Tâche 3 - Classification automatique de descriptions d'incidents

Cette tâche vise à classifier de courts textes décrivant des incidents qui se sont produits sur des chantiers de construction. Pour chaque incident, on connaît une étiquette de 1 à 4. Ces étiquettes sont inconnues et vous devrez tenter de les identifier à la section 3 de ce *notebook*.

Les objectifs de cette tâche sont:

- de se familiariser avec la classification de texte
- d'apprendre à utiliser les fonctions de base de scikit-learn
- de comprendre comment représenter un texte sous la forme d'un sac de mots (*bag of words*)
- de faire l'évaluation d'un modèle de classification avec un corpus de test
- de tenter d'interpréter les résultats d'un modèle à l'aide des poids d'attributs.

Pour la première partie, vous devrez construire une fonction (*train_and_test_classifier*) qui entraine un modèle (les options étant la régression logistique et le naïf bayésien) et en faire l'évaluation sur des données d'entrainement et des données de test. Deux fichiers de textes sont disponibles pour mener votre expérimentation (voir Section 1).

Pour la deuxième partie, vous devez tenter de déterminer à quoi correspond chacune des classes d'incident. Faites une analyse sur poids des modèles pour proposer des étiquettes pour chacune des classes. Vous pouvez vous inspirer des *notebooks* disponibles sur le site du cours. Expliquez clairement comment vous êtes arrivé à vos conclusions.

Merci de respecter les signatures des fonctions *train_and_test_classifier* et *load_incident_dataset*.

Section 1 - Lecture des fichiers de données

Voici les fichiers mis à votre disposition pour mener vos expérimentations. La fonction *load_incident_data* peut être utilisée pour lire les 2 fichiers (train et test). Rien à modifier dans cette section.

In [1]:
```
import json

train_json_fn = "./data/t3_train.json"
test_json_fn = "./data/t3_test.json"


def load_incident_dataset(filename):
    with open(filename, 'r') as fp:
        incident_list = json.load(fp)
    return incident_list
```

In [ ]:
```
train_list = load_incident_dataset(train_json_fn)
print("Nombre d'incidents:", len(train_list))
print("\nUn exemple:\n", train_list[10])
```

In [ ]:
```
test_list = load_incident_dataset(test_json_fn)
print("Nombre d'incidents", len(test_list))
incident = test_list[10]
print("\nUne description d'incident:", incident["text"])
print("\nSon étiquette:", incident["label"])
```

Chargement des librairies

In [4]:
```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
import pandas as pd
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import cross_val_score

import warnings
from sklearn.exceptions import ConvergenceWarning

# Suppress ConvergenceWarning
warnings.filterwarnings("ignore", category=ConvergenceWarning)
```

Section 2 - Entraînement et évaluation des modèles

Vous pouvez ajouter tout le code dont vous avez besoin pour l'entraînement. Merci de ne pas modifier la signature de la fonction *d'entraînement* et de bien expliquer votre démarche et vos résultats. N'oubliez pas de faire une recommandation de modèle. Vous pouvez ajouter des cellules au *anotebook* si nécessaire.

In [5]:
```
# Création du sac de mots
def get_bows(train_text_set, test_text_set):
    """
    Vectorise un ensemble de phrases au moyen de la technique Bag of Words.

    Parameters
    ----------
    train_text_set: list of dict: liste contenant un dictionnaire par
                                  texte d'entrainement de la forme suivante:
                                  {"text": str, "label": int}
    test_text_set: list of dict: liste contenant un dictionnaire par
                                 texte de test de la forme suivante:
                                 {"text": str, "label": int}

    Returns
    -------
    train_bow: np array: "sac de mots" des données d'entrainement, tableau numpy
    avec un vecteur par phrase de forme (i, nombre de mots dans le vocabulaire)
    test_bow: np array: "sac de mots" des données de test, tableau numpy avec
    un vecteur par phrase de forme (i, nombre de mots dans le vocabulaire)

    """
    # -- Récupère les textes des 2 jeux de données dans des listes
    train_text_corpus = [text["text"].strip() for text in train_text_set]
    test_text_corpus = [text["text"].strip() for text in test_text_set]

    # -- Initialise une instance du CountVectorizer de sklearn qui permet de vectoriser
    # -- un ensemble de phrases selon la méthode Bag of Words
    vectorizer = CountVectorizer(lowercase=True, max_df=0.85, max_features=270)

    # -- Entraine le vectorizer et transforme le corpus d'entrainement en vecteurs
    train_bow = vectorizer.fit_transform(train_text_corpus)
    test_bow = vectorizer.transform(test_text_corpus)

    # -- Transforme les matrices scipy en matrices numpy
    train_bow = train_bow.toarray()
    test_bow = test_bow.toarray()

    # -- Construit un dataframe avec les mots présents dans le vectorizer
    df = pd.DataFrame(vectorizer.get_feature_names_out(), columns =['Mots'])

    return train_bow, test_bow, df

def get_labels(text_set):
    """
    Petite fonction utilitaire pour récupérer les labels dans un vecteur
    numpy compatible pour le Naive Bayes Classifier de sklearn.

    Parameters
    ----------
    text_set: list of dict: liste contenant un dictionnaire par
                            texte de la forme suivante:
                            {"text": str, "label": int}


    Returns
    -------
    labels: numpy array: vecteur numpy du format (nombre d'éléments dans text_set, 1)

    """
    # -- Récupérer tous les labels dans une liste
    labels_list = [text["label"] for text in text_set]
    # -- Convertir la liste en tableau numpy et transposer le vecteur obtenu
    # -- pour correspondre au format du classificateur de sklearn
    labels = np.transpose(np.array(labels_list))

    return labels

def train_and_test_classifier(train_fn, test_fn, model='NB'):
    """
    :param train_fn et test_fn: les 2 fichiers utilisées pour entraîner et tester les classificateurs
    :param model: le type de classificateur. NB = Naive Bayes, LR = Régression logistique.
    :return: un dictionnaire contenant 3 valeurs:
                - l'accuracy à l'entrainement (validation croisée)
                - l'accuracy sur le jeu de test
                - la matrice de confusion calculée par scikit-learn sur les données de test
    """
    # Récupération des sacs de mots des 2 jeux de données
    train_bow, test_bow, df = get_bows(train_fn, test_fn)
    # Récupération des labels d'entraînements dans un vecteur cible "y"
    y_train = get_labels(train_fn)
    y_test = get_labels(test_fn)

    # Initialisation et entrainement du classificateur
    if model == "NB": # Si le modèle souhaité est le Naive Bayes Classifier
        clf = MultinomialNB(alpha=0.01)
        clf.fit(train_bow, y_train)
        for i in range(len(clf.classes_)):
            ## Pour chaque classe ajoute au dataframe de mot la log probabilité
            ## que le mot se trouve dans la classe
            df[clf.classes_[i]] = list(clf.feature_log_prob_[i])

    if model == "LR":
        clf = LogisticRegression(random_state=0)
        clf.fit(train_bow, y_train)
        for i in range(len(clf.classes_)):
            ## Pour chaque classe ajoute au dataframe de mot la log probabilité
            ## que le mot se trouve dans la classe
            df[clf.classes_[i]] = list(clf.coef_[i])

    ## -- Trie le dataframe de la log prob la plus élevée à la plus basse
    ## -- pour que les mots les plus probables apparaissent en premier lieu
    for i in range(0,4):
        sorted_df = df[["Mots", clf.classes_[i]]].sort_values(
                    by=(clf.classes_[i]), ascending=False, ignore_index=True
                )
        ## Liste des terms qui ne donne pas d'indication sur la nature de la classe
        unpertinent_words = ['how', 'what', 'when', 'where', 'which', 'who',
                             'whom', 'whose', 'why', 'the', 'is', 'are',
                             'for', 'the', 'each', 'some', 'did', 'to', 'that',
                             'had', 'as', 'it', 'while', 'into', 'of', 'he', 'in',
                             'at', 'his', 'were', 'from', 'an', 'by', 'with',
                             'not']
        ## Affiche les 20 mots qui ont le plus de chance de se trouver dans
        ## la classe i et donc les plus représentatifs
        print("MOTS + PROBABILITÉS DE LA CLASSE %d (= FONCTION×DES COEFFICIENTS OU MODÈLE"%clf.classes_[i])
        print("-----------------------------------------")
        display(
                sorted_df[~sorted_df["Mots"].isin(unpertinent_words)][:20]
            )

    # -- Evaluation sur les données d'entrainement
    train_scores = cross_val_score(clf, train_bow, y_train, cv=10) # Validation croisée
    mean_train_accuracy = train_scores.mean()                      # Moyenne des exactitudes obtenues

    # -- Evaluation sur les données de test
    y_pred = clf.predict(test_bow)
    test_accuracy = accuracy_score(y_test, y_pred)
    conf_matrix = confusion_matrix(y_test, y_pred)

    # Les résultats à retourner
    results = dict()
    results['accuracy_train'] = mean_train_accuracy
    results['accuracy_test'] = test_accuracy
    results['confusion_matrix'] = conf_matrix

    return results
```

Classification avec le Classificateur Naïf Bayésien

In [ ]:
```
results_nb = train_and_test_classifier(train_list, test_list, model='NB')
results_nb
```

Commentaire des Résultats

Avec le vectorizer et le classificateur de base, les résultats n'étaient pas excellents (autour des 65%). Nous avons donc joué avec les différents paramètres pour améliorer la classification:

- Tout d'abord, le paramètre "lowercase" fixé à True permet un prétraitement du texte en mettant tous les termes en minuscules. Ça permet d'éviter les doublons. Cette manipulation n'a pas augmenté les résultats de manière significative, mais c'est une bonne pratique de prétraitement selon nous (sauf peut-être pour les tâches de reconnaissance d'entités nommées pour lesquelles la majuscule est un bon indicateur).
- Ensuite, nous avons diminué "max_df" initialement fixé à 1. Ce paramètre définit le seuil au dessus duquel les termes ne doivent pas être gardés dans le vocabulaire. Autrement dit, les termes présent dans plus de 85% des documents du corpus en l'occurrence ne seront pas gardés. Grâce à ça, le score a augmenté de manière drastique (jusqu'à 85% environ), ce qui est logique car les termes présents dans beaucoup de documents sont sûrement très communs et peu informatifs sur la nature du document. En éliminant, on réduit le nombre de variables à prendre en compte dans le classificateurs et celui-ci s'en sort mieux.
- Enfin, la dernière manipulation pour la vectorisation qui nous a fait atteindre les 95% environ est le passage du paramètre *max_feature*, initié à 270, au vectorizer. Ce paramètre fait en sorte que le vectorizer ne garde que les 270 meilleurs token en terme de "term frequency", c'est-à-dire les mots qui apparaissent le plus par document et donc qui sont les plus informatifs.
- Nous avons également joué sur le paramètre *alpha* dans le classificateur en le fixant à 0.01. Ce dernier assignera aux termes qui n'apparaissent pas dans une classe donnée une probabilité de 0.01 d'apparition dans cette classe pour éviter d'avoir des probabilités de classe de 0. Cette dernière manipulation a permis d'augmenter encore légèrement les résultats en entrainement et en test avec respectivement **96,18%** et **97,04%** d'exactitude

Classification avec la Régression Logistique

In [ ]:
```
results_lr = train_and_test_classifier(train_list, test_list, model='LR')
results_lr
```

Commentaire des Résultats

Ici, seuls les paramètres du vectorizer ont été modifiés. A elles seules, ces modifications ont permis au modèle d'atteindre une exactitude de 99,24% en entrainement et de 99,43% en test. Les résultats de la régression logistique sont meilleurs que ceux du classificateur naïf bayésien car elle permet de modéliser des relations plus complexes entre les variables et est donc plus adaptée pour ce genre de tâche. Cependant, de manière générale les résultats sont très élevés et une dernière raison peut être avancer pour justifier cela. Cela est mentionnée dans la section suivante.

Section 3 - À quoi correspondent les classes? Explicabilité du modèle

En utilisant les poids des modèles, tentez d'attribuer une signification aux différentes classes. Il devrait être possible de définir précisément la nature et la composition des classes. L'important est d'utiliser ce qu'on observe dans les modèles pour fournir une explication plausible.

Vous pouvez ajouter tout le code et toutes les cellules dont vous avez besoin.

Tout le code utilisé pour montrer les coefficients se trouve dans la fonction 2 pour faciliter l'implémentation

Comme le montre les dataframes des coefficients en Section 2, les mots pertinents qui diffèrent d'une classe à l'autre sont uniquement **les mots**. On peut donc en déduire que chaque classe répertorie les incidents survenus sur le chantier lors d'un **trimestre en particulier.**

- La classe 1 répertorie les incidents de janvier-février-mars
- La 2 ceux d'avril-mai-juin
- La 3 ceux de juillet-août-septembre
- La 4 ceux d'octobre-novembre-décembre.

Ça nous donne également des explications quant aux résultats extrêmement élevés des deux classificateurs. En effet, puisque la nature des textes est très similaire, mais que seuls les mois changent, le nom du mois est un excellent indicateur du label à assigner. Combiné au fait que la taille du jeu de données d'entrainement est élevée par rapport à la facilité de la tâche, les modèles peuvent facilement distinguer un pattern.

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

Section 4 - Section réservée pour nos tests

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```