# INF442-PI9 - Rapport CHAABOUNI OGIER

May 21, 2020

# 1 INF442-PI9 | GDPR in practice: data anonymization


### 1.0.1 ~ Amine Chaabouni et Valentin Ogier


## 1.1 1. Introduction

Ce projet présente l'application de méthodes d'analyse de données à un problème d'anonymisation : l'objectif est de censurer les noms de personnes dans un texte donné par l'utilisateur, et ce de manière tout à fait automatique. Ce problème est celui de la classification : on doit pouvoir distinguer entre la classe "nom de personne" et la classe formée par tous les autres mots de la langue du texte. Plus précisemment, on s'intéresse ici au problème de la *reconnaissance d'entités nommées* (NER, de l'anglais *Named-entity recognition*) : dans un texte non structuré, on va faire ressortir des "entités" (i.e. un groupe de mots) en les catégorisant dans des classes ("personne" ou "non-personne" d'abord, puis "personne", "lieu", "organisation", etc ensuite).

On dispose pour cela de plusieurs jeux de données :

- `eng.train.conll` : un ensemble de phrases en anglais annoté, les annotations identifient certains groupes de mots comme des personnes, des lieux, ou autres. Ce jeu de donnée servira à l'entraînement de nos méthodes.
- `eng.testa.conll` et `eng.testb.conll` : deux jeux de données de *test* sur lesquels nous pourront évaler la performance de nos méthodes.

L'objectif de ce projet est donc d'implémenter des méthodes d'analyse de données qui seront entraînées sur le premier jeu de données afin de repérer les noms de personnes dans les jeux de données de *test*. Chaque mot du texte étant vu comme une observation, nos méthodes seront des classifieurs : elles attribueront à chaque observation un label, et ces labels ne peuvent prendre qu'un nombre fini de valeur.

## 1.2 2. Description du problème

### 1.2.1 2.1 Reformulation

Ce problème de reconnaissance d'entités nommées peut se décomposer en deux étapes :

1. Transformation du texte en un ensemble de vecteurs, chaque vecteur correspondant à un mot du texte.
2. Entraînement d'un classifieur sur cet ensemble de vecteur.

Le succès de l'étape 2. dépend de la qualité de l'étape 1. : il faut non seulement que le représentant vectoriel d'un mot capture suffisament bien son sens, et son contexte, mais que la position relative de ces représentants dans l'espace vectoriel les englobants permettent de le comparer entre eux.

Le modèle de langage BERT constitue aujourd'hui une des méthodes les plus performantes pour effectuer la transformation 1. Les jeux de données décrits précédemment ont été transformés par BERT, et stockés sous forme de fichier binaire pouvant être chargé depuis Python et C++. Nous présenterons également dans ce rapport des transformations plus simples que nous avons implémentées nous-même.

Pour réaliser l'étape 2, nous avons implémenté des classifieurs classiques de science des données, en prenant en compte les contraintes du problème.

Nous analyserons dans ce rapport les méthodes que nous avons implémentés : les classifiers *k-nearest-neighbors* binaire et multiclasse, le classifieur binaire par régression logistique, et le classifieur multiclasse par régression logistique multinomiale. Nous les comparerons ces méthodes, et nous détaillerons les choix d'implémentation que nous avons effectués

#### 1.2.2 2.2 Les contraintes du problème

Deux contraintes principales apparaissent lors de l'étude du problème :

1. Les jeux de données ont des tailles considérable. Ils sont encore suffisament petit pour tenir intégralement en mémoire RAM : charger la version du jeu de données `eng.train.conll` transformée par BERT (à partir du fichier binaire `train.hdf5`) rempli à lui seul environ 1 GiB de mémoire RAM.
2. Le ratio du nombre d'observations catégorisées "I-PER" sur le nombre total d'observations est faible dans le jeu de données d'entraînement ($\approx 5\%$). Le nombre d'observations est encore plus faible pour "I-LOC", "I-MISC" et "I-ORG".

Enfin, une remarque pratique sera importante pour l'évaluation des performances de nos classifieurs : la mesure pertinente est le taux de détéction $TD$ :

$$TD = \frac{N(\text{positifs correctement identifiés})}{N(\text{positifs dans jeu de données})} \tag{1}$$

En effet, dans un scénario d'anonymisation, le but est de retirer toutes les mentions de noms de personnes, quitte à avoir des faux positifs.

### 1.3 3. Aspects techniques

#### 1.3.1 3.1 Fichier `README.md`

Le fichier `README.md` dans le dossier principal de ce projet détaille comment préparer votre système pour y compilier notre projet. Il décrit également l'organisation des dossiers de ce projet, et explique également comment lancer les exemples C++, et le exemples Python.

#### 1.3.2 3.2. Architecture hybride Python / C++

Notre projet présente une architecture hybride :

- un *back-end*, partie du programme qui fait le gros des calculs et de la gestion de la mémoire, implémentée dans une librairie C++ appelée `libinfo9`.
- un *front-end* Python qui présente certaines fonctions et certains objets de la librairie `libinfo9` sous la forme de fonctions et d'objets Python dans une librairie Python appelée `info9`.

Cette architecture permet d'utiliser les fonctions que nous avons implémentées en C++ dans un *Jupyter Notebook*, par exemple. Le *back-end* C++ peut également être appelé depuis des programmes C++ traditionnels.

L'architecture Python / C++ de notre projet est rendue possible par l'utilisation de la librairie `pybind11` : celle-ci permet de créer des *bindings* Python pour une librairie C++ existante. Les *bindings* ainsi générés prennent la forme d'une librairie qui peut être importée par un script Python de manière transparente (i.e. le script Python n'a aucune connaissance de la librarie `pybind11` ; il contient simplement la ligne `import info9`).

La librairie C++ `libinfo9` (en dehors d'un fichier `wrap.cpp` qui permet de générer les bindings) n'a elle-même aucune connaissance de la librairie `pybind11` : elle peut être compilée et utilisée par un programme C++ standard.

La grande force de `pybind11` est donc de découpler très largement la partie C++ de la partie Python, tout en permettant des appels de fonction C++ par Python de façon peu coûteuse. Notre projet utilise notamment un mécanisme très intéressant de `pybind11` qui consiste à unifier deux types :

- les `numpy.ndarray` de dimension 2 de la librairie `numpy` du côté Python
- les `Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>` de la librairie `Eigen` du côté C++, qui correspondent à une version "row-major" des `Eigen::MatrixXd`, qui sont elles stockées en "column-major". Par souci de simplicité, ce type a été renommé `RMatrixXd` dans notre code.

En pratique, il est possible de faire transiter ces types entre la librairie C++ et le script Python **sans copie** ! De même, les `numpy.ndarray` de dimension 1 et les `Eigen::VectorXd` sont unifiés. `pybind11` permet également de gérer facilement la durée de vie des objets partagés (ou déplacés) entre C++ et Python en faisant coopérer les `std::shared_ptr` avec le *refcounting* de Python.

**Exemple :** La fonction suivante définie dans le code C++ :

```
RMatrixXd add(Eigen::Ref<RMatrixXd const> const &a, Eigen::Ref<RMatrixXd const> const &b)
    return a + b;
}
```

pourra être appelée par un script Python sans copie de ses arguments :

```
import info9
import numpy as np
A = np.array([1.,2.,3.])
B = info9.add(a, a) # zero-copy!
```

### 1.3.3 3.3. Stockage des données

Les jeux de données ont été convertis au format HDF5. Ce format de fichier binaire est très bien supporté depuis Python par la librairie h5py qui permet d'y lire des données sous la forme de `numpy.ndarray`, et en C++ par la librairie HighFive, qui permet d'y lire des données sous la forme d'objets `Eigen`.

### 1.3.4 3.4. Librairies externes

Notre implémentation utilise quelques librairies externes : + C++: - `pybind11` : pour générer une librairie Python à partir de la librairie C++. - `Eigen` : calcul vectoriel et matriciel. - `ANN` : implémentation de kd-trees pour le problème de la recherche des $k$ plus proches voisins. - `OpenMP` : le simple fait de *linker* `OpenMP` dans notre librairie active la parallélisation de certaines opérations matricielles de la librairie `Eigen`. - `HighFive` : lecture/écriture de fichiers au format HDF5. + Python: - `numpy` : calcul vectoriel et matriciel. - `pandas` : statistique sur des tableaux de données, ici simplement utilisée pour manipuler des fichiers csv. - `h5py` : lecture/écriture de fichiers au format HDF5.

## 1.4 4. Problème de la classification binaire

Commençons par nous intéresser au problème de la classification binaire (problème 5.1 de l'énoncé). On considére seulement deux classes dans notre jeu de de données : "I-PER" d'une part (marqué par 1), et tous le reste d'autre part (marqué par 0). On va ici travailler sur les jeux de données (textes annotés) préalablement transformés par BERT en des ensembles de vecteurs dans un espace de dimension 1024.

### 1.4.1 4.1 Préparation

Cette partie du notebook charge tous les paquets nécessaires à son exécution, pour cette partie et pour les suivantes

[1]:
```
# Charger la librairie créée pour ce projet.
# Avant d'exécuter cette ligne, suivre les instructions du README.md
# pour l'installer dans votre environnement.
import info9
```

[2]:
```
# Charger les librairies externes, et des fonctionnalités de la
# librairie standard Python
from pathlib import Path

import numpy as np
import pandas as pd
import h5py
```

[3]:
```
# Variables utiles dans la suite du rapport

# Ensemble des labels IOB
IOB_LABELS = ["O", "B-MISC", "I-MISC", "B-PER", "I-PER", "B-ORG", "I-ORG",␣
 ↪"B-LOC", "I-LOC"]

# Emplacement des jeux de données transformés par BERT
# Un fichier HDF5 peut contenir plusieurs jeu de donnés (appelés `datasets`).
# Ces fichiers contiennent deux datasets:
#     - "representation" : représentations vectorielles des mots du texte.
#     - "true_labels" : entiers correspondant aux labels IOB des mots du texte.
ftrain_bert = Path("../data/train.hdf5")
```

```
ftesta_bert = Path("../data/testa.hdf5")
ftestb_bert = Path("../data/testa.hdf5")
```

Les vecteurs représentants les mots, et leurs labels seront stockés dans un objet `Dataset`, qui sera créé à partir d'un fichier HDF5 par la fonction suivante:

[4]:
```
def read_dataset(fdataset, *, row_limit=None, projecter=None,␣
↪label_of_interest=None,):
    # Build Dataset containing row_limit rows of vectors, and the same number␣
↪of labels.
    # Optionnaly project the datapoints with `projecter` before building the␣
↪Dataset.
    # If label_of_interest is defined, the labels will be saved as binary in␣
↪the Dataset:
    #    -  1 if label == label_of_interest
    #    -  0 in other cases
    hf = h5py.File(fdataset, "r")
    datapoints = hf["representation"]
    labels = hf["true_labels"]
    if row_limit:
        datapoints = datapoints[:row_limit]
        labels = labels[:row_limit]
    if projecter:
        datapoints = projecter.project(datapoints)
    if label_of_interest:
        labels = np.array(np.asarray(labels, dtype=int) == label_of_interest,␣
↪dtype=int)
    return info9.Dataset(datapoints, labels)
```

### 1.4.2 4.2 Classifieur K-nn binaire

Le classifieur K-nn binaire est défini dans le fichier `KnnClassificationBinary.hpp`. Son implémentation est très proche de celle qui a été proposée lors du TD6 du cours INF442. Nous allons le tester avec les paramètres suivants:

[5]:
```
# Number of neighbors to consider
k_nn = 10
# Ratio of positive neighbors needed to predict that the point has a positive␣
↪label
positive_threshold = 0.2
```

Remarquons que nous n'avons pas pris le seuil de décision (`positive_threshold`) égal à 0.5 : le nombre de points négatifs étant très supérieur au nombre de points positifs, on peut considérer que les points positifs sont plus significatifs que les points négatifs. Cette idée sera développée dans la partie *Classifieur K-nn multilabel*.

#### Sans projection

[6]:
```
training_dataset = read_dataset(ftrain_bert, row_limit=10000)
```

[7]:
```
%%time
classifier = info9.KnnClassificationBinary(k_nn, training_dataset,␣
 ↪positive_threshold)
training_dataset = None
```

```
CPU times: user 201 ms, sys: 20.6 ms, total: 222 ms
Wall time: 221 ms
```

La phase d'entraînement (qui se réduit à la construction d'un kd-tree) est très rapide.

[8]:
```
test_dataset = read_dataset(ftesta_bert, row_limit=130, label_of_interest=4)
```

[9]:
```
%%time
confusion_matrix = classifier.estimate_all(test_dataset)
#print(confusion_matrix.PrintEvaluation())
```

```
CPU times: user 1.46 s, sys: 4.83 ms, total: 1.46 s
Wall time: 1.46 s
```

Sans projection, la classification est lente même avec un "petit" nombre de voisins à considérer (ici 20000) : l'estimation sur 100 vecteurs du jeu de données test prend presque deux secondes sur mon PC. On a choisi de na pas afficher la matric de confusion de ces estimations : le nombre d'estimation étant très faible, les résultats obtenus sont peu pertinents.

**Avec projection** Pour diminuer le coût des estimations, on va se placer dans un espace de plus petite dimension en projetant les vecteurs de dimension 1024 représentant les mots dans un espace de dimension 128. L'objet utilisé est défini dans la classe `RandomProjection.hpp`.

[10]:
```
random_projecter = info9.RandomProjection(1024, 128, "Rademacher")
```

[11]:
```
training_dataset = read_dataset(ftrain_bert, row_limit=10000,␣
 ↪projecter=random_projecter, label_of_interest=4)
```

[12]:
```
%%time
classifier = info9.KnnClassificationBinary(k_nn, training_dataset,␣
 ↪positive_threshold)
training_dataset = None
```

```
CPU times: user 47.2 ms, sys: 8.24 ms, total: 55.5 ms
Wall time: 51.1 ms
```

Sur les 130 premières lignes, pour comparer avec le cas sans projection :

[13]:
```
test_dataset = read_dataset(ftesta_bert,␣
 ↪row_limit=130,projecter=random_projecter, label_of_interest=4)
```

[14]:
```
%%time
confusion_matrix = classifier.estimate_all(test_dataset)
#print(confusion_matrix.PrintEvaluation())
```

```
CPU times: user 505 ms, sys: 452 µs, total: 505 ms
Wall time: 476 ms
```

Sur un exemple plus significatif :

[15]:
```
test_dataset = read_dataset(ftesta_bert,␣
↪row_limit=1000,projecter=random_projecter, label_of_interest=4)
```

[16]:
```
%%time
confusion_matrix = classifier.estimate_all(test_dataset)
print(confusion_matrix.PrintEvaluation())
```

```
                Predicted
                0       1
Actual  0       900     24
        1       4       72

Error rate              0.028
False alarm rate        0.025974
Detection rate          0.947368
F-score                 0.837209
Precision               0.75

CPU times: user 3.49 s, sys: 4.86 ms, total: 3.49 s
Wall time: 3.44 s
```

[17]:
```
# Unload memory
train_dataset = test_dataset = confusion_matrix = None
```

**Analyse :** Faire des estimations sur le jeu de données projeté est bien plus rapide : on a gagné un facteur 5 en temps d'exécution. Le classifieur est plutôt performant en terme de qualité des prédictions, même en travaillant sur des données projetées dans un espace de dimension 128.

En terme de performance, par contre, ce classifieur est très mauvais : il n'est pas utilisable sur des jeux de données de la taille de ceux considérés dans ce projet ! Prédire les 1000 labels en utilisant 10000 points de références du jeu de données d'entraînement prend environ 3s sur mon système ; prédire les labels des 150000 lignes du jeu de données test en utilisant les 200000 points de référence est inenvisageable.

Le choix de $k$ est important pour obtenir de bonnes performances prédictives. Les méthodes de *cross-validation* permettent d'explorer les valeurs possibles pour l'hyperparamètre $k$, et d'évaluer de manière fiable les performances de classifieurs pour les différentes valeurs de $k$. Etant donné les limitations pratiques du classifieur K-nn en terme de temps de calcul, et ce même pour des valeurs de faibles $k$, du nombre d'observations pour l'entraînement, du nombre d'estimations à réaliser, et de la dimension des observations, nous avons décidé de ne pas suivre cette voie, et de plutôt essayer

des méthodes différentes.

### 1.4.3 4.3 Classifieur binaire par régression logistique

Un deuxième classifieur envisageable est le classifieur binaire par régression logistique. Etant donné la faiblesse du classifieur k-nn sur notre jeu de données, cette proposition est pertinente : l'étape d'estimation est en effet très peu coûteuse pour la régression logistique ($O(d)$ où $d$ est la dimension de l'espace des observations). Au contraire, l'étape coûteuse va être l'entraînement du classifieur : sera-t-elle trop coûteuse pour notre jeu de données relativement large ? Pour répondre à cette, nous allons devoir considérer les différentes implémentations possibles de cette étape d'entraînement.

**Commentaires sur les choix effectués** Le classifieur `LogisticReg` est défini dans le fichier `LogisticReg.hpp`. L'étape d'entraînement y est effectuée par minimisation de la fonction de coût régularisée $J(\beta)$ par descente de gradient. Les formules, et une partie des notations utilisées dans notre implémentation sont calqués sur celles utilisées sur cette page du cours *Machine Learning* d'Andrew Ng.

Les choix d'implémentations de la descente de gradient se sont révélés être d'une importance fondamentale, et ont nécessité du travail pour assurer :

- que la méthode converge rapidement : itérer avec des pas de taille trop petite n'est pas efficace.
- que la méthode est stable : itérer avec des pas de taille trop grande ne permet pas à la méthode de converger, voire la pousse à diverger.

Nous avons implémenté différentes méthodes de descente de gradient, qui correspondent toute à un compromis :

1. `fit_gd` : descente de gradient simple avec un pas de taille fixée $\alpha$.
   - $\oplus$ simple à implémenter.
   - $\ominus$ pas généralement trop petit au début (convergence lente), et trop grand à la fin (instable).
   - $\ominus$ itération sur toutes les observations à chaque pas : coût $O(Nd)$
2. `fit_newton` : méthode de Newton-Rhapson. La hessienne est inversée à l'aide de la librairie Eigen.
   - $\oplus$ assure une convergence quadratique sous des conditions faibles.
   - $\ominus$ inverser la hessienne est très coûteux ($\approx O(N^3)$).
3. `fit_sgd` : *stochastic gradient descent*, à chaque pas on estime le gradient sur un nombre réduit d'observations prises au hasard.
   - $\oplus$ coût du pas ne dépend plus du nombre d'observations, mais seulement de la dimension des observations ($O(d)$).
   - $\oplus$ *minibatching* : on a pris soin de vectoriser les opérations sur les *batchs* de 512 instances prises au hasard, en utilisant des opérations Eigen.
   - $\oplus$ relativement simple à implémenter.
   - $\ominus$ même problème de choix de la taille du pas que la descente de gradient simple.

De ces trois solutions, la méthode stochastique est la plus prometteuse : les deux autres ne sont plus applicables lorsque le nombre d'observation $N$ devient grand (en pratique dès $N > 10000$). Le choix de la taille du pas reste problématique : il serait intéressant d'avoir un grand pas pour

les premières itérations, et un plus petit pas à mesure qu'on s'approche de l'optimum. Nous avons donc décidé d'implémenter une méthode de descente de gradient stochastique à pas adaptatif.

De nombreuses méthodes, appelées "accélérateurs de convergence", peuvent être associés à la descente de gradient pour en faire varier le pas à chaque itération. Les notes de cours de Marc Lelarge, nous ont permis d'avoir un aperçu de ces méthodes. Nous avons choisi d'implémenter la méthode RMSProp, qui constitue un bon compromis entre difficulté d'implémentation (en particulier avec des opérations vectorisées) et performance. La page dédiée du cours *Dive into deep learning* nous a servi de référence. On obtient en fin de compte la méthode

4. `fit_sgd_rmsprop` :
   - $\oplus$ stochastique, et donc dont le coût d'un pas ne dépend pas du nombre $N$ d'observations.
   - $\oplus$ avec *minibatching*, donc utilisant des opérations vectorisées.
   - $\oplus$ avec pas adaptatif, et ce par coefficient du gradient, donnant une convergence rapide et stable.
   - $\ominus$ avec une implémentation relativement complexe.

### Démonstration et comparaisons

[18]:
```
# Constante de régularistation
LAMBDA = 1
```

[19]:
```
# On charge ici toutes les lignes en mémoire.
train_dataset = read_dataset(ftrain_bert, label_of_interest=4)
test_dataset = read_dataset(ftesta_bert, label_of_interest=4)
```

### 1.Descente de gradient simple

[20]:
```
%%time
classifier = info9.LogisticReg(train_dataset, LAMBDA, decision_threshold=0.5)
classifier.fit_gd(epsilon=0.75, alpha=0.01)
print(f"J(beta) = {classifier.J():.2f}\n")
```

```
J(beta) = 55.37

CPU times: user 4.17 s, sys: 16.6 ms, total: 4.19 s
Wall time: 4.15 s
```

[21]:
```
%%time
cm = classifier.estimate_all(test_dataset)
print(cm.PrintEvaluation())
```

```
                Predicted
                0       1
Actual  0       143974  585
        1       6090    3318

Error rate              0.0433534
False alarm rate        0.00404679
Detection rate          0.352679
```

```
F-score                   0.498535
Precision                 0.850115

CPU times: user 174 ms, sys: 104 µs, total: 174 ms
Wall time: 173 ms
```

### 2.Descente de gradient simple par méthode de Newton-Rhapson

Il faut prendre un plus petit jeu de données pour pouvoir tester cette méthode.

[22]:
```
smaller_train_dataset = read_dataset(ftrain_bert,␣
 ↪row_limit=400,label_of_interest=4)
classifier = info9.LogisticReg(smaller_train_dataset, LAMBDA,␣
 ↪decision_threshold=0.5)
smaller_train_dataset = None
```

[23]:
```
%%time
classifier.fit_newton(epsilon=1)
print(f"J_small(beta) = {classifier.J():.2f}\n")
```

```
J_small(beta) = 0.02

CPU times: user 3.05 s, sys: 10.2 ms, total: 3.06 s
Wall time: 2.83 s
```

### 3.Descente de gradient stochastique avec mini-batching

[24]:
```
%%time
classifier = info9.LogisticReg(train_dataset, LAMBDA, decision_threshold=0.5)
classifier.fit_sgd(epsilon=1, alpha=0.1)
print(f"J(beta) = {classifier.J():.2f}\n")
```

```
J(beta) = 8.49

CPU times: user 3.08 s, sys: 3.48 ms, total: 3.08 s
Wall time: 3.06 s
```

### 4. Descente de gradient stochastique avec mini-batching + accélérateur de convergence RMSProp

[25]:
```
%%time
classifier = info9.LogisticReg(train_dataset, LAMBDA, decision_threshold=0.5)
classifier.fit_sgd_rmsprop(epsilon=0.1)
print(f"J(beta) = {classifier.J():.2f}\n")
```

```
J(beta) = 7.03

CPU times: user 3.99 s, sys: 10.9 ms, total: 4 s
Wall time: 3.98 s
```

[26]:
```
%%time
cm = classifier.estimate_all(test_dataset)
print(cm.PrintEvaluation())
```

```
                Predicted
                0       1
Actual  0       144250  309
        1       1956    7452

Error rate              0.0147109
False alarm rate        0.00213754
Detection rate          0.792092
F-score                 0.868076
Precision               0.960186

CPU times: user 180 ms, sys: 0 ns, total: 180 ms
Wall time: 179 ms
```

[27]:
```
train_dataset = test_dataset = cm = classifier = None
```

**Comparaison :**

Un réal atout de la régression logistique est mis en avant ici : sa très bonne performance pour la prédiction, une fois qu'il a été entraîné (≈ 200ms sur mon PC pour estimer les labels des 150000 vecteurs du jeu de données `testa`).

En ce qui concerne la phase d'entraînement, on a ici choisi des paramètres pour obtenir une durée d'entraînement de l'ordre de 3s ; ce choix a été fait pour garder l'interactivité du notebook, mais ne serait pas justifié pour une situation d'entraînement réel. On peut en effet comparer la performance des différentes méthodes en comparant la valeur de la fonction de coût régularisée $J(\beta)$ après l'entraînement : les méthodes étant entraînées sur les mêmes données (sauf pour la méthode de Newton, trop lente), la fonction de coût est la même, et une plus petite valeur de $J(\beta)$ indique donc une plus grande proximité avec le $\beta_{\text{opt}}$ qui minimise le coût. Pour 3s de temps de calcul, on obtient le tableau suivant :

| Méthode | J($\beta$) |
|---|---|
| Fixed-step GD | 55.4 |
| Newton | *échec* |
| Stochastic GD | 8.5 |
| Stochastic GD + RMSProp | 7.1 |

On obtient bien les gains de performances espérés avec les méthodes stochastiques. Par ailleurs, d'un point de vue expérimental, nous avons observé une bien plus grande stabilité de la méthode stochastique avec RMSProp par rapport à la méthode stochastique en les essayant avec différents paramètres sur les jeux de données.

Enfin, les performances en terme de prédiction du classifieur ainsi obtenu sont excellentes : presque 80% de taux de détéction.

## 1.5 5. Problème de la classification multilabel

En passant par la classification binaire, on "jette" une partie de l'information dont on dispose sur le jeu de données d'entraînement. Serait-il possible d'obtenir de meilleure performance en revenant au problème plus général de la reconnaissance d'entités nommés ? Nous allons maintenant commenter nos implémentations de classifieurs multiclasses (problème 5.3 de l'énoncé).

### 1.5.1 5.1. Classifieur K-nn multilabel

Le classifieur K-nn multilabel classique suit le même principe que le classifieur K-nn binaire : il détermine la classe d'un point en effectuant un vote majoritaire sur les K plus proches voisins de ce point. On retrouve ici le problème posé par le désequilibre des effectifs des classes : la classe "O" étant très largement plus grande que les autres, elle risque de gagner tous les votes majoritaires. Traiter la classe "O" séparément n'est pas raisonnable : la classe "I-PER" est elle-même plus de deux fois plus large que la classe "I-MISC" dans le jeu de données `testa`.

Une solution qui traite toute les classes de la même manière consiste à donner moins de poids aux classes les plus grandes en divisant la valeur d'un vote par l'effectif total de la classe, i.e. à normaliser à 1 la valeur totale des votes de chaque classe :

$$p(c \in k) = \sum_{n \in K_{nn}(c)} \frac{\mathbf{1}\{n \in k\}}{|k|}$$

La valeur des votes cumulés des membres de la classe $k$ pour le point $c$ est égal au nombre de votants de la classe $k$ parmi les voisins de $c$ divisé par l'effectif total de la classe $k$.

C'est cette solution que nous avons utilisée dans notre implémentation de la classe `KnnClassificationMulticlass`.

**Démonstration :**

Nous avons vu que le classifieur K-nn est peu performant lorsque les observations sont dans un espace de grande dimension ; on va donc présenter l'utilisation du classifieur K-nn multilabel directement sur un jeu de donnée projeté.

[28]:
```
k = 20
random_projecter = info9.RandomProjection(1024, 128, "Rademacher")
```

[29]:
```
# Cette fois-ci, le jeu de données contient des labels prenant des valeurs dans␣
↪{0, ..., 8}
train_dataset = read_dataset(ftrain_bert, row_limit=10000,␣
↪projecter=random_projecter)
test_dataset = read_dataset(ftesta_bert, row_limit=1000,␣
↪projecter=random_projecter)
```

[30]:
```
classifier = info9.KnnClassificationMulticlass(k, train_dataset, IOB_LABELS)
```

[31]:
```
%%time
cm = classifier.estimate_all(test_dataset)
print(cm.PrintMatrix())
```

```
        O       B-MISC  I-MISC  B-PER   I-PER   B-ORG   I-ORG   B-LOC   I-LOC
O       446     7       75      0       72      0       83      0       103
B-MISC  0       0       0       0       0       0       0       0       0
I-MISC  0       0       7       0       1       0       0       0       3
B-PER   0       0       0       0       0       0       0       0       0
I-PER   0       0       1       0       70      0       3       0       2
B-ORG   0       0       0       0       0       0       0       0       0
I-ORG   1       5       4       0       3       0       61      0       3
B-LOC   0       0       0       0       0       0       0       0       0
I-LOC   3       0       2       0       3       0       15      0       27

CPU times: user 2.77 s, sys: 6.86 ms, total: 2.78 s
Wall time: 2.76 s
```

[32]:
```
print("One vs all for class \"I-PER\"")
print(cm.OneVsAllConfusionMatrix(4).PrintEvaluation())
```

```
One vs all for class "I-PER"
                Predicted
                0       1
Actual  0       845     79
        1       6       70

Error rate              0.085
False alarm rate        0.0854978
Detection rate          0.921053
F-score                 0.622222
Precision               0.469799
```

[33]:
```
# Unload data
train_dataset = test_dataset = classifier = cm = None
```

**Analyse :** Le classifieur K-nn multilabel présente les même caractéristiques que le classifieur K-nn binaire :

- il est très lent pour la prédiction, et nécessite que les données soit projetées dans un espace de petite dimension pour avoir un temps d'exécution raisonnable.
- la qualité de ses prédictions est bonne, mais pas excellente.

Sur un même jeu de données (les 10000 premières lignes des données d'entraînements, et les 1000 premières lignes des données de test), le classifieur multilabel est marginalement plus lent sur mon système (de 4%). Comparons la qualité de leurs prédictions pour la classe "I-PER" seule :

| | Binaire | Multilabel |
|---|---|---|
| Error rate | 3% | 7% |
| False alarm rate | 2% | 6% |
| Detection rate | 93% | 81% |
| F-Score | 84% | 63% |

| | Binaire | Multilabel |
|---|---|---|
| Precision | 75% | 50% |

Il semble donc que le classifieur K-nn binaire est plus performant pour discriminer la classe “I-PER” des autres classes que le classifieur multilabel.

### 1.5.2 5.2. Régression logistique multinomiale

La formulation de la régression logistique pour deux classes se généralise pour plusieurs classes en faisant apparaître la fonction *softmax* :

$$softmax(\mathbf{z}) = \frac{[\exp(z_1)\,,\,\ldots\,,\,\exp(z_k)]}{\sum_{i=1}^{k}\exp(z_i)}$$

Les notations utilisées dans notre code, ainsi que la dérivations de la fonction de perte, et de son gradient, proviennent de ce cours de l'Université de Buffalo. Nous avons ajouté à cette fonction de perte un terme de régularisation.

Ayant constaté le succès de la méthode stochastique avec RMSProp (`fit_sgd_rmsprop`) pour la régression logistique binaire, nous avons décidé d'implémenter cette méthode pour la version multinomiale. Afin d'éviter l'introduction de trop de complexité d'un seul coup, nous avons commencé par en implémenter une version à pas fixe (`fit_sgd`).

Ici encore, nous mettons en oeuvre du minibatching pour améliorer les performances.

**Démonstration :**

[34]:
```
# Constante de régularisation
LAMBDA = 1.
```

[35]:
```
# On charge ici toutes les lignes en mémoire. Ici, les labels prennent leur␣
 ↪valeur dans {0, ..., 7}
train_dataset = read_dataset(ftrain_bert)
test_dataset = read_dataset(ftesta_bert)
```

[36]:
```
%%time
classifier = info9.LogisticRegMultinomial(train_dataset, LAMBDA, IOB_LABELS)
classifier.fit_sgd(epsilon=1,alpha=0.1)
print(f"J(W) = {classifier.J():.2f}\n")
```

```
J(W) = 24.14

CPU times: user 16.2 s, sys: 53.1 ms, total: 16.3 s
Wall time: 16.2 s
```

[37]:
```
%%time
classifier = info9.LogisticRegMultinomial(train_dataset, LAMBDA, IOB_LABELS)
classifier.fit_sgd_rmsprop(epsilon=1)
```

```
print(f"J(W) = {classifier.J():.2f}\n")
```

```
J(W) = 21.94

CPU times: user 12.4 s, sys: 42.9 ms, total: 12.4 s
Wall time: 12.3 s
```

[38]:
```
cm = classifier.estimate_all(test_dataset)
print(cm.PrintMatrix())
```

```
        O       B-MISC  I-MISC  B-PER   I-PER   B-ORG   I-ORG   B-LOC   I-LOC
O       127711  0       66      0       204     0       150     0       66
B-MISC  0       0       12      0       0       0       0       0       0
I-MISC  714     0       2394    0       54      0       228     0       402
B-PER   0       0       0       0       0       0       0       0       0
I-PER   1209    0       18      0       8076    0       63      0       42
B-ORG   0       0       0       0       0       0       0       0       0
I-ORG   576     0       177     0       90      0       5103    0       330
B-LOC   0       0       0       0       0       0       0       0       0
I-LOC   303     0       129     0       69      0       246     0       5535
```

[39]:
```
print("One vs all for class \"I-PER\"")
print(cm.OneVsAllConfusionMatrix(4).PrintEvaluation())
```

```
One vs all for class "I-PER"
                Predicted
                0       1
Actual  0       144142  417
        1       1332    8076

Error rate              0.0113596
False alarm rate        0.00288464
Detection rate          0.858418
F-score                 0.902296
Precision               0.950901
```

**Analyse :** Comme pour la version binaire, les méthodes stochastiques nous permettent d'entraîner notre classifieur sur l'ensemble des observations du jeu de données d'entraînement. L'ajout d'un optimiseur de convergence, dont l'implémentation est plus complexe que pour la version binaire, est très bénéfique : il permet à la fois d'accélerer et de stabiliser la convergence.

On obtient en fin de compte un classifieur multiclasse très performant (>90% detection rate pour chaque classe) après une période d'entraînement très courte (seulement quelques secondes !). En observant les résultats de *one-vs-all* pour "I-PER", on constate que pour la régression logistique, on arrive aussi bien à distinguer "I-PER" des autres dans le contexte multilabel que dans le contexte binaire.

## 1.6 6. Conclusion

Dans ce projet, nous avons étudié l'utilisation de techniques de science des données pour identifier les noms de personnes dans un texte, et plus généralement d'y reconnaître des entités. Notre rapport se termine par une belle réussite : la régression logistique multinomiale s'est révélée être très performante, à la fois sur le plan du temps de calcul, que sur celui de la qualité des prédictions.

La question qui reste ouverte dans ce rapport est celle de la transformation du texte à anonymiser en un ensemble de vecteurs. Les techniques simples que nous avions mises en place n'ayant pas donné de résultats probants, nous avons choisi de nous concentrer ici sur notre travail pour la deuxième étape, celle de l'entraînement d'un classifieur. Ce travail a d'abord été un travail de sciences des données au sens de sciences des méthodes statistiques, mais il nous a également donné l'occasion d'en explorer d'autres facettes : calcul haute performance, méthode d'optimisation, interfaçage entre les langages C++ et Python, par exemple. Il nous aura également donné l'occasion de nous plonger dans les cours, exemples, et projets partagés au sein de la communauté des *data scientists*. Dans ce contexte, une méthode d'analyse de données est toujours imparfaite : parmi les multiples variantes, extensions, choix d'implémentation, elle constitue un compromis. Ce rapport aura été pour nous l'occasion de voir les limites et les atouts de certaines de ces méthodes, et d'aboutir à un compromis qui réponde à nos attentes.

## 1.7 7. Références

[1] : Le cours *Machine Learning* d'Andrew Ng, notamment la page "Exercice 5 - Regularization" pour la formulation de la régression logistique pour la méthode de Newton.

[2] : La documentation de scikit-learn pour sa page "Stochastic Gradient Descent" qui donne une formulation mathématique et des conseils d'implémentation pour la SGD. Egalement intéressant sur le sujet de la régularisation.

[3] : Le cours *Dive into Deep Learning* pour sa discussion sur les minibatchs pour la SGD.

[4] : Le cours *Deep learning: DIY!* de Marc Lelarge notamment le chapitre "Optimization for deep learning" pour les accélérateurs de convergence.

[5] : Le cours *Multiclass Logistic Regression* de Sargur N. Srihari, pour la formulation de la régression logistique multinomiale.