TP Brain Imaging : Convergence of the Spacenet classifier – Lilian Besson

> Veuillez trouver en pièce jointe le code Python pour le TD#11[1] du cours "Functional Brain Imaging" du Master MVA. Voici le rapport concernant ce sujet de TD#11.

Le code, rapport et les illustrations sont aussi distribuées sous licence libre (MIT), sur ce dépôt git sur Bitbucket :

`https://bitbucket.org/lbesson/mva16-tp-brain-imaging-nilearn`

Vulgarisation de l'expérience : Nous utilisons des données d'une expérience de neuro-imagerie datant de 2001 : 6 patients auxquels on a montré des petits dessins, de différentes catégories (un chat, une maison, un visage etc), [Haxby 2001]. Pendant qu'ils regardent ces dessins, l'activité de leur cerveau est enregistrée par un IRM fonctionnel, durant plusieurs sessions, et stockée sous forme d'"images".

Ensuite le but est de "lire dans leurs pensées" : prédire quelle catégorie ils regardaient (chat, maison, chaise, ciseau etc), selon l'apparence générale de la carte d'activité de leur cerveau. Ces données ont la forme de 216 "photos en 3D" de leur cerveau ($40 \times 64 \times 64$ voxels), qu'on sépare 144 photos et étiquettes, pour apprendre le modèle mathématiques, et 72 photos (sans étiquette) pour tester la prédiction, ce qui donne un score en % de réussite.

Le but de cette expérience numérique d'apprentissage supervisé est de confirmer que le cerveau moyen (des 6 patients) réagit différemment à des catégories différentes, et de démontrer l'efficacité de l'outil mathématique d'apprentissage (un classifieur "SpaceNet"). Par exemple, différencier "chat" vs. "maison" ou "visage" vs. "maison" fonctionne très bien (98% de réussite). Mais il sera plus difficile de différencier "bouteille" vs. "ciseau", ou "chaise" vs. "ciseau" (seulement 67% de réussite).

Remarques préliminaires

- Le TD se focalisait sur la convergence du classificateur SpaceNet, dont la documentation se trouve ici si nécessaire et un tutoriel se trouve là (sur nilearn.github.io).
- Le sujet de TD suggérait d'exécuter le code sur un serveur distant, ce que j'ai fait sur le serveur zamok[2], merci à l'association CRANS de l'ENS Cachan. J'ai utilisé Python[3] 3, installé avec miniconda v4.0.5.
- Le sujet utilisait les données de la base Haxby2001 : `http://data.pymvpa.org/datasets/haxby2001/`, issue de cet article [Haxby et. al., 2001]. Ces données sont obtenues en Python avec la fonction `nilearn.datasets.fetch_haxby` (environ 315 Mo).

Les exemples[4] ci-dessous montrent les résultats obtenus pour les questions 1, 2 et 3 du TP #11.

[1] `https://nilearn.github.io/auto_examples/02_decoding/plot_haxby_space_net.html`.

[2] Debian jessie, 16 Gb RAM, 8 cœurs à 3.2 GHz.

[3] Pour la reproductibilité : Python v3.5.1, IPython v4.1.1, nilearn v0.2.3, numpy v1.10.4, scipy v0.17, matplotlib v1.5.1, scikit-learn v0.17.1. Notez que mon code est normalement compatible avec Python v2.7+.

[4] Les autres graphiques sont tous dans le dossier `fig` fournit dans l'archive `zip` envoyé avec ce TP, notamment pour la question 2.

Question 1

Quelques détails concernant les paramètres des simulations:

- Comme suggéré, j'ai demandé au SpaceNetClassifier d'utiliser le nombre maximal de cœurs[5], en choisissant[6] `n_jobs = -1`.
- `nb_chunks_train = 7`, utilisé pour séparer les données en deux morceaux (entraînement, test), selon la condition `labels['chunks'] <= nb_chunks_train` ou `> nb_chunks_train`. Plus cette valeur est grande, plus on a de données étiquetées d'entraînement, et donc plus la précision sur les données de test sera grande (observation classique, c'est une tâche d'apprentissage supervisé). Pour information, avec cette valeur de `nb_chunks_train`, les quatre vecteurs de données `X_train, X_test, y_train, y_test` ont les dimensions suivantes :

Vecteur	dimensions
`X_train`	$(40, 64, 64, 144)$
`X_test`	$(40, 64, 64, 72)$
`y_train`	$(144,)$
`y_test`	$(72,)$

  Nous sommes effectivement dans un cas typique de "haute dimension" : 144 (resp. 72) données en dimension d'entraînement (resp. de test) de dimensions égales à $x \times y \times z = 40 \times 64 \times 64 = 163840$.
- J'ai testé à la fois la pénalisation "Graph-Net" et "TV-L1". Comme annoncé dans le sujet de TP, la première est plus rapide que la seconde ($3 \times +$), elles donnent en général la même précision, mais la seconde donne des "features maps" plus satisfaisante graphiquement (plus régulière, avec de grandes zones constantes, c'est bien une image type "cartoon"), cf. ci dessous.
- La première tâche correspondait à implémenter une expérience de classification binaire, implémentée par la fonction `oneExperiment`, qui accepte les paramètres suivants :

```
oneExperiment(X_train, X_test, y_train, y_test, background_img,  # ↩
    Data
                target1=b'face', target2=b'house',   # Binary pb
                SHOW=True, tol=1e-4, verbose=1, nb_chunks_train=7,
                use_graph_net=True, use_tv_l1=False) # Options
```

- Les "features maps" ont été affichées avec la fonction `nilearn.plotting.plot_stat_map`, et les paramètres `cut_coords=(-34, -16), display_mode="yz"`. Ci-dessous à la question 2 se trouvent quelques "features maps", pour différents problèmes de classification, et différentes pénalisations.

J'ai commencé par considérer le problème de classification binaire "facile", "house" vs. "face", puis un autre "facile", "house" vs. "cat"; et après j'ai regardé le problème annoncé comme plus difficile, "scissors" vs. "chair". On constate effectivement que la convergence est plus lente à atteindre, et que la précision obtenue est bien moins bonne (entre 95% et 98% pour les problèmes faciles, mais seulement 65% pour le plus différentes).

[5] Et pour ne pas sur-exploiter toutes les ressources du serveur multi-utilisateurs que j'ai utilisé, j'ai exécuté le script avec `nice -n 19`, pour sélectionner la priorité la plus faible, cf. le script `Makefile` joint avec le code.

[6] Notez que ce morceau de code doit être bogué, comme le processus utilisait quand même 100% des 8 cœurs si je demandais `n_jobs = 1` (un seul job, un seul cœur normalement).

Cf. l'article initial [Haxby et. al., 2001] pour plus de détails concernant les données Haxby2001. Voici quelques exemples de stimuli visuels montrés aux sujets de cette expérience, classés par catégorie ("house", "scissors", "cat", "chairs" etc) :

Figure 1: (Qu1) Stimuli visuels utilisés dans l'article [Haxby et. al., 2001].

Cette première expérience est implémentée par la fonction `experiment1()` de mon script, qui accepte comme argument `targets_to_try`, une liste de problèmes binaires à considérer.

Question 2

Dans ce question, on essaie plusieurs valeurs du paramètre `tol`, selon une grille décroissante, géométrique (mais non régulièrement espacée), avec 9 valeurs entre `1` et `1e-5` :

```
grid_tol = [1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0001, 1e-05]
```

J'ai uniquement essayé le problème de classification binaire le plus simple, "face" vs. "house", et uniquement la pénalisation "Graph-Net" (la plus rapide), pour garder un temps de calcul raisonnable (de l'ordre de 28 minutes).

Pour moins de valeurs de `tol` (`grid_tol = [1e-1, 1e-3, 1e-7]`), j'avais tenté les deux pénalisation "Graph-Net" et "TV-L1", mais cette dernière est trop lente (environ 85 minutes pour un problème binaire, trois valeurs de `tol` et les deux pénalisations). En fait, "TV-L1" devient très lente pour des tolérance trop petite. J'ai aussi constaté que sur ces données, avec ces paramètres, diminuer la tolérance au dessous de `1e-5` ne change pas *du tout* le score final, mais augmente (beaucoup) le temps de calcul (donc je m'arrête à `1e-5`).

Cette deuxième expérience est implémentée par la fonction `experiment2()` de mon script, qui accepte comme paramètres `grid_tol, target1, target2`, une grille de valeur de `tol` à consider, et un problème de classification binaire.

Notez que pour les deux premières courbes ci-dessous, l'axe x correspond à la tolérance, en échelle *anti-logarithmique* (`x = - np.log10(tol)`), `tol` *diminue* vers la droite.

Temps de calcul en fonction de la tolérance

De façon assez logique, le temps de calcul augmente strictement quand on diminue la tolérance : une faible `tol` demande une convergence plus précise et donc plus de calcul. De façon assez surprenante, le temps de calcul est *quasi-linéaire* en fonction de `-log(tol)`, pour les premières valeurs.

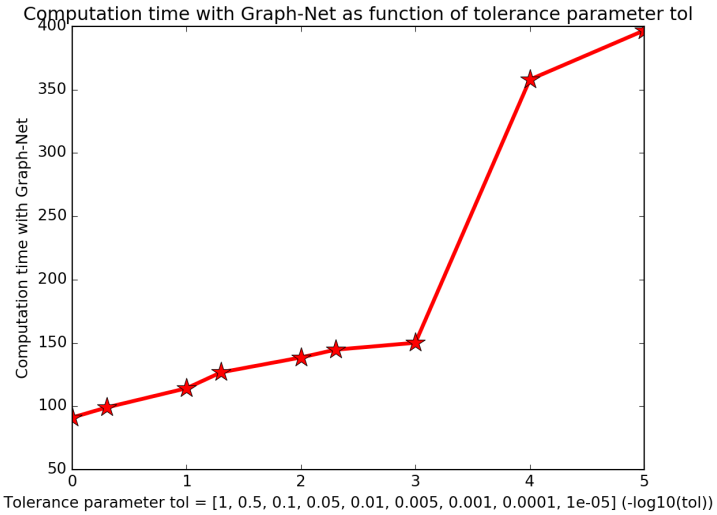


Figure 2: (Qu2) Temps de calcul en secondes, en fonction du paramètre `tol` (9 valeurs essayées).

Précision en fonction de la tolérance

Et de même, de façon très cohérente, la précision augmente (strictement) quand on diminue le paramètre `tol`. Notez que la précision est bornée, par 100% au maximum, et qu'il est irréaliste d'espérer atteindre 100%. Une précision de 96%–98% telle qu'obtenue pour `tol = 1e-4` est déjà excellente !

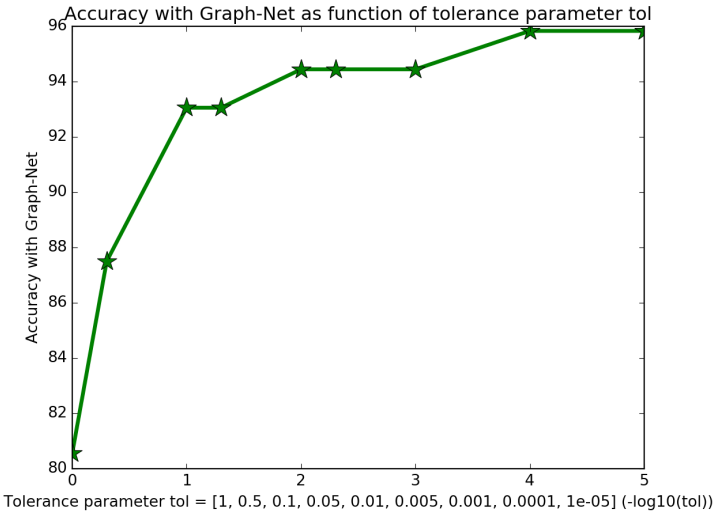
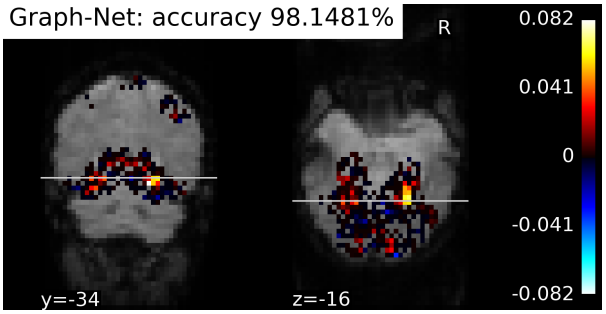


Figure 3: (Qu2) Précision en %, en fonction du paramètre `tol` (9 valeurs essayées).

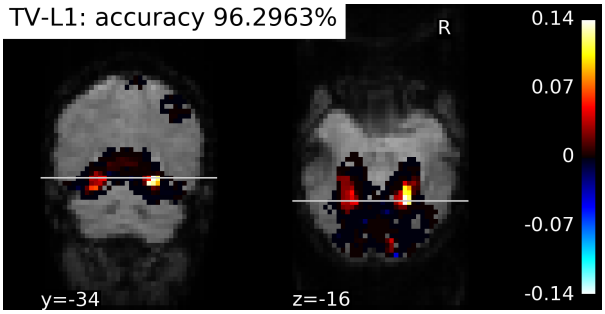
Quelques “features maps”

Voici quelques “features maps” pour différentes valeurs de ce paramètre `tol`, comparant la pénalisation “Graph-Net” et “TV-L1” :

“face” vs. “house” : Pour la même tolérance, “Graph-Net” donne un meilleur score que “TV-L1”, mis une feature map moins régulière.

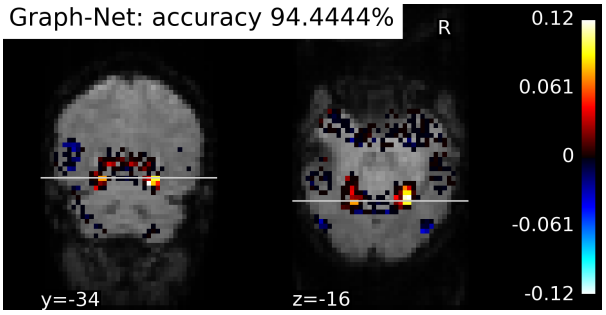


(a) (Qu2) Pénalisation “Graph-Net” et `tol = 1e-4` : précision 98.1%.

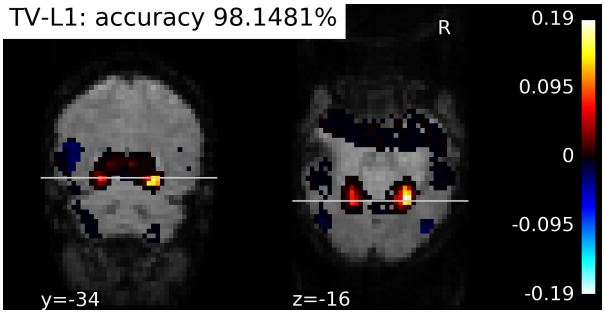


(b) (Qu2) Pénalisation “TV-L1” et `tol = 1e-4` : précision 96.3%.

“cat” vs. “house” : Pour la même tolérance, “Graph-Net” donne cette fois un moins bon score que “TV-L1”.

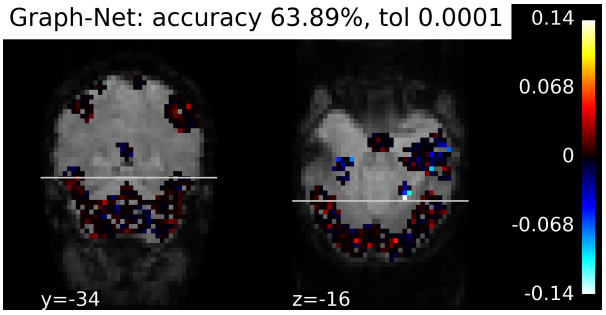


(a) (Qu2) Pénalisation “Graph-Net” et `tol = 1e-4` : précision 94.4%.

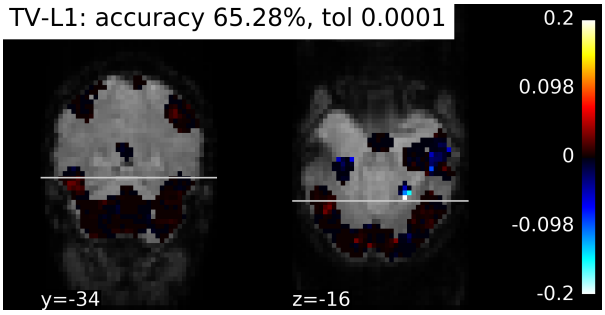


(b) (Qu2) Pénalisation “TV-L1” et `tol = 1e-4` : précision 98.1%.

“scissors” vs. “chair” : Problème plus difficile, la précision est bien moins satisfaisante :



(a) (Qu2) Pénalisation “Graph-Net” et `tol = 1e-5` : précision 63.9%.



(b) (Qu2) Pénalisation “TV-L1” et `tol = 1e-5` : précision 65.3%.

Question 3

Comme souvent en apprentissage statistique et en mathématiques appliquées, adapter ce paramètre `tol` demande un compromis ("trade-off") entre temps de calcul et qualité du résultat (comme observé dans les graphiques de la question 2) :

- Une trop grande valeur `tol` donnera une convergence rapide mais pas très précise;
- Et inversement, une trop petite valeur `tol` donnera une convergence très lente mais bien plus précise.

Une valeur par défaut pour `tol` ne peut pas être choisie pour n'importe quel problème. Dans certains cas, on aimerait une convergence rapide et des résultats approximatifs, dans d'autres on souhaite des résultats fiables (et on peut se permettre de laisser calculer longtemps).

On peut néanmoins imaginer qu'une valeur "intermédiaire" entre `1` et `1e-8`, par exemple `1e-4`, offre en général un bon compromis entre ces deux aspects de temps de calcul et de précision de la classification. Notez que c'est cette valeur qui est mise par défaut dans tous les classificateurs implémentés par `nilearn` et `scikit-learn`, cf. par exemple `sklearn.linear_model.Ridge`.

Une bonne pratique pour choisir un paramètre de tolérance[7] comme `tol` consiste par commencer par une valeur relativement grande, comme `1e-2`, pour avoir rapidement des premiers résultats approximatifs assez rapidement; puis diminuer progressivement `tol` jusqu'à avoir de meilleurs résultats, et s'arrêter quand la convergence est trop lente à obtenir.

On peut aussi envisager une exploration automatique d'une grille de différentes valeurs de `tol`, par validation croisée (ce qui peut être fait avec scikit-learn via `GridSearchCV` notamment).

Conclusion

Ce TP a été l'occasion d'étudier la convergence du classificateur SpaceNet, et notamment l'influence du paramètre `tol` sur la vitesse de convergence et la précision du classificateur obtenu. Les résultats observés dans cette expérience avec la base de donnée Haxby, pour un problème de classification binaire ("house" vs. "scissors", or "scissors" vs. "chair"), ont confirmé l'intuition initiale : le temps de calcul et la précision augmentent quand `tol` diminue vers 0.

Je reste bien sûr à votre disposition, si besoin.

[7] Aussi appelé ε dans la plupart des algorithmes en mathématiques appliqués.