IDL TP Final `scikit-learn`

```
from IPython.display import display
```

scikit-learn ?

Documentation scikit-learn.

Scikit-learn est une bibliothèque Python dédiée à l'apprentissage artificiel qui repose sur NumPy et SciPy. Il est écrit en Python et Cython. Il s'interface très bien avec matplotlib, seaborn ou pandas (qui lui-même marche très bien avec plotnine). C'est devenu un incontournable du *machine learning* et des *data sciences* en Python.

Dans ce notebook on se limitera à la classification, une partie seulement de ce qu'offre `scikit-learn`.

La classification est souvent utilisée en TAL, par exemple dans les tâches d'analyse de sentiment, de détection d'émotion ou l'identification de la langue.

On va faire de l'apprentissage supervisé de classifieurs : l'idée est d'apprendre un modèle à partir de données réparties en classes (une classe et une seule pour chaque exemple), puis de ce servir de ce modèle pour répartir parmi les mêmes classes des données nouvelles.

Dit autrement, on a un échantillon d'entraînement $\mathcal{D}$, composé de n couples $(X_i, Y_i), i = 1, \ldots, n$ où les X_i sont les entrées (en général des **vecteurs** de traits ou *features*) et les y_i seront les sorties, les classes à prédire. On cherche alors dans une famille $\mathbb{M}$ de modèles un modèle de classification M qui soit le plus performant possible sur $\mathcal{D}$.

`scikit-learn` offre beaucoup d'algorithmes d'apprentissage. Vous en trouverez un aperçu sur cette carte et sur ces listes : supervisé / nonsupervisé.

Mais `scikit-learn` offre également les outils pour mener à bien les étapes d'une tâche de d'apprentissage :

- Manipuler les données, constituer un jeu de données d'entraînement et de test

- Entraînement du modèle
- Évaluation
- Optimisation des hyperparamètres

```
pip install -U scikit-learn
```

Un premier exemple

Les données

Comme vous le savez, c'est la clé de voute du *machine learning*. Nous allons travailler avec un des jeux de données fourni par scikit-learn : le jeu de données de reconnaissance des vins

C'est plus facile pour commencer parce que les données sont déjà nettoyées et organisées, mais vous pourrez bien sûr par la suite charger des données venant d'autres sources.

```
from sklearn import datasets
wine = datasets.load_wine()
type(wine)
```

La recommandation des développeurs et développeuses de `scikit-learn` est d'importer uniquement les parties qui nous intéresse plutôt que tout le package. Notez aussi le nom `sklearn` pour l'import.)

Ces jeux de données sont des objets `sklearn.utils.Bunch`. Organisés un peu comme des dictionnaires Python, ces objets contiennent :

- `data` : array NumPy à deux dimensions d'échantillons de données de dimensions `(n_samples, n_features)`, les inputs, les X
- `target` : les variables à prédire, les catégories des échantillons si vous voulez, les outputs, les y
- `feature_names`
- `target_names`

Et d'autres trucs comme

```
print(wine.DESCR)
```

```
wine.feature_names
```

```
wine.target_names
```

Après avoir installé `pandas` ou `polars` :

```
pip install -U pandas polars
```

On peut convertir ces données en `DataFrame` pandas si on veut.

```
import pandas as pd

df = pd.DataFrame(data=wine.data,columns=wine.feature_names)
df["target"] = wine.target
df.head()
```

```
import polars as pl

df = pl.DataFrame(
    data=wine.data, schema=wine.feature_names).with_columns(
    target=pl.Series(wine.target)
)
df.head()
```

Mais l'essentiel est de retrouver nos inputs X et outputs y nécessaires à l'apprentissage.

```
X_wine, y_wine = wine.data, wine.target
```

```
X_wine.shape
```

```
y_wine
```

Vous pouvez séparer les données en train et test facilement à l'aide de `sklearn.model_selection.train_test_split` (voir la doc)

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_wine, y_wine, test_size=0.3)
y_train
```

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.hist(y_train, align="right", label="train")
plt.hist(y_test, align="left", label="test")
plt.legend()
plt.xlabel("Classe")
plt.ylabel("Nombre d'exemples")
plt.title("Répartition des classes")
plt.show()
```

Il ne faut pas hésiter à recourir à des représentations graphiques quand vous manipulez les données.
Ici on voit que la répartition des classes à prédire n'est pas homogène pour les données de test.
On peut y remédier en utilisant le paramètre `stratify`, qui fait appel à `StratifiedShuffleSplit` pour préserver la même répartition des classes dans le train et dans le test.

```
X_train, X_test, y_train, y_test = train_test_split(X_wine, y_wine, test_size=0.25, strati
plt.hist(y_train, align="right", label="train")
plt.hist(y_test, align="left", label="test")
plt.legend()
plt.xlabel("Classe")
plt.ylabel("Nombre d'exemples")
plt.title("Répartition des classes avec échantillonnage stratifié")
plt.show()
```

Entraînement

L'étape suivante est de choisir un algorithme (un *estimator* dans la terminologie de scikit-learn), de l'entraîner sur nos données (avec la fonction `fit()`) puis de faire la prédiction (avec la fonction `predict`).

Quelque soit l'algo choisi vous allez retrouver les fonctions `fit` et `predict`. Ce qui changera ce seront les paramètres à passer au constructeur de la classe de l'algo. Votre travail portera sur le choix de ces paramètres.

Exemple un peu bateau avec une méthode de type SVM.

```
from sklearn.svm import LinearSVC
clf = LinearSVC(dual=True)
clf.fit(X_train, y_train)
```

```
clf.predict(X_test)
```

Évaluation

On fait l'évaluation en confrontant les prédictions sur les `X_test` et les `y_test`. La fonction `score` nous donne l'exactitude (*accuracy*) moyenne du modèle.

```
clf.score(X_test, y_test)
```

Pour la classification il existe une classe bien pratique
: `sklearn.metrics.classification_report`

```
from sklearn.metrics import classification_report

y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

Exercice (à rendre)

Refaites une partition train/test différente et comparez les résultats.

Validation croisée

Pour améliorer la robustesse de l'évaluation on peut utiliser la validation croisée (*cross-validation*).
En pratique, on divise notre dataset en 5 (`part0` , `part1` , `part2` , `part3` , `part4`) puis on entraîne notre modèle sur les parties `[part0->part3]` et on évalue sur les 20% restants, c'est-à-dire `part4` , puis on fait "glisser la fenêtre de sélection", en entraînant le modèle sur les parties `[part1->part4]` et on teste le résultat sur `part0` , et ainsi de suite. Dans cette configuration, on obtient donc 5 mesures. Quel est l'intérêt de cette méthode selon vous ?

`scikit-learn` a des classes pour ça.

```
from sklearn.model_selection import cross_validate, cross_val_score
print(cross_validate(LinearSVC(), X_wine, y_wine)) # infos d'accuracy mais aussi de temps
print(cross_val_score(LinearSVC(), X_wine, y_wine)) # uniquement accuracy
```

Optimisation des hyperparamètres

L'optimisation des hyperparamètres est la dernière étape. Ici encore `scikit-learn` nous permet de le faire de manière simple et efficace. Nous utiliserons `sklearn.model_selection.GridSearchCV` qui fait une recherche exhaustive sur tous les paramètres donnés au constructeur. Cette classe utilise aussi la validation croisée.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

param_grid =  {'C': [0.1, 0.5, 1, 10, 100, 1000], 'kernel':['linear']}
grid = GridSearchCV(LinearSVC(), param_grid, cv = 5, scoring = 'accuracy')
estimator = grid.fit(X_wine, y_wine)
print(estimator.cv_results_)
```

```
df = pd.DataFrame(estimator.cv_results_)
df.sort_values('rank_test_score')
```

Classification de textes

Le dataset 20 newsgroups est un exemple de classification de textes proposé par `scikit-learn`. Il y a aussi de la doc sur les traits (*features*) des documents textuels.

La classification avec des techniques non neuronales repose en grande partie sur les traits utilisés pour représenter les textes.

```
from sklearn.datasets import fetch_20newsgroups

categories = [
    "sci.crypt",
    "sci.electronics",
    "sci.med",
    "sci.space",
]

data_train = fetch_20newsgroups(
    subset="train",
    categories=categories,
    shuffle=True,
)

data_test = fetch_20newsgroups(
    subset="test",
    categories=categories,
    shuffle=True,
)
```

```
print(len(data_train.data))
print(len(data_test.data))
```

Ici on a un jeu de 2373 textes catégorisés pour train. À nous d'en extraire les features désirées.

Le modèle des sacs de mots est le plus basique.

Attention aux valeurs par défaut des paramètres. Ici par exemple on passe tout en minuscule et la tokenisation est rudimentaire. Ça fonctionnera mal pour d'autres langues que l'anglais. Cependant, presque tout est modifiable et vous pouvez passer des fonctions de prétraitement personnalisées.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(stop_words="english")
X_train = vectorizer.fit_transform(data_train.data) # données de train vectorisées
y_train = data_train.target
X_train.shape
```

Voilà la tête que ça a:

```
print(X_train[0, :])
```

```
X_test = vectorizer.transform(data_test.data)
y_test = data_test.target
```

Pour l'entraînement et l'évaluation on reprend le code vu auparavant

```
clf = LinearSVC(C=0.5)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

TF·IDF est un raffinement de ce modèle, qui donne en général de meilleurs résultats. Il existe une librairie qui fait tous les calculs pour vous :

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(
    sublinear_tf=True,
    max_df=0.5,
    stop_words='english'
)
X_train = vectorizer.fit_transform(data_train.data) # données de train vectorisées
y_train = data_train.target
X_train.shape

X_test = vectorizer.transform(data_test.data)
y_test = data_test.target

clf = LinearSVC(C=0.5)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

Projet à présenter le 25/04

1. Un projet complet

L'archive `imdb_smol.tar.gz` contient 602 critiques de films sous formes de fichiers textes, réparties en deux classes : positives et négatives (matérialisées par des sous-dossiers). Votre mission est de réaliser un script qui :

- Charge et vectorise ces données
- Entraîne et compare des classifieurs sur ce jeu de données

L'objectif est de déterminer quel type de vectorisation et de modèle semble le plus adapté et quels hyperparamètres choisir. Vous pouvez par exemple tester des SVM comme ci-dessus, un modèle de régression logistique, un arbre de décision, un modèle bayésien naïf ou une forêt d'arbres de décision.

2. D'autres traits

Essayez avec d'autres *features* : La longueur moyenne des mots, le nombre ou le type d'adjectifs, la présence d'entités nommées, ...

Pour récupérer ce genre de *features*, vous pouvez regarder du côté de spaCy comme prétraitement de vos données.

TP conçu par L. Grobol