TAL
Rapport
Application de l'analyse de discours au résumé automatique de texte

Rémi Cadène
Niki Rohani

7 mai 2015

Table des matières

1 Introduction **3**

2 Organisation au sein du bînome **3**
2.1 Module Hilda 3
2.2 Module RST-DT vers DEP-DT 3
2.3 Module TKP et ILP 3
2.4 Module d'évaluation ROUGE 3
2.5 Assemblage des modules 4

3 Modules **4**
3.1 Création d'un résumé automatique de texte 4
3.1.1 Création de l'arbre RST avec Hilda 4
3.1.2 Transformation de l'arbre RST en un arbre DEP 5
3.1.3 Formulation du problème et résolution 5
3.1.4 Classe Summarize 6
3.2 Evaluation du résumé avec le calcul du Score ROUGE 6
3.2.1 Classe Rouge 6
3.2.2 Classe Evaluation 7

4 Architecture de l'application **7**
4.1 Installation de Hilda 7
4.2 Cheminement vers DEP 7
4.3 Solveur ILP 8
4.4 Évaluation avec le score ROUGE 8
4.5 Architecture complète et utilisation 9

5 Expérience **9**
5.1 Recherche de résultats à comparer 9
5.2 Comparaison des résultats 9

6 Conclusion **10**

1 Introduction

Ce rapport est destiné à présenter une application utilisant l'analyse de discours pour effectuer un résumé automatique de texte. Les méthodes de résumé automatique de texte que l'on peut trouver dans la littérature sont très variées. Patil et al. (2015) [5] dénotent deux grandes familles de résumé automatique de texte en faisant la différence entre les méthodes extractives et abstractives. Les méthodes extractives sélectionnent des phrases d'un texte afin d'en constituer un résumé, tandis que les méthodes abstractives sont issus du Natural Langage Processing (NLP). Dans ce rapport, nous nous concentrons sur une méthode extractive. L'idée intuitive est qu'il existe un lien fort entre les relations de langage existant entre plusieurs unités de discours (EDUs) dans un texte et leur importance dans le sens général du texte. Ainsi, l'analyse des structures discursives peut nous permettre de sélectionner des unités de discours jouant un rôle dans le sens de ce dernier. En effet, Hirao et al. (2013) [3] proposent une méthode utilisant l'analyse des relations rhétoriques existantes entre les unités de textes afin de créer un résumé.

Dans ce rapport nous expliquons comment nous avons implémenté cette méthode de l'état de l'art afin de pouvoir tester ses résultats. Ainsi, dans une première section, nous présentons l'organisation et le partage des tâches au sein de notre bînome. Dans une seconde section, nous détaillons chronologiquement les différents modules utilisés par notre application. Dans une troisième section, nous expliquons l'architecture de notre application. Enfin, dans une dernière section, nous détaillons la mise en place de nos expériences en incluant une interprétation et une comparaison de nos résultats.

2 Organisation au sein du bînome

2.1 Module Hilda

Niki Rohani a installé Hilda.

2.2 Module RST-DT vers DEP-DT

Rémi Cadène a codé le module RST-DT vers DEP-DT qui comprend un parser et l'implémentation d'un algorithme présent dans l'article de recherche de Hiro et al. (2013) [3].

2.3 Module TKP et ILP

Niki Rohani a codé le module TKP et ILP.

2.4 Module d'évaluation ROUGE

Rémi Cadène a codé le module d'évaluation ROUGE (EvaluationApp) qui comprend un parser de résumés, un outil de générations de deux types de fi-

chiers (summary.html et settings.xml) qui encapsule le logiciel ROUGE-1.5.5.pl (summaries2rouge.py), un outil d'évaluation de notre méthode par le calcul des scores ROUGE1 et ROUGE2 pour chaque résumé de référence du corpus RST-DT (evaluation.py), et le logiciel rouge2csv.pl permettant de mettre en forme les résultats obtenus. Ce dernier a toutefois été modifié pour une utilisation plus facile.

2.5 Assemblage des modules

Niki Rohani a assemblé les trois premiers modules afin de créer notre application permettant de générer un résumé automatique à partir d'un texte.

3 Modules

3.1 Création d'un résumé automatique de texte

3.1.1 Création de l'arbre RST avec Hilda

Afin de pouvoir analyser le discours, nous avons dans un premier temps besoin de transformer le texte sous forme d'un Rhetorical Structure Theory Discourse Tree (RST-DT). Un RST-DT est un arbre de relation rhétoriques entre plusieurs élements de texte appelés EDU[4] Pour effectuer ce travail nous avons opté pour une solution implémenté en python appelé HILDA[2]. [1] Nous avons choisis d'utiliser la version 1.01 et avons modifié le code source original qui comportait quelques erreur d'implémentation. L'analyseur HILDA procède en 4 étapes distinctes :

1. Tout d'abord, le texte est segmenté en EDUs.
2. Ensuite, l'étape de labélisation des relations évalue l'appartenance d'un certain type de relations entre des EDUs consécutifs avec un maximum de vraisemblance. Les deux EDUs les plus plus probablement connectés par une relation rhétorique sont alors fusionnés en une structure rhétorique arborescente de deux EDUs.
3. Puis, de façon itérative, l'étape de labélisation est à nouveau appliquée afin de ré-évaluer quelles relations sont les plus vraisemblables entre deux structures rhétorique arborescentes de n'importe quelle taille en incluant une structure atomique composée d'un seul EDU.
4. Cette procédure est répétée jusqu'à ce que toutes les structures rhétoriques arborescentes aient été fusionnées.

Une fois le texte analysé, nous obtenons un arbre exploitable par n'importe quel langage.

1. HILDA est un analyseur de texte et permet de segmenter le texte en EDU puis de fournir un arbre RST. HILDA est disponible à l'adresse : http ://www.cs.toronto.edu/ weifeng/software.html

3.1.2 Transformation de l'arbre RST en un arbre DEP

Le parser Hilda nous permet d'obtenir un fichier *.tree* contenant sous forme textuelle un arbre RST dont les feuilles sont des EDUs et les noeuds des relations de discours. En revanche, l'algorithme de résolution pour le résumé se fonde sur un problème connu sous le nom de Tree Knapsack Problem (TKP) qui utilise un arbre Dependency based Discourse Tree (DEP-DT) dont les noeuds sont des EDUs. Notre arbre RST doit donc être transformé en un arbre DEP. Nous avons ainsi implémenté l'algorithme énoncé dans notre synthèse bibliographique à la page 13 (3.1.2 Algorithme).

La classe python RST_DT est composée de deux méthodes. La méthode *load* parse le fichier généré par le parser HILDA afin de charger l'arbre RST en mémoire en utilisant la bibliothèque treelib comme structure de donnée. Dans l'implémentation de cette méthode, deux arbres sont créés : l'arbre RST ayant pour noeuds non EDUs le type de relation de discours et pour feuilles les EDUs, et l'arbre RST_NS ayant pour noeuds son type (N ou S) en fonction de la relation de discours du noeud père. Puis la méthode *toDEP*, qui implémente l'algorithme précédent, renvoie l'arbre DEP en utilisant treelib. L'implémentation de cette dernière est expliquée ci-après.

1. Tout d'abord un parcours en largeur de l'arbre RST permet de récupérer l'identifiant et la profondeur de chaque noeud et prépare ainsi la création de l'arbre Head.
2. Ensuite un arbre Head de la même structure que RST est construit en partant de la plus grande profondeur et associe à chaque noeud non EDU/feuille son Head.
3. Puis l'arbre DEP est initialisé. C'est à dire qu'il a pour racine le Head de la racine de l'arbre Head, et cette racine a pour fils toutes les feuilles/EDUs de l'arbre RST.
4. Pour finir, tous les autres noeuds de l'arbre Head, qui sont en fait des identifiants vers les EDUs de l'arbre RST, sont évalués selon la dernière étape de l'algorithme de notre synthèse. Si le noeud associé dans l'arbre RST_NS est de type N, alors un parcours en largeur est effectué dans ce même arbre jusqu'à trouver un noeud de type S associé à un head dans l'arbre Head qui ne soit pas un EDU/feuille dans l'arbre RST, alors on modifie le père de ce noeud dans l'arbre DEP en enfant du head du parent de ce noeud de type S trouvé. Sinon le noeud est de type S et le père de ce noeud devient le head du parent de ce noeud.

3.1.3 Formulation du problème et résolution

La classe TKP permet de formuler le problème sous la forme d'un Tree Knapsack Problem (TKP) défini dans notre synthèse bibliographique (3.2.1 Définition d'un TKP). Sa méthode *load* utilise la classe RST pour parser le fichier obtenu en sortie du parser Hilda et pour générer l'arbre DEP. Puis, celle-ci applique des *stopwords* du corpus de nltk aux EDUs contenus dans chaque

noeuds de l'arbre DEP. Puis, sa méthode *solve*, qui prend un argument L le nombre d'EDUs à conserver, utilise la bibliothèque PuLP de programmation linéaire pour modéliser le problème sous forme mathématique. Enfin, sa méthode *summary* retourne le résumé de texte sous la forme d'une liste python d'EDUs conservés.

3.1.4 Classe Summarize

La classe Summarize utilise les trois précédents modules afin de générer un résumé.

```
usage: summarize.py [-h] -l LENGTH [-o OUTPUT_FILE] -i INPUT_FILE

optional arguments:
  -h, --help            show this help message and exit
  -l LENGTH, --length LENGTH
                        max number of words
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        output file
  -i INPUT_FILE, --input_file INPUT_FILE
                        input file
```

3.2 Evaluation du résumé avec le calcul du Score ROUGE

3.2.1 Classe Rouge

La classe Rouge permet d'évaluer notre résumé en appelant le logiciel *ROUGE-1.5.5* (rouge-1.5.5.pl) et en générant au préalable les fichiers utilisés par celui-ci. Ce logiciel fournit un score en comparant pour un même texte des résumés générés par une méthode et des résumés générés manuellement. Le premier fichier généré par notre classe est *settings.xml* contenant les chemins d'accès vers les différents résumés dans un format spécifique et très peu robuste. En effet, aucune méthode de génération et d'utilisation de ces fichiers n'est spécifiée dans la documentation. Aussi, l'extension *.html* n'est visiblement pas en cohérence avec le contenu du fichier qui n'est pas aux normes xml (exemple : id=1 au lieu de id="1"). De plus, Rouge n'utilise pas un parser XML pour parser ces fichiers. En effet, si un espace est absent entre les deux balises *a* spécifiant un EDU, le logiciel *ROUGE-1.5.5* retourne un score 0 sans afficher d'erreur de parsing. Ainsi, la méthode *createSystems* permet de convertir notre résumé dans ce format spécial, la méthode *createModels* permet de convertir les résumés manuels, la méthode *createSettings* permet de créer le fichier de configuration et la méthode *evaluation* permet d'effectuer un appel système au logiciel *ROUGE-1.5.5* et affiche ainsi le score ROUGE.

```
usage: summaries2rouge.py [-h] -s SYSTEMS [SYSTEMS ...] -m MODELS ...
    [MODELS ...] path2rouge path
```

```
positional arguments:
  path2rouge            path to the directory ROUGE-1.5.5
  path                  path to the directory used for the ...
      generation of the rouge files

optional arguments:
  -h, --help            show this help message and exit
  -s SYSTEMS [SYSTEMS ...], --systems SYSTEMS [SYSTEMS ...]
                        path to the systems generated summary files
  -m MODELS [MODELS ...], --models MODELS [MODELS ...]
                        path to the reference summary files
```

3.2.2 Classe Evaluation

Le module EvaluationApp contient la classe Evaluation qui génère les différents résumés des textes de références du corpus RST-DT. Celle-ci fait appel à la classe Rouge pour générer les scores pour chaque texte et au module rouge2csv afin de calculer les scores globaux de rappel, de précision et de f-mesure en agglomérant les scores obtenus précédemment.

```
usage: evaluation.py [-h] [-p PATH2SUMMARIZE]

optional arguments:
  -h, --help            show this help message and exit
  -p PATH2SUMMARIZE, --path2summarize PATH2SUMMARIZE
                        path to the file summarize.py
```

4 Architecture de l'application

4.1 Installation de Hilda

Afin d'installer HILDA,il suffit de télécharger les sources sur http ://www.cs.toronto.edu/ weifeng/software.html. Nous avons utilisé la version 1.01. Hilda demande les corpus nltk suivants : wordnet ic, verbnet. Une fois Hilda installé, nous appelons sa classe principale afin de parser le texte et de le transformer en rst dt. Notre code python "summarize.py" contient les lignes suivantes : rst parser = DiscourseParser(), rst parser.parse(). Une fois le rst dt créé, nous le sauvegardons dans le répertoire du fichier original.

4.2 Cheminement vers DEP

Notre fichier va ensuite appeler un outil que nous avons développé afin de transformer le rst dt en dep dt. Pour cela la classe DEP, permet contient tout les outils et les fonctions expliqués précédemment afin de construire l'arbre DEP. Nous pouvons maintenant résoudre le problème de maximisation TKP.

4.3 Solveur ILP

Afin de construire un sous arbre maximisant la fonction que nous avons décrite plus haut nous avons utilisé une librairie permettant de résoudre un problème Integer Linear Problem. Cette librairie est disponible à l'adresse : https ://pypi.python.org/pypi/PuLP. Nous avons donc définis le problème TKP en émettant des contraintes linéaires. Voici le code de la section :

4.4 Évaluation avec le score ROUGE

L'évaluation des résumés avec le calcul du score ROUGE est effectuée par le module externe EvaluationApp décrit brièvement ci-après.

- EvaluationApp -

- evaluation.py récupère le nombre d'EDUs selectionnés pour chaque résumés manuels de RST-DT, et appelle summaries2rouge.py, puis rouge2csv.pl
- src contient le code source des différents modules utilisés.
  - summaries2rouge
    - summaries2rouge.py permet de générer les fichiers formatés des résumés (models, systems, settings.xml) et encapsule ROUGE-1.5.5.
  - rouge2csv
    - rouge2csv.pl permet de convertir les scores ROUGE générés par summaries2rouge en fichier csv avec des scores de rappel, précision et f-mesure.
  - ROUGE-1.5.5
    - ROUGE-1.5.5.pl permet de calculer le score rouge.
    - data contient des stopwords et les exceptions de WordNet-1.6 et WordNet-2.0
- corpus
  - RST-DT
    - EXT-EDUS-30 contient 30*5 fichiers qui sont des résumés manuels à partir de 30 textes issus du corpus RST-DT
  - summaries contiendra à l'exécution les 30*5 résumés générés par notre méthode.
  - txt contient les 30 textes originaux extrait du corpus RST-DT.
- rslt contiendra à l'exécution 30*5 dossiers, générés par summaries2rouge.py, contenant les résumés sous un certain format afin d'être interprété par ROUGE-1.5.5.pl
  - ROUGE1_scores.txt contiendra les scores en sortie de summaries2rouge.py
  - ROUGE1_score.csv contiendra le score en sortie de rouge2csv.pl
  - ROUGE2_scores.txt contiendra les scores en sortie de summaries2rouge.py
  - ROUGE2_score.csv contiendra le score en sortie de rouge2csv.pl

4.5 Architecture complète et utilisation

Le code source du projet est contenu dans le dossier discourse/src. Le fichier parse.py contient le code de HILDA modifié. le fichier Summarize.py permet d'effectuer le résumé de texte. Il prend un texte en entré et enregistre le résumé dans le même dossier que le texte sous la forme de NAME.EXT.sum. Ce fichier fait appel aux fichier DEP.py, qui contient le code pour transformer un rst dt en dep dt, tkp.py qui permet de résoudre un problème TKP en utilisant une maximisation avec ILP. Afin d'appeler le fichier il suffit d'écrire : python Summarize.py -l 100 FILE. Où FILE = le fichier à résumer et 100 = L = le nombre maximum de mots.
Les autres fichiers sont des fichiers du parser HILDA que nous n'avons pas modifié.

5 Expérience

5.1 Recherche de résultats à comparer

Un des articles scientifiques dont nous nous sommes inspirés pour implémenter une méthode de résumé automatique utilisant l'analyse de discours (Yoshida et al., 2014) [6] effectue l'évaluation d'une nouvelle méthode en utilisant le corpus RST-DT (Carson et al., 2002) [1] composé de 385 articles du Wall Street Journal annotés, dont 30 sont résumés manuellement. Les chercheurs ont utilisé les 30 documents comme documents de référence pour l'évaluation du résumé, et ont utilisé les 355 autres comme données d'apprentissage pour le parser Hilda.

Yoshida et al. ont comparé différentes méthodes qui diffèrent dans la manière dont elles convertissent l'arbre RST en arbre DEP. La méthode de référence étant TKP-GOLD qui utilise les arbres RST gold du corpus RST-DT sans appliquer le parser Hilda. Puis, deux autres méthodes qui utilisent leur parser transformant un texte en arbre DEP sans passer par un arbre RST. Enfin, une dernière (TKP-HILDA) utilise le parser. Puis ils comparent ses méthodes en fonction de leurs scores ROUGE-1 et ROUGE-2 de rappel (ROUGE Recall). Ils comparent aussi les trois dernières méthodes en fonction de la moyenne de F-Score (Average F-value), de la précision de la racine (Root Accuracy RA), et de la moyenne de la précision de dépendance en résumé (Dependency Accuracy in Summary DAS). Ils en déduisent que la méthode TKP-DIS-DEP, qui n'utilisent pas le parser Hilda, donne les meilleurs résultats.

5.2 Comparaison des résultats

Nous avons sélectionné les résultats de deux méthodes décrites par Yoshia et al. (2014) [6], à savoir TKP-DIS-DEP considérée comme la plus performante et TKP-HILDA que nous avons implémenté dans notre projet. La méthode TKP-Hilda utilise 355 articles du Wall Street Journal annotés du corpus RST-DT pour entraîner le parser Hilda fonctionnant avec SVM, tandis que notre

méthode TKP-Hilda-Default utilise les paramètres par défaut du parser, car nous n'avons pas eu en notre possession les articles utilisés pour l'entraînement.

Afin de pouvoir évaluer notre méthode, il a fallu récupérer les 30 résumés du corpus RST-DT, les nettoyer afin d'obtenir les textes originaux, appliquer notre méthode pour obtenir les résumés automatiques, et pour finir appliquer notre module d'évaluation en spécifiant les chemins d'accès vers 5 résumés de références pour chaque texte. Nous noterons tout de même qu'il nous a fallu faire varier le nombre de mots sélectionnés pour chaque texte afin de comparer des résumés de même taille en nombre de mots, puis nous avons calculé une moyenne des scores obtenus en utilisant le module externe EvaluationApp.

	TKP-DIS-DEP	TKP-Hilda	TKP-Hilda-Default
Recall ROUGE-1	0.319	0.284	?
Recall ROUGE-2	0.109	0.093	?
Avg F-value	0.532	0.415	?
RA	0.933	0.733	?
Avg DAS	0.847	0.596	?

TABLE 1: Comparaison des résultats d'expériences

Etant donné que nous avons obtenu des résultats supérieurs (0.521 de score de Recall ROUGE-1), notre système d'évaluation diffère visiblement de celui décrit brièvement dans l'article de Yoshia et al. Ainsi, il nous est impossible de comparer notre méthode de façon scientifique. Toutefois, nous pouvons attester "visuellement" de l'efficacité de notre méthode de résumé qui semble donner des résultats satisfaisants.

6 Conclusion

Dans ce rapport, nous avons présenté l'implémentation d'une méthode extractive de résumé automatique de texte. Notre application prend en argument un fichier à résumer et un nombre maximal de mots, puis rend un fichier résumé contenant une suite d'EDUs sélectionnés. Nous avons aussi développé une méthode d'évaluation à partir du logiciel ROUGE et du corpus RST-DT en nous inspirant d'un article scientifique du domaine.

Références

[1] Lynn Carlson, Mary Ellen Okurowski, Daniel Marcu, Linguistic Data Consortium, et al. *RST discourse treebank*. Linguistic Data Consortium, University of Pennsylvania, 2002.

[2] Hugo Hernault, Helmut Prendinger, Mitsuru Ishizuka, et al. Hilda : a discourse parser using support vector machine classification. *Dialogue & Discourse*, 1(3), 2010.

[3] Tsutomu Hirao, Yasuhisa Yoshida, Masaaki Nishino, Norihito Yasuda, and Masaaki Nagata. Single-document summarization as a tree knapsack problem. pages 1515–1520, 2013.

[4] William C Mann and Sandra A Thompson. Rhetorical structure theory : Toward a functional theory of text organization. *Text*, 8(3) :243–281, 1988.

[5] Nale Dipali Agrawal Roshani Patil Aarti, Pharande Komal. Automatic text summarization. *International Journal of Computer Applications*, 109, 2015.

[6] Yasuhisa Yoshida, Jun Suzuki, Tsutomu Hirao, and Masaaki Nagata. Dependency-based discourse parser for single-document summarization. EMNLP, 2014.