Université de Paris

Automatiser l'écriture inclusive

Projet de traitement automatique du langage

Alice Hammel, Marjolaine Ray
M1 Linguistique Informatique, Université de Paris

16 juin 2021

Résumé

Nous présentons une nouvelle tâche de traitement automatique du langage – la conversion en écriture inclusive – ainsi que le développement d'un corpus et de modèles qui constituent une preuve de concept pour la résolution de cette tâche. Nos modèles (forêt aléatoire et SVM) obtiennent des résultats meilleurs que la *baseline* proposée pour résoudre cette tâche. Nous proposons des pistes d'améliorations pour résoudre cette tâche, notamment concernant le choix des caractéristiques à prendre en compte et les modèles à utiliser.

Table des matières

1 Théorie

1.1 Description de la tâche

L'objectif de ce projet était de réaliser un outil qui automatiserait la conversion de textes en écriture inclusive. L'écriture inclusive en français peut prendre de nombreuses formes. Son objectif principal est de ne pas avoir recours au genre masculin comme à un genre neutre. Ces formes d'écritures ont émergé dans les écrits militants, notamment féministes, puis la question a également été abordée par de nombreux articles de psycholinguistique qui ont montré que les expressions au masculin ne permettaient pas de faire référence à des femmes autant qu'à des hommes (voir [Burnett et Pozniak, 2020] pour un état de l'art en la matière). On remarque notamment qu'une lectrice a moins de chance de se représenter dans une situation si celle-ci n'est décrite qu'avec des acteurs masculins [Brauer, 2008]. La littérature sur le sujet insiste également sur l'utilisation de l'écriture inclusive comme d'un marqueur de l'identité politique des auteur·rices ou des organisations qu'ils/elles représentent ([Burnett et Pozniak, 2020]).

Les formes d'écriture inclusive qui nous intéressent ici sont celles qui combinent les flexions masculines et féminines d'un mot grâce à un séparateur (point médian, tiret, barre oblique ou majuscule au début de la deuxième flexion) lorsqu'un terme fait soit référence à un groupe mixte, composé d'hommes et de femmes, soit lorsqu'il a une référence générique (voir exemples dans la table 1). D'autres formes existent également, comme les doublets (*celles et ceux*, *agriculteurs et agricultrices*) et l'utilisation de termes épicènes, c'est-à-dire qui ont une forme unique pour désigner hommes et femmes (*les artistes, les personnes qui...*).

Cette pratique, qui reste très controversée, est toutefois adoptée par de plus en plus de personnes. Elle peut cependant être intimidante car elle est assez récente et de nombreuses

(1) Derrière chaque enquête se cachent des intérêts ; par exemple, la publication de certaines données sur les intentions de vote dans les périodes pré-électorales motive ou démotive les **votant-e-s** d'un parti ou d'un autre.

(2) En théorie, on attend **du/de** la journaliste qu'**il/elle** recherche les sources susceptibles de lui apporter l'information la plus abondante, désintéressée et diversifiée, et qu'**il/elle** recoure donc à la plus grande variété de sources.

(3) Merci à **touTEs celleux** avec qui, à un moment donné, j'ai partagé un atelier !

(4) Cette confiance va au-delà des partenaires ou **potentielLEs petitEs amiEs**.

(5) Quelle que soit votre expérience avant d'ouvrir ce livre, vous en sortirez plus **confiant·e** et **outillé·e**.

FIGURE 1 – Exemples de formes en écriture inclusive issues de notre corpus

pratiques différentes existent. Un assistant d'écriture inclusive qui permettrait d'aider l'auteur·rice à détecter les mots à écrire sous forme inclusive et à les transformer pourrait donc faciliter et encourager cette pratique. Une automatisation pourrait également rendre un ensemble d'écrits produits par une personne ou une organisation plus cohérent en appliquant les mêmes stratégies d'écriture inclusive.

Parmi nos objectifs, nous souhaitions composer un système qui, grâce à un entraînement, serait à même d'identifier les mots susceptibles d'être accordés aux deux genres. Cette tâche de traitement automatique du langage, qui revient à déterminer quels mots ont une référence générique ou mixte, nous paraît être une tâche originale, qui ne s'apparente pas à une tâche de TAL déjà résolue ou même explorée.

Secondairement, l'outil devait aussi pouvoir fournir des propositions de mots pour remplacer ceux identifiés comme sujets potentiels à l'écriture inclusive. L'utilisateur·rice pourrait utiliser la marque typographique de son choix : point médian, tiret, barre oblique, etc.

1.2 Approche

Nous souhaitions donc implémenter un classifieur binaire qui utilise des représentations vectorielles de mots. Nous avons tout d'abord cherché à trouver un corpus annoté qui contiendrait les types des références (générique/spécifique, mixtes), mais ce type de corpus ne semble pas exister, car cette information n'est pas utilisée par d'autres tâches TAL courantes. Nous avons donc décidé de créer notre propre corpus à partir de textes en écriture inclusive.

Nous avons associé à chaque terme de notre corpus un booléen indiquant si le mot était une forme inclusive ou non. A noter que les expressions épicènes ont donc été étiquettées comme *False*, car elles ne contenaient pas de marque typographiques d'écriture inclusive.

Pour créer un modèle capable de prédire cette étiquette, nous avons suivi la méthode suivante :

— Étape 1 : Recherche d'un corpus de textes contenant de l'écriture inclusive.
— Étape 2 : Création d'un corpus annoté indiquant si les mots sont ou non en écriture inclusive.
— Étape 3 :Création du corpus annoté sans formes inclusives destiné à l'apprentissage.
— Étape 4 : Implémentation et entraînement d'un ou plusieurs modèles sur le corpus de l'étape 3.
— Étape 5 : Évaluation des prédictions des modèles.

Une approche par apprentissage automatique nous a semblé pertinente pour cette tâche, car sa résolution est difficilement explicitable – quels indices linguistiques permettent de déterminer si un mot devrait être en écriture inclusive ? –, et donc ne paraît pas possible à implémenter via des règles.

2 Expérience et résultats

2.1 Création du corpus

Comme nous ne disposions pas de corpus adapté pour réaliser l'apprentissage pour cette tâche, nous avons entrepris de créer ce corpus nous-même. Pour cela nous avons récolté des textes écrits en écriture inclusive, principalement par *scraping*, puis nous avons détecté les formes en écriture inclusive grâce à l'utilisation d'expressions régulières, et enfin nous avons transformé ces formes en formes au masculin, afin de pouvoir effectuer l'apprentissage et l'évaluation sur ce corpus.

Cette tâche s'est avérée longue et difficile, mais elle nous a permis de mieux réaliser la difficulté et l'importance que représentent l'acquisition, le traitement et l'annotation de corpus.

2.1.1 Récupération des textes

Notre corpus se compose de cinq articles du magazine Causette, déjà analysés dans [Araujo, 2019], d'un roman (*Alana et l'enfant vampire*, de Cordélia) et 643 pamphlets publiés sur le site infokiosques.net. Ce dernier choix de source nous a été inspiré par [Abbou, 2017], ainsi que par la facilité à récupérer l'ensemble de ces textes en les *scrapant*, grâce à la régularité des URIs des pamphlets publiés sur le site, et du code HTML de chaque page. Notre corpus est donc fortement biaisé par cette source majoritaire composée de textes militants au vocabulaire bien spécifique, comprenant notamment un grand nombre de néologismes.

2.1.2 Détection des formes en écriture inclusive

Notre système de détection s'appuie principalement sur des expressions régulières, puis sur une relecture manuelle.

Nous avons tout d'abord découpé l'ensemble des documents en tokens. Cette première étape n'était pas anodine, car une tokenisation standard ne permet pas de conserver certaines formes en écriture inclusive intactes. Effectivement, par exemple le mot *étudiant-e-s*

serait découpé en trois tokens par un *tokenizer* classique. Nous avons donc créé notre propre fonction de tokenisation, qui n'effectue pas de séparation sur les tirets, points et barres obliques. Bien sûr cela a provoqué des difficultés, comme par exemple le fait que des inversions sujet-verbe dans des phrases interrogatives (*Peut-il*) n'étaient pas séparées, ce qui peut poser des problèmes de parsing. Il nous a toutefois semblé intéressant de remarquer les implications de l'écriture inclusive pour une tâche aussi simple que la tokenisation.

Chaque token a ensuite été classé comme forme inclusive ou non grâce à deux expressions régulières : une pour les formes du types *manifestantEs* avec une majuscule mais sans séparateur, et une pour formes en écriture inclusive qui contenait un séparateur typographique (tiret, point, point médian) avant un suffixe. A ces expressions régulières nous avons ajouté une liste de termes courants, en particuliers des pronoms, tels que *iels* et *toustes*, ainsi qu'une liste d'exceptions, qui nous permettait de ne pas étiqueter d'autres mots contenant des tirets (comme *trente-quatre, elle-même, vas-y*, etc.).

Afin d'éliminer une grande partie des mots composés qui représentaient un grand nombre de faux positifs, nous avons développé une fonction qui déterminé si un token contenant un séparateur typographique était entièrement composé de plusieurs mots présents dans le dictionnaire, et de le rejeter le cas échéant.

Nous avons ensuite effectué une relecture non-exhaustive, en faisant attention à ne pas seulement chercher des faux positifs, mais aussi des faux négatifs, et en corrigeant chaque erreur dans l'ensemble du corpus à chaque fois qu'elle était trouvée.

C'est ce système qui nous a permis de trouver, parmi les 1 387 textes récupérés sur infokiosques.net, les 643 qui contenaient effectivement des formes en écriture inclusive. Nous avons détecté dans ce corpus 14 197 formes en écriture inclusive (tokens de classe positive) et 4 134 506 de classe négative, contenues dans 147 232 phrases issues de 649 documents.

Nous n'avons pas effectué d'évaluation de cette partie du projet, car notre objectif n'était pas de mettre en place un système infaillible pour la détection automatique des formes en écriture inclusive, mais d'élaborer un corpus qui nous permettrait de réaliser l'apprentissage.

2.1.3 Suppression des formes en écriture inclusive

Une fois les formes en écriture inclusive détectées, nous avons créé un corpus équivalent sans formes en écriture inclusive en retrouvant la forme masculine équivalente. Pour cela nous avons retrouvé dans le dictionnaire le mot masculin le plus proche du token, ou d'une forme tronqué de ce token, selon les séparateurs typographiques. Pour cela nous avons créé une version filtrée d'un dictionnaire français [1] contenant une liste de mots ainsi que le genre et leur nombre afin de ne conserver que les noms, pronoms, adjectifs et participes passés masculins. Nous avons ensuite stocké ces mots dans une trie, une structure de données qui permet de rapidement récupérer une liste de mots à partir d'un préfixe (voir figure 2). En effet si nous avions utilisé une liste, il aurait été très coûteux de trouver tous les mots

1. source : https://github.com/hbenbel/French-Dictionary

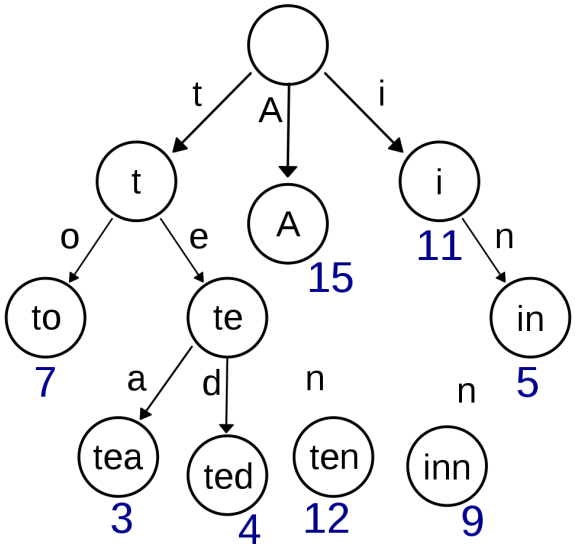

FIGURE 2 – Représentation d'une trie contenant les mots *A,i, in, inn, to, tea, ted, ten.* source : Wikipedia

du dictionnaire qui commencent par le même préfixe que le token, puisqu'il aurait fallu parcourir toute la liste, jusqu'à trouver ces mots.

Une fois que les mots commençant par le même préfixe (de taille 4) que le token sont récupéré, l'algorithme calcule la distance de Levensthein – c'est-à-dire le nombre de modifications nécessaires pour passer du token au mot – entre chaque mot et le token. Le mot le plus proche est retenu comme la version "désinclusifiée" du token.

Cette étape n'était finalement pas nécessaire, car dans la majorité des cas il suffisait de tronquer le token au niveau du séparateur, et d'ajouter le -s final au radical, s'il y en avait un. Par exemple le token *étranger-e-s* devient *étranger* puis *étrangers*. C'est d'ailleurs cette approche que nous avons suivi lorsqu'aucun terme suffisamment proche ne figurait dans le dictionnaire.

Là encore, nous avons effectué une relecture manuelle du corpus en suivant la démarche décrite précédent pour nous assurer d'éliminer le plus d'erreurs possibles. Le résultat de ces deux étapes a ensuite été stocké dans un fichier texte suivant une structure inspiré du format CONLL (voir figure 3).

2.2 Extraction des caractéristiques (features)

Nous avons choisi d'utiliser comme features les mots et leurs catégories morphosyntaxiques, ainsi que celles des mots qui les entourent. Nous avons fait ce choix car ces caractéristiques nous paraissaient à la fois pertinentes pour résoudre notre tâche (tout en étant limitées) et faciles à obtenir. D'autres caractéristiques, telles que l'animéité et les chaînes de coréférence étaient beaucoup plus difficilement accessibles.

Nous avons utilisé la bibliothèque Python de traitement automatique du langage spaCy [Honnibal et Montani, 2017] afin d'extraire pour chaque phrase "désinclusifiée", les caté-

```
# sent_id = 248
# text = Combien de fois montre-t-on l'ertzaina qui charge contre
des manifestant-e-s ?
0       Combien       False       Combien
1       de       False       de
2       fois       False       fois
3       montre-t-on       False       montre-t-on
4       l       False       l
5       '       False       '
6       ertzaina       False       ertzaina
7       qui       False       qui
8       charge       False       charge
9       contre       False       contre
10       des       False       des
11       manifestant-e-s       True       manifestants
12       ?       False       ?
# text_no_ei =  Combien de fois montre-t-on l ' ertzaina qui charge
contre des manifestants ?
```

FIGURE 3 – Exemple d'une phrase dont les tokens sont annotés et désinclusifiés

gories morphosyntaxiques et les caractéristique de genre, de nombre, etc. de chaque mot. Nous avons exporté l'ensemble de ces informations au format CONLL-U, ainsi que la version originale en écriture inclusive le cas échéant, dans la colonne "misc" (voir figure 4). Pour chaque mot nous disposons donc de sa catégorie (nom, adjectif, verbe, etc.) mais aussi de caractéristiques plus fines, en fonction de cette catégorie : genre, nombre, mode, temps, etc.

Plutôt que d'utiliser le tokenizer de spaCy2, nous avons réutilisé la tokenisation déjà effectuée en découpant sur les espaces. Cela cause des erreurs d'étiquetage, qui limitent sûrement la précision du modèle (voir figure 4, token 3).

Nous avons décidé d'utiliser uniquement les informations de la colonne xpos, c'est-à-dire les étiquettes spécifiques à la langue française, car elles comportaient les informations qui semblaient être les plus pertinentes. Nous n'avons pas utilisé l'analyse en dépendance, mais cela pourrait être une piste d'amélioration.

```
# sent_id = 248
# text_no_ei =  Combien de fois montre-t-on l ' ertzaina qui charge contre des manifestants ?
 id          form        lemma upostag                                                    xpostag feats  head     deprel deps misc
  0       Combien      combien     ADV                                        ADV__PronType=Int     _     4     advmod    _    _
  1            de           de     ADP                                                    ADP     _     4       case    _    _
  2          fois         fois    NOUN                          NOUN__Gender=Fem|Number=Sing     _     4     nummod    _    _
  3   montre-t-on  montre-t-on     ADV                                                    ADV     _     6       amod    _    _
  4             l            l    NOUN                                                   NOUN     _     6       amod    _    _
  5             '            '   PROPN                                                  PROPN     _     0       ROOT    _    _
  6      ertzaina     ertzaina   PROPN                                                  PROPN     _     6  flat:name    _    _
  7           qui          qui    PRON                                        PRON__PronType=Rel     _     9      nsubj    _    _
  8        charge      charger    VERB  VERB__Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin     _     6  acl:relcl    _    _
  9        contre       contre     ADP                                                    ADP     _    12       case    _    _
 10           des           un     DET                   DET__Definite=Ind|Number=Plur|PronType=Art     _    12        det    _    _
 11  manifestants  manifestant    NOUN                                NOUN__Gender=Masc|Number=Plur     _     9    obl:arg    _    ei=manifestant-e-s
 12             ?            ?   PUNCT                                                  PUNCT     _     6      punct    _    _
```

FIGURE 4 – Extrait du corpus après annotation morphosyntaxique

2.3 Création et vectorisation des exemples

Nous avons fait le choix de travailler avec une fenêtre de taille 2, mais bien sûr cela peut-être changé. Pour chaque token de notre corpus, nous avons généré un exemple contenant le token et ses informations morphosyntaxiques, les tokens contenus dans la fenêtre (2 à droite et 2 à gauche), et leurs informations morphosyntaxiques, ainsi que l'étiquette ("gold label").

Nous avons ensuite encodé ces exemples en utilisant un encodage Bag-of-Words, à l'aide de deux objets CountVectorizer [2] : un pour encoder les mots et un pour encoder les caractéristiques morphosyntaxiques. Chaque exemple se compose ensuite d'une concaténation de 4 vecteurs creux BoW :

1. Un vecteur one-hot de taille 106 931 (taille du vocabulaire) encodant le token.
2. Un vecteur BoW de taille 92 encodant les caractéristiques morphosyntaxiques du token.
3. Un vecteur BoW de de taille 106 931 (taille du vocabulaire) encodant les tokens du contexte.
4. Un vecteur BoW de taille 92 encodant les caractéristiques morphosyntaxiques des tokens du contexte.

Ces "vectoriseurs" ont été enregistrés afin d'être réutilisé lorsqu'un·e utilisateur·rice veut effectuer une prédiction et conversion d'une nouvelle phrase.

Notre jeu de donnée contenait une classe minoritaire ce qui induisait un risque de biais dans l'apprentissage, ainsi qu'un temps d'apprentissage très long. Nous avons donc choisi de sélectionner aléatoirement un certains nombres d'exemples de classe négative, après vectorisation. Après quelques essais nous avons choisi d'utiliser un corpus contenant dix fois plus d'exemples négatifs que positifs, ce qui bien sûr ne reflète pas la fréquence des formes en écriture inclusive dans un texte. Cette approche a le défaut de sélectionner des exemples après la création des vecteurs, menant ainsi à l'existence de nombreuses "colonnes" inutiles dans les vecteurs encodant les tokens.

Une autre piste d'amélioration de cette étape aurait été la prise en compte des mots inconnus. Actuellement un mot inconnu ne sera tout simplement pas encodé, alors que si nous avions introduit un token inconnu (`UNK`) dans notre corpus en remplaçant une partie des hapax par ce token, nous aurions pu intégrer une meilleure prise en compte des tokens inconnus.

2.4 Apprentissage

2.4.1 Modèle forêt aléatoire

Présentation du modèle Nous avons choisi d'expérimenter un modèle de forêt aléatoire car ce modèle est relativement simple à paramétrer et interpréter, et offre rapidement des résultats satisfaisants.

2. sklearn.feature_extraction.text.CountVectorizer

Les forêts aléatoires sont des ensemble d'arbres de décisions entraînés sur des sous-ensembles des données d'apprentissages et des caractéristiques. Un arbre de décision est un classifieur qui prend la forme d'un arbre binaire et qui prédit la classe d'un objet en appliquant successivement des structures **if - else** aux caractéristiques de cet objet. Ces décisions successives permettent d'aboutir à une feuille qui correspond à la classe prédite pour cet objet.

L'apprentissage d'un arbre de décision se fait au travers d'un algorithme glouton qui détermine à chaque étape quelle caractéristique n'ayant pas encore été utilisée dans l'arbre et quel seuil (si c'est une caractéristique numérique, ce qui est notre cas) "séparent" le mieux l'ensemble d'apprentissage. Pour cela nous avons choisi d'utiliser la mesure d'impureté de Gini, qui correspond à la probabilité de mal classer un objet de l'échantillon en s'appuyant sur cette caractéristique. Si on s'intéresse au noeud racine d'un arbre de décision issu de notre forêt aléatoire (voir figure 6), on voit que si on sépare notre échantillon en deux catégories : ceux qui ont moins de 0.5 dans la caractéristique "voulaient" [3] et ceux qui ont plus de 0.5 dans cette catégorie, nous nous tromperions en moyenne 50% du temps (puisque le score de Gini est de 0.5). Quand le score de Gini vaut 0, seule une classe est possible et on est donc dans une feuille de l'arbre de décision.

Un arbre de décision complet serait très long à apprendre sur l'ensemble de nos caractéristiques, puisque nous en avons plusieurs centaines de milliers, de plus la rigidité de cette structure ne permettrait pas d'avoir une bonne capacité de généralisation. La forêt aléatoire ajoute de la flexibilité à ce modèle et donc de la capacité de généralisation en entraînant plusieurs arbres de décision sur des sous-ensembles des données, et ne prenant pas en considération l'ensemble des caractéristique dans chaque arbre. La prédiction pour chaque objet est alors effectuée dans l'ensemble des arbres de la forêt, et la prédiction finale correspond à la classe qui a été prédite par le plus d'arbres. C'est une approche ensembliste, car nous utilisons un ensemble d'estimateurs pour effectuer la prédiction.

Recherche des hyperparamètres Les hyperparamètres principaux à choisir pour optimiser la forêt aléatoire sont donc le nombre d'arbres (`n_estimators`) et leur profondeur (`max_depth`). Nous avons donc effectué une recherche d'hyperparamètres en grille (*grid-search*) afin de déterminer les hyperparamètres optimaux. Cette recherche en grille a été effectuée sur l'ensemble d'apprentissage, via une validation croisée à 5 blocs : à chaque entraînement, le score f1 (voir 2.5.1) de la forêt était évalué sur un cinquième de l'ensemble d'apprentissage qui n'avait pas été utilisé pour l'apprentissage. Le score final de chaque modèle était la moyenne de ces cinq scores f1. Pour `max_depth` nous avons évalué les valeurs 10, 15, 20 et 30 et pour `n_estimators` 50, 75, 100, 150, 175 et 200. Nous avons finalement retenu les valeurs `max_depth = 30` et `n_estimators = 150`, qui permettaient d'obtenir le meilleur score f1 moyen, 0.5859.

3. Malheureusement nous n'avons pas indiqué dans cette représentation s'il s'agit de la partie du vecteur qui concerne le mot lui-même ou son contexte.

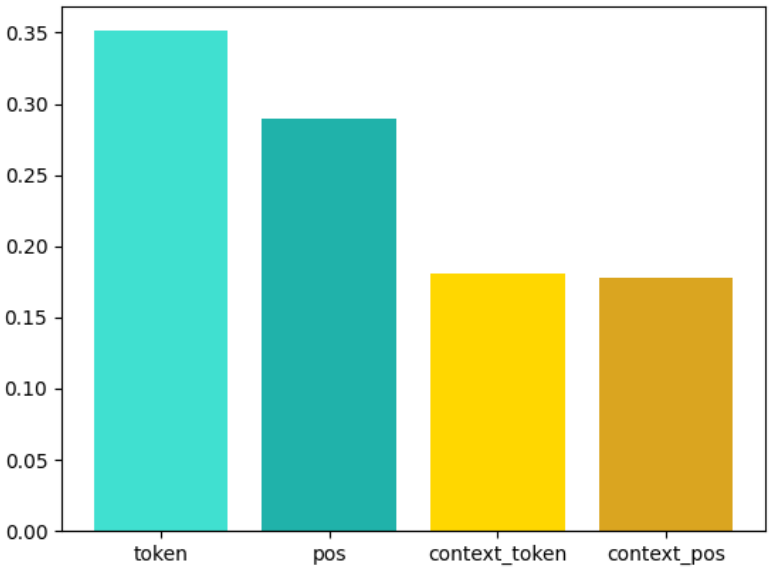

FIGURE 5 – Importance de chaque classe de features

Importance des caractéristiques Un avantage de ce type de modèle est qu'il permet de connaître l'importance de chaque caractéristique dans l'apprentissage, et peut donc être très utile pour améliorer le choix de ces caractéristiques. La figure 5 montre l'importance relative de chaque classe de caractéristiques. On remarque que la caractéristique la plus importante est le mot lui-même, puis ses caractéristiques morphosyntaxiques, tandis que le contexte a moins d'importance pour la prédiction. Le modèle s'appuie donc beaucoup sur les mots eux-même pour effectuer ses prédictions, alors que nous aurions aimé qu'il utilise le contexte pour gagner en précision et en généralisation.

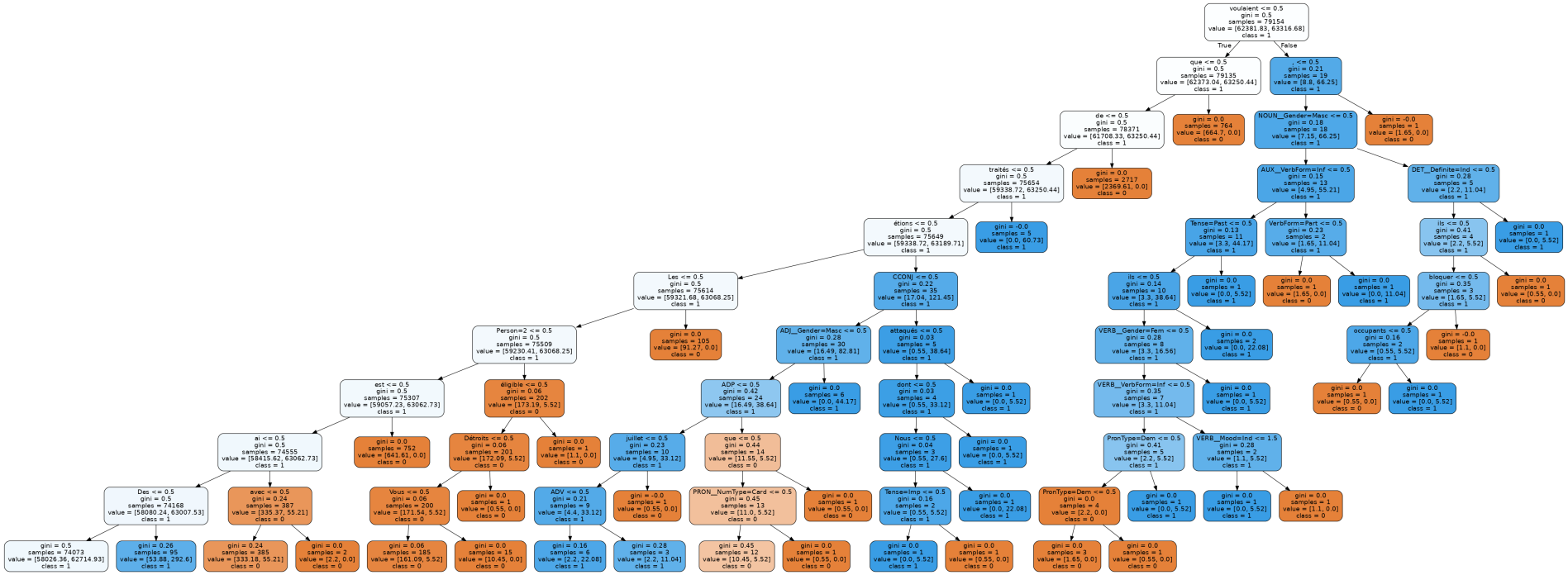

FIGURE 6 – Visualisation d'un des arbres de décision de la forêt aléatoire

2.4.2 Modèle Support Vector Machine

Présentation du modèle Le Support Vector Machine (SVM) permet d'optimiser les paramètres du modèle jusqu'à déterminer un hyperplan séparateur avec la marge la plus élevée possible entre l'hyperplan et les données les plus proches de celui-ci (les vecteurs supports). Néanmoins, lorsqu'il est utilisé dans sa version d'origine (hard-margin ou soft-margin linéaire), il est peu efficace si les données ne sont pas linéairement séparables. Pour améliorer ses performances, le SVM est généralement implémenté avec un noyau non-linéaire, dont on choisit le type : polynomial, linéaire, avec fonction de base radiale (désormais RBF) ou avec fonction sigmoïde. le noyau permet d'augmenter la dimensionalité, et donc l'expressivité des données et la probabilité qu'elles soient séparables, de manière très peu couteuse computationnellement.

Recherche des hyperparamètres Les hyperparamètres du modèle SVM sont les suivants :

— Le choix du type de noyau (parmi les plus utilisés : RBF, polynomiale, linéaire et sigmoïde).
— Le nombre d'itérations de l'entraînement.
— Le paramètre `C`
— Le paramètre `gamma`

Notre SVM utilise un noyau RBF avec 10000 itérations.

Le paramètre `gamma` est le coefficient appliqué à la fonction du noyau. Ici, pour la RBF, il définit quelle sera l'influence du rayon d'un seul exemple sur l'ensemble du calcul. C'est finalement l'inverse du rayon d'influence des échantillons sélectionnés par le modèle comme vecteur de support. Nous avons utilisé la valeur par défaut de `gamma`, soit `gamma = 1 / (n_features * X.var())`.

Le paramètre `C` manipule la maximisation de la marge de la fonction de décision contre le niveau d'exemples d'entraînement correctement classifiés. Pour une plus grande valeur de `C`, une marge plus petite sera accepté en échange d'une meilleure classification alors qu'une faible valeur de `C` encouragera une grande marge, au prix de l'exactitude de la classification, mais en offrant calcul moins lourd. Nous avons utilisé la valeur par défaut de `C`, soit `C = 1`.

Le nombre d'itérations a été porté à 100, 1000 puis 10000 sans qu'il y ait de convergence vers un hyperplan séparateur stable. C'est finalement à partir de 10000 itérations que les évaluations du modèle sont devenues prometteuses. Nous avons aussi essayé plusieurs types de normalisations proposées par Scikit (StandardScaler, MinMaxScaler, MaxAbsScaler) sans résoudre le problème de convergence du SVM. En utilisant GridSearchCV, nous avons testé plusieurs types de kernels pour le SVM, en espérant qu'une des fonctions utilisées pourrait converger vers un hyperplan séparateur (Radial Basis Function, Polynomial et Sigmoid). Cela n'a pas abouti à une convergence mais l'augmentation du nombre d'itération suffisait finalement à améliorer le modèle.

Notre SVC actuel a été entraîné avec les paramètres suivants :

— Un StandardScaler appliqué aux données

— Un noyau RBF
— 10000 itérations
— `C` = 1
— `gamma = scale`, soit `gamma = 1 / (n_features * X.var())`

2.5 Evaluation

2.5.1 Métriques utilisées

Accuracy (exactitude) L'*accuracy* est le rapport entre le nombre de prédiction correctes et le nombre d'objets à prédire. Dans le cas d'une classe minoritaire celle-ci n'est pas très informative, comme nous avons 10 fois plus d'exemples de classe négative que de classe positive, classer tous les exemples dans la classe positive permettrait d'avoir une très bonne exactitude, mais ne résoudrait pas la tâche. C'est pour cela que nous nous sommes davantage intéressé au rappel et à la précision.

Rappel Le rappel (*recall*) permet de répondre à la question : combien des exemples de classes positives ont-ils été bien classés ? Il s'agit du rapport entre le nombre d'exemples bien classifiés en classe positive/minoritaire, et le nombre d'exemples étiquetés en classe positive/minoritaire. Il permet d'évaluer le silence, c'est-à-dire la quantité de faux négatifs.

Précision La précision est le rapport entre le nombre d'exemples bien classés dans la classe minoritaire, et le nombre d'exemple classés en classe minoritaire. Cette métrique nous permet donc de répondre à la question : trop d'exemples sont-ils classés en classe minoritaire ? Y a-t-il beaucoup de faux positifs ?

Score f1 Le score f1 est une métrique qui combine le rappel et la précision, et permet donc de comparer deux modèles, notamment dans le cadre de la recherche d'hyperparamètres (voir 2.4.1).

Matrice de confusion La matrice permet de voir le nombre d'exemples correctement classés dans chaque classe (dans la diagonale), ainsi que le nombre de faux positifs et de faux négatifs. La table 1 offre une clé de lecture pour interpréter ces matrices de confusion.

		prédiction	
		0	1
gold label	0	vrais négatifs	faux positifs
	1	faux négatifs	vrais positifs

TABLE 1 – Clé de lecture des matrices de confusion binaires

2.5.2 Baseline

Nous avons développé un modèle naïf de résolution de la tâche afin de s'en servir comme point de comparaison pour nos modèles d'apprentissage automatique. Ce modèle

naïf consiste à prédire comme étant de classe positive l'ensemble des noms, pronoms, adjectifs et participes passés. Les résultats obtenus par cette *baseline* sont présentés dans la figure 7.

```
ACCURACY : 0.654959339181661
RECALL :  0.8816793893129771
PRECISION :  0.19580796794328428
F1 SCORE :  0.3204489564285264
[[17916 10436]
 [  341  2541]]
```

FIGURE 7 – Métriques d'évaluation de la *baseline*

Sans surprise le nombre de faux positifs est extrêmement élevé, et donc que la précision est très basse, puisque la grande majorité des mots de ces catégories n'étaient pas en écriture inclusive dans notre corpus. Il est intéressant de s'assurer que nos modèles ne s'appuieront pas uniquement sur la catégorie grammaticale, qui leur est fournie, pour effectuer leurs prédictions. En revanche le rappel est élevé puisque la grande majorité des formes en écriture inclusive appartiennent à ces catégories.

2.5.3 Forêt aléatoire

Les métriques d'évaluation de notre forêt aléatoire se trouvent dans la figure 8. Le rappel est assez similaire à celui de la baseline, mais la précision est bien meilleure. En effet notre modèle réduit nettement le nombre de faux positifs par rapport à notre baseline.

```
ACCURACY : 0.8875264135237242
RECALL :  0.8358778625954199
PRECISION :  0.44209946779225545
F1 SCORE :  0.5783219301404392
[[25312  3040]
 [  473  2409]]
```

FIGURE 8 – Métriques d'évaluation de la forêt aléatoire (`max_depth = 30`, `n_estimators = 150`)

Une rapide analyse qualitative des faux positifs révèle qu'un certain nombre de ces faux positifs sont en fait des prédiction tout à fait pertinentes, mais portent sur des termes épicènes, ou qui auraient pu être en écriture inclusive mais ne le sont pas dans le corpus (exemples 6, 7 et 8).[4]. De manière plus pragmatique, un grand nombre de ces erreurs ne serait tout simplement pas visible dans notre tâche finale, qui est de créer une forme qui contient la marque du féminin et du masculin, puisqu'il s'agit de mots qui n'ont pas de

4. Vous pouvez consulter l'ensemble des faux positifs sur l'ensemble d'évaluation dans le fichier `RandomForestClassifier_V2_false_positives.txt`

formes féminines ou masculines (exemples 8 et 9). Cela nécessite toutefois l'utilisation d'un dictionnaire au moment de la transformation des mots (voir partie 2.6).

(6) En plus , ils ont essayé à plusieurs reprises de rencontrer le préfet , **ils** ont obtenu des rendez-vous avec les mairies de Bagnolet et Montreuil , et ils ont pris la parole et/ou perturbé de nombreux conseils municipaux .

(7) C ' est exactement ce que se passe chez les **transsexuels** .

(8) Le mouvement divise encore au sein des syndicats , d ' autant que les **Gilets** jaunes n ' interviennent que très peu dans le monde du travail et n ' appellent que peu à la grève .

(9) Je n ' ai plus le délire , je ne parle plus en métaphores absurdes , je n ' ai plus de **sentiments** .

FIGURE 9 – Exemples de faux positifs du corpus d'évaluation avec le modèle forêt aléatoire (en gras)

2.5.4 Support Vector Machine

Les métriques d'évaluation du classifieur par SVM sont présentées en Figure 10. Le classifieur obtenu à la dix-millième itération avec un noyau RBF montre une exactitude (*accuracy*) très élevée par rapport à la *baseline* et une matrice de confusion très favorable où le nombre de faux positifs et de faux négatifs est faible. Le rappel est plus faible qu'avec le modèle de RandomForest, donc le SVC crée plus de silence que la RandomForest pour ces données. En revanche la précision est beaucoup plus élevée que pour la RandomForest, il y a donc moins de faux positifs.

```
ACCURACY : 0.9563616571684702
RECALL :  0.7081887578070785
PRECISION :  0.796332422941865
F1 SCORE :  0.7496786042240589
[[27830   522]
 [  841  2041]]
```

FIGURE 10 – Métriques d'évaluation du Support Vector Machine (`max_iter = 10000`, `kernel = RBF`, `C = 1`, `gamma = scale`)

De la même manière que pour le classifieur RandomForest, une rapide analyse qualitative nous montre que certains faux positifs sont des items qui ont été correctement prédits mais que le corpus n'a pas annoté en tant qu'écriture inclusive (à cause d'une erreur d'annotation, parce que le texte d'origine n'était pas en écriture inclusive ou que le mot est épicène, cf. les exemples présentés en Figure 9). De la même manière, une partie des faux négatifs sont correctement prédits mais l'annotation du corpus les considère comme de

l'écriture inclusive, par erreur (souvent à cause d'une expression contenant un séparateur aussi utilisé en écriture inclusive).

2.6 Transformation des formes en écriture inclusive

Notre implémentation de la transformation des formes prédites dans la classe positive en écriture inclusive est restée assez simple par manque de temps. L'idée principale est d'utiliser un lexique ([New *et al.*, 2004], la version utilisée est Lexique 3.83) afin de fusionner chaque forme au masculin avec la forme au féminin ayant le même lemme et le même nombre en concaténant la forme masculine (sans la marque du pluriel le cas échéant), le séparateur (un point médian par défaut) et le suffixe féminin. Ce suffixe est déterminé en comparant la forme masculine et la forme féminine pour trouver ce qu'elles ont en commun. On aurait transformerait donc par exemples *agriculteurs* en *agriculteur·rices*.

Notre système s'appuie donc à la fois sur spaCy pour correctement reconnaître le genre et le nombre, ainsi que sur Lexique, et rencontre des difficultés si l'étiquetage comporte des erreurs, ou que les mots ne sont pas présents dans le lexique. Le but serait d'ajouter des règles pour traiter les mots qui ne sont pas présents dans le lexique en nous appuyant notamment sur [Lessard et Zaccour, 2018], qui déduiraient la forme féminine à partir des suffixes des formes masculines. Malheureusement nous n'avons eu le temps de les développer.

3 Développement

3.1 Organisation du code

Nos programmes, développés en Python 3, ont recours à plusieurs bibliothèques Python – `scikit-learn`, `conllu`, `numpy`, `pandas`, `spacy`, `spacy-conll`... – dont les versions sont détaillées dans le fichier `requirements.txt`.

3.1.1 Création du corpus annoté

1. La première étape d'annotation est effectuée par le programme `EIannotator.py`, qui lit l'ensemble des fichiers `.txt` contenu dans le dossier du corpus, les tokenize, détecte les formes en écriture inclusive grâce à `findEI.py` et exporte les résultats dans un fichier dans un format inspiré du format CONLL (`1_corpus_annote_ei.conll`).
2. Ensuite `desinclusify.py` lit ce fichier, transforme chaque token de classe positive, recrée une phrase à partir des tokens sans écriture inclusive et exporte le résultat dans un nouveau fichier (`2.corpus_no_ei.conll`, voir figure 3). Ce programme utilise les fictionnaires contenus dans le dossier `French-Dictionary-master` afin de générer les formes non inclusives.
3. Enfin `extract_features_spacy.py` lit le fichier issu de l'étape précédente, extrait les catégories morphosyntaxiques et exporte ces données au format CONLL-U (`3.1.corpus_spacied.conll`, voir figure 4).

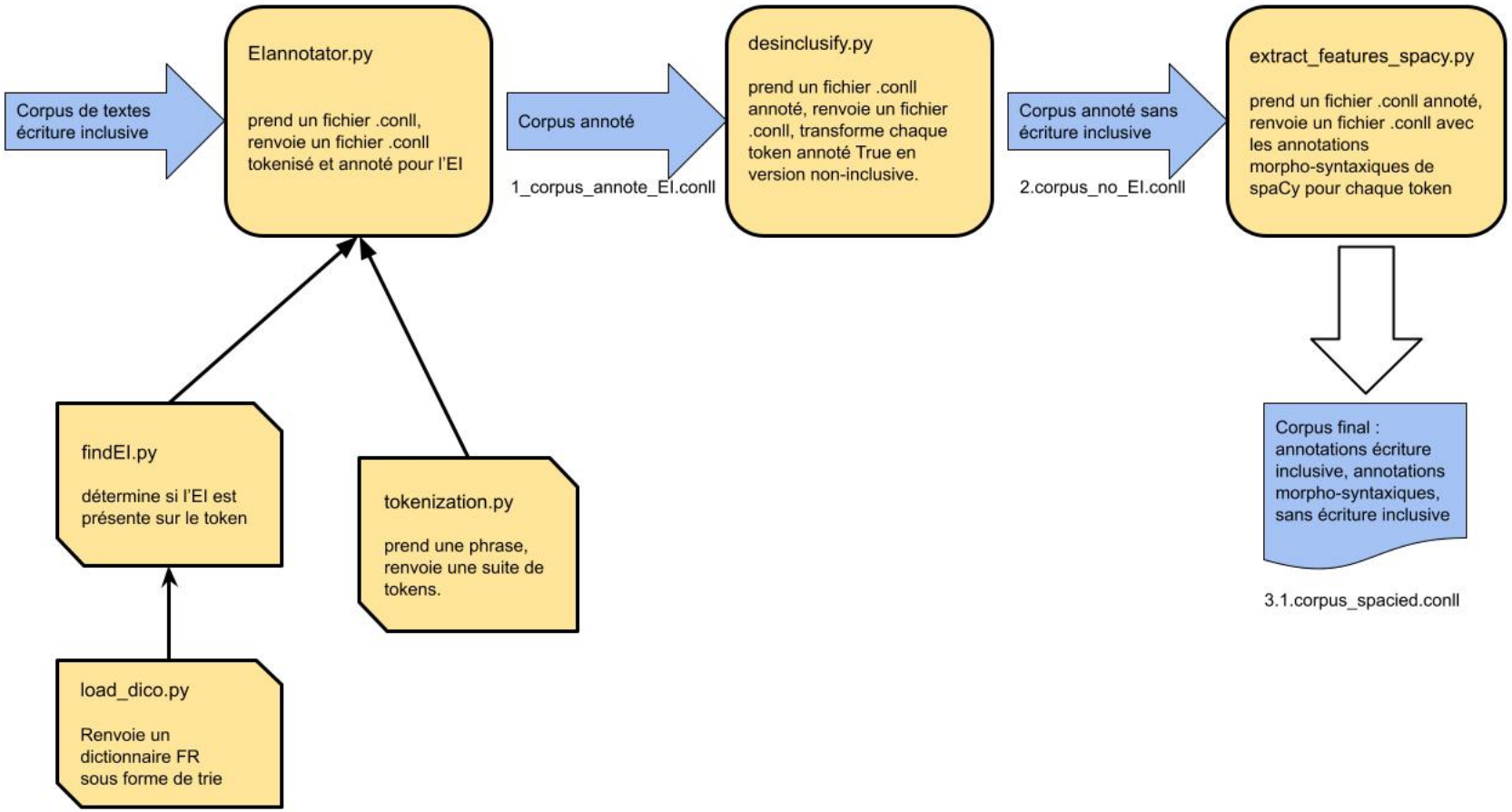

FIGURE 11 – Représentation de la méthode de génération du corpus d'écriture inclusive

3.1.2 Développement des modèles

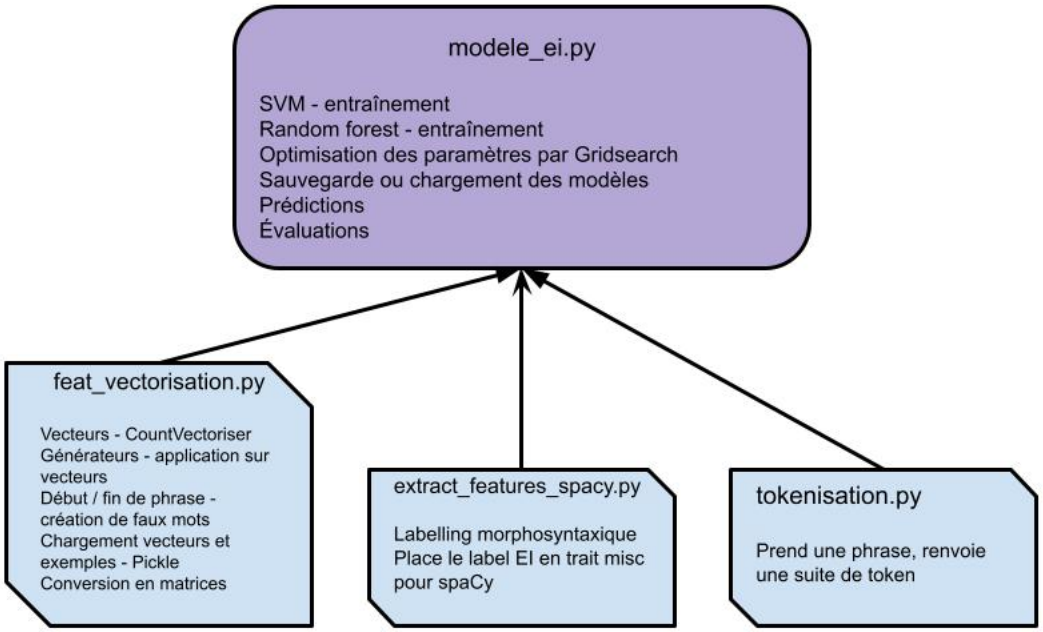

FIGURE 12 – Représentation de l'architecture du programme d'entraînement et de prédiction des modèles.

Le programme `feat_vectorisation.py` génère les *vectoriseurs* (qui sont stockés dans un fichier grâce au module `pickle`) et les exemples vectorisés.

Le programme `model_ei.py` contient les principales fonction d'apprentissage, de prédiction, d'évaluation et de transformation des phrases en écriture inclusive. Ce programme importe les exemples et *vectoriseurs* générés par le programme précédent, puis divise les exemples en un ensemble d'apprentissage et un ensemble de test (80%-20%) en fixant le paramètre *random_state* afin de toujours obtenir la même répartition des exemples.

- `trainSVM` et `trainRandomForest` renvoient un modèle entraîné, avec ou sans gridsearch.
- `save_model` et `load_model` permettent de gérer la persistence de ces modèles.
- `eval` évalue un modèle sur à un ensemble donné et les étiquettes correspondantes, et affiche également l'évaluation de la baseline (effectuée par la fontcion `baseline`). `false_positives` et `false_negatives` affichent les faux positifs et négatifs pour un modèle et un ensemble étiqueté donné.
- `most_important_RF` affiche l'importance des classes de features d'un modèle de type forêt aléatoire, et `visualize_tree` exporte un de ses arbres au format .png.
- La fonction `process_sentence` transforme une phrase en écriture inclusive en faisant appel à `predict_sentence`, qui extrait les caractéristiques morphosyntaxiques, et crée le vecteur correspondant à chaque mot, puis à `convert` qui renvoie la phrase

transformée (en s'appuyant sur `merge` et `find_root`). Ces fonctions utilisent Lexique ([New *et al.*, 2004]) stocké dans le fichier `Lexique383.tsv`.

Certaines de ces fonctions peuvent être utilisées grâce à des commandes détaillées dans le manuel utilisateur·rice (section 5) et dans l'aide en ligne.

3.2 Difficultés rencontrées

Une difficulté importante a été rencontrée lors de l'extraction des caractéristiques morphosyntaxiques avec spaCy, nous rencontrions de nombreuses erreurs lors de l'initialisation du parser spaCy, et nous avons mis longtemps à trouver la source de cette erreur. Il s'agissait en fait d'un conflit entre les versions de plusieurs modules (`spacy, spacy-conll, spacy-stanza`), car le module `spacy-conll` n'avait pas été mis à jour pour être utilisé avec spaCy3 [5]. Nous avons alors pu réaliser l'utilité des environnements virtuels afin d'effectuer les installations des versions des modules correspondant exactement à nos besoins et d'ensuite transmettre ces informations à d'autres utilisateurs. Vous trouverez donc un document `requirements.txt` spécifiant la liste des modules utilisés et leurs versions.

Nous avons également été confrontée à une nouvelle problématique : comment permettre la persistence d'objets que nous voulions réutiliser, comme les modèles entraînés, les vectoriseurs ou encore les exemples ? Nous avons alors découvert l'existence du module `pickle`, qui nous a facilement permis de stocker dans des fichiers ces objets, et de les importer à nouveaux. Nous avons toutefois rencontré des difficultés lorsque nous avons voulu importer un objet depuis un autre programme que celui où il avait été créé. Notre solution a été de générer les objets depuis `model_ei.py` en faisant appel aux autres programmes, afin de pouvoir ensuite utiliser ces objets dans notre programme principal.

4 Pistes d'amélioration

Lors du développement de projet, nous avons envisagé de nombreuses pistes d'améliorations, sans malheureusement avoir le temps de les explorer davantage. Nous les décrivons succinctement dans cette partie.

Le corpus pourrait être amélioré de plusieurs manières :

— en diversifiant le type de documents.

— en améliorant la tokenisation du corpus.

— en annotant dans des textes (pas forcément en écriture inclsuive) l'ensemble des mots qui ont des références génériques, ou à des groupes mixtes.

Le choix des caractéristiques pourrait être amélioré en diminuant leur nombre et en augmentant l'apport informationnel de chaque caractéristique, en s'appuyant par exemple sur leur importance dans le modèle RandomForest.

Nous utilisons dans le projet une fenêtre de taille 2, ce qui ne nous permet pas de prendre en compte des informations plus éloignées qui sont sans doute cruciales pour déterminer si un mot devrait être écrit sous forme inclusive. Pour prendre davantage d'informations

5. C'est ce ticket Github qui nous a permis de comprendre la source du problème.

en compte, nous pourrions augmenter la taille de cette fenêtre (ce qui aurait un coût computationnel) ou utiliser d'autres modèles qui intègrent cette problématique, comme les réseaux de neurones récurrents *Long short-term-memory* (LSTM).

Au lieu d'utiliser un encodage *one-hot* pour encoder les mots, l'utilisation de plongements lexicaux pourraient permettre au modèle de prendre en compte des traits tels que l'animéité, tout en diminuant le nombre de caractéristiques.

Au lieu de créer des exemples pour chaque mot, il aurait été pertinent de créer un exemple par syntagme, puisque c'est à l'échelle du syntagme que la question de la transformation en écriture inclusive se pose. Si des incohérences persistaient – comme dans l'exemple 6, ou seul un des pronoms coréférents *ils* est prédit en classe positive – un module de résolution des coréférences pourraient être implémenté.

Nous aurions également pu améliorer l'implémentation du programme `model_ei.py` en adoptant une approche orientée objet, par exemple en créant une classe `Exemples`, qui aurait permis de mieux organiser les features et les exemples correspondants.

5 Manuel utilisateur·rice

Un fichier `readme.txt` (figure 13) indique les étapes à suivre pour faire fonctionner le projet sur son ordinateur. Plusieurs fonctions du programme principal `model_ei.py` (comme l'apprentissage, l'évaluation, la conversion de phrase en écriture inclusive, etc.) peuvent être utilisé en indiquant des arguments, précisé dans le message d'aide (voir figure 14).

Voici quelques exemples d'utilisation du programme :

```
python3 codeEI/model_ei.py --convert "Les étudiants doivent rendre leur
rapport le 17 juin." -v
```

Le programme charge le modèle par défaut et renvoie une version en écriture inclusive de cette phrase. Le drapeau `-v` permet aussi de visualiser des informations supplémentaires, comme l'étiquette prédite pour chaque mot de la phrase.

```
python3 codeEI/model_ei.py --train RandomForest --eval test --save 5
```

Le modèle effectue un entraînement du SVM avec les hyperparamètres que nous avons choisi après avoir effectué la gridSearch, évalue le modèle sur le corpus de test et le compare à la baseline, puis enregistre le modèle sous le nom `RandomForestClassifier_V5`. L'option `-train` ne fait que générer un modèle déjà fourni dans l'archive, le but étant seulement de montrer que le programme est fonctionnel, pas d'améliorer les résultats du modèle.

```
python3 codeEI/model_ei.py --load SVC_V2 --eval test
```

Le programme charge un modèle stocké dans le fichier `SVC_V2` et l'évalue sur le corpus de test, en le comparant à la baseline.

```
1) Dézipper le projet
2) Ouvrir le terminal et se placer dans le répertoire du projet

3) Nous recommandons d'utiliser un environnement virtuel et d'y installer
les dépendances de ce projet :

    pip install virtualenv
    python3 -m venv env
    source env/bin/activate
    pip install -r requirements.txt

Pour sortir de l'environnement virtuel après utilisation du programme :
    deactivate

Il est particulièrement important de ne pas utiliser spaCy 3, car le
module spacy-conll n'est que compatible avec spaCy 2.

4) Le modèle fr-core-news-sm de spaCy doit être téléchargé :
    python3 -m spacy download fr_core_news_sm

5) Le programme principal model_ei.py permet la conversion de phrases en
écriture inclusive à partir de modèles, ainsi que l'entraînement et
l'évaluation de ces modèles.

Il doit être exécuté depuis le répertoire parent (pas depuis codeEI) :
    python3 codeEI/model_ei.py

il dispose d'une aide en ligne qui détaille son usage :
    python3 codeEI/model_ei.py --help

Deux modèles déjà entraînés peuvent être chargés :

- Pipeline_V2 (modèle SVM, chargé par défaut)
- RandomForestClassifier_V4
```

FIGURE 13 – Fichier README.txt pour l'ensemble du projet

```
usage: model_ei.py [-h] [--convert CONVERT] [-s SEPARATOR] [--load LOAD]
                   [--eval {test,train}] [--train {RandomForest,SVM}]
                   [--gridsearch] [--save SAVE] [--verbose]

Automatically transform sentences into inclusive writing (French).

optional arguments:
  -h, --help            show this help message and exit
  --convert CONVERT     convert the given string into écriture inclusive.
  -s SEPARATOR, --separator SEPARATOR
                        Choose separator for EI forms. Default : ·
  --load LOAD           load model from a file. By default the model used
                        is the one with the best results.
  --eval {test,train}   display evaluation metrics for current model.
                        Choose "all" for baseline.
  --train {RandomForest,SVM}
                        Choose a model to train.
  --gridsearch, -g      search best parameters during training with a grid
                        search
  --save SAVE           Save the model with a chosen version number.
  --verbose, -v
```

FIGURE 14 – Manuel d'utilisateur·rice du programme `model_ei.py`

Références

[Abbou, 2017] ABBOU, J. (2017). (typo)graphies anarchistes. où le genre révèle l'espace politique de la langue. *Mots. Les langages du politique*, (113):53–72.

[Araujo, 2019] ARAUJO, R. (2019). Développement d'un outil de transformation écriture inclusive écriture paritaire : Caractérisation du point médian. Mémoire de master, Sorbonne Université, Paris, Fr.

[Brauer, 2008] BRAUER, M. (2008). Un ministre peut-il tomber enceinte ? l'impact du générique masculin sur les représentations mentales.

[Burnett et Pozniak, 2020] BURNETT, H. et POZNIAK, C. (2020). Political dimensions of Écriture inclusive in parisian universities.

[Honnibal et Montani, 2017] HONNIBAL, M. et MONTANI, I. (2017). spaCy 2 : Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear.

[Lessard et Zaccour, 2018] LESSARD, M. et ZACCOUR, S. (2018). *Manuel de grammaire non sexiste et inclusive*. Editions Syllepses.

[New *et al.*, 2004] NEW, B., PALLIER, C., BRYSBAERT, M. et FERRAND, L. (2004). Lexique 2 : A new french lexical database. *Behavior Research Methods, Instruments, Computers*, (36 (3)):516–524.

Table des figures