TP1: détecteur de spams avec les k plus proches voisins (k-NN)

Issu d'un TP conçu par : Anaël Bonneton & Ulysse Beaugnon

De nombreux services de messagerie électronique contiennent un détecteur de spam capable de classifier les mails en deux catégories, spam et non spam (appelé ham en anglais). Au cours de ce TD vous allez implémenter votre propre détecteur de spams basé sur k-NN et le paralléliser grâce à MPI. k-NN nécessite des données d'apprentissage, cad des mails pour lesquels on sait si ce sont des spams. SpamAssassin Public Corpus donne accès à de nombreux mails labélisés. Les données utilisées dans ce TD proviennent de ce corpus.

Transformation des mails en vecteurs binaires de même taille

Tout d'abord, on effectue un pré-traitement sur les mails afin d'obtenir une représentation normalisée :

1. Enlever l'en-tête du mail
2. Mettre toutes les lettres en minuscules
3. Remplacer les URLs, les adresses mails, les tags HTML, les nombres et les dollars par des tags (`URL, MAIL, HTML, NUM, DOLLAR`)
4. Réduire les mots à leur racine (par exemple *birds* devient *bird* et *called* devient *call*).
5. Enlever les caractères n'étant pas des mots

Ensuite, chaque mail est représenté par une liste de mots. À partir de ces mails normalisés on crée un dictionnaire contenant les mots apparaissant au moins 100 fois dans le corpus. On obtient alors un dictionnaire de taille 1899. Chaque mail normalisé m est transformé en un vecteur binaire v de taille 1899 tel que v[i] = 1 si et seulement si le i-ème mot du dictionnaire apparaît dans le mail m.

Vous n'avez pas besoin d'effectuer le pré-traitement des données. On vous fournit les données sous forme de vecteurs avec leur label. Les données sont séparées en deux fichiers: `train.txt` et `test.txt`. Le label *ham* est représenté par l'entier `0` et le label *spam* par `1`.

L'algorithme des k plus proches voisins

L'algorithme des k plus proches voisins prédit le label d'une instance (en machine learning, une *instance* est une donnée) en cherchant parmi un ensemble de données d'apprentissage (pour lesquelles les labels sont connus) quelles sont les k instances les plus proches de l'instance à classifier. L'algorithme des k plus proches vosins "classique" prédit alors le label le plus présent parmi les k plus proches voisins. L'algorithme exact est détaillé ci-dessous :

Data: Une instance à classer $mail\_to\_classify$, des instances d'apprentissage $train\_mail$.
Result: Le label prédit pour $mail\_to\_classify$.

`/* Deux tableaux pour stocker les labels et les distances des` k `plus proches voisins. Respectivement initialisés à -1 et MAX_INT. */`
$labels \leftarrow -1$
$distances \leftarrow$ `MAX_INT`

`/* Trouver les` k `plus proches voisins de` $mail\_to\_classify$. `*/`
foreach $mail \in train\_mail$ **do**
  if *mail fait parti des k plus proches voisins parmi ceux déjà rencontrés* **then**
    inserer $mail$ dans $labels$ et $distances$
  end
end

return *le label le plus présent dans labels.*

Évaluation du détecteur de spams

Une fois que vous aurez implémenté votre détecteur de spam il faudra l'évaluer. La méthode d'évaluation la plus simple est le taux d'erreurs qui correspond simplement au nombre de mails mal classifiés sur le nombre total de mails. Cependant, si dans vos données vous avez 95% de non spams et seulement 5% de spams, même si votre classifieur prédit toujours non-spam son taux d'erreur sera seulement de 5% alors qu'il ne s'agit pas d'un bon classifieur.
Afin de mieux évaluer le détecteur de spam, il faut dans un premier temps calculer la matrice de confusion qui considère chaque label séparément. La matrice de confusion avec laquelle nous allons travailler est présentée dans ci-dessous.

Label prédit		**`Spam`**	**`Ham`**
Vrai label	**Spam**	True Positive (TP)	False Negative (FN)
	Ham	False Positive (FP)	True Negative (TN)

À partir de la matrice de confusion on peut définir de nouveaux estimateurs : taux de détection (aussi appelé recall), taux de fausses alarmes, precision et F-score.

Detection rate = Recall = TP / (TP + FN)
False alarm rate = FP /(FP + TN)
Precision = TP / (TP + FP)
F-score = 2.Precision.Recall / (Precision}+ Recall)

Dans l'exemple donné précédemment, le taux de fausses alarmes serait nul et la précision serait de 100%, cependant le taux de détection (ou recall) sera null. Ainsi, globalement le F-score sera nul. Le F-score, grâce à sa moyenne harmonique, donne autant de poids aux faux positifs qu'aux faux négatifs.

k plus proches voisins sans MPI

Dans un premier temps vous allez implémenter l'algorithme k-NN de manière séquentielle à partir du code fourni. Les classes `Mail`, `Instances` et `ConfusionMatrix` vous sont fournies en intégralité. En ce qui concerne la classe `Knn` le header (`knn.h`) vous est fourni et vous devez implémenter quelques fonctions (voir ci-dessous) dans le fichier `knn.cpp`.

Exercice 1 : `Insert`

Implémentez la méthode `Insert` :

```
/*
 * The labels of the k nearest neighbours are stored into labels
 * and the corresponding distances in distances.
 * The arrays are sorted according to distances.
 * labels[0] and distances[0] refer to the nearest neighbour
 * labels[k_-1] and distances[k_-1] to the kth nearest neighbour
 *
 * Insert updates labels and distances to include train_mail in the array
 * of the k nearest neighbors if needed
 *
 * If train_mail is as close to mail_to_classify as a current neighbour
 * then it is inserted after this neighbour.
 *
 */
void Knn::Insert(int* labels, int* distances, Mail* mail_to_classify, Mail*
train_mail) const;
```

Vous pouvez tester votre fonction avec les commandes suivantes :

- `make test_knnSequential_Insert`
- `./test_knnSequential_Insert`

On vérifie que le programme donne le résultat escompté :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Train mail is too far away, not inserted
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mail_to_classify: features: [5], label: 0
train_mail: features: [0], label: 1

%%%%% INPUTS %%%%
labels:       0 0 1 0
distances:    1 2 3 4

%%%% OUTPUTS %%%%
labels:       0 0 1 0
distances:    1 2 3 4


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Train_mail is as close as the first current neighbour
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mail_to_classify: features: [5], label: 0
train_mail: features: [4], label: 1

%%%%% INPUTS %%%%
labels:       0 0 1 0
distances:    1 2 3 4

%%%% OUTPUTS %%%%
labels:       0 1 0 1
distances:    1 1 2 3


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Train_mail is as close as the last current neighbour
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mail_to_classify: features: [5], label: 0
train_mail: features: [2], label: 0

%%%%% INPUTS %%%%
labels:       0 1 0 1
distances:    1 1 2 3

%%%% OUTPUTS %%%%
labels:       0 1 0 1
distances:    1 1 2 3
```

Exercice 2 : Nearest Neighbours

Implémentez la méthode `NearestNeighbours`

```
/* labels and distances have been initialised with InitNearestNeighbours
 * Computes the nearest neighbours for mail_to_classify
 * (labels and distances) */
 void Knn::NearestNeighbours(Mail* mail_to_classify, int* labels, int* distances)
const;
```

Toutes les méthodes nécessaires à la fonction `Classify` sont maintenant implémentées. Vous pouvez désormais tester votre knn séquentiel sur les données d'apprentissage `train.txt`, et les données de test `test.txt` avec les commandes suivantes :

- `make launch_knn`
- `./launch_knn train.txt test.txt k` où k represent le nombre de voisins à prendre en compte. Attention, l'exécution peut prendre plusieurs minutes.

Si vous lancez `launch_knn` avec k = 3, vous devriez obtenir des résultats similaires à :

```
              Predicted
              HAM   SPAM
Actual  HAM   605   87
        SPAM  16    292

Error rate              0.103
False alarm rate        0.125723
Detection rate          0.948052
F-score                 0.850073
Precision               0.770449
```

Réduire le taux de fausses alarmes

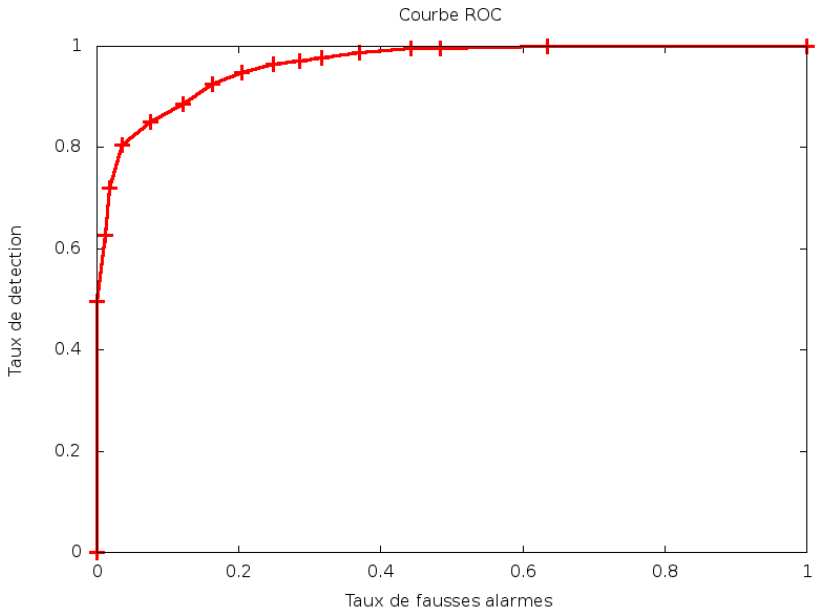
Pour la détection de spams on essaie en général de réduire le taux de fausses alarmes, quitte à avoir un moins bon taux de détection. Nous allons désormais adapter l'algorithme de prédiction afin de réduire le taux de fausses alarmes. Pour cela, on ajoute un seuil de détection. On prédit spam seulement si parmi les k plus proches voisins t% sont des spams. Voici la fonction qui permet de prédire le label en fonction du seuil de détection:

```
int Knn::PredictedLabel(int* labels, double threshold) const {
  int num_spam = 0;
  int num_ham = 0;
  for (int i = 0; i < k_; ++i) {
    switch(labels[i]) {
      case 0:
        num_ham++;
        break;
      case 1:
        num_spam++;
        break;
      default:
        break;
    }
  }
  return num_spam >= threshold * (num_ham + num_spam) ? 1 : 0;
}
```

Grâce à cette nouvelle fonction de prédiction, nous pouvons maintenant exécuter l'algorithme k-NN avec différentes valeurs de seuil de détection. Nous pouvons ainsi tracer la courbe ROC (Receiver Operating Characteristic) représentant le taux de détection en fonction du taux de fausses alarmes. Afin de la réaliser la courbe ROC (pour $k=10$ avec 12 points) exécutez les commandes suivantes:

- `make launch_knn_roc`
- `./launch_knn_roc train.txt test.txt 10 12 > knn_roc.out`
- `gnuplot plot_roc.gnuplot &`

(Le script de gnuplot `plot_roc.gnuplot` utilise la sortie de notre programme, `knn_roc.out`, comme entrée.)
Vous devriez obtenir une courbe semblable à celle ci-dessous:



k plus proches voisins avec MPI

Vous devez maintenant implémenter la fonction `ClassifyMpi` ayant la même signature que `Classify`. Cette fonction doit appliquer k-NN parallélisé. Chacun des p processeurs prend en charge $\frac{1}{p}$ des données d'apprentissage et cherche les k plus proches voisins parmi leurs instances. Ensuite, le processeur racine ($rank$ = 0) fusionne les résultats pour obtenir les k plus proches voisins sur l'ensemble des instances et ainsi prédire le label du mail à classifier.

Remarque : On n'utilise pas de structures de données de la STL (telles que `vector` ou `list`) dans les méthodes privées de la classe `Knn` car les tableaux basiques sont plus faciles à transmettre via MPI.

Exercice 3 : Fusionner les résultats sur le processus racine

Afin de fusionner les résultats sur le processus racine, on commencera par implémenter une méthode `Merge` qui prend deux tableaux triés contenant les labels et les distances des k voisins les plus proches calculés sur des noeuds différents et qui les fusionnent :

```
/*
 * local_distances and local_labels refer to
 * the current k_ nearest neighbours in the root node.
 * other_distances and other_labels refer to
 * the k_ nearest neighbours computed by another node
 * on its training data.
 *
 * Merges the current k_ nearest neighbours in the root node
 * with the k_ nearest neighbours computed by another node.
 * The result is stored in local_distances and local_labels
 *
 * If two neighbours are at equal distance, the one stored
 * in the root node is inserted before the one stored
 * in another node.
 *
 * This method is used in the MPI implementation only
 */
void Knn::Merge(int* local_labels, int* local_distances,
    const int* other_labels, const int* other_distances) const;
```

Vous pouvez tester cette fonction avec les commandes:

- `make test_merge`
- `./test_merge`

On vérifie que le programme donne le résultat escompté :

```
%%%%%%%%%%%%%%%%%
%%%%% INPUTS %%%%
%%%%%%%%%%%%%%%%%
Local:
labels:       0 0 0 0 1
distances:    1 3 5 7 8

Other:
labels:       1 1 1 0 -1
distances:    2 3 5 9 2147483647

%%%%%%%%%%%%%%%%%
%%%% OUTPUTS %%%%
%%%%%%%%%%%%%%%%%
Local:
labels:       0 1 0 1 0
distances:    1 2 3 3 5

Other:
labels:       1 1 1 0 -1
distances:    2 3 5 9 2147483647
```

Exercice 4 : `ClassifyMpi`

Vous pouvez maintenant implémenter `ClassifyMpi`. Vous réutiliserez pour cela les méthodes `Insert`, `InitNearestNeigbours` et `Merge`. Vous n'aurez pas besoin d'appeler les fonctions `MPI_Init` et `MPI_Finalize` car elles seront appelées par le `main` (déjà implémenté dans `launch_knn_mpi.cpp`).

```
/*
 * Returns the predicted label for mail_to_classify
 * This methods uses an MPI version of k-NN
 */
int Knn::ClassifyMpi(Mail* mail_to_classify) const;
```

Plus précisément, le jeu de données sera découpé en blocs, et chaque processus mettra un jour les plus proches voisins du Mail à classifier à partir d'un des blocs. Les deux tableaux `distance` et `label` correspondants seront ensuite envoyés au processus maitre qui effectuera une fusion avec ses propres tableaux `distance` et `label`. Vous compilerez et exécuterez votre code avec les commandes

- `make`
- `mpirun -np 4 ./launch_knn_mpi train.txt test.txt 10`

où `10` représente le nombre de voisins à prendre en compte et 8 le nombre de processus MPI.

- Essayez de changer le nombre de processus pour voir comment évolue le temps d'exécution (par exemple avec 1, 2, 3, 5, 10, 15, 20 et 25 processus).
- Comparez aussi le temps d'exécution avec 1 processus avec la version séquentielle pour mesurer l'overhead induit par la parallélisation de k-NN.