise en place d'un tel système n'est certainement pas à la portée de tous. Cependant, elle est simplifiée par les outils mis à disposition par la communauté *open source*. Il suffit d'avoir un PC (ou, à défaut, un Raspberry Pi si la quantité de données à traiter n'est pas énorme), de connaitre Python (que vous pouvez approfondir en suivant un tutoriel comme celui-ci [2]) et, bien sûr les «outils» que je vous propose de découvrir ensemble !

Outils

Il faut savoir créer des programmes en Python. Pour cela, nous devrons installer l'interpréteur que vous trouverez sur le site officiel de Python [3]. À partir d'ici, nous supposerons que Python est installé et ajouté aux variables d'environnement de votre système.

L'environnement virtuel

Une fois l'installation de Python terminée, mettons en place un environnement virtuel, qui est en quelque sorte un «conteneur» séparé du reste de notre système et dans lequel sont installées les bibliothèques utilisées. La raison de l'utilisation d'un environnement virtuel pour l'installation globale des bibliothèques est liée à l'évolution rapide du monde Python. Très souvent, des différences substantielles surviennent même entre versions mineures de l'interpréteur, ce qui entraîne l'incompatibilité de bibliothèques

et, par conséquent, des programmes, car elles ont été écrites pour différentes versions de Python. En aménageant un environnement déterministe où la version de chaque bibliothèque installée est connue, nous aurons une sorte de garantie que nos programmes fonctionneront. En fait, il suffira de reproduire précisément la configuration de l'environnement virtuel et nous pouvons être sûrs que tout fonctionnera.
Pour gérer nos environnements virtuels, nous utilisons un logiciel appelé `virtualenvwrapper`. Celui-ci peut être installé à partir du shell en utilisant `pip` :

```
$ pip install virtualenvwrapper
```

Une fois l'installation terminée, nous créons un nouvel environnement virtuel comme suit :

```
$ mkvirtualenv ml-python
```

Notez que `ml-python` est le nom de l'environnement virtuel choisi pour notre exemple de scénario. Évidemment, ces noms peuvent varier et n'importe quel nom approprié pourra être choisi par le concepteur. Nous procédons ensuite à l'activation de l'environnement virtuel :

```
$ workon ml-python
```

Nous sommes prêts maintenant à installer les éléments nécessaires pour la suite de cet article.

Bibliothèques

Les bibliothèques présentées et utilisées ici sont les cinq plus utilisées pour l'analyse des données en Python.
La première, et peut-être la plus célèbre, est *NumPy*, qui est une sorte de port de MATLAB pour Python. NumPy est une bibliothèque pour calculs algébriques et matriciels. Par conséquent, les utilisateurs familiers de MATLAB trouveront de nombreuses similitudes, en termes de syntaxe et d'optimisation. L'utilisation du calcul algébrique dans NumPy est, en fait, plus efficace que les cycles imbriqués dans MATLAB (pour en savoir plus [4]). Comme on pouvait s'y attendre, le type de données au cœur de la fonction de NumPy est le *tableau* ou *array* en anglais. Il ne doit pas être confondu avec le *vecteur* informatique correspondant, mais doit plutôt être compris au sens algébrique et géométrique du terme comme une *matrice*. Comme l'analyse des données est basée sur des opérations algébriques et matricielles, NumPy est également à la base de deux des cadres les plus utilisés : scikit-learn (dont il sera question plus loin) et TensorFlow.
Un complément naturel à NumPy est la bibliothèque *pandas*, qui gère et lit des données provenant de sources hétérogènes, y compris des tableaux Excel, des fichiers CSV, ou même des bases de données JSON et SQL. *pandas* est extrêmement flexible et puissant, vous permettant d'organiser les données en structures appelées *dataframes* manipulables selon les besoins et exportables facilement directement dans des tableaux NumPy.
La troisième bibliothèque que nous utiliserons est *scikit-learn*. Née d'un projet universitaire, scikit-learn est un cadre qui met en œuvre la plupart des algorithmes d'apprentissage automatique utilisés de nos jours en fournissant une interface commune. Ce dernier concept est précisément celui de la programmation orientée objet : il est en effet possible d'utiliser pratiquement tous les algorithmes proposés par scikit-learn grâce à la méthode `fit_transform` en passant au moins deux paramètres, tels que les données analysées et les étiquettes associées.
Les deux dernières bibliothèques que nous utiliserons sont *Matplotlib* et *Jupyter*. La première, avec son complément *Seaborn*, est nécessaire pour visualiser les résultats de nos expériences sous forme de graphiques. La seconde nous offre l'utilisation de *notebooks* ou *carnets*, environnements interactifs d'utilisation simple et immédiate qui permettent à l'analyste de données d'écrire et d'exécuter des parties de code indépendamment des autres.
Avant d'aller plus loin, cependant, voici quelques concepts théoriques nécessaires à la construction d'une «base commune» pour le discours.

Les concepts

Le premier concept requis est celui des *ensembles de données* (ou *datasets*), ce qui est souvent considéré comme allant de soi. Il s'agit d'ensembles d'échantillons, chacun d'entre eux étant caractérisé par

Figure 1. L'écran d'accueil pour la gestion des carnets (notebook) dans Jupyter.

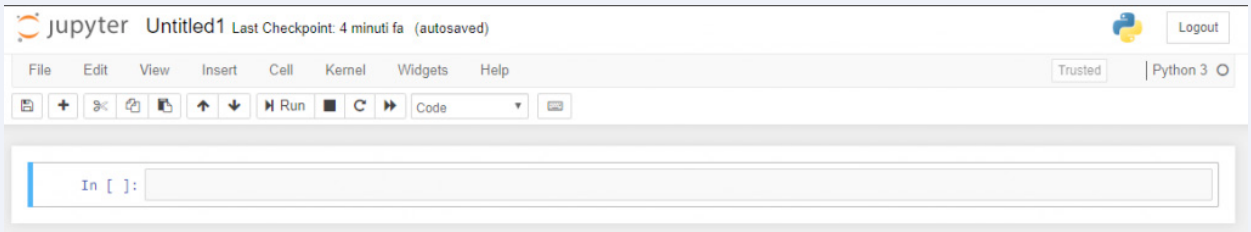

Figure 2. Un carnet (notebook) vide.

un certain nombre de variables ou de caractéristiques, qui décrivent le phénomène observé. Par souci de simplicité, nous pouvons considérer un ensemble de données comme un tableau Excel. Les lignes fournissent les *échantillons*, c'est-à-dire les *observations individuelles* du phénomène, tandis que les colonnes fournissent les *caractéristiques, c'est-à-dire les* valeurs qui caractérisent chacun des aspects du processus. Pour revenir à l'exemple de la fabrication intelligente, chaque ligne représentera les conditions de la chaîne de production à un moment donné tandis que chaque colonne comportera la valeur lue sur un capteur donné.

À propos de scikit-learn, il y a lieu de mentionner le concept de *label* ou de *classe*. La présence ou l'absence d'étiquettes permet de distinguer les algorithmes supervisés des algorithmes non supervisés. La différence est, du moins en principe, assez simple : les *algorithmes supervisés* requièrent une connaissance a priori de la classe de chaque échantillon de l'ensemble de données de l'exemple, pas les *algorithmes non supervisés*. En pratique, pour utiliser un algorithme supervisé, il est nécessaire qu'un expert du domaine établisse la *classe d'appartenance de* chaque échantillon. Dans le cas d'un processus de fabrication intelligent, un «expert» pourrait déterminer si un ensemble de lectures, à un moment précis, représente une situation anormale ou non. Ainsi, l'échantillon unique peut être associé à l'une de ces deux classes possibles (anormal/normal). Cela n'est pas nécessaire pour les algorithmes non supervisés.

Il faut aussi distinguer les processus avec des données *indépendantes et identiquement distribuées* (IID) des données dans un *ordre chronologique*. La différence est liée à la nature du phénomène observé. Les échantillons d'un processus IID sont indépendants les uns des autres, alors que dans une série temporelle, chaque échantillon dépend d'une combinaison linéaire ou non linéaire des valeurs que le processus a produites à des moments précédents.

Commençons !

Maintenant que les termes théoriques et pratiques requis sont connus, nous passons à l'utilisation d'un ensemble de données approprié pour notre exemple. L'ensemble de données utilisé est *SECOM*, un acronyme qui signifie *SEmiCOnductor Manufacturing*, qui contient les valeurs lues par un ensemble de capteurs lors de la surveillance d'un processus de fabrication de semi-conducteurs. Dans l'ensemble de données, téléchargeable à partir de différentes sources (comme Kaggle [5]), il y a 590 variables, représentatives chacune de la lecture d'un seul capteur à un moment donné. L'ensemble de données contient également des étiquettes qui permettent de distinguer les défaillances et les anomalies du bon fonctionnement du système.

Une fois l'ensemble de données téléchargé, nous installons les bibliothèques mentionnées ci-dessus. Depuis la ligne de commande, saisissez :

```
$ pip install numpy pandas scikit-learn matplotlib
   seaborn jupyter numpy pandas install
```

Une fois les bibliothèques installées, nous pouvons mettre en place un simple pipeline pour l'analyse des données.

Le premier carnet

La première étape consiste à créer un nouveau *carnet*. À partir de la ligne de commande, nous lançons Jupyter par les instructions suivantes :

```
$ jupyter-notebook
```

Un écran similaire à celui de la **figure 1** s'ouvre. Nous créons un *carnet* en sélectionnant *Nouveau* > *Python 3*. Un nouvel onglet s'ouvrira dans notre navigateur avec le carnet créé. Prenons le temps de nous familiariser avec l'interface (**fig. 2**) qui ressemble (très vaguement) à une ligne de commande interactive, un menu et plusieurs options.

Ce qui saute aux yeux est la *cellule*. L'exécution de cellules individuelles est lancée par le bouton *Run*. Elle est indépendante de celle des autres cellules (gardons à l'esprit la validité du concept de *portée des variables*).

Les trois boutons immédiatement à droite du bouton *Run* permettent d'arrêter, de redémarrer et d'initialiser le *noyau*, c'est-à-dire l'instance associée à notre carnet par Jupyter. Le redémarrage de l'instance peut être nécessaire pour réinitialiser les variables locales et globales associées au script, ce qui est particulièrement utile lorsque vous expérimentez avec de nouvelles méthodes et bibliothèques.

	a21	a87	a88	a89	a114	a115	a116	a117	a118	a120	...	a528
count	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	1567.000000	...	1567.000000
mean	1.405054	2.401872	0.982420	1807.815021	0.945424	0.000123	747.383792	0.987130	58.625908	0.970777	...	6.395717
std	0.016737	0.037332	0.012848	53.537262	0.012133	0.001668	48.949250	0.009497	6.485174	0.008949	...	1.888698
min	1.179700	2.242500	0.774900	1627.471400	0.853400	0.000000	544.025400	0.890000	52.806800	0.841100	...	2.170000
25%	1.396500	2.376850	0.975800	1777.470300	0.938600	0.000000	721.023000	0.989500	57.978300	0.964800	...	4.895450
50%	1.406000	2.403900	0.987400	1809.249200	0.946400	0.000000	750.861400	0.990500	58.549100	0.969400	...	6.410800
75%	1.415000	2.428600	0.989700	1841.873000	0.952300	0.000000	776.781850	0.990900	59.133900	0.978300	...	7.594250
max	1.453400	2.555500	0.993500	2105.182300	0.976300	0.041400	924.531800	0.992400	311.734400	0.982700	...	14.447900

Figure 3. Les cinq premières lignes de l'ensemble de données SECOM.

	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	...	a582	a583	a584	a585	a586	a587	a588	a58
0	3030.93	2564	2187.7333	1411.1265	1.3602	100	97.6133	0.1242	1.5005	0.0162	...	?	0.5005	0.0118	0.0035	2.363	?	?	
1	3095.78	2465.14	2230.4222	1463.6606	0.8294	100	102.3433	0.1247	1.4966	-0.0005	...	208.2045	0.5019	0.0223	0.0055	4.4447	0.0096	0.0201	0.00
2	2932.61	2559.94	2186.4111	1698.0172	1.5102	100	95.4878	0.1241	1.4436	0.0041	...	82.8602	0.4958	0.0157	0.0039	3.1745	0.0584	0.0484	0.014
3	2988.72	2479.9	2199.0333	909.7926	1.3204	100	104.2367	0.1217	1.4882	-0.0124	...	73.8432	0.499	0.0103	0.0025	2.0544	0.0202	0.0149	0.004
4	3032.24	2502.87	2233.3667	1326.52	1.5334	100	100.3967	0.1235	1.5031	-0.0031	...	?	0.48	0.4766	0.1045	99.3032	0.0202	0.0149	0.004

5 rows × 591 columns

Figure 4. Brève description statistique de l'ensemble de données SECOM.

Une autre option utile est le choix possible du type de cellule, entre *Code* (c'est-à-dire code Python), *Markdown* (utile pour insérer des commentaires et des descriptions dans le format utilisé, par exemple, par GitHub READMEs), *Raw NBContent* (texte *brut)* et *Heading* (offrant un raccourci pour insérer des titres).

Importation et affichage des données

Une fois familiarisés avec l'interface, nous pouvons passer à la mise en œuvre de notre script. Ici, nous importons les bibliothèques et les modules que nous utiliserons :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from ipywidgets import interact
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix,
   accuracy_score
from sklearn.utils import resample
```

Signalons l'instruction `%matplotlib inline` qui nous permet d'afficher correctement les graphiques produits par Matplotlib. Ensuite, les données du fichier contenant l'ensemble de données SECOM sont importées à l'aide de la fonction `read_csv` de *pandas*. Notez que, dans cet exemple, pour simplifier, le chemin relatif du fichier est codé en dur. Cependant, il est conseillé d'utiliser le paquet *os* Python pour permettre à notre programme de déterminer lui-même ce chemin lorsque c'est nécessaire.

```
data = pd.read_csv('data/secom.csv')
```

L'instruction précédente lit les données contenues dans le fichier *secom.csv, en les* organisant dans un cadre de données nommé *data*. Nous pouvons afficher les cinq premières lignes de la trame de données grâce à l'instruction `head()` (**fig. 3**).

```
data.head()
```

La visualisation des premières lignes du cadre de données peut être utile pour obtenir un premier aperçu des données à analyser. Dans ce cas, on remarque immédiatement la présence de certaines valeurs égales à « ?» qui représentent probablement des valeurs nulles. En outre, il est évident que la plage des valeurs varie considérablement, un facteur que nous devrons garder à l'esprit plus tard. Nous pouvons également utiliser la fonction `describe()` pour obtenir un aperçu rapide des caractéristiques statistiques de chaque variable (**fig. 4)**.

```
data.describe()
```

L'analyse statistique peut, en général, mettre en évidence des conditions présentant un manque de normalité (c'est-à-dire des données dont la distribution n'est pas paramétrique), ou la présence d'anomalies. À titre d'exemple, nous constatons que l'écart-type (std) associé aux variables *a116* et *a118* est assez élevé proportionnellement, de sorte que nous nous attendons à une forte signification de ces variables lors de leur analyse. D'autre part, des variables telles que *a114* ont un faible std, on s'attend donc à ce qu'elles soient écartées faute de pertinence dans le processus analysé.
Une fois le chargement et l'affichage de la trame de données terminés, nous pouvons passer à une partie fondamentale du pipeline : le *prétraitement*.

Prétraitement des données

Dans un premier temps, nous affichons le nombre d'échantillons associés à chaque classe. Pour ce faire, nous utiliserons la fonction `value_counts()` sur la colonne `classvalue` car elle contient les étiquettes associées à chaque échantillon.

```
data[«classvalue»]. value_counts()
```

Nous voyons qu'il y a 1463 échantillons prélevés en situation normale de fonctionnement (*classe -1*) et 104 en situation de défaillance (*classe 1*). L'ensemble de données est donc fortement déséquilibré et il serait approprié de prendre des mesures pour uniformiser la répartition des échantillons entre différentes classes. Cela renvoie à la fonction intrinsèque des algorithmes d'apprentissage machine qui apprennent sur la base des données dont ils

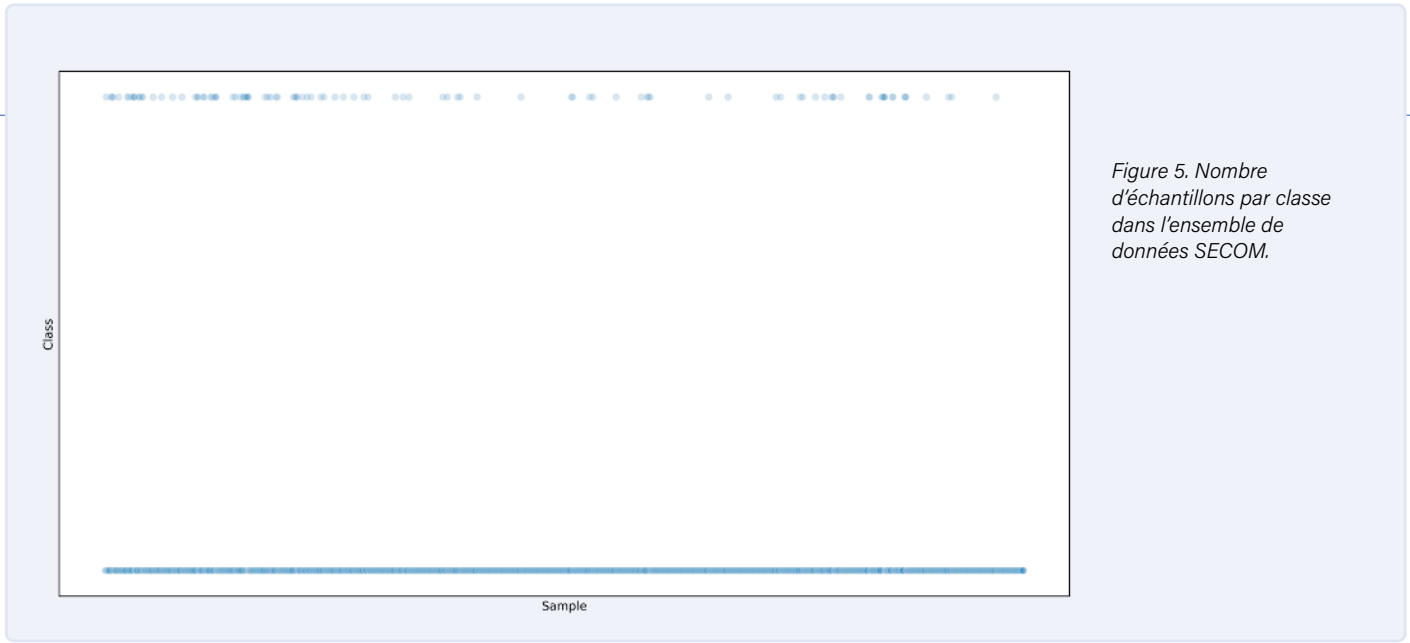

Figure 5. Nombre d'échantillons par classe dans l'ensemble de données SECOM.

disposent. Dans ce cas précis, l'algorithme apprendra à caractériser avec succès une situation de comportement standard, mais sa caractérisation des situations anormales sera affectée par de «incertitudes». Le déséquilibre est encore plus évident lorsque l'on observe le nuage de points (**fig. 5**) :

```
sns.scatterplot(data.index, data['classvalue'],
   alpha=0.2)
plt.show()
```

En gardant à l'esprit ce déséquilibre (auquel nous reviendrons), nous séparons les étiquettes des données :

```
labels = data['classvalue']
data.drop('classvalue', axis=1, inplace=True)
```

Notez l'utilisation du paramètre `axis` dans la fonction `drop` qui nous permet de spécifier que la fonction doit opérer sur les colonnes du cadre de donnés (*dataframe*). Par défaut, les fonctions *pandas* opèrent sur les lignes.

Un autre aspect qui peut être extrapolé de l'analyse des ensembles de données est la présence, dans cette version spécifique des données SECOM, de nombreuses colonnes contenant des données de différents types, c'est-à-dire à la fois des chaînes de caractères et des chiffres. Par conséquent, les *pandas,* incapables de déterminer avec certitude le type de données avec lequel chaque caractéristique est représentée, en reportent la définition sur l'utilisateur. Pour mettre toutes les données sous forme numérique, il faut donc utiliser trois fonctions offertes par les *pandas*.
La première fonction que nous utiliserons est `replace()`, avec laquelle nous pouvons remplacer tous les points d'interrogation par la valeur constante *numpy.nan,* le caractère de remplacement utilisé pour traiter les valeurs nulles dans les *tableaux* NumPy.

```
data = data.replace('?', np.nan, regex=False)
```

Le premier paramètre de la fonction est la valeur à remplacer, le deuxième est la valeur à utiliser pour le remplacement, et le troisième est un drapeau indiquant si le premier paramètre représente ou non une expression régulière. Nous pourrions également utiliser une syntaxe alternative en utilisant le paramètre `inplace` réglé sur `True`, comme suit :

```
data.replace('?', np.nan, regex=False, inplace=True)
```

Les deuxième et troisième fonctions que nous pouvons utiliser pour résoudre les problèmes signalés ci-dessus sont les fonctions `apply()` et `to_numeric()` respectivement. La première permet d'appliquer une certaine fonction à toutes les colonnes (ou lignes) d'une trame de données, tandis que la deuxième convertit une seule colonne en valeurs numériques. En les combinant, nous produisons des données uniques et nous supprimons également les valeurs que NumPy et scikit-learn ne peuvent traiter :

```
data.apply(pd.to_numeric)
```

Nous devons maintenant évaluer quelles caractéristiques contenues dans l'ensemble de données sont réellement utiles. Nous utilisons généralement des techniques (plus ou moins complexes) de *sélection des caractéristiques* pour réduire les redondances et la taille du problème à traiter, ce qui présente des avantages évidents en termes de temps de traitement et de performance de l'algorithme. Dans notre cas, nous nous appuyons sur une technique moins complexe qui implique l'élimination des caractéristiques de variance faible (et donc, comme mentionné, de faible importance). Nous créons donc un widget interactif qui nous permet de visualiser, sous la forme d'un histogramme, la distribution des données pour chaque caractéristique :

```
@interact(col=(0, len(df.columns) - 1)
def show_hist(col=1):
    data['a' + str(col)]. value_counts().
   hist(grid=False, figsize=(8, 6))
```

L'*interactivité* est assurée par le *décorateur* (= *decorator*) `@interact`, dont la valeur de référence (c'est-à-dire `col`) varie entre 0 et le nombre de caractéristiques présentes dans l'ensemble de données. En explorant les données affichées par le widget, nous déterminerons combien de caractéristiques prennent une valeur unique, ce qui signifie qu'elles peuvent être simplement ignorées dans l'analyse. Nous pouvons alors décider de les éliminer comme suit :

```
single_val_cols = data.columns [len(data)/data.
   nunique() < 2]
secom = data.drop(single_val_cols, axis=1)
```

Bien sûr, il existe des techniques de sélection de caractéristiques plus pertinentes et plus raffinées qui utilisent par exemple des paramètres statistiques. La documentation scikit-learn [6] en donne un aperçu complet.

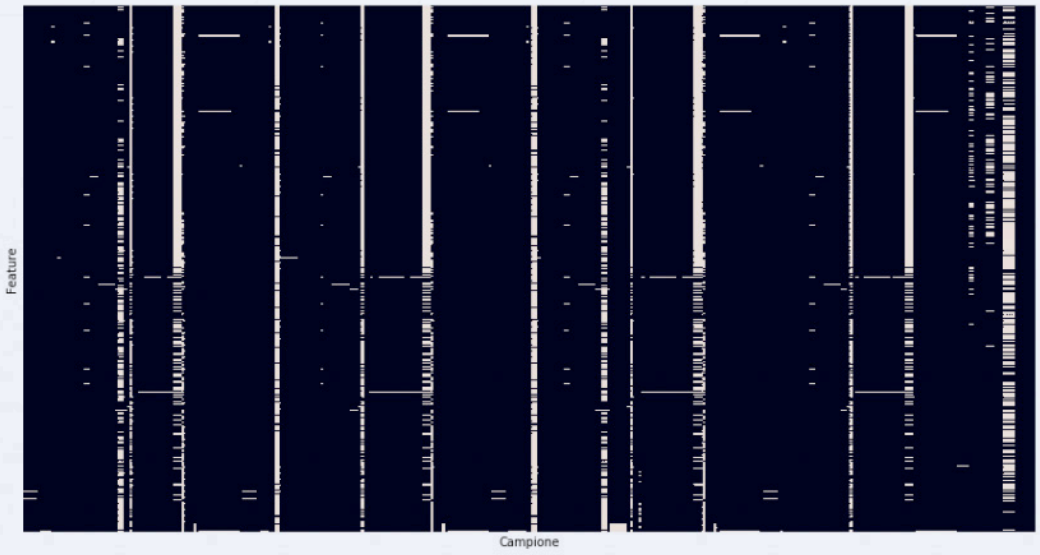

Figure 6. Carte des valeurs nulles dans l'ensemble de données SECOM.

La dernière étape consiste à traiter les valeurs nulles (que nous avons remplacées précédemment par *np.nan*). Nous inspectons l'ensemble de données pour voir combien il y en a ; pour ce faire, nous utilisons une carte thermique (**fig. 6**) où les points blancs représentent les valeurs nulles.

```
sns.heatmap(secom.isnull(), cbar=False)
```

Il est évident que de nombreux échantillons présentent un pourcentage élevé de valeurs nulles qui ne doivent pas être prises en compte afin d'éliminer l'effet de *biais* sur les données.

```
na_cols = [col for col in secom.columns if
   secom[col]. isnull().sum() / len(secom) > 0.4]
secom_clean = secom.drop(na_cols, axis=1)
secom_clean.head()
```

Grâce aux commandes précédentes, une *liste de compréhension* (= *comprehension list*) a maintenant isolé toutes les caractéristiques ayant plus de 40 % de valeurs nulles, ce qui permet de les supprimer de l'ensemble de données.

Les caractéristiques ayant moins de 40 % de valeurs nulles doivent encore être traitées. Nous pouvons utiliser ici notre premier objet scikit-learn, le `SimpleImputer`, qui attribue des valeurs à tous les NaNs en fonction d'une stratégie définie par l'utilisateur. Dans ce cas, nous utiliserons une stratégie `moyenne`, associant la valeur moyenne supposée par l'objet à chaque NaN.

```
imputer = SimpleImputer(strategy='mean')
secom_imputed = pd.DataFrame(imputer.
   fit_transform(secom_clean))
secom_imputed.columns = secom_clean.columns
```

À titre d'exercice, nous vérifions que nous n'avons aucun zéro dans l'ensemble de données avec une autre carte thermique qui, comme on peut s'y attendre, sera uniformément foncée. Ensuite, nous pouvons passer au traitement proprement dit.

Traitement des données

Divisons notre ensemble de données en deux sous-ensembles : un ensemble de *formation* et un ensemble de *test*. Cette subdivision est nécessaire pour atténuer le phénomène de surapprentissage (= *overequipment*), par lequel l'algorithme adhère trop aux données [7], et garantit l'applicabilité du modèle à des cas différents de ceux pour lesquels il a été formé. Pour ce faire, nous utilisons la fonction `train_test_split` :

```
X_train, X_test, y_train, y_test = train_test_
   split(secom_imputed, labels, test_size=0.3)
```

Le paramètre `test_size` permet de spécifier le pourcentage de données réservées au test ; les valeurs standard de ce paramètre se situent généralement entre 0,2 et 0,3.

Il est également important de *normaliser* les données. Nous avions remarqué que les valeurs de certaines caractéristiques présentent des variations beaucoup plus fortes que d'autres, ce qui leur donne plus de poids. Le fait de les normaliser les ramène dans une seule plage de valeurs afin d'éviter les déséquilibres dus aux décalages initiaux. Pour ce faire, nous utilisons `StandardScaler` :

```
scaler = StandardScaler()
X_train = pd.DataFrame(scaler.fit_transform(X_train),
   index=X_train.index, columns=X_train.columns)
X_test = pd.DataFrame(scaler.fit_transform(X_test),
   index=X_test.index, columns=X_test.columns)
```

Il est intéressant de noter l'utilité de l'interface commune offerte par scikit-learn. Tant le `scaler` que l'`imputer` utilisent la méthode `fit_transform` pour traiter les données qui, dans des pipelines complexes, simplifient grandement l'écriture du code et la compréhension de la bibliothèque.

Nous voici enfin prêts à classer les données. En particulier, nous utiliserons une *forêt aléatoire (random forest)* [8], en obtenant, après formation, un modèle capable de distinguer les situations normales et anormales. Nous allons vérifier les performances du modèle identifié de deux manières. La première est le *score de précision (accuracy score)*. Il s'agit du pourcentage d'échantillons appartenant à la série de tests correctement classés par l'algorithme. La seconde est la *matrice de confusion* [9] qui met en évidence le nombre de faux positifs et de faux négatifs.

Créons d'abord le classificateur :

```
clf = RandomForestClassifier(n_estimators=500,
   max_depth=4)
```

Cela crée une forêt aléatoire avec 500 estimateurs dont la profondeur maximale est de 4 niveaux. Nous pouvons maintenant entraîner notre modèle sur des données d'entraînement :

```
clf.fit(X_train, y_train)
```

Une fois la formation terminée, le modèle formé est utilisé pour classer les échantillons d'essai :

```
y_pred = clf.predict(X_test)
```

Il en résulte deux étiquettes qui se rapportent à chacun des deux tests. Le premier, qui appartient à `y_test`, représente la «vérité», tandis que le second, qui appartient à `y_pred`, est la valeur prédite par l'algorithme. En les comparant, nous déterminons à la fois `accuracy` et `confusion_matrix`.

```
accuracy = accuracy_score(y_test, y_pred))
 cf = confusion_matrix(y_test, y_pred))
```

L'exemple donne les résultats suivants :

```
Model accuracy on test set is: 0.9341825902335457
The confusion matrix of the model is:
[[440    0]
 [ 31    0]]
```

La précision, d'environ 93 %, est excellente, donc le modèle semble bon. Cependant, nous constatons que le modèle est très précis pour la classification des échantillons de la classe prédominante, mais très imprécis pour la classification des échantillons de la classe minoritaire. Donc il y a manifestement un biais. Il nous faut une stratégie pour améliorer cette situation. C'est possible en suréchantillonnant les données appartenant à la classe minoritaire de manière à équilibrer, au moins partiellement, l'ensemble des données. Pour ce faire, nous utiliserons la fonction `resample` (`rééchantillonnage`) de *pandas*.

```
normals = data[data['classvalue'] == -1]
anomalies = data[data['classvalue'] == 1]
anomalies_upsampled = resample(anomalies,
   replace=True, n_samples=len(normals))
```

Cela permet d'augmenter la taille de l'ensemble de données afin de rapprocher le nombre d'échantillons normaux du nombre d'échantillons anormaux. En conséquence, nous devons redéfinir `X` et `Y` comme suit :

```
upsampled = pd.concat([normals, anomalies_upsampled])
X_upsampled = upsampled.drop('classvalue', axis=1)
y_upsampled = upsampled['classvalue']
```

En effectuant à nouveau la formation (y compris en répétant les procédures de fractionnement et de normalisation), nous obtenons les résultats suivants pour notre exemple :

```
Model accuracy on test set is: 0.8631921824104235
The confusion matrix of the model is:
[[276 41]
 [ 43 254]]
```

Visiblement la précision du modèle a diminué, probablement à cause de la plus grande hétérogénéité induite dans l'ensemble des données. Cependant, en regardant la matrice de confusion, nous remarquons immédiatement que le modèle a en réalité amélioré ses capacités de généralisation, réussissant également à classer correctement les échantillons appartenant à des situations anormales.

Conclusions et références

Dans cet article, nous avons présenté un pipeline pour l'analyse de données provenant de processus réels en Python. Manifestement chacun des sujets abordés est extrêmement vaste. Une expérience théorique et pratique est essentielle si vous voulez vous engager sérieusement dans l'analyse de données. Nous avons appris également qu'il ne faut pas s'arrêter au premier résultat obtenu, même dans des situations aussi complexes que celle dont il est question. Il convient d'interpréter de différents points de vue les résultats obtenus afin de distinguer un modèle opérationnel d'un modèle plus ou moins évidemment faussé.

Le message à retenir est donc le suivant : l'analyse des données ne peut être une discipline mécanique, elle exige au contraire une analyse critique, approfondie et variée du phénomène observé, guidée par des notions théoriques et des compétences pratiques. Je recommande vivement les références ci-dessous grâce auxquelles vous pourrez approfondir vos connaissances sur certains des aspects abordés dans l'article, ainsi que le lien vers le dépôt du GitLab où vous pourrez consulter le code écrit pour cet article.

200505-03

Article publié en italien par Elettronica Open Source (https://it.emcelettronica.com) et traduit par Elektor avec sa permission.

LIENS

[1] **Support informatique GPU MATLAB** : https://uk.mathworks.com/solutions/gpu-computing.html

[2] **Guides pour les débutants en programmation Python** : https://wiki.python.org/moin/BeginnersGuide/Programmers

[3] **Python** : www.python.org/

[4] **Bonnes pratiques d'optimisation dans MATLAB** : ttps://uk.mathworks.com/videos/best-practices-for-optimisation-in-matlab-96756.html

[5] **Ensemble de données UCI SECOM** : https://www.kaggle.com/paresh2047/uci-semcom/kernels

[6] **Guide de l'utilisateur Scikit-Learn** : https://scikit-learn.org/stable/user_guide.html

[7] **Sur- ou sous-apprentissage – un exemple complet** : https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765

[8] **Comprendre la forêt aléatoire** : https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[9] **Comprendre la matrice de confusion** : https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[10] **Dépôt GitLab pour cet article** : https://gitlab.com/eos-acard/machine-learning-in-python