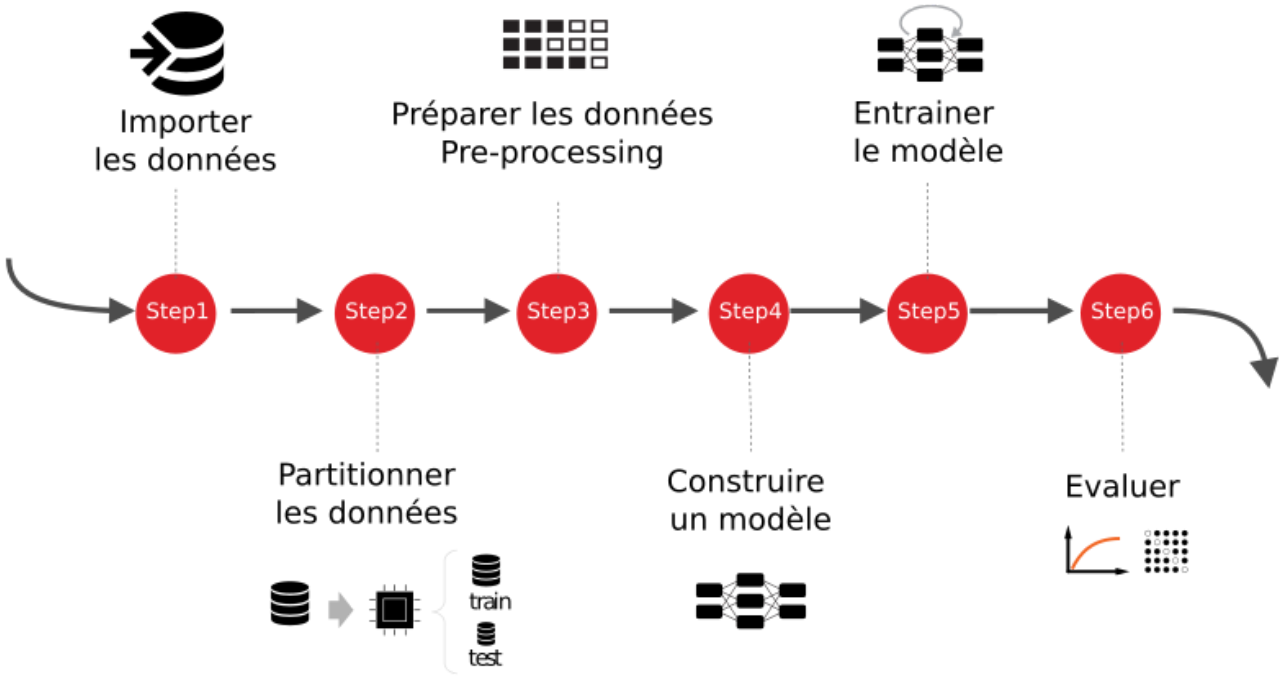
PT2 : notes/commentaires de code



1. Télécharger les données

- soit en local sur votre machine
- soit sur un drive pour ceux qui travaillent sous colab

solution pour ceux qui travaillent sous google drive

télécharger le fichier dataPT2.csv et le placer dans un dossier "PT2" sur votre drive

> il semble que moodle dezippe automatiquement le fichier lors du download ????

```
from google.colab import drive
drive.mount('/content/drive')

import pandas as pd
df = pd.read_csv('/content/drive/My Drive/PT2/dataPT2.csv',sep=";",
 decimal=',')
```

2. Partitionner les données

importer la fonction de sklearn

```
from sklearn.model_selection import train_test_split
```

séparer les outcome y des features X

```
# Drop all rows where the 'hospital_death' column is NaN
df_filtered = df[~df['hospital_death'].isnull()]

# Separate features, outcome and predictions calculated by MedDocs
X = df_filtered.drop(['hospital_death', 'apache_4a_icu_death_prob',
'apache_4a_hospital_death_prob'], axis=1)
y = df_filtered['hospital_death']
medProb = df_filtered[['apache_4a_icu_death_prob',
'apache_4a_hospital_death_prob']]
```

Split the data

```
# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
            test_size=0.10, random_state=42, stratify = y)

# Print the shape of the training and test sets
print('Training set shape:', X_train.shape)
print('Test set shape:', X_test.shape)
```

> on peut stratifier sur y pour s'assurer d'avoir les mêmes répartitions de 0 et de 1 dans y dans le test et dans le train !!

On peut également faire le choix de spliter deux fois pour avoir un test et un valid

```
# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 random_state=42)

# Split the training set into train and validation sets
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test,
test_size=0.5, random_state=42)

# Print the shape of the train, test, and validation sets
print('Train set shape:', X_train.shape)
print('Test set shape:', X_test.shape)
print('Validation set shape:', X_val.shape)
```

pour la suite de l'exemple, je sélectionne au hasard quelques colonnes (mais vous allez faire beaucoup mieux...)

```
X_train = X_train.iloc[:,29:45]
X_test = X_test.iloc[:,29:45]
```

Le pipeline de sklearn permet de combiner les étapes de pré-processing avec celles du modèles. C'est donc très pratique pour appliquer les mêmes transformations au jeu de test pour l'étape de prédiction.

Getting Started — scikit-learn 1.3.1 documentation

A titre d'illustration, j'ai utilisé un scaler standard et une imputation par la moyenne

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler


# Create a pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('imputer', SimpleImputer(strategy='mean')),
    ('logistic_regression', LogisticRegression())
])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions on the test data
y_pred = pipeline.predict(X_test)
```

pour R, voir `pipeliner` package

5. Evaluer les résultats

Une matrice de confusion est très pratique pour visualiser les résultats et la qualité des prédictions binaires.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Create the confusion matrix
confusion_matrix = confusion_matrix( y_test, y_pred)

# Print the confusion matrix
print(confusion_matrix)
```

C'est bien mais c'est pas très lisible. Un display est prévu pour rendre les choses visuellement plus simples

```
# Create the ConfusionMatrixDisplay object
confusion_matrix_display = ConfusionMatrixDisplay(confusion_matrix,
display_labels=["0", "1"])

# Plot the confusion matrix
confusion_matrix_display.plot()
```

Ensuite, il est intéressant de mesurer quelques métriques pour quantifier les qualité de la prédiction

```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score


# Calculez les métriques de performance
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred) #PPV
recall = recall_score(y_test, y_pred) #Sensibility
f1 = f1_score(y_test, y_pred) #F1 = 2. (precision.recall)/(precision + recall)

# Imprimez les métriques de performance
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 score:", f1)
```

On se rappelera de leurs définitions

Precision and recall - Wikipedia

> Si on utilise souvent les mêmes lignes de code, c'est pratique de les mettre sous forme de fonction

```
def printMetrics(y_test, y_pred):
  # Calculez les métriques de performance
  accuracy = accuracy_score(y_test, y_pred)
  precision = precision_score(y_test, y_pred)
  recall = recall_score(y_test, y_pred)
  f1 = f1_score(y_test, y_pred)

  # Imprimez les métriques de performance
  print("Accuracy:", accuracy)
  print("Precision:", precision)
  print("Recall:", recall)
  print("F1 score:", f1)


printMetrics(y_test, y_pred)
```

L'exactitude peut paraitre bonne (bien prédit dans ~92% des cas). Mais attention, ça peut être trompeur...

Un Dummy Classifieur: le minimum à battre !!!

```
from sklearn.dummy import DummyClassifier

# Create a dummy classifier with the 'most_frequent' strategy
dummy_classifier = DummyClassifier(strategy='most_frequent')

# Fit the dummy classifier to the training data
dummy_classifier.fit(X_train, y_train)

# Make predictions on new data
predDummy = dummy_classifier.predict(X_test)

confusion_matrixDummy = confusion_matrix( y_test, predDummy)

# Create the ConfusionMatrixDisplay object
confusion_matrix_dummy_display = ConfusionMatrixDisplay(confusion_matrixDummy,
display_labels=["0", "1"])

# Plot the confusion matrix
confusion_matrix_dummy_display.plot()

printMetrics(y_test, predDummy)
```

6. Gérer les classes déséquilibrées

Under or OverSampling

1. Introduction — Version 0.11.0

L'undersampling consiste à n'utiliser qu'une partie (sélectionnée au hasard) des données de la classe majoritaire. En conséquence, le déséquilibre entre les deux classe est réduit (on peut choisir avec l'argument sampling_strategy la répartition entre les classes)

```
import numpy as np
from imblearn.pipeline import Pipeline
from imblearn.under_sampling import RandomUnderSampler
#from imblearn.over_sampling import RandomOverSampler

#over = RandomOverSampler(sampling_strategy = .1)
under = RandomOverSampler(sampling_strategy = 0.3)

X_resampled, y_resampled = under.fit_resample(X_train, y_train)

# Create a pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('imputer', SimpleImputer(strategy='mean')),
    ('logistic_regression', LogisticRegression())
])

# Fit the pipeline to the training data
pipeline.fit(X_resampled, y_resampled )

# Make predictions on the test data
y_pred_under = pipeline.predict(X_test)

printMetrics(y_test, y_pred_over)
```

SMOTE

SMOTE (Synthetic Minority Oversampling TEchnique) permet d'augmenter les nombres sous-représentés dans l'ensemble de données d'un modèle d'apprentissage automatique. SMOTE synthétise les nouvelles instances minoritaires similaires aux instances minoritaires réelles.

https://kobia.fr/imbalanced-data-smote/

```
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression

# Create a pipeline with a SMOTE resampler and a LogisticRegression estimator
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('imputer', SimpleImputer(strategy='mean')),
    ('smote', SMOTE(k_neighbors=10, sampling_strategy='auto',
random_state=42)),
    ('logistic_regression', LogisticRegression())
])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions on new data
y_pred = pipeline.predict(X_test)

printMetrics(y_test, y_pred)
```

ADASYN : une alternative à SMOTE

https://medium.com/@ruinian/an-introduction-to-adasyn-with-code-1383a5ece7aa

```
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import ADASYN
from sklearn.linear_model import LogisticRegression

# Create a pipeline with an ADASYN resampler and a LogisticRegression estimator
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('imputer', SimpleImputer(strategy='mean')),
    ('adasyn', ADASYN(n_neighbors=5, random_state=42)),
    ('logistic_regression', LogisticRegression())
])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions on new data
y_pred = pipeline.predict(X_test)

printMetrics(y_test, y_pred)
```