Apprentissage de règles de tagging

Brill Tagger : Un Étiqueteur basé sur des règles

Yiwei Zuo & Peng Chen

Linguistique Informatique
UFR Linguistique

Projet de fin d'études pour la Licence ès Sciences du Langage

Sous la direction de : Marie Candito

06.2025

Table des matières

Apprentissage de règles de tagging
Brill Tagger

Yiwei ZUO ; Peng CHEN

Résumé

Ce projet vise à développer un système d'étiquetage morphosyntaxique automatique basé sur l'apprentissage transformationnel, inspiré de la méthode de Brill. À partir d'un corpus annoté, le système apprend des règles permettant de corriger les erreurs d'étiquetage produites par un étiqueteur initial simple. Le rendu se divise en quatre parties :

— La Partie théorique décrit la tâche à automatiser et sa motivation, puis présente la méthode automatique adoptée ainsi que ses avantages et ses limites.

— La Partie expériences et résultats décrit les tentatives réalisées et les résultats obtenus.

— La Partie informatique présente systématiquement l'implémentation des modules en Python.

— Le Manuel utilisateur explique comment utiliser le programme avec un corpus donné.

En résumé, ce travail propose une approche simple mais efficace pour améliorer l'étiquetage morphosyntaxique automatique à l'aide de règles apprises automatiquement.

1 Partie théorique

1.1 La tâche à automatiser et sa motivation

L'étiquetage morpho-syntaxique, ou POS tagging en anglais, représente une composante fondamentale du traitement automatique des langues. Il s'agit d'une tâche consistant à assigner automatiquement à chaque mot d'un texte sa catégorie grammaticale (nom, verbe, adjectif, etc.). Notre présent projet s'inscrit dans cette perspective en visant à approfondir l'étude et le développement de méthodes efficaces pour accomplir cette annotation morpho-syntaxique. Plus précisément, lorsqu'un texte est donnée, le système d'étiquetage est capable

d'assigner automatiquement à chaque mot sa catégorie grammaticale. Ainsi, la phrase « *Colorless green ideas sleep furiously.* » sera annotée sous la forme : *Colorless/ADJ green/ADJ ideas/NC sleep/V furiously/ADV ./PONCT*

Les langues se caractérisent par une grande diversité, tant au niveau lexical que syntaxique. Lorsqu'un mot apparaît de manière isolée, il est généralement aisé de lui attribuer sa catégorie grammaticale propre. Toutefois, la catégorie grammaticale d'un mot peut varier selon son contexte syntaxique et sémantique. Cette variabilité rend impossible l'évaluation simple d'un mot à partir de sa catégorie grammaticale d'origine. C'est précisément en raison de cette complexité linguistique qu'il devient nécessaire de concevoir un système intelligent d'étiquetage morpho-syntaxique, capable d'attribuer avec précision la catégorie grammaticale appropriée à chaque mot dans un environnement textuel cohérent. Par exemple, en anglais :

(1) This is a good ***book****.*
(2) I would like to ***book*** *a table.*

le mot *book* peut être soit un nom dans *(1)*, soit un verbe dans *(2)* selon le contexte. De même, en français :

(3) Il me montre sa ***passe*** *de ski.*
(4) Elle ***passe*** *tous les matins devant cette école*

le mot *passe* peut désigner un nom dans *(3)* ou fonctionner comme un verbe dans *(4)*. Ces phénomènes d'ambiguïté contextuelle montrent bien la nécessité de concevoir un système d'étiquetage morpho-syntaxique intelligent, capable d'attribuer avec précision la catégorie grammaticale appropriée à chaque mot dans un environnement textuel cohérent.

Réaliser automatiquement cette tâche présente une valeur théorique et une importance pratique considérables. Tout d'abord, du point de vue du système technique de traitement du langage naturel, l'étiquetage des parties du discours, en tant que technologie clé de base, fournit non seulement le support nécessaire aux tâches de haut niveau telles que l'analyse syntaxique et l'étiquetage des rôles sémantiques, mais constitue également une condition préalable à la construction de systèmes d'application pratiques tels que la traduction automatique, l'extraction d'informations et le questionnement intelligent. Deuxièmement, en termes de pratique d'annotation de corpus, les méthodes d'annotation manuelle traditionnelles présentent d'importantes limites : d'une part, leur efficacité est faible et leur coût de main-d'œuvre élevé. Par exemple, un volume de plusieurs millions de mots du corpus rend l'annotation manuelle quasiment impossible ; d'autre part, la cohérence et la précision des annotations manuelles sont difficiles à garantir, notamment pour des textes spécialisés ou des phénomènes linguistiques ambigus. Par conséquent, le développement de la technologie d'étiquetage automatique des parties du discours joue un rôle clé dans le développement du traitement du langage naturel.

Parmi les approches existantes, les méthodes statistiques et celles fondées sur des règles sont particulièrement remarquables. Depuis longtemps, les méthodes statistiques dominent la

tâche d'étiquetage morpho-syntaxique (POS tagging) en traitement automatique des langues (TAL) (par.ex., [1]), tandis que les approches basées sur des règles sont souvent considérées comme moins performantes. Toutefois, cet état de fait a précisément suscité notre intérêt de recherche : nous avons souhaité réévaluer, par une exploration systématique, le potentiel des méthodes fondées sur des règles dans ce domaine, et examiner dans quelle mesure elles pourraient remettre en question la suprématie des approches statistiques.

1.2 Méthode réalisée et ses avantages et inconvénients

Dans le cadre de notre projet, nous nous sommes inspirés du *Brill Tagger*, proposé par Eric Brill en 1992 lors de la Troisième Conférence sur la linguistique informatique appliquée (ACL), dans un article intitulé *A Simple Rule-Based Part of Speech Tagger*[2]. Cet article présente un système d'étiquetage morpho-syntaxique fondé sur des règles, permettant d'attribuer automatiquement à chaque mot d'un texte sa catégorie grammaticale, telle que nom, verbe, adjectif, etc. Il y propose un étiqueteur basé sur des règles, se distinguant des approches statistiques dominantes de l'époque.

Nous avons mis en œuvre cette tâche d'automatisation en suivant une approche basée sur des règles. Afin de construire notre système d'étiquetage morpho-syntaxique, notre travail se divise en trois grandes parties : le traitement du corpus, l'annotation initiale, l'apprentissage et l'application des règles (patches). Chaque étape contribue progressivement à l'amélioration de la qualité du marquage, en partant d'une base simple jusqu'à une annotation affinée grâce à des règles apprises automatiquement.

Dans un premier temps, le corpus est divisé selon une répartition de 90% / 5% / 5% en trois parties : corpus d'entraînement, patch corpus (corpus de développement), et corpus de test.

Dans un deuxième temps, sans tenir compte du contexte, nous avons effectué une analyse de la fréquence des mots dans le corpus, puis attribué à chaque mot l'étiquette la plus probable. Par exemple :

(5) Il veut ***savoir*** *la vérité.*
(6) Le ***savoir*** *est une arme puissante.*

Dans la phrase *(5)*, *savoir* est un verbe (V), tandis que dans la phrase *(6)*, *savoir* est un nom (N). Bien que ce mot puisse appartenir à deux catégories grammaticales, nous lui attribuons par défaut l'étiquette de nom (N) lors de l'annotation, car son usage en tant que nom est plus fréquent. Lorsque qu'un mot n'apparaît nulle part dans le corpus, nous utilisons un ensemble de règles morpho-syntaxiques afin d'attribuer une étiquette grammaticale initiale. Par exemple :

(7) La ***vibration*** *du téléphone m'a réveillé.*

Dans cette phrase, nous supposons que le mot *vibration* est absent du corpus. L'une des

règles que nous avons définies stipule que tout mot se terminant par « *-tion* » doit être étiqueté comme un nom (N). Au total, huit règles différentes de ce type ont été formulées pour traiter les mots inconnus. (Voir Partie 2 pour une présentation détaillée de ces règles.)

Dans un troisième temps, après avoir appliqué l'étiquetage initial basé sur la fréquence des mots la plus élevée ainsi que les règles morpho-syntaxiques, nous avons achevé l'annotation initiale du corpus. À ce stade, nous avons conçu 15 templates, chacun définissant un contexte spécifique, un mot à corriger (A) et le mot correcteur (B). Si un certain contexte est rempli, le mot A est remplacé par le mot B, c'est-à-dire A $\rightarrow$ B. Les templates prennent la forme suivante :

- *template1* IF PREVIOUS WORD TAG IS **Z** : **A** $\rightarrow$ **B**
- *template2* IF EITHER OF PREVIOUS TWO WORDS HAS TAG **Z** : **A** $\rightarrow$ **B**
- *template3* IF PREVIOUS WORD IS **Z** AND NEXT WORD IS **W** : **A** $\rightarrow$ **B**
- $\cdots$

Chaque template peut être instancié en plusieurs règles de correction (patches) spécifiques selon le contexte. Par exemple, à partir du gabarit « LE TAG DU MOT PRÉCÉDENT EST **Z** », on peut générer les règles suivantes :

- *Patch1* IF PREVIOUS WORD TAG IS **DET** : **V** $\rightarrow$ **N**
- *Patch2* IF PREVIOUS WORD TAG IS **CLS** : **ADJ** $\rightarrow$ **V**
- $\cdots$

Ces règles spécifiques sont apprises automatiquement à partir des contextes observés dans le corpus d'entraînement. Les templates fournissent la structure générale des règles (patches), tandis que les patches en sont les instanciations concrètes, utilisées pour corriger les erreurs d'annotation initiales.

Nous générons d'abord l'ensemble des *patches* possibles à partir des contextes observés dans le corpus, en nous appuyant sur des *templates* prédéfinis. Ces patches sont ensuite filtrés afin de ne conserver que ceux qui apportent une amélioration nette sur le corpus de développement.

Le score de chaque patch est calculé en fonction du gain net, défini comme suit :

- chaque erreur corrigée rapporte **+1 point** ;
- chaque nouvelle erreur introduite entraîne **-1 point** ;
- le score est donc : **nombre de corrections - nombre d'erreurs introduites**

Les patches sont ensuite triés par ordre décroissant de leur score, puis appliqués au corpus de développement de manière itérative : à chaque étape, nous avons appliqué au corpus le patch ayant le meilleur score en mettant à jour les annotations. Les scores des patches restants sont alors recalculés en fonction du nouvel état du corpus, et le processus se répète. L'apprentissage s'arrête dès qu'aucun patch restant ne permet d'obtenir un gain net

strictement positif. Enfin, nous avons ainsi finalisé un étiqueteur, qui est capable d'apprendre automatiquement des règles.

Eric Brill [2] a clairement souligné dans son étude que les étiqueteurs basés sur des règles présentent plusieurs avantages. Tout d'abord, ils permettent de réduire de manière significative les besoins en ressources de stockage. Ensuite, un petit nombre de règles de base suffit pour établir une norme d'étiquetage claire. De plus, le processus d'optimisation du système est transparent et facile à mettre en œuvre. Enfin, ces étiqueteurs font preuve d'une excellente adaptabilité aux différents systèmes d'étiquetage, aux corpus variés et aux contextes multilingues.

Cependant, puisque les méthodes d'apprentissage basées sur des règles dépendent des instances déjà observées dans le corpus, lorsqu'un mot totalement inédit dans le corpus d'entraînement est rencontré, même s'il respecte les règles morphologiques de base de la langue, le système est très susceptible de produire une erreur d'étiquetage. Cela provoque un problème d'*out-of-domain*. L'introduction directe d'un dictionnaire externe pourrait peut-être permettre de mieux pallier cette limitation.

2 Partie expériences et résultats

2.1 Origine et format du corpus

Le corpus que nous avons utilisé dans ce projet, issu du corpus *Sequoia* (version 9.2)[3, 4], est entièrement composé de textes en langue française. Il se présente sous la forme de phrases annotées, où chaque mot est suivi de son étiquette morpho-syntaxique, séparés par une barre oblique « / » (par exemple : mot/étiquette). Les phrases sont séparées par des retours à la ligne. Voici un exemple de la première phrase du corpus :

> Cette/DET exposition/NC nous/CLO apprend/V que/CS dès/P le/DET XIIe/ADJ siècle/NC ,/PONCT à/P Dammarie-sur-Saulx/NPP ,/PONCT entre/P autres/ADJ sites/NC ,/PONCT une/DET industrie/NC métallurgique/ADJ existait/V ./PONCT

Au cours de l'avancement du projet, nous avons été confrontés à plusieurs défis techniques et questionnements théoriques. Afin d'y répondre de manière systématique, nous avons conçu et mené une série d'expériences de validation. Grâce à des testes itératifs et à une optimisation progressive, nous avons pu affiner continuellement notre approche. Ces résultats intermédiaires constituent une base pour des recherches plus approfondies, lesquelles seront détaillées dans les sections suivantes.

2.2 Amélioration de la méthode de partition du corpus

Lors de la phase de répartition des données, nous avons adopté une méthode améliorées. Au début, nous avons utilisé la méthode de découpage séquentiel (90% / 5% / 5%). Cependant, comparée à cette approche, nous avons désormais systématiquement introduit une stratégie de répartition aléatoire (90% / 5% / 5%). Cette optimisation a apporté des avantages significatifs. D'un côté, la randomisation de la distribution des données permet une évaluation plus globale et multidimensionnelle des performances du modèle ; de l'autre, elle renforce la fiabilité des résultats en réduisant considérablement le risque de surapprentissage (over-fitting) dû à un biais de distribution. Cela permet de poser un cadre plus fiable pour l'entraînement et l'évaluation.

2.3 Simplification des règles morphologiques de base

Comme mentionné dans la première partie, l'étiquetage initial ne prend pas en compte le contexte. Lorsqu'un mot n'apparaît pas dans le corpus d'entraînement, il est impossible de lui attribuer une étiquette en se basant sur sa fréquence. Pour résoudre ce problème, nous avons défini un ensemble de règles morpho-syntaxiques simples, fondées uniquement sur la forme du mot, afin d'effectuer une première étiquette de manière heuristique. Les règles utilisées (basic morphosyntactic rules) sont les suivantes :

1. Si un mot est entièrement composé de chiffres, ou représente un nombre décimal (comme `123` ou `45,67`), il est étiqueté comme **nom commun (NC)**.
   Exemples : `"123"`, `"45,6"` → **NC**
2. Si un mot se termine par l'un des suffixes suivants : `"age"`, `"tion"`, `"ment"`, `"isme"`, `"eur"`, `"ité"`, il est étiqueté comme **nom commun (NC)**.
   Exemples : `"nation"`, `"fromage"` → **NC**
3. Si un mot commence par une majuscule, a plus de 3 lettres, alors il est étiqueté comme **nom propre (NPP)**, afin d'éviter les erreurs sur les mots courts comme `"En"`.
   Exemples : `"Paris"`, `"Gutenberg"` → **NPP**
4. Si un mot se termine par l'un des suffixes suivants : `"tif"`, `"tive"`, `"eux"`, `"able"`, `"ible"`, `"ique"`, alors il est étiqueté comme **adjectif (ADJ)**.
   Exemples : `"créatif"`, `"dangereuse"`, `"adorable"` → **ADJ**
5. Si un mot se termine par `"is"` ou `"it"`, alors il est étiqueté comme **(V)**.
   Exemples : `"manger"`, `"finir"`, `"faisait"`, `"choisit"` → **V**
6. Si un mot se termine par l'un des suffixes suivants : `"er"`, `"ir"`, `"re"`, `"oir"`, `"aire"`, `"dre"`, `"tre"`, `"uire"`, `"enir"`, `"aître"`, alors il est étiqueté comme **verbe à l'infinitif (VINF)**. Exemples : `"manger"`, `"finir"`, `"croire"`, `"voir"` → **VINF**
7. Si un mot se termine par `"é"` ou `"ée"`, formes fréquentes du participe passé, alors il est étiqueté comme **verbe au participe passé (VPP)**.
   Exemples : `"mangé"`, `"arrivée"` → **VPP**

8. Si aucune des règles précédentes ne s'applique, le mot est étiqueté par défaut comme **nom commun (NC)**, car c'est la catégorie la plus fréquente dans le corpus.
   Exemple : `"machine"` (si elle ne correspond à aucune règle précédente) → **NC**

Concernant cette partie, nous avons tenté d'introduire des règles plus détaillées, mais avons constaté que cette approche n'était pas souhaitable. Concernant les formes conjuguées des verbes, lorsqu'un mot se termine par certaines terminaisons verbales caractéristiques, il est étiqueté comme verbe conjugué (V). Ces terminaisons sont typiques de différents temps et modes de la conjugaison française. Par exemple :

Les terminaisons du présent de l'indicatif ou du subjonctif : -e, -es, -ent

Celles du futur simple : -ai, -a, -as, -ons, -ez, -ont

Celles de l'imparfait ou du conditionnel présent : -ais, -ait, -ions, -iez, -aient

Celles du passé simple ou du subjonctif imparfait : -is, -it, -issent, -ît

Celles du futur simple : -ra, -ras, -rez, -rons, -ront

Nous avons constaté que l'ajout excessif de règles morphologiques peut effectivement entraîner un surapprentissage ou des conflits entre règles. Par exemple, la terminaison « -ions » peut correspondre soit à un verbe au conditionnel (mot *parlerions*), soit à un nom au pluriel (mot *nations*). De même, la terminaison « -es » peut renvoyer à un déterminant (*les*) suivi d'un nom au pluriel (*tables*). Nous avons donc préféré limiter les règles susceptibles d'entrer en conflit, et éviter d'introduire un trop grand nombre de règles trop spécifiques.

Deuxièmement, même dans les étapes ultérieures où des règles peuvent être apprises automatiquement, pour l'étiquetage morpho-syntaxique, nous avons choisi de ne pas utiliser l'étiquette « UNK » (unknown). D'un point de vue linguistique, « UNK » ne constitue pas une catégorie grammaticale valide ; son utilisation correspond, en réalité, à une erreur d'étiquetage. En pratique, ce type de marquage temporaire finit toujours par deux issues possibles : soit il est considéré comme une erreur, soit il est corrigé par une règle lors de l'apprentissage ultérieur. Par conséquent, pour que les mots ne peuvent être étiquetés par des règles morphologiques explicites, nous avons adopté pour une solution simple : leur attribuer la catégorie la plus fréquente du corpus, à savoir le nom commun (NC). Au cours du processus, nous avons également calculé avec précision son nombre à 15684. De cette manière, un mot inédit dans le corpus d'entraînement conserve au moins une certaine probabilité d'être correctement annoté.

2.4 Multiples tentatives avec différentes méthodes algorithmiques

Au cours de l'élaboration et des expérimentations de l'algorithme, nous avons rencontré certaines difficultés. Concevoir un algorithme efficace tout en trouvant un bon compromis entre les performances du modèle et le nombre d'itérations constitue l'un des objectifs fondamentaux de notre projet.

Nous avons procédé à deux tentatives majeures de notre algorithme. Dans cette section, nous présentons brièvement les deux premières versions, en identifiant leurs problèmes. Ensuite, nous expliquons comment la troisième version du modèle permet de résoudre ces problèmes. Pour illustrer concrètement les différences entre ces versions, nous comparons leur charge de calcul.

Hypothèses de base :

- Nombre de templates disponibles : $T = 15$
- Taille du corpus : $\alpha = 10\,000$ mots
- Nombre de mots mal étiquetés : $\beta = 200$
- Nombre final de patches conservés : $P = 30$
- Nombre des étiquetages (POS tags) apparus dans le corpus : $E = 20$

2.4.1 Pour la première tentative (Problème identifié : coût computationnel élevé)

1. Parcours du corpus et détection d'un mot mal étiqueté (par exemple, *chat* étiqueté `V` au lieu de `NC`).
2. Génération des patches candidats à partir des 15 templates pour chaque parcours du corpus :
   - Par exemple, le template `prev_tag` donne une règle comme : *« Si le mot précédent est un `DET`, alors changer `V` en `NC` »*.
   - Chaque tag erroné peut produire jusqu'à 15 patches.
3. Évaluation de chaque patch candidat :
   - Utilisation de la fonction `_evaluate_patch` pour reparcourir les 10 000 mots du corpus.
   - Calcul du nombre d'erreurs corrigées et du nombre de nouvelles erreurs introduites.
4. Sélection du patch avec le meilleur score, puis passage au tag erroné suivant.
5. Application des patches apprises

Estimation de la charge de calcul :

- Charge estimée : $T \times \alpha \times \beta \times P + \alpha \times P$
- À chaque itération :
  - 200 tags erronés $\times$ 15 patches = 3 000 patches
  - Évaluation de chaque patch sur 10 000 mots $\Rightarrow 3\,000 \times 10\,000 = 30\,000\,000$ opérations

— Application d'un patch choisi sur 10 000 mots = 10 000 opérations

- Pour 30 itérations (donc 30 patchs conservés) :

$$\text{Charge totale} = 30 \times (30\,000\,000 + 10\,000) = 900\,300\,000 \text{ opérations}$$

- Soit environ **900 millions d'opérations**.

Nous observons qu'avec un corpus de grande taille, la cette méthode entraîne une vitesse de calcul insupportablement lente, en raison du nombre élevé de mots (α) et de mots mal étiquetés (β).

2.4.2 Pour la deuxième tentative (Problème identifié : patch non optimal appliqué à chaque itération)

Tout début nous parcourons seulement un fois du corpus, après avoir généré tous les patches candidats, une seule opération de tri a été effectuée sur la base des scores initiaux. Ensuite, les patches ont été appliqués successivement dans l'ordre décroissant. Après l'application de chaque patch, les scores des patchs restants étaient recalculés, **mais l'évaluation s'arrêtait dès qu'un patch ayant un score positif ($\geq$ 0) était trouvé**. Le problème principal vient donc du fait que l'algorithme ne sélectionnait **pas le meilleur patch global pour chaque application du patch**, mais se contentait d'appliquer le premier patch non négatif trouvé. Cette stratégie empêche une optimisation itérative efficace, car elle peut ignorer des patchs plus performants disponibles à ce moment-là.

$$\text{Charge estimée} = T \times \text{E} \times \alpha + \text{nb\_applications\_of\_patches} \times \alpha \ (\geq 30)$$

où **nb_applications_of_patches** : nombre d'applications de patches, au minimum 30, car les patchs de score négatif sont ignorés, mais le nombre attendu ne sera pas particulièrement élevé, car nous disposons déjà d'un tri initial de patches.

Bien que cet algorithme résolve le problème du coût de calcul excessif dès la première tentative, il ne garantit pas le choix du meilleur patch global pour chaque application de patch. Cet algorithme est incorrect.

2.4.3 Pour la troisième tentative (Version finale)

Voici la description synthétique de la logique algorithmique que nous avons finalement adoptée :

1. Recueillir l'ensemble des tags cibles possibles ($E = 20$ catégories).
2. Initialiser une liste de règles (patches).

3. Parcourir tous les mots du corpus ; pour chaque mot, tester les templates afin de détecter les contextes et générer des règles candidates.

$$15\text{templates} \times 20 \text{ catégories} \times 10\,000 \text{ mots} = 3\,000\,000 \text{ règles candidates}$$

(en supposant que chaque contexte est unique, bien qu'il y ait en réalité des répétitions).

4. Calculer le score de chaque règle en utilisant un dictionnaire avec des mises à jour simultanément (pas besoin d'itération supplémentaire du corpus), conserver celles ayant un score positif et sélectionner la meilleure à appliquer. L'apprentissage d'un patch optimal sur le corpus correspond à une itération (un parcours complet du corpus).
5. Mettre à jour les prédictions et répéter le processus jusqu'à ce que le nombre maximal de règles soit atteint.

$$\text{Charge estimée} = T \times \mathrm{E} \times \alpha \times P + \alpha \times P \ (\geq 30)$$

Globalement, cette méthode améliore significativement l'efficacité par rapport à la première tentative en évitant une charge de calcul trop importante. Bien qu'elle soit moins rapide que la deuxième approche, son avantage réside dans la capacité à apprendre et sélectionner précisément le patch optimal, garantissant ainsi la qualité du modèle. Cette méthode trouve un bon compromis entre performance et nombre d'itérations, assurant à la fois une bonne qualité d'apprentissage tout en limitant les coûts de calcul, ce qui lui confère une grande valeur pratique.

2.5 Analyse comparative des résultats : découpage aléatoire et séquentiel

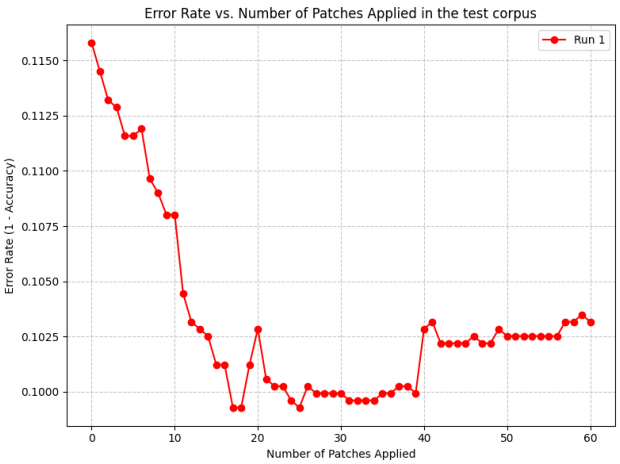

FIGURE 1 – Taux d'erreur (découpage séquentiel, 60 règles max)

La courbe présentée dans la FIGURE 1 illustre le taux d'erreur obtenu en découpant les données de manière séquentielle, avec un maximum de 60 règles d'apprentissage. Le graphique montre qu'à partir du 30e patch appris, le score atteint un niveau relativement bas et reste globalement stable autour de cette valeur. Cela indique que les performances du modèle se stabilisent à ce stade. Cependant, à partir du 40e patch, nous pouvons observer que une augmentation marquée du taux d'erreur. Cela suggère que les correctifs appris au-delà de ce point n'ont pas eu d'effet positif sur l'étiquetage, et ont même parfois entraîné des effets négatifs. Ce phénomène est caractéristique d'un surapprentissage, où le modèle commence à perdre en capacité de généralisation en s'ajustant excessivement aux données d'apprentissage.

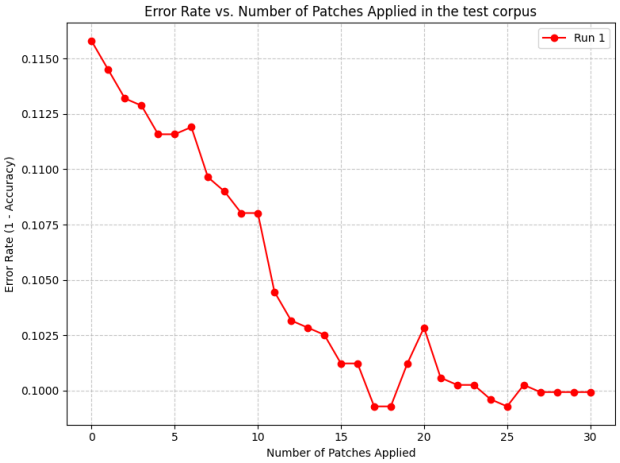

FIGURE 2 – Taux d'erreur (découpage séquentiel, 30 règles max)

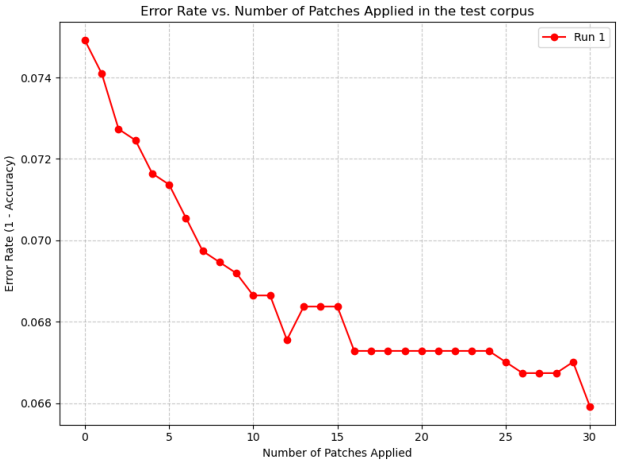

FIGURE 3 – Taux d'erreur (découpage aléatoire, seed = 42, 30 règles max)

Nous avons donc limité le nombre maximal de règles apprises à 30 et tracé la FIGURE 2. Les résultats montrent qu'à la 25e règle, le taux d'erreur est de 0.09929, et qu'à la 30e règle, il est de 0.09994. Toutefois, ces résultats sont obtenus à partir d'un corpus découpé de manière séquentielle.

Comme mentionné dans la première partie, ce mode de découpage présente certaines limites. Par conséquent, nous avons également effectué un découpage aléatoire du corpus, en fixant la graine aléatoire à 42 (FIGURE 3). Les résultats montrent qu'avec la même limite de 30 règles apprises, le modèle entraîné sur le corpus aléatoirement découpé de cette fois-ci obtient un taux d'erreur nettement inférieur à celui obtenu avec le découpage séquentiel.

2.6 Résultats du modèle pré-entraîné

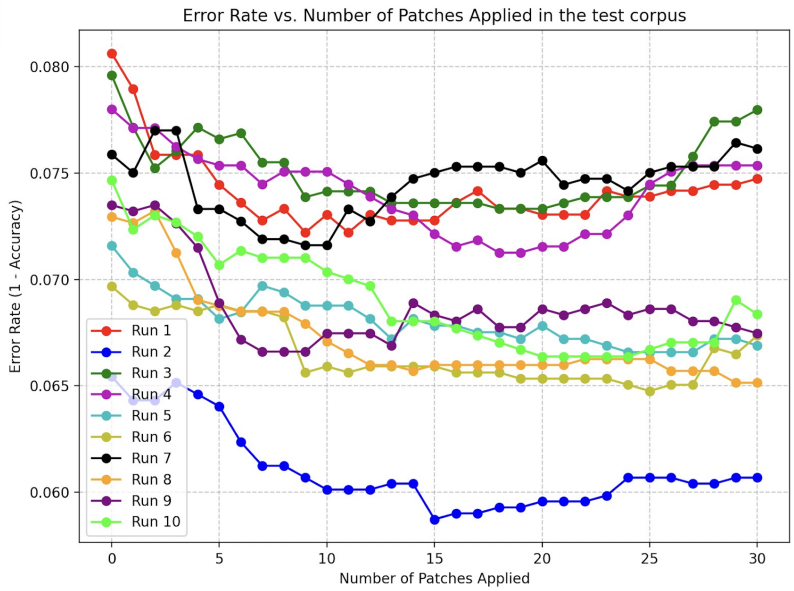

FIGURE 4 – Taux d'erreur (découpage aléatoire, 10 fois, 30 règles max)

Finalement, pour l'entraînement de notre modèle, nous avons adopté une approche consistant à effectuer dix découpages aléatoires du corpus. Parmi ces dix entraînements, nous avons sélectionné celui ayant obtenu le taux d'erreur le plus faible pour constituer notre modèle final. Sur la FIGURE 4, nous pouvons clairement observer que la courbe bleue présente un taux d'erreur nettement inférieur aux neuf autres. Le modèle correspondant atteint un taux d'erreur final de **93.93177%** sur la base d'un corpus de développement annoté d'environ 3000 mots. Cela montre que le découpage aléatoire permet non seulement d'atténuer les biais potentiels du découpage séquentiel, mais aussi d'améliorer les performances globales du modèle.

Les premiers cinq patches trouvés par notre modèle sont les suivant :

1. `IF PREVIOUS WORD TAG IS CLR : NC → V`
2. `IF PREVIOUS WORD TAG IS DET : DET → ADJ`
3. `IF PREVIOUS TWO TAGS ARE DET_NC : NC → ADJ`
4. `IF PREVIOUS WORD TAG IS V : NC → VPP`
5. `IF PREVIOUS TWO TAGS ARE P+D_NC : NC → ADJ`

3 Partie informatique

3.1 Organisation générale du code

L'ensemble du projet est organisé sous un seul répertoire intitulé `projet_brill_tagger/`, qui regroupe les fichiers sources, les données d'entraînement, les modèles générés ainsi que les fichiers de documentation. Cette organisation modulaire permet une lisibilité du code, une séparation claire des responsabilités, et facilite la maintenance du système.

Ce répertoire est constitué par les fichiers et dossiers suivants :

1) `interface.py` : C'est le point d'entrée principal du programme. Il propose une interface en ligne de commande permettant à l'utilisateur d'interagir avec le système. L'utilisateur peut ainsi entraîner un nouveau modèle, utiliser un modèle pré-entraîné pour annoter un texte, ou consulter l'aide intégrée.
2) `brill_tagger_trainer.py` : Ce fichier contient la logique d'apprentissage du système. Il regroupe plusieurs classes qui permettent de traiter le corpus, d'attribuer des étiquettes initiales aux mots, et d'apprendre des règles de transformation (patches) selon l'algorithme de Brill.
3) `sequoia-9.2.fine.brown` : Le corpus d'entraînement au format Brown.
4) `model_trained_results/` : Ce dossier contient les résultats d'un entraînement effectué par l'utilisateur. Nous pouvons y trouver :
   a) `best_patches.json` : la liste des règles apprises par le système.
   b) `most_common_pos_dict.json` : un dictionnaire associant à chaque mot l'étiquette la plus probable.
5) `pretrained_model/` : Ce dossier contient un modèle préentraîné fourni avec le projet, permettant de tester rapidement le système. Il inclut :
   a) `pretrained_pos_dict.json` : dictionnaire des étiquettes initiales.
   b) `pretrained_patches.json` : règles (patches) issues d'un apprentissage préalable.

6) `README.md` : Un fichier expliquant brièvement l'objectif du projet, la conficuration requise, les étapes d'installation et les instructions d'utilisation.
7) `rapport.pdf` : Le présent rapport, qui décrit en détail la description et motivation de tâche, les méthodes utilisées et les résultats expérimentaux, la partie informatique, et manuel utilisateur.

Pour que le projet reste clair et facile à gérer, chaque partie a son propre endroit : les données (corpus), les modèles entraînés (fichiers `.json`), le code source, et l'interface sont bien séparés.

Cela permet de garder les choses bien rangées, de ne pas tout mélanger, et surtout de pouvoir modifier un morceau sans casser le reste. Avec cette organisation, nous pouvons facilement tester, réutiliser ou améliorer une partie du projet.

Dans les sections suivantes, nous détaillerons le fonctionnement interne des deux fichiers principaux : `brill_tagger_trainer.py`, responsable de la logique d'apprentissage, et `interface.py`, chargé de la communication avec l'utilisateur.

3.2 Implementation du fichier *brill_ tagger_ trainer.py*

Ce fichier regroupe les éléments principaux du système d'apprentissage. On a choisi une organisation en classes (architecture orientée objet), pour être est plus clair à gérer et plus facile à faire évoluer. Chaque classe a son rôle bien défini, on peut les réutiliser ailleurs sans trop de modification, et ça simplifie les tests si on veut vérifier que tout fonctionne comme prévu. Nous avons quatre classe totalement : `Classe CorpusProcessor`, `Classe MostCommonPosBuilder`, `Class PatchLearner`, `Classe InitialTagger`.

Pour faire tourner tout ça, le fichier importe quelques bibliothèques bien connues de Python : `math`, `os`, `re`, `random`, `json`, `copy`, `Counter`, `defaultdict`, et aussi `matplotlib` pour tracer les courbes. Bref, tout ce qu'il faut pour traiter des corpus, gérer les données, et visualiser les résultats.

3.2.1 Classe *CorpusProcessor*

La classe `CorpusProcessor` sert à préparer les données linguistiques devant de l'apprentissage. En gros, elle gère la lecture du corpus annoté morphosyntaxiquement, le nettoyage des données, ainsi que la séparation du corpus en trois sous-ensembles : entraînement (90%), développement (5%) et test (5%). Cette classe repose principalement sur trois méthodes internes :

— `_load_corpus_lines()` lit le fichier ligne par ligne, enlève les lignes vides, et les stocke dans une liste.

— `_get_corpus_parts()` divise le corpus en trois sous-ensembles (entraînement, développement, test).
— `_corpus_without_pos()` enlève les étiquettes morphosyntaxiques des mots. Cette fonction utilise une regex du style (\S+)/ pour ne garder que la forme brute de chaque mot.

3.2.2 Classe *MostCommonPosBuilder*

Cette classe sert à construire un dictionnaire associant à chaque mot l'étiquette grammaticale la plus fréquente dans le corpus d'entraînement. Ce dictionnaire est utilisé pour une première étape d'étiquetage automatique, avant d'appliquer les règles apprises. Le code utilise une combinaison de `defaultdict` et de `Counter`, ce qui permet de compter facilement le nombre d'occurrences de chaque étiquette pour chaque mot. La méthode principale est :

— `_most_common_pos()` parcourt toutes les phrases du corpus. Pour chaque mot, elle sépare la forme et l'étiquette grâce à `rsplit('/', 1)`, puis met à jour le compteur associé au mot. À la fin, un dictionnaire simple `{mot : étiquette la plus fréquente}` est créé. Ce dictionnaire peut être facilement sauvegardé au format JSON pour être réutilisé ultérieurement.

3.2.3 Classe *InitialTagger*

Cette classe réalise l'étiquetage initial des phrases sans étiquettes morphosyntaxiques. Elle utilise un dictionnaire (`most_common_pos_dic`) construit auparavant pour attribuer à chaque mot son étiquette, et applique des règles simples pour les mots inconnus dans le dictionnaire.

Le fonctionnement principal s'appuie sur les méthodes suivantes :

— `__init__(most_common_pos_dic)` initialise l'objet avec un dictionnaire `{mot : pos}` issu du corpus d'entraînement.
— `_tag_sentence()` sert à attribuer une étiquette morphosyntaxique à chaque mot d'une phrase sans annotation. Elle prend une chaîne de caractères en entrée et est utilisée à l'intérieur de la méthode `_tag_corpus()`.

  Nous commençons par découper la phrase avec `split()`, ce qui permet de récupérer chaque mot séparément dans une liste.

  Il faut noter que l'argument d'entrée de la fonction `_tag_corpus()` provient du corpus transformé par la fonction `_corpus_without_pos()`, qui extrait uniquement les formes de surface à partir d'un corpus annoté au format Brown. Ainsi, les signes de ponctuation (étiquetés comme `PONCT`) sont séparés des mots et entourés d'espaces.

Par exemple, une phrase comme « `le/DET chat/N dort/V ./PONCT` » devient « `le chat dort .` » avec un espace entre « dort » et le point. Cela signifie qu'on peut utiliser simplement `split()` dans `_tag_sentence()` pour découper la phrase sans risque d'oublier une ponctuation.

Et même dans les cas particuliers comme « qu' » ou « c'est-à-dire », certains signes restent attachés aux mots dans le corpus annoté d'origine. Comme, dans le corpus annoté, ce type de ponctuation est déjà rattaché au mot auquel elle appartient (par exemple : « `L'/DET eau/NC` » devient « `l' eau` »), il n'est donc pas nécessaire de s'inquiéter : ne pas séparer ces ponctuations n'aura pas d'impact sur le résultat.

Ensuite, on parcourt chaque mot. Si le mot est présent dans `most_common_pos_dic`, on lui attribue son étiquette la plus fréquente ; sinon, on applique la méthode de secours `basic_morphosyntactic_rules()`.

À la fin, la méthode retourne deux listes parallèles : une avec les mots, l'autre avec les étiquettes correspondantes — ce qui est pratique pour la suite du traitement, car ça garde bien séparées les formes et leurs annotations.

— `_tag_corpus()` combine ces deux listes en une liste de tuples (mot, étiquette) pour chaque phrase, ce qui correspond à une structure `list[list[tuple[str, str]]]`. Cette structure est plus pratique pour les traitements ultérieurs, notamment pour l'application des règles d'apprentissage ou la mise à jour des annotations dans les phases suivantes du système. En effet, manipuler directement des tuples (mot, tag) au lieu de listes séparées simplifie la lecture, la modification et l'export des données annotées.

— `_basic_morphosyntactic_rules()` est une méthode de secours qui devine le POS d'un mot inconnu en se basant sur sa terminaison ou sa forme (ex. majuscule, chiffre, etc.).

Cette classe utilise uniquement des types simples (str, list, dict) sans dépendance externe, ce qui garantit que le code reste léger, facile à tester, et peut tourner partout. C'est simple à modifier ou à faire évoluer, par exemple : ajouter de nouvelles règles plus tard.

3.2.4 Classe *PatchLearner*

Cette classe est au cœur du système d'apprentissage des règles de transformation (patches) visant à corriger automatiquement les erreurs d'étiquetage morphosyntaxique. Elle identifie et apprend, parmi un ensemble de templates contextuels prédéfinis, des règles capables de modifier la catégorie grammaticale d'un mot pour améliorer la précision globale de l'étiquette.

Elle organise ce processus en plusieurs parties :

— Une liste de templates représentant différents contextes linguistiques (par exemple, le tag du mot précédent, la capitalisation du mot courant, la présence d'un tag spécifique autour, etc.).

— Un mécanisme qui applique ces templates à chaque mot du corpus pour générer des règles candidates.

— Une structure pour stocker ces règles (patches) et mesurer leur efficacité via un système de scoring.

Les méthodes détaillées sont les suivantes :

— `__init__()` prépare la liste des templates et initialise la liste vide des patches destinée à contenir les règles apprises durant l'entraînement.

Structure des templates Pour faciliter la gestion, la consultation et l'extension des templates, chacun d'eux a été conçu sous la forme d'un dictionnaire structuré autour de trois clés :

`name` : un identifiant court et explicite ;

`desc` : une brève description de la logique du template ;

`func` : une référence vers la fonction qui implémente la règle (ou patch) correspondante.

Les templates sont ensuite regroupés dans une liste, ce qui permet un accès simple et clair à leurs différents composants. Cette organisation améliore la lisibilité, la maintenabilité du code et facilite leur intégration dans les différentes étapes du processus d'apprentissage.

— `_prev_tag()`, `_next_tag()`, `_prev_two_tag()` etc. correspondant à chaque template, qui extraient le contexte linguistique utile pour la construction des règles(patches).

— `_parse_corpus()` convertit le corpus brut (liste de chaînes avec mots/étiquettes) en liste de listes de tuples (mot, étiquette). Cette séparation claire facilite la manipulation des données pour d'autres opérations.

— `_learn_patches()` est au cœur de l'apprentissage des règles correctrices. Concrètement, nous avons deux versions d'un même corpus sous type *list[list[tuple[str, str]]]* : l'une avec les étiquettes prédictives (`parsed_tagged`) et l'autre avec les étiquettes de référence correctes (`parsed_gold`). Ce qui suit décrit de manière synthétique le fonctionnement général du code d'apprentissage :

1. Parcourir `parsed_tagged` pour récupérer tous les tags possibles à utiliser comme `to_tag` par la suite.
2. Initialiser la situation de départ en créant une copie des prédictions actuelles (`parsed_tagged`) afin de pouvoir les corriger au fur et à mesure. Utiliser la liste `self.patches` pour stocker les règles apprises.

3. Entrer dans une boucle d'apprentissage. Tant que le nombre maximal de règles n'est pas atteint, continuer les itérations : comparer chaque phrase prédite avec sa version de référence ("gold"). Pour chaque mot, appliquer successivement tous les templates disponibles. Autrement dit, exécuter la fonction associée à chaque template à la position du mot afin de détecter un éventuel contexte pertinent. Si le contexte est jugé valide, le template est alors applicable à cette position. Dans ce cas, tester tous les tags possibles (`to_tag`) différents de l'étiquette actuelle, afin d'évaluer si un changement permettrait de corriger une erreur.
4. Pour chaque combinaison *tag actuel* → *nouveau tag*, une sorte de règle-candidate est crée, sous forme str `'template_name | from_tag -> to_tag | contexte'`. Son score est ensuite calculé.
5. Une fois l'ensemble du corpus parcouru, seules les règles ayant un score positif sont conservées. Les meilleures sont ensuite triées, et la première règle non utilisée est appliquée.
6. Appliquer la règle sélectionnée à l'ensemble des phrases à l'aide de la méthode `_apply_patch`, afin de mettre à jour les prédictions. Afficher l'amélioration en recalculant la précision à l'aide de la fonction `compute_accuracy_from_parsed()`.
7. Arrêter le processus lorsque plus aucun progrès n'est observé ou lorsque le nombre maximal de règles a été atteint.
8. Imprimer un bilan des patchs retenus à l'aide de la fonction `_print_patch_stats()`, et retourner la liste finale. Chaque patch est représenté sous forme de dictionnaire contenant les clés suivantes : `from_tag`, `to_tag`, `template` (le contexte déclencheur), `context_tag` (valeur spécifique du contexte), `score`, `corrected` (nombre d'erreurs corrigées) et `introduced` (nombre de nouvelles erreurs générées).

Ici, la manière de mémoriser efficacement les informations des patchs et leurs scores représente un défi dans le processus d'apprentissage. Pour cela, on conçu deux dictionnaires principaux pour réaliser l'apprentissage des patchs, à savoir `patch2score` et `patch_infos`.

Avant chaque itération sur le corpus, initialiser ces deux structures : `patch2score = defaultdict(lambda: 'score': 0, 'corrected': 0, 'introduced': 0)` pour enregistrer le score net et les effets de chaque patch candidat, et `patch_infos = {}` pour stocker les informations contextuelles nécessaires à l'identification de chaque patch.

Tout d'abord, `patch2score` sert à accumuler les performances de chaque patch lors de l'itération courante. La clé de ce dictionnaire est une chaîne unique identifiant le patch, construite en concaténant le nom du template, l'étiquette source, l'étiquette cible, et le contexte d'application. Sa valeur est un dictionnaire contenant plusieurs statistiques : le score net `'score'`, le nombre d'erreurs corrigées `'corrected'` et celui des erreurs introduites `'introduced'`. Cette organisation permet de mettre à jour rapidement, pour chaque patch possible, ses statistiques en parcourant les mots, et d'évaluer en temps réel son efficacité.

Ensuite, `patch_infos` stocke les informations détaillées sur chaque patch, telles que l'étiquette source `from_tag`, l'étiquette cible `to_tag`, la fonction template utilisée ainsi que le contexte `context_tag`. La clé est identique à celle de `patch2score`, ce qui assure une correspondance directe entre les données statistiques et les métadonnées du patch. Ainsi, lors de la sélection finale du meilleur patch, il est facile d'accéder à toutes ses informations sans recalculs ni recherches supplémentaires.

En bref, la procédure générale de la fonction consiste d'abord à comparer mot à mot les annotations prédites et de référence pour générer tous les patchs possibles, en mettant à jour en temps réel ces deux dictionnaires. Ensuite, la fonction filtre `patch2score` pour ne retenir que les patchs à score net positif, qu'on associe aux infos détaillées dans `patch_infos` pour constituer une liste de candidats. Et puis, la fonction entre dans une boucle `while`, appliquant à chaque tour le patch ayant le meilleur score. Cette application modifie la copie des annotations courantes, garantissant que chaque étape s'appuie sur les résultats actualisés. L'apprentissage s'arrête lorsque le nombre maximal de patchs est atteint ou qu'aucun nouveau patch bénéfique n'est trouvé.

— `_apply_patch()` applique un patch à un corpus déjà étiqueté, structuré sous la forme *list[list[tuple[str, str]]]*. Cette structure facilite le traitement du corpus de manière séquentielle, phrase par phrase, puis mot par mot. Le principe est de parcourir toutes les phrases et tous les mots du corpus, et lorsque le mot a pour étiquette `from_tag` du patch et que la condition contextuelle du template est satisfaite, remplacer cette étiquette par `to_tag`. La modification se fait directement dans le corpus, sans créer de copie. Cette méthode sert à corriger les erreurs selon une règle précise.

— `_apply_patches()` sert à appliquer tous les patches appris sur une copie indépendante du corpus initial. Nous avons adopté un `deepcopy` pour ne pas modifier les données originales, ce qui est important pour pouvoir les comparer ou les réutiliser plus tard. Ensuite, nous utilisons `_apply_patch` pour chaque patch stocké. Le résultat est un nouveau corpus corrigé avec toutes les règles appliquées.

— `_print_patch_stats()` : Cette fonction est pour vérifier l'efficacité de l'apprentissage des patches en affichant un bilan détaillé.

3.2.5 Fonctions utilitaires

Nous utilisons certaines fonctions utilitaires pour le traitement des phrases ou l'analyse des performances. Elles n'appartiennent pas à la classe principale, mais sont essentielles au fonctionnement de l'ensemble du code. Nous avons conçu des fonctions pratiques qui aident à calculer la précision, compter les tags, et lancer l'entraînement.

— `compute_accuracy_from_parsed()` :
qui permet d'évaluer la précision du système d'étiquetage en comparant deux corpus annotés : l'un prédit par *InitialTagger* (`predicted_parsed`), l'autre de référence (`gold_parsed`). Les deux corpus sont des listes de phrases, où chaque phrase est une

liste de paires (mot, étiquette). La fonction parcourt les deux corpus en parallèle, mot par mot, et compte le nombre d'étiquettes correctes lorsque les mots sont alignés. Elle retourne le ratio entre le nombre de tags corrects et le nombre total de comparaisons valides, ce qui donne le taux de précision final.

— `count_pos_tags()` : Elle compte la fréquence de chaque tag dans le corpus. Pour cela, elle exploite un Counter Python, parcourt chaque phrase et extrait les tags en séparant les mots des étiquettes.

— `train_from_corpus()` : Cette fonction coordonne tout le processus d'apprentissage en s'appuyant sur les méthodes de la classe responsable du chargement et du prétraitement du corpus :

  — construction d'un dictionnaire de tags les plus fréquents,
  — tagging initial,
  — apprentissage et application des règles correctrices (patches),
  — évaluation de la performance,
  — sauvegarde des meilleurs résultats.

  Elle supporte un ou cinq runs avec découpage aléatoire du corpus pour l'entraînement du modèle, et peut afficher une courbe d'erreur.

— `main()` : Point d'entrée simple qui demande à l'utilisateur le chemin du corpus et le mode de découpage, puis lance l'apprentissage.

3.3 Implementation du fichier *interface.py*

Ce fichier sert à utiliser un modèle Brill tagger déjà entraîné. Il s'occupe de charger le modèle, de restaurer les règles, et de faire l'étiquetage des phrases qu'on lui donne.

Pour faire fonctionner ce fichier, il faut importer`InitialTagger` et `PatchLearner` depuis `brill_tagger_trainer` pour l'annotation, ainsi que `json` pour la manipulation des fichiers JSON, `re` pour le traitement avec des expressions régulières, et `sys` pour gérer les entrées et sorties en ligne de commande.

Sur le plan technique, plusieurs choix de conception ont été faits pour répondre à des besoins spécifiques :

— `load_model_from_file()` : lit les fichiers JSON contenant le dictionnaire des étiquettes les plus fréquentes et les règles apprises afin de réutiliser un modèle sauvegardé.

— `add_func_to_patches()` : Comme les fonctions Python ne peuvent pas être sérialisées dans un fichier JSON, cette fonction permet de restaurer les liens entre chaque règle et la fonction de template correspondante, en s'appuyant sur le champ `'name'` conservé lors de l'enregistrement. Lors du chargement, les règles perdent leur logique fonctionnelle, car l'objet template n'est plus une fonction active. Pour y remédier,

nous utilisons un dictionnaire `name_to_template` qui associe chaque nom de template à sa fonction réelle. Chaque règle est ensuite mise à jour dynamiquement. ALors, nous pouvons reconstituer intégralement un modèle entraîné sans avoir à relancer l'apprentissage depuis le début.

— `tag_my_sentence()` : Cette méthode gère principalement la façon dont nous utilisons notre modèle pour annoter un texte saisi par l'utilisateur dans une interface interactive.

Classe `InitialTagger` et sa méthode `_tag_sentence()` reçoit des phrases déjà prétraitées, issues d'un corpus nettoyé avec *classe* `CorpusProcessor` `_corpus_without_pos()` présentée en *3.2.1.* Dans ce corpus, la ponctuation est très souvent annotée séparément.

Pour s'adapter à une saisie libre de l'utilisateur, la fonction commence donc par quelques traitements préliminaires sur la chaîne de caractères :

- Il est important de noter que dans le corpus *Brown* en français, tous les apostrophes (') sont généralement attachés au mot précédent lors de l'annotation. Cependant, lorsque l'utilisateur saisie un texte comme : *l'homme !*, il ne peut pas être segmenté correctement, sa forme est toujours `l'homme!` Donc, cette fonction peut automatiquement découper les mots, gérer les contractions et séparer la ponctuation à l'aide de la fonction `re.sub()` du module Python re : `re.sub(pattern, repl, string)`.
- Ces étapes permettent à l'utilisateur de saisir du texte librement, tout en assurant que la ponctuation soit correctement isolée avant l'annotation.

Ensuite, un tag par défaut est appliqué grâce à `InitialTagger`. `add_func_to_patches (learner, patches)` restaure ensuite les liens entre chaque patch et sa fonction template associée, en se basant sur le nom du template. Les règles apprises sont appliquées via la méthode `apply_patches()` de la classe `PatchLearner`, ce qui corrige progressivement les erreurs du premier étiquetage.

Enfin, le résultat annoté est converti en chaîne de caractères et renvoyé.

— `show_help()` : Afficher l'aide d'utilisation en ligne.

— `main()` : Le tout est encapsulé dans `main()`, la fonction `main()` gère en ligne de commande les entrées/sorties : chargement des fichiers et appel à l'étiquetage.

4 Discussion et conclusion

Dans ce projet, nous avons implémenté un Brill Tagger, un étiqueteur morpho-syntaxique fondé sur des règles transformationnelles. Sur la base d'un corpus de développement annoté d'environ 3000 mots, notre modèle a atteint une précision finale de plus d'environ 94%.

Nous avons constaté que l'erreur se stabilisait à un niveau relativement bas aux alentours du trentième patch.

Dans son article, Eric Brill rapporte une précision d'environ 95%. En comparaison, notre taux de précision est légèrement inférieur, ce qui pourrait s'expliquer par la complexité grammaticale plus élevée du français par rapport à l'anglais, ou encore par la taille plus réduite de notre corpus (environ 3000 mots dans le corpus de développement), par rapport au corpus Brown utilisé dans son travail (environ 60000 mots dans le corpus de développement).

Ce résultat montre néanmoins que les approches basées sur des règles peuvent rester compétitives lorsqu'elles sont bien entraînées et adaptées au corpus.

Pour aller plus loin, nous pourrions envisager d'intégrer un dictionnaire externe afin de compenser la taille limitée du corpus et de mieux gérer les cas out-of-domain, ainsi que d'élargir le corpus d'apprentissage pour améliorer la précision du modèle et obtenir des résultats plus fiables.

A Manuel Utilisateur

Le manuel utilisateur présenté dans ce rapport est identique de l'*aide en ligne* dans le fichier `interface.py`.

```
1  ================================================
2  AIDE - Brill Tagger
3  ================================================
4
5  Ce système permet d'étiqueter du texte en français avec un étiqueteur
       morphosyntaxique basé sur les règles (Brill Tagger) [1].
6  Vous pouvez soit utiliser un modèle pré-entraîné, soit entraîner le v
       ôtre avec un corpus compatible.
7
8  Fonctionnalités principales :
9  1. Utiliser le modèle pré-entraîné pour l'étiquetage
10 2. Entraîner un nouveau modèle avec un corpus français au format
       Brown, ou en utilisant le fichier fourni 'sequoia-9.2.fine.brown'
11 3. Afficher ce guide d'aide
12 4. Quitter le programme
13
14 FORMAT DU CORPUS (format Brown) :
15 - Une phrase par ligne
16 - Chaque mot est annoté sous la forme mot/POS
17   Exemple : Le/DET chat/NC mange/V la/DET souris/NC ./PONCT
18
19 ÉTAPES D'UTILISATION :
20 1. Téléchargez le dossier 'projet_brill_tagger' et lancez le
       programme avec : python interface.py
```

```
2. Choisissez une des options du menu
3. Pour l'entraînement :
   - Fournissez le chemin vers un corpus compatible
   - Le programme entraîne un modèle et affiche une courbe de pré
     cision (si matplotlib est installé)
   - Les fichiers du modèle sont enregistrés dans '
     model_trained_results/'
4. Pour l'étiquetage :
   - Entrez un texte (ex : Le chat dort.)
   - Le système affiche la phrase étiquetée (ex : Le/DET chat/NC dort
     /V ./PONCT)
   - Tapez 'back' pour revenir ou 'exit' pour quitter

TRAITEMENT AUTOMATIQUE :
Vous pouvez entrer du texte librement, le système s'occupe du
    traitement et du formatage automatique.

MODÈLE PRÉ-ENTRAÎNÉ :
Le répertoire 'pretrained_model/' contient un modèle basé sur le
    corpus Sequoia :
  - pretrained_pos_dict.json : dictionnaire morphosyntaxique
  - pretrained_patches.json : règles de transformation

MODÈLES ENREGISTRÉS APRÈS ENTRAÎNEMENT :
  - model_trained_results/most_common_pos_dict.json
  - model_trained_results/best_patches.json

DÉPENDANCES :
  - Python 3.12 ou version ultérieure
  - Bibliothèque 'matplotlib'
    À installer avec : pip install matplotlib

DOCUMENTATION :
  - Consultez 'rapport.pdf' pour des détails techniques sur l'
    algorithme et l'implémentation.

BIBLIOGRAPHIE :

[1] Brill, Eric. "A simple rule-based part of speech tagger." *Speech
    and Natural Language: Proceedings of a Workshop Held at Harriman
    *, New York, February 23-26, 1992. 1992.

==================================================
Tapez 'back' pour revenir au menu principal
==================================================
```

Références

[1] K. W. Church, "A stochastic parts program and noun phrase parser for unrestricted text," in *International Conference on Acoustics, Speech, and Signal Processing,*. IEEE, 1989, pp. 695–698.

[2] E. Brill, "A simple rule-based part of speech tagger," in *Speech and Natural Language : Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.

[3] M. Candito and D. Seddah, "Le corpus Sequoia : annotation syntaxique et exploitation pour l'adaptation d'analyseur par pont lexical," in *TALN 2012 - 19e conférence sur le Traitement Automatique des Langues Naturelles*, Grenoble, France, Jun. 2012.

[4] B. Guillaume, "Deep Sequoia corpus, version 9.2," GitLab, Oct. 2020. [Online]. Available : https://gitlab.inria.fr/sequoia/deep-sequoia/-/blob/master/tags/sequoia-9.2/README-distrib.md