# Rapport d'expérimentation du travail 1 $TP_1$

Sekou Kaba
Salah Eddine Khalil
Filière : Maîtrise en informatique - Intelligence Artificielle

07 October 2022

## 1 Introduction

Ce rapport a pour but de se familiariser avec l'analyse de texte et aux difficultés que cela comporte même pour accomplir des tâches relativement simples, d'exercer des expérimentations avec des expressions régulières pour extraire des informations dans des textes, de comprendre comment on peut construire un modèle N-gramme simple à partir d'un corpus de textes, d'utiliser des modèles N-grammes pour choisir les mots les plus appropiés à un contexte et d'utiliser des techniques de classification de textes pour catégoriser des textes courts. Ce travail nous a permis de manipuler des expressions régulières dans un premier temps, des modèles N-grammes dans un second temps, puis des modèles de classification dans un but d'extraction d'information, de prédiction et de classification de texte. Le tout est codé en Python 3.9.7 en utilisant principalement les librairies NLTK, re, Scikit-learn et Spacy.

## 2 Tâhce 1 : Les expréssions régulières

Dans cette partie, on a un fichier texte ingredients.txt contenant les ingrédients de diverses recettes ainsi que leur quantité correspondante. Le but est, à partir de ce fichier, d'extraire et de séparer les ingrédients ainsi que les quantités correspondantes en utilisant une ou plusieurs expressions régulières en cascade.

### 2.1 Présentation du problème

On commence par regarder comment est organisé le fichier duquel il faut tirer les informations et de quelle forme sont les réponses attendues.
0,25 tasse beurre
0,5 tasse sucre
2 bananes en pure
Le fichier des réponses attendues correspondates est le suivant :
0,25 tasse beurre QUANTITE: 0,25 tasse INGREDIENT: beurre
0,5 tasse sucre QUANTITE: 0,5 tasse INGREDIENT: sucre

2 bananes en purée QUANTITE: 2 INGREDIENT: bananes.

Pour quantités, on attend et la quantité et l'unité de mesure. On remarque également que certaines expressions ne doivent pas être prises en compte comme le "en purée".
Au niveau du fichier en entrée, on peut d'ores et déjà analyser des structures réecurentes qui vont nous aider à construire nos expressions régulières. Les quantités correspondent toujours (à une ou deux exceptions) à la première partie de la ligne du fichier. Dans les cas les plus simples, tout ce qui suit correspond à la partie ingrédients de la réponse. Attention cependant, la plupart des lignes en entrée obtiennent des séquences qui ne sont ni attendues dans quantité ni dans ingrédient. Il faudra donc les éliminer.

## 2.2 L'expréssion régulière

Pour construire l'exprésion régulières on peut soit tout faire en une seule ligne et ensuite extraire les groupes qui nous intéressent ou alors utiliser plusieurs expréssions régulières plus petites en cascade. Après avoir testé les deux approches, nous avons opté pour la prémière option

Les expréssions régulières sont les suivantes :

```
REGEX = r"""
    (([0-99])+(\,\d{0,})?(\/[0-9])?
    (\snoix?)?(\senveloppe?)?(\tasse?)?(\smorceau?)?(\strait?)?)
    (\ ?(gousses|(B|b)ouquet|(R|r)ondelle|feuilles|tasse(s|)|
        cuillÃ¨re(s|)((\sÃ(\s*)cafÃ©)|
        (\sÃ(\s*)soupe))?|m(L|l)|(g\s)|l(b|B)|tranches|pintes|gallon|pincÃ©e|
        cl|oz|(c\.\s{0,}Ã\s{0,}((s\.)|(c\.)|(\.c)|(\.s)|thÃ©|soupe))))?|
    ((.*)\)|
    (U|u)ne pincÃ©e|(A|a)u goÃ»t|(Q|q)uelques|(F|f)euilles)
"""
REGEX_TEXT = r"^(de |d'|dâ€™|du|des)"
pattern = re.compile(REGEX, re.VERBOSE | re.IGNORECASE)
item_pattern = re.compile(REGEX_TEXT, re.VERBOSE | re.IGNORECASE)
#
```

## 2.3 Performance

Au niveau des performance on est à : 50% des ingredients correspondent à la solution contenue dans le fichier solution et 50% ne correspondent pas.

```
Result: CORRECT(65) x WRONG(65) - 50.00%
```

## 2.4 Interpretation

Dans la partie principale du script (hors de la fonction getIngredient, on peut avoir les erreurs faites par la fonction get ingredient en commentant/décommentant une ligne dans un statement else. Pour les quantités, les erreur sont sovent liées au postulat de départ qui admettait que toutes les quantités étaient en première partie de ligne et qu'elles étaient composées de chiffres. En effet, certaines quantités sont de la forme 'quelques ...' ou alors ne sont pas positionnées en début de ligne. On a aussi le problème de certaines quantités qui sont une concaténation de deux sous chaines non juxtaposées dans la chaîne en entrée. Concernant les

ingrédients, beaucoup d'erreur sont dûs à des précisions ou adjectif descriptifs que l'on aurait pas dû matcher. Une grande partie des erreurs est également imputable aux erreurs sur les quantités puisque si on n'a pas réussi à matcher la quantité, on ne pourra rien substituer à la chaîne pour appliquer les regex des ingrédients qui par conséquent, renverront elles aussi une chaîne erronée ou nulle.

# 3 Tâhce 2 : Modèle de langue N-grammes- comme le disait le proverbe...

L'objectif de cette partie est la mise en place d'un modèle N-gramme afin de prédire des mots dans un proverbe à trou à partir de propositions. Pour cela, on utilise cinq fonctions que nous allons détailler. Ces fonctions se chargent dans l'ordre de l'entrainement du modèle à partir d'un corpus du lissage de Laplace sur le modèle, l'obtention des comptes reconstitués d'un tuple, le calcul des logprob et la perplexité d'une séquence de mots et enfin une fonction permettant de tester tous les proverbes et propositions du fichier texte.

## 3.1 Présentation du problème

Pour construire notre modèle Ngramme, on dispose d'un corpus de proverbes dans proverbes.txt et de proverbes à trous ainsi que de propositions dans test proverbes.txt On va donc devoir dans l'ordre entraîner le modèle, puis le lisser, puis, pour chaque proposition d'un proverbe, calculer le logprob et la perplexité du proverbe complété par la proposition et choisir la proposition avec le maximum de logprob ou le minimum de la perplexité comme prédiction.

## 3.2 Entraînement du modèle

A partir du corpus qu'on a à disposition, on va entraîner notre modèle. Pour chaque N-gramme (ici 1-gramm, 2-gramme et 3-gramme), on va extraire pour chaque proverbe du corpus, les tuples de N mots se suivant dans le proverbe. Ces tuples formeront les clés d'un dictionnaire dont les valeurs seront incrémentées de 1 à chaque fois que l'on rencontre le tuple. On va également avoir besoin de compter les tokens de début et fin de phrases. Pour ce faire, on va tokenizer le proverbe grâce à NLTK et y ajouter des symboles de début et de fin. Ainsi pour la construction des modèles n-grammes,pour chaque modèle n-gramme, nous avons construit le vocabulaire qui contient tous les types de mots présents dans notre fichier proverbes.txt, ensuite nous avons construit les n-grammes en respectant les consignes du travail, nous avons lisser le modèle et le stocker dans un dictionnaire.La fonction train$_{m}odelsnousapermisd'arriveràcebut.$

### 3.3 Estimation

Pour pouvoir compléter un proverbe en utilisant le fichier text proverbes.txt, pour un proverbe incomplet donné avec une liste de choix de mot, nous avons remplacé *** par chaque mot contenu dans la liste de choix pour avoir une proposition pour chaque mot de cette liste et pour chacune de ces propositions on l'appliquer de même manière que pendant la phase d'entraînement, la tokénisation, l'ajout des symbôles de début et de fin et selon le critète choisi à l'entrée, on calcule soit la perplexité ou le logprob, après on le stocke dans un dictionnaire. Après si le critère choisi est la perplexité on choisi la proposition qui a la perplexité minimale ou le logprob maximal si le critère choisi et le LOGPROB. On retourne donc la proposition et sa valeur selon le critère choisi. La fonction $\text{cloze}_t estnousapermisd'arriveràcebut$.

### 3.4 Test sur les proverbes

Afin de tester le modèle, on appelle la fonction clozetest qui lit chaque proverbe de l'ensemble de test et crée un dictionnaire de proverbe complèté par chacune des propositions. A noter que les propositions répresentent les valeurs du dictionnaire et les clés réprésentent les valeurs de la probabilité selon le critère choisi. On obtient donc au fianal la proposition ayant la plus haute valeur de logprob si le critère=logprob ou la plus petite valeur si le critère=perplexity et sa valeur correspondante. La fonction se charge de retourner le proverbe reconstitué avec la plus grande probabilité selon le critère.

### 3.5 Résultats

Après avoir testé le modèle avec pour critère perplexity et pour n=3 on obtient :

```
Nombre de proverbes pour entraîner les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'pr
ogramme']
        Solution = a beau mentir qui mange de loin , Valeur = 1802.5365484919682

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau temps qui vient de loin , Valeur = 1736.7143819934936

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pou
voir']
        Solution = lâ€™occasion fait le bonheur , Valeur = 2226.8736663807913
```

Pour n=1 avec perplexity comme critère

```
Nombre de proverbes pour entraîner les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'pr
ogramme']
        Solution = a beau mentir qui vient de loin , Valeur = 434.15031558678913

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau temps qui vient de loin , Valeur = 327.68151952835603

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pou
```

Pour n=2 avec perplexity comme critère

```
Nombre de proverbes pour entraîner les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'prog
ramme']
        Solution = a beau mentir qui mange de loin , Valeur = 1173.9398479086117

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau temps qui vient de loin , Valeur = 1110.8277034612327

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pouvo
ir']
        Solution = lâ€™occasion fait le bonheur , Valeur = 1606.7129584469426

        Proverbe incomplet: aide-toi, le ciel tâ€™*** , Options: ['aidera', 'a', 'en', 'armera']
        Solution = aide-toi, le ciel tâ€™armera , Valeur = 2511.788582016655
```

Pour n=1 avec logprob comme critère

```
Nombre de proverbes pour entraîner les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'pr
ogramme']
        Solution = a beau mentir qui programme de loin , Valeur = -12.126704472843189

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau visage qui vient de loin , Valeur = -12.126704472843189

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pou
voir']
        Solution = lâ€™occasion fait le pouvoir , Valeur = -12.126704472843189
```

Pour n=2 avec logprob comme critère

```
Nombre de proverbes pour entrainer les modeles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'prog
ramme']
        Solution = a beau mentir qui programme de loin , Valeur = -12.126704472843189

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau prÃªcher qui vient de loin , Valeur = -12.126704472843189

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pouvo
ir']
        Solution = lâ€™occasion fait le malin , Valeur = -12.128316602903544

        Proverbe incomplet: aide-toi, le ciel tâ€™*** , Options: ['aidera', 'a', 'en', 'armera']
        Solution = aide-toi, le ciel tâ€™armera , Valeur = -12.127027043004599

        Proverbe incomplet: annÃ©e de gelÃ©e, *** de blÃ© , Options: ['annÃ©e', 'faute', 'annÃ©es',
```

Pour n=3 avec logprob comme critère

```
Nombre de proverbes pour entrainer les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

        Proverbe incomplet: a beau mentir qui *** de loin , Options: ['vient', 'part', 'mange', 'prog
ramme']
        Solution = a beau mentir qui programme de loin , Valeur = -12.126704472843189

        Proverbe incomplet: a beau *** qui vient de loin , Options: ['mentir', 'prÃªcher', 'temps',
'visage']
        Solution = a beau visage qui vient de loin , Valeur = -12.126704472843189

        Proverbe incomplet: lâ€™occasion fait le *** , Options: ['larron', 'malin', 'bonheur', 'pouvo
ir']
        Solution = lâ€™occasion fait le pouvoir , Valeur = -12.126704472843189

        Proverbe incomplet: aide-toi, le ciel tâ€™*** , Options: ['aidera', 'a', 'en', 'armera']
        Solution = aide-toi, le ciel tâ€™armera , Valeur = -12.126704472843189
```

Nous constatons que les différents modèles ne capturent pas trèss bien le langage utilisé dans les proverbes

On constate aussi que la longueur "n" a un impact, plus "n" est grand plus la perplexité augmente et plus le logprob diminue un petit peu.

on remarque qu'il n'y a presque aucune différence entre la classification via le logprob ou via la perplexit´e. Les modèles obtiennent à peu prêt les même résultats et ce pour chaque longueur de N-grammes. Cela est assez logique car il s'agit simplement de deux façons différentes de calculer la probabilité totale d'une séquence de plusieurs probabilités. Compte tenu que cela n'a presque aucun impact au niveau des probabilités des N-grammes individuelles, il est normale qu'on obtienne a peu prêt la même hiérarchie de probabilité.
Nous avons ensuite fait notre propre modèle sans respecter les signatures du fichier t2completerproverbes voir les résultats obtenus

```
Nombre de proverbes pour entraîner les modèles :  3108

Nombre de tests du fichier ./test_proverbes.txt: 46

Les résultats des tests sont:

## NGRAM(1) GUESS | Laplace 1
# 0.01 seconds to create table prod with 4472 elements
a beau mentir qui part de loin | Prob 0.00022358859698155394
# 0.43 seconds to create table prod with 4471 elements
# Log Prob is -248.6791900433568 - Perplexity is 1.3807164513178058e-75
a beau mentir qui vient de loin | Prob 0.00022358859698155394
# 0.47 seconds to create table prod with 4471 elements
# Log Prob is -257.889530415333 - Perplexity is 2.3308577904141576e-78
lâ€™occasion fait le larron | Prob 0.00022358859698155394
# 0.47 seconds to create table prod with 4471 elements
# Log Prob is -276.3102111592854 - Perplexity is 6.642603543208124e-84
aide-toi, le ciel tâ€™en | Prob 0.00022358859698155394
```

scale

# 4 Tâche4 : CLASSIFICATION DE TEXTES - IDENTIFICATION DE LANGUE

## 4.1 LogProb

```
        Vietnamese ['Nguyen', 'Ho', 'Le
Classification via logprob et avec n = 1
6/10 pour la langue Arabic
7/10 pour la langue Chinese
4/10 pour la langue Czech
1/10 pour la langue Dutch
2/10 pour la langue English
3/10 pour la langue French
5/10 pour la langue German
5/10 pour la langue Greek
4/10 pour la langue Irish
4/10 pour la langue Italian
7/10 pour la langue Japanese
4/10 pour la langue Korean
7/10 pour la langue Polish
2/10 pour la langue Portuguese
3/10 pour la langue Russian
2/10 pour la langue Scottish
3/10 pour la langue Spanish
7/10 pour la langue Vietnamese
76/180 pour tout les noms
Précision : 0.4222222222222222
```

```
Classification via logprob et avec n = 2
5/10 pour la langue Arabic
9/10 pour la langue Chinese
6/10 pour la langue Czech
2/10 pour la langue Dutch
8/10 pour la langue English
7/10 pour la langue French
7/10 pour la langue German
9/10 pour la langue Greek
3/10 pour la langue Irish
10/10 pour la langue Italian
10/10 pour la langue Japanese
6/10 pour la langue Korean
6/10 pour la langue Polish
4/10 pour la langue Portuguese
10/10 pour la langue Russian
2/10 pour la langue Scottish
7/10 pour la langue Spanish
8/10 pour la langue Vietnamese
119/180 pour tout les noms
Précision : 0.6611111111111111
```

```
Classification via logprob et avec n = 3
7/10 pour la langue Arabic
8/10 pour la langue Chinese
6/10 pour la langue Czech
2/10 pour la langue Dutch
10/10 pour la langue English
3/10 pour la langue French
6/10 pour la langue German
9/10 pour la langue Greek
3/10 pour la langue Irish
10/10 pour la langue Italian
10/10 pour la langue Japanese
5/10 pour la langue Korean
6/10 pour la langue Polish
4/10 pour la langue Portuguese
10/10 pour la langue Russian
0/10 pour la langue Scottish
6/10 pour la langue Spanish
7/10 pour la langue Vietnamese
112/180 pour tout les noms
Précision : 0.6222222222222222
```

## 4.2 Perplexité

```
Classification via perplexity et avec n = 1
6/10 pour la langue Arabic
7/10 pour la langue Chinese
4/10 pour la langue Czech
1/10 pour la langue Dutch
2/10 pour la langue English
3/10 pour la langue French
5/10 pour la langue German
5/10 pour la langue Greek
4/10 pour la langue Irish
4/10 pour la langue Italian
7/10 pour la langue Japanese
4/10 pour la langue Korean
7/10 pour la langue Polish
2/10 pour la langue Portuguese
3/10 pour la langue Russian
2/10 pour la langue Scottish
3/10 pour la langue Spanish
7/10 pour la langue Vietnamese
76/180 pour tout les noms
Précision : 0.4222222222222222
```

```
Classification via perplexity et avec n = 2
5/10 pour la langue Arabic
9/10 pour la langue Chinese
6/10 pour la langue Czech
2/10 pour la langue Dutch
8/10 pour la langue English
7/10 pour la langue French
7/10 pour la langue German
9/10 pour la langue Greek
3/10 pour la langue Irish
10/10 pour la langue Italian
10/10 pour la langue Japanese
6/10 pour la langue Korean
6/10 pour la langue Polish
4/10 pour la langue Portuguese
10/10 pour la langue Russian
2/10 pour la langue Scottish
7/10 pour la langue Spanish
8/10 pour la langue Vietnamese
119/180 pour tout les noms
Précision : 0.6611111111111111
```

```
Classification via perplexity et avec n = 3
7/10 pour la langue Arabic
8/10 pour la langue Chinese
6/10 pour la langue Czech
2/10 pour la langue Dutch
10/10 pour la langue English
3/10 pour la langue French
6/10 pour la langue German
9/10 pour la langue Greek
3/10 pour la langue Irish
10/10 pour la langue Italian
10/10 pour la langue Japanese
5/10 pour la langue Korean
6/10 pour la langue Polish
4/10 pour la langue Portuguese
10/10 pour la langue Russian
0/10 pour la langue Scottish
6/10 pour la langue Spanish
7/10 pour la langue Vietnamese
112/180 pour tout les noms
Précision : 0.6222222222222222
```

On remarque donc que la capacité générale de classification du modèle semble changer selon la longueur de l'historique. En effet, la précision moyenne du modèle augmente lorsqu'on passe d'un modèle uni-gramme à bi-gramme (passant de 0.42 à 0.66) alors que la précision diminue légèrement lorsqu'on passe à la classification via un modèle tri-gramme (0.62). Il peut sembler contre-intuitif que d'augmenter la longueur des séquences puisse diminuer la performance du modèle mais il faut garder à l'esprit que l'augmentation de la longueur diminue l'impact de chacun des éléments, y compris les uni-grammes très discriminants pour les différents langages.

On note aussi que certaines langues semblent être bien mieux classifier que d'autre. Les langues ayant un très mauvais ratio sur les séquences de tri-grammes/bigrammes risquent fort probablement de ne pas avoir une bonne performance sur les uni-grammes. Cela parait logique, une bonne classification en unigramme signifie que cette langue utilise des lettres qui ne sont pas souvent utilisées dans les autres langues. On peut donc déduire que ces lettres discriminantes formerons des bi-grammes/tri-grammes qui seront eu aussi discriminant dans leur modèles respectif. Il y a aussi certaines langues comme le Dutch, le Irish, le Portuguese et le Scottish qui obtiennent des mauvaises performance et ce peu importe la longueur des N-grammes. Cela signifie simplement que ces langues n'ont pas beaucoup de séquences de une, deux ou trois lettres qui se distingue des autres langages.

Finalement, on remarque qu'il n'y a aucune différence entre la classification via le logprob ou via la perplexité. Les modèles obtiennent exactement les même résultats et ce pour chaque langue et pour chaque longueur de N-grammes. Cela est assez logique car il s'agit simplement de deux façon différentes de calculer la probabilité totale d'une séquence de plusieurs probabilités. Compte tenu que cela n'a aucun impact au niveau des probabilités des N-grammes individuelles, il est normale qu'on obtienne la même hiérarchie de probabilité et que la langue la plus probable soit la même.

# 5 Conclusion

Ce premier travail aura été l'occasion de mettre en pratique les connaissances théoriques précédemment engrangées et de se confronter aux limites et difficultés du passages de l'algorithme et de la théorie à l'implémentation en pratique.

## TÂCHE 3 – CLASSIFICATION DE TEXTES – ANALYSE DE SENTIMENT

### 1. Présentation du Problème :

Cette tâche vise à classifier des critiques de produits selon leur polarité (positive ou négative). Plus précisément, la tâche consiste à évaluer un analyseur de sentiment construit à partir des mots des textes. En comparant 3 configurations :

- Les mots des textes sans normalisation
- Les mots normalisés avec le *stemming*
- Les mots normalisés avec la lemmatisation

les algorithmes d'apprentissage utilisés pour accomplir cette tâche sont :

**naïf bayésien** (*Naive Bayes*) et **régression logistique**.

### 2. Les mots de texte sans normalisation :

D'abord, on construit notre jeu de données pour l'entrainement ainsi que pour le test

```
X = np.append(partitions['train_pos_fn'],partitions['train_neg_fn'])
y = np.append(np.ones(len(partitions['train_pos_fn'])), np.zeros(len(partitions['train_neg_fn'])))

X_test= np.append(partitions['test_pos_fn'],partitions['test_neg_fn'])
y_test = np.append(np.ones(len(partitions['test_pos_fn'])), np.zeros(len(partitions['test_neg_fn'])))
```

La tokénization, et la construction d'un vecteur de mots en éliminant les stopwords :

```
vectorizer = CountVectorizer(stop_words = stop, tokenizer = word_tokenize,lowercase=True)
if normalization=='words':

    XVectorised = vectorizer.fit_transform(X)
    X_test_vectorized = vectorizer.transform(X_test)
```

#### 2.1. Modèle naïf bayésien :

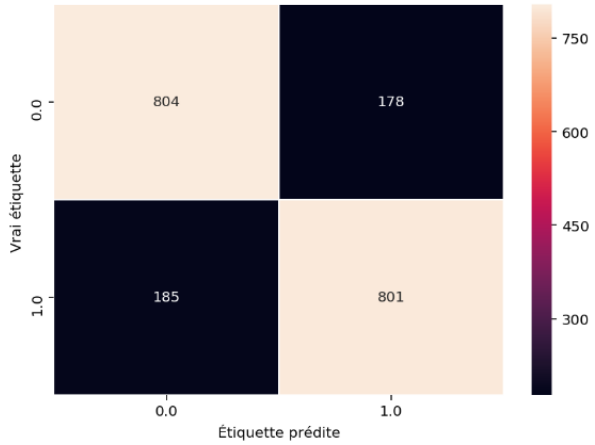
```
train_and_test_classifier(reviews_dataset, model='NB', normalization='words')
```

Les performances obtenues :

On remarque que notre modèle a une accuracy assez élevé au niveau de test ce qui va lui permettre de bien généraliser sur des exemples jamais vus.
On peut dire que parmi les 986 exemples de sentiments positives, il a pu classifier 801 comme étant Positives, et parmi les 982 exemples de négatives, il a bien classifié 804 de ces derniers, ce qui donne un score assez important.

```
Taille des partitions du jeu de données
        train_pos_fn : 3000
        train_neg_fn : 3000
        test_pos_fn : 986
        test_neg_fn : 982
Accuracy - entraînement:  0.8053333333333335
Accuracy - test:  0.8155487804878049
Matrice de confusion:  [[804 178]
 [185 801]]


Version graphique de la matrice de confusion
```



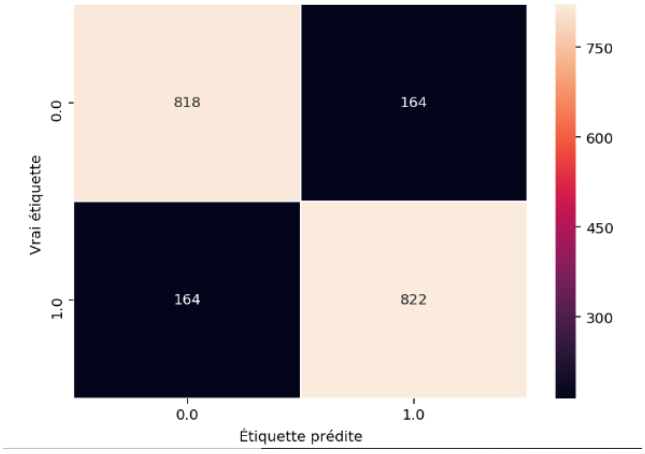

## 2.2. Modèle régression logistique:

```
results = train_and_test_classifier(reviews_dataset, model='LG', normalization='words')
```

Les performances obtenues :

```
Taille des partitions du jeu de données
        train_pos_fn : 3000
        train_neg_fn : 3000
        test_pos_fn : 986
        test_neg_fn : 982
Accuracy - entraînement:  0.8233333333333333
Accuracy - test:  0.8333333333333334
Matrice de confusion:  [[818 164]
 [164 822]]


Version graphique de la matrice de confusion
```



On Remarque que ce modèle lui aussi performe bien avec un taux de 83.33% pour une bonne classification.
Donc, on peut conclure à partir de cet exemple que le modèle de régression logistique reste le bon choix au niveau des performances pour une classification binaire. Cependant pour la durée d'exécution, on constate que le modèle naive bayès est plus rapide(23 sec) que celui du régression logistique(29 sec).

## 3. Les mots normalisés avec le stemming :

### 3.1. Modèle naïf bayésien :

```
results = train_and_test_classifier(reviews_dataset, model='NB', normalization='stem')
```

On applique le stemming à l'aide de nltk.stem.porter.PorterStemmer() sur les jeux d'entrainement et de test :

```
elif normalization=='stem':
    XStem=[]
    X_test_Stem=[]
    for i in range(len(X)):
        XStem.append(stem(X[i]))
    for i in range(len(X_test)):
        X_test_Stem.append(stem(X_test[i]))

    XVectorised = vectorizer.fit_transform(XStem)
    X_test_vectorized = vectorizer.transform(X_test_Stem)
```

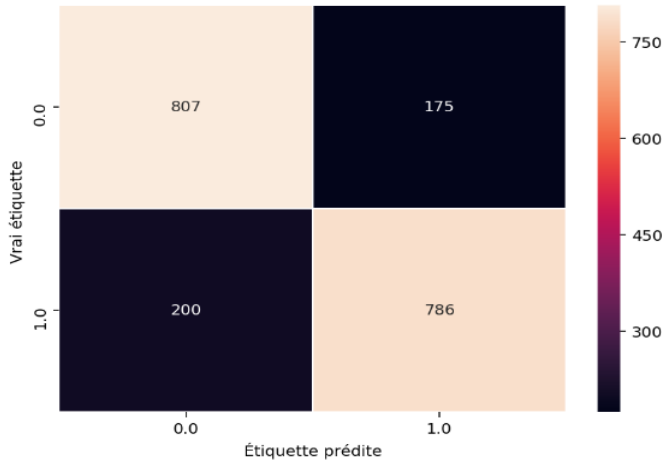
La fonction **stem** :

```
def stem(string):
        porter_stemmer = nltk.stem.porter.PorterStemmer()
        tokenizedStr = word_tokenize(string)
        stemsStr = list(map(lambda word: porter_stemmer.stem(word), tokenizedStr))
        space = " "
        return space.join(stemsStr)
```

Les performances obtenues :

```
Accuracy - entraînement:  0.8081666666666667
Accuracy - test:  0.8094512195121951
Matrice de confusion:  [[807 175]
 [200 786]]

Version graphique de la matrice de confusion
```



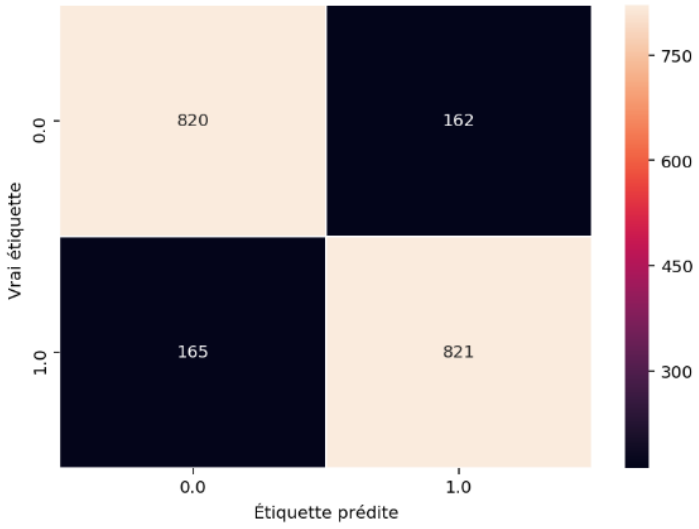

### 3.2. Modèle régression logistique:

```
results = train_and_test_classifier(reviews_dataset, model='LG', normalization='stem')
```

Les performances obtenues :

```
Accuracy - entraînement:  0.8236666666666667
Accuracy - test:  0.8338414634146342
Matrice de confusion:  [[820 162]
 [165 821]]

Version graphique de la matrice de confusion
```



Donc, on peut conclure à partir de cet exemple que le modèle de régression logistique reste le bon choix au niveau des performances pour une classification binaire. Cependant la durée d'exécution est beaucoup plus longue dans ce cas de stemming (à peu près 1 min 46 sec) pour les deux modèles.

## 4. Les mots normalisés avec la lemmatisation :

Lemmatisation :

```
else:
    Xlemm=[]
    X_test_lemm=[]

    for i in range(len(X)):
        Xlemm.append(lemm(X[i]))
    for i in range(len(X_test)):
        X_test_lemm.append(lemm(X_test[i]))


    XVectorised = vectorizer.fit_transform(Xlemm)
    X_test_vectorized = vectorizer.transform(X_test_lemm)
```

La fonction **lemm** :

```
def lemm(string):
        wordnet_lemmatizer = nltk.stem.wordnet.WordNetLemmatizer()
        tokenizedStr = word_tokenize(string)
        lemsStr = []

        for token in tokenizedStr:
            lemmatized_word = wordnet_lemmatizer.lemmatize(token)
            lemsStr.append(lemmatized_word)

        return " ".join(lemsStr)
```

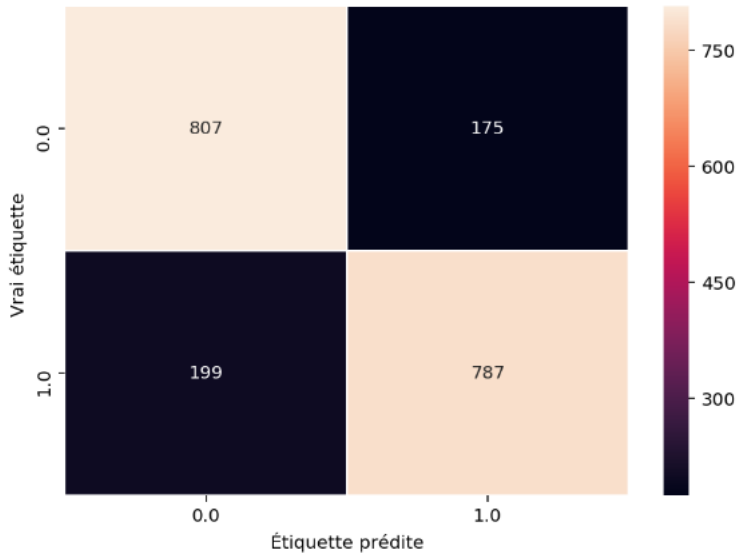
### 4.1. Modèle naïf bayésien :

```
results = train_and_test_classifier(reviews_dataset, model='NB', normalization='lemm')
```

Les performances obtenues :

```
Accuracy - entraînement:  0.8106666666666666
Accuracy - test:  0.8099593495934959
Matrice de confusion:  [[807 175]
 [199 787]]
```

Version graphique de la matrice de confusion



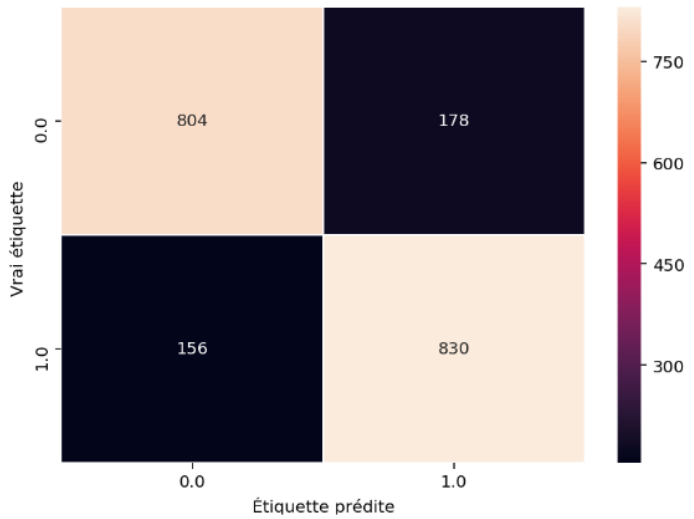

### 4.2. Modèle régression logistique:

```
results = train_and_test_classifier(reviews_dataset, model='LG', normalization='lemm')
```

<u>Les performances obtenues :</u>

```
Accuracy - entraînement:  0.8220000000000001
Accuracy - test:  0.8302845528455285
Matrice de confusion:  [[804 178]
 [156 830]]
```

Version graphique de la matrice de confusion



On remarque que les deux modèles donnent de bonnes performances, la regression logistique reste la plus privilégiée parmi les deux en terme d'accuracy, le temps d'exécution des deux modèles est d'environ "1 min 9 sec" qui est mieux que celui du normalisation avec stemming .

Donc On peut conclure que la configuration la plus intéressante est celle du lemmatisation avec la régression logistique qui donne de bonnes performances au niveau du généralisation, ainsi qu'un temps d'exécution relativement acceptable par rapport au cas de normalisation avec stemming .