POStagging

January 7, 2021

1 Projet TAL M1 S1 : POS Tagging

2 Implémentation du classifieur

2.1 Import du corpus

On charge les trois corpus *in-domain* sous la forme de listes de dictionnaires : chaque phrase a une clé "mots" qui est associée à une liste des mots, et une clé "POS" associée à une liste de POS.

On utilise les trois corpus distincts de French-GSD : - Train : apprentissage. - Dev : validation. Pour tester et améliorer le modèle. - Test : évaluation. On ne l'utilisera pas pendant l'apprentissage ou les tests.

[1]:
```
def load_corpus(file):
    with open(file, "r", encoding = "utf8") as f:
        content = f.read() # chargement du corpus
    content = content.split("\n\n") # séparation en phrases
    corpus = []
    for phrase in content: # pour chaque phrase
        phrase_dico = {"mots" : [], "gold_labels" : []} # liste qui contiendra␣
↪1 dictionnaire par mot de la phrase
        for line in phrase.splitlines():
            if not line.startswith("#"): # on ignore les lignes qui commencent␣
↪par #
                features = line.split("\t")
                phrase_dico["mots"].append(features[1])
                # phrase_dico["lemme"].append(features[2])
                phrase_dico["gold_labels"].append(features[3])
        corpus.append(phrase_dico)
    return corpus

gsd_train = load_corpus("corpus-in-domain/fr_gsd-ud-train.conllu")
gsd_test = load_corpus("corpus-in-domain/fr_gsd-ud-test.conllu")
gsd_dev = load_corpus("corpus-in-domain/fr_gsd-ud-dev.conllu")
```

[2]:
```
print("---- Aperçus d'une phrase de chaque corpus-----", end="\n\n")
print(gsd_train[1], end="\n\n")
print(gsd_test[102], end="\n\n")
print(gsd_dev[564])
```

```
---- Aperçus d'une phrase de chaque corpus-----

{'mots': ["L'", 'œuvre', 'est', 'située', 'dans', 'la', 'galerie', 'des', 'de',
'les', 'batailles', ',', 'dans', 'le', 'château', 'de', 'Versailles', '.'],
'gold_labels': ['DET', 'NOUN', 'AUX', 'VERB', 'ADP', 'DET', 'NOUN', '_', 'ADP',
'DET', 'NOUN', 'PUNCT', 'ADP', 'DET', 'NOUN', 'ADP', 'PROPN', 'PUNCT']}

{'mots': ['La', 'gestion', 'et', "l'", 'exploitation', 'de', 'la', 'salle',
'de', 'concert', 'Wagram', ',', 'récemment', 'rénovée', ',', 'sera', 'assurée',
'par', 'Eurosites', ',', 'leader', 'en', 'France', 'de', 'la', 'location', 'de',
'salles', '.'], 'gold_labels': ['DET', 'NOUN', 'CCONJ', 'DET', 'NOUN', 'ADP',
'DET', 'NOUN', 'ADP', 'NOUN', 'PROPN', 'PUNCT', 'ADV', 'VERB', 'PUNCT', 'AUX',
'VERB', 'ADP', 'PROPN', 'PUNCT', 'NOUN', 'ADP', 'PROPN', 'ADP', 'DET', 'NOUN',
'ADP', 'NOUN', 'PUNCT']}

{'mots': ['Cette', 'espèce', 'est', 'endémique', 'du', 'de', 'le',
'département', 'de', 'Nariño', 'en', 'Colombie', '.'], 'gold_labels': ['DET',
'NOUN', 'AUX', 'ADJ', '_', 'ADP', 'DET', 'NOUN', 'ADP', 'PROPN', 'ADP', 'PROPN',
'PUNCT']}
```

2.2 Extraction des caractéristiques

La fonction feature_extraction renvoie une liste de dictionnaires (un par mot) qui contiennent les caractéristiques suivantes pour chaque mot : - mot - mot précédent : pour le premier mot de la phrase son mot précédent sera “START”, ce qui permettra de prendre en compte la caractéristique “être le premier mot”. - mot suivant : pour le dernier mot, ce sera “END”. - commence par une lettre majuscule - est entièrement en majuscules - contient des chiffres - contient des caractères non alphanumériques - longueur du mot (3 caractéristiques binaires) : a 1 seule caractère, a moins de 3 caractères, a plus de 3 caractères. - a un suffixe nominal - a un suffixe adjectival - a un suffixe verbal - a un suffixe adverbial

On ne conserve plus la structure des phrases, qui n’est plus nécessaire une fois qu’on a extrait les informations comme mot précédent et mot suivant.

Ces caractéristiques sont encodées sous la forme d’un dictionnaire pour chaque mot dont les clés désignent les caractéristiques (“mot - commotions”, “prec - les”, “long”, “suff_nom”, etc.) et les valeurs valent 1 quand la caractéristique est vraie pour ce mot. Si la caractéristique n’est pas vraie, alors on n’ajoute pas cette entrée dans le dictionnaire, ce qui permettrait d’optimiser l’apprentissage et la prédiction ensuite.

On a ajouté des booléens pour les différentes catégorie de caractéristiques (mots, majuscules, longueur, caractères non alphabétique et suffixes) qui sont pas défaut True et qu’on utilisera plus tard pour tester l’utilité de ces caractéristiques.

[3]:
```
def feature_extraction(corpus, feat_mots=True, feat_maj=True,␣
↪feat_non_alpha=True, feat_long=True, feat_suff=True):

    corpus_features = []
    """
```

```
    Listes à utiliser pour les lemmes :
    list_adj = ["ain", "aine","aire","é", "ée","iel", "uel", "lle",  "al",␣
↪"ales", "al", "ial","er", "ère", "ier", "esque", "eur", "euse",␣
↪"ieux","ueux", "if","ive", "in","ine","ique","atoire", "u","ue", "issime",␣
↪"able","ible", "uble", "ième","uple"]
    list_noun = ["ade", "age","aille", "aison", "ison","oison", "ation",␣
↪"ition","ssion", "sion","xion", "isation","ment", "ement","erie",␣
↪"ure","ature","at", "ance","ence", "escence","ité", "eté","té", "ie","erie",␣
↪"esse","ise", "eur","isme", "iste", "seur", "isseur", "isateur",␣
↪"euse","isseuse", "atrice","ier","ière", "aire","ien",␣
↪ienne","iste",er","eron","eronne","trice","oir",␣
↪"oire","ier","ière","erie","anderie", "aire","ain","aille", "ée",␣
↪"ard","asse", "assier","âtre","aut","eau","ceau", "ereau","eteau", "elle",␣
↪"et","elet","ette",
"elette","in","otin", "ine" "illon","on", "ille", "erole","ole","iche"]
    """

    list_vb = ["iser","ifier", "oyer","ailler", "asser","eler", "eter","iller",␣
↪"iner","nicher", "ocher","onner","otter","oter", "ouiller"]
    list_adj = ["ain", "aine","ains", "aines","aire", "aires","é", "ée","ées",␣
↪"és","iel", "iels","uel", "uels",
"lle", "lles","els", "el" "al", "ales", "al", "ial", "aux","iaux", "er","ers",␣
↪"ère","ères", "ier", "iers",      "esque","esques", "eur","eurs",␣
↪"euse","euses", "ieux","ueux", "if", "ifs","ive", "ives","in", "ins","ine", ␣
↪    "ines","iques", "ique","atoire", "u","ue", "us","ues",␣
↪"issime","issimes","able","ible", "ibles","ables",
    "uble","ubles", "ième","ièmes", "uple"]
    list_noun = ["ade", "ades", "age", "ages","aille", "ailles", "aison",␣
↪"ison", "isons","oison", "ation", "itions", "ition", "ssion", "sion","xion",␣
↪"isation","ment", "ement","erie", "eries","ure","ures","ature",␣
↪"atures","at", "ance","ence", "escence","ité", "eté","té", "ie","erie",␣
↪"esse", "ise", "eur","isme", "iste", "istes","eurs", "seur","seurs",␣
↪"isseur","isseurs", "isateur","euse", "euses","isseuse", "isseuses",      ␣
↪"atrice", "atrices","ier", "iers","ière", "ières","aire","aires","ien",␣
↪"iens","ienne", "iennes","iste",         "istes","er", "ers","eron",␣
↪"erons","eronne","trice","oir", "oire","oires", "oirs","ier", "iers","ière",␣
↪        "ières","erie","eries","anderie","aire", "aires","ain", "aines",␣
↪"ée","ées","aille", "ard","asse", "asses", "assier","âtre","aut","eau",␣
↪"eaux","ceau", "ereau","eteau", "elle","elles",␣
↪"et","elet","ets","ette","elette","ettes", "elettes","in", "ins","otin",␣
↪"ine","ines", "illon","on","ons","ille", "erole","eroles", "ole","oles",␣
↪"iche"]

    for phrase in corpus: # ajout des features additionnelles
        for prev, word, suiv in zip(["START"] + phrase["mots"][:-1],␣
↪phrase["mots"], phrase["mots"][1:] + ["END"]):
```

```
            # création de triplets (mot précédent, mot, mot suivant)
            # avec "START" en prev pour le 1er mot
            # et "END" en suiv pour le dernier

            # dictionnaire de features du mot
            if feat_mots :
                features_mot = {
                    # on récupère le gold_label correspondant
                    f"mot - {word.lower()}" : 1,
                    f"prec - {prev.lower()}" : 1,
                    f"mot_suiv - {suiv.lower()}" : 1,
                    }
            else:
                features_mot = {}

            if feat_maj:
                if word.istitle(): features_mot["maj"] = 1
                if word.isupper(): features_mot["all_caps"] = 1

            if feat_non_alpha:
                if any(char.isdigit() for char in word): features_mot["num"] =␣
↪1# mieux que isnumeric(), car renvoie false si espace (40 000) ou virgule␣
↪(50,6) par ex
                if not word.isalnum(): features_mot["nonAlphanum"] = 1

            if feat_long:
                if len(word) <= 3: features_mot["court"] = 1
                if len(word) > 3: features_mot["long"] = 1
                if len(word) == 1: features_mot["un_car"] = 1

            if feat_suff:
                if word.endswith("ment"): features_mot["suff_adv"] = 1
                if any(word.endswith(elem) and len(word) != len(elem) for elem␣
↪in list_noun): features_mot["suff_noun"] =1
                if any(word.endswith(elem) and len(word) != len(elem) for elem␣
↪in list_adj): features_mot["suff_adj"] = 1
                if any(word.endswith(elem) for elem in list_vb):␣
↪features_mot["suff_vb"] = 1
                # on vérifie la longueur du mot pour être sûr que ce soit un␣
↪suffixe car on peut avoir le mot                    age avec le suffixe␣
↪age par exemple ou bien aux
                # suff_noun : any(lemma.endswith(elem) and len(word) !=␣
↪len(elem) for elem in list_noun),
                # suff_adj : any(lemma.endswith(elem) for elem in list_adj),
                # suff_vb : any(lemma.endswith(elem) for elem in list_vb)
```

```
            # ajout au corpus
            corpus_features.append(features_mot)

    return corpus_features # renvoie les features transformés en vecteurs␣
↪one-hot
```

[4]:
```
gsd_train_features = feature_extraction(gsd_train)

print(len(gsd_train_features))
print(*gsd_train_features[10:15], sep="\n")
```

```
364349
{"mot - qu'": 1, 'prec - sport': 1, 'mot_suiv - on': 1, 'nonAlphanum': 1,
'court': 1}
{'mot - on': 1, "prec - qu'": 1, 'mot_suiv - les': 1, 'court': 1}
{'mot - les': 1, 'prec - on': 1, 'mot_suiv - considére': 1, 'court': 1}
{'mot - considére': 1, 'prec - les': 1, 'mot_suiv - presque': 1, 'long': 1}
{'mot - presque': 1, 'prec - considére': 1, 'mot_suiv - comme': 1, 'long': 1,
'suff_adj': 1}
```

[44]:
```
def add_gold(features, corpus, addMot=False):
    '''Ajoute les gold labels pour créer un corpus d'entraînement / de test'''
    i = 0
    gold_corpus = []
    for phrase in corpus:
        for word, pos_gold in zip(phrase["mots"],phrase["gold_labels"]):
            if not pos_gold == "_": # on ignore les mots sans gold_labels
                if addMot:
                    gold_corpus.append((features[i], pos_gold, word))
                else:
                    gold_corpus.append((features[i], pos_gold))
                i += 1

    return gold_corpus
```

[45]:
```
gsd_train_features_gold = add_gold(gsd_train_features, gsd_train)
print(*gsd_train_features_gold[100:110], sep="\n")
```

```
({'mot - avec': 1, 'prec - reviendrais': 1, 'mot_suiv - plaisir': 1, 'long': 1},
'PUNCT')
({'mot - plaisir': 1, 'prec - avec': 1, 'mot_suiv - !': 1, 'long': 1}, 'DET')
({'mot - !': 1, 'prec - plaisir': 1, 'mot_suiv - end': 1, 'nonAlphanum': 1,
'court': 1, 'un_car': 1}, 'NOUN')
({'mot - les': 1, 'prec - start': 1, 'mot_suiv - forfaits': 1, 'maj': 1,
'court': 1}, 'VERB')
({'mot - forfaits': 1, 'prec - les': 1, 'mot_suiv - comprennent': 1, 'long': 1},
'DET')
```

```
({'mot - comprennent': 1, 'prec - forfaits': 1, 'mot_suiv - le': 1, 'long': 1},
'NOUN')
({'mot - le': 1, 'prec - comprennent': 1, 'mot_suiv - transport': 1, 'court':
1}, 'ADP')
({'mot - transport': 1, 'prec - le': 1, 'mot_suiv - en': 1, 'long': 1}, 'NOUN')
({'mot - en': 1, 'prec - transport': 1, 'mot_suiv - car': 1, 'court': 1}, 'ADJ')
({'mot - car': 1, 'prec - en': 1, 'mot_suiv - grand': 1, 'court': 1}, 'NOUN')
```

2.3 Implémentation de l'algorithme de classification

On a choisi d'implémenter la classification avec un perceptron moyenné.

La fonction predict sera utilisée à la fois dans l'apprentissage et dans la "prédiction". Elle correspond à la recherche de l'étiquette avec le plus grand score (argmax…)

[7]:
```python
def predict(word_features, weights):
    """Renvoie l'étiquette avec le plus gros score (argmax)"""
    scores = {}
    for tag, w in weights.items():
        scores[tag] =  sum(word_features.get(feat)*w.get(feat, 0) for feat in␣
↪word_features)
    #print(scores)

    return max(scores, key=scores.get)
```

[47]:
```python
from random import shuffle

def perceptron_train(training_set, MAX_EPOCH=3):

    tags = ["ADJ", "ADP", "ADV", "AUX", "CCONJ", "DET", "INTJ", "NOUN", "NUM",␣
↪"PART", "PRON", "PROPN", "PUNCT", "SCONJ", "SYM", "VERB", "X"]

    # initialisation de a (poids totaux)
    a = {}
    for tag in tags:
        a[tag] = {}

    # initialisation des vecteurs de poids
    w = {} # TODO defaultdictionary
    for tag in tags:
        w[tag] = {}

    n_update = 0 # nombre de mots sur lequel l'entraînement a été effectué

    last_update = 0 # dictionnaire de dictionnaire suivant la même structure␣
↪que les vecteurs de poids a et w
    # et qui stocke la valeur de n lors de la dernière modification d'un poid
```

```
    for i in range(0, MAX_EPOCH):
        shuffled_set = training_set.copy() # copie du training set
        shuffle(shuffled_set) # mélange du training set

        for word in shuffled_set    :
            n_update += 1 # on compte le nb de mots déjà vus

            vec, gold = word
            prediction = predict(vec, w) # trouve étiquette plus probable avec les poids w

            if not prediction == gold: # si le gold_label n'est pas égal à celui prédit
                # on ignore les mots dont le gold_label est "_" ("au" et "du") car ils sont ensuite analysés comme "à le" et "de le"

                # mise à jour de a : on ajoute chaque poids * le nombre d'updates sans modifications
                for tag in tags:
                    for feat in w[tag]:
                        if not w[tag].get(feat,0) == 0:
                            a[tag][feat] = a[tag].get(feat,0) + w[tag].get(feat,0)*(n_update-last_update)

                # modification de w
                for feat in vec: # pour chaque feature du mot
                    # on modifie les poids de w pour les 2 étiquettes concernées
                    w[gold][feat] = w[gold].get(feat,0) + 1 #  on ajoute x(i) à chaque poids de l'étiquette correcte
                    w[prediction][feat] = w[prediction].get(feat,0) - 1 #  on retire x(i) à chaque poids de l'étiquette mal prédite

                # modification de last_update
                last_update = n_update

    # mise à jour finale de a
    for tag in w:
        for feat in w[tag]:
            a[tag][feat] = a[tag].get(feat, 0) + w[tag][feat]*(n_update - last_update)

    print(list(a["DET"].items())[:20], end="\n\n")
    print(list(w["DET"].items())[:20], end="\n\n")

    return a
```

Pour éviter de mettre entièrement à jour a après chaque mot, puisque la grande majorité des poids reste inchangé, on stocke après chaque modification d'un poids l'index de sa dernière modification, ce qui permet de mettre à jour dans a seulement les vecteurs qui sont modifiés, en ajoutant dans a le poids correspondant dans w, multiplié par le nombre d'update pendant lesquels il n'a pas été modifié.

A la fin de l'apprentissage, on met également à jour tous les poids qui sont "en attente" de mise à jour, c'est-à-dire ceux dont la dernière mise à jour est inférieur à n_update.

[9]:
```
# poids_gsd_train = perceptron_train(gsd_train_features_gold)
```

3 Evaluation des performances sur le corpus de validation (Dev)

La fonction test() renvoie le taux d'erreur sur un corpus en utilisant la fonction predict() avec les poids issus de l'apprentissage. On effectue ce test sur le corpus dev avec différents nombres d'itération à l'apprentissage pour voir l'impact des itérations sur ce taux d'erreur.

[43]:
```
gsd_dev_features = feature_extraction(gsd_dev)
gsd_dev_features_gold = add_gold(gsd_dev_features, gsd_dev)


def test(corpus, poids):
    """Prédit les étiquettes et renvoie un taux d'erreur"""
    nb_erreurs = 0
    # output = open("test.txt", "w")
    # output.write("MOT\tPREDICT\tGOLD\tRESULTAT\n")
    for word in corpus:
        vec = word[0]
        gold = word[1]
        prediction = predict(vec, poids)
       # print(f"{vec}\t{prediction}\t{gold}\t")
        if not gold == "_" and not prediction == gold:
            nb_erreurs +=1
            #output.write("ERREUR")
        #else:
            #output.write("OK")
        #output.write("\n")

    #output.write(f"Nombre d\'erreurs : {nb_erreurs}")
    #output.write(f"\nTaux d\'erreur : {nb_erreurs/len(corpus)}")
    #output.close()

    return nb_erreurs/len(corpus)
```

[48]:
```
from time import time

taux = []
```

```
x_range = range(1, 3)

for i in x_range:
    t0 = time()
    poids_gsd_train = perceptron_train(gsd_train_features_gold, MAX_EPOCH=i)
    tx_erreur = test(gsd_dev_features_gold, poids_gsd_train)
    taux.append(tx_erreur)
    print(f"{i} epochs : {tx_erreur:.2%} d'erreurs - temps training + test :␣
 ↪{time()-t0:.2f}s")
```

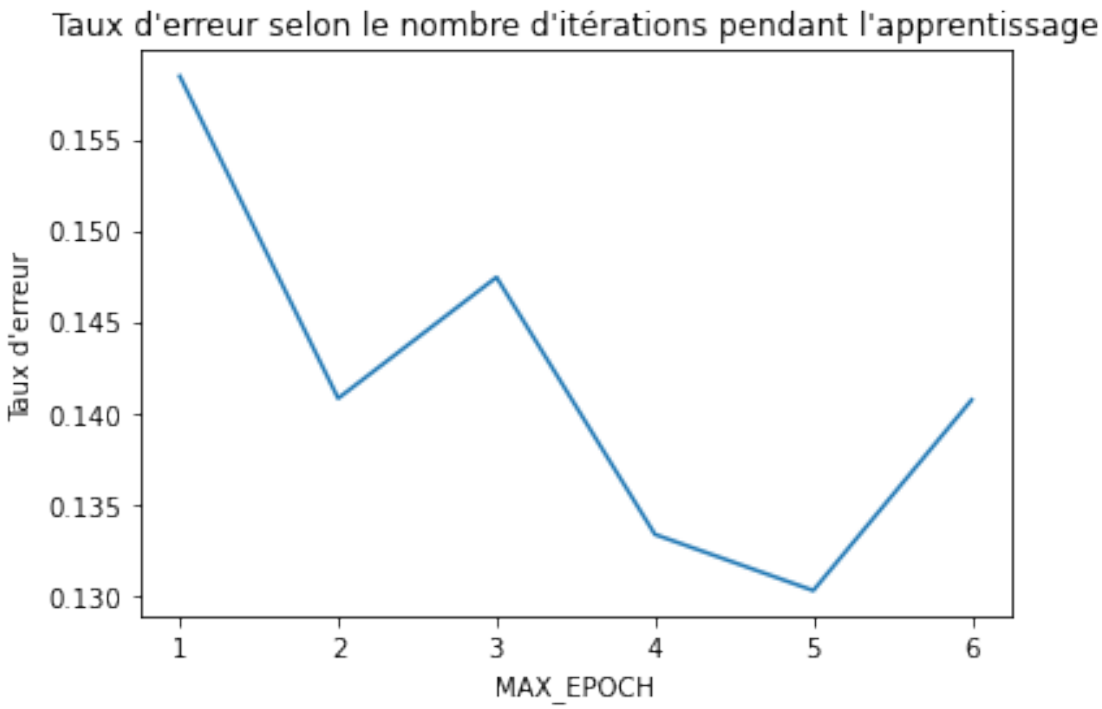
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-48-cfffd5ee0c99> in <module>
      6 for i in x_range:
      7     t0 = time()
----> 8     poids_gsd_train = perceptron_train(gsd_train_features_gold,␣
 ↪MAX_EPOCH=i)
      9     tx_erreur = test(gsd_dev_features_gold, poids_gsd_train)
     10     taux.append(tx_erreur)

<ipython-input-47-dc711689a998> in perceptron_train(training_set, MAX_EPOCH)
     38                     for feat in w[tag]:
     39                         if not w[tag].get(feat,0) == 0:
---> 40                             a[tag][feat] = a[tag].get(feat,0) + w[tag].
 ↪get(feat,0)*(n_update-last_update)
     41
     42                 # modification de w

KeyboardInterrupt:
```

[12]:
```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(x_range,taux)
plt.ylabel("Taux d'erreur")
plt.xlabel("MAX_EPOCH")
plt.title("Taux d'erreur selon le nombre d'itérations pendant l'apprentissage")
plt.show()
```



On remarque qu'à partir de 2 ou 3 itérations on descend en dessous de 5% d'erreurs, mais que des itérations additionnelles n'améliorent pas beaucoup ce taux, en tout cas pas assez par rapport au temps additionnel que demande l'entraînement avec ces itérations additionnelles. Nous choisissont donc 3 itérations par défaut.

3.0.1 Evaluation des caractéristiques

On peut se demander si on pourrait obtenir des taux similaires avec moins de features, si certaines features apportent plus de précision que d'autre. Pour cela on réeffectue un apprentissage avec 3 itérations, sans modifier l'extraction de caractéristiques. Puis on teste différentes combinaisons d'extractions de features sur le corpus dev avant d'effectuer le test : seulement les mots (mot, mot précédent et mot suivant), et tout sauf les mots. On pourrait bien sûr tester bien d'autres combinaisons. Les features non extraites sur le corpus dev valent donc toutes zéro pour ce corpus et ne seront pas prises en compte.

[13]:
```
poids_gsd_train = perceptron_train(gsd_train_features_gold) # entraînement avec␣
↪nombre d'itérations par défaut
```

[14]:
```
def test_features(corpus, poids, test_mots=False, test_maj=False,␣
↪test_non_alpha=False, test_long=False, test_suff=False):
    '''Effectue les 3 étapes nécessaires pour effectuer le test'''
    features = feature_extraction(corpus, feat_mots=test_mots,␣
↪feat_maj=test_maj, feat_non_alpha=test_non_alpha, feat_long=test_long,␣
↪feat_suff=test_suff)
```

```
    features_gold = add_gold(features, corpus)
    tx_erreur = test(features_gold, poids)

    return tx_erreur

print(f"Test avec seulement les mots : {test_features(gsd_train,␣
 ↪poids_gsd_train, test_mots=True):.3%}")
print(f"Test avec tout sauf les mots : {test_features(gsd_train,␣
 ↪poids_gsd_train, test_maj=True, test_suff=True, test_non_alpha=True,␣
 ↪test_long=True):.3%}")
```

```
Test avec seulement les mots : 19.251%
Test avec tout sauf les mots : 77.670%
```

On remarque que sans la prise en compte des mots le résultat est catastrophique ! Avec seulement les mots on fait environ 2 fois plus d'erreurs que si on prend en compte l'ensemble des caractéristiques. Tester quelques combinaisons de mots + autres caractéristiques :

[15]:
```
print(f"Test avec mots + longueur : {test_features(gsd_train, poids_gsd_train,␣
 ↪test_mots=True, test_long=True):.3%}")
print(f"Test avec mots + maj : {test_features(gsd_train, poids_gsd_train,␣
 ↪test_mots=True, test_maj=True):.3%}")
print(f"Test avec mots + caractères non alphanumériques :␣
 ↪{test_features(gsd_train, poids_gsd_train, test_mots=True,␣
 ↪test_non_alpha=True):.3%}")
print(f"Test avec mots + suffixes : {test_features(gsd_train, poids_gsd_train,␣
 ↪test_mots=True, test_suff=True):.3%}")
```

```
Test avec mots + longueur : 20.896%
Test avec mots + maj : 17.825%
Test avec mots + caractères non alphanumériques : 15.188%
Test avec mots + suffixes : 19.204%
```

A partir de ces tests non exhaustifs, il semblerait que la prise en compte des majuscules apporte davantage de précision, tandis que la prise en compte des autres caractéristiques seules, en plus des mots, apporte beaucoup moins de précision et pourrait peut-être être éliminés.

3.0.2 Erreurs fréquentes

On peut analyser les erreurs les plus fréquentes à l'aide d'une matrice de confusion.

[16]:
```
# matrice de confusion avec Pandas. source : https://stackoverflow.com/
 ↪questions/2148543/how-to-write-a-confusion-matrix-in-python

import pandas as pd

def matrice_confusion(corpus_feat, poids):
    predictions = []
    gold = []
```

```
    for word in corpus_feat:
        predictions.append(predict(word[0], poids))
        gold.append(word[1])

    preds = pd.Series((item for item in predictions), name = "Prédictions")
    refs = pd.Series((item for item in gold), name = "Références")
    matrice_confusion = pd.crosstab(refs, preds, margins=True)
    print(matrice_confusion)

matrice_confusion(gsd_dev_features_gold, poids_gsd_train)
```

```
Prédictions   ADJ   ADP  ADV   AUX  CCONJ   DET  NOUN  NUM  PART  PRON  PROPN  \
Références
ADJ          1295     0    0     0      0     3   695    1     0     0     73
ADP            30  5001    2     2      0     4    40    0     0     1    216
ADV            34     1  810     2      1     0   212    0     0     4     97
AUX             9     0    0   969      0     0    35    0     0     0      2
CCONJ           1     0    1     1    818     0    17    0     0     2     21
DET            24    44    0     0      0  4408   419    0     0     3    611
INTJ            0     0    0     0      0     0     0    0     0     0      4
NOUN           53     0    2     7      0     1  6273    2     0     0    341
NUM             7     0    0     0      0    11   138  756     0     0     41
PART            0     0    1     0      0     0     0    0     6     1      0
PRON           27     6    1     0      0    69    82    0     0   987    291
PROPN          24     3    1     0      0     0    63    4     0     1   2534
PUNCT           1     0    0     0      0     0    69    0     0     1      0
SCONJ           2     1    0     0      0     0     3    0     0     9     10
SYM             0     0    0     0      0     0     7    1     0     0      8
VERB          129     2    1    24      0     0   358    0     0     0     40
X               9     1    1     0      0     1    14    2     0     2    102
_               5     0    0     0      0    12   728    0     0     1     23
All          1650  5059  820  1005    819  4509  9153  766     6  1012   4414

Prédictions  PUNCT  SCONJ  SYM  VERB   X    All
Références
ADJ              0      0    0   133   1   2201
ADP              0      1    0   447   7   5751
ADV              0     23    0   123   0   1307
AUX              0      0    0   109   0   1124
CCONJ            0      0    0    23   0    884
DET              0      0    0     5   0   5514
INTJ             0      0    0     1   0      5
NOUN             0      0    0    49   1   6729
NUM              0      0    0     6   0    959
PART             0      0    0     1   0      9
PRON             0     29    0    50   0   1542
```

PROPN	0	0	0	2	2	2634
PUNCT	3719	0	1	9	4	3804
SCONJ	0	201	0	30	0	256
SYM	0	0	45	0	0	61
VERB	0	0	0	2226	1	2781
X	0	0	0	3	22	157
_	0	0	0	288	0	1057
All	3719	254	46	3505	38	36775

Les erreurs les plus fréquentes sont : - VERB au lieu d'ADJ (234) - PROPN au lieu de NOUN (217) - NOUN au lieu d'ADJ (124) - NOUN au lieu de SYM - VERB au lieu de NOUN (102)

3.0.3 Précision sur mots hors vocabulaire d'apprentissage

Comparons la précision sur les mots présents dans le corpus d'apprentissage et ceux absents du corpus d'apprentissage.

[17]:
```python
def getVoc(corpus):
    '''Renvoie le vocabulaire (set)'''
    voc = set()
    for phrase in corpus:
        for mot in phrase["mots"]:
            voc.add(mot)
    return voc

def test_hors_voc(features_gold_mot, poids, voc):
    """Prédit les étiquettes des mots hors vocabulaire et renvoie un taux␣
 ↪d'erreur"""
    nb_erreurs_hors_voc = 0
    nb_erreurs_in_voc = 0

    mots_hors_voc=0
    mots_in_voc = 0

    for vec, gold, mot in features_gold_mot:
        if mot in voc:
            mots_in_voc +=1
            prediction = predict(vec, poids)
            if not gold == "_" and not prediction == gold:
                nb_erreurs_in_voc +=1
        else:
            mots_hors_voc +=1
            prediction = predict(vec, poids)
            if not gold == "_" and not prediction == gold:
                # print(mot, vec, prediction, gold, sep="\t", end="\n")
                nb_erreurs_hors_voc +=1
```

```
    return (nb_erreurs_hors_voc/mots_hors_voc, nb_erreurs_in_voc/mots_in_voc)
```

[18]:
```
# nouvelle extraction de features mais avec les mots, pour pouvoir ensuite voir␣
↪s'ils sont dans le vocabulaire ou pas
gsd_dev_features_gold_mot = add_gold(gsd_dev_features, gsd_dev, addMot=True)

resultats_test_voc = test_hors_voc(gsd_dev_features_gold_mot, poids_gsd_train,␣
↪getVoc(gsd_train))

print(f"Taux d'erreur hors voc : {resultats_test_voc[0]:.3%}")
print(f"Taux d'erreur dans voc : {resultats_test_voc[1]:.3%}")
```

```
Taux d'erreur hors voc : 20.599%
Taux d'erreur dans voc : 14.942%
```

Le taux d'erreur sur les mots non présents dans le vocabulaire d'apprentissage est très élevé tandis que celui sur les mots présents dans le vocabulaire d'apprentissage est très bas. Le perceptron se repose beaucoup sur l'association des mots à leur catégorie grammaticale, par rapport aux autres caractéristiques.

3.1 Evaluation sur le corpus Test

Après un apprentissage avec l'ensemble des caractéristiques et 3 itérations sur le corpus d'apprentissage, on obtient un taux d'erreur de moins de 5% lors de l'évaluation sur le corpus test.

[19]:
```
gsd_test_features = feature_extraction(gsd_test)
gsd_test_features_gold = add_gold(gsd_test_features, gsd_test)

print(f"Taux d'erreur sur corpus d'évaluation in-domaine :␣
↪{test(gsd_test_features_gold, poids_gsd_train):.3%}")
```

```
Taux d'erreur sur corpus d'évaluation in-domaine : 15.731%
```

4 Evaluation hors-domaine

4.1 Analyse de l'impact du changement de domaine

- identifier causes de la baisse de performance : analyse de sortie, matrice de confusion, erreurs les + fréquentes
- réfléchir à des caractéristiques plus adaptées
- sélection d'un nouvel ensemble d'apprentissage avec des exemples représentatif (généré par un modèle de langue type TP2)

4.1.1 Corpus oral

[20]:
```
oral_dev = load_corpus("corpus-hors-domaine/spoken/fr_spoken-ud-dev.conllu")
oral_dev_features = feature_extraction(oral_dev)
oral_dev_features_gold = add_gold(oral_dev_features, oral_dev)

print(f"Taux d'erreur sur corpus d'évaluation hors domaine (oral) :␣
↪{test(oral_dev_features_gold, poids_gsd_train):.3%}")
```

```
Taux d'erreur sur corpus d'évaluation hors domaine (oral) : 21.369%
```

4.1.2 Corpus littéraire

4.2 Développement de systèmes robutes au changement de domaine