# Modèle génératif et modèle discriminant pour l'étiquetage morpho-syntaxique : Rendu

Pacôme Perrotin

## Table des matières

## 0.1 Introduction

Ce projet a pour objectif de fournir un étiquetage morpho-syntaxique simple à partir d'un corpus complet. Ce travail est exécuté par le biais d'un modèle caché de Markov basé sur des bigrammes, c'est à dire un algorithme présupposant que les mots possèdent des états et que ces états se composent deux par deux par un certaine loi probabiliste. Ce HMM est implémenté de deux façon différentes, l'une grâce à un modèle génératif, l'autre grâce à un modèle discriminant.

Dans ce document nous présenterons l'implémentation de ce double problème sous de nombreux aspects. D'abord nous approcherons en plus de détails le modèle mathématique ici implémenté, puis nous aborderons l'implémentation elle-même. Une note sera adressée durant ces deux parties quant aux améliorations qui ont été ajoutées au modèle de base. Enfin les résultats obtenus en faisant varier les paramètres proposés seront détaillés et critiqués.

# 1 Modèle

Dans ce projet nous nous intéressons à l'étiquetage d'une séquence de mots. Cet étiquetage peut être représenté de la façon suivante :

Soit $S = \{S_1, \cdots, S_N\}$ un ensemble de catégories et $O = O_1, \cdots, O_T$ une séquence de mots. L'étiquetage estimé de cette séquence est calculé par :

$$\hat{Q} = arg \max_{Q \in S^T} P(O, Q)$$

Avec $P(O, Q)$ la probabilité d'observer la séquence $O$ étiquetée par $Q$.

Le calcul en soi de cet étiquetage est pris en charge par l'algorithme de Viterbi. Cet algorithme présuppose l'existence des scores suivants :

— Pour chaque catégorie $S_i$, $\pi_i$ est le score de l'apparition de la catégorie $S_i$ au début d'une séquence observée.

— Pour chaque paire de catégorie $(S_i, S_j)$, $a_{ij}$ est le score de l'apparition successive de $S_i$, $S_j$ n'importe où dans une séquence observée.

— Pour chaque mot $V_j$ et chaque catégorie $S_i$, $b_i(j)$ est le score de l'affectation du mot $V_j$ à la catégorie $S_i$ n'importe où dans une séquence observée.

Derrière un score se cache un procédé qui donne un ordre d'importance numérique à un ensemble d'objets. Cet ordre d'importance peut-être représenté grâce à une distribution de probabilités, ou grâce à n'importe quelle autre méthode, par exemple un calcul de poids via un algorithme perceptron. Dans le reste de notre rapport, nous considérerons les scores comme devant être **minimisés** plutôt que maximisés afin d'obtenir un étiquetage optimal.

Notre approche du problème de l'étiquetage sera de calculer d'abord l'ensemble de ces scores suivant une certaine méthode grâce à un corpus d'apprentissage, puis d'appliquer l'algorithme de Viterbi sur un corpus de test pour en calculer un ensemble d'étiquettes. La comparaison de ces étiquettes calculées, $\hat{Q}$, avec les étiquettes réelles de la séquence, $Q$, nous permet d'obtenir un score de précision. Heureusement, avec l'énoncé de ce projet nous ont été fournis un corpus d'apprentissage et un corpus de test entièrement étiquetés, ce qui simplifie grandement le travail d'étiquetage automatique.

## 1.1 L'algorithme de Viterbi

L'algorithme de Viterbi est un algorithme classique de programmation dynamique de calcul sur les treillis, ici utilisé pour calculer la meilleure séquence d'étiquette sans avoir à considérer un nombre exponentiel d'éléments. Le voici sous une version proche de celle implémentée dans le projet :

1. Soit O la séquence d'observables de taille N, et $\pi_i$, $a_{ij}$ et $b_i(j)$ les fonctions de score contenues dans un HMM. On pose K le nombre d'étiquettes différentes.
2. On considèrera SCORE et BACKTRACK deux tableaux de dimension 2 et de taille $N \times K$.
3. $\forall i \in K, SCORE[i,1] \leftarrow \pi_i \times b_i(O_1), BACKTRACK[i,1] \leftarrow 0$
4. $\forall i$ de 2 à $N$, $\forall j$ de 1 à $K$ :
   4.1. $SCORE[j,i] \leftarrow max_k(SCORE[k,i-1] \times a_{kj} \times b_{jq_i})$
   4.2. $BACKTRACK[j,i] \leftarrow arg\ max_k(SCORE[k,i-1] \times a_{kj})$
5. Soit $Q = q_1, q_2, \ldots, q_N$ une séquence d'étiquette.
6. $q_N \leftarrow S_{arg\ max_k(SCORE[k,N])}$
7. $\forall i$ de $N$ à 2 :
   7.1. $q_{i-1} \leftarrow S_{BACKTRACK[q_i,i]}$
8. retourner $Q$.

Reste à savoir comment extraire du corpus d'apprentissage les scores utilisés dans l'algorithme. Deux modèles différents de cette extraction seront abordés lors de ce projet : un modèle génératif basé sur un calcul de probabilité, et un modèle discriminant basé sur un algorithme de perceptron.

## 1.2 Modèle génératif

Ce modèle considère nos scores de la façon suivante :

— Scores initiaux :

$$\pi_i = -\ log\ P(q_1 = S_i) \approx -\ log\ \frac{C_\pi(S_i) + \alpha}{Nb_{sentences} + N \times \alpha}$$

— Scores de transition :

$$a_{ij} = \sum_{t=2}^{T} -\ log\ P(q_t = S_j \mid q_{t-1} = S_i) \approx -\ log\ \frac{C_a(S_i, S_j) + \alpha}{C(S_i) + N \times \alpha}$$

— Scores d'émission :

$$b_i(j) = \sum_{t=1}^{T} -\ log\ P(O_t = V_j \mid q_t = S_i) \approx -\ log\ \frac{C_b(S_i, V_j) + \alpha}{C(S_i) + K \times \alpha}$$

Avec, par ordre d'apparition :

— $C_\pi(S_i)$ = le nombre total de fois que $S_i$ a été observée au début d'une sous-séquence.

— $\alpha$ = le paramètre de lissage du modèle

— $Nb_{sentences}$ = le nombre total de sous-séquences dans le corpus d'apprentissage.

— $N$ = le nombre d'étiquettes différentes.

— $T$ = la longueur du corpus.

— $C_a(S_i, S_j)$ = le nombre total de fois que le bigramme $(S_i, S_j)$ a été observé dans le corpus d'apprentissage.

— $C(S_i)$ = le nombre total de fois que l'étiquette $S_i$ a été observée dans le corpus d'apprentissage.

— $C_b(S_i, V_j)$ = le nombre total de fois que l'étiquette $S_i$ a coincidée avec l'observable $V_j$ dans le corpus d'apprentissage.

— $K$ = le nombre d'observables différents.

Ce premier modèle est aussi le modèle le plus simple : étant donné un certain corpus d'apprentissage étiqueté, il se contente d'en extraire plusieurs statistiques, qu'il transforme ensuite en score grâce à une simple fraction. On notera l'utilisation systématique d'un logarithmique et d'une négation : ces modifications sont purement d'ordre pratique et permettent de repartitionner les probabilitées obtenues dans l'ensemble $[0, 1]$ sur l'ensemble $[0, +\infty[$ avec $0 \to +\infty$, ce qui donne une plus grande précision sur les données en programmation. La négation permet de ne pas travailler avec des nombres négatifs ; c'est son usage qui oblige le reste de notre programme à travailler sur la **minimisation** des scores obtenus plutôt que sur leur maximisation.

Le paramètre de lissage $\alpha$ fait partie des améliorations ajoutées au modèle initial proposé dans l'énoncé. Ce paramètre a pour mission de donner une probabilité minimale à l'ensemble des évènements ; ainsi, aucun évènement n'a de probabilité nulle (et donc de score $+\infty$). Ce paramètre aura généralement une valeur comprise dans $[0, 1]$. Cette garantie minimale de probabilité se fait en ajoutant $\alpha$ à la partie supérieure de chaque fraction, puis en ajoutant $N \times \alpha$ ou $K \times \alpha$ dans la partie inférieure de chaque fraction, afin que la valeur de la somme soit toujours équivalente à la somme des valeurs individuelles.

Ce modèle est très simple, car il n'exige qu'une seule lecture du corpus d'apprentissage, sur lequel il effectue un calcul de complexité linéaire.

## 1.3 Modèle discriminant

Ce second modèle, de type discriminant, emploie une solution plus complexe algorithmiquement ; plutôt que d'extraire directement les scores du corpus, on approxime leurs valeurs grâce à un algorithme de perceptron. Le vecteur de poids du perceptron considéré contiendra l'ensemble de nos scores $\pi_i$, $a_{ij}$ et $b_i(j)$.

Voici l'algorithme de perceptron utilisé par ce modèle, avec pour paramètre un nombre entier $I$ dans $[1, +\infty[$ :

1. Initialiser les poids $\pi_i$, $a_{ij}$ et $b_i(j)$ à 0.
2. Faire les opérations suivantes $I$ fois :
   - 2.1. Pour chaque phrase $O = O_1 \cdots O_T$ de séquence réelle d'étiquettes $Q = q_1 \cdots q_T$, faire :
     - 2.1.1. Calculer $\hat{Q} = \hat{q_1} \cdots \hat{q_T}$ grâce à l'algorithme de Viterbi sur $\pi_i$, $a_{ij}$ et $b_i(j)$

2.1.2. Si $Q \neq \hat{Q}$, faire :

2.1.2.1. $\pi_i = \pi_i - \phi_{\pi_i}(Q) + \phi_{\pi_i}(\hat{Q})$

2.1.2.2. $a_{ij} = a_{ij} - \sum_{t=2}^{T} \phi_{a_{ij}}(t, Q) + \sum_{t=2}^{T} \phi_{a_{ij}}(t, \hat{Q})$

2.1.2.3. $b_i(j) = b_i(j) - \sum_{t=1}^{T} \phi_{b_i(j)}(t, Q) + \sum_{t=1}^{T} \phi_{b_i(j)}(t, \hat{Q})$

3. Renvoyer les poids $\pi_i$, $a_{ij}$ et $b_i(j)$.

Avec, par ordre d'apparition :

— $\phi_{\pi_i}(Q) = 1$ *si* $q_1 = S_i$, 0 *sinon*

— $\phi_{a_{ij}}(t, Q) = 1$ *si* $q_{t-1} = S_i$ *et* $q_t = S_j$, 0 *sinon*

— $\phi_{b_i(j)}(t, Q) = 1$ *si* $q_t = S_i$ *et* $O_t = V_j$, 0 *sinon*

Ce perceptron , à chaque itération, diminue les poids correspondant à l'étiquetage réel et augmente les poids correspondant à l'étiquetage approximé. Cela résulte en un calcul bien plus complexe que pour le modèle précédent, qui affecte cependant bien les scores minimaux aux évènements les plus probables.

# 2 Implémentation

Notre implémentation a été réalisée en langage C, choisi pour son efficacité algorithmique. Le code qui a été produit dans le cadre de ce projet se divise en dix modules : data, evaluation, hmm, hmm init, log sum, main, parameters, parser, perceptron et viterbi. Chacun de ces modules possède un fichier en extension .h et un fichier en extension .c qui lui correspondent dans le dossier src. Afin de détailler notre implémentation, nous allons présenter ces modules un à un, dans l'ordre de complexité.

Il est à noter que chacun des fichier de ce projet dispose de commentaires qui présentent les différentes fonctions et expliquent le fonctionnement interne des fonctions les plus algorithmiques.

## 2.1 Module log sum

Le module log sum est sans doute le module le plus simple de l'ensemble du projet. Il donne accès à de très simples fonctions de calcul de probabilités sous forme logarithmique. Il permet notamment de diviser un nombre par un autre afin d'obtenir une probabilité sous forme logarithmique négative, et de multiplier deux probabilités sous forme logarithmique.

## 2.2 Module parameters

Le travail de ce module est de prendre en charge les paramètres données en ligne de commande par l'utilisateur, et de stocker les informations intéressantes sous la forme d'une structure de donnée. Les paramètres retenus dans la structure sont les suivants :

— Le bruit de l'exécution : silencieux, normal, bruyant. Définit le montant d'affichage console généré par le programme.

— Le type d'exécution : seulement le modèle discriminant, seulement le modèle génératif ou les deux en même temps. Nous permettons de tester les deux modèles consécutivement pour une seule lecture du corpus, ce qui permet de gagner un peu de temps lors de tests.

— c : chiffre flottant dans $[0, 1]$, qui correspond au ratio du nombre de phrases du corpus d'apprentissage qui sera effectivement lu et enregistré. (Voir 2.3 pour l'utilisation de ce paramètre en pratique)

— I : la constante qui indique le nombre d'itérations de l'algorithme perceptron.

— $\alpha$ : la constante de lissage utilisée dans le modèle génératif. (Voir 1.2, 2.7)

Tout ces paramètres peuvent être affectés par des paramètres en ligne de commande :

— Bruit : -s implique une exécution silencieuse, -v implique une exécution bruyante. Par défaut le bruit de l'exécution est normal.

— Type d'exécution : -C force le programme à travailler seulement sur un modèle génératif, -P force le programme à fonctionner seulement avec un modèle discriminant. Par défaut le programme exécutera les deux modèles à la suite.

— c : $-c = x$ affecte à x ce paramètre. Par défaut c possède une valeur de 1, ce qui implique que le programme lira par défaut tout le corpus lors de l'apprentissage.

— I : $-I = x$ affecte à x ce paramètre. Par défaut I vaut 10, ce qui implique 10 itérations de l'algorithme perceptron dans le cas où le modèle génératif est pris en compte.

— $\alpha$ : $-a = x$ affecte à x ce paramètre. Par défaut $\alpha$ est affecté à 0, ce qui implique que le modèle génératif ne pratique pas de lissage par défaut.

Il faut également mentionner que l'exécution du programme requiert l'usage de deux paramètres obligatoires, qui correspondent aux corpus d'ap-

prentissage et de test. Plus de détails sur l'utilisation pratique du projet sont disponible dans le readme.txt.

Ces paramètres sont lus au tout début de l'exécution du programme, puis sont disponible d'accès pour tout le reste de l'exécution grâce à la structure de donnée détaillée juste après.

## 2.3 Module data

Ce module rend accessible un ensemble d'informations cruciales à l'ensemble des modules suivants sous la forme du structure de donnée qui sera passée en paramètre de nombre de fonctions. Cette structure de donnée englobe les choses suivantes :

- L'ensemble des paramètres du programmes (détaillés en 2.2)
- Les informations détaillées du corpus d'apprentissage, qui peuvent être décrites par :
  - Une séquence de mots
  - Une séquence d'étiquettes
  - La longueur de ces deux séquences
  - Le nombre de phrases comprises dans ce corpus
- Les informations détaillées du corpus de test, qui peuvent être décrites par :
  - Une séquence de mots
  - Une séquence d'étiquettes
  - La longueur de ces deux séquences
  - Le nombre de phrases comprises dans ce corpus

Ce module est également le module qui prend en charge la lecture des corpus d'apprentissage et de test à partir des chemins donnés en paramètres du programme. Ce design permet une lecture minimale des fichiers, ici réduite à deux parcours par fichiers. Les fonctions du module parser (2.5) sont ici employées.

C'est dans ce module que vient en compte une autre amélioration qui a été apportée à l'énoncé de base ; la capacité de choisir de ne lire qu'une fraction du corpus de base. Ce choix se fait grâce au paramètre $c$ : celui çi définit quelle fraction du corpus va être lue. Si $c = 0.5$, uniquement la moitié des

phrases du corpus d'apprentissage seront lues. La lecture ne s'arrête jamais au milieu d'un phrase ; lorsque $c \times N$ mots ont été lus (avec $N$ le nombre total de mots du corpus), l'algorithme de lecture s'arrête dès qu'il a finit sa phrase courante. À l'issue de la lecture, toutes les phrases lues sont inscrites dans la structure de donnée.

## 2.4 Module hmm

Ce module fournit une structure de donnée représentant les scores d'un modèle caché de Markov. Ces différends scores, $\pi_i$, $a_{ij}$ et $b_i(j)$, sont cruciaux dans notre projet. Ils sont stockés dans la structure proposée par ce module et sont manipulés exclusivement par elle.

## 2.5 Module parser

Ce module fournit un ensemble de fonctions pratique permettant la lecture d'un fichier. Il permet notamment d'extraire simultanément en un nombre minimal de parcours le nombre de lignes du fichier, la liste de mots qu'il contient, la liste d'étiquettes correspondant à ces mots qu'il contient, et le nombre de phrases contenues. Les fichiers acceptés sont de la forme suivante :

$$
\begin{array}{c}
O_{1_1}\ q_{1_1} \\
O_{1_2}\ q_{1_2} \\
\ldots\ldots \\
O_{1_{n_1}}\ q_{1_{n_1}}
\end{array}
$$

$$
\begin{array}{c}
O_{2_1}\ q_{2_1} \\
O_{2_1}\ q_{2_1} \\
\ldots\ldots \\
O_{2_{n_2}}\ q_{2_{n_2}}
\end{array}
$$

$$\ldots\ldots$$

$$
\begin{array}{c}
O_{m_1}\ q_{m_1} \\
O_{m_2}\ q_{m_2} \\
\ldots\ldots \\
O_{m_{n_m}}\ q_{m_{n_m}}
\end{array}
$$

Avec $m$ le nombre de sous séquences, et $n_i$ la taille de la sous séquence $i$.

## 2.6 Module viterbi

Ce module est sans doute le plus important, puisqu'il implémente l'algorithme de Viterbi. Il s'agit d'un algorithme déjà détaillé dans ce rapport dans la partie 1.1, nous n'ajouterons donc pas grand chose dans la présente section.

## 2.7 Module hmm init

Ce module a pour principale mission d'initialiser les paramètres du HMM d'après les règles du modèle génératif. C'est en effet dans ce module que sont présentes les fonctions comptant les occurences d'évènements dans le corpus d'apprentissage. Pour ce faire, le module hmm init utilise une structure de donnée permettant de retenir les informations suivantes au fur et à mesure de la lecture :

- Le nombre d'états différends $K$
- Le nombre d'observables différends $V$
- La taille de la séquence total lue $N$
- Un tableau de taille $K$ contenant les occurences de chaque étiquette en début de phrase
- Le nombre de phrases $M$
- Un tableau de dimension 2 et de taille $K \times K$ contenant les occurences de transition d'état
- Un tableau de dimension 2 et de taille $K \times V$ contenant les occurences d'émissions
- Un tableau de taille $K$ contenat les occurences d'apparition totale de chaque état

Ces informations sont accumulées pour chaque phrase du corpus lues séparément. Elle sont enfin utilisées à la fin de la lecture pour générer les scores du HMM comme décrit dans la section 1.2.

## 2.8 Module perceptron

Ce module a pour principale mission d'initialiser les paramètres du HMM d'après les règles du modèle discriminant. C'est ici que l'on va trouver l'implémentation de l'algorithme perceptron explicité en section 1.3.

### 2.9 Module main

Enfin, le module main contient la fonction principale du programme. Cette fonction effectue les opérations suivantes :

1. Initialisation d'une structure de données DATA en fonction des paramètres en ligne de commande (Voir 2.3)
2. Si les paramètres d'exécution le demandent, exécuter l'algorithme de viterbi par modèle génératif
3. Et si les paramètres d'exécution le demandent, exécuter l'algorithme de viterbi par modèle discriminant

## 3 Résultats

Suite à l'implémentation de ce programme, nous avons collecté des résultats sur les différends modèles en faisant varier nos paramètres. La section qui va suivre va détailler ces résultats et fournir une conclusion sur chacun d'entre eux. Nous conclurons ensuite ce rapport en prenant un peu de recul sur l'ensemble de ces résultats, et de leur signification dans le cadre du travail demandé.

### 3.1 Résultats via le modèle génératif

Nos tests s'articulent autour de deux paramètres : le paramètre de lissage $\alpha$ et le paramètre de taille de corpus $c$. Pour chacune de ces variations, nous donnerons une table correspondant aux performances d'étiquetage, et une table de performance en temps.

Note : l'ensemble des valeurs de précisions sont en pourcentage. Il s'agit du montant d'étiquettes qui ont été bien estimées par l'algorithme.

La première table que nous allons consulter représente les résultats en efficacité du projet en fonction de $\alpha$ et $c$ :

| % de précision | c = 1.0 | c = 0.9 | c = 0.8 | c = 0.7 | c = 0.6 | c = 0.5 | c = 0.4 |
|---|---|---|---|---|---|---|---|
| $\alpha = 0.0$ | 60.7% | 58.4% | 56.0% | 53.9% | 51.6% | 48.9% | 45.5% |
| $\alpha = 0.1$ | 94.4% | 94.3% | 94.1% | 94.0% | 93.8% | 93.6% | 93.3% |
| $\alpha = 0.2$ | 94.2% | 94.2% | 94.0% | 94.0% | 93.9% | 93.6% | 93.3% |
| $\alpha = 0.3$ | 94.3% | 94.1% | 94.0% | 93.8% | 93.6% | 93.4% | 93.0% |
| $\alpha = 0.4$ | 94.1% | 94.0% | 93.8% | 93.6% | 93.4% | 93.3% | 92.9% |
| $\alpha = 0.5$ | 94.0% | 93.8% | 93.6% | 93.4% | 93.3% | 93.1% | 92.6% |
| $\alpha = 0.6$ | 94.0% | 93.6% | 93.6% | 93.3% | 93.2% | 92.8% | 92.4% |
| $\alpha = 0.7$ | 93.9% | 93.6% | 93.5% | 93.2% | 92.9% | 92.6% | 92.1% |
| $\alpha = 0.8$ | 93.7% | 93.4% | 93.3% | 93.0% | 92.7% | 92.4% | 91.9% |
| $\alpha = 0.9$ | 93.6% | 93.3% | 93.1% | 92.8% | 92.5% | 92.2% | 91.6% |
| $\alpha = 1.0$ | 93.4% | 93.2% | 93.0% | 92.7% | 92.3% | 92.0% | 91.4% |

Par observation, on sépare ces résultats en deux parties, constituées par :

— La première ligne, bornée dans $[45.5\%, 60.7\%]$, qui regroupe les résultats sans lissage

— Le reste du tableau, borné dans $[91.4\%, 94.4\%]$, qui regroupe les résultats avec lissage

Cette observation est une claire preuve que l'utilisation du lissage permet une bien meilleure précision de notre modèle génératif, étant donné notre jeu de données.

La seconde observation que l'on peut faire se concentre sur la seconde partie précédemment distinguée. On observe en effet dans cette partie que les valeurs diminuent très régulièrement de gauche à droite ainsi que de haut en bas ; il y a une augmentation très lisse de notre pourcentage d'erreur en fonction de la fraction du corpus que l'on considère et de la valeur de lissage.

Les conséquences immédiates de tels résultats sont, d'un première part, que le modèle génératif sera de plus en plus préçis en fonction du montant d'informations d'apprentissage ; et d'autre part, que le paramètre de lissage doit être strictement positif pour donner de bons résultats, et que ces résultats seront visiblement d'autant plus préçis que ce paramètre sera faible. Quelques tests successifs ont permis de baliser l'optimum de ce paramètre à environ $\alpha = 0.008$.

Jetons maintenant un coup d'oeil sur cette matrice contenant les temps de calcul correspondant aux résultats obtenus dans la matrice précédente :

| Temps | c = 1.0 | c = 0.9 | c = 0.8 | c = 0.7 | c = 0.6 | c = 0.5 | c = 0.4 |
|---|---|---|---|---|---|---|---|
| $\alpha = 0.0$ | $2.88s$ | $2.87s$ | $2.88s$ | $2.87s$ | $2.86s$ | $2.85s$ | $2.85s$ |
| $\alpha = 0.1$ | $3.00s$ | $2.99s$ | $2.99s$ | $3.00s$ | $2.99s$ | $2.99s$ | $2.99s$ |
| $\alpha = 0.2$ | $3.00s$ | $3.00s$ | $2.99s$ | $3.00s$ | $2.99s$ | $2.99s$ | $3.01s$ |
| $\alpha = 0.3$ | $3.00s$ | $3.00s$ | $3.00s$ | $2.99s$ | $2.98s$ | $2.98s$ | $2.98s$ |
| $\alpha = 0.4$ | $3.00s$ | $3.00s$ | $3.00s$ | $2.99s$ | $3.00s$ | $2.99s$ | $2.99s$ |
| $\alpha = 0.5$ | $3.00s$ | $3.00s$ | $3.05s$ | $3.01s$ | $3.01s$ | $2.99s$ | $2.98s$ |
| $\alpha = 0.6$ | $3.01s$ | $3.00s$ | $3.00s$ | $2.99s$ | $2.98s$ | $2.99s$ | $2.98s$ |
| $\alpha = 0.7$ | 3.01 | $3.00s$ | $2.99s$ | $2.99s$ | $3.00s$ | $2.98s$ | $2.98s$ |
| $\alpha = 0.8$ | 3.00 | $3.00s$ | $3.07s$ | $3.00s$ | $2.99s$ | $2.99s$ | $2.98s$ |
| $\alpha = 0.9$ | $3.03s$ | $3.00s$ | $2.99s$ | $3.00s$ | $2.99s$ | $2.99s$ | $2.99s$ |
| $\alpha = 1.0$ | $2.99s$ | $2.98s$ | $2.98s$ | $2.97s$ | $2.98s$ | $2.98s$ | $2.97s$ |

Devant analyse, ces résultats en temps du modèle génératif apparaissent très constant. Le programme prend toujours entre 2.85 et 3.05 secondes pour effectuer son étiquetage. Le paramètre $\alpha$ semble avoir une incidence ; lorsqu'il est égal à 0 ou 1, le temps global diminue de quelques centièmes de secondes. Cela s'explique vraisemblablement par l'optimisation de calcul du C lorsqu'il n'a pas à utiliser de nombre flottant.

Le paramètre $c$, quant à lui, semble avoir une très légère influence sur le temps d'exécution. Ignorer 60% du corpus nous sauve dans les deux centième de seconde. C'est un résultat attendu, puisque le modèle génératif est linéaire en la taille du corpus d'apprentissage.

Devant ces résultats, il est correct de dire que le modèle génératif est un très bon modèle conçernant notre corpus. Le faible taux d'erreur s'accompagne d'une très bonne garantie en temps ; cela est idéal lorsque l'on veut estimer la valeur optimale d'un paramètre, ou effectuer un étiquetage sur un très grand corpus.

## 3.2 Résultats via le modèle discriminant

Dans le cadre du modèle discriminant, nous avons effectué des tests de nature très similaire. Au lieu de faire varier le paramètre $\alpha$, nous avons fait

varier le paramètre $I$ qui représente le nombre d'itération de l'algorithme perceptron utilisé dans le modèle discriminant. Ce paramètre va avoir, comme nous le verrons, un grand impact autant sur le temps de calcul que la précision des résultats.

Jetons pour commencer un regard sur ce tableau répertoriant les pourcentages de précision obtenus par notre modèle discriminant :

| % de précision | c = 1.0 | c = 0.9 | c = 0.8 | c = 0.7 | c = 0.6 | c = 0.5 | c = 0.4 |
|---|---|---|---|---|---|---|---|
| $I = 10$ | 94.3% | 93.7% | 94.0% | 93.1% | 92.0% | 92.3% | 91.5% |
| $I = 9$ | 93.8% | 93.1% | 93.5% | 93.0% | 93.5% | 92.0% | 92.0% |
| $I = 8$ | 94.0% | 93.2% | 93.1% | 93.1% | 93.0% | 92.4% | 91.9% |
| $I = 7$ | 94.0% | 93.5% | 93.8% | 92.6% | 93.4% | 93.4% | 91.9% |
| $I = 6$ | 94.1% | 93.0% | 93.0% | 92.5% | 93.0% | 91.4% | 92.3% |
| $I = 5$ | 94.0% | 93.0% | 93.5% | 92.8% | 91.6% | 92.3% | 91.5% |
| $I = 4$ | 93.5% | 92.6% | 93.3% | 92.2% | 92.3% | 93.0% | 90.9% |
| $I = 3$ | 92.9% | 92.8% | 93.6% | 92.6% | 93.0% | 90.4% | 91.8% |
| $I = 2$ | 94.0% | 92.8% | 93.0% | 90.8% | 90.9% | 91.0% | 90.4% |
| $I = 1$ | 91.4% | 92.0% | 89.7% | 91.9% | 90.6% | 86.2% | 88.8% |

De ces données se dégagent plusieurs tendances. Pour commencer, on notera que la diminution indépendante ou simultanée des paramètres $c$ et $I$ a un effet globalement négatif sur la précision de notre modèle. La seconde observation est que les résultats deviennent de plus en plus instable lorsque $I$ s'approche de 0 : on note à la dernière ligne par exemple des fluctuations très aléatoires, le score montant de 91% à 92%, puis descendant jusqu'à 89%, alternant ainsi jusqu'à obtenir le minimum en terme de score de 88.8% à la dernière case.

Ces données pointent vers le fait assez simple que notre modèle obtiendra de meilleurs résultats s'il dispose de plus de données. Il obtiendra également une meilleure précision lorsque l'on augmente le nombre d'itérations. En effet, avec peu d'itérations, les erreurs que fait le perceptron n'ont pas le temps de se corriger, d'où l'instabilité globale des résultats de ce modèle.

Enfin, terminons l'analyse des données avec ce dernier tableau, assignant à chaque pourcentage de réussite précédent le nombre de secondes qui a été

nécessaire à son calcul :

| Temps | c = 1.0 | c = 0.9 | c = 0.8 | c = 0.7 | c = 0.6 | c = 0.5 | c = 0.4 |
|---|---|---|---|---|---|---|---|
| $I = 10$ | $146s$ | $131s$ | $117s$ | $103s$ | $88.4s$ | $74.4s$ | $59.7s$ |
| $I = 9$ | $131s$ | $118s$ | $106s$ | $92.8s$ | $79.9s$ | $67.1s$ | $54.2s$ |
| $I = 8$ | $117s$ | $105s$ | $94.2s$ | $82.7s$ | $71.4s$ | $59.9s$ | $48.5s$ |
| $I = 7$ | $103s$ | $92.4s$ | $82.9s$ | $72.6s$ | $62.8s$ | $52.6s$ | $42.7s$ |
| $I = 6$ | $88.8s$ | $80.0s$ | $71.4s$ | $62.7s$ | $54.1s$ | $45.6s$ | $37.1s$ |
| $I = 5$ | $74.3s$ | $67.0s$ | $59.9s$ | $52.8s$ | $45.6s$ | $38.5s$ | $31.3s$ |
| $I = 4$ | $60.0s$ | $54.1s$ | $48.4s$ | $42.8s$ | $37.1s$ | $31.3s$ | $25.7s$ |
| $I = 3$ | $45.7s$ | $41.4s$ | $37.0s$ | $32.7s$ | $28.5s$ | $24.2s$ | $19.9s$ |
| $I = 2$ | $31.3s$ | $28.5s$ | $25.6s$ | $22.7s$ | $19.8s$ | $17.1s$ | $14.1s$ |
| $I = 1$ | $17.1s$ | $15.6s$ | $14.1s$ | $12.8s$ | $11.3s$ | $9.84s$ | $8.43s$ |

Ici, on observe une très claire corrélation entre le temps pris par l'algorithme et le nombre de ses itérations, ce qui semble tout à fait normal. On observe un effet similaire avec le paramètre $c$ : pour 146s à $c = 1$, le temps tombe à 74.4s à $c = 0.5$, ce qui est quasiment la moitié. Les données tendent à mettre en avant une tendance linéaire entre le temps de calcul et $c$.

Ces quatre tableau offrent dans leur ensemble une idée assez claire de la situation ; et il est juste de pouvoir dire que le modèle génératif est ici un meilleur modèle que le modèle discriminant. Les deux modèles semblent offrir dans les données un optimum similaire, autour des 5.7% d'erreur. Cependant la comparaison en temps est, elle, ne porte à aucune hésitation, le modèle génératif restant remarquablement constant dans son temps d'exécution, là où le modèle discriminant prend très rapidement de longues minutes pour offrir un résultat similaire.

## 3.3 Conclusion

Le problème de l'étiquetage est, dans la pratique, un problème complexe à aborder efficacement. En effet, et comme la plupart des problèmes qu'abordent les sciences du traitement automatique de la langue, c'est un problème qui repose sur des grandes masses de données ; et cela pose plusieurs contraintes. La première, c'est qu'il n'arrive jamais en pratique que cette donnée soit d'une forme pratique et utilisable directement. La seconde, c'est que la moindre complexité algorithmique prend des dimensions astronomiques très rapidement en terme de temps d'exécution.

Ici, nous avons été chanceux ; le corpus à notre disposition était en effet entièrement étiqueté. Cette disposition des choses nous a permis de mettre en place un modèle génératif, qui possède la très grande qualité de fonctionner très vite. Ici le modèle discriminant se trouve inopérant en comparaison, car bien trop lent sur des instances de plus en plus grandes.

On serait tenté de qualifier un algorithme comme le perceptron inefficace quant au problème d'étiquetage ; cependant, les choses sont un peu plus complexes. En effet, les corpus entièrement étiquetés sont dans la pratique extrêmement rares et les informaticiens doivent constamment jouer avec données dont ils ne savent pas tout. Bien souvent les modèles génératifs sont trop simplistes pour fonctionner, car trop exigeants sur la forme du corpus.

Dans un exemple comme celui de ce projet, un modèle génératif comme nous l'avons implémenté est sans aucun doute le meilleur compromis de temps et de résultat. Essayer quelque chose d'autre, comme un perceptron, ce serait faire la même chose en moins bien ; les données dont nous disposons sont suffisament riches pour faire des prédictions précises après un simple comptage. Cependant dans la pratique les données sont toujours plus difficile à exploiter. Les modèles génératifs sont ainsi inopérants. L'informaticien, s'il veut pouvoir effectuer des prédictions et des calculs sur le langage, doit savoir s'armer dans la pratique d'outils plus complexes.