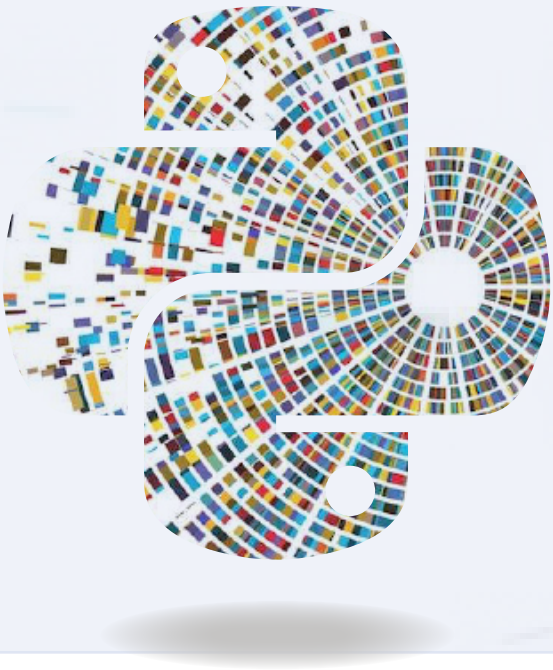


Data-analyse en kunstmatige intelligentie in Python

echte gegevens interpreteren met NumPy, pandas en scikit-learn

Er is steeds meer belangstelling voor de analyse en verwerking van gegevens die afkomstig zijn uit onze omgeving. We komen dat soort data overal tegen, variërend van de gegevens over het klimaat tot gegevens die afkomstig zijn van slimme productieprocessen. Met zoveel data zouden we, in theorie, alles kunnen beschrijven en analyseren. Maar de omgang met al die gegevens is niet eenvoudig. Daar zijn allerlei vaardigheden voor nodig, zowel theoretische als praktische. In dit artikel onderzoeken we enkele van de gebruikte technieken. We maken daarbij gebruik van Python voor de analyse van deze real-world data.

Angelo Cardellicchio (Italië)

We horen tegenwoordig overal kreten zoals *big data* en *kunstmatige intelligentie*. Daar zijn twee oorzaken voor. Ten eerste is er de toenemende en alomtegenwoordige verspreiding van data-acquisitiesystemen die het mogelijk maken om allerlei *kennisreservoirs te* creëren. Ten tweede is er de voortdurende groei van de rekencapaciteit, als gevolg van het wijdverbreide gebruik van GPGPU's (*general-purpose graphics processing units*) [1], die het mogelijk maken om rekenklussen aan te pakken waar we vroeger alleen maar van konden dromen. Laten we beginnen met de beschrijving van een toepassingsscenario dat we in dit artikel gaan gebruiken. Het gaat om het bewaken van een hele productieketen (het exacte product speelt hier geen rol). We kunnen gegevens uit allerlei bronnen verkrijgen. Zo kunnen we bijvoorbeeld sensoren plaatsen langs de hele productielijn, of gebruik maken van *contextuele informatie* over de leeftijd en het type van elke machine. Deze verzameling van gegevens, oftewel *dataset*, kan voor verschillende doeleinden worden gebruikt. Zo kunnen we bijvoorbeeld voorspellend onderhoud uitvoeren, waardoor we het optreden van abnormale situaties kunnen evalueren en voorspellen, bestellingen van reserveonderdelen kunnen plannen of reparaties kunnen uitvoeren voordat er storingen optreden, wat allemaal leidt tot kostenbesparingen en een hogere productiviteit. Bovendien stelt de kennis van de geschiedenis van de gegevens ons in staat om de gegevens uit de verschillende sensoren met elkaar te correleren, waarbij mogelijke oorzaak/gevolg-relaties aan het licht komen. Als bijvoorbeeld een plotselinge stijging van de temperatuur en de vochtigheidsgraad van de ruimte gevolgd werd door een daling van het aantal geproduceerde producten, kan het nodig zijn om veranderingen door te voeren die de klimaatomstandigheden met behulp van airconditioning constant houden. De implementatie van zo'n systeem ligt zeker niet binnen ieders bereik. Het wordt echter wel vereenvoudigd door de tools die de open source-community ter beschikking stelt. Het enige wat nodig is, is een PC (of onze vertrouwde Raspberry Pi, als de hoeveelheid te verwerken gegevens niet te groot is), kennis van Python (die u kunt verdiepen door een tutorial als [2] te volgen) en natuurlijk enige kennis van het te gebruiken 'gereedschap'. Goed, we gaan aan de slag en we gaan die samen ontdekken!

Het gereedschap

Natuurlijk moeten we programma's in Python kunnen schrijven. Daarvoor zullen we de interpreter moeten installeren. Die is te vinden op de officiële website van Python [3]. In de rest van dit artikel gaan we ervan uit dat Python al is geïnstalleerd en toegevoegd aan de omgevingsvariabelen in uw systeem.

De virtuele omgeving

Zodra Python is geïnstalleerd, kunnen we een virtuele omgeving gaan opzetten. Die wordt geïmplementeerd als een soort 'container' die los staat van de rest van ons systeem en waarin de gebruikte bibliotheken worden geïnstalleerd. De reden om een virtuele omgeving te gebruiken voor de installatie van bibliotheken heeft te maken met de snelle evolutie van de Python-wereld. Vaak ontstaan er, zelfs bij kleine updates van de interpreter, grote verschillen waardoor de bibliotheken (en dus ook de programma's) niet compa-

tibel zijn met de verschillende Python-versies. Door een deterministische omgeving te hebben waarin we de versie van elke afzonderlijke geïnstalleerde bibliotheek kennen, hebben we een soort 'garantie' dat onze programma's zullen functioneren. Het is dan voldoende om de configuratie van de virtuele omgeving exact te repliceren om er zeker van te zijn dat alles zal werken.
Voor het beheer van onze virtuele omgevingen gebruiken we een pakket dat `virtualenvwrapper` heet. Dit kan vanuit de shell worden geïnstalleerd met behulp van `pip`:

```
$ pip install virtualenvwrapper
```

Als de installatie is voltooid, creëren we als volgt een nieuwe virtuele omgeving:

```
$ mkvirtualenv ml-python
```

`ml-python` is de naam is van de virtuele omgeving die we voor ons voorbeeldscenario hebben gekozen. Het spreekt vanzelf dat u ook een andere naam mag kiezen. De ontwikkelaar kiest een naam die het beste past bij zijn of haar werkzaamheden. Vervolgens gaan we verder met het activeren van de virtuele omgeving:

```
$ workon ml-python
```

We zijn nu klaar om de elementen te installeren die nodig zijn om de rest van het artikel te volgen.

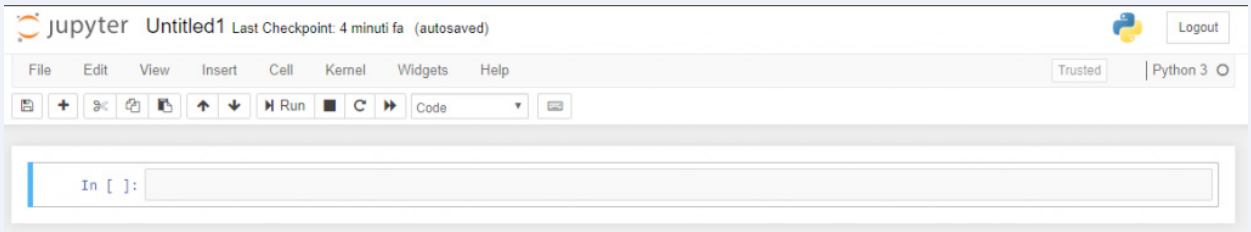
Bibliotheken

De bibliotheken die we hier presenteren en toepassen zijn de vijf meest gebruikte voor data-analyse in Python.
De eerste en misschien wel bekendste is *NumPy*, dat kan worden beschouwd als een soort interface tussen MATLAB en Python. NumPy is een bibliotheek voor algebraïsche en matrixberekeningen. Daarom zullen mensen die regelmatig met MATLAB werken veel overeenkomsten vinden, zowel wat betreft de syntax als de optimalisatie. Het gebruik van algebraïsche berekeningen in NumPy is zelfs efficiënter dan geneste lussen in MATLAB (meer informatie onder [4]). Het spreekt vanzelf dat het belangrijkste datatype in NumPy het *array* is. U moet dat niet verwarren met de corresponderende *vector*; u moet deze in algebraïsche en geometrische zin zien als een *matrix*. Omdat data-analyse gebaseerd is op algebraïsche en matrixbewerkingen, is NumPy ook de basis voor twee van de meest gebruikte frameworks: scikit-learn (dat we zometeen zullen bespreken) en TensorFlow.
Een natuurlijke aanvulling op NumPy is *pandas*, een bibliotheek die gegevens uit heterogene bronnen beheert en leest. Deze kan overweg met Excel-spreadsheets, CSV-bestanden en zelfs met JSON- en SQL-databases. pandas is uiterst flexibel en krachtig, waardoor u gegevens kunt ordenen in structuren die *dataframes* worden genoemd en die u gemakkelijk direct kunt exporteren naar NumPy-arrays.
De derde bibliotheek die we gaan gebruiken is *scikit-learn*. Scikit-learn is een framework dat is begonnen als een academisch project. Het implementeert de meeste machinale leeralgoritmes die tegenwoordig gebruikt worden en geeft ze een gemeenschappelijke interface. Precies zoals het hoort bij objectgeoriënteerd programmeren: vrijwel elk algoritme dat in scikit-learn beschikbaar is, is aan te roepen via de methode `fit_transform`. Daarbij worden tenminste twee parameters doorgegeven, zoals de geanalyseerde gegevens en de bijbehorende labels.
De laatste twee bibliotheken die we gaan gebruiken zijn *Matplotlib* en *Jupyter*. De eerste is, samen met de aanvulling *Seaborn*, nodig om de resultaten van onze experimenten in de vorm van grafieken te visualiseren. De tweede biedt ons *notebooks*, dat zijn interactieve omgevingen waarmee de data-analist stukken code onafhankelijk van elkaar kan schrijven en uitvoeren.
Voordat we ingaan op de praktijk, willen we een aantal theoretische begrippen bespreken, dat praat gemakkelijker.

De begrippen

Het eerste dat we nodig hebben, zijn *datasets*. Vaak wordt dat gewoon als vanzelfsprekend gezien. Datasets zijn verzamelingen van meetgegevens, waarmee het te observeren fenomeen wordt beschreven. Voor het gemak kunnen we denken aan een

Figuur 1. Het startscherm voor het beheer van notebooks in Jupyter.



Figuur 2. Een leeg notebook.

Excel-spreadsheet. De rijen vormen de *samples*, dat wil zeggen de *individuele waarnemingen* van het fenomeen op een bepaald moment, de kolommen bevatten de waarden van de waargenomen aspecten van het proces (*features*). Om terug te komen op het voorbeeld van een slimme productielijn: elke rij geeft de gegevens van de productieketen op een bepaald moment, elke kolom geeft de waarde van een bepaalde sensor.
Bij het bespreken van scikit-learn noemden we al even het begrip *label* of *klasse*. Het al dan niet aanwezig zijn van labels maakt het verschil tussen bewaakte en onbewaakte algoritmes. Voor een *bewaakt algoritme* is a priori kennis nodig van de klasse van elk sample in de dataset, terwijl dat bij *onbewaakte algoritmen* niet zo is. Praktisch gezien is het voor het gebruik van een bewaakt algoritme nodig dat een domeinexpert voor elk sample vaststelt tot welke *klasse* het behoort. Bij een slim fabricageproces zou een 'expert' kunnen bepalen of een set metingen, vanaf een bepaald moment, een abnormale situatie vertegenwoordigt of niet. Zo kan elk sample worden geassocieerd met één van twee mogelijke klassen (abnormaal/normaal). Dat is niet nodig voor onbewaakte algoritmen.
Verder moet onderscheid worden gemaakt tussen processen met onafhankelijke en identiek verdeelde gegevens (*independent and identically distributed* data, IID) en met gegevens in een *chronologische volgorde*. Dit verschil heeft te maken met de aard van het te observeren proces. Samples van een IID-proces zijn onafhankelijk van elkaar, terwijl in een tijdreeks elk sample afhankelijk is van een lineaire of niet-lineaire combinatie van de waarden die het proces op eerdere punten in de tijd produceert.

Aan de slag!

Met de genoemde theoretische en praktische begrippen in het achterhoofd gaan we over tot het gebruik van een geschikte dataset voor ons voorbeeld. De gebruikte dataset is *SECOM*, een acroniem dat staat voor *SEmiCOnductor Manufacturing*. Deze dataset bevat de waarden die door een aantal sensoren zijn ingelezen tijdens de bewaking van een halfgeleiderproductieproces. In de dataset, die kan worden gedownload uit verschillende bronnen (zoals Kaggle [5]), zitten 590 variabelen die elke gemeten waarde van één enkele sensor op een bepaald moment in de tijd weergeven. De dataset bevat ook labels die storingen en anomalieën onderscheiden van de goede werking van het systeem.
Als de dataset is gedownload, installeren we de bovengenoemde bibliotheken. Voer vanaf de opdrachtregel in:

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

Wanneer de bibliotheken zijn geïnstalleerd, kunnen we een eenvoudige pijplijn opzetten voor de analyse van de gegevens.

Het eerste notebook

De eerste stap is het maken van een nieuw notebook. We starten Jupyter met behulp van de volgende instructie:

```
$ jupyter-notebook
```

Er wordt dan een scherm geopend dat er uitziet zoals in **figuur 1**. We maken een notebook door *New* > *Python 3* te selecteren. Dan verschijnt een nieuw tabblad in onze browser met het nieuw aangemaakte notebook. Laten we even de tijd nemen om ons vertrouwd te maken met de interface, (zie **figuur 2**), die lijkt op een interactieve opdrachtregel, een menubalk en verschillende opties.
Het eerste wat in het oog springt is de *cell*, een deel van de weergave die we nu kunnen zien. De uitvoering van individuele cellen wordt gestart met de knop *Run* en is onafhankelijk van die van de andere cellen (we moeten er rekening mee houden dat het concept van de *scope van variabelen* geldig blijft).

Met de drie knoppen rechts van de *Run*-knop kunt u de *kernel*, d.w.z. de instantie die Jupyter associeert met ons notebook, stoppen, herstarten en resetten. Het herstarten van de instantie kan nodig zijn om de lokale en globale variabelen die geassocieerd zijn met het script te resetten, wat vooral nuttig is wanneer u experimenteert met nieuwe methoden en bibliotheken.
Een andere nuttige optie is die waarmee u het celtype kunt selecteren, waarbij u kunt kiezen tussen *Code* (d.w.z. Python-code), *Markdown* (nuttig voor het invoegen van commentaar en beschrij-

	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

Figuur 3. De eerste vijf regels van de SECOM-dataset.

	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

Figuur 4. Beknopte statistische beschrijving van de SECOM-dataset.

vingen in de indeling die bijvoorbeeld door GitHub README's wordt gebruikt), *Raw NBContent* (platte tekst) en *Heading* (voor het invoegen van titels).

Importeren en weergeven van gegevens

Als we eenmaal bekend zijn met de interface kunnen we verder gaan met de implementatie van ons script. Hier importeren we de bibliotheken en modules die we gaan gebruiken:

```
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
```

Let op de instructie `%matplotlib inline`. Die maakt het mogelijk om de door Matplotlib geproduceerde grafieken correct weer te geven. Vervolgens worden de gegevens van het bestand met de SECOM-dataset geïmporteerd met de pandas-functie `read_csv`. In dit voorbeeld is voor het gemak het relatieve pad naar het bestand hard gecodeerd. In de praktijk zou het natuurlijk beter zijn om gebruik te maken van het Python-pakket *os*, waarmee ons programma zelf het pad kan instellen.

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

Deze instructie leest de gegevens in het bestand *secom*.csv in en organiseert ze in een dataframe met de naam *data*. We kunnen de eerste vijf regels van het dataframe weergeven met de instructie `head()`, zoals te zien in **figuur 3**.

```
data.head()
```

Het weergeven van de eerste regels van het dataframe kan nuttig zijn om een eerste overzicht van de te analyseren gegevens te krijgen. In dit geval zien we meteen dat enkele waarden gelijk zijn aan '?', vermoedelijk zijn dat nulwaarden. Ook is het duidelijk dat het bereik van de waarden sterk verschilt, daar moeten we straks rekening mee houden. We kunnen ook de functie `describe()` gebruiken om een snel overzicht te krijgen van de statistische kenmerken van elke variabele (**figuur 4**).

```
data.describe()
```

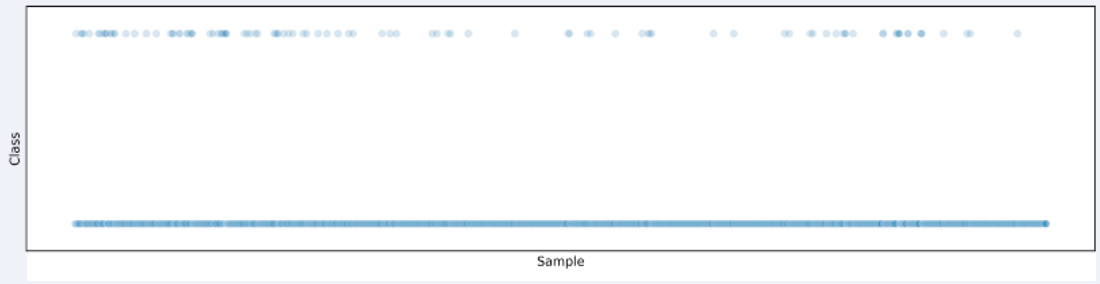
Statistische analyse kan in het algemeen omstandigheden met een gebrek aan normaliteit (d.w.z. gegevens verdeeld volgens een niet-parametrische verdeling), of de aanwezigheid van anomalieën aan het licht brengen. Om een voorbeeld te geven, merken we op dat de standaardafwijking (std) geassocieerd met de variabelen *a116* en *a118* verhoudingsgewijs vrij hoog is, dus verwachten we een grote significantie van deze variabelen bij de analyse. Aan de andere kant hebben variabelen zoals *a114* een lage std, zodat we verwachten dat ze worden verwaarloosd omdat ze weinig zeggen over het proces dat wordt geanalyseerd.
Als het laden en weergeven van het dataframe is voltooid, kunnen we overgaan tot een fundamenteel onderdeel van de pijplijn: de *voorbewerking*.

Voorbewerking van gegevens

Als eerste stap tonen we het aantal samples dat bij elke klasse hoort. We passen daarvoor de functie `value_counts()` toe op de kolom `classvalue`, omdat die de labels bevat die geassocieerd zijn met elk sample.

```
data['classvalue']. value_counts()
```

We zien dat er 1463 samples zijn verzameld in de normale bedrijfssituatie (*class -1*) en 104 in de storingssituatie (*class 1*). De dataset is dus sterk uit balans en het zou goed zijn om stappen te ondernemen om de verdeling van de monsters over de verschillende klassen 'gelijkmatiger' te maken. Dat heeft te maken met de intrinsieke



Figuur 5. Aantal samples per klasse in de SECOM-dataset.

functionaliteit van de machine learning-algoritmen die leren op basis van de beschikbare gegevens. In dit specifieke geval zal het algoritme met succes een situatie van standaardgedrag leren karakteriseren, maar zal het 'onzekerheden' hebben in het karakteriseren van abnormale situaties. De onbalans komt nog duidelijker naar voren als we kijken naar de scatterplot (zie **figuur 5**):

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

Met deze onbalans in het achterhoofd (we komen er later op terug), gaan we over tot het 'scheiden' van de labels van de gegevens:

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

Let op het gebruik van de parameter `axis` in de functie `drop`. Daarmee kunnen we aangeven dat de functie moet werken op de kolommen van het dataframe (standaard werken pandas-functies op de rijen).

Een ander feit dat blijkt uit de analyse van de dataset is dat in deze specifieke versie van SECOM-data veel kolommen gegevens van verschillend type bevatten (d.w.z. zowel strings als getallen). Daardoor kan pandas niet op unieke wijze het datatype van elke feature bepalen en wordt de definitie ervan overgelaten aan de gebruiker. Daarom is het noodzakelijk om drie pandas-functies te gebruiken om alle gegevens in numeriek formaat te brengen. De eerste functie die we zullen gebruiken is `replace()`, waarmee we alle vraagtekens kunnen vervangen door de constante waarde *numpy.nan*, de waarde die gebruikt wordt om nulwaarden in NumPy arrays te verwerken.

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

De eerste parameter van de functie is de waarde die moet worden vervangen, de tweede is de waarde die ervoor in de plaats moet komen, en de derde is een vlag die aangeeft of de eerste parameter een reguliere expressie is of niet. We kunnen ook een alternatieve syntax gebruiken met behulp van de parameter `inplace` die we instellen op `True`. Dat gaat als volgt:

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

De tweede en derde functie die we kunnen gebruiken om de bovenstaande problemen op te lossen zijn `apply()` en `to_numeric()`. Met de eerste kunnen we een bepaalde functie toepassen op alle kolommen (of rijen) van een dataframe, terwijl de tweede een kolom omzet in numerieke waarden. Door ze te combineren genereren we unieke data en verwijderen we ook de waarden die niet door NumPy en scikit-learn behandeld kunnen worden:

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

Nu moeten we evalueren welke features in de dataset daadwerkelijk nuttig zijn. We gebruiken meestal technieken (van meer of minder grote complexiteit) van *feature selection* om de redundantie en de omvang van het te behandelen probleem te verminderen Dat levert duidelijke voordelen op qua verwerkingstijd en prestaties van het algoritme. In ons geval vertrouwen we op een minder complexe techniek die werkt met het elimineren van features met een lage variantie (en dus, zoals hierboven vermeld, van geringe betekenis). We creëren daarom een interactieve widget waarmee we de verdeling van de gegevens voor elke feature kunnen visualiseren in de vorm van een histogram:

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

De *decorator* `@interact` zorgt voor *interactiviteit*, waarbij de referentiewaarde (d.w.z. `col`) varieert tussen 0 en het aantal aanwezige features in de dataset. Door de getoonde gegevens te verkennen via de widget, bepalen we hoeveel features maar één enkele waarde aannemen, wat betekent dat we ze eenvoudigweg kunnen weglaten in de analyse. We kunnen ze dan als volgt elimineren:

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

Natuurlijk zijn er meer relevante en verfijnde technieken voor de selectie van features, bijvoorbeeld met behulp van statistische parameters. Zie voor een compleet overzicht de documentatie van scikit-learn [6].

De laatste stap is het behandelen van de nulwaarden (die we eerder hebben vervangen door *np.nan*). We inspecteren de dataset om te zien hoeveel het er zijn; we gebruiken daarvoor een heatmap, zoals weergegeven in **figuur 6**, waarin de witte punten de nulwaarden vertegenwoordigen.

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

Het is duidelijk dat veel samples een hoog percentage aan nulwaarden bevatten, die niet in aanmerking moeten worden genomen, om het *bias*-effect op de gegevens weg te nemen.

```
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()
```

Dankzij de vorige commando's is nu een *comprehension list* geïsoleerd met alle features met meer dan 40% nulwaarden, waardoor ze uit de dataset kunnen worden verwijderd.
De features met minder dan 40% van de nulwaarden moeten nog worden behandeld. Hier kunnen we gebruik maken van ons eerste scikit-learn-object, de `SimpleImputer`, die waarden toekent aan alle NaN's volgens een door de gebruiker gedefinieerde strategie. In dit geval gebruiken we een *gemiddelde* (`mean`)-strategie, waarbij we aan elke NaN de gemiddelde waarde van de feature toekennen.

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

Als oefening kunnen we controleren of we geen nulwaarden in de dataset hebben door opnieuw een heatmap te maken (die, zoals we al verwachtten, overal donker is). Dan kunnen we overgaan tot de eigenlijke verwerking.

Verwerking van de gegevens

We splitsen dataset op in twee subsets: een *training* en een *test*. Deze onderverdeling is nodig om het verschijnsel van overfitting te verminderen, waarbij het algoritme 'zich te veel vasthecht' aan de gegevens (zie [7] voor meer achtergrondinformatie). Daarmee maken we het model toepasbaar op andere gevallen dan die waarop het is getraind. Hiervoor gebruiken we de `train_test_split` functie:

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

Met behulp van de parameter `test_size` kunnen we het percentage van de gegevens die gereserveerd zijn voor de test specificeren; de standaardwaarden voor deze parameter liggen meestal tussen 0,2 en 0,3.
Het is ook belangrijk om de gegevens te *normaliseren*. We hebben al gemerkt dat sommige features waarden aannemen met een veel grotere variatie dan andere, en dat zal ertoe leiden dat ze meer gewicht krijgen. Door ze te normaliseren kunt u ze binnen een enkele reeks van waarden brengen, zodat er geen onevenwichtigheden zijn als gevolg van de initiële offsets. Dit doen we met `StandardScaler`:

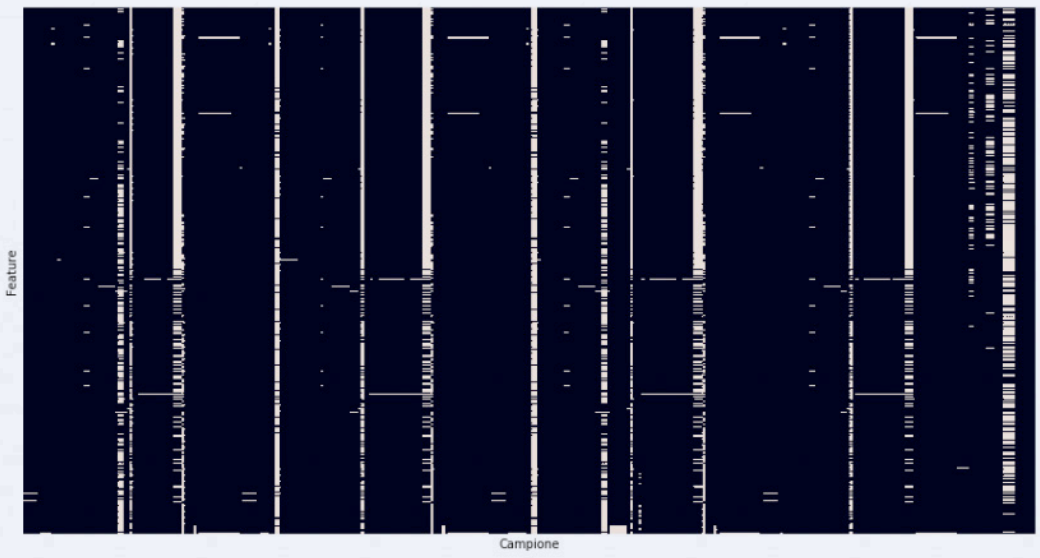
```
scaler = StandardScaler()
X_train = pd.DataFrame(scaler.fit_transform(X_train),
   index=X_train.index, columns=X_train.columns)
X_test = pd.DataFrame(scaler.fit_transform(X_test),
   index=X_test.index, columns=X_test.columns)
```

Hier blijkt het nut van de gemeenschappelijke interface die scikit-learn biedt: zowel `scaler` als `imputer` gebruiken de methode `fit_transform` om gegevens te verwerken. Dat maakt het in complexe pijplijnen veel gemakkelijker om de code te schrijven en de bibliotheek te begrijpen.

We zijn nu eindelijk klaar om de gegevens te classificeren. We gaan gebruik maken van een *random forest* [8], waarmee we na de training een model krijgen dat een onderscheid kan maken tussen normale en abnormale situaties. We verifiëren de prestaties van het geïdentificeerde model op twee manieren. De eerste is de *accuracy score*. Dat is het percentage van de samples uit de testset dat correct is geclassificeerd door het algoritme. De tweede is de *confusion matrix* [9] die het aantal fout-positieven en fout-negatieven weergeeft.
Eerst maken we de classificator:

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

Hierdoor ontstaat een random forest met 500 *estimators* met een maximale diepte van 4 niveaus. Nu kunnen we ons model trainen op trainingsgegevens:



Figuur 6. De nulwaarden in de SECOM-dataset.

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

Na afloop van de training wordt het getrainde model gebruikt om de testsamples te classificeren:

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

Dit resulteert in twee labels die betrekking hebben op elk van de twee tests. Het eerste, dat bij `y_test` hoort, vertegenwoordigt de 'waarheid', terwijl het tweede, dat bij `y_pred` hoort, de waarde is die door het algoritme wordt voorspeld. Door ze te vergelijken bepalen we zowel de `accuracy` als de `confusion_matrix`.

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

Het voorbeeld genereert de volgende resultaten:

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

De nauwkeurigheid, die rond de 93% ligt, is heel goed, dus het model ziet er goed uit. Maar het valt wel op, dat het model zeer nauwkeurig is in het classificeren van de samples van de meest voorkomende klasse, maar onnauwkeurig in het classificeren van de monsters die tot de minderheidsklasse behoren. Er is dus duidelijk sprake van een vertekening. Daarom hebben we een strategie nodig om deze situatie te verbeteren. Dit kan worden gedaan door de gegevens van de minderheidsklasse op te waarderen, zodat de dataset in ieder geval gedeeltelijk in evenwicht is. Hiervoor zullen we gebruik maken van de `resample`-functie van pandas.

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

Dit vergroot de omvang van de dataset om het aantal normale monsters dichter bij het aantal abnormale monsters te brengen. Als gevolg daarvan moeten we zowel `X` als `Y` als volgt herdefiniëren:

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

Door de training opnieuw uit te voeren (inclusief het herhalen van de splitsings- en normaliseringsprocedures) krijgen we de volgende resultaten voor ons voorbeeld:

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

Het valt meteen op dat de nauwkeurigheid van het model is afgenomen. Dat komt waarschijnlijk door de grotere heterogeniteit die in de dataset is ingebracht. Maar als we naar de confusion matrix kijken, zien we meteen dat het model zijn generalisatiemogelijkheden heeft verbeterd, en dat het ook is gelukt om samples die tot afwijkende situaties behoren correct te classificeren.

Conclusies en verwijzingen

In dit artikel hebben we een pijplijn geïntroduceerd voor de analyse van gegevens van echte processen in Python. Het is duidelijk dat de behandelde onderwerpen heel divers zijn en dat theoretische en praktische ervaring essentieel zijn voor wie zich serieus wil bezighouden met de analyse van gegevens. We hebben ook geleerd dat je niet moet stoppen bij het eerste bereikte resultaat, zelfs niet in zulke complexe situaties als hier besproken. In plaats daarvan is het noodzakelijk om de verkregen resultaten vanuit verschillende gezichtspunten te interpreteren om het verschil te ontdekken tussen een werkend model en een model dat min of meer duidelijk vertekend is.
De conclusie is dus als volgt: gegevensanalyse kan geen mechanische discipline zijn, maar vereist een kritische, diepgaande en gevarieerde analyse van het waargenomen fenomeen, waar theoretische kennis en praktische vaardigheden aan te pas komen. Als u uw kennis in een aantal aspecten die in dit artikel zijn besproken, is het aan te bevelen om de verwijzingen te volgen en ook de link naar de GitLab-repository waar u de code kunt raadplegen die voor dit artikel is geschreven.

200505-04

Dit artikel is voor het eerst gepubliceerd in het Italiaans door Elettronica Open Source (https://it.emcelettronica.com). Elektor heeft het met toestemming vertaald.

WEBLINKS

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

[2] **Beginnersgidsen voor Python-programmering**: https://wiki.python.org/moin/BeginnersGuide/Programmers

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

[4] **Best practices voor optimalisatie in MATLAB**: https://uk.mathworks.com/videos/best-practices-for-optimisation-in-matlab-96756.html

[5] **UCI SECOM-dataset**: www.kaggle.com/paresh2047/uci-semcom/kernels

[6] **Scikit-Learn gebruikershandleiding**: https://scikit-learn.org/stable/user_guide.html

[7] **Overfitting vs. underfitting: a complete example**: https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765

[8] **Understanding random forest**: https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[9] **Understanding confusion matrix**: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[10] **GitLab repository voor dit artikel**: https://gitlab.com/eos-acard/machine-learning-in-python