Sklearn_Preprocessing

August 17, 2020

Document rédigé par BOUNGOTO BIBAYI Yoanne

Attention: ce notebook converti en format pdf contient des erreur du à la conversion comme (nos formules matrices)

Data Preprocessing

C'est l'une des étapes les plus importantes pour dévélopper des modules avec le plus de performances.

Les algorithmes de machine learning apprennent à partir des données qui leur sont fournies, si ces données sont de mauvaises qualités, quelles sont erronées, incomplètes, redondantes alors l'algoritme qui en resulte sera assez mauvais, puisqu'il est censé refléter ce qu'il y'a dans les données. c'est pour cette raison qu'il est impératif de préparer les données avant leur passage dans la machine, il faut les nettoyer, les filtrer, les normaliser et c'est cette étape que l'on appelle le Preprocessing

Parmi les opérations de Preprocessing les plus importantes il y'a:

Encodage

Normalisation

Imputation

Selection

Extraction

Dans ce tutoriel nous ne verrons que la technique d'encodage et de Normalisation. on n'abordera le reste dans un autre tutoriel.

Encodage

Pour faire des calcules une machine à besoin des valeurs numériques. Si notre dataset contient des valeurs qualitatives sous forme de mots, il est indispensable de convertir ces mots en valeurs numériques (Quantitatives). Il existe 2 types d'encodages:

Encodage Ordinal

–> Label Encoder: Associe chaque catégorie à une valeur décimale unique (pour un tableau contenant une seul variable y)

$$\begin{pmatrix} \text{chat} \\ \text{chien} \\ \text{Oiseau} \end{pmatrix} - - > \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

–> Ordinal Encoder: Pour un tableau qui contient plusieurs variables x

$$\begin{pmatrix} \text{chat} \\ \text{Poils} \\ \text{chien} \\ \text{poils} \\ \text{Oiseau} \\ \text{Plumes} \end{pmatrix} - - > \begin{pmatrix} 1 \\ 1 \\ 2 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Encodage One-Hot
LabelBinarizer(), MultiLabelBinarizer, OneHotEncoder().
Ensemble nous allons juste voir LabelBinarizer() et OneHotEncoder()

$$\begin{pmatrix} \text{Chat} \\ \text{Chat} \\ \text{Chien} \\ \text{Oiseau} \end{pmatrix} -- > \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

La lecture est differente contrairement aux autres matrices dont l'identification se faisaient en ligne. Ici la classe est representée de façon binaire dans une colonne qui lui est propre. ex: chat pour colonne 1, chien colonne 2 et Oiseau colonne 3. La colonne sera la même pour les classes identiques(exemple:chat).

Autant de colonne seront créees pour le même nombre de variable ainsi les algorithmes ne pourront plus comparer chat, chien et oiseau sur un même axe car ces catégories sont desormais séparer en plusieurs dimensions

Discussion

En ce qui concerne l'encodage Ordinal d'un point de vu arithmétique ça reviendrai à dire 0<1<2 –> Oiseau< Chat <Chien ça n'a pas de sens. car ces catégories n'ont rien d'ordinaire. On risque de pénaliser la plus part des modèles machine learning à l'exception des arbres de décisions qui eux ne sont pas sensibles au rélation d'ordre. Pour éviter cela l'encodage One Hot est la solution (Car ils seront séparés sur plusieurs axes et donc ne pourront avoir aucune comparaison arithmétique possible).

Encodage

LabelEncoder

LabelEncoder

[1]:
```
import numpy as np
from sklearn.preprocessing import LabelEncoder
```

[2]:
```
y=np.array(['chat','chien','chat','Oiseau'])
```

[3]:
```
encoder=LabelEncoder()
encoder.fit_transform(y)
```

[3]:
```
array([1, 2, 1, 0], dtype=int64)
```

[4]:
```
encoder.inverse_transform(np.array([0, 0, 2, 2]))
```

[4]:
```
array(['Oiseau', 'Oiseau', 'chien', 'chien'], dtype='<U6')
```

Interprétation

On crée notre tableau y avec les données chat, chien et Oiseau On crée un objet de la classe LabelEncoder() –> encoder=LabelEncoder Puis on entraîne avec la méthodes transform_fit toutes

les données contenues dans notre variable y qu'on lui a fournit.

La méthode inverse_transform permet de décoder les données. Par exemple, Si on entraîne un modèle de machine learning à identifier des animeaux, ce modèle nous donnera les valeurs numérique 0, 1 ou 2 et que nous voulons savoir à quoi correspond chaque numéro, on pourra utiliser cette méthode.

OrdinalEncoder

[5]:
```
from sklearn.preprocessing import OrdinalEncoder
```

[6]:
```
x=np.array([['chat', 'poilu'],
            ['chien', 'poils'],
            ['poule', 'Plumes'],
            ['ourse', 'poilu']])
```

[7]:
```
encoder=OrdinalEncoder()
encoder.fit_transform(x)
```

[7]:
```
array([[0., 2.],
       [1., 1.],
       [3., 0.],
       [2., 2.]])
```

Encodage One-Hot

LabelBinarizer

[8]:
```
from sklearn.preprocessing import LabelBinarizer
```

[9]:
```
y=np.array(['Paris', 'Marseille', 'Lyon', 'Bordeaux', 'Toulouse', 'Lille'])
```

[10]:
```
encoder=LabelBinarizer()
encoder.fit_transform(y)
```

[10]:
```
array([[0, 0, 0, 0, 1, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1],
       [0, 1, 0, 0, 0, 0]])
```

Interpretation:

Vous saurez maintenant l'interpréter car nous l'avons déjà fait précédemment. Mais concentrons nous sur le résultat de notre matrice. Dans le cas ou on aura un grand nombre de catégories, dans un fichier immobilier par exemple avec un très grand nombre de villes, alors le resultat va nous donner un tableau extrêmement large. On pourrait aussi penser que ça occuperait une très grande mémoire de notre ordinateur, mais il n'en est rien, car en observant bien la matrice vous verrez quelle est essentiellement composée de zero, nous somme devant une matrice creuse, l'ordinateur ne stockera que la valeur 1.

Ainsi ne pas voir s'afficher un grand tableau, on a l'option de choisir si notre matrice doit être compressée ou non. Pour cela on mettra dans la classe LabelBinarizer(), sparse_output=True et dans le résultat vous n'aurez plus la matrice mais le resultat :Compressed Sparse Row format.

[11]:
```
encoder=LabelBinarizer(sparse_output=True)
encoder.fit_transform(y)
```

[11]:
```
<6x6 sparse matrix of type '<class 'numpy.int32'>'
        with 6 stored elements in Compressed Sparse Row format>
```

OneHotEncoder

[12]:
```
from sklearn.preprocessing import OneHotEncoder
```

[13]:
```
y=np.array([['Paris', 'Marseille'],
            ['Lyon', 'Bordeaux'],
            ['Toulouse', 'Lille']])
```

[14]:
```
encoder=OneHotEncoder()
```

[15]:
```
encoder.fit_transform(y)
```

[15]:
```
<3x6 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Row format>
```

Observation:

Vous avez remarqué que pour le résultat du transformer OneHotEncoder la compression est incluse par défaut.

Normalisation

En data science il est indispensable de normaliser nos données quantitatives, c'est à dire les mettre toutes sur une même echelle, cela facilite l'apprentissage de nos modèles machine learning qui sont basés sur les calcules de variances, la descente de gradient, les calcules de distances. il existe beaucoup de techniques de normalisation, nous allons voir les 2 plus connues à savoir: La normalisation MinMax La standardisation.

NormalisationMinMax

Transforme chaque variable X de telle sorte à être comprise entre 0 et 1

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}.$$

[16]:
```
from sklearn.preprocessing import MinMaxScaler
```

[17]:
```
X=np.array([[70],
            [80],
            [120]])
```

[18]:
```
scaler=MinMaxScaler()
scaler.fit_transform(X)
```

[18]:
```
array([[0. ],
       [0.2],
       [1. ]])
```

Discussion:

La normalisationMinMax transforme chaque variable X de telle sorte à être comprise entre 0 et 1. Vous pourriez vos dire que si on deforme nos données, ça ne veut plus rien dire, notre modèle ne va rien comprendre, ou perdra de l'information sur nos véritables données. En réalité non, on ne perd aucune information car les normalisations ont conservé les rapports de distances qu'ils y 'avaient dans nos données, c'est à dire l'écart qu'il y'a entre 0 et 0.2 est égale à l'écart entre 70 et 80et ainsi de suite.

A présent prenant une valeur test quelconque 90.

[19]:
```
x_test= np.array([[90]])
```

[20]:
```
scaler.transform(x_test)
```

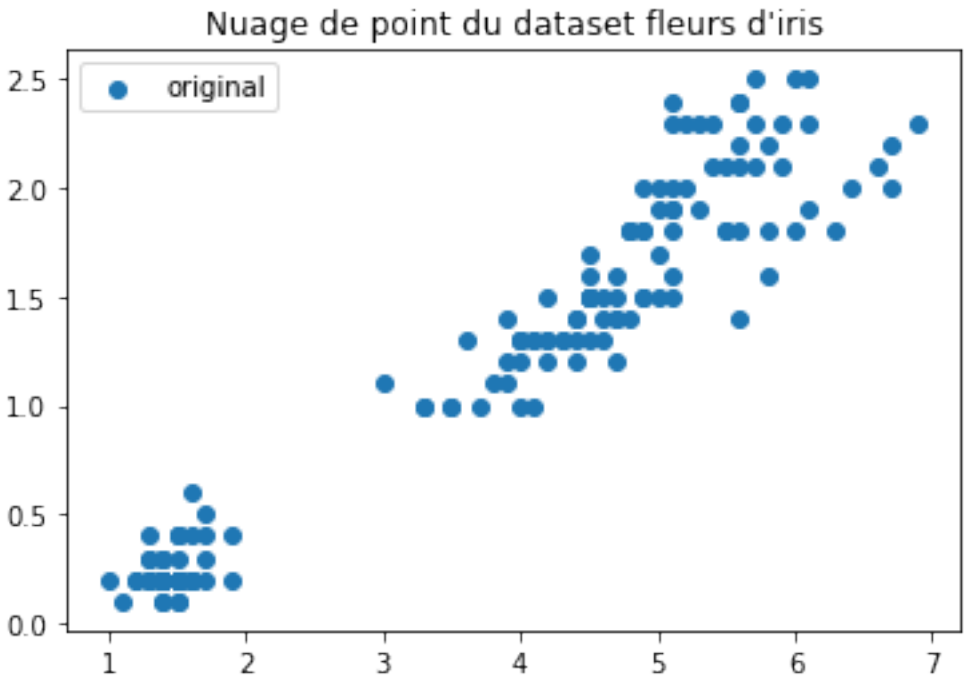
[20]:
```
array([[0.4]])
```

Notre transformer(lire en anglais), entraîné avec les données train a bien fonctionnée, elle a juste inserée la donnée x=90, dans la fonction créee avec les données train (70,80,120).

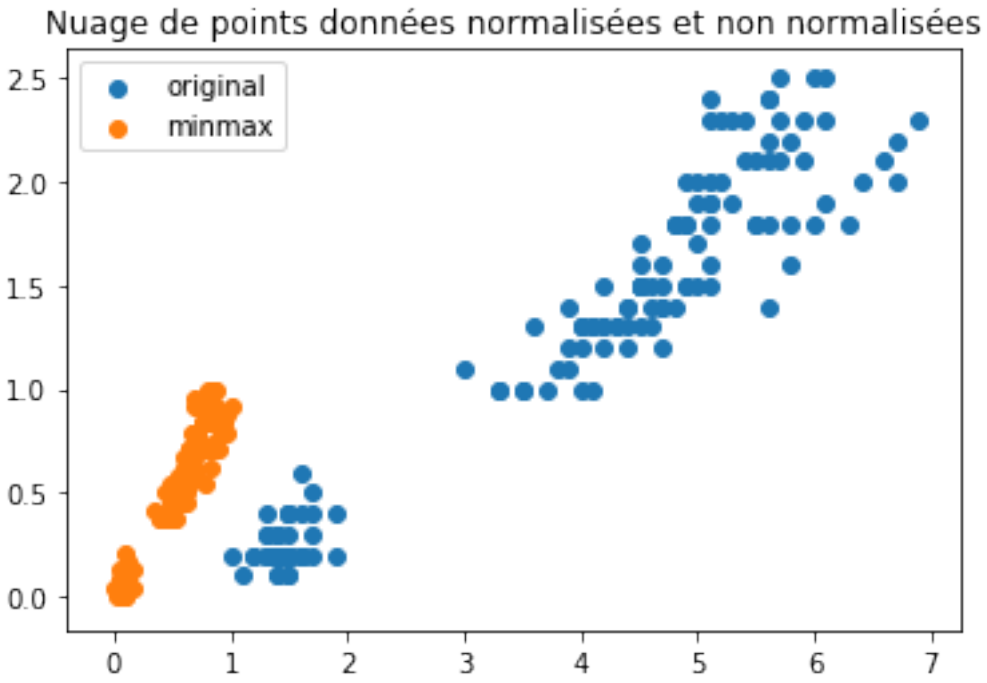
Pour bien visualiser l'effet de la nomrmalisation MinMax nous allons charger le dataset des fleurs d'iris.

[21]:
```
from sklearn import datasets
import matplotlib.pyplot as plt
```

[22]:
```
iris=datasets.load_iris()
x=iris.data
plt.scatter(x[:,2], x[:,3], label="original")
plt.title("Nuage de point du dataset fleurs d'iris")
plt.legend()
plt.show()
```



[23]:
```
X_minmax = MinMaxScaler()
X_minmax=MinMaxScaler().fit_transform(x)
plt.scatter(x[:,2], x[:,3], label="original")
plt.scatter(X_minmax[:,2], X_minmax[:,3], label="minmax")
plt.title("Nuage de points données normalisées et non normalisées")
plt.legend()
plt.show()
```



Observation:

Nous pouvons voir que nos données normaliser avec minmax sont bien comprises entre 0 et 1 (données en orange) et ces mêmes données non normalisées vont au-délà de 1.

Standardisation

Cette technique transforme nos données de telle sorte à ce que chaque variable est une moyenne est nulle et un écart-type égale à 1. Pour ça il faut soustraire de chaque valeur la moyenne de l'ensemble de données initiale et diviser le tout de l'écart-type de ces même données. On obtient ainsi des données très simple à utiliser pour la plus part des modèles statistiques.

$$X_{scale} = \frac{X - x}{\sigma}$$

[24]:
```
from sklearn.preprocessing import StandardScaler
```

[25]:
```
x_scale=np.array([[70],
                  [80],
                  [120]])
```

[26]:
```
scaler=StandardScaler()
scaler.fit_transform(x_scale)
```

[26]:
```
array([[-0.9258201 ],
       [-0.46291005],
       [ 1.38873015]])
```

Observation:

Cette technique est très utilisée, mais néanmoins elle est sensible aux valeurs aberrantes (Outliers) qui pourraient être contenues dans nos données. cela rendra nos modèles très difficiles à

exploités. Dans cette situation il faudra utiliser un transformateur très peu sensible aux Outliers qui est le RobustScaler que nous verrons très prochainement.

Conclusion:

Le preprocessing est important pour améliorer la qualité de nos données avant leur passage dans la machine. Pour ça sklearn ont développé un objet appelé transformer. Son rôle est de transformer nos données de façon cohérente, en appliquant sur les données du testset la même fonction de transformation qui a servie à traiter les données du trainset. Pour ça, les transformers disposent d'une méthode fit(X_train) et d'une méthode transform(X_test) qui peut être reuni en fit_transform().

En un mot pour résumer ce que nous avons vu, on pourrait dire: On encode les variables qualitatives et on Normalise les variables quantitatives.

[ ]:

Pour plus d'analyses aller sur ma page https://github.com/BOUNGOTO/Travaux_Machine_learning

[ ]: