MLEA - Rapport de TP 2

Florent D'Halluin

18 Mai 2009

Quickstart

Parcours de correction rapide et efficace:

- `make` dans un term libre (Compilation des tests et géneration des images en arrière-plan — peut prendre 2 heures).
- Lecture du rapport.
- *Optionnel: Ctrl-c le make (les images les plus significatives sont générées en premier dans classification).*
- Jeter un oeil aux images générées, tester quelques commandes.
- Noter.

1 Magic mushroom

`cd classification`. `make` (génère les images pour toutes les sous-sections).

1.1 Naive Bayes Classifier

`make mushroom` `python classification.py`

Fichiers:

- `mushroom_data`: Dataset.
- `mushroom.py`: Extraction du dataset (ordre aléatoire).
- `bayes.py`: Naive Bayes Classifier.
- `kfcv.py`: K-Fold Cross-Validation.
- `classification.py`: Tests et génération des courbes.

1.1.1 K-Fold Cross-Validation

Emplacement: `kfcrossval()` dans `kfcv.py`.

Le code a été repris du TP 1.

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.

Si le paramètre `results` est fourni, tous les résultats (classe reconnue et probabilité de certitude) y sont stockés afin d'être interprétés par les fonctions de construction de courbes ROC.

1.1.2 Naive Bayes Classifier

Emplacement: `class BAYES` dans `bayes.py`.

Lors de l'entrainement les valeurs lues pour chaque caractéristique sont stockées dans des dictionnaires (un par caractéristique observée). L'ensemble des dictionnaires est ensuite parcouru pour associer à chaque valeur rencontrée sa probabilité d'apparition par rapport aux autres valeurs de la même caractéristique.

Lors de la reconnaissance, les probabilités d'apparition de la valeur de chaque caractéristique sont multipliées entre-elles puis par la probabilité d'apparition de la classe considérée. Pour chaque point à traiter, la classe ayant la probabilité d'apparition maximale est renvoyée.

Sur le dataset mushroom, le Naive Bayes Classifier a un taux de reconnaissance excellent (de même que les autres classifieurs — Figure 1). Lorsque les tests sont effectués sur 2% seulement du dataset, les performances restent bonnes (Figure 2). On peut noter que ce classifieur est bien plus rapide que KNN (sans partitionnement, KNN est de complexité quadratique par rapport au nombre de points ; Bayes est de complexité linéaire).

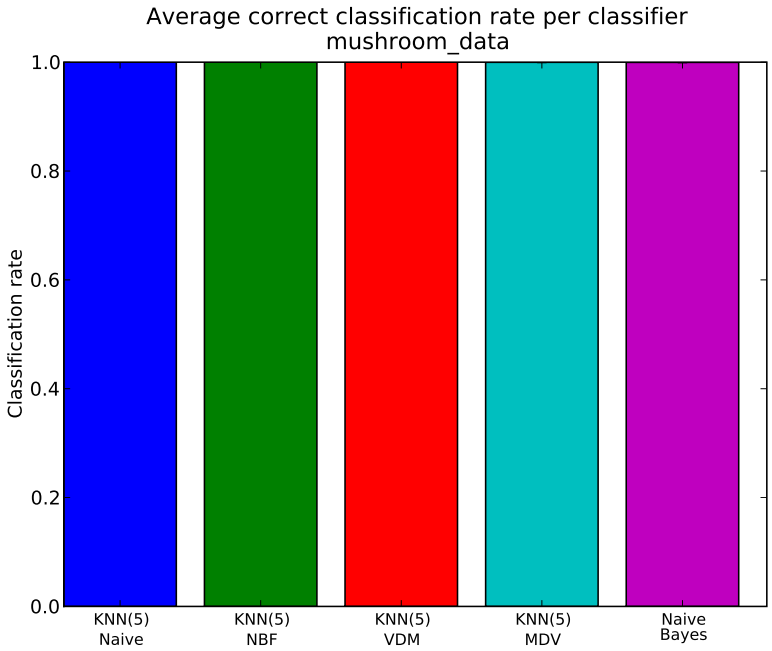

Figure 1: Comparaison de classifieurs sur le dataset mushroom.

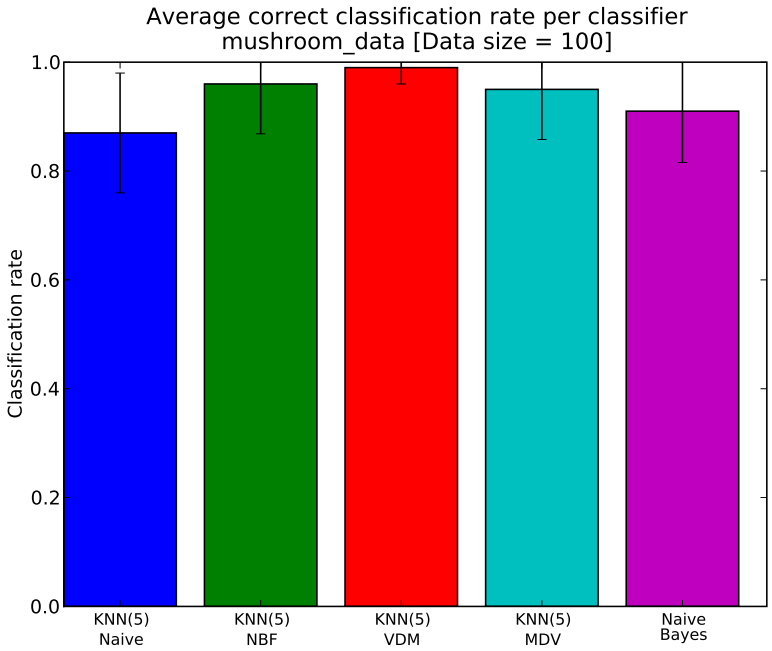

Figure 2: Comparaison de classifieurs sur une partie du dataset mushroom.

1.2 ROC

`make mushroom python classification.py`

Fichiers:

- `roc.py`: Génération des graphes ROC et des courbes ROC.

Emplacement: `roc()` dans `roc.py`.

La génération des graphes et courbes ROC se déroulent en 2 étapes:

- Traitement des résultats
- Génération des points de courbe ROC

Le traitement des résultats (première boucle `for`) a pour but d'ordonner les résultats par taux de certitude décroissant pour la classe positive (paramêtre `plabel`). Pour les KNN, le taux de certitude correspond au nombre de voisins de la classe positive par rapport à K, d'où l'aspect haché des courbes de ROC (un point par valeur distincte du taux de certitude).

La génération des points de courbe ROC se fait en suivant l'algorithme 1 décrit dans le dossier *Introduction to ROC analysis*, mentionné sur la ML.

Les graphes et courbes ROC ne sont disponibles que lorsqu'une classe "positive" est identifiable (i.e. pour donut et mushroom). Plusieurs exemples de graphe et courbes sont inclus dans ce rapport (Figure 3, Figure 4, Figure 5).

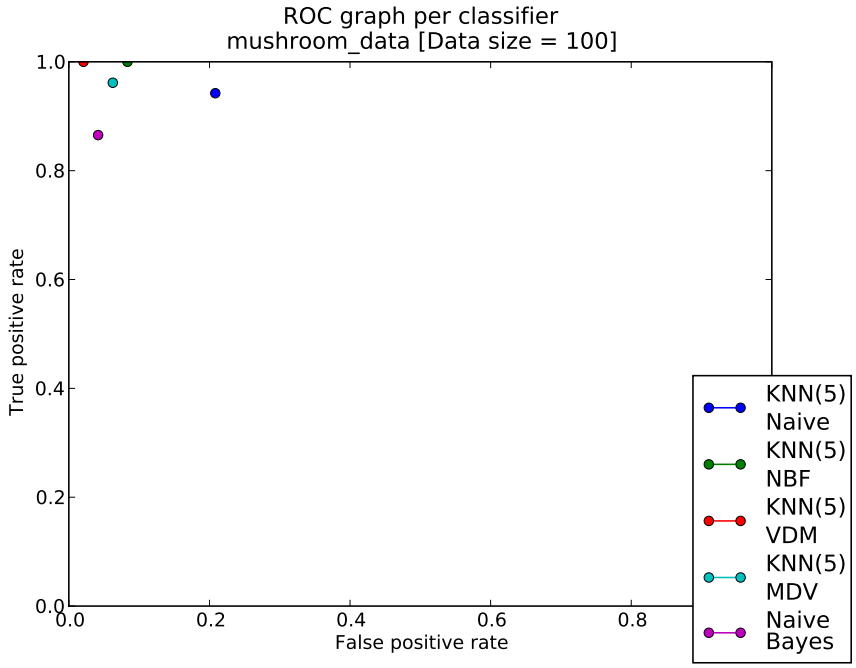

Figure 3: Graphe ROC pour plusieurs classifieurs, sur une partie du dataset mushroom.

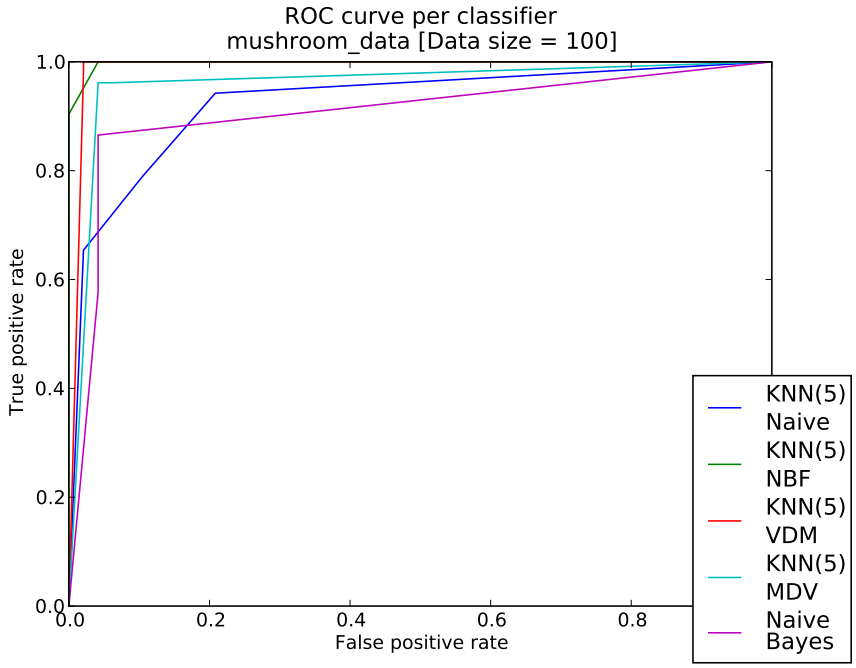

Figure 4: Courbe ROC pour plusieurs classifieurs, sur une partie du dataset mushroom.

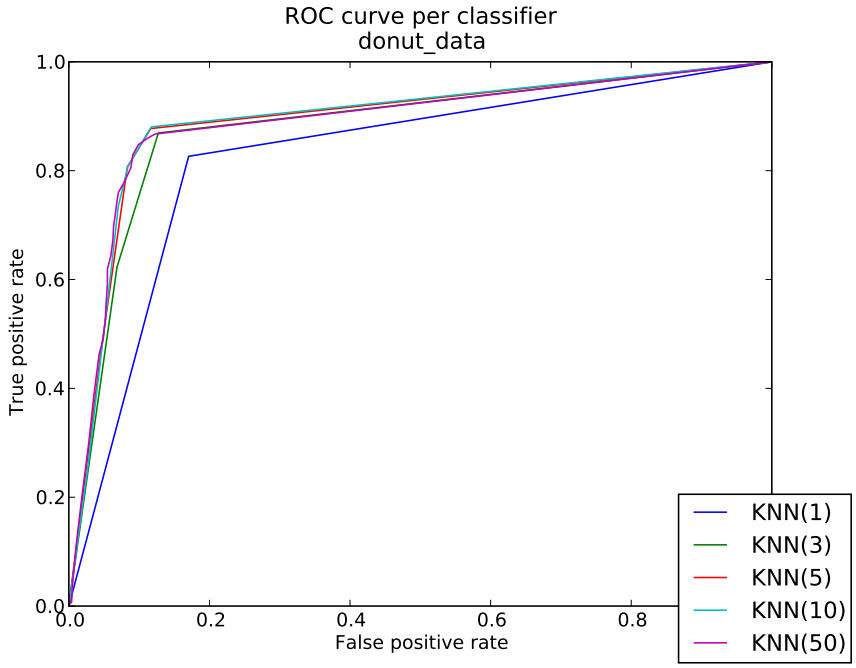

Figure 5: Courbe ROC pour plusieurs classifieurs, sur une partie du dataset donut.

1.3 Continuousification

`make mushroom python classification.py`

Fichiers:

- `knn_nbf.py`: Continuousification NBF.
- `knn_vdm.py`: Continuousification VDM.
- `knn_mdv.py`: Continuousification MDV.

1.3.1 Continuousification NBF

Emplacement: `class KNN_NBF` dans `knn_nbf.py`.

Comme pour les autres méthodes de continuousification, la classe `KNN_NBF` est un simple wrapper autour de la classe `KNN`. Elle se charge de réécrire les données d'entrée de `train()` et `process()` sous la forme correspondante à la méthode de continuousification choisie. Les données de sortie sont ensuite traitées par un KNN générique.

Pour la continuousification NBF, pendant l'entrainement, chaque valeur distincte de chaque attribut des données d'entrée engendre l'ajout d'une dimension aux données de sortie. Pour chaque point, la valeur de sortie est 1 pour les dimensions dont la valeur associée est observée dans les données d'entrée, 0 sinon. L'implémentation est faite à l'aide de dictionnaires.

1.3.2 Continuousification VDM

Emplacement: `class KNN_VDM` dans `knn_vdm.py`.

La méthode de continuousification VDM est implémentée à partie de la publication *Transformations of Symbolic Data for Continuous Data Oriented Models* citée sur la ML. Elle n'a pas posé de problèmes.

1.3.3 Continuousification MDV

Emplacement: `class KNN_MDV` dans `knn_mdv.py`.

La méthode de continuousification MDV est implémentée à partie de la publication *Transformations of Symbolic Data for Continuous Data Oriented Models* citée sur la ML. Elle est similaire à l'implémentation VDM.

1.3.4 Comparaison

Le dataset `mushroom` est très simple, et les taux de classification excellents, même sur une faible partie du dataset (Figure 6, Figure 7, Figure 8). Sur 2% du dataset, VDM est la meilleure méthode (Figure 2). Lorsque les données sont continues (e.g. sur `donut`), la continuousification est évidemment inefficace (Figure 9).

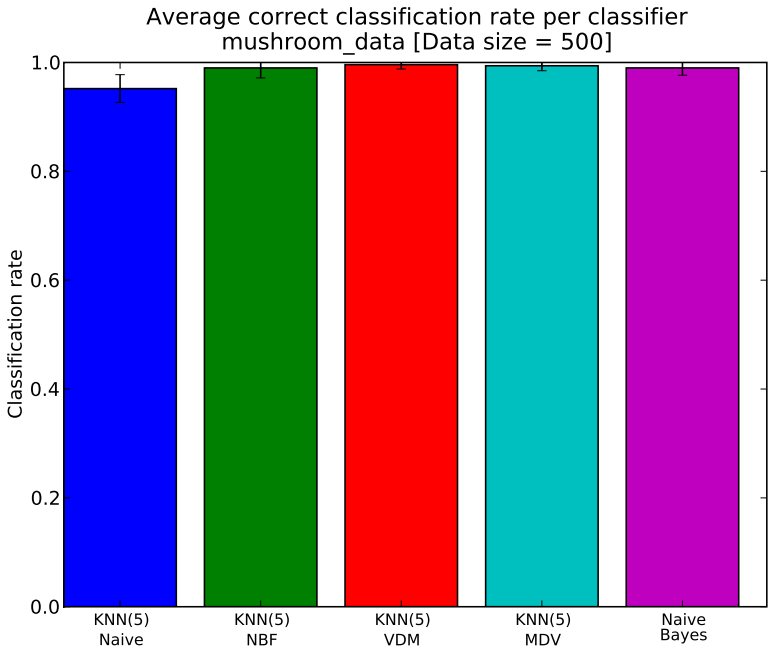

Figure 6: Continuousificaion sur une partie du dataset mushroom.

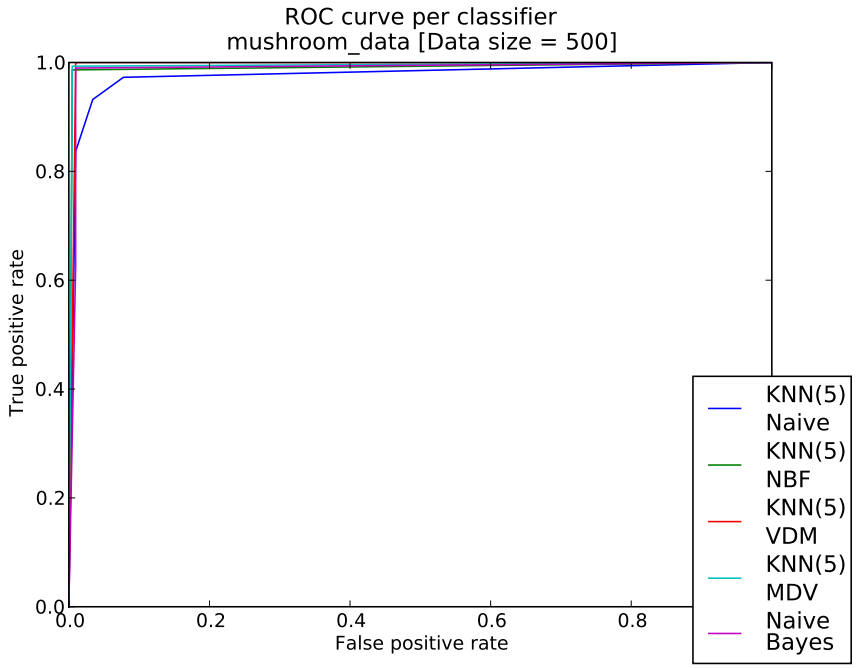

Figure 7: Continuousificaion sur une partie du dataset mushroom (courbe ROC).

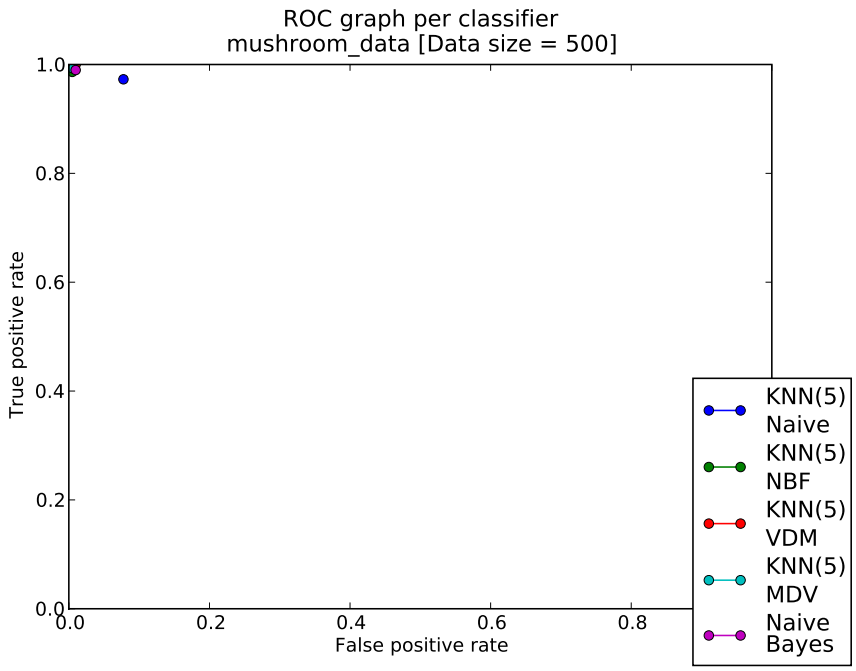

Figure 8: Continuousificaion sur une partie du dataset mushroom (graphe ROC).

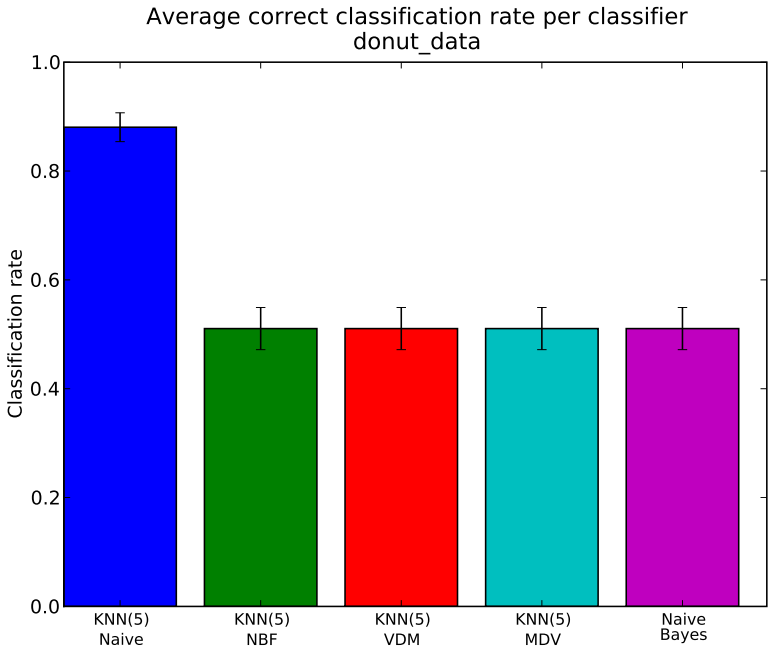

Figure 9: Continuousificaion sur le dataset donut.

2 Clustering

`cd clustering`. `make` (génère les images pour toutes les sous-sections).

Fichiers:

- `k_means.py`: K-MEANS.
- `k_means_pp.py`: K-MEANS++.
- `clustering.py`: Test functions, funky verbose.
- `tiny_toy.py`: Dataset tiny-toy.
- `teddy_toy.py`: Dataset teddy-toy.

2.1 Tiny toy

Emplacement: `class KMEANS` dans `k_means.py`.

L'implémentation du K-MEANS est faite telle que décrite dans la littérature. Les résultats de toutes les itérations sont retournés afin de pouvoir en extraire l'avancement du clustering à chaque itération.

Le déroulement de K-MEANS sur tiny-toy est simple et rapide (Figure 10 et Figure 11).

2.2 Teddy toy

Emplacement: `gen_grid_normal_dataset()` dans `teddy_toy.py`.

Le code est simple. Figure 12 et Figure 13 montrent deux exemples de grilles générées.

2.3 Funky verbose

Emplacement: `__display_clusters()` dans `clustering.py`.

La fonction d'affichage est simple et la sortie similaire à celle du sujet du TP (Figure 14 et Figure 15).

Sur le dataset teddy-toy (grille donnée en exemple dans le sujet), on observe que les centres tendent à se retrouver entre deux clusters, ou deux par cluster (Figure 15). Étant donné la sélection naïve (aléatoire) des centres initiaux, dont certains sont proches les uns des autres, c'est un résultat attendu (Figure 16). Le dataset teddy-toy-small, plus simple, n'a pas ce problème.

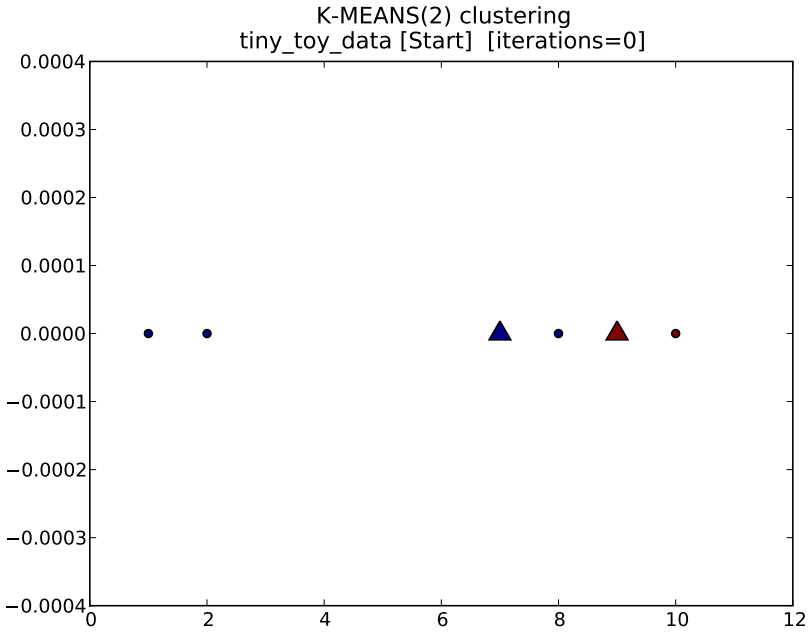

Figure 10: Clustering sur tiny-toy (première itération).

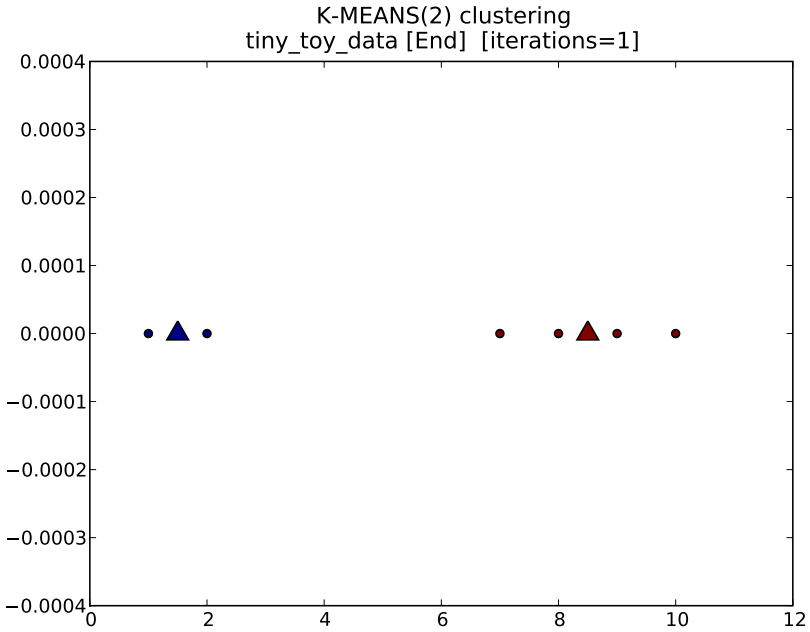

Figure 11: Clustering sur tiny-toy (dernière itération).

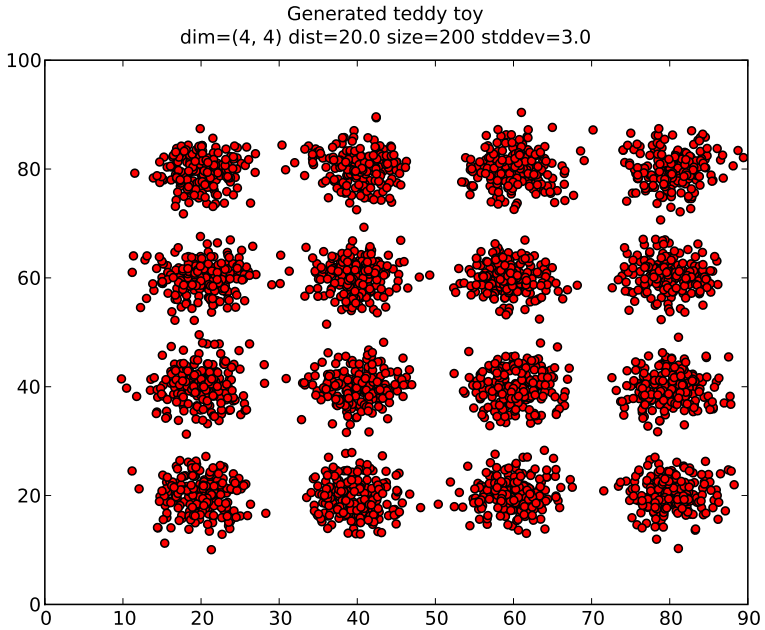

Figure 12: Dataset teddy-toy.

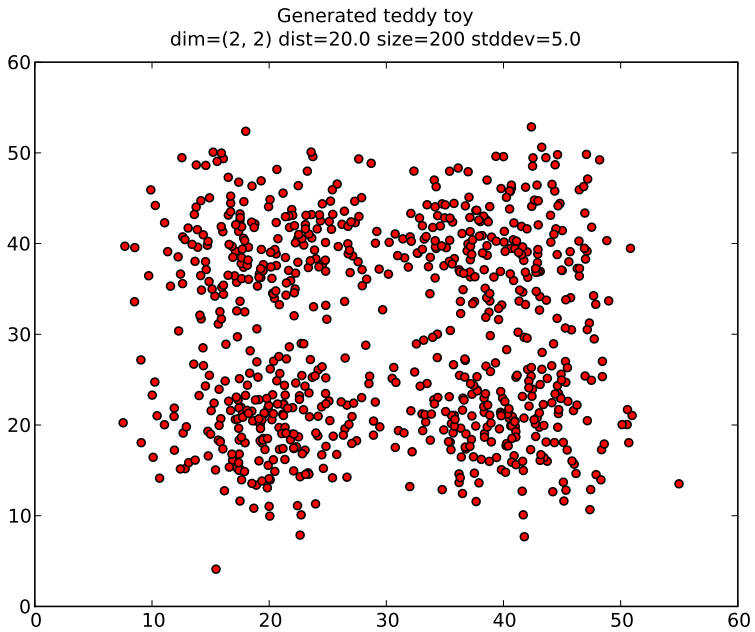

Figure 13: Dataset teddy-toy.

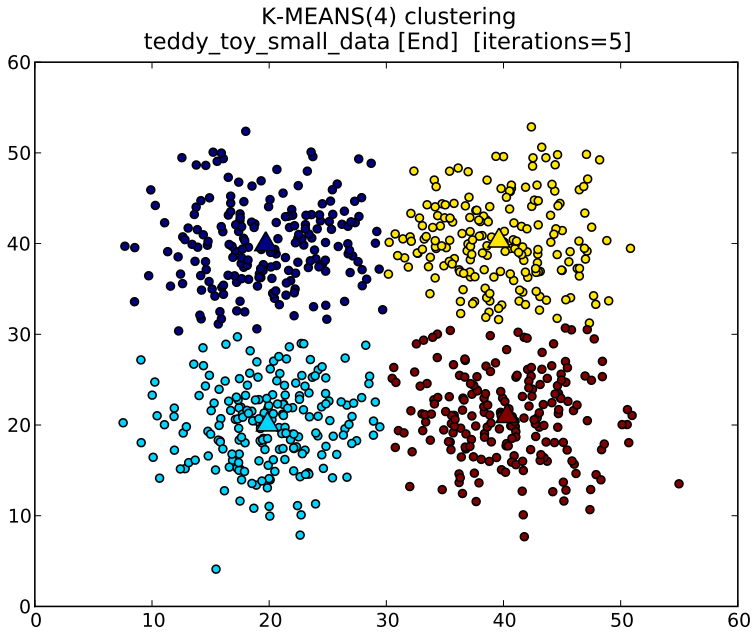

Figure 14: K-MEANS clustering sur le dataset teddy-toy-small.

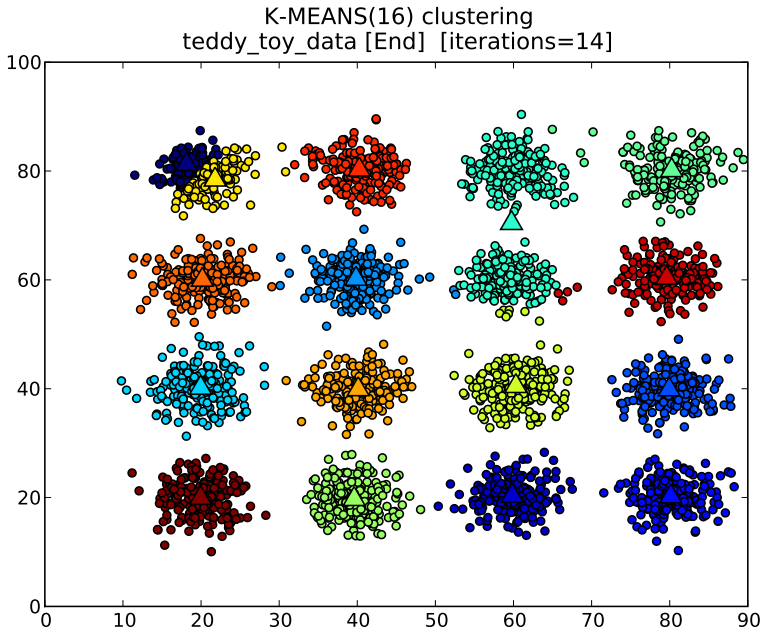

Figure 15: K-MEANS clustering sur le dataset teddy-toy.

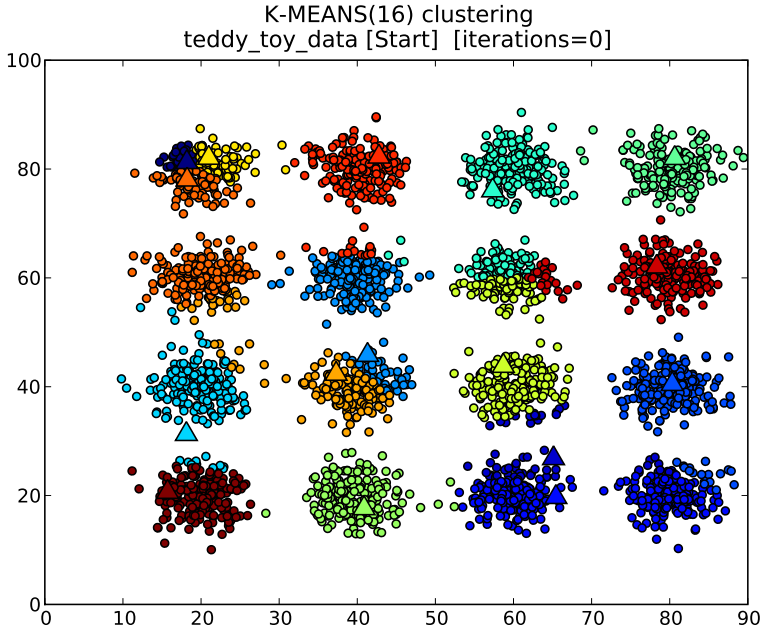

Figure 16: K-MEANS clustering sur le dataset teddy-toy, centres initiaux.

2.4 K-MEANS++

Emplacement: `choose_std()` et `class KMEANSPP` dans `k_means_pp.py`.

La différence entre K-MEANS et K-MEANS++ est uniquement dans le choix des centres initiaux. Une meilleure répartition des centres dans les données est susceptible de donner des meilleurs résultats de clustering.

L'implémentation suit la formule donnée dans le cours, et donne des centres initiaux bien mieux répartis, à faible coût (Figure 17).

Un test sur 10 itération montre qu'en effet, le clustering est bien meilleur avec K-MEANS++ qu'avec K-MEANS (Figure 18). Le nombre d'itérations plus faible pour K-MEANS++ peut également induire un gain de vitesse d'exécution (Figure 19). Répété 100 fois, le test donne les mêmes résultats.

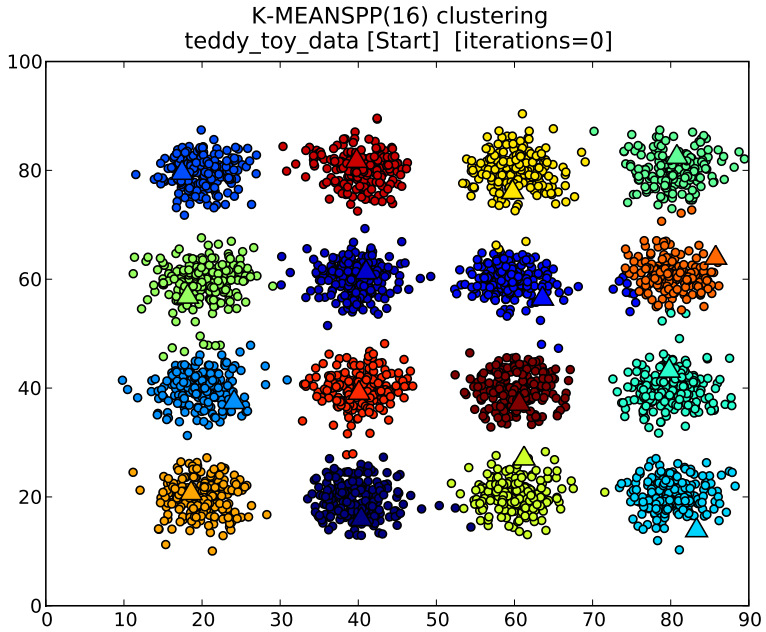

Figure 17: K-MEANSPP clustering sur le dataset teddy-toy, centres initiaux.

2.5 Bonus

Emplacement: `choose_max()` et `class KMEANSPP` dans `k_means_pp.py`.

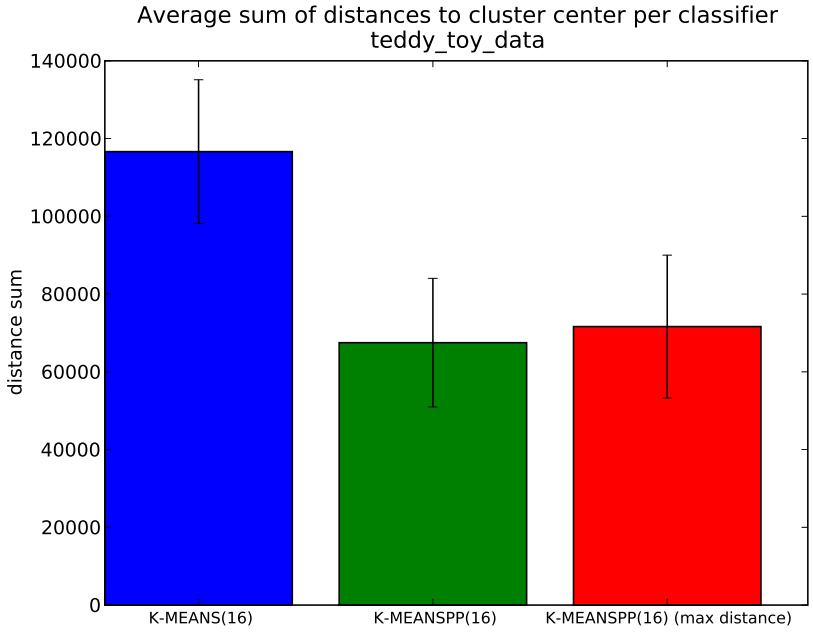

Figure 18: Somme des distances intra-cluster pour le dataset teddy-toy.

Une méthode de sélection des centres alternative consiste à maximiser la distance entre les centres initiaux au lieux de les répartir selon une loi de probabilité, mais cette méthode pourrait être très sensible au bruit.

Le positionnement des centre initiaux semble mieux qu'avec K-MEANS, mais moins bon qu'avec K-MEANS++ classique (Figure 20). Les résultats finaux sont similaires à ceux de K-MEANS++, bien que légèrement moins bons (Figure 18 et Figure 19).

On peut noter que d'autres indicateurs de performance que la somme des distances intra-clusters peuvent être représentatifs pour comparer les algorithmes de clustering. Un indicateur alternatif (la somme de toutes les distances intra-cluster mises au carré) a été utilisé, mais ne révèle rien de plus (Figure 21).

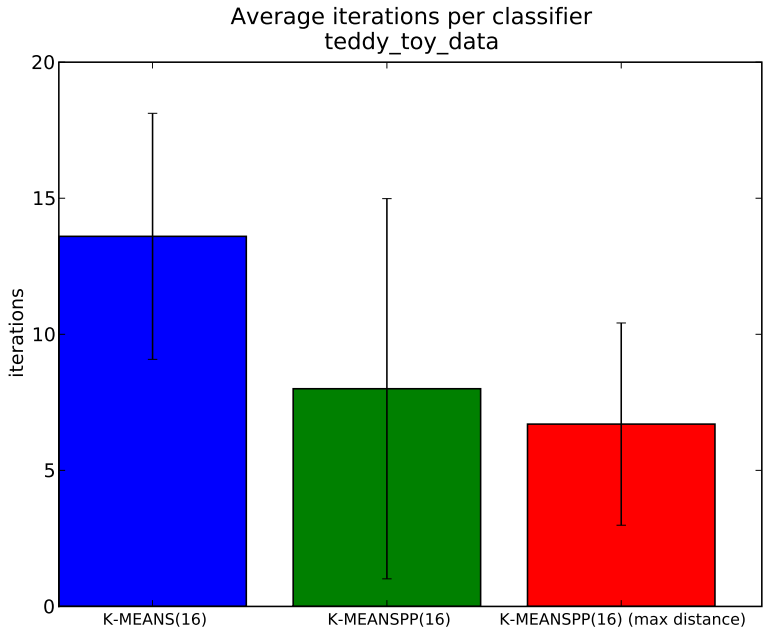

Figure 19: Nombre d'itérations pour le dataset teddy-toy.

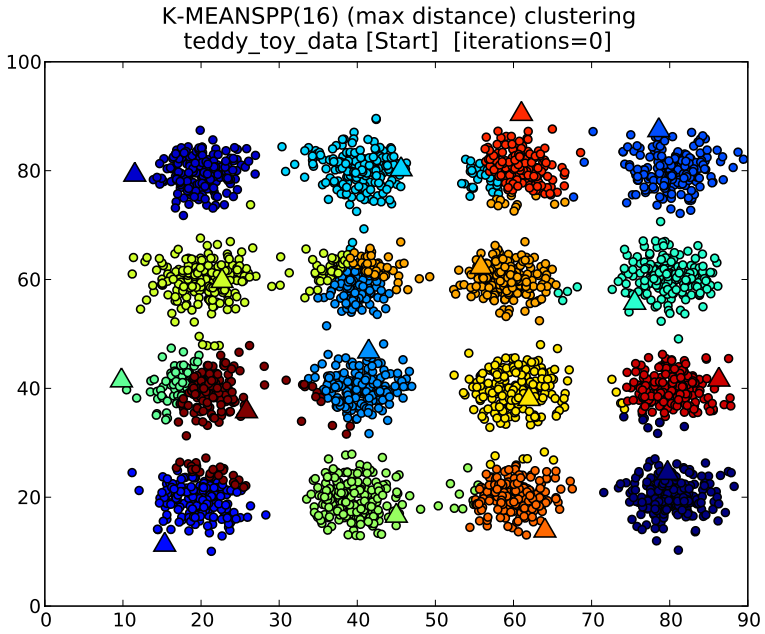

Figure 20: K-MEANSPP (max distance) clustering sur le dataset teddy-toy, centres initiaux.

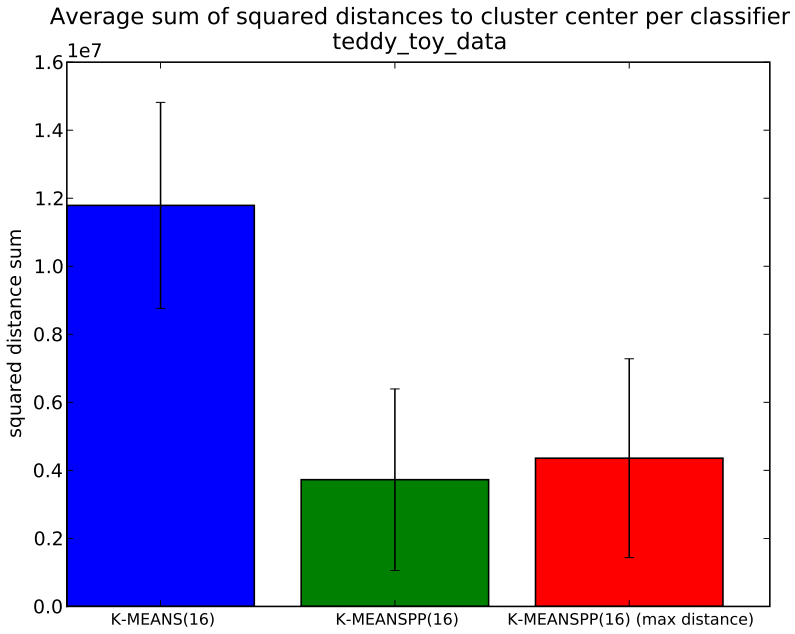

Figure 21: Somme des distances intra-cluster au carré pour le dataset teddy-toy.

3 GMM

`cd classification. make.`

Fichiers:

- `bayes_ndist.py`: Normal Distribution Bayes Classifier.
- `optdigits.py`: Dataset optdigits. Deux datasets sont fournis, training et test. Les résultats sont présentés sur le dataset training, plus large.

Seule la méthode Normal Distribution a été implémentée, parce qu'il faut vivre, aussi, à côté.

3.1 Normal distribution

La formule pour la distribution normale a été reprise du cours. L'implémentation du classifieur est proche de l'implémentation du classifieur Bayes naïf, à ceci près que seules les valeurs moyennes et de la déviation standard pour chaque attribut sont stockées pendant l'entrainement.

Ces valeurs permettent d'estimer la probabilité d'avoir la valeur de l'attribut considéré pour chaque classe. Les probabilités sont multipliées entre elles par classe, puis par la probabilité d'apparition de la classe. La classe la plus probable pour le point considéré est alors retournée.

Cette méthode est efficace sur des données continues, mais elle est légèrement en dessous de KNN sur les datasets considérés (Figure 22 et Figure 23). On peut noter que le classifieur Bayes naïf a un taux de reconnaissance relativement élevé étant donné que les valeurs de chaque attribut ne sont pas toutes distinctes. Sur une faible portion du dataset optdigits, le Bayes naïf est en revanche très mauvais, alors que les deux autres sont plutôt bons (Figure 24).

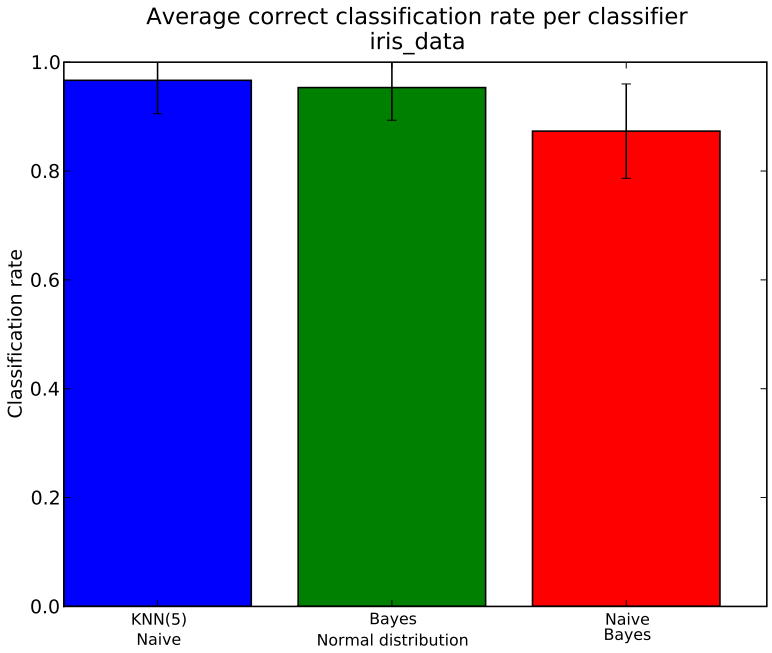

Figure 22: Comparaison de classifieurs sur le dataset iris.

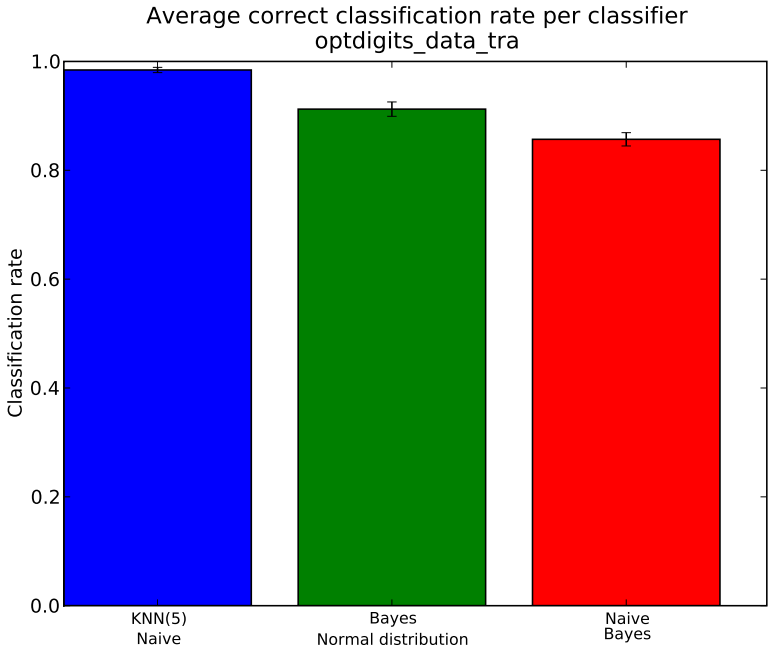

Figure 23: Comparaison de classifieurs sur le dataset optdigits (training).

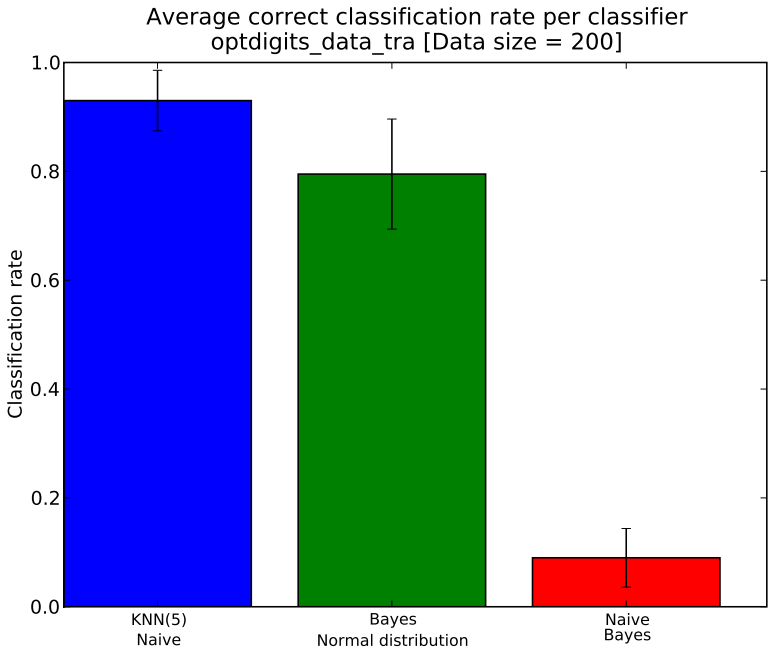

Figure 24: Comparaison de classifieurs sur une partie du dataset optdigits (training).