# TP 1 :

Le TAL (traitement automatique du langage) est généralement composé de deux à trois grandes étapes:

- Pré-traitement : une étape qui cherche à standardiser du texte afin de rendre son usage plus facile
- Représentation du texte comme un vecteur : Cette étape peut être effectuée via des techniques de sac de mots (Bag of Words) ou Term Frequency-Inverse Document Frequency (Tf-IdF). On peut également apprendre des représentations vectorielles (embedding) par apprentissage profond.
- Classification, trouver la phrase la plus similaire.

Dans ce TP, nous allons couvrir les tâches de TAL les plus communes pour lesquelles des outils spécifiques au français existent.

Nous utiliserons principalement SpaCy. SpaCy est une jeune librairie (2015) qui offre des modèles pré-entraînés pour diverses applications, y compris la reconnaissance d'entités nommées. SpaCy est la principale alternative à NLTK (Natural Language Tool Kit), la librairie historique pour le TAL avec Python, et propose de nombreuses innovations et options de visualisation qui sont très intéressantes.

Après avoir installé la librairie SpaCy (pip install spacy), il faut télécharger les modèles français.

```
python -m spacy download fr_core_news_sm
```

Ce modèle est un réseau convolutionnel entraîne sur deux corpus, WikiNER et Sequoia, ce qui représente de gros volumes de données en français (typiquement plusieurs dizaines de Go). Dans un notebook Jupyter, on peut alors importer SpaCy et charger le modèle français.

```
import spacy
nlp = spacy.load("fr_core_news_sm")
```

On va également créer une phrase d'exemple pour toutes les tâches de TAL que nous allons illustrer.

```
test = "Bouygues a eu une coupure de réseau à Marseille"
```

## 1. Tokenisation

La tokenisation cherche à transformer un texte en une série de tokens individuels. Dans l'idée, chaque token représente un mot, et identifier des mots semble être une tâche

relativement simple. Mais comment gérer des exemples tels que: « J'ai froid ». Il faut que le modèle de tokenisation sépare le « J' » comme étant un premier mot.

SpaCy offre une fonctionnalité de tokenisation en utilisant la fonction nlp. Cette fonction est le point d'entrée vers toutes les fonctionnalités de SpaCy. Il sert à représenter le texte sous une forme interprétable par la librairie.

```
def return_token(sentence):
    # Tokeniser la phrase
    doc = nlp(sentence)
    # Retourner le texte de chaque token
    return [X.text for X in doc]
```

Ainsi, en appliquant cette tokenisation à notre phrase, on obtient:

```
return_token(test)
['Bouygues', 'a', 'eu', 'une', 'coupure', 'de', 'réseau', 'à',
'Marseille']
```

## 2. Enlever les mots les plus fréquents

Certains mots se retrouvent très fréquemment dans la langue française. En anglais, on les appelle les « stop words ». Ces mots, bien souvent, n'apportent pas d'information dans les tâches suivantes. Lorsque l'on effectue par exemple une classification par la méthode Tf-IdF, on souhaite limiter la quantité de mots dans les données d'entraînement.

Les « stop words » sont établis comme des listes de mots. Ces listes sont généralement disponibles dans une librairie appelée NLTK (Natural Language Tool Kit), et dans beaucoup de langues différentes. On accède aux listes de cette manière:

```
from nltk.corpus import stopwords
stopWords = set(stopwords.words('french'))
{'ai',
 'aie',
 'aient',
 'aies',
 'ait',
```

```
 'as',
 'au',
 'aura',
 'aurai',
 'auraient',
 'aurais',
 ...
```

Pour filtrer le contenu de la phrase, on enlève tous les mots présents dans cette liste:

```
clean_words = []
for token in return_token(test):
    if token not in stopWords:
        clean_words.append(token)

clean_words
['Bouygues', 'a', 'coupure', 'réseau', 'Marseille', '.']
```

## 3. Tokenisation par phrases

On peut également appliquer une tokenisation par phrase afin d'identifier les différentes phrases d'un texte. Cette étape peut à nouveau sembler facile, puisque a priori, il suffit de couper chaque phrase lorsqu'un point est rencontré (ou un point d'exclamation ou d'interrogation).

Mais que se passerait-t-il dans ce cas-là?

`Bouygues a eu une coupure de réseau à Marseille. La panne a affecté 300.000 utilisateurs.`

Il faut donc une compréhension du contexte afin d'effectuer une bonne tokenisation par phrase.

```
def return_token_sent(sentence):
    # Tokeniser la phrase
```

```
    doc = nlp(sentence)
    # Retourner le texte de chaque phrase
    return [X.text for X in doc.sents]
```

On applique alors la tokenisation à la phrase mentionnée précédemment.

```
return_token_sent("Bouygues a eu une coupure de réseau à
Marseille. La panne a affecté 300.000 utilisateurs.")
```

La tokenisation des phrases retourne alors :

```
['Bouygues a eu une coupure de réseau à Marseille.',
 'La panne a affecté 300.000 utilisateurs.']
```

## 4. Stemming

Le stemming consiste à réduire un mot dans sa forme « racine ». Le but du stemming est de regrouper de nombreuses variantes d'un mot comme un seul et même mot. Par exemple, une fois que l'on applique un stemming sur « Chiens » ou « Chien », le mot résultant est le même. Cela permet notamment de réduire la taille du vocabulaire dans les approches de type sac de mots ou Tf-IdF.

Un des stemmers les plus connus est le Snowball Stemmer. Ce stemmer est disponible en français.

```
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer(language='french')

def return_stem(sentence):
    doc = nlp(sentence)
    return [stemmer.stem(X.text) for X in doc]
```

Si on applique ce stemmer à notre phrase d'exemple :

```
return_stem(test)
```

```
['bouygu', 'a', 'eu', 'une', 'coupur', 'de', 'réseau', 'à',
'marseil', '.']
```

## 5. Reconnaissance d'entités nommées (NER)

En traitement automatique du langage, la reconnaissance d'entités nommées cherche à détecter les entités telles que des personnes, des entreprises ou des lieux dans un texte. Cela s'effectue très facilement avec SpaCy.

```
def return_NER(sentence):
    # Tokeniser la phrase
    doc = nlp(sentence)
    # Retourner le texte et le label pour chaque entité
    return [(X.text, X.label_) for X in doc.ents]
```

Puis on peut appliquer la fonction à une phrase d'exemple.

```
return_NER(test)
```

Les entités identifiées sont les suivantes :

```
[('Bouygues', 'ORG'), ('Marseille', 'LOC')]
```

Bouygues est reconnue comme une organisation, et Marseille comme un lieu.

Spacy offre des « Visualizers », des outils graphiques qui permettent d'afficher les résultats de reconnaissances d'entités nommées ou d'étiquetage par exemple.

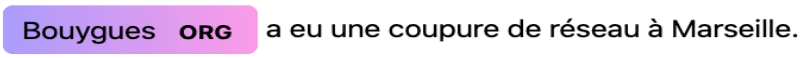
```
from spacy import displacy

doc = nlp(test)
displacy.render(doc, style="ent", jupyter=True)
```


On peut également préciser des options à DisplaCy pour n'afficher que les organisations, d'une certaine couleur:

```
doc = nlp(test)
colors = {"ORG": "linear-gradient(90deg, #aa9cfc, #fc9ce7)"}
options = {"ents": ["ORG"], "colors": colors}

displacy.render(doc, style="ent", jupyter=True,
options=options)
```




## 6. L'étiquetage morpho-syntaxique

L'étiquetage morpho-syntaxique ou Part-of-Speech (POS) Tagging en anglais essaye d'attribuer une étiquette à chaque mot d'une phrase mentionnant la fonctionnalité grammaticale d'un mot (Nom propre, adjectif, déterminant…).

```
def return_POS(sentence):
    # Tokeniser la phrase
    doc = nlp(sentence)
    # Retourner les étiquettes de chaque token
    return [(X, X.pos_) for X in doc]
```

Les étiquettes identifiées sont les suivants :

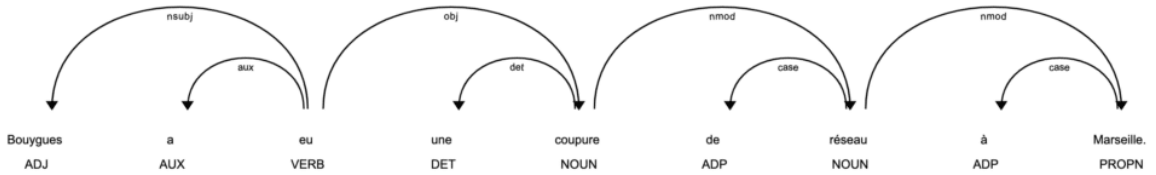
```
return_POS(test)
[(Bouygues, 'ADJ'),
 (a, 'AUX'),
```

```
 (eu, 'VERB'),
 (une, 'DET'),
 (coupure, 'NOUN'),
 (de, 'ADP'),
 (réseau, 'NOUN'),
 (à, 'ADP'),
 (Marseille, 'PROPN')]
```

On remarque que Bouygues n'est pas identifié comme un nom propre, mais comme un Adjectif. Pour le reste, les tags sont corrects.

Spacy dispose également d'une option de visualisation qui nous permet simplement d'afficher les étiquettes identifiées ainsi que les dépendances entre ces étiquettes.

```
doc = nlp(test)
displacy.serve(doc, style="dep")
```




## 7. Embedding par mot

Avec SpaCy, on peut facilement récupérer le vecteur correspondant à chaque mot une fois passé dans le modèle pré-entraîné en français.

Cela nous sert à représenter chaque mot comme étant un vecteur de taille 96.

```
import numpy as np

def return_word_embedding(sentence):
    # Tokeniser la phrase
    doc = nlp(sentence)
    # Retourner le vecteur lié à chaque token
```

```
    return [(X.vector) for X in doc]
```

Ainsi, lorsqu'appliqué à la phrase test, on obtient :

```
return_word_embedding(test)
[array([ -1.8685186 ,   1.3645297 ,  -2.3505871 ,  -1.233012
,
         -3.702136  ,   1.3316352 ,  -1.3532144 ,  -3.879726
,
         -7.051861  ,  -2.8570302 ,  -2.409908  ,   3.3500502
,
          3.8512042 ,  -0.5462021 ,  -1.7187259 ,  -5.341373
,
    …
```

Cette information nous sert notamment lorsque l'on cherche à caractériser la similarité entre deux mots ou deux phrases.

## 8. Similarité entre phrases

Afin de déterminer la similarité entre deux phrases, nous allons opter pour une méthode très simple :

- déterminer l'embedding moyen d'une phrase en moyennant l'embedding de tous les mots de la phrase
- calculer la distance entre deux phrases par simple distance euclidienne

Cela s'effectue très facilement avec SpaCy !

```
def return_mean_embedding(sentence):
    # Tokeniser la phrase
    doc = nlp(sentence)
    # Retourner la moyenne des vecteurs pour chaque phrase
    return np.mean([(X.vector) for X in doc], axis=0)
```

On peut alors tester notre fonction avec plusieurs phrases :

```
test_2 = "Le réseau sera bientot rétabli à Marseille"
```

```
test_3 = "La panne réseau affecte plusieurs utilisateurs de l'opérateur"
test_4 = "Il fait 18 degrés ici"
```

Ici, on s'attend à ce que la phrase 2 soit la plus proche de la phrase test, puis la phrase 3 et 4.

Avec Numpy, la distance euclidienne se calcule simplement :

```
np.linalg.norm(return_tensor(test)-return_tensor(test_2))
np.linalg.norm(return_tensor(test)-return_tensor(test_3))
np.linalg.norm(return_tensor(test)-return_tensor(test_4))
```

On obtient alors les résultats suivants :

```
16.104986
17.035103
22.039303
```

La phrase 2 est bien identifiée comme la plus proche, puis la phrase 3 et 4. On peut également utiliser cette approche pour classifier si une phrase appartient à une classe ou à une autre.

## 9. Transformers

Si vous avez suivi l'actualité du traitement automatique du langage (TAL) ces derniers mois, vous avez surement entendu parlé des Transformers, des modèles état-de-l'art qui apprennent des représentations vectorielles à partir d'un texte d'entrée et qui dépassent les capacités humaines sur certains points. Ces modèles peuvent être pré-entraînés et sont mis à disposition par Hugging Face, une startup spécialisée dans le traitement de langage naturel.

Les modèles les plus récents mis à disposition par Hugging Face spécifiques au français sont entraînés avec XLM, une architecture qui a depuis été battue par de nombreux autres modèles, dont BERT.

BERT est également disponible dans une version multi-linguage, entraîné sur le Wikipedia de plus de 104 langues, sous le nom de : `bert-base-multilingual-cased`.

Les transformers sont particulièrement bons pour:

- la génération de texte
- apprendre une représentation vectorielle
- prédire si une phrase est la suite d'une autre

- les tâches de questions/réponses

Nous allons ici donner un exemple de comment utiliser les transformers pour prédire si une phrase est la suite d'une autre. On trouve des applications de ceci lorsque l'on cherche par exemple à segmenter des fils de discussions en plusieurs blocs distincts.

On suppose ici que PyTorch et Transformers sont installés ( `pip install transformers` ).

```
import torch
from transformers import *
```

Ensuite, on doit charger un tokenizer et un modèle.

```
# Tokeniser, Model
tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')
model = BertForNextSentencePrediction.from_pretrained('bert-base-multilingual-cased')
model.eval()
```

La ligne `model.eval()` permet de passer le modèle en mode évaluation.

Par la suite, on passe au modèle une séquence de texte convertie en tokens, et des délimitations des phrases:

```
text = "Comment ça va ? Bien merci, un peu stressé avant l'examen"
# Texte tokenisé
tokenized_text = tokenizer.tokenize(text)
# Convertir le texte en indexs
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
segments_ids = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

On transforme le tout en tenseurs interprétables par le modèle :

```
# Transformer en tenseurs
```

```
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
```

Finalement, on prédit si la deuxième phrase est la suite de la première :

```
predictions = model(tokens_tensor, segments_tensors)
```

Si l'élément en position 0 est plus grand que celui en position 1, la phrase est la suite de la première. Autrement, c'est une nouvelle discussion qui commence.

```
if np.argmax(predictions) == 0:
    print("Suite")
else:
    print("Pas la suite")
```

## Conclusion

Nous avons couvert dans ce TP de nombreuses applications de Traitement Automatique du Langage Naturel. Bien que de très nombreuses ressources existent exclusivement en Anglais, il existe tout de même des outils intéressants pour le Français, que votre problème de TAL concerne une classification de texte par Tf-IdF, ou une approche par Deep Learning ou Transformers.