MLEA - Rapport de TP

Florent D'Halluin

18 Avril 2009

Quickstart

Parcours de correction rapide et efficace:

- `make quiet` dans un term libre (Compilation des tests et géneration des images en arrière-plan - peut prendre jusqu'à 20 minutes).
- Lecture en diagonale du rapport.
- Relecture des points intéressants (Partie 3).
- *Optionnel: Ctrl-c le make quiet.*
- `cd classification && make donut_low` (`make donut` peut prendre plusieurs minutes).
- Jeter un oeil aux images générées, tester quelques commandes.
- Noter.

1 Premiers Pas

Dossier `premiers-pas`.

1.1 Hello World!

`make hello`

Fichiers:

- `hello.py`: Implémentation Python, wrappers.
- `hello.c`: Implémentation C.
- `hello.cc`: Implémentation C++.

Le code est simple.

1.2 Multiplication de matrices

1.3 Benchmarking

`make matrix`

Fichiers:

- `matrix.py`: Implémentation Python/Numpy, benchmarks et tests.
- `matrix.cc`: Implémentation C++.

Pour utiliser les fonctions du module séparément, commenter `__work()` *en fin de fichier.*

Le code dans `matrix.cc` est clair. Dans `matrix.py`, les benchmarks sont fait en entourant l'appel aux fonctions mesurées d'appels à `time.time()`. `single_test()` mesure les performances des trois langages pour la multiplication de matrices de taille importante *(modifiable en fin de fichier)* [Figure 1]. `bench()` mesure les performances des trois langages pour la multiplication de matrices carrées de taille croissante *(modifiable en début de fichier)* [Figure 2].

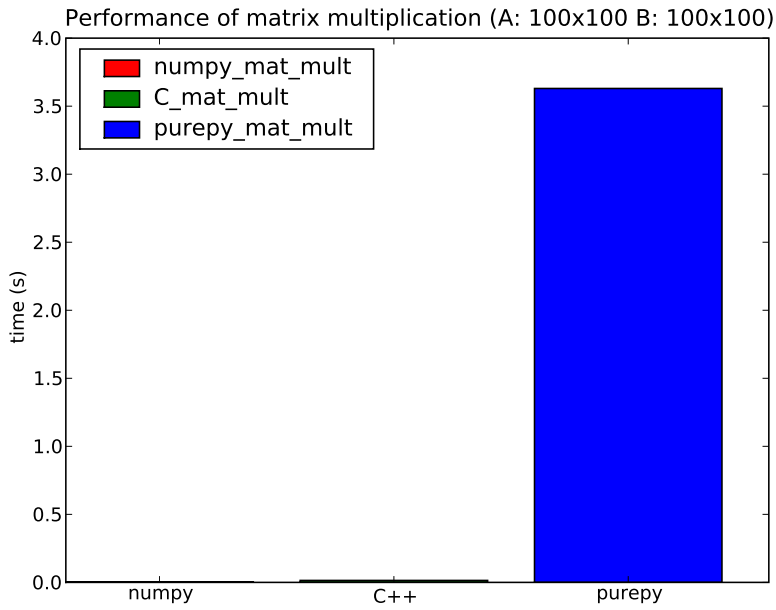

Figure 1: Performance de la multiplication de matrices

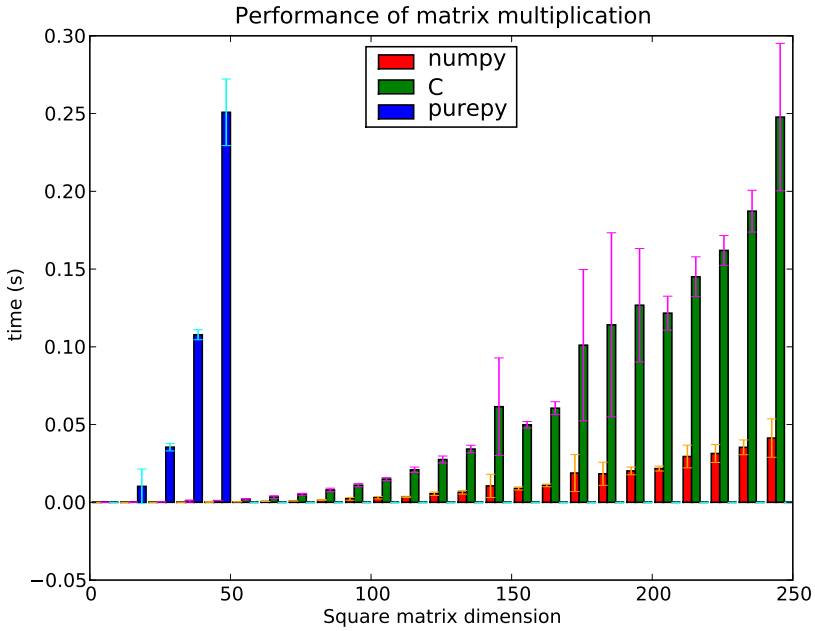

Figure 2: Performance de la multiplication de matrices (benchmark)

On observe que l'implémentation en Python pur est bien plus lente que celles en C++ et Numpy (suffisamment pour justifier que les benchmarks soient interrompus rapidement pour l'implémentation en Python pur). Étant donné l'implémentation naïve en C++ et Python pur *(cf code)*, il n'est pas inattendu que l'implémentation de Numpy soit la plus rapide.

2 Toy Problem

Dossier `classification`.

2.1 Prototype

2.2 Affichage

2.3 Stockage

`make toy`

Fichiers:

- `donut.py`: Implémentation Python

Pour utiliser les fonctions du module séparément, commenter `__work()` *en fin de fichier.*

Tests:

`python donut.py <samples> <noise> <name> [--noshow]`
e.g.: `python donut.py 500 0.2 donut_noisy`

Pour ne pas affichier les figures sous pylab, utiliser `--noshow` *en dernier paramètre.*

Le code dans `donut.py` est plus ou moins clair. L'ajout de bruit est effectué en déterminant pour chaque vecteur de données généré s'il doit ou non avoir la classe correcte (en fonction du seuil donné). Visuellement, le bruit a un impact fort [Figures 3, 4, 5].

La fonction `display()` est souvent reprise dans les autres fichiers sous une forme légèrement différente. Les figures sont générées depuis le code (en format pdf, eps et svg). Le module `pickle` est utilisé pour sauvegarder les données produites (elles sont chargées dans `classification.py`).

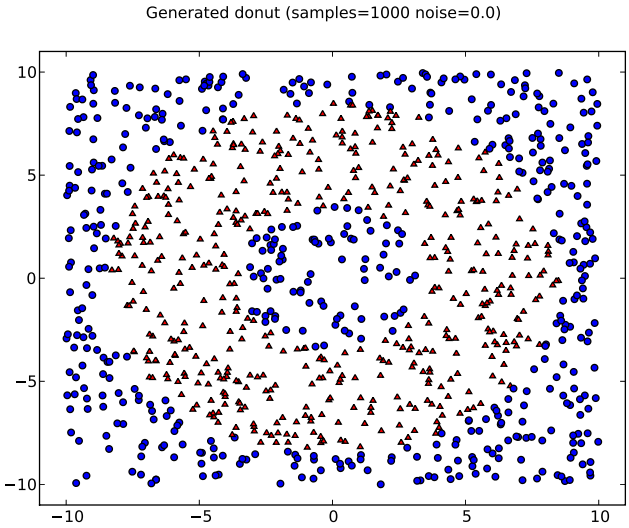

Figure 3: Génération de donut (0% de bruit)

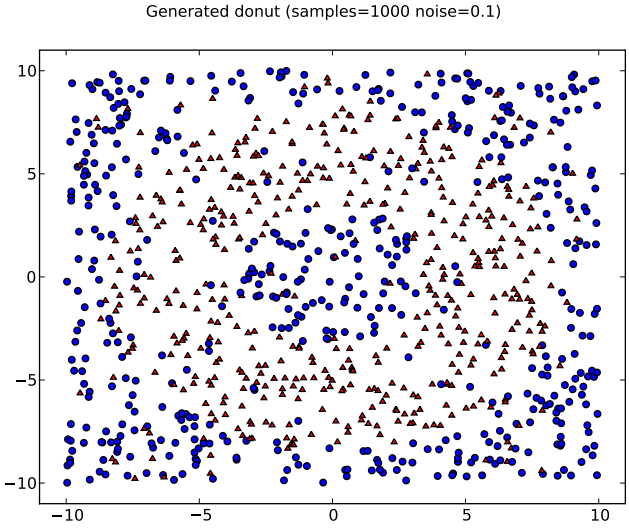

Figure 4: Génération de donut (10% de bruit)

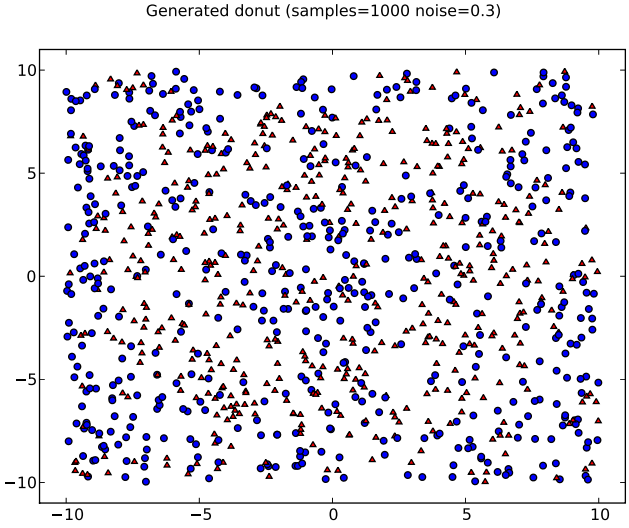

Figure 5: Génération de donut (30% de bruit)

3 Classification

Dossier `classification`.

```
make donut
```

Fichiers:

- `donut.py`: Création du set de données donut
- `iris.py`: Création du set de données iris
- `iris.data`: Données brutes iris
- `classification.py`: Implémentation de KNN, K-Fold Cross Validation, fonction de décision, et bonus.

Pour utiliser les fonctions des modules séparément, commenter `__work()` *en fin de fichier.*

Tests:

```
python iris.py <name> [--noshow]
python donut.py <samples> <noise> <name> [--noshow]
python classification.py <name> [--noshow]
```

e.g.:

```
python donut.py 500 0.2 my_donut
python classification.py my_donut
```

Pour ne pas affichier les figures sous pylab, utiliser `--noshow` *en dernier paramètre.*

3.1 KNN classifier

Emplacement: `Class KNN` dans `classification.py`.

L'implémentation est la plus simple possible:

- `__init__()` peut prendre en paramètre une fonction de décision et une fonction de poids. Des exemples sont fournis en début de fichier.
- `train()` ne fait que stocker les valeurs passées.
- `__nn()` calcule les plus proches voisins d'un point donné (sur l'ensemble des données d'apprentissage). Le prétraitement des données pendant l'apprentissage (partitionnement de l'espace, cleaning) pourrait permettre un gain de performances important.
- `process()` Classifie un ensemble de données en fonction des plus proches voisins de chacun des points.
- `decision()` calcule la valeur de la fonction de decision en chacun des points d'un ensemble de données.

La fonction `test_knn()` (ligne 203) classifie un ensemble de données aléatoires [Figure 6].

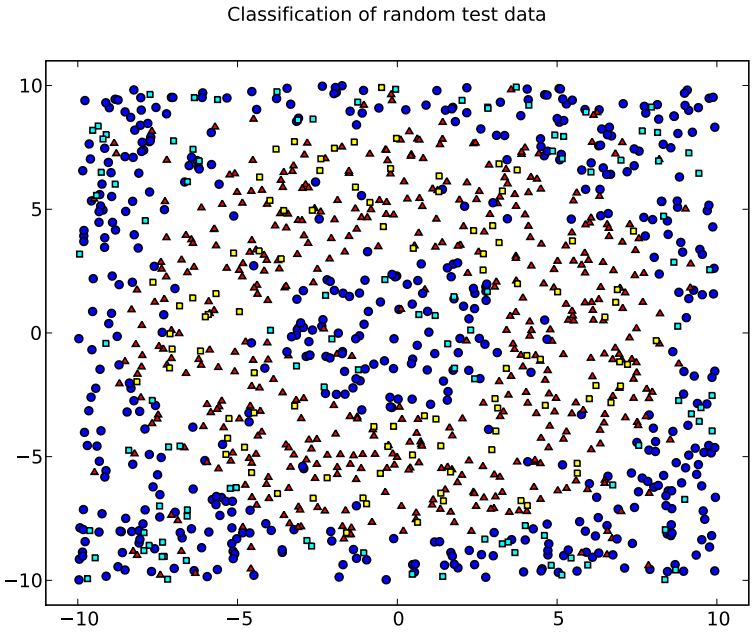

Figure 6: Classification de points aléatoires (10% de bruit, KNN(5))

3.2 K-Fold Cross Validation

Emplacement: `kfcrossval()` dans `classification.py` (ligne 148).

Un nombre arbitraire de sections (paramètre k) sont extraites des données fournies. Chaque section sert une fois de données de tests, les autres étant utilisées pour l'apprentissage. Au bout des k itérations, les moyennes des taux de reconnaissances et les écarts-types sont calculés pour chaque classifieur, puis retournés.

Les fonctions `test_kfcrossval()` et `test_kfcrossval_knns()` (ligne 270) testent et affichent l'algorithme sur une liste de classifieurs. Ces tests sont effectuées sur des classifieurs standards, sur des classifieurs dont la fonction de distance a été modifiée, et sur des classifieurs dont la fonction de poids a été modifiée.

On observe une meilleure résistance au bruit pour les classifieurs KNN de degré important (5 et 10) que pour ceux de degré faible (1 et 3) [Figures 7, 8, 9].

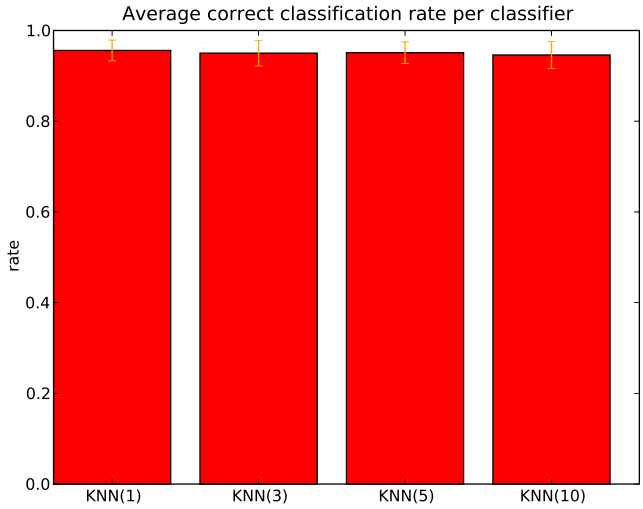

Figure 7: K-Fold Cross Validation (0% de bruit)

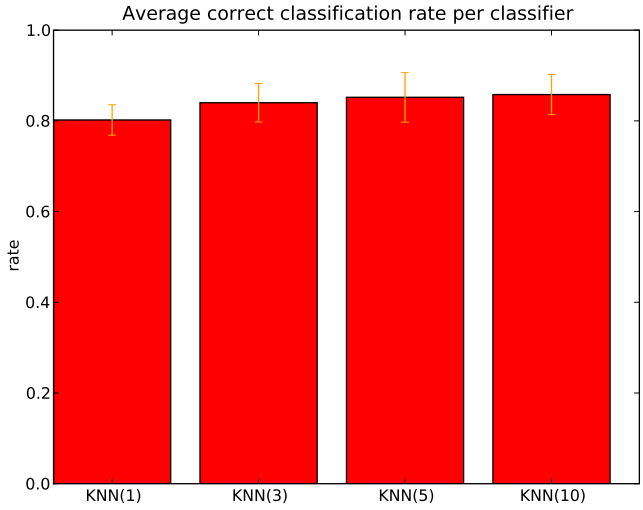

Figure 8: K-Fold Cross Validation (10% de bruit)

3.3 Fonction de décision

Emplacement: `KNN.decision()` dans `classification.py` (ligne 126).

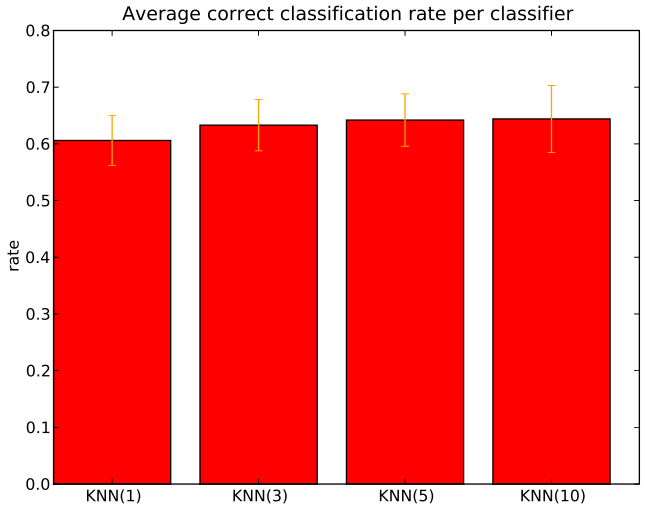

Figure 9: K-Fold Cross Validation (30% de bruit)

La fonction de décision parcours la liste des classes apprises et une liste de points passés en paramètre, et pour chaque classe et chaque point, associe le nombre de plus proches voisins de la classe donnée pour le point donné (les points on un nombre de dimensions arbitraire).

La fonction `test_decision()` (ligne 222) génère une grille de dimension 2 et affiche la fonction de décision correspondante pour un KNN entrainé sur l'exemple du donut. Les tests sont effectués avec une grille de 30 par 30, soit 900 points *(modifiable en haut de fichier)*.

Le bruit original a un impact fort sur la fonction de décision, notamment pour des KNNs de degré faible [Figure 10].

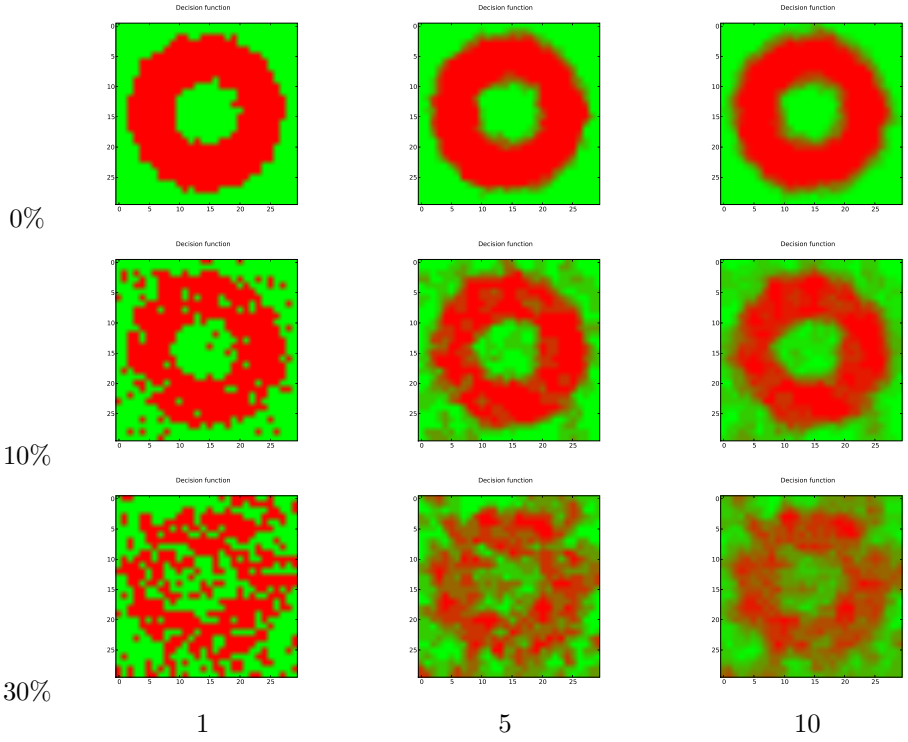

Figure 10: Fonction de décision (H: degré du KNN, V: bruit))

3.4 Iris

`make iris`

Emplacement: `iris.py`.

L'extraction des données fournies est triviale. La fonction `display()` génère et affiche pour chaque classe de fleur la moyenne de la valeur de chaque composante dans les données fournies, ainsi que l'écart-type [Figure 11].

La K-Fold Cross Validation indique que les KNNs d'ordre faible on des performances légèrement meilleures que les KNNs d'ordre plus élevé [Figure 12]. Malgré le faible nombre de données (150), la classification est bonne (95%).

3.5 Bonus

Les deux suggestions ont été implémentées *(cf KNN-classifier)*. Les fonctions d'exemples sont les suivantes:

Distances:

- `stddistance()`: Distance standard (Euclidienne).
- `distance_manhattan()`: Distance manhattan (somme des valeurs absolues des différences entre chaque composante).
- `distance_max()`: Maximum des valeurs absolues des différences entre chaque composante).

Poids: 4 fonctions (cf. `classification.py` lignes 55-69).

L'impact de ces fonctions est réduit [Figures 13, 14], mais de légères variations apparaissent selon les fonctions utilisées.

Note: D'autres images de test sont générées par `make quiet` *ou* `make`*, dans le dossier* `classification`*.*

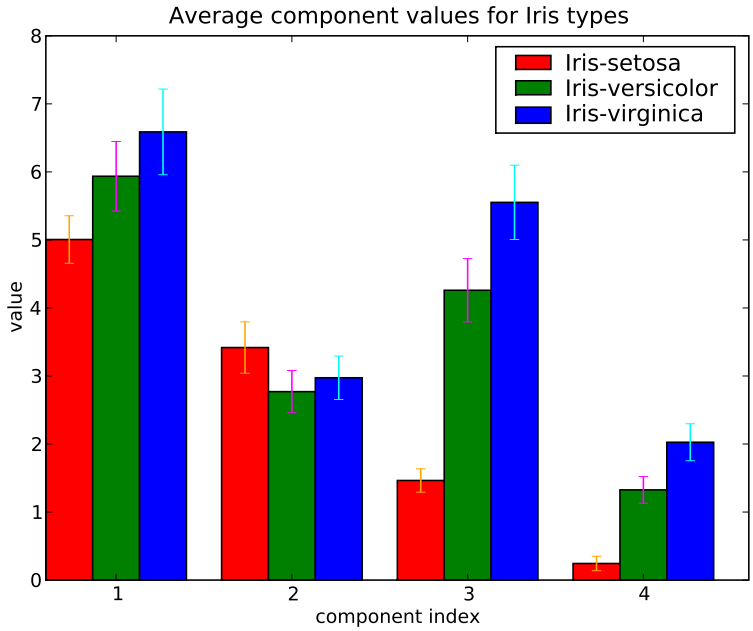

Figure 11: Valeur moyenne des composantes pour chaque type d'Iris.

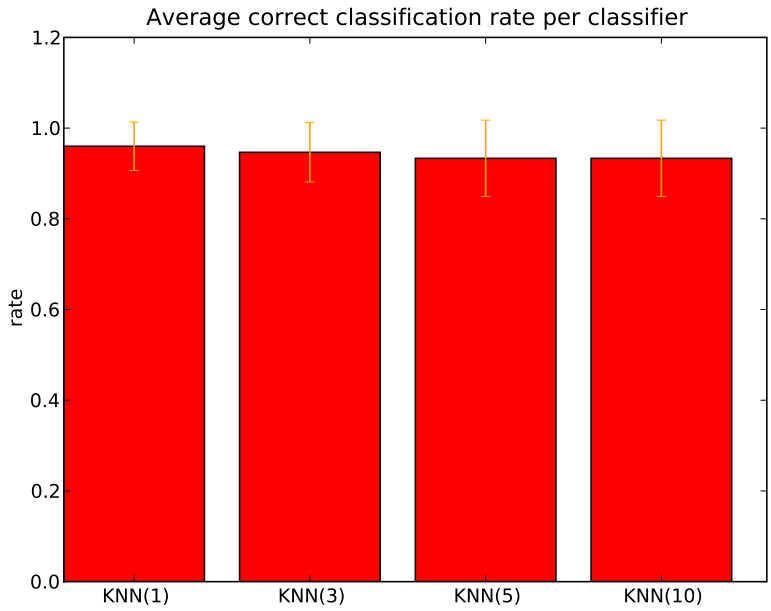

Figure 12: K-Fold Cross Validation (Iris).

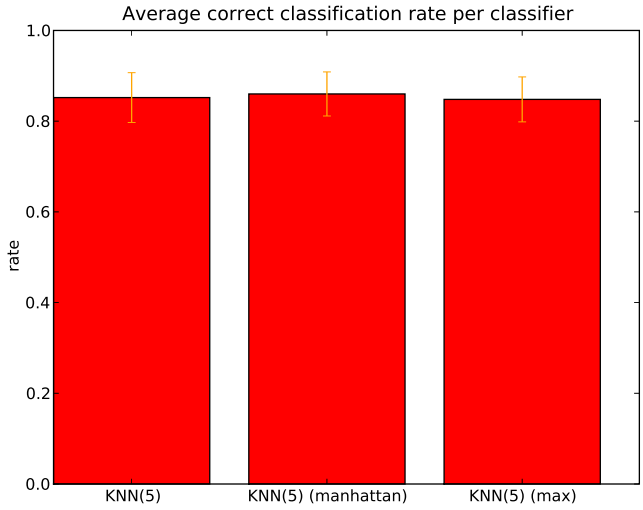

Figure 13: KNN à fonction de distance différente (10% de bruit)

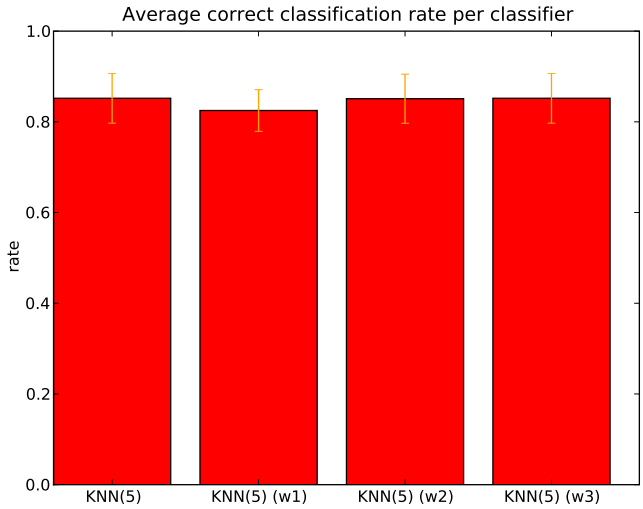

Figure 14: KNN à fonction de poids différente (10% de bruit)