Rapport CR12:
Reconnaissance de locuteur

Antoine Plet
Thomas Sibut-Pinote

Janvier 2014

1 Introduction

Le but de ce projet était d'implémenter un programme de reconnaissance de locuteur, c'est-à-dire, à partir de données d'entraînement, d'identifier si une personne qui parle est l'un des sujets entraînés ou non.

Nous avons essayé de retrouver les résultats du papier "Kernel Based Text-Independent Speaker Verification" de Mariéthoz, Grandvalet et Bengio.

Nous avons commencé par coder un svm(6).

Puis nous avons codé le kernel perceptron pour pouvoir tester plusieurs noyaux, l'objectif (atteint(?) A CHANGER QUAND ON SAURA) étant d'arriver au noyau de Fisher(9).

Nous avons enfin fait diverses validations (10) sur des données générées (6.1.1) par nous-mêmes.

2 Particularités d'implémentation

Nous avons travaillé sur un dépôt github (`https://github.com/tomsib2001/speaker-recognition`).

Nous avons utilisé le langage Python qui nous était le plus familier. Cependant, la très utile bibliothèque Voicebox étant écrite en Matlab/Octave, nous avons dû interfacer Python avec Octave (`octaveIO.py`).

3 Répartition du travail

3.1 Antoine

3.2 Thomas

3.3 Ensemble

4 Choix des données

Nous avons pu disposer de 5 enregistrements de personnes lisant tous la même série de 12 phrases.

Nous nous sommes de plus enregistrés tous les deux.

Enfin, nous avons pris un enregistrement d'une personnalité politique ainsi que celui d'un imitateur, pour voir à quel point il arrivait à se faire passer pour le premier. Les résultats sont en 11.1.

5 Représentation des données

Conformément à ce que propose l'article, nous avons choisi de transformer les données en extrayant des caractéristiques discriminantes, dont les premières sont des coefficients d'une transformée discrète de cosinus (DCT), et les suivantes sont leur dérivée puis diverses fonctions du signal.

Pour cela, nous avons utilisé la librairie Voicebox (`http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html`).

6 SVM

6.1 SGD

Nous avons commencé par programmer utilisant la technique de la Descente de Gradient Stochastique en considérant le problème:

$$\begin{cases} \min\limits_{w,b,\xi_i} & \frac{1}{2}\|w\|^2 + C\sum\limits_{i=1}^{n}\xi_i \\ \text{avec} & y_i(w^\top x_i + b) \geq 1 - \xi_i, \qquad i = 1,\dots,n \\ \text{et} & \xi_i \geq 0, \qquad i = 1,\dots,n \end{cases} \tag{1}$$

On va appliquer la technique de la descente stochastique de gradient vue en cours, à la fonction convexe

$$f_{obj}(w) := \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} l_i(w)$$

où $l_i(w) = \max(0, 1 - y_i(w \cdot x_i + b)) = \text{hinge}(y_i(w \cdot x_i + b))$ est la fonction de coût. On calcule un subgradient de $\text{hinge}(x) = max(0, 1 - x)$:

$$\partial\text{hinge}(x)(h) = \begin{cases} 0 & \text{si } x > 1 \\ -\frac{h}{2} & \text{si } x = 1 \\ -h & \text{si } x < 1 \end{cases}$$

soit

$$\nabla\text{hinge}(x) = \begin{cases} 0 & \text{si } x > 1 \\ -\frac{1}{2} & \text{si } x = 1 \\ -1 & \text{si } x < 1 \end{cases}$$

On écrit en fait f_{obj} comme une fonction de $\tilde{w} = (w, b)$ (car b est un paramètre à trouver). On a $l_i(\tilde{w}) = max(0, 1 - y_i\langle\tilde{w}, (x, 1)\rangle)$ de sorte que

$$\nabla l_i(\tilde{w}) = \begin{cases} 0 & \text{si } \langle\tilde{w}, (x, 1)\rangle > 1 \\ -\frac{y_i}{2}(x, 1) & \text{si } \langle\tilde{w}, (x, 1)\rangle = 1 \\ -y_i(x, 1) & \text{si } \langle\tilde{w}, (x, 1)\rangle < 1 \end{cases}$$

et enfin

$$\nabla f_{obj}(w, b) = w + C \cdot \sum_{i=1}^{n} \nabla l_i(\tilde{w}).$$

6.1.1 Premiers Tests

Afin de tester "visuellement" notre implémentation [1] en Python, nous avons écrit un générateur de données en dimension 2 qui fabrique des points répartis selon deux lois de moyennes différentes μ_1 et μ_2 uniformes respectivement sur $[\mu_1 - 1, \mu_1 + 1]$ et $[\mu_2 - 1, \mu_2 + 1]$. Nous avons ensuite écrit une fonction qui affiche les points ainsi que la solution trouvée par la descente stochastique.

7 Kernel Perceptron

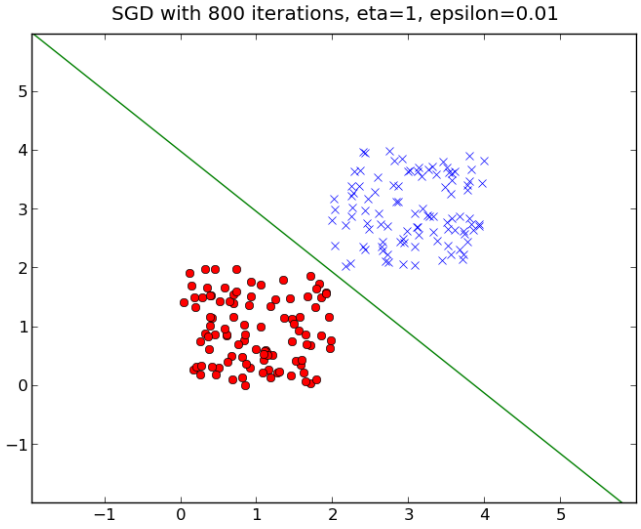
En vue de tester différents noyaux mentionnés par l'article, nous avons implémenté l'algorithme du Kernel Perceptron [2].

7.1 Premiers tests

Comme pour la SGD, nous avons testé le Kernel Perceptron sur des données générées avec le noyau trivial qui correspond à rester dans l'espace ambiant avec son produit scalaire) en dimension 2:

IMAGE GÉNÉRÉE PAR ANTOINE A INSERER ICI

1. `sgd.py`
2. `kernel_perceptron.py`



8 Noyau Gaussien

9 Noyau de Fisher

9.1 GMM

Le noyau de Fisher est construit à partir de modèles générateurs, dans notre cas, un GMM [3]. Celui-ci produit à partir des données d'entraînement des paramètres $\theta_0 = \{\mu_m^0, \sigma_m, \pi_m\}_{m=1}^M$ décrivant M gaussiennes [4] de moyenne μ_m et de variance σ_m, et qui sont une approximation de la distribution des données.

Dans notre cas, on réalise un GMM pour un M fixé (à partir de tests empiriques) sur des paquets de T « frames » [5].

Pour cela, nous avons d'abord utilisé la librairie `gaussmix` de Voicebox avant de passer à la fonction mixture de `Scikit-Learn` pour Python à cause de problèmes de terminaison.

9.2 Noyau de Fisher

Étant donné un échantillon $\bar{x}$, le vecteur des scores de Fisher est:

$$u_{\bar{x}} = \nabla_\theta \log p(\bar{x}|\theta)$$

3. Gaussian Mixture Model
4. C'est-à-dire plus précisément de vecteurs gaussiens, dans le cas général où les données sont en dimension k.
5. On rappelle qu'une frame représente une fenêtre de 20ms.

Alors le Kernel de Fisher est défini par:

$$K(\bar{x}_i, \bar{x}_j) = u_{\bar{x}_i}^T I(\theta)^{-1} u_{\bar{x}_j}$$

où $I(\theta)$ est la matrice d'information de Fisher, dont l'article affirme qu'on peut sans grande perte l'approximer par l'identité pour cette application. Nous ferons donc cette hypothèse et ne parlerons plus de cette matrice.

(EXPLICATIONS INTERMEDIAIRES)

Finalement, on a:

$$K(\bar{x}_i, \bar{x}_j) = \sum_{m=1}^{M} (n_{i,m}\Sigma_m^{-2}(\mu_m^i - \mu_m^0))^T (n_{j,m}\Sigma_m^{-2}(\mu_m^j - \mu_m^0))$$

avec

$$n_{i,m} = \sum_{t=1}^{T} P(m|x_i^t),$$

x_i^t étant la t-ième frame de x_i, $P(m|x_i^t)$ mesurant le taux d'appartenance de x_i^t à la m-ième gaussienne, donnée qui est fournie par le GMM.

10 Validation

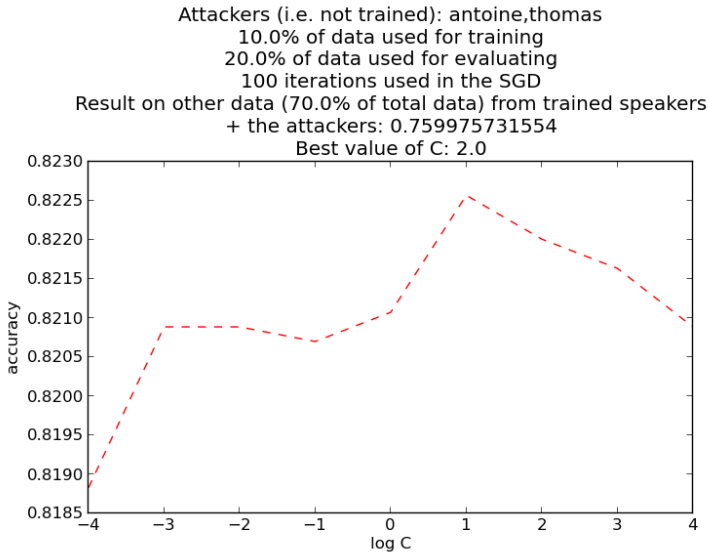
10.1 Validation croisée

Pour choisir la constante C la plus adaptée à un ensemble d'entraînement pour la technique de la SGD (6.1), nous avons utilisé le protocole suivant:

- On choisit un sous-ensemble S des locuteurs enregistrés.
- Pour chaque locuteur, on partage les données selon les ratios suivants:
  - r_1 vont dans l'ensemble d'entraînement;
  - r_2 vont dans l'ensemble de validation;
  - $1 - r_1 - r_2$ vont dans l'ensemble de « post-validation », accompagnés des locuteurs que l'on avait exclus à la première étape.
- Pour chaque C dans $2^{-k}, \cdots, 1, \cdots 2^k$, on calcule un w correspondant à la valeur de C, et on calcule le taux de réussite sur l'ensemble de validation.
- On en déduit une meilleure valeur de C, que l'on teste sur le troisième ensemble de « post-validation ». Cela permet entre autres de voir si le résultat trouvé résiste aux « attaques », c'est-à-dire des locuteurs qui n'ont jamais été vus ni traités.

Cela donne par exemple la courbe ci-dessus, pour $k = 4$ et les autres paramètres donnés sur le graphique:

On obtient 75.99% de réussite sur le troisième ensemble qui contient, en plus de données des locuteurs déjà entraînés, deux « attaquants ». Ici, la meilleure valeur de C est $2^1 = 2$.



11 Résultats

11.1 Imitateur