femto-st SCIENCES & TECHNOLOGIES

M2 info. I2A : Applications pratiques

Jean-François COUCHOT
Université de Franche-Comté

cnrs UBFC UNIVERSITÉ BOURGOGNE-FRANCHE-COMTÉ UNIVERSITÉ DE FRANCHE-COMTÉ SUP MICRO TECH ENSMM utbm université de technologie Belfort-Montbéliard

Plan

Plan

Apprentissages supervisés

- Classification bayésienne naïve : détection de spam
- Arbres de décision : classification de données tabulaires
- Réseaux de neurones et dérivés : classification d'images

Apprentissages non supervisés

Plan

Détection de SPAM

Idées générales

1. Dataset de l'UCI [1] de 5572 SMS dont 747 étiquetés comme spam et 4825 comme non-spam
2. Étape de nettoyage : dans chaque SMS, suppression de la ponctuation, des mots courants tels déterminants, conjonctions,...
3. Étape de tokenization : conversion de chaque SMS en un vecteur représentant les fréquences de chacun des mots dans celui-ci
4. Étape d'apprentissage : Multinomial Naive Bayes, adaptation au comptage de la méthode naïve bayésienne gaussienne
5. Étape de validation et d'évaluation

1. https ://archive.ics.uci.edu/dataset/228/sms+spam+collection

Bibliothèques et dataset

```
#1
import pandas as pd
import numpy as np

#2
import string
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords

#3
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_score,
     StratifiedShuffleSplit, train_test_split

#4
spam_df = pd.read_csv('sample_data/SMSSpamCollection',
                      sep='\t',header=None, names=['spam','text'])
spam_df['spam'] = spam_df['spam'].replace({'spam': 1, 'ham': 0})
spam_df.groupby('spam').describe()
```

#1 imports de science des données

#2 imports pour le traitement de la ponctuation et des mots courants

#3 imports pour

- le comptage des token
- la construction du modèle bayésien et de son évaluation

#4 chargement du dataset et remplacement dans la colonne `spam` de “spam” par 1, 0 ailleurs

Préparation du dataset

```
#1
def message_cleaning(message):
    punc_removed = [char for char in message
                                if char not in string.punctuation]
    punc_removed_join = ''.join(punc_removed)
    message_clean = [word for word in punc_removed_join.split()
                     if word.lower() not in stopwords.words('english')]
    return message_clean

#2
vectorizer = CountVectorizer(analyzer = message_cleaning)
spamham_countvectorizer = vectorizer.fit_transform(spam_df['text'])

print(spamham_countvectorizer.toarray())
print(spamham_countvectorizer.shape)
```

#1 fonction nettoyant la chaîne de caractères `message` :

- `punc_removed_join` : chaîne de caractères `message`, sans les symboles de ponctuation
- `message_clean` : liste des mots de `punc_removed_join` non courant et en minuscule

#2

- `vectorizer` : objet CountVectorizer s'appliquant sur du texte nettoyé par `message_cleaning`
- `spamham_countvectorizer` : tableau où chaque ligne donne le nombre d'occ. des mots du texte nettoyé associé

Modèle : construction, évaluation

```python
#1
X = spamham_countvectorizer
y = spam_df['spam'].values
NB_classifier = MultinomialNB()

#2
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=0)
scores = cross_val_score(NB_classifier, X, y, cv=cv, scoring='accuracy')
mean_score = np.mean(scores)
std_deviation = np.std(scores)
print(f"Scores de validation croisée: {scores}")
print(f"Précision moyenne: {mean_score:.4f}")
print(f"Écart type des précisions: {std_deviation:.4f}")
```

#1 Définition des données et du modèle

#2 Évaluation du modèle

Modèle : utilisation

Code

```
# Définition du modèle et apprentissage sur tout le dataset
NB_classifier = MultinomialNB()
NB_classifier.fit(X,y)


# testing_sample_countvectorizer: contiendra une vectorisation des SMS
test = ["You've won a free iPhone! Click here to claim your prize now.",
        "Hey, are you free for dinner tonight?",
        """Get a free $100 gift card!
           Reply YES to this message to participate.""",
        "Just reminding you about the meeting tom at 10 am.",
        """Urgent! Your bank account has been compromised.
         Please verify your details by clicking on the link below."""]
test_countvectorizer = vectorizer.transform(test)

# Prédictions sur ces données et affichage de celles-ci
test_predict = NB_classifier.predict(test_sample_countvectorizer)
test_predict
```

Sortie

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

Plan

Classification du cancer du sein du Wisconsin [2]

Idées générales

1. Dataset du cancer du sein de Wisconsin : collection de caractéristiques médicales calculées à partir d'images de biopsies, (taille, forme, texture des cellules...) permettant de classifier des tumeurs du sein comme bénignes ou malignes
2. Dataset accessible au moyen de la fonction `load_breast_cancer()` de `sklearn`
3. Pas d'étape de nettoyage
4. Modèle d'apprentissage : xgboost
5. Recherche des hyperparmètres qui maximisent la métrique mesurant la proportion de classifications correctes (accuracy)
6. Affichage des variables qui expliquent le mieux les décisions de l'apprentissage

2. Wolberg,William, Mangasarian,Olvi, Street,Nick, and Street,W.. (1995). Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. https://doi.org/10.24432/C5DW2B.

Bibliothèques et dataset

```python
# imports usuels
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# import du modèles xgboost
from xgboost import XGBClassifier

# imports pour la data science
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, StratifiedShuffleSplit, GridSearchCV

# Chargement du dataset
data = load_breast_cancer()
X = data.data
y = data.target
```

Hyperparamètres optimisés pour l'accuracy

Code

```python
# Construction du modèle XGBoost sans paramètre particulier
model = XGBClassifier(random_state=42)

# Définir la grille de recherche des hyperparamètres
param_grid = {
    'max_depth': range(3,11,2),
    'learning_rate': [0.1, 0.01, 0.001],
    'n_estimators': [10,50,100,150]
}
# Créer un objet GridSearchCV
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=0)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
                           cv=cv, scoring='accuracy')

# Effectuer la recherche d'hyperparamètres
grid_search.fit(X,y)

print("Meilleur score obtenu :", grid_search.best_score_)
xgb_best_param = grid_search.best_params_
print("Meilleurs hyperparamètres:")
for param, value in xgb_best_param.items():
    print(f"- {param}: {value}")
```

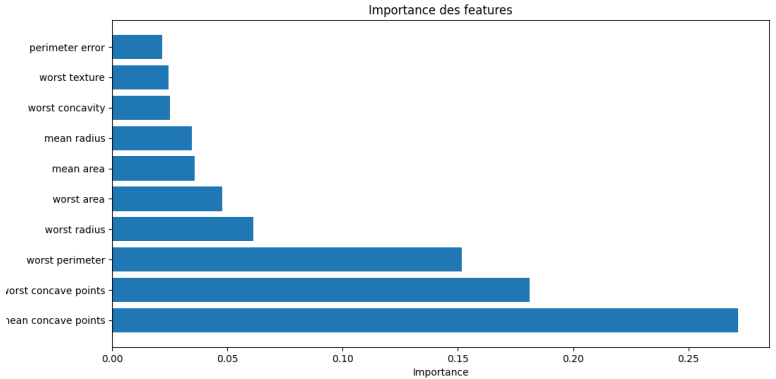
Sortie

```
Meilleur score obtenu : 0.9614035087719298
Meilleurs hyperparamètres:
- learning_rate: 0.1
- max_depth: 7
- n_estimators: 100
```

Classement des caractéristiques importantes

```python
# Instanciation du modèle avec les meilleurs hyperparamètres
xgb_optimized = XGBClassifier(**grid_search.best_params_)

# Entraînement du modèle sur les données d'entraînement
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
xgb_optimized.fit(X_train, y_train)

# Obtenir les scores d'importance (basés sur la couverture par défaut)
importance = xgb_optimized.feature_importances_

# Créer un DataFrame pour une meilleure visualisation
feature_importance = pd.DataFrame({'feature': data.feature_names, 'importance': importance})

# Trier les features par importance décroissante
feature_importance = feature_importance.sort_values('importance', ascending=False)

# Visualisation
plt.figure(figsize=(12, 6))
plt.barh(feature_importance['feature'][:10], feature_importance['importance'][:10])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Importance des features')
plt.show()
```

Classement des caractéristiques importantes



Caractéristiques par ordre décroissant d'importance pour l'accuracy de laclassification : "mean concave points", "worst concave points", "worst perimeter"

Plan

Idées générales

1. Inspiré d'une compétition kaggle [3]
2. Classification d'images (dataset CIFAR) à l'aide de réseaux de neurones (MLP) et évolutions (CNN et RESNET)
3. Préparation des donnée pour le MLP
4. Entraînements de MLP de plus en plus complets
5. Entraînement d'un convolutional Neural Network (CNN)
6. Particularisation d'un CNN déjà entraîné ResNet50

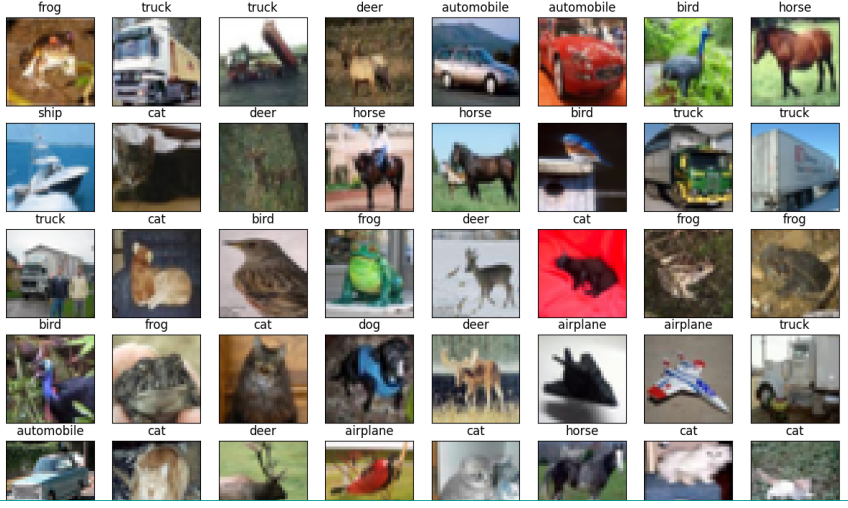
3. https://www.kaggle.com/code/yorkyong/exploring-cifar-10-w-mlp-cnn-resnet

Bibliothèques et dataset

```python
# Imports usuels
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
# Pour les MLP et les CNN
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Pour  les CNN
from tensorflow.keras.layers import Flatten, Conv2D, MaxPooling2D, Dropout, BatchNormalization
# Pour les ResNet
from keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import GlobalAveragePooling2D,  UpSampling2D
from tensorflow.keras.models import Model
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
#4 Pour vérifier qu'on exécute avec le GPU
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
#5 Chargement du jeu de données et affichage
cifar10 = tf.keras.datasets.cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('Train Images and Labels Shapes: ', x_train.shape, y_train.shape)
print('Test  Images and Labels Shapes: ', x_test.shape, y_test.shape)
#Train Images and Labels Shapes: (50000, 32, 32, 3) (50000, 1)
#Test  Images and Labels Shapes: (10000, 32, 32, 3) (10000, 1)
# images en 32*32 sur trois couches RVB
```

Cifar : 60000 images RVB 32×32

- 10 catégories différentes, 6000 images dans chaque



Préparation des données pour le MLP

```python
# Normalisation de chaque pixel vers [0,1] en exploitant minimums et maximums globaux
x_train_min, x_train_max = np.min(x_train), np.max(x_train)
x_train_normalized = (x_train - x_train_min) / (x_train_max - x_train_min)
x_test_normalized = (x_test - x_train_min) / (x_train_max - x_train_min)

# Normalisation de chaque pixel vers [-1,1] en exploitant moyenne et écart-type
x_train_mean, x_train_std = np.mean(x_train), np.std(x_train)
x_train_standardized = (x_train - x_train_mean) / x_train_std
x_test_standardized = (x_test - x_train_mean) / x_train_std

# Applatissage des images: de 32x32x3 à un vecteur de 3072 pour le MLP
x_train_flat = x_train_normalized.reshape(x_train_normalized.shape[0], -1)
x_test_flat = x_test_normalized.reshape(x_test_normalized.shape[0], -1)
```

Première proposition arbitraire de MLP

```python
# Modèle de type MLP
model = Sequential(name="MLP_model")

# Première tentative de configuration : une couche avec 256 neurones et une avec 10 en sortie (= nombre de classe)
model.add(Dense(256, activation='relu', input_shape=(3072,))) # 3072 = 32x32x3
model.add(Dense(10, activation='softmax'))
optimizer = Adam() # Adam est une descente de gradient optimisée
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model on the CIFAR-10 training set
history = model.fit(x_train_flat, y_train,
                    epochs=10, # on va parcourir 10 fois le jeu de données
                    batch_size=64, # on ne mets à jour les poids toutes les 64 données
                    validation_split=0.1) # 10% du training_data réservé pour de la validation

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test_flat, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Epoch 1/10
704/704 -------------------- 12s 14ms/step - accuracy: 0.2792 - loss: 2.0657 - val_accuracy: 0.3656 - val_loss: 1.7964
...
Epoch 10/10
704/704 -------------------- 13s 19ms/step - accuracy: 0.4690 - loss: 1.4968 - val_accuracy: 0.4606 - val_loss: 1.5251
Test Loss: 1.5279182195663452
Test Accuracy: 0.46480000019073486... plutôt faible
```

Pistes d'amélioration du MLP : les modèles

Hyperparamètres optimisables

- Nombre de couches, de neurones sur chacune d'elles et fonction d'activation (pas adressé ici)
- Nombre d'époques : augmenter le nombre de vues des données (attention au sur-apprentissage !)
- Taille du batch : augmenter peut permettre d'éviter la particularisation aux données limites

```python
def create_mlp_model_1():
    return Sequential([
        # Hidden layer with 128 neurons and ReLU activation
        Dense(128, activation='relu', input_shape=(3072,)), # 3072 = 32x32x3
        # Second hidden layer with 128 neurons and ReLU activation
        Dense(128, activation='relu'),
        # Output layer with softmax activation for multi-class classification
        Dense(10, activation='softmax')])

def create_mlp_model_2():
    return Sequential([
        Dense(256, activation='relu', input_shape=(3072,)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')])

def create_mlp_model_3():
    return Sequential([
        Dense(512, activation='relu', input_shape=(3072,)),
        Dense(256, activation='relu'),
```

Pistes d'amélioration du MLP : compilation et apprentissage

Fonction de compilation du modèle et d'apprentissage

```python
def compile_and_train_model(model):
    optimizer = Adam()
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    model.fit(x_train_flat, y_train, epochs=20, batch_size=128, validation_split=0.1)
    test_loss, test_accuracy = model.evaluate(x_test_flat, y_test)
    print("Test Loss:", test_loss)
    print("Test Accuracy:", test_accuracy)
    return model
```

Apprentissage : demande et résultats

```python
mlp_model_1 = create_mlp_model_1()
mlp_model_1.summary()
compile_and_train_model(mlp_model_1)

mlp_model_2 = create_mlp_model_2()
mlp_model_2.summary()
compile_and_train_model(mlp_model_2)

mlp_model_3 = create_mlp_model_3()
mlp_model_3.summary()
compile_and_train_model(mlp_model_3)
```

```
model_3...
Epoch 1/20
352/352---18s 46ms/step - accur: 0.253 - loss: 2.049 - val_accur: 0.375 - val_loss: 1.754
...
Epoch 20/20
352/352 --17s 49ms/step - accur: 0.599 - loss: 1.122 - val_accur: 0.523 - val_loss: 1.383
Test Loss: 1.3905723094940186
Test Accuracy: 0.5142999887466431 => léger sur-apprentissage, mais accuracy > 50%
```

Convolutional Neural Network : création du modèle

```
model = Sequential(name="CNN_model")

# 2 couches avec des filtres de convolution de taille 3x3
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2))) # slection des valeur max dans les fenêtres 2x2
model.add(Dropout(0.25))  # désactivation de 25% des neurones
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))  # Dropout layer

model.add(Flatten()) # applatissement pour passer à des RN classiques
model.add(Dense(128, activation='relu')) # RN classique
model.add(BatchNormalization())  # normalisation
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))  # sortie

# Compilation
optimizer = Adam(learning_rate=0.0001)  # Use Adam optimizer
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Affichage
model.summary()
```

Convolutional Neural Network : entraînement du modèle

Code

```python
# Entraînement du modèle
history = model.fit(x_train_normalized, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.1)  # Use 10% of training data as validation

# Evaluate the CNN model on the test set
test_loss, test_accuracy = model.evaluate(x_test_normalized, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

Sorties

```
Epoch 1/10
352/352 -------------------- 16s 25ms/step - accuracy: 0.1929 - loss: 2.4694 - val_accuracy: 0.3846 - val_loss: 2.0150
...
Epoch 10/10
352/352 -------------------- 4s 10ms/step - accuracy: 0.6434 - loss: 1.0235 - val_accuracy: 0.6856 - val_loss: 0.9039
313/313 -------------------- 2s 3ms/step - accuracy: 0.6803 - loss: 0.9180
Test Loss: 0.9267022609710693
Test Accuracy: 0.6773999929428101
```

ResNet : nettoyage des données

```
#reload des données
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
# Preprocess input images
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Preprocess input images using ResNet50 preprocessing
x_train = preprocess_input(x_train)
x_test = preprocess_input(x_test)

# dans (x_train,y_train) extraction de 10% pour construire (x_val,y_val)
x_train,x_val,y_train,y_val=train_test_split(x_train,y_train,test_size=.1)

# Encodage binaire des classes : 1 vecteur de 10 bits, tous à 0 sauf 1
y_train=to_categorical(y_train)
y_val=to_categorical(y_val)
y_test=to_categorical(y_test)

#Print the dimensions of the datasets to check
print((x_train.shape,y_train.shape))
print((x_val.shape,y_val.shape))
print((x_test.shape,y_test.shape))
```

ResNet : Augmentation du volume de données

Code

```
train_generator = ImageDataGenerator(rotation_range=10, width_shift_range=0.1, height_shift_range=0.1,
                                     horizontal_flip=True, shear_range=0.1, zoom_range=0.1)

val_generator = ImageDataGenerator(rotation_range=10, width_shift_range=0.1, height_shift_range=0.1,
                                     horizontal_flip=True, shear_range=0.1, zoom_range=0.1)

test_generator = ImageDataGenerator(rotation_range=10, width_shift_range=0.1, height_shift_range=0.1,
                                     horizontal_flip=True, shear_range=0.1, zoom_range=0.1)
```

ImageDataGenerator pour augmenter le nombre d'images

- Outil permettant de générer de nouvelles images selon les caractéristiques imposées
  - Angle de rotation : ici entre -10° et +10°
  - Zoom aléatoire des images jusqu'à 10% dans chaque direction
  - ...

ResNet : particularisation du modèle

```
inputs = tf.keras.layers.Input(shape=(32, 32, 3))

# Upsample the input image to match the size expected by ResNet50
resized_inputs = UpSampling2D(size=(7, 7))(inputs)

# Load the ResNet50 model with pretrained weights
resnet_model = ResNet50( include_top=False, eights='imagenet', input_tensor=resized_inputs)

# Gèle toutes les couches, sauf les 30 dernières
for layer in resnet_model.layers:
    layer.trainable = False
for layer in resnet_model.layers[-30:]:
    layer.trainable = True

# Global average pooling and classification layers
x = GlobalAveragePooling2D()(resnet_model.output)
x = Dense(1024, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
x = Dense(512, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.2)(x)
x = Dense(256, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.2)(x)
classification_output = Dense(10, activation="softmax", name="classification")(x)

# Connect the feature extraction and "classifier" layers to build the model
ResNet_model = Model(inputs=inputs, outputs=classification_output, name="ResNet")
```

ResNet50 : entraînement du modèle

Code

```
# Here, the learning rate will be reduced by half (factor=0.5) if no improvement in validation loss is observed for 1 epochs
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=1, min_lr=0.00001)

# Arret prématuré si val_loss ne diminue pas durant 5 époques
# Poids du modèle ayant la val_los la plus faible restaurés
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True, verbose=1)
batch_size=64
history = ResNet_model.fit(train_generator.flow(x_train,y_train,batch_size=batch_size),
                                    epochs=15,
                                    steps_per_epoch=x_train.shape[0]//batch_size,
                                    validation_data=val_generator.flow(x_val,y_val,batch_size=batch_size),
                                    callbacks=[reduce_lr, early_stopping],
                                    shuffle = True)
test_loss, test_accuracy = ResNet_model.evaluate(test_generator.flow(x_test,y_test,batch_size=batch_size))
print("Test Loss, test accuracy:", test_loss, test_accuracy)
```

Sorties

```
Epoch 1/15
703/703 -- 213s 261ms/step - accur: 0.625 - loss: 1.133 - val_accur: 0.746 - val_loss: 0.761 - learning_rate: 0.001
Epoch 15/15
703/703 -- 167s 237ms/step - accur: 0.943 - loss: 0.167 - val_accur: 0.895 - val_loss: 0.315 - learning_rate: 1.25e-04
Restoring model weights from the end of the best epoch: 14.
Test Loss, test Accury: 0.3173533082008362 0.8913999795913696
```

Plan

Plan

Regroupement de comportements similaires

Idées générales

1. Données : séries chronologiques de ventes au détail, accessible sur kaggle [4]
2. Question : regrouper les vendeurs au comportement similaire
3. Inspiré du notebook [5]
4. Étape de nettoyage : uniformisation de la taille des séries et normalisation entre 0 et 1
5. Estimation d'un nombre acceptable de regroupements
6. Mise en place du regroupement et analyse

4. https://www.kaggle.com/datasets/census/retail-and-retailers-sales-time-series-collection/data
5. https://www.kaggle.com/code/izzettunc/introduction-to-time-series-clustering

Préliminaires et chargement

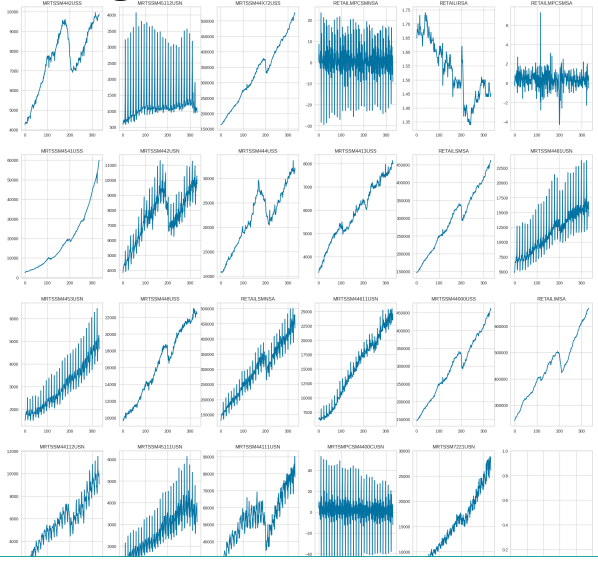
Bibliothèques

```
# pour method elbow
!pip install yellowbrick

# Bibliotheques usuelles
import os
import math
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
# Preprocessing et clustering
from sklearn.preprocessing import MinMaxScaler
from yellowbrick.cluster import KElbowVisualizer
from sklearn.cluster import KMeans
```

Chargement des données et affichage

```
mySeries, namesofMySeries = [], []
directory = './retail_data/'
for filename in os.listdir(directory):
    if filename.endswith(".csv"):
        df = pd.read_csv(directory+filename)
        df = df.loc[:,["date","value"]]
        df.set_index("date",inplace=True)
        df.sort_index(inplace=True)
        mySeries.append(df)
        namesofMySeries.append(filename[:-4])

fig, axs = plt.subplots(4,6,figsize=(25,25))
fig.suptitle('Series')
for i in range(4):
    for j in range(6):
        if i*6+j+1>len(mySeries):
            continue
        axs[i, j].plot(mySeries[i*6+j].values)
        axs[i, j].set_title(namesofMySeries[i*6+j])
plt.show()
```

Données originales



Intuition de clusters :

1. croissance avec 1 décrochage : `MRTSSM44X72USS`, `RETAILSMSA`, `MRTSSM44000USS`, ...
2. croissance avec oscillations : `MRTSSM4453USN`, `MRTSSM4541USS`, `MRTSSM45111USN`,...
3. constantes avec oscillations : `MRTSMPCSM4400CUSN`, `RETAILMPCSMSA`, `RETAILMPCSMNSA`
4. décroissante : `RETAILIRSA`

Preprocessing –1

Analyse des données

```
series_lengths = {len(series) for series in mySeries}
max_len = max(series_lengths)
print(series_lengths)
ind = 0
for series in mySeries[:5]:
    print("["+str(ind)+"] "
            +series.index[0]+" "
            +series.index[len(series)-1])
    ind+=1
max_len = max(series_lengths)
longest_series = None
for series in mySeries:
    if len(series) == max_len:
        longest_series = series
print(longest_series)
```

Sortie

```
{332, 333} # 2 tailles différentes : 332 ou 333
[0] 1992-01-01 2019-09-01
[1] 1992-01-01 2019-09-01
[2] 1992-01-01 2019-09-01
[3] 1992-02-01 2019-09-01
[4] 1992-01-01 2019-09-01
# la série [3] commence 1 mois plus tard
date        value
1992-01-01   6887
1992-02-01   6937
...           ...
2019-08-01  28734
2019-09-01  26084
# les séries vont être indexées sur cette séquence de date
```

Preprocessing –2

Réindexation des données et normalisation

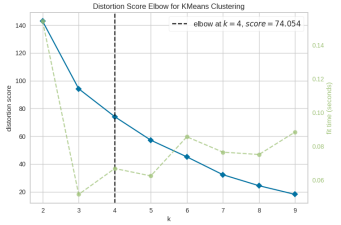
```
#reindexation (complêtée avec Nan, puis par interpolation linéaire)
for i in range(len(mySeries)):
    if len(mySeries[i])!= max_len:
        mySeries[i] = mySeries[i].reindex(longest_series.index)
        mySeries[i].interpolate(limit_direction="both",inplace=True)

# normalisation de chaque série (sur [0,1]), puis données de chaque série mises en ligne
for i in range(len(mySeries)):
    scaler = MinMaxScaler()
    mySeries[i] = MinMaxScaler().fit_transform(mySeries[i])
    mySeries[i]= mySeries[i].reshape(len(mySeries[i]))

# mise au format numpy.array
ms = np.array(mySeries)
```

Recherche du nombre declusters selon Elbow [6]

Code

```
km = KMeans(random_state=42)

visualizer = KElbowVisualizer(km, k=(2,10))
visualizer.fit(ms)
visualizer.show()
```

Sortie



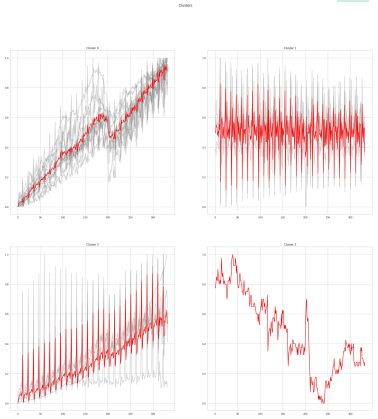
- 4 clusters semble adapté

6. https://en.wikipedia.org/wiki/Elbow_method_(clustering)

Application directe de kmeans

```
#clustering
cluster_count = 4
kmeans = KMeans(n_clusters=cluster_count,max_iter=100)
labels = kmeans.fit_predict(ms)
#affichage
plot_count = math.ceil(math.sqrt(cluster_count))
fig, axs = plt.subplots(plot_count,plot_count,figsize=(25,25))
fig.suptitle('Clusters')
row_i=0
column_j=0
for label in set(labels):
    cluster = []
    for i in range(len(labels)):
            if(labels[i]==label):
                axs[row_i, column_j].plot(mySeries[i],c="gray",alpha=0.4)
                cluster.append(mySeries[i])
    if len(cluster) > 0:
        axs[row_i, column_j].plot(np.average(np.vstack(cluster),axis=0),c="red")
    axs[row_i, column_j].set_title("Cluster "+str(row_i*plot_count+column_j))
    column_j+=1
    if column_j%plot_count == 0:
        row_i+=1
        column_j=0
plt.show()
```

Affichage, moyennes en rouge



Affichage des clusters

Code

```
fancy_names_for_labels = [f"Cluster {label}" for label in labels]
pd.DataFrame(zip(namesofMySeries,fancy_names_for_labels),
             columns=["Series","Cluster"]).sort_values(by="Cluster").set_index("Series")
```

Résultats

- Cluster 0 : MRTSSM442USS, MRTSSM44111USN, MRTSSM44112USN, RETAILIMSA, MRTSSM44000USS, MRTSSM44611USN, RETAILSMNSA, MRTSSM448USS, RETAILSMSA, MRTSSM7221USN, MRTSSM444USS, MRTSSM442USN, MRTSSM44X72USS, MRTSSM4413USS
- Cluster 1 : MRTSMPCSM4400CUSN, RETAILMPCSMSA, RETAILMPCSMNSA
- Cluster 2 : MRTSSM4453USN, MRTSSM4541USS, MRTSSM45111USN, MRTSSM45112USN, MRTSSM4481USN
- Cluster 3 : RETAILIRSA