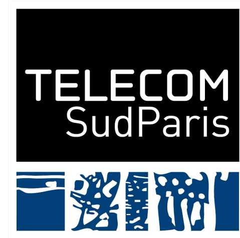


Introduction au traitement automatique du langage

Julien Romero

Qu'est que le TAL ?

- Le *traitement automatique du langage* (TAL) ou *Natural Language Processing* (NLP) est un domaine multidisciplinaire entre la linguistique, l'information, l'intelligence artificielle et les sciences sociales travaillant sur des **données en langues naturelles** (= humainement compréhensibles, souvent textuelles ou sonores)
- Domaines d'applications :
  - Traduction automatique
  - Génération de texte
  - Correction orthographique
  - Extraction d'information
  - Chatbot
  - Moteur de recherche

Le TAL à l'ère de ChatGPT

- Beaucoup de techniques "classiques" sont devenues obsolètes avec les modèles de langage
- Cependant :
  - Les problèmes traités restent les mêmes
  - Utiliser un modèle de langage est souvent plus coûteux en temps et ressources
  - Les modèles de langage sont souvent des boîtes noires
    - Difficulté à expliquer les résultats
  - Avoir recours à un modèle de langage revient souvent à tirer avec un bazooka sur une mouche
    - Bon sens informatique souvent plus rapide et efficace
  - Les modèles de langages sont difficile à contrôler dans certaines situations
    - Hallucination, résultats faux

Les langages formels

Langages formels

- Les langages formels sont des outils mathématiques et informatiques permettant de décrire un ensemble de “phrases” appartenant au langage
- Exemple de langages :
  - Une langue naturelle (difficile à représenter à cause de l’aspect sémantique)
  - Un langage de programmation
  - Les adresses mails, numéros de téléphone

Les expressions régulières (ou regex)

- Permettent de décrire des langages dits réguliers à travers une description de motifs (c.f. CSC3102 pour des motifs simples)
- Définition récursive :
  - Soit Σ un ensemble symboles appelé *alphabet*
  - Pour tout symbole *a* dans Σ, a est une expression régulière. On ajoute souvent ε pour représenter le caractère vide.
  - Pour toute expression régulière *R* et *S*, nous pouvons créer une nouvelle regex en prenant :
    - La concaténation *RS* qui “colle” deux regex (ex. : ab concatène a et b et représente l'ensemble {ab})
    - L'union R|S qui donne le choix entre deux regex (ex. : a|b représente l'ensemble {a, b})
    - L'étoile de Kleene R* qui permet la répétition de R de 0 à n fois (pour tout n) (ex. : a* représente l'ensemble {ε, a, aa, aaa, aaaa, ...}
  - On peut utiliser des parenthèses pour regrouper des termes et donner des priorités

Opérations avancées

- On peut définir des opérations avancées simplifiant l'écriture (mais n'ajoutant pas à l'expressivité)
  - R? dit que R est présent 0 ou 1 fois (= R|ε)
  - R+ dit que R est répété au minimum une fois (= RR*)
  - R{n} répète n fois R (= RRRRR...)
  - R{n, m} répète R entre n et m fois
  - R{n,} répète R au moins n fois
  - ^ = début de ligne, $ = fin de ligne
  - [liste] : union de tous les éléments dans la liste ([abc] = a|b|c)
    - Possibilité de faire des intervalles avec - ([0-9] = 0|1|2|3|4|5|6|7|8|9]
  - [^liste] : union de tous les éléments *pas* dans la liste ([^abc] = d|e|f|g|...)
  - On peut échapper des caractères spéciaux avec \ (ex.: \|)

Raccourcis utiles

- . représente n’importe quel caractère (= a|b|c|d ... pour tout symbole dans Σ)
- \d représente les chiffres (= [0-9])
- \s représente les “espaces” (= [\t\n\r\f\v])
- \w représente les caractères alphanumériques (= [a-zA-Z0-9_])

Exemples

À essayer sur https://regex101.com/

- Numéros de téléphones français : ((\+33)|0)\d{9}
- Adresses mails : [\w\-\.]+@([\w-]+\.)+[\w-]{2,4}
- Fichiers images : \w+\.(gif|jpeg|jpg|eps|svg|png)
- Extraction d'un type : \w+ (is|are|was|were) ((a|an))?\w+
- Extraction d'un message d'erreur : ^\[Error\].*$

Regex en Python

```
import re

if __name__ == '__main__':
    regex = re.compile(r'[a-z]+')  # Les mots
    # match commence au début de la string
    print(regex.match("bonjour à toi"))  # <re.Match object; span=(0, 7), match='bonjour'>
    print(regex.match("0123456789"))  # None
    print(regex.match("0 bonjour"))  # None
    # search commence n'importe où
    print(regex.search("0 bonjour"))  # <re.Match object; span=(2, 9), match='bonjour'>

    # On peut accéder à des groupes et leur donner des noms
    regex_images = re.compile(r"(?P<filename>\w+)\.(gif|jpeg|jpg|eps|svg|png) ")
    print(regex_images.match("toto.gif").group("filename"))  # toto
```

Grammaire non contextuelles

- Tout n'est pas exprimable avec expressions régulières :
  - Ex.: a^{n}b^{n} (a et b répétés le même nombre de fois) = {ε, ab, aabb, aaabbb}
  - Ex.: les expressions bien parenthésées
  - Ex.: les langages de programmation (c.f. Cours sur les compilateurs) !
  - Il existe un outil plus expressif : les grammaires hors contexte (= context-free grammar = CFG)
- Une CFG est définie par :
  - Un ensemble de symboles non terminaux V
  - Un ensemble de terminaux Σ, l'alphabet, disjoint de V
  - Un symbole de départ S dans V
  - Un ensemble de règles de productions X -> Y où X est dans V et Y est la concaténation de symboles dans V et Σ (on autorise parfois l'union pour simplifier les écritures)

Exemples

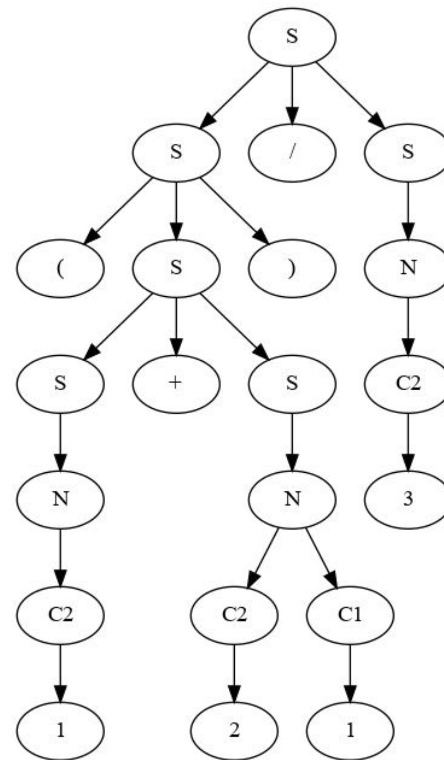
- a^{n}b^{n}
  - V = {S}, Σ = {a, b}
  - S -> aSb
  - S -> ε
  - (ou S -> ε|aSb pour simplifier)
- Comment obtient-on aaabbb ?
  - S -> aSa -> aaSbb -> aaaSbbb -> aaabbb

Exemples

- Les expressions arithmétiques
  - C1 -> 0|1|2|3|4|5|6|7|8|9
  - C1 -> C1 | C1
  - C2 -> 1|2|3|4|5|6|7|8|9
  - N -> C2
  - N -> C2 C1
  - S -> N | S + S | S - S | S * S | S / S | (S)
- Comment obtient-on (1 + 21) / 3 ?
  - S -> S / S -> S / N -> S / C2 -> S / 3 -> (S) / 3 -> (S + S) / 3 -> (N + S) / 3 -> (N + N) / 3 -> (C2 + N) / 3 -> (1 + N) / 3 -> (1 + C2 C1) / 3 -> (1 + 2 C1) / 3 -> (1 + 21) / 3

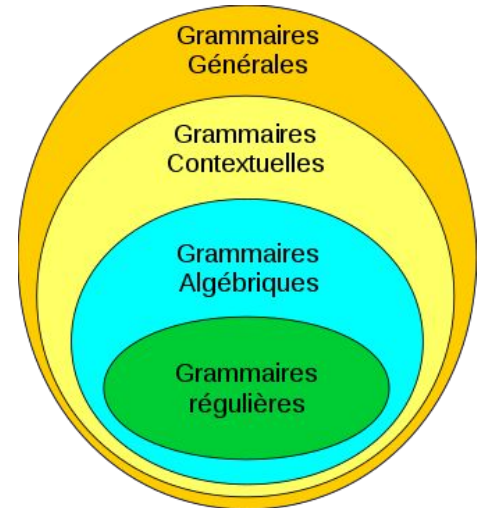
En Python

```
from pyformlang.cfg import CFG
from pyformlang.cfg.recursive decent parser  import RecursiveDecentParser
# NLTK a aussi un module pour les CFG

if   name    == '  main  ':
    cfg = CFG.from text("""
    C1 -> 0|1|2|3|4|5|6|7|8|9
    C2 -> 1|2|3|4|5|6|7|8|9
    N -> C2
    N -> C2 C1
    S -> N | S + S | S - S | S * S | S / S | ( S )
    """)
    print("(1+21)/3" in cfg)  # True
    print("(1+21/3" in cfg)  # False

    # L'arbre syntaxique donne une représentation visuelle
    parser = RecursiveDecentParser(cfg)
    parse tree = parser.get parse tree("(1+21)/3")
    parse_tree.write_as_dot("parse_tree.dot")
```



Est-ce que les CFG sont suffisantes ?

- On ne peut pas tout exprimer avec des CFG !
  - Ex: {a^{n}b^{n^2}}, {www for w in {a, b}*}
- Beaucoup de travaux sur la théorie des langages pour qualifier ce qui exprimable ou non
- La hiérarchie de Chomsky définit quatre classes
  - Regex : grammaires régulières
  - CFG : grammaire algébriques
  - Machine de Turing : grammaires contextuelles
  - Algorithmes : grammaires générales



En pratique

- Très utile et efficace de savoir écrire des regex pour retrouver une information dans un texte en suivant un motif
- CFG moins utilisées (sauf en compilation)

Langages Naturels

Les mots

- Les mots sont les éléments primitifs de tout texte
- La première tâche d'un programme de NLP consiste souvent à **tokeniser** un texte, c'est-à-dire à séparer le texte en l'ensemble des éléments qui le constitue
    - En général, en mot
    - Mais aussi en groupe de lettres (c.f. Grands modèles de language)
- La séparation en mots n'est pas si triviale

He obtained his Ph.D., and then said: I am happy!

- Il faut souvent faire plus que séparer avec les espaces et enlever la ponctuation.
    - Parfois, on veut même grouper des mots ou ponctuations ensemble (ex. : New York, émoticones, hashtags)
- On appelle le **vocabulaire** l'ensemble des mots d'un texte
- Parfois, on a besoin de séparer les phrases avec un tokenizer spécifique.

Tokenisation en Python

```
import spacy

if __name__ == '__main__':
    # python -m spacy download en_core_web_sm
    nlp = spacy.load("en_core_web_sm")
    doc = nlp("He obtained his Ph.D., and then said: I am happy! ")
    print([token.text for token in doc])
    # ['He', 'obtained', 'his', 'Ph.D.', ',', 'and', 'then', 'said', ':', 'I', 'am', 'happy',
'!']
```

Normalisation des mots

- Souvent, un mot peut avoir plusieurs formes : avec majuscule, au singulier, pluriel, conjuguer, ...
- Avoir un vocabulaire trop grand rend le traitement compliqué et réduit les performances
  - Il faut donc normaliser les textes, quitte à perdre en qualité
- Exemples de normalisations :
  - Tout en minuscule. Ex.: He -> he
  - **Stemmatisation**: Réduit un mot à sa racine en suivant des règles qui enlèvent des suffixes.
    - Ex.: obtained -> obtain, laws -> law
  - **Lemmatisation**: Attribution à chaque groupe de mots un représentant unique.
    - Am, are, is, was, were, been, be -> be
    - Plus coûteux que la stemmatisation
  - Enlever les **stop words** et la ponctuation: mots très utilisés dans une langue (and, the, a, in, to, ...)

Normalisation en Python

```
import spacy
from nltk.stem import *

if __name__ == '__main__':
    # python -m spacy download en_core_web_sm
    nlp = spacy.load("en_core_web_sm")
    doc = nlp("He obtained his Ph.D., and then said: I am happy! ")
    print([token.text for token in doc])
    # ['He', 'obtained', 'his', 'Ph.D.', ',', 'and', 'then', 'said', ':', 'I', 'am', 'happy',
'!']

    print([token.lemma_ for token in doc])
    # ['he', 'obtain', 'his', 'ph.d.', ',', 'and', 'then', 'say', ':', 'I', 'be', 'happy', '!']
    stemmer = PorterStemmer()
    print([stemmer.stem(token.text) for token in doc])
    # ['he', 'obtain', 'hi', 'ph.d.', ',', 'and', 'then', 'said', ':', 'i', 'am', 'happi', '!']
    print([token.lemma_ for token in doc if not token.is_stop and not token.is_punct])
    # ['obtain', 'ph.d.', 'say', 'happy']
```

Annotations grammaticales - Part-of-Speech

- Il est souvent utile de connaître la nature grammaticale d'un mot dans une phrase (nom, verbe, adverbe, adjectif, ...) . C'est ce qu'on appelle le **Part-of-Speech** (PoS)
- Certains mots sont ambigus. En anglais, *book* peut vouloir dire *livre* ou *réserver*.
- L'attribution d'un label à chaque mot d'une séquence s'appelle du **sequence labeling**
- Traditionnellement, l'attribution des PoS est faite avec des statistiques sur la langue, des modèles de Markov, et de la programmation dynamique (Viterbi)
  - Toujours très efficace
- Maintenant, on voit des modèles à base de modèles de langages.

Annotations grammaticales - Dependency parsing

- Le PoS donne la nature d'un mot. Il arrive que l'on ait besoin de plus d'information grammatical en sachant comment chaque mot est connecté dans la phrase.
- Ex.: Quel est le sujet du verbe *am* dans *I, as always, am happy!*
- En connectant les mots entre eux, on obtient une structure d'arbre appelée l'arbre de dépendances.

Annotations grammaticales en Python

```
print([token.pos_ for token in doc])  # PoS simple
# ['PRON', 'VERB', 'PRON', 'NOUN', 'PUNCT', 'CCONJ', 'ADV', 'VERB', 'PUNCT', 'PRON', 'AUX',
'ADJ', 'PUNCT']
print([token.tag_ for token in doc])  # PoS détaillé
# ['PRP', 'VBD', 'PRP$', 'NN', ',', 'CC', 'RB', 'VBD', ':', 'PRP', 'VBP', 'JJ', '.']
print([token.dep_ for token in doc])
# ['nsubj', 'ROOT', 'poss', 'dobj', 'punct', 'cc', 'advmod', 'conj', 'punct', 'nsubj', 'ccomp',
'acomp', 'punct']

# Visualisation Graphique
image = displacy.render(doc, style="dep")
output_path = Path("dependency_plot.svg")
output_path.open("w", encoding="utf-8").write(image)
```

Annotations grammaticales en Python

```
print([token.pos_ for token in doc])  # PoS simple
# ['PRON', 'VERB', 'PRON', 'NOUN', 'PUNCT', 'CCONJ', 'ADV', 'VERB', 'PUNCT', 'PRON', 'AUX',
'ADJ',
print
# ['PI
print
# ['ns                                                                              comp',
'acomp

# Visu
image
outpu _                  _
output_path.open("w", encoding="utf-8").write(image)
```

conj
cc
dobj
ccomp
nsubj
poss
advmod
nsubj
acomp

He PRON
obtained VERB
his PRON
Ph.D., NOUN
and CCONJ
then ADV
said: VERB
I PRON
am AUX
happy! ADJ

Annotations grammaticales - Named Entity Recognition

- La reconnaissance d'entités nommées (Named Entity Recognition = NER) consiste à trouver les noms propres dans un texte et à les classifier (personne, organisation, city, ...)

NER en Python

```
doc = nlp("Elon Musk secretly acquired Twitter stock before buying company, Morgan Stanley hid
detail, says shareholder lawsuit ")

for ent in doc.ents:
    print(ent.text, ent.label_)
# Elon Musk PERSON
# Twitter PRODUCT
# Morgan Stanley ORG
```

Réprésentations

Vectorisation

- En général, on veut donner le texte prétraité à un algorithme de machine learning pour prédire quelque chose
  - Problème : Les mots de sont pas des chiffres !
- Il faut transformer notre texte en représentation vectorielles de nombres réels
- **Bag-of-words (BoW):** cet représentation consiste à attribuer à chaque mot du vocabulaire un index (en partant de 0). On obtient ensuite l'encodage en attribuant à la dimension d'un mot son nombre d'occurrence dans le texte
  - Ex.: I eat the food and drink the water. What do I do now ?
    - Vocabulaire = [I, eat, the, food, and, drink, water, what, do, now]
    - Bag-of-words = [2, 1, 2, 1, 1, 1, 1, 1, 2, 1]

Vectorisation

- En général, on veut donner le texte prétraité à un algorithme de machine learning pour prédire quelque chose
  - Problème : Les mots de sont pas des chiffres !
- Il faut transformer notre texte en représentation vectorielles de nombres réels
- **TF-IDF**: Variante de BoW adaptant la pondération pour donner plus d'importance aux mots rares (c.f. TP)
- **Embeddings appris**: Base de modèles de langage (voir l'année prochaine)

Qu'est-ce qu'un modèle de langage?

Étant donné une séquence composée de plusieurs mots $w_1 w_2 ... w_n$, nous voulons savoir à quel point cette phrase est probable. Autrement dit, nous voulons:

$$P(w_1, w_2, \ldots, w_n)$$

Exemples:

- "Le renard mange la poule." a une plus forte probabilité que "La renard le poule"
- "Il vient de Paris. Son train est en retard." a une plus forte probabilité que "Il vient de Paris. Le renard mange la poule.". (Le contexte est important).

Décomposition d'un modèle de langage

- Décomposition causale:

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_2, w_1) \ldots P(w_n \mid w_{n-1} \ldots w_1)$$

(On cherche la probabilité du mot suivant étant donné les mots précédents)

- Décomposition avec un masque

$$P(w_1, w_2, \ldots, w_n) = P(w_k \mid w_1, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n) \cdot P(w_1, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n)$$

Comment calculer les probabilités?

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_2, w_1) \ldots P(w_n \mid w_{n-1} \ldots w_1)$$

On va vouloir estimer chacune des probabilités $P(w_k \mid w_{<k})$

Cette décomposition est très utile pour générer du texte. Dans certaines applications comme la traduction, on s'intéresse à d'autres probabilités.

L’approche statistique

$$P(w_k \mid w_{<k}) = \frac{Nbr\ observations\ w_1 w_2 \ldots w_k}{Nbr\ observations\ w_1 w_2 \ldots w_{k-1}}$$

Problème: Plus k est élevé, moins on a d’observations (voire pas du tout).

On doit donc faire des approximations N-grams:

- Bigram: $P(w_k \mid w_{<k}) \approx P(w_k \mid w_{k-1})$
- Trigram: $P(w_k \mid w_{<k}) \approx P(w_k \mid w_{k-1}, w_{k-2})$

Augmenter trop le nombre de mots considérés donne des probabilités imprécises et difficiles à stocker (Google s’est arrêté à environ 1 milliard de 5-grams).

La suite l’année prochaine…