Projet de Syntaxe et Sémantique: Amélioration du Perceptron

Margin Infused Relaxed Algorithm (MIRA)

Luc Giffon

7 février 2017

MACAON2 est une suite logicielle permettant d'effectuer des tâches standard de traitement automatique de la langue. Elle est composée de plusieurs modules réalisant des traitements classiques (découpage des phrases en mots, étiquetage morpho-syntaxique, analyse morpho-syntaxique). Parmis les modules d'analyse syntaxique, il existe un analyseur, "*Transition-Based Parser*", dont le rôle est de reconnaitre les dépendances syntaxiques dans un texte, c'est à dire les relations de dépendances entre mots. Cet analyseur utilise un algorithme glouton qui, à chaque itération (e.g. pour chaque exemple d'apprentissage), utilise un perceptron pour prendre une décision basée sur l'état courant de l'analyse. Cet état est caractérisé par plusieurs milliers de paramètres binaires.

L'objet de ce projet est d'améliorer ses résultats via une implémentation alternative : MIRA, *Margin Infused Relaxed Algorithm*, dont le but général est de creuser la frontière de décision entre les différents choix possibles pour un exemple donné.

1 Introduction

Une phrase peut être vue comme un ensemble de mots liés entre eux d'après des relations de dépendance hiérarchique où, pour chaque connexion [1] :

— une catégorie fonctionnelle y est associée sous la forme d'une étiquette
— le terme supérieur y est appelé *Régissant* et le terme inférieur *Subordonné*.

L'objectif est de fournir une analyse syntaxique à partir d'une phrase donnée. C'est à dire de découvrir les liens entre Régissants et Subordonnés, les étiqueter et construire l'arbre de dépendances de la phrase (Figure 1). Pour se faire, il y a deux approches à disposition :

— la première est basée sur des règles de grammaire explicites
— la seconde n'utilise pas de règles de grammaire mais propose un algorithme d'analyse par transition offrant un résultat considéré comme *le plus probable* mais sans garantie de validité

L'algorithme d'analyse par transition est un algorithme glouton qui produit des arbres de dépendance. A chaque étape du parcours de l'arbre des possibilités, l'algorithme utilise un système de décision pour définir la meilleure solution. L'idée étant d'espérer obtenir le *meilleur résultat final* en empruntant le chemin des *meilleures décisions locales*.

Dans la suite logicielle MACAON2 , cette décision est prise à l'aide d'un perceptron qui est capable de retourner la meilleure transition à effectuer à partir d'un état représenté par un vecteur de features binaires (étiquettes du mot courant et des mots du voisinage,

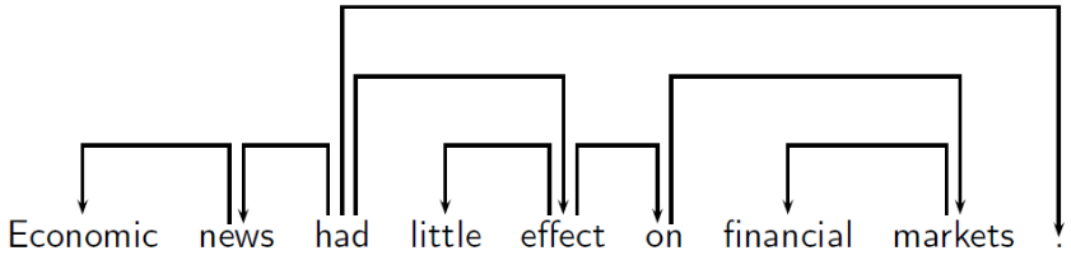

FIGURE 1 – Exemple de liens de dépendance. Une flèche d'un Régissant vers un subordonné

...). Ce perceptron ayant entrainé son modèle sur un corpus d'apprentissage correctement annoté.

L'objet de ce projet est d'implémenter et comparer des systèmes de décision alternatif :
— le perceptron moyenné
— MIRA : *Margin Infused Relaxed Algorithm*
— ADAGRAD : *ADAptative GRADient algorithm*

L'algorithme MIRA, proposé par Crammer en 2003 [3] a déjà été utilisé dans le cadre du traitement automatique du langage pour de la traduction automatique [6] [4] mais aussi pour de l'analyse Syntaxique des dépendances [5] [2]. Cependant, les approches de McDonald et Bohnet sont basées sur les *maximum spanning trees* alors que MACAON2 met en œuvre une analyse par transition.

D'autre part, le fonctionnement de MACAON2 se fait en utilisant en entrée des vecteurs de features binaires pour le perceptron. Ceux-ci avaient déjà montré des résultats intéressants dans le cadre de la traduction de l'Arabe à l'Anglais [6].

L'implémentation en Python du perceptron moyenné, les problèmes rencontrés et les résultats obtenus avec ce système seront discutés dans la section 2. En section 3, nous verrons le fonctionnement de MIRA et les modifications qu'il apporte à l'algorithme du perceptron.

Les difficultés rencontrées avec le perceptron moyenné et MIRA ne m'ont pas laissé le temps de réaliser une implémentation de ADAGRAD.

2 Perceptron Moyenné

Il est connu que la version originelle du perceptron est particulièrement sensible aux dernières instances d'un ensemble d'apprentissage car à l'issu du-dit apprentissage, c'est la version la plus récente du modèle qui est retenue.

L'une des idées imaginées pour contrer ce problème consiste en la construction de plusieurs classifieurs au cours de l'apprentissage, dont chacun aurait des forces et des faiblesses. Pour chaque prédiction, ce serait la moyenne des décisions prises par ces classifieurs qui serait retenue.

L'implémentation de ce perceptron moyenné étant déjà mis à disposition dans MACAON2, je l'ai simplement portée vers Python, le langage utilisé pour la réalisation du projet.

2.1 Implémentation choisie

L'ensemble du code du projet est articulé autour d'une classe principale appelée `Perceptron`. Les objets de cette classe sont initialisés avec une matrice, `feature_class_weights_matrix`, qui associe un poids pour chaque feature et pour chaque classe. Éventuellement, on peut utiliser des paramètres booléens pour spécifier l'utilisation d'un perceptron moyenné, `averaged`, et / ou de l'algorithme MIRA, `MIRA`.

Cette classe comporte trois méthodes principales relatives aux trois cas d'utilisation principaux du perceptron :

- Une méthode `train` qui utilise un corpus (argument `corpus`) représenté sous la forme d'un itérable dont chaque entrée est composée d'un tuple (**vecteur de features**, classe). Cette méthode est paramétrisée par le nombre d'itération voulues via `n_iter`
- Une méthode `test` qui utilise aussi un corpus (argument `corpus` représenté de la même façon que dans `train`)
- Une méthode `evaluate` qui, à partir d'un **vecteur de features** (argument `tpl_i_features_vector`), retourne la classe prédite par le perceptron avec éventuellement son score (argument `get_score`)

2.2 Problèmes rencontrés

Bien que cette étape du projet aurait dû être réalisée très rapidement, j'ai rencontré un problème que j'ai mis de longs jours à comprendre avant de pouvoir le résoudre. C'est dommage car il était extrêmement simple : j'utilisais le corpus `train.cff` comme corpus d'entraînement et le corpus `test.cff` comme corpus de test. Cela semble à peu près normal mais, en fait, je devais utiliser le corpus `train.cutoff.cff` ! Qui, si j'ai bien compris, retire toutes les features de `train.cff` qui n'apparaissent pas dans `test.cff`. Il est évident que l'apprentissage réalisé en tenant compte de features absentes des données test ne pouvait pas obtenir une bonne performance de généralisation sur ces données.

Il est dommage que je n'ai plus les résultats que j'avais obtenu lorsque j'avais rencontré ce problème mais, dans les grandes lignes : plus j'augmentais le nombre d'itération, meilleurs étaient les résultats sur le corpus d'entrainement et pire étaient les résultats sur le corpus de test. Cela faisait clairement penser à du sur-apprentissage, à ceci près que les résultats sur le corpus de test étaient non seulement de pire en pire mais aussi très médiocres : plus de 70% d'échec dans le meilleur des cas (une seule itération) !

2.3 Résultats obtenus

Les expériences ont été menées comme suit : pour chacun des deux types de classifieur, on réalise un entraînement sur les données issues du fichier `train.cutoff.cff`. Cet entraînement est réalisé avec différents nombres d'itérations allant de zéro à quinze. Une fois les 30 classifieurs entraînés, ils sont mis à l'épreuve sur les données test issues du fichier `test.cff`. Les résultats numériques sont présentés dans le tableau 2.3 où le meilleur score obtenu pour chaque classifieur est représenté en rouge. Des résultats visuels sont présentés dans la figure 2.3.

Sur la figure 2.3, on voit nettement que peu importe le nombre d'itération, le perceptron moyenné apporte une amélioration significative au perceptron. Plus précisément, dans le tableau 2.3, on voit que le meilleur résultat obtenu par le perceptron vanilla est de 6.656% d'erreur pour 3 itérations contre 4.900% d'erreur pour 7 itérations par le perceptron moyenné.

n iteration	Taux d'erreur : Vanilla	Taux d'erreur : Averaged
1	7,076	5,071
2	7,313	4,966
3	6,656	4,929
4	7,724	4,954
5	7,921	4,918
6	7,203	4,949
7	7,681	4,900
8	6,863	5,018
9	6,664	4,958
10	7,669	4,963
11	6,809	5,067
12	6,725	5,007
13	6,725	5,053
14	6,679	5,058
15	6,812	5,064

TABLE 1 – Résultats obtenus avec le perceptron moyenné contre le perceptron dit "Vanilla"

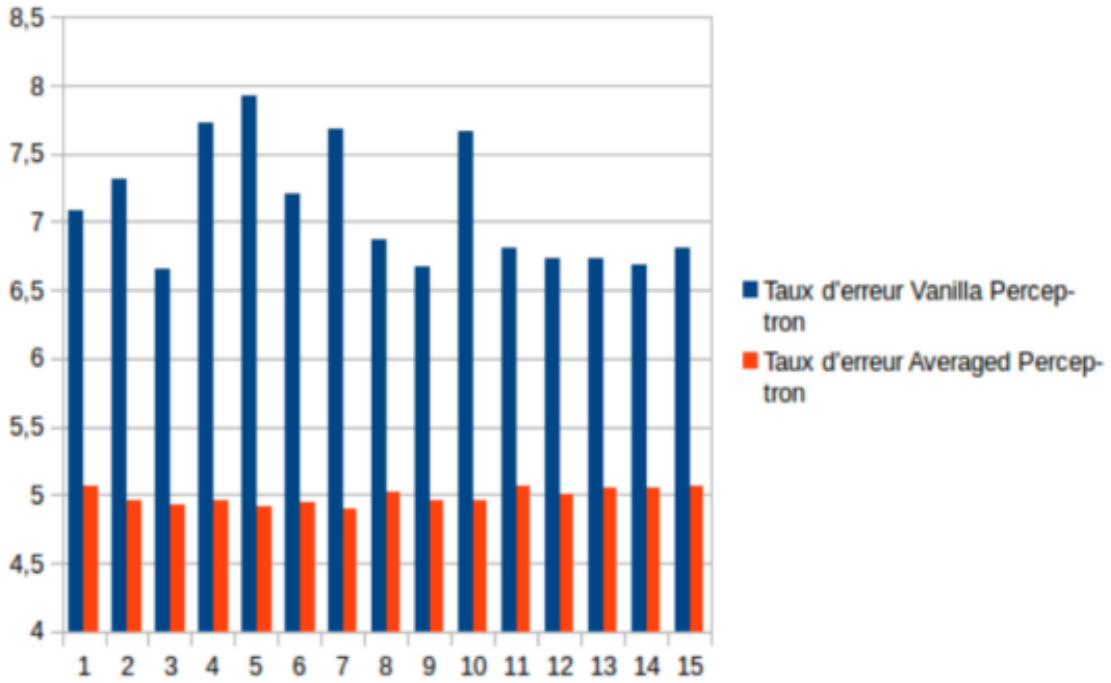

FIGURE 2 – Histogramme comparatif du perceptron vanilla contre le perceptron moyenné

n iteration	Taux d'erreur MIRA	Taux d'erreur averaged MIRA
1	7,311	5,085
2	7,848	4,996
3	7,499	4,885
4	7,189	4,910
5	7,527	4,969
6	6,807	4,958
7	6,455	4,930
8	6,624	4,981
9	6,409	4,998
10	6,212	4,988
11	6,657	4,992
12	6,543	5,031
13	7,076	5,045
14	8,033	5,018
15	6,543	5,047

TABLE 2 – Résultats obtenus avec MIRA contre MIRA moyenné

3 MIRA

3.1 Implémentation choisie

Pour l'implémentation de MIRA, c'est la même classe que celle construite pour le `Perceptron` (section 2.1) qui est utilisée. La différence majeure est en fait située au niveau de la condition sur l'attribut `mira` de l'objet, pendant l'entraînement : si `mira` est à `True`, alors le traitement spécifique est effectué.

3.2 Résultats obtenus

L'ensemble des expériences concernant MIRA ont été réalisées de façon similaire à celles sur le perceptron moyenné. C'est à dire que, pour des nombres d'itérations allant de un à quinze, les classifieurs ont été entraînés sur un corpus d'entraînement et mise à l'épreuve sur un corpus de test.

3.2.1 MIRA vs Perceptron Vanilla

En figure 3.2.1, on peut avoir l'impression que, d'une façon générale, MIRA n'apporte pas de réelle plus-value au perceptron Vanilla car il est parfois meilleur et parfois moins bon. En revanche, si on regarde les résultats numériques présentés dans la tableau 3, on voit que le meilleur résultat obtenu par MIRA est de 6.212% d'erreur pour 10 itérations, ce qui est quand même environ 0.4% meilleur que le perceptron.

En revanche, on pourrait se poser la question de savoir si multiplier par trois le nombre d'itérations pour obtenir 0.4% de réussite supplémentaire est réellement rentable ? A mon avis, ça dépend : est-ce-qu'on va apprendre le classifieur une bonne fois pour toutes (mode batch) ? Ou bien va-t-on utiliser l'algorithme *on-line* [6] pour une mise à jour dynamique ?

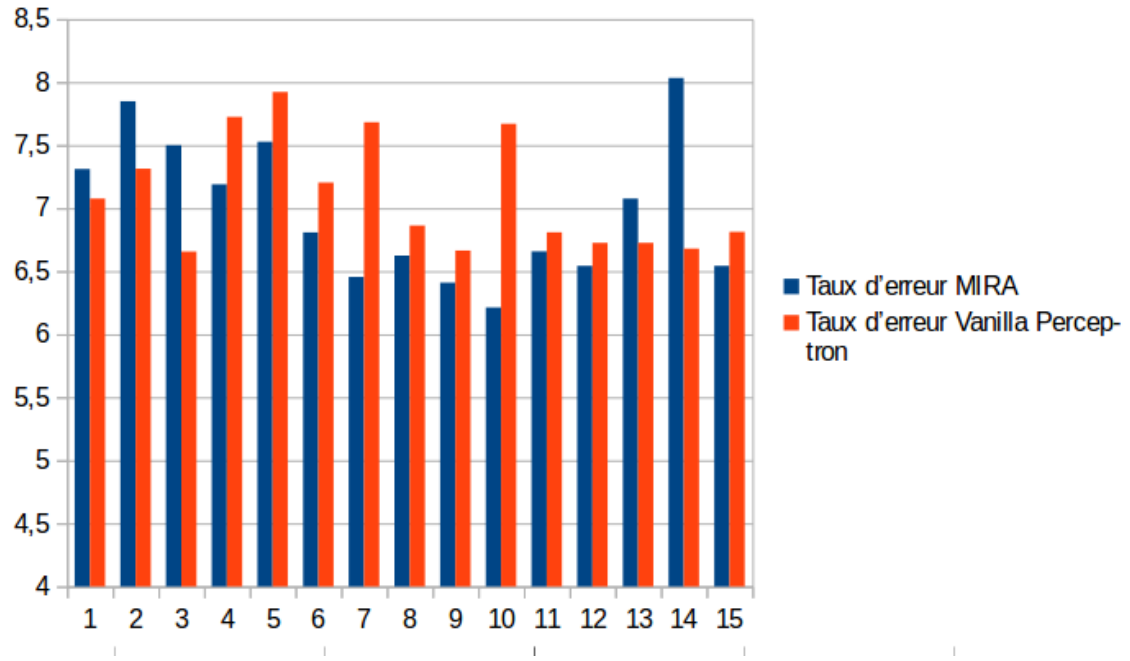

FIGURE 3 – Histogramme comparatif du perceptron vanilla contre MIRA

3.2.2 MIRA moyenné vs Perceptron Moyenné

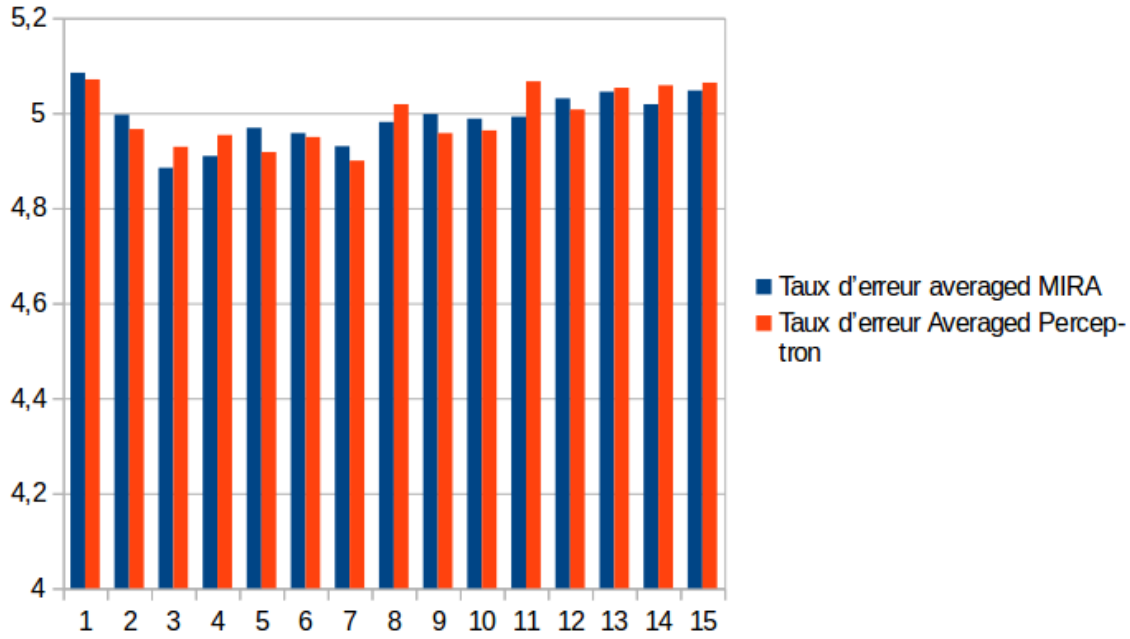

FIGURE 4 – Histogramme comparatif du perceptron moyenné contre MIRA moyenné

En figure 3.2.2, on constate que les performances générales de MIRA et du perceptron moyenné sont très similaires. Cette fois-ci, ce n'est pas simplement une impression en ce sens que le meilleur résultat obtenu par MIRA moyenné est de 4.885% d'erreur contre 4.900% par le perceptron moyenné.

Ces résultats auraient pu être assez décevants mais Benoit Favre me les a expliqué : l'amélioration du Perceptron par l'utilisation de la moyenne va lui permettre d'améliorer sa justesse, c'est à dire que d'une façon générale, le classifieur a une meilleure idée de la

position des classes. En revanche, l'algorithme MIRA améliore la précision du classifieur, c'est à dire qu'il va permettre de mieux détecter les *out-liers*, e.g. les examples qui ne respectent pas tout à fait la distribution habituelle des données.

Il est donc normal d'observer des résultats généraux similaires pour le perceptron moyenné et MIRA moyenné car justement, en général le fait que les deux classifieurs soient moyennés leur assure une bonne capacité de prédiction mais la différence va se jouer sur les cas particuliers (qu'on ne peut naturellement pas observer avec des résultats généraux).

3.2.3 MIRA vs MIRA moyenné

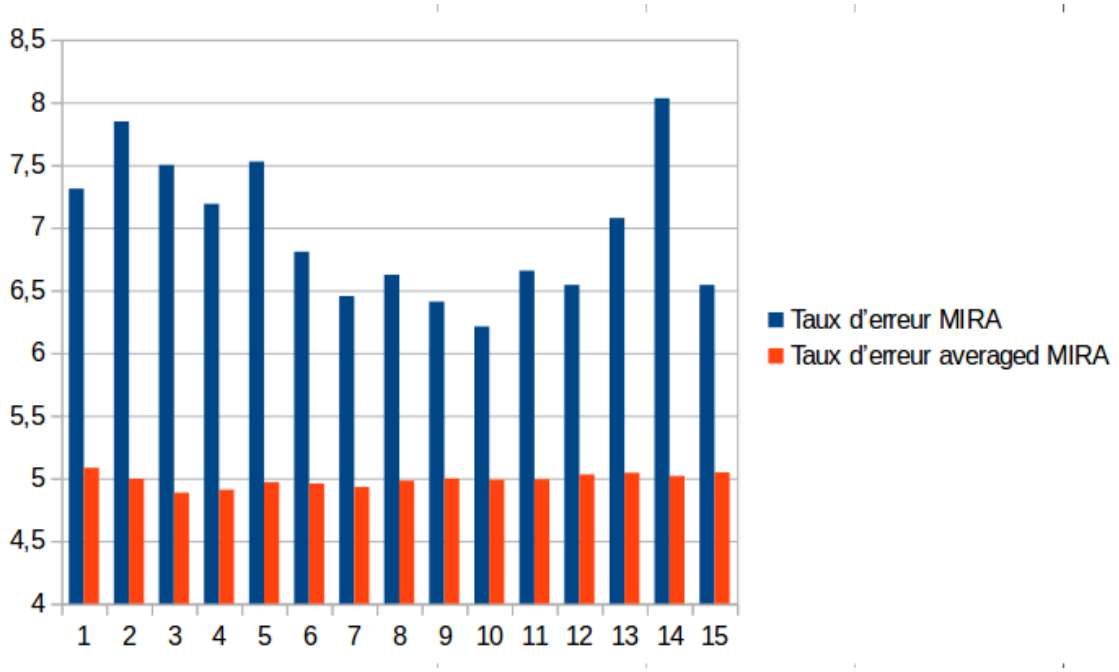

FIGURE 5 – Histogramme comparatif de MIRA contre MIRA moyenné

Enfin, en figure 3.2.3, on peut observer les résultats de MIRA contre MIRA moyenné. Ces résultats font écho avec ce qui a été écrit plus haut : moyenner les poids obtenus avec l'algorithme MIRA permet de récupérer un classifieur globalement plus juste.

4 Utilisation

Maintenant que je vous ai présenté les résultats que j'ai obtenus, vous voudrez peut-être vérifier que je ne vous ai pas dit n'importe quoi depuis le début (mais promis, mes résultats sont vrais). Il est également possible que vous vouliez réaliser d'autres expériences. Dans cette section, nous allons voir comment utiliser le programme.

4.1 Paramètres

En lançant le programme avec l'option `-h`, vous devriez obtenir l'affichage suivant :

Example 4.1. Output de l'option `-h`

```
rewritted_perceptron

Usage:
  rewritted_perceptron TRAIN TEST [--size_train=int] [--size_test=int] [--
      iteration_number=int] [--averaged] [--mira] [-h]
```

```
  rewritted_perceptron TRAIN TEST [--size_train=int] [--size_test=int] [--
      iteration_number=int] [--averaged] [--mira] [-h]

Arguments:
    TRAIN                   filename train corpus
    TEST                    filename test corpus

Options:
  -h --help                 Show this screen.
  --size_train=int          Size of train corpus. -1 means all corpus [default:
       -1].
  --size_test=int           Size of train corpus. -1 means all corpus [default:
       -1].
  --iteration_number=int    Number of iterations [default: 1].
  --averaged                Run averaged Perceptron [default: False].
  --mira                    Run mira Perceptron [default: False]
```

4.2 Installation et Execution

Le code est disponible en ligne sur mon *github* personnel : https://github.com/lucgiffon/morpho_tagger). Une fois le dépôt cloné, allez dans le répertoire racine et installez les modules Python requis : `pip install -r requirements.txt`. (assurez vous que `pip` fonctionne avec Python3)

Maintenant que les dépendances du programme sont installées, il ne reste qu'à exécuter :

Example 4.2. Construire un classifieur de type MIRA moyenné sur l'ensemble du corpus `train.cutoff.cff` et le tester sur `test.cff`

```
python3 rewritted_perceptron.py train.cutoff.cff test.cff --iteration_number=3
    --mira --averaged
```

5 Conclusion

Enfin, cette conclusion va être découpée en deux parties : la première, une conclusion sur les résultats obtenus avec MIRA et sur les potentielles questions auxquelles répondre. La seconde concernera la méthode que j'ai employé et les enseignement que je peux en tirer.

5.1 MIRA

Il semble que MIRA offre des résultats globaux à peu près équivalents au perceptron moyenné. On peut donc se demander si son implémentation dans MACAON2 est vraiment rentable.

D'une part, comme je l'ai déjà souligné plus tôt dans la rapport, je pense que tout dépend de l'utilisation qui sera faite de MACAON2 (*batch* ou *on-line*).

D'autre part, une expérience me semble intéressante à réaliser : il faudrait trouver un jeu de donnée dont les *out-liers* seraient signalés (mais sans prendre cette information en compte dans l'apprentissage) et confronter les performance de généralisation de MIRA par rapport au perceptron sur ces données anormales. De cette façon, je pense qu'on pourrait apporter une conclusion plus intéressante aux performances de MIRA.

Une autre expérience pourrait consister à considérer, pour chaque mise à jour du classifieur, les n-best éléments au lieu d'un seul. Je m'explique : pour le moment, lorsque MIRA commet une erreur de classification lors de l'entraînement, les poids associés à la bonne classe son majorés et ceux associés à la classe prédite (donc la fausse classe) sont minorés. Il pourrait être intéressant de, au lieu de considérer uniquement la classe prédite, considérer

toutes les classes qui ont obtenu un meilleur score que la bonne classe et pour chacune d'elle, majorer les poids de la bonne classe et minorer ceux des mauvaises.

Il s'agit d'un principe de mise à jour déjà proposé par Eva Hasler [4] pour la traduction qui m'a semblé intéressant.

5.2 Méthode

En ce qui concerne ma méthode de travail, je me suis rendu compte de plusieurs choses dans ce projet qui me semblent intéressant de partager :

— Pour la réalisation des expériences, j'ai écris un petit script python qui lançait mes différentes expériences sur différents threads de ma machine avec une certaine plasticité (quelques paramètres pour configurer les runs). Cette idée est bonne, mais il me semble qu'il existe déjà des outils pour réaliser ce genre de choses : quand je faisais de la bio-informatique, on me parlait souvent de `snakemake` pour réaliser des pipelines d'expérience, il faudrait regarder de ce côté là.

— Toujours pour la réalisation des expériences, je me suis rendu compte lors de la rédaction que, pour certains résultats (par exemple la comparaison entre MIRA moyenné et le perceptron moyenné), il aurait été intéressant de réaliser plusieurs fois les expériences et de dessiner des "boites à moustache" sur mes histogrammes afin d'avoir une meilleure idée de si deux résultats étaient équivalents.

— Enfin, je me suis aperçu lors de la rédaction du rapport qu'il y avait beaucoup de documents que j'aurais dû lire plus tôt dans le projet et qui m'auraient été d'une grande aide pour ma compréhension générale. Je crois qu'ici, il s'agit surtout d'un manque d'expérience mais maintenant je le saurais : ne pas lésiner sur la "pré-documentation"

Cette section doit être atypique dans un rapport mais, en ce qui me concerne, ce projet m'a plus apporté du point de vue "méta" (e.g. sur la façon de faire de la science) que du point de vue technique et il m'a semblé important de le signaler. Merci.

Références

[1] M Arrivé. Les éléments de syntaxe structurale, de l. tesnière. *Langue française*, 1(1) :36–40, 1969.

[2] Bernd Bohnet. Efficient parsing of syntactic and semantic dependency structures. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning : Shared Task*, pages 67–72. Association for Computational Linguistics, 2009.

[3] Koby Crammer and Yoram Singer. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3(Jan) :951–991, 2003.

[4] Eva Hasler, Barry Haddow, and Philipp Koehn. Margin infused relaxed algorithm for moses. *Prague Bull. Math. Linguistics*, 96 :69–78, 2011.

[5] Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics, 2005.

[6] Taro Watanabe, Jun Suzuki, Hajime Tsukada, and Hideki Isozaki. Online large-margin training for statistical machine translation. In *In Proc. of EMNLP*. Citeseer, 2007.