Approximation de la codéviance

Rémi Hutin Rémy Sun

23 Novembre 2015

Abstract

Dans ce rapport, nous rendrons compte de l'implémentation en Python de l'algorithme CM-sketch d'approximation de la codéviance, ainsi que de l'analyse statistique obtenue en appliquant l'implémentation à un triplet de traces réelles.

Mots-clés. Codéviance; CM-sketch

1 Outils de travail

1.1 Pourquoi Python?

Simplicité Tout d'abord, nous avons utilisé Python dans le but de fournir un code aussi clair et lisible que possible.

Structure de liste/tableau La structure de liste en Python a la particularité de permettre directement l'accès à n'importe quel élément, ce qui permet une implémentation plus comfortable des vecteurs de fréquence. Inversement, on pourrait dire que les tableaux Python ont une taille dynamique, ce qui est utile pour la "standardisation" de vecteurs.

Beaucoup de fonctions natives Python possède beaucoup de bibliothèques évitant d'avoir à implémenter nous même des fonctions comme le logarithme. Cela nous a également permis d'utiliser les expressions rationnelle de Python.

Typage dynamique Nous avons été ammenés à manipuler beaucoup de types différents, le typage de Python permet d'éviter des notations très lourdes, ce qui aurait été le cas avec un langage fortement typé comme Ocaml.

1.2 Parsage des traces

En l'état, les traces fournies sont difficiles à exploiter puisqu'il s'agit de lignes de textes (ce sont des logs). De fait, nous avons écrit une fonction de parsage en tirant avantage du fait que Python supporte la reconnaissance d'expressions rationnelles. Par exemple, le typage du protocole epahttp est fourni ci-dessous:

```
pattern["epahttp"] = re.compile("""
(?P<host>.*)
\s\[
(?P<D>[0-9]*)
:
(?P<H>[0-9]*)
:
(?P<M>[0-9]*)
:
(?P<S>[0-9]*)
\]\s"
(?P<request>.*)
"\s
(?P<code>[0-9]*)
\s
(?P<size>[0-9]*)
""", re.VERBOSE)
```

A partir de cette définition, on peut poser une fonction parseFile qui crée un vecteur qui découpe chaque ligne en différents blocs correspondant à une information différente. Nous nous sommes interessé dans ce document à l'adresse hôte ("host")

1.3 Fonction de hachage 2-Universelle

Il est nécessaire d'avoir une famille de fonctions de hachage 2-universelles. La famille retenue est celle des add-multiply-shift qui permet d'obtenir une fonction de hachage très rapidement.
Cette famille est définie dans une classe, de sorte à pouvoir créer une nouvelle instance dès qu'il faut utiliser une nouvelle fonction de hachage aléatoire :

```
class Hash:
    def __init__(self, w, M):
        self.w = w
        self.M = M
        self.a = random.randint(1, 1 << (w-1)) - 1
        self.b = random.randint(1, 1 << (w-M)) - 1

    def __call__(self, x):
        return (((2*self.a-1)*x+self.b) % (1 << self.w)) // (1 << (self.w-self.M))
```

1.4 Standardisation des flux parsés

Nous avons créé une fonction makestandard qui transforme les informations extraites au moment du parsage ("chaines de caractères") en des données numériques exploitables.

A chaque chaîne de caractère de la liste parsée, on associe un numéro, de sorte à pouvoir traiter le problème. On arrête aussi le parsage aux 25000 premières lignes. Cela n'a aucune influence sur l'algorithme, mais permet de réduire significativement le temps de calcul.

Si deux lignes possèdent les mêmes caractères sur un champ particulier, elles porteront la même étiquette puisqu'on parlera de la valeur.

2 Analyse numérique

2.1 Analyse préliminaire

Une analyse numérique préliminaire donne les résultats suivants:

X	distinct(X)	maxfreq(X)
epahttp	1318	243
sdschttp	75	12710
calgary	4	16380

Par la suite on supposera donc travailler sur des flux dont les éléments font partie d'un univers de taille 1318

X	Y	cod(X,Y)
epahttp	sdschttp	-185
sdschttp	calgary	177646
epahttp	calgary	-254

2.2 Analyse sur le CM-sketch

Evaluation sur 20 essais Une première évaluation des résultats sur 20 évaluations CM-sketch sont regroupées dans le tableau ci-dessous (avec k = 16, δ = 0.0001 ou 0.0000001).
On note que même si les résultats sont assez éloignés des valeurs réelles (ce qui était attendu), la "topologie", l'allure de la situation est préservée (position relative des résultats)

X	Y	Moyenne CM_sketch(X,Y, 16, 0.0001) 20 essais	Ecart type CM_sketch(X,Y, 16, 0.0001) 20 essais
epahttp	sdschttp	-248344	116404
sdschttp	calgary	12311366	55424
epahttp	calgary	-369948	114896

Evaluation sur 100 mesures Une évaluation plus fine a été effectuée avec 100 essais, k=64. On notera déjà que les estimations sont bien plus proches de la réalité car k est plus proche de la taille de l'univers (une évaluation en k=1318 donne d'ailleurs les valeurs de codéviance exacte). Nous avons cherché à observer l'effet d'une diminution du δ, censée augmenter la précision. Nous avons constaté une nette diminution de l'écart type.

X	Y	Moyenne CM_sketch(X,Y, 64, 0.0001) 100 essais	Ecart type CM_sketch(X,Y, 64, 0.0001) 100 essais	Ecart type CM_sketch(X,Y, 64, 0.000001) 100 essais
epahttp	sdschttp	-34199	17683	15939
sdschttp	calgary	3513236	670	325
epahttp	calgary	-45931	22742	19680

Remarque. *Une implémentation de l'algorithme distribué est proposée dans le code, bien qu'aucune application numérique n'en a été faite.*

3 Conclusion

En conclusion, l'algorithme d'approximation permet de minimiser le temps de calcul de la codéviance approchée, avec une influence des paramètres sur l'écart type confirmée par nos observations sur machine.
Par ailleurs, il aurait été intéressant de réaliser de nombreux tests, par exemple en découpant les entrées par plages horaires.