Apprentissage pour le TAL

Projet final: classifieur PoS

HU Wei, KE Zong-You

14 janvier 2021

1 Introduction

Ce rapport porte sur le dévloppement d'un classifieur statistique capable de prédire les PoS d'une phrase. Notre réalisation du projet est basée sur des corpus français du projet Universal Dependencies (UD French), ainsi que sur l'algorithme perceptron moyenné. Le travail sera présenté en quatre parties: ressources, méthodes, résultats et discussions.

2 Ressources

Pour le développement du classifieur souhaité, nous avons pris trois corpus du UD French, à savoir le UD French GSD (ci-après dénommé "GSD") pour l'apprentissage et l'évaluation in-domain, ainsi que le UD Old French-SRCMF (ci-après dénommé "Old French") et le UD French-Spoken (ci-après dénommé "Spoken French") pour des évaluations hors-domaines. Plus spécifiquement, nous avons utilisé le corpus *fr_gsd-ud-train.conllu* comme l'ensemble d'apprentissage, et le corpus *fr_gsd-ud-test.conllu* comme l'ensemble d'évaluation in-domain. Pour les évaluations hors-domaines, les corpus *fro_srcmf-ud-test.conllu* et *fr_spoken-ud-test.conllu* sont utilisés pour évaluer notre modèle.

3 Méthode

Notre méthode contient les étapes suivantes : le prétraitement des corpus, le développment du modèle du classifieur à travers la méthode d'apprentissage avec l'algorithme perceptron moyenné, et les évaluations et l'optimisation du modèle. Par la suite, la méthode sera détaillée étape par étape avec des bouts de codes pertinents.

3.1 Prétraitement des corpus

Cette étape consiste à extraire des corpus les mots et les PoS correspondants, et générer des fichiers contenant deux colonnes, l'une affichant les mots et l'autre les PoS correspondants. réalisée par les fonctions *read_corpus* et *write* qui se trouvent dans le fichier *readfile.py*. Le fichier *fr_gsd-ud-test.conllu* est aussi manipulé par la fonction *make_repetition* pour générer deux mini-corpus d'évaluation qui ne contiennent que des mots dans le vocabulaire et que des mots hors-vocabulaire, respectivement.

Quelques lignes essentielles de la fonction *read_corpus*:

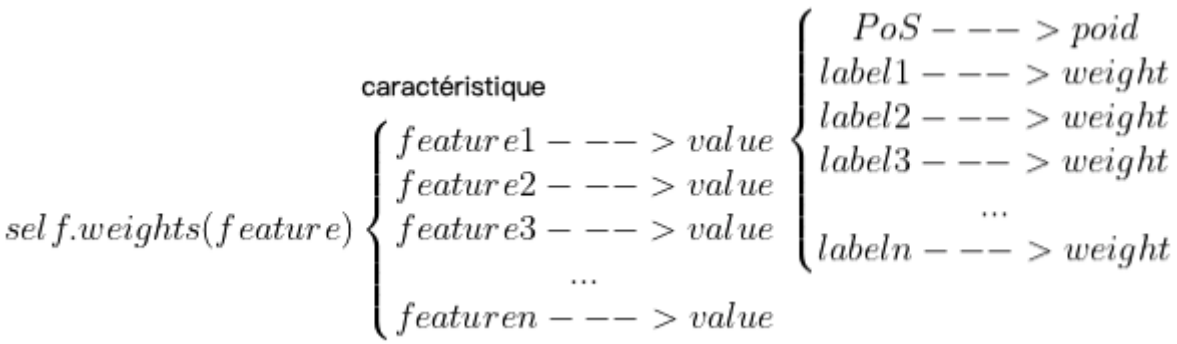
In [ ]:

```
def read_corpus(filename):
    f = open(filename, 'r',encoding='UTF-8')
    lines = f.readlines() # le corpus est lu ligne par ligne

    n = len(lines)
    i = 0

    word = [] # liste qui sauvegardera les mots
    POS = []  # liste qui sauvegardera les PoSs correspondants


    # la première condition est imposée pour les corpus GSD et Spoken French, la deuxième pour le corpus Old French, afin de localiser les mots et les PoS organisé dans les tableaux
    while i < n:
        if (lines[i][2:6] == 'text' and lines[i-1][2:6] == 'sent') or\
             (lines[i][2:6] == 'sent' and lines[i-1][2:8] == 'newdoc'):
```

3.2 Développment du modèle

Nous avons implémenté un algorithme qui définit manuellement des caractéristiques pour chaque mot lu, et stocke les PoS dans un dictionnaire. Cet algorithme contient aussi un perceptron moyenné qui permet de s'optimiser afin de calculer le poids idéal pour chaque caractéristique associée à chaque PoS.

Une fois les poids trouvés après plusieurs itérations d'auto-optimisation, l'algorithme génère un modèle qui stocke les poids idéaux. En modifiant les caractéristiques, on peut obtenir des modèles différents sous forme d'un dictionnaire des dictionnaires. Ce dernier possède les caractéristiques commme entrées, et les dictionnaire contenant les POS et les poids correspondants comme valeurs. Il est représenté graphiquement ci-dessous.



Perceptropn moyenné

Voici des explications détaillées sur le perceptron moyenné que nous avons implémenté. L'algorithme est divisé en plusieurs parties, dont la définition des caracteristques (fonctions *_get_features* et *add*), la prédiction de PoS durant l'apprentissage, la mise à jour des poids, et le calcul des poids moyennés attribués à chaque PoS pour chaque caractéristique.

(1) Définition des caractéristiques

Nous avons défini plusieurs caracteristques par rapport aux mots eux-même (le suffixe, la première lettre etc.), et d'autres par rapport au contexte des mots (les mots avant et après, leurs PoS etc.) Nous avons aussi défini des étiquettes pour marquer le début (par '-START-') et la fin (par '-END-') des contextes. D'abord, nous n'avons pas mis les caractéristiques avec les marqueurs de context. Au cours de l'optimisation, ces derniers ont été ajoutés petit à petit pour établir un modèle optimisé. Voici le code des fonctions *_get_features* et *add*:

In [24]:

```
def _get_features(self, i, word, context, prev, prev2):
# extraire les caractéristiques des mots, renvoyer un dic de features
    def add(name, *args):  # lier les chaine de caractères, pour nommer les feat
ures
        features[' '.join((name,) + tuple(args))] += 1

    i += len(self.START)
    features = defaultdict(int)
# personnaliser des features comme : préfixes, suffixes, mot précédent, mot suiv
ant...
    add('bias')
    add('i suffix', word[-3:])  #le suffixe du mot
    add('i pref1', word[0])     #la première lettre du mot
    add('i-1 tag', prev)        #la nature de mot i-1
    add('i-2 tag', prev2)       #la nature de mot i-2
    add('i tag+i-2 tag', prev, prev2)
    add('i word', context[i])
    add('i-1 tag+i word', prev, context[i])
    add('i-1 word', context[i - 1])
    add('i-1 suffix', context[i - 1][-3:])
    add('i-2 word', context[i - 2])
    add('i+1 word', context[i + 1])
    add('i+1 suffix', context[i + 1][-3:])
    add('i+2 word', context[i + 2])
    return features
```

(2) Prédiction de PoS

Ensuite, on commence à lire et prédire le PoS de chaque élément dans l'ensemble d'apprentissage. La prédiction se fait soit à travers les comparaisons avec certains PoS des mots fréquents qui sont déjà identifiés, soit à travers les caractéristiques internes et externes d'un mot lu. Dans le deuxième cas, on appelle la fonction *predict* pour calculer le score sous forme d'un produit du poids et valeurs d'une caratéristique. Les meilleurs scores seront envoyés pour la mise à jour des poids.

In [26]:

```
def predict(self, features):  # le vecteur de caractéristique * le vecteur de po
ids => l'étiquette lexicale
    scores = defaultdict(float)
    for feat, value in features.items():
        if feat not in self.weights or value == 0:
            continue
        weights = self.weights[feat]
        for label, weight in weights.items():
            scores[label] += value * weight

    # renvoyer les meilleurs scores
    return max(self.classes, key=lambda label: (scores[label], label)) # renvoye
r le label ayant le socre le plus élevé, si des scores sont égals, on prend la l
ettre la plus grande
```

(3) Mise à jour des poids

Une fois les meilleurs scores trouvés, les poids seront mis à jour. Si la prédiction est correcte, la mise à jour ne sera pas déclenchée. Si incorrecte, on ajoute 1 point au poids associé au PoS correcte (gold), et -1 au poids associé au PoS incorrectement prédit (pred).

In [27]:

```
def update(self, truth, guess, features): # mis à jour les valeurs des pondérati
ons

    def upd_feat(c, f, w, v): # c: la nature correcte/prédite du mots; f:feature
; w:la valeur du poids correspondant; v:1(ou -1)
        param = (f, c)
        self._totals[param] += (self.i - self._tstamps[param]) * w  # l'accumula
tion :[i (à ce moment là) - i (mis à jour la prochaine fois)]*la valuer du poid
        self._tstamps[param] = self.i # i mis à jour à ce moment-là
        self.weights[f][c] = w + v  # mis à jour de la valeur du poid

    self.i += 1
    if truth == guess:  # si le prédiction est correct, on n'en revouvelle pas
        return None
    for f in features:
        weights = self.weights.setdefault(f, {})  # renvoyer la valuer du f qui
 est dans le key du dic weights , si il n'y a pas la valeur, renvoit un dic vid
e:{}
        upd_feat(truth, f, weights.get(truth, 0.0), 1.0)  # plus 1 dans des pois
de caractéristiques(feature, la nature correcte du mot)
        upd_feat(guess, f, weights.get(guess, 0.0), -1.0) # moins 1 dans des poi
s de caractéristiques(feature, la nature correcte du mot)
    return None
```

(4) Calcul de poids moyennés

La procédure précédente se déroule à plusieurs itérations. A la fin de toutes les itérations, on récupère tous les poids optimisés et en calcule des moyennes. C'est le calcul de poids moyennés. Après cela, un modèle contenant des poids sera établi et sauvegardé.

In [29]:

```
def average_weights(self):  # calculer les poids moyen
    for feat, weights in self.weights.items():
        new_feat_weights = {}
        for clas, weight in weights.items():
            param = (feat, clas)
            total = self._totals[param]
            total += (self.i - self._tstamps[param]) * weight
            averaged = round(total / float(self.i), 3)     # pour chaque poid :
 faire la moyenne par rapport ses itérations de i fois
            if averaged:
                new_feat_weights[clas] = averaged
        self.weights[feat] = new_feat_weights
    return None
```

3.3 Evaluations et optimisation du modèle

Pour évaluer le modèle établi, on passe les ensembles d'évaluation in-domain et hors-domaines pour calculer le taux de précision. A chaque test, un taux sera calculé est affiché pour indiquer la fiabilité du modèle. Les données testées sont lues entrée par entrée, et le PoS de chaque entrée est prédit en même temps. Les PoS prédits sont comparés avec les bonnes étiquette annotées dans les ensembles d'évaluation, et on fera le calcul des taux de précision.

Afin d'optimiser notre modèle, nous avons modifié les caractéristiques au fur et à mesure et observé les changements sur le taux de précision. Finalement, nous avons obtenus six modèles, le sixième modèle étant considéré comme le modèle optimisé pour ce projet.

Le code pour passer les évaluations est comme suit :

In [ ]:

```
if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
#implémenter la façon / le forme de l'exportation par la foncito logging.basicCo
nfig
    tagger = PerceptronTagger(False)
    try:
        tagger.load(PICKLE)
        print(tagger.tag("Comment allez-vous ?"))
        logging.info('Start testing...')
        right = 0.0
        total = 0.0
        sentence = ([], [])
        for line in open('data/DSG_French/my_gsd_test.txt'):
            params = line.split()
            if len(params) != 2: continue
            sentence[0].append(params[0])  # mots
            sentence[1].append(params[1])  # POS
            if params[0] == '.':
# le point indique la fin de phrase, combiner les mots en une phrase reliée par
 des espaces
                text = ''
                words = sentence[0]
                tags = sentence[1]
                for i, word in enumerate(words):
                    text += word
                    if i < len(words): text += ' '
                outputs = tagger.tag(text)  # outputs comme[(mot1, pos1),(mot2,
 pos2)...]
                assert len(tags) == len(outputs)  # il y aurais une erreur tandi
s que des codes après assert sont fausses
                total += len(tags)
                for o, t in zip(outputs, tags):
                    if o[1].strip() == t: right += 1  # la prédiction est correc
te : right+1
                sentence = ([], [])
        logging.info("Precision : %f", right / total)  # le taux de la prédictio
n
```

4 Résultats et discussions

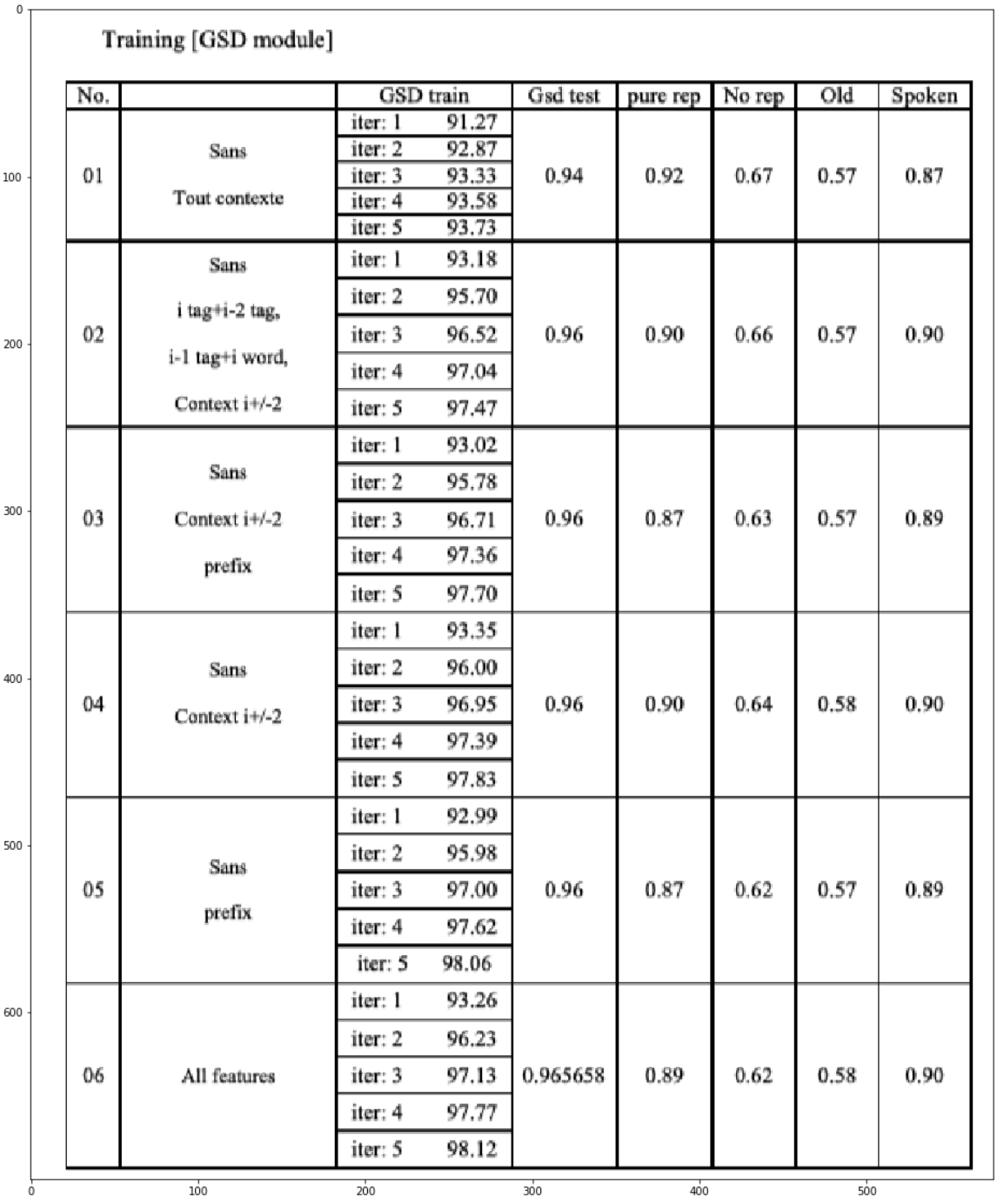
Nous avons développé six modèles, chaque modèle produisant des taux de précision et des erreurs fréquentes pour les corpus différents. Les résultats sont représentés en graphes, matrices de confusion et tableaux en DataFrame. Les modifications des caractéristiques pour chaque modèle et les taux de précision pour chaque type d'ensemble de données sont résumés dans l'image qui suit :

In [2]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

plt.figure(figsize=(18,20))
I = mpimg.imread('images/GSD_model.png')
plt.imshow(I)
```

Out[2]:

```
<matplotlib.image.AxesImage at 0x11a0f8780>
```



Training [GSD module]

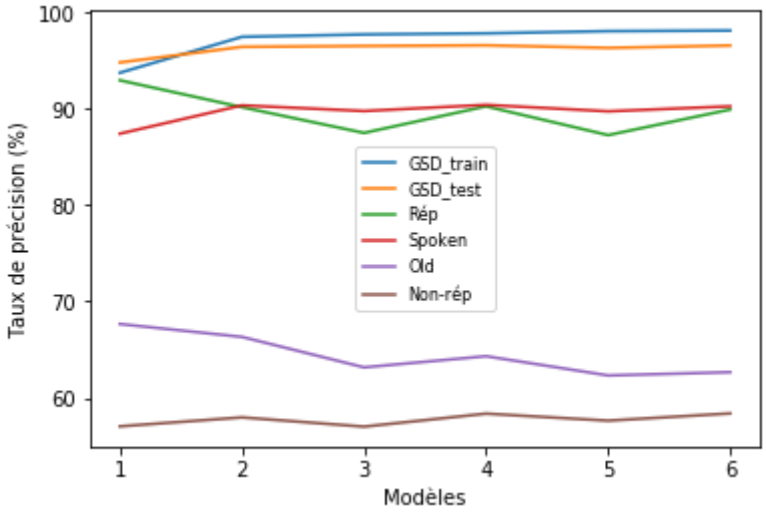
No.		GSD train		Gsd test	pure rep	No rep	Old	Spoken
01	Sans Tout contexte	iter: 1	91.27	0.94	0.92	0.67	0.57	0.87
		iter: 2	92.87					
		iter: 3	93.33					
		iter: 4	93.58					
		iter: 5	93.73					
02	Sans i tag+i-2 tag, i-1 tag+i word, Context i+/-2	iter: 1	93.18	0.96	0.90	0.66	0.57	0.90
		iter: 2	95.70					
		iter: 3	96.52					
		iter: 4	97.04					
		iter: 5	97.47					
03	Sans Context i+/-2 prefix	iter: 1	93.02	0.96	0.87	0.63	0.57	0.89
		iter: 2	95.78					
		iter: 3	96.71					
		iter: 4	97.36					
		iter: 5	97.70					
04	Sans Context i+/-2	iter: 1	93.35	0.96	0.90	0.64	0.58	0.90
		iter: 2	96.00					
		iter: 3	96.95					
		iter: 4	97.39					
		iter: 5	97.83					
05	Sans prefix	iter: 1	92.99	0.96	0.87	0.62	0.57	0.89
		iter: 2	95.98					
		iter: 3	97.00					
		iter: 4	97.62					
		iter: 5	98.06					
06	All features	iter: 1	93.26	0.965658	0.89	0.62	0.58	0.90
		iter: 2	96.23					
		iter: 3	97.13					
		iter: 4	97.77					
		iter: 5	98.12					

4.1 Taux de précision

Les taux de présicion pour les modèles et les corpus sont représentés par le graphe suivant :

In [5]:

```
y1 = [93.74, 97.47, 97.72, 97.83, 98.06, 98.12]
y2 = [94.81, 96.43, 96.52, 96.59, 96.32, 96.57]
y3 = [92.96, 90.17, 87.50, 90.26, 87.27, 89.93]
y4 = [87.42, 90.36, 89.77, 90.41, 89.74, 90.26]
y5 = [67.67, 66.33, 63.17, 64.33, 62.33, 62.67]
y6 = [57.05, 57.98, 57.02, 58.37, 57.64, 58.39]
x = ['1','2','3','4','5','6']
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.plot(x, y4)
plt.plot(x, y5)
plt.plot(x, y6)
plt.xlabel('Modèles')
plt.ylabel('Taux de précision (%)')
plt.legend(['GSD_train','GSD_test','Rép','Spoken','Old','Non-rép'], loc='center'
, fontsize=8)
plt.show()
```



On peut constater que la performance du modèle pour les évaluations in-domain (supérieur à 90%) est en générale meilleure que celle pour les évaluations hors-domaine (qui peut chuter jusqu'à un niveau inférieur à 60%, notamment pour le Old French). Toutefois, le modèle a du mal à générer de bons résultats pour l'ensemble d'évaluation hors-vocabulaire, qui est considéré comme un corpus in-domain.

On constate aussi que les taux de précision ne fluctuent pas beaucoup, mais sont croissants pour quatre catégories (GSD_train, GSD_test, Spoken et Non-rép) et décroissants pour les deux autres catégories (Rép et Old). Il est possible d'interpréter que l'optimisation jusqu'au sixième modèle est généralement effective.

Néanmoins, nous n'avons pas eu le temps pour optimiser le modèle afin de faire augmenter les taux de précision pour les catégories Rép et Old. Ce sera un objectif à atteindre dans le futur.

4.2 Erreurs fréquentes

Les matrices de confusion et les trois erreurs les fréquentes pour chaque modèle et les corpus sont générés par les codes suivants. Ici, nous prenons un exemple du modèle optimisé avec le GSD_test comme ensemble d'évaluation.

In [17]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

pred_res = []
gold_res = []


f = open("fr_gsd_test_confusion.txt", "r")
lines = f.readlines()

for line in lines:
    res = line.strip("\n").split("\t")
    pred_res.append(res[0])
    gold_res.append(res[1])

data = {"y_pred": pred_res, "y_gold": gold_res}
df = pd.DataFrame(data)
labels = np.unique(df.values)
cm = confusion_matrix(df["y_pred"], df["y_gold"], labels=labels)

XX,YY = np.meshgrid(range(cm.shape[0]), range(cm.shape[1]))

df_out = pd.DataFrame({
    "values": cm.reshape(-1),
    "i_gold": XX.reshape(-1),
    "i_pred": YY.reshape(-1)
})


"""
Générer un tableau affichant les trois erreurs les plus fréquentes

"""

df_out["is_correct"] = (df_out["i_gold"] == df_out["i_pred"])
df_out["label_gold"] = df_out["i_gold"].apply(lambda elem: labels[elem])
df_out["label_pred"] = df_out["i_pred"].apply(lambda elem: labels[elem])
df_out[~df_out["is_correct"]].sort_values("values", ascending=False).iloc[:3]
```

Out[17]:

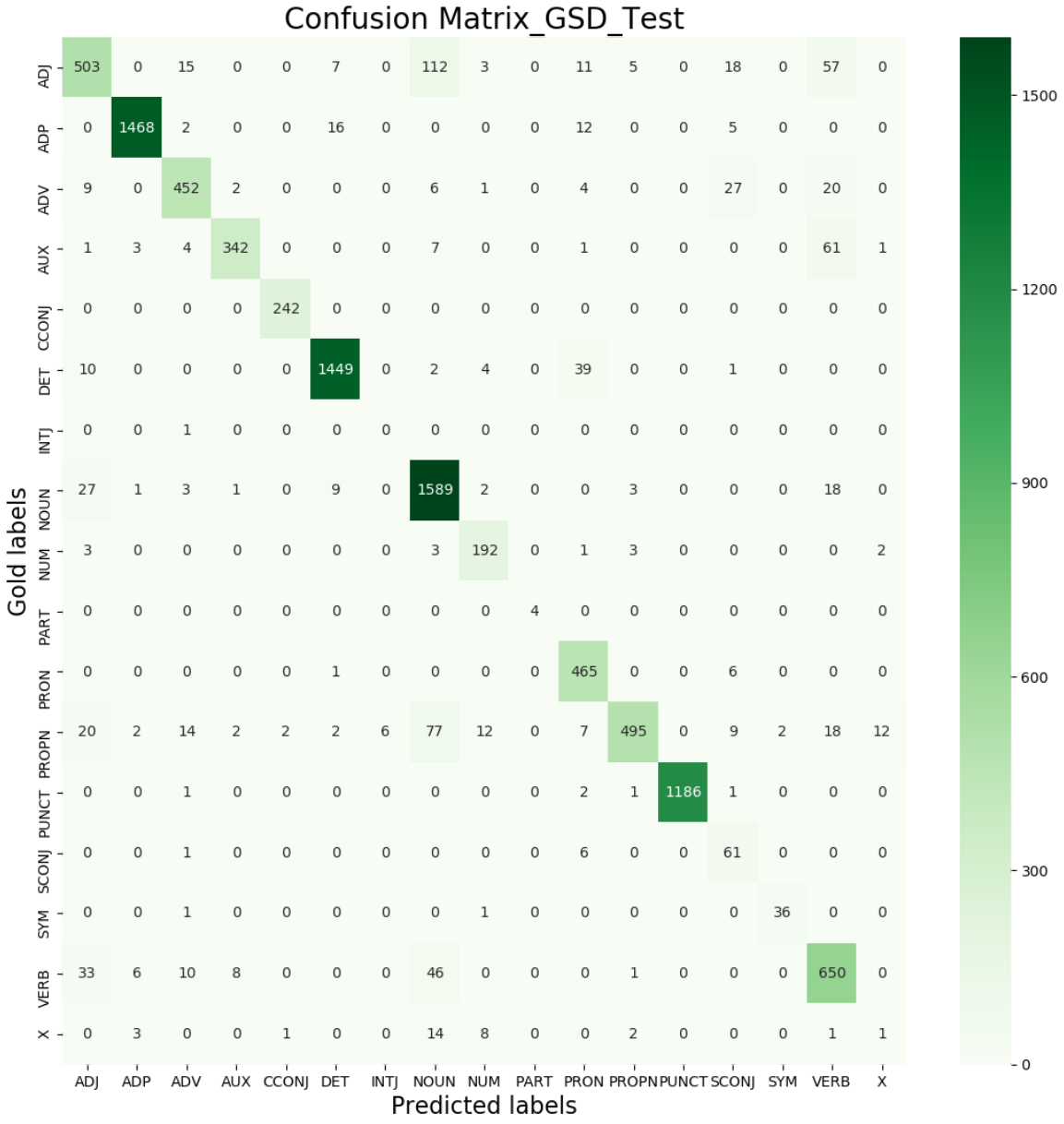
	values	i_gold	i_pred	is_correct	label_gold	label_pred
7	112	7	0	False	NOUN	ADJ
194	77	7	11	False	NOUN	PROPN
66	61	15	3	False	VERB	AUX

In [18]:

```
"""
Générer une matrice de confusion

"""

fig = plt.figure(figsize=(13, 13), dpi=100)
ax = fig.add_subplot(1, 1, 1)
ax.set_title(f"Confusion Matrix_GSD_Test", fontsize=20)
sns.heatmap(cm, annot=True, cmap="Greens", fmt="d")
ax.set_xlabel("Predicted labels", fontsize=16)
ax.set_xticklabels(labels, fontsize=10)
ax.set_ylabel("Gold labels", fontsize=16)
ax.set_yticklabels(labels, fontsize=10)
```

Out[18]:

```
[Text(0, 0.5, 'ADJ'),
 Text(0, 1.5, 'ADP'),
 Text(0, 2.5, 'ADV'),
 Text(0, 3.5, 'AUX'),
 Text(0, 4.5, 'CCONJ'),
 Text(0, 5.5, 'DET'),
 Text(0, 6.5, 'INTJ'),
 Text(0, 7.5, 'NOUN'),
 Text(0, 8.5, 'NUM'),
 Text(0, 9.5, 'PART'),
 Text(0, 10.5, 'PRON'),
 Text(0, 11.5, 'PROPN'),
 Text(0, 12.5, 'PUNCT'),
 Text(0, 13.5, 'SCONJ'),
 Text(0, 14.5, 'SYM'),
 Text(0, 15.5, 'VERB'),
 Text(0, 16.5, 'X')]
```


Confusion Matrix_GSD_Test
Gold labels
Predicted labels
ADJ ADP ADV AUX CCONJ DET INTJ NOUN NUM PART PRON PROPN PUNCT SCONJ SYM VERB X
1500
1200
900
600
300
0

On constate que pour les catégories GSD_train, GSD_test et Old French, les noms et pronoms ont tendance à être incorrectement prédits (noms pris pour pronoms ou adjectifs et pronoms pour noms, par exemple). Par contre, pour le Spoken French, ce sont plutôt les interjections qui ne sont généralement pas bien prédites.

Il est évident que les types des erreurs les plus fréquentes varient en fonction de catégorie. Une stratégie d'optimisation du modèle, ce serait de définir les caractéristiques propres aux types de PoS qui sont mal reconnues facilement. Or, nous n'avons pu définir que des caractéristiques globales pour tous les types de PoS. Cette tâche est aussi à accomplir dans le futur.

5 Références

Corpus

fr_gsd-ud-train.conllu (https://github.com/UniversalDependencies/UD_French-GSD/blob/master/fr_gsd-ud-train.conllu)

fr_gsd-ud-test.conllu (https://github.com/UniversalDependencies/UD_French-GSD/blob/master/fr_gsd-ud-test.conllu)

fro_srcmf-ud-test.conllu (https://github.com/UniversalDependencies/UD_Old_French-SRCMF/blob/master/fro_srcmf-ud-test.conllu)

fr_spoken-ud-test.conllu (https://github.com/UniversalDependencies/UD_French-Spoken/blob/master/fr_spoken-ud-test.conllu)

Articles

"Perceptron" sur Wikipedia (https://en.wikipedia.org/wiki/Perceptron)

Perceptron moyenné par Guillaume Wisniewski (https://moodle.u-paris.fr/pluginfile.php/649525/mod_resource/content/1/perceptron.pdf)