TP : détection de messages indésirables (*spam*)

Gábor Bella, Yannis Haralambous
IMT Atlantique

Introduction

Le but de ce TP est d'explorer la tâche de détection automatique de messages indésirables à travers des méthodes d'apprentissage automatique. Il s'agit d'un cas particulier d'un ensemble de problèmes connus sous le nom de *classification binaire* appliquée au texte : pour chaque message reçu, il faut émettre une prédiction binaire correspondant aux catégories indésirable–désirable, ou *spam–non-spam*.

Données

Nous allons utiliser deux corpus (ensembles de messages) en anglais, pré-annotées :

— `sms_spam.csv` : 5 532 messages courts (textos) ;
— `email_spam.csv` : 4 196 courriers électroniques (emails).

Ces deux fichiers (zippés), ainsi que tous les autres nécessaires pour ce TP, sont téléchargeables à partir de

`https://github.com/gbella/NLP/tree/main/SPAM`.

Exercice 0 : Exploration des données (10 minutes)

Téléchargez les données et observez-les. La colonne `label` contient la catégorie : 1 pour les spam, 0 pour les non-spam. Quelques autres considérations :

— quels sont les points communs et les différences entre les textos et les emails ?
— Quels sont les points communs et les différences entre les spam et les non-spam ?
— Y a-t-il des propriétés (*features*) typiques pour les spam ? Un exemple : la présence de numéros de téléphone, adresses mail, ou sites Web que le récipient est invité à consulter. Est-ce que ces propriétés sont présentes systématiquement ? Sont-elles présentes également au sein de messages désirables ?

Exercice 1 : classification de textos par régression logistique (30 minutes)

Pour les exercices suivants, nous aurons besoin de Python 3 ainsi que de quelques bibliothèques spécifiques pour le TAL, qui s'installent à partir de la ligne de commande comme suit :

```
pip install numpy
pip install pandas
pip install nltk
pip install scikit-learn
```

Une fois ces bibliothèques installées, ouvrez dans l'éditeur de texte de votre choix le code Python `spam_classification_exercise.py`.

Commençons par la fin : comment évaluer la performance de notre solution ? Une première idée pourrait être de compter le nombre de fautes que nous faisons par rapport à la vérité terrain. Cette mesure s'appelle *exactitude* (*accuracy* en anglais) :

$$\text{exactitude} = \frac{\text{nombre de prédictions correctes}}{\text{nombre total des prédictions}}.$$

Cependant, toutes les fautes n'ont pas la même gravité : « oublier » de détecter un spam (ce que l'on appelle un *faux négatif*) est probablement moins grave que mettre à la poubelle un message authentique (*faux positif*). Pour cette raison, nous faisons appel à des mesures plus utiles, très fréquemment utilisées en TAL, à savoir la *précision* et le *rappel* (*recall* en anglais) :

$$\text{précision} = \frac{\text{vrai positifs}}{\text{vrai positifs} + \text{faux positifs}}$$

$$\text{rappel} = \frac{\text{vrai positifs}}{\text{vrai positifs} + \text{faux négatifs}}$$

où un *vrai positif* est un spam correctement identifié, un *faux positif* est un message authentique identifié par erreur comme étant du spam, et un *faux négatif* est un spam non détecté par erreur. La précision nous indique la proportion de messages authentiques que nous avons mis à la poubelle par erreur, alors que le rappel nous dit la proportion de spam correctement identifiés. La mesure F_1, finalement, est calculée comme la moyenne harmonique de la précision et du rappel :

$$F_1 = \frac{2 \times \text{précision} \times \text{rappel}}{\text{précision} + \text{rappel}}$$

Complétez les quatre méthodes `accuracy_score`, `precision_score`, `recall_score` et `f1_score` au sein du code source. Exécutez le code :

```
python spam_classification_exercise.py
```

Observez les résultats obtenus et sauvegardez-les pour plus tard. Est-ce que ces résultats seraient acceptables « dans le monde réel » ?

Finalement, analysez le code pour comprendre la solution implémentée, qui consiste en les étapes suivantes classiques de toute chaîne de traitement d'apprentissage automatique en TAL :

1. lecture de données annotées ;
2. partitionnement des données pour l'entraînement et l'évaluation ;
3. pré-traitement sur les données ;
4. extraction de propriétés ;
5. entraînement d'un modèle d'apprentissage automatique ;
6. évaluation de la performance du modèle entraîné.

Le partitionnement des données sert à réserver une petite partie des données (20% dans notre cas) pour l'évaluation de la solution : ces données ne seront pas employés pour l'entraînement.

Le pré-traitement comprend, en général, des opérations qui simplifient les données d'entrée afin d'améliorer la performance de l'apprentissage automatique. Dans notre cas, il s'agit de trouver les mots individuels (*tokenisation*) et d'éliminer les *stop words*, à savoir les mots les plus fréquents qui ne sont pas informatifs quant à la nature du message, tels que les articles ou les prépositions.

L'extraction de propriétés vise à trouver au sein d'un texte les informations pertinentes pour l'apprentissage automatique. Nous avons choisi une méthode classique, à savoir le calcul de TF–IDF. L'idée est que les mots employés dans les spam sont, de manière générale, différents de ceux des messages authentiques. La mesure TF–IDF sert à faire émerger les termes les plus caractéristiques d'un message. Le calcul est réalisé par la classe `TfidfVectorizer` de la bibliothèque `scikit-learn`. Pour chaque message, il s'agit de calculer un vecteur dont chaque élément représente la valeur TF–IDF d'un des mots qui y apparaît. La longueur du vecteur est un paramètre du système : nous avons choisi 500, ce qui signifie que pour l'entraînement nous ne considérerons que les 500 mots les plus fréquents de notre corpus.

Finalement, nous implémentons l'entraînement à travers une des méthodes d'apprentissage automatique les plus simples, à savoir la *régression logistique*.

Exercice 2 : classifications de mails (5 minutes)

Nous allons appliquer la même méthode de régression logistique au corpus `email_spam.csv`. Il suffit de commenter les lignes de code correspondant à l'exercice 1 et dé-commenter celles de l'exercice 2 : c'est uniquement la lecture des données qui change, le reste de la chaîne de traitement est le même.

Observez les résultats et comparez-les à ceux de l'exercice 1. Avez-vous des hypothèses pour expliquer la différence ?

Exercice 3 : classification interdomaine (5 minutes)

Nous allons procéder à l'expérience suivante : que se passe-t-il si l'on applique le modèle entraîné sur les textos à la détection de mails indésirables ? Ceci correspond à un cas d'utilisation fréquent, appelé *classification interdomaine*, où les modèles entraînés sont réutilisés sur des données plus ou moins différentes, faute de possibilité d'entraîner un modèle adapté.

Dé-commentez les lignes correspondantes à cette exercice (et commentez les autres). Cette fois-ci, nous utiliserons 100% des textos pour l'entraînement, et 100% des emails pour l'évaluation.

Observez les résultats. La perte de performance observée est typique lorsqu'un modèle est appliqué à des corpus sensiblement différentes de celui de l'entraînement.

Exercice 4 : corpus combinés (5 minutes)

Nous allons fusionner les deux corpus afin d'entraîner un modèle plus grand et plus robuste, qui s'applique aux deux types de messages. Pour le faire, il suffit de dé-commenter les lignes correspondant à cette exercice.

La performance obtenue est remarquablement élevée, étant donné l'extrême simplicité de notre solution. Pour autant, est-ce que de tels résultats seraient suffisants en pratique ?

Exercice 5 : BERT + *finetuning* (30 minutes)

Dans les exercices précédents, nous avons entraîné des modèles d'apprentissage automatique à partir de zéro, c'est-à-dire fondés sur des données annotées que nous avons fournies. Ici, nous profiterons de BERT, un très gros modèle pré-entraîné par Google sur des machines ultra-puissantes. Le modèle `bert-base-cased` que nous allons utiliser est un réseau neuronal de type *transformer*, pré-entraîné d'une façon non-supervisée sur de vastes corpus de textes en anglais.

La technique de *finetuning* consiste à ajouter des couches neuronales supplémentaires à des réseaux profonds de large échelle tels que BERT afin de les adapter à des tâches particulières. En effet, de nombreuses expériences ont démontré que les informations linguistiques « de sens commun », apprises par les modèles tels que BERT, peuvent être exploitées pour améliorer la performance d'une multitude d'applications de traitement automatique des langues. Le plus souvent, le processus d'adaptation à ces applications s'effectue d'une manière supervisée, à l'aide de corpus annotés de taille petite à moyenne. Autrement dit, en étendant un modèle large existant par un corpus relativement petit, on obtient des performances compétitives avec des modèles plus onéreux à annoter (en termes de travail humain). Le prix à payer est un besoin d'infrastructure de calcul beaucoup plus puissante.

Nous allons donc procéder au *finetuning* de BERT à l'aide de nos deux corpus fusionnés. Étant donné la taille de BERT et donc le coût de cette opération en terme de puissance de calcul, nous allons profiter de la disponibilité de GPU sous *Google Colab*. **Un compte Google sera nécessaire pour exécuter ce qui suit.**

Allez sur `http://colab.research.google.com` et faites *upload* sur le *notebook* `SPAM.ipynb` préalablement téléchargé. Activez l'usage du GPU à partir du menu *Runtime / Change runtime type / Hardware accelerator*. Puis exécutez tout en lançant *Runtime / Run all*. L'exécution devrait prendre environ huit minutes en total (avec le nombre d'*epochs* = 6).