MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies de l'information et de la communication (TIC)

BIO-SELECT

ENSEMBLES DE CLASSIFICATEURS POUR LA SÉLECTION DE VARIABLES DIAGNOSTIQUES AVEC *MICROARRAYS*

Fait par :

Gary MARIGLIANO

Sous la direction de :
Prof. Carlos Andrés PEÑA
A la Haute École d'ingénierie et de gestion du canton de Vaud

Expert externe :

Dr. Vincent GARDEUX

Lausanne, HES-SO//Master, 10 février 2017

Version 0.0.3

Historique

Version	Date	Auteur(s)	Modifications
0.0.1	28.12.16	Gary Marigliano	Création du document
0.0.2	05.02.17	Gary Marigliano	Ajout des chapitres Pipeline, Données, Pré-traitement, Algorithmes, Transmission des listes de features, Déroulement du projet, Reprise du projet
0.0.3	10.02.17	Gary Marigliano	Version rendue

Accepté par la HES-SO // Master (Suisse, Lausanne) sur proposition de

Professeur Carlos Andrés Peña, conseiller du travail de Master
Dr. Vincent Gardeux, expert principal

Lausanne, le 10 février 2017

Professeur Carlos Andrés Peña	Fariba Moghaddam
Conseiller	Responsable de la filière MSE

1. Résumé du document

1.1 Contexte

Ce document est le rapport de travail de Master "BIO-SELECT : Ensembles de classificateurs pour la sélection de variables diagnostiques avec *microarrays*". Un des buts de ce projet est d'obtenir des listes de gènes à partir de plusieurs algorithmes de classification et de tester plusieurs techniques de combinaison de ces listes pour obtenir une "bonne" liste.

En effet, en raison du nombre très élevé de gènes, on souhaiterait pouvoir le réduire notamment pour l'analyse de ces *microarrays* afin de diminuer le coût d'une analyse, voire de l'améliorer.

Pour parvenir à ces objectifs, le travail a été séparé en plusieurs parties. La première consiste à récupérer des datasets et de les pré-traiter. La deuxième s'occupe d'utiliser des algorithmes de classification ou d'analyse statistique pour extraire plusieurs listes de gènes. Enfin, la dernière étape s'intéresse à la mise en place et à l'évaluation de plusieurs techniques de combinaison de ces listes afin d'en obtenir une meilleure.

1.2 Objectifs du projet

1. Extraire des listes de gènes en utilisant plusieurs algorithmes (*machine learning based* ou *statistical based*) ;
2. Développer plusieurs techniques de combinaison de ces listes afin d'en obtenir une meilleure ;
3. Évaluer ces techniques de combinaison.

1.3 Résultats obtenus

Le projet a été matérialisé sous la forme d'une pipeline (chaîne de traitements) dans laquelle on a utilisé des algorithmes pour sélectionner les features les plus importantes. Puis, des techniques de fusion de ces listes de features ont été développées et évaluées. Bien qu'on n'ait pas trouvé de technique de fusion réellement plus efficace par rapport à une autre, on profite, au final, d'une plateforme réutilisable et extensible pour la sélection et la combinaison de features.

Table des matières

2. Introduction

2.1 Contexte

Ce document est le rapport de travail de Master "BIO-SELECT : Ensembles de classificateurs pour la sélection de variables diagnostiques avec *microarrays*". Un des buts de ce projet est d'obtenir des listes de gènes à partir de plusieurs algorithmes de classification et de tester plusieurs techniques de combinaison de ces listes pour obtenir une "bonne" liste.

En effet, en raison du nombre très élevé de gènes, on souhaiterait pouvoir le réduire notamment pour l'analyse de ces *microarrays* afin de diminuer le coût d'une analyse, voire de l'améliorer.

Pour parvenir à ces objectifs, le travail a été séparé en plusieurs parties. La première consiste à récupérer des datasets et de les pré-traiter. La deuxième s'occupe d'utiliser des algorithmes de classification ou d'analyse statistique pour extraire plusieurs listes de gènes. Enfin, la dernière étape s'intéresse à la mise en place et à l'évaluation de plusieurs techniques de combinaison de ces listes afin d'en obtenir une meilleure.

2.2 Introduction rapide à la problématique des microarrays

Avant de se lancer dans la sélection de features, il peut être intéressant de savoir d'où proviennent les données et comment elles ont été produites. Les paragraphes qui suivent sont tirés de l'article Wikipédia "Puce à ADN"[38]. Les datasets utilisés contiennent des *samples*. Il y a des *samples* sains et d'autres malades. Chaque *sample* contient le niveau d'expression génétique d'une cellule ou d'un tissu, d'un organe... Ces niveaux d'expression génétiques peuvent être utilisés pour mesurer le dysfonctionnement de gènes en cas de maladie.

Le principe général, presque caricatural, de la procédure est décrit comme suit :

1. A partir de l'ADN, on isole l'ARN messager (*mRNA* en anglais). Cette opération est appelée transcription [1] ;
2. Ces mRNA sont ensuite transformées en ADN complémentaire (*cDNA* en anglais) en utilisant une technique appelée rétrotranscription ;
3. Les cDNA sont marquées grâce à une substance fluorescente généralement verte et rouge afin de distinguer les échantillons à tester (aussi appelées cibles, ceux où il y a la maladie par exemple) et les échantillons de référence. Cette étape se nomme le marquage (*labelling* en anglais) ;
4. Cette étape s'appelle l'hybridation. Les cibles marquées (cDNA) sont ensuite mises au contact avec la puce à ADN (*microarray* en anglais). Cette puce contient des sondes à sa surface sur lesquelles vont s'accrocher les cibles. Chaque brin de cDNA va chercher son complément sur la sonde pour reformer une structure à double hélice ;
5. On mesure ensuite le niveau d'expression en regardant la couleur – tantôt verte, tantôt rouge ou encore un mélange orangé – de chaque sonde pour connaître les impacts de la maladie, par exemple.

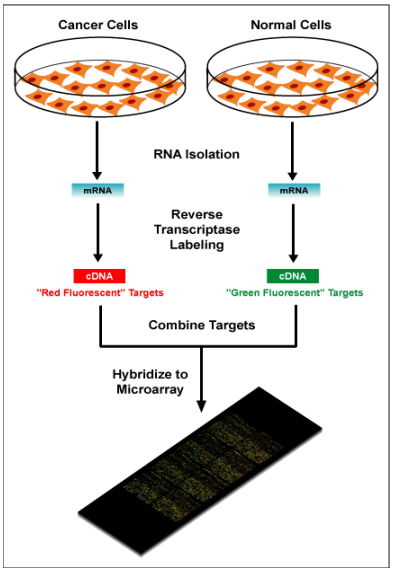

FIGURE 2.1 – Étapes principales d'une expérience utilisant un *microarray*

1. Transcription ADN : `https://fr.wikipedia.org/wiki/Transcription_(biologie)`

Concrètement, pour ce projet, on dispose de fichiers contenant les niveaux d'expression de la dernière étape. L'idée étant d'entraîner des algorithmes à utiliser ces niveaux pour décider si l'échantillon appartient à la classe saine ou malade. Puis, une fois entraînés, identifier quels ont été les principaux gènes utilisés pour permettre cette classification.

La figure 2.1 montre schématiquement les étapes 1 à 4 décrites ci-dessus. Les mêmes étapes et surtout la 5^{e} sont illustrées à la figure 2.2.

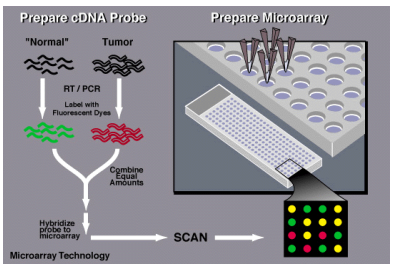

FIGURE 2.2 – Étapes principales d'une expérience utilisant un *microarray*

2.3 Objectifs

1. Extraire des listes de gènes en utilisant plusieurs algorithmes (*machine learning based* ou *statistical based*) ;
2. Développer plusieurs techniques de combinaison de ces listes afin d'en obtenir une meilleure ;
3. Évaluer ces techniques de combinaison.

3. Pipeline

Avant de rentrer dans le vif du sujet, il convient d'introduire le concept de pipeline et d'en voir la structure.

Une pipeline est une chaîne de traitements séquentiels dont chaque maillon a une tâche spécifique.

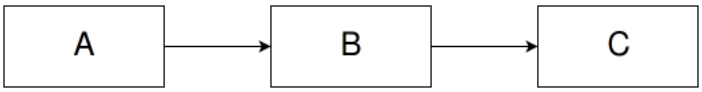

FIGURE 3.1 – Exemple de pipeline basique

3.1 Pipeline du projet

Dans le cadre de ce projet, la pipeline suivante a été conçue :

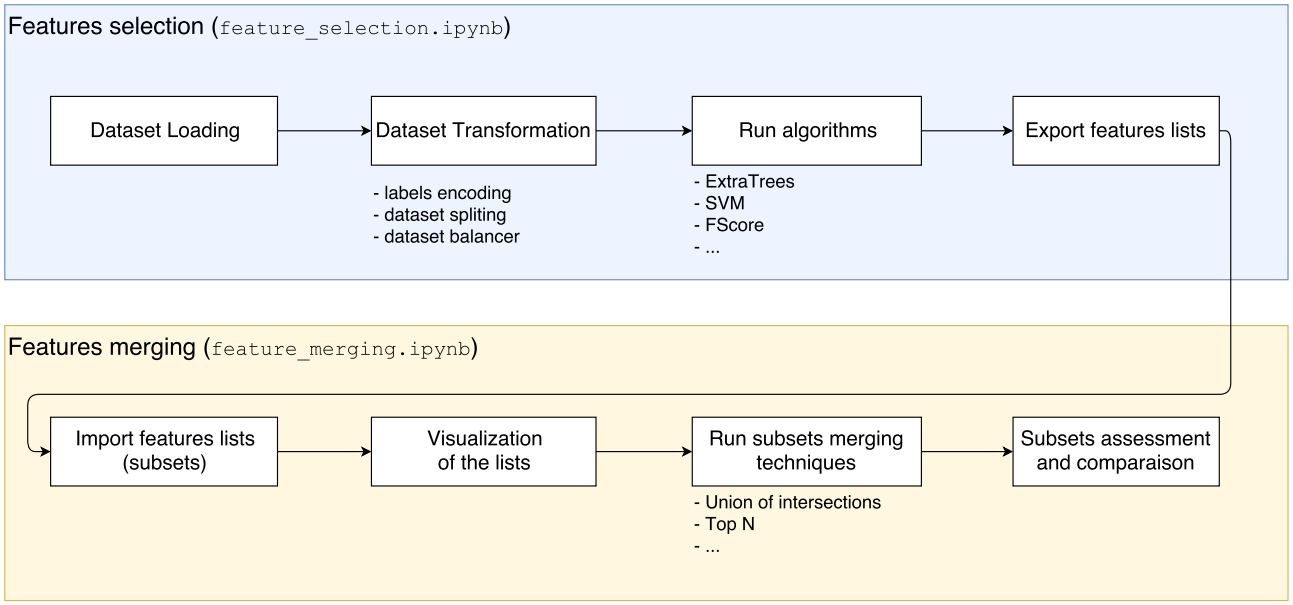

FIGURE 3.2 – Pipeline du projet

La pipeline est composée de deux parties. La première, "features selection", s'occupe de charger un dataset, de lancer plusieurs algorithmes fournissant chacun une liste de features qu'il juge pertinent et d'exporter ces listes dans des fichiers CSV. La seconde, "features merging", s'intéresse à la combinaison de ces listes pour en obtenir une meilleure. Pour ce faire, on va importer les listes, les visualiser puis essayer et comparer plusieurs techniques de *merging* (combinaison ou fusion, en français).

3.2 Choix et implémentation de la pipeline

Cette pipeline a été implémentée en Python grâce à, notamment, Jupyter[37] et Scikit-learn[78].

Scikit-learn a été choisi car il dispose d'une documentation de qualité et parce qu'il est en Python, langage appris à l'école. Jupyter a été retenu car il permet de rapidement faire du prototypage, idéal pour ce genre de projet où l'on doit tester une multitude de techniques pour un problème donné. Il permet également d'afficher simplement des graphes et autres commentaires dans un seul *notebook*.

Ainsi, ces *notebooks* peuvent être lus facilement, y compris par quelqu'un n'ayant pas de connaissances en Python. Enfin, les notebooks produits pour ce projet respectent le principe de séparation "modèle"/vue. Ceci permet d'écrire uniquement du code de présentation et de *workflow* dans les *notebooks* afin de pouvoir être lu comme on lirait une procédure et de l'autre côté, on écrit le code métier (les algorithmes principalement) dans des classes Python standard.

Comme on peut le voir avec la figure 3.2, on dispose de deux *notebooks* Jupyter. Ceci permet de séparer la génération des listes de features de la combinaison de ces dernières. Ainsi, on peut travailler sur les techniques de combinaison sans devoir réexécuter les algorithmes de sélection de features.

Ces *notebooks* – fichiers `.ipynb` – et leurs sorties – fichiers html – sont disponibles en annexes.

Le langage R aurait lui aussi pu être sélectionné car très utilisé dans le domaine du *machine learning* et du biomédical. Cependant, il a été jugé qu'il était plus judicieux d'utiliser Python car le langage de programmation pour ce projet ne doit pas être une difficulté supplémentaire. De plus, il est plus pertinent, il semble, de se concentrer sur les problématiques principales du projet.

4. Données

4.1 Datasets choisis

Plusieurs datasets ont été proposés au début du projet. Deux ont été retenus ; MILE et Golub car ils sont populaires. On note que l'architecture de code du projet permet facilement l'ajout de nouveaux datasets grâce à sa modularité.

4.1.1 MILE

MILE ou *Microarray Innovations in LEukemia (MILE) study : Stage 1 data* est un dataset qui regroupe des *samples* de personnes saines et de personnes atteintes de leucémie. Initialement, il y a 18 classes dont 17 variantes de leucémie. Afin d'avoir des classes qui soient plus équilibrées, il a été décidé de réduire les leucémies aux 4 types principaux communément reconnus. Ces 4 types sont les suivants [1] : ALL, CLL, AML et CML. Il y a donc 6 classes, les 4 types de leucémie, MDS (un autre type de cancer) et la classe représentant les personnes saines.

En terme de taille, MILE contient 2096 *samples* de 54675 gènes. Il est téléchargeable à cette adresse : `http://www.ebi.ac.uk/arrayexpress/experiments/E-GEOD-13159/`.

Pour *parser* le dataset, on dispose d'un fichier de référence (`mile-sample-and-data.csv`), qui fait le lien entre le nom du *sample* et la classe assignée, et les nombreux fichiers (par exemple : `GSM329407_sample_table.txt`) *samples* contenant les expressions génétiques. Voici à quoi ressemble un fichier *sample* de MILE.

```
1 ID_REF              VALUE_DS  ABS_CALL  DETECTION P-VALUE   VALUE
2 AFFX-BioB-5_at      1019.38   P         0.000662269
3 AFFX-BioB-M_at      1099.16   P         4.42873e-05
4 AFFX-BioB-3_at      688.488   P         0.00010954
5 [...]
6 1007_s_at           257.18    P         0.00998544          0.361345559358597
7 1053_at             626.218   P         0.000959383         0.450274109840393
8 117_at              525.837   P         0.000265621         0.43117818236351
9 121_at              407.935   P         0.0393654           0.405516684055328
```

Il s'agit donc de fichiers CSV. Les premières lignes, celles qui commencent par `AFFX-` sont des sondes, il faut donc les ignorer. La valeur qui est retenue pour l'analyse est la colonne `VALUE`.

4.1.2 Golub

Golub ou *Golub et al. (1999)* est un dataset proposant 72 *samples* contenant 2 classes (AML et ALL). Les gènes sont au nombre de 7129. Ce dataset a été téléchargé à cette adresse : `http://eps.upo.es/bigs/datasets.html`. Ce fichier est au format AARF qui est utilisé par le logiciel Weka [2]. Il a donc fallu le convertir en CSV pour pouvoir utiliser les *samples* dans ce projet. Pour ce faire, un script a été développé et est disponible à l'appendice B. Ce script va donc lire les fichiers AARF et créer des fichiers CSV – un par *sample*, suffixé par sa classe – et les ranger dans deux dossiers, `train` et `test` conformément aux fichiers téléchargés, `leukemia_train_38x7129.arff` et `leukemia_test_34x7129.arff`.

1. 4 types principaux de leucémie : `https://en.wikipedia.org/wiki/Leukemia`
2. Weka : `http://www.cs.waikato.ac.nz/ml/weka/`

On dispose désormais de fichiers CSV mais contrairement à MILE, on ne dispose pas d'un fichier de référence alliant *samples* et classes, mais ces dernières sont directement inscrites dans le nom du fichier du *sample* (par exemple : `sample_6_ALL.csv`). De plus, le dataset est déjà séparé en *train set* et *test set*. Il a été décidé de fusionner ces deux sets et de les séparer plus tard, au moment du pré-traitement.

```
ID_REF              VALUE
AFFX-BioB-5_at      -342.0
AFFX-BioB-M_at      -200.0
AFFX-BioB-3_at      41.0
[...]
AB002318_at         1233.0
AB002365_at         207.0
AB002366_at         0.0
AB002380_at         43.0
AB002382_at         767.0
```

FIGURE 4.1 – Fichier *sample* de Golub

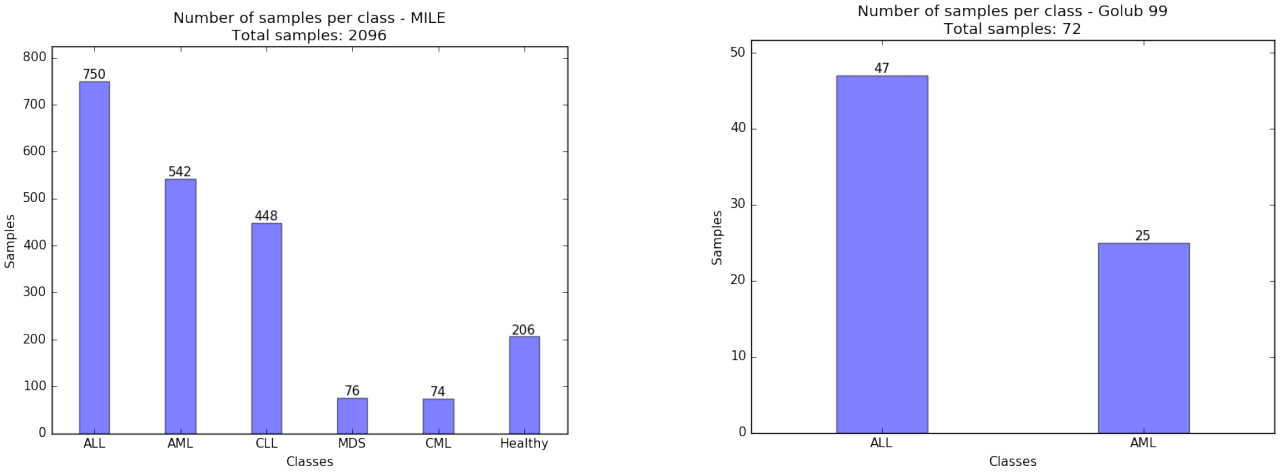

FIGURE 4.2 – Visualisation des datasets

Comme on peut le voir avec la figure 4.2, les classes ne sont pas totalement équilibrées. Ce problème sera mitigé dans le chapitre Pré-traitement.

4.2 Implémentation

Dans le code, ce sont les classes `MileDataset`, `GolubDataset`, `MileDatasetSampleParser` et `GolubDatasetSampleParser` qui sont responsables du *parsing*.

Pour les tests unitaires, un dataset fictif, appelé *DummyDataset*, a été créé. Il contient de fausses données et permet de valider rapidement le bon fonctionnement de classes utilisant les datasets.

5. Pré-traitement

Avant de pouvoir travailler avec les datasets, il convient de les pré-traiter. En plus des opérations classiques qui consistent par exemple à éliminer les valeurs manquantes, il est nécessaire de procéder à des opérations de pré-traitement propre au *machine learning*.

5.1 Encodage des labels en nombres

Quand on travaille avec des datasets, il se peut que les classes (labels) soient des catégories textuelles (comme `"ALL"`, `"AML"`, ...). Cependant, plusieurs algorithmes demandent que les classes soient des nombres pour pouvoir fonctionner [1].

Il faut donc encoder ces labels en nombre. Pour ce faire, la classe `LabelEncoder` de scikit-learn[78]. Cette opération a été enveloppée dans la classe `DatasetEncoder`.

En revanche, il faut bien veiller à comparer les classes deux à deux, par exemple en utilisant la technique *One versus One*, afin d'éviter de donner un sens ou un ordre à ces labels devenus nombres.

5.2 Séparation en *train* et *test* sets

Les datasets ont été séparés en *train set* et *test set* afin de d'obtenir un score après avoir entraîné les algorithmes de *machine learning*. Ce score peut être utilisé comme mesure de confiance par rapport à la liste de features renvoyée ou permet d'écarter un algorithme peu performant.

La pipeline étant séparée en deux grandes parties – *features selection* et *features merging* – il est nécessaire de garder la même séparation train/test pour les deux parties. En effet, si on vient à refaire une séparation train/test dans la partie *features merging* (pour l'évaluation des techniques de merging notamment), on pourrait avoir des samples de tests qui ont été utilisés pour l'entraînement dans la partie *features selection*. Ceci biaise le résultat dans le sens où l'on a déjà vu ces samples lors de l'entraînement.

Par conséquent, il a été décidé d'utiliser Pickle [2] pour sérialiser le dataset – et donc le split effectué – afin de pouvoir le réutiliser tel quel dans la partie *features merging*.

5.3 Équilibrage des classes

Comme on l'a aperçu à la figure 4.2 du chapitre 4, les classes ne sont pas équilibrées, ce qui peut poser problème. En effet, si une classe est largement plus représentée qu'une autre, alors il suffit de bien classifier cette classe pour obtenir une bonne précision (au sens statistique du terme [3]).

Il convient donc d'équilibrer, artificiellement, ces classes. Plusieurs techniques existent[68] et principalement deux : *l'over-sampling* et *l'under-sampling*. Le premier consiste à augmenter le nombre de *samples* pour une classe donnée grâce à une technique plus ou moins élaborée. Par exemple, on peut simplement dupliquer les échantillons présents jusqu'à obtenir un équilibre. Pour *l'under-sampling* le fonctionnement est inverse ; on va réduire le nombre de *samples* pour la/les classes la(les) plus représentée(s).

1. Voir commentaire sur StackExchange : `http://stats.stackexchange.com/a/134562`
2. Pickle : `https://docs.python.org/2/library/pickle.html`
3. `https://en.wikipedia.org/wiki/Precision_and_recall`

Dans ce projet, on a choisi d'utiliser une technique *d'over-sampling* car, avec MILE par exemple, on a une différence notable entre les classes en termes de nombre de *samples*. En effet, l'utilisation de *l'under-sampling* ne laisserait que 444 *samples* au total. $444 = 74 * 6$ où 74 est le nombre de *samples* pour la classe la moins représentée (CML) au lieu des 2096 *samples* initiaux.

Dans le code, c'est la classe `DatasetBalancer` qui est responsable de cette opération.

Remarque : Dans la pratique, on a décidé que l'équilibrage des classes ne se ferait que sur le *train set* afin de garder un nombre de *samples* raisonnables et parce que le *test set* ne doit pas être modifié. De plus on n'a pas besoin de faire cette opération étant donné que la performance brute de classification des algorithmes n'est pas le critère le plus important. On s'intéresse d'avantage aux listes renvoyées par l'algorithme qu'à sa performance.

6. Algorithmes

On rappelle que l'idée générale du projet est d'utiliser plusieurs algorithmes, si possible relativement différents entre eux, et de combiner les features les plus importantes selon eux.

On distingue deux types d'algorithmes ; les algorithmes basés *machine learning* et les algorithmes basés statistiques. Les premiers utilisent les données comme exemples afin de pouvoir les reconnaître dans une deuxième phase, il s'agit donc d'un processus itératif. Les seconds analysent les données, notamment la variance pour obtenir une liste d'importance des features. Ces derniers sont bien souvent déterministes.

Ce chapitre énonce les critères de sélection pour les algorithmes et présente ceux qui ont été utilisés dans ce projet.

6.1 Critères de sélection des algorithmes

Les critères de sélection d'un algorithme sont les suivants :

- Doit pouvoir fournir une liste d'importance des features utilisées. Cette liste peut être :
  - accompagnée d'un **score** par feature. Elle sera triée avec la feature ayant le meilleur score en premier ;
  - classée. Chaque feature a un **rang**, plus le rang de la feature est haut (commence à 0), plus la feature apparaîtra au début de la liste ;
  - **plate**. La position de la feature dans la liste n'a pas d'importance ;

  Évidemment, une liste avec un score peut être transformée en une liste classée qui peut elle-même être aplatie.
- Doit pouvoir gérer plusieurs classes à la fois. Pour y remédier, on peut utiliser une technique de *One versus One* ou *One versus All* ;
- Doit, si possible, être déterministe ou, sinon, donner des résultats – donc des listes de features – stables, c'est-à-dire qui sont similaires entre deux lancements de l'algorithme.

Remarque : Il convient de différencier les termes *features selection* et *dimensionality reduction*. Le premier consiste à extraire un sous-ensemble des features originales, c'est l'objectif souhaité. Le second cherche à réduire la dimension de l'espace d'entrée. Cela peut se faire en sélectionnant un sous-ensemble de features – ce qui vient à faire de la *features selection* – ou en transformant cet espace d'entrée en un plus petit. La différence est importante car ces nouvelles dimensions n'ont plus le même sens que les features originales et donc on ne peut pas les extraire. C'est pour cette raison que l'utilisation de PCA n'a pas été entreprise.

6.2 Algorithmes utilisés

- ExtraTrees [1] ;
- ReliefF [2] ;
- Fisher Score [3] ;
- SVM [4] ;
- SVM-RFE [5] ;
- GAANN : voir 6.3.5 ;

1. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html
2. http://featureselection.asu.edu/html/skfeature.function.similarity_based.reliefF.html
3. http://featureselection.asu.edu/html/skfeature.function.similarity_based.fisher_score.html
4. http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
5. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

— CFS [6] ;
— MRMR [7].

6.3 Principes généraux de certains algorithmes

Le but de cette section est de donner un aperçu du fonctionnement de certains algorithmes utilisés. On explique les algorithmes qui ont été implémentés personnellement et ceux qui sont jugés peu communs.

6.3.1 ReliefF

Relief et son extension multi-classes ReliefF sont des algorithmes qui mesurent les poids des features. Il est univarié et déterministe. Les algorithmes vont sélectionner les features qui contribuent à la séparation des *samples* des différentes classes[90].

6.3.2 Fisher Score

Fisher Score est un algorithme statistique qui met en avant les features qui sont discriminantes. Il assigne le score le plus haut à la feature pour laquelle les points des différentes classes sont les plus loins des autres, tout en veillant à ce que les points des mêmes classes soient les plus proches possibles[64].

6.3.3 CFS

CFS (*Correlation-based Feature Selection*) est un algorithme de type filtre multivarié qui donne une liste triée en utilisant une heuristique (appelée mérite) des features sélectionnées. Il va sélectionner les features qui sont hautement corrélées à une classe et ignorer celles qui ne le sont pas. Les features redondantes devraient donc être éliminées [56].

6.3.4 MRMR

MRMR (*Minimum Redundancy Maximum Relevance*) est une méthode de type filtre multivarié qui sélectionne les features les plus pertinentes pour une classe donnée et qui sont les moins redondantes possibles. Les critères d'optimisation (pour Minimum Redundancy et Maximum Relevance) sont basés sur l'information mutuelle[56][90].

6.3.5 Réseaux de neurones

Un perceptron multicouche est un type de réseau de neurones composé de couches, elles-mêmes composées de neurones. Chaque neurone possède un poids synaptique pour chaque neurone de la couche suivante auquel il est connecté.

Extraire une liste de features en utilisant un réseau de neurones peut se faire de plusieurs manières. Une technique consiste à entraîner le réseau puis à interpréter les valeurs des poids synaptiques. On peut, par exemple, lire uniquement les valeurs des poids de la première couche, celle directement après la couche d'entrée, voir figure 6.1. Or, si un poids synaptique est faible pour une feature dans la première couche, il se peut que cette dernière devienne importante dans les couches suivantes et on risquerait de passer à côté d'une bonne feature. Inversement, un poids synaptique élevé dans la première couche ne garantit pas d'avoir une bonne feature en sortie.

6. `http://featureselection.asu.edu/html/skfeature.function.statistical_based.CFS.html`
7. `http://featureselection.asu.edu/html/skfeature.function.information_theoretical_based.MRMR.html`

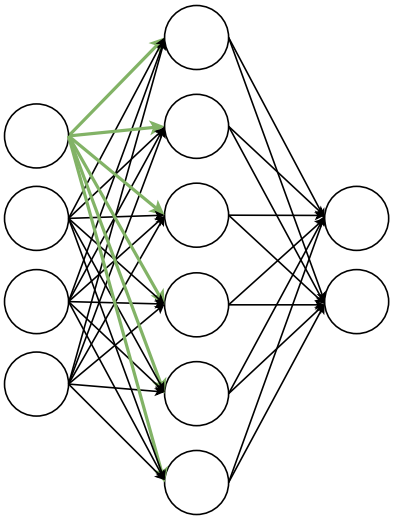

FIGURE 6.1 – Les lignes en vert représentent les poids pour la première couche de la première feature

En gardant l'idée d'utiliser les poids synaptiques, on pourrait également utiliser l'ensemble des poids synaptiques d'un chemin du réseau, voir figure 6.2. Il faut ensuite utiliser une méthode permettant d'évaluer ce chemin et de lui attribuer un score. Une technique simple consisterait à utiliser la moyenne des poids du chemin. Cependant, l'idée même d'utiliser les poids synaptiques est discutable[86] car il est difficile de choisir ou de trouver une méthode qui met correctement en avant les features importantes.

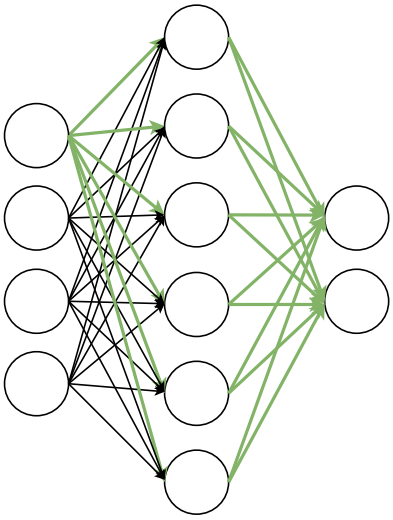

FIGURE 6.2 – Les lignes en vert représentent les poids pour la première feature

Il existe cependant un autre type de méthode pour obtenir une liste des features à partir d'un réseau de neurones (ou tout autre algorithme, en réalité). L'idée est d'envelopper un algorithme dans une boucle, voir figure 6.3. A chaque passage de la boucle, on soumet à l'algorithme, une ou plusieurs liste(s) de feature qui sont différentes de la boucle précédente et on utilise la sortie du *classifier*, son score par exemple, pour juger de la qualité des entrées (la/les liste(s) de features) et ainsi converger vers de meilleures listes.

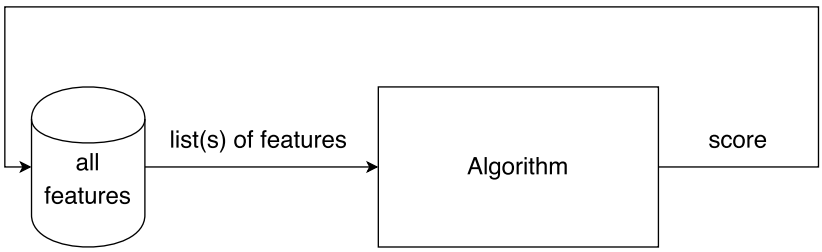

FIGURE 6.3 – L'algorithme est alimenté par des *subsets* de features à chaque itération

GAANN

Pour ce projet, il été décidé de développer un algorithme itératif comme expliqué ci-dessus. Cet algorithme a été nommé GAANN, pour *Genetic Algorithm Artificial Neural Network*. En effet, cet algorithme utilise un algorithme génétique qui va fournir des listes à un réseau de neurones. Au fil des générations, on obtiendrait de meilleures listes et on ne garderait que la meilleure.

On utilisera les termes suivants pour la suite de cet algorithme :

— **individu :** Un individu est une liste plus petite que la liste complète des features à disposition. Par exemple pour le dataset Golub, la liste des features complète est composée d'environ 7000 features. Un individu est une liste composée de, par exemple, 100 features uniques à partir de la liste complète. Un autre moyen de coder un individu aurait été de représenter les features à garder sous forme d'une liste de booléens. Problème : les listes seraient très grandes (autant grandes que le nombre de gènes du dataset) et contiendraient principalement des valeurs à `false` ;

— **population :** Une population regroupe un ensemble d'individus. A chaque génération, c'est l'ensemble de la population qui est évaluée.

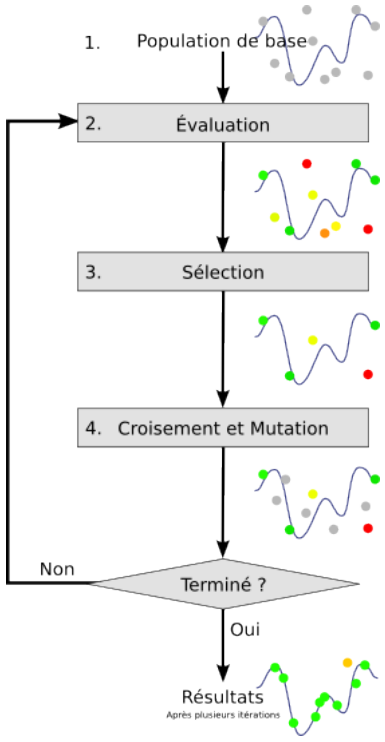

FIGURE 6.4 – *Rappel :* Fonctionnement général d'un algorithme génétique [8]

8. source : `https://upload.wikimedia.org/wikipedia/commons/4/42/Schema_simple_algorithme_genetique.png?uselang=fr`

Méthode de sélection La méthode de sélection utilisée pour l'algorithme génétique est la méthode dite de la roulette [9].

Méthode de croisement Il s'agit de combiner deux individus, appelés mâle et femelle dans le code afin de créer un nouvel individu, appelé enfant. Le croisement consiste à prendre l'union des features des parents, puis d'ajouter autant de features que nécessaire afin de garder un enfant avec une taille de liste équivalente à celle du reste de la population. Les features ajoutées sont tirées aléatoirement en veillant à ce qu'elles ne soient pas déjà présentes dans l'union des parents.

Méthode de mutation Afin de conserver une certaine diversité dans la population, on utilise une méthode de mutation. Concrètement, on replace certaines features par d'autres tirées aléatoirement. Le taux de mutation a été fixé à 0.05.

Élitisme Afin de ne pas perdre le meilleur individu d'une population, on choisit de l'introduire dans la génération suivante.

Méthode de fitness Comme expliqué plus haut, le score d'un individu est directement lié au score du classifier vis-à-vis de la sous-liste de features utilisée. Pour plus de robustesse, l'individu est cross-validé 3 fois et la médiane de ses scores est retenue. Plus le score est haut, meilleur est l'individu. La méthode de score utilisée est F1-score. Ceci permet d'éviter de se baser uniquement sur l'*accuracy* absolue, ce qui peut mener à une mauvaise performance avec l'utilisation de datasets déséquilibrés.

L'analyse de cet algorithme est proposée à la section 7.5.

6.3.6 Limma

Limma est un *package* R pour l'analyse de l'expression des gènes à partir de *microarrays*. Il est utilisé dans ce projet pour extraire une liste de features. Pour utiliser ce package, il faut disposer de R, chercher un *wrapper* Python ou une librairie native en Python de Limma. Après plusieurs recherches, aucune librairie native n'a été trouvée et utiliser un *wrapper* Python pour appeler du code R n'a pas été retenu.

Il a donc été décidé d'apprendre les bases de R afin de pouvoir utiliser Limma. Après cette première difficulté, il a fallu prendre en main le package Limma et charger les datasets à analyser.

Pour Golub, il existe un package[62] qui contient l'ensemble des *samples* du dataset. Limma a été lancé sur `golubMerge`.

Pour MILE, c'est plus compliqué. Il a d'abord été tenté d'utiliser les fichiers bruts (CEL)[17] mais il était impossible de tout charger en mémoire RAM à la fois. Finalement, au lieu d'utiliser les fichiers bruts, on a utilisé une matrice contenant le résumé [10] de l'ensemble des expressions génétiques pour tous les *samples*. Le script présent dans le code source a été adapté du script généré par l'outil GEO2R [11]. Cet outil en ligne permet également de spécifier la *design matrix*, c'est-à-dire quelles sont les données à analyser et à quelles classes elles appartiennent. De manière similaire à ce qui avait été fait à la section 4.1.1, on a réduit le nombre de classes de MILE pour n'en garder que 6, à savoir les 4 types principaux de leucémie, MDS et le type représentant les personnes saines.

9. `https://en.wikipedia.org/wiki/Fitness_proportionate_selection`
10. Qui pèse tout de même 800 MB !
11. GEO2R : `https://www.ncbi.nlm.nih.gov/geo/geo2r/?acc=GSE13159`

Limma renvoie un tableau contenant les features les plus importantes sous la forme d'un fichier CSV. Ce dernier sera parsé en Python et la liste extraite sera ajoutée à la liste des features des algorithmes en Python. Dans la pipeline (voir 3.1), on se situe dès lors entre la partie *features selection* et la partie *features merging*.

```
1 "ID"          "logFC"  "AveExpr"  "t"       "P.Value"    "adj.P.Val"   "B"
2 "M23197_at"   2.258    8.058      11.530    3.503e-18    2.324e-14     30.745
3 "M31523_at"   -2.073   9.512      -10.800   7.454e-17    2.473e-13     27.814
4 "U46499_at"   4.161    7.432      10.494    5.305e-16    1.173e-12     25.921
5 "D88422_at"   3.708    8.151      10.130    1.738e-15    2.883e-12     24.785
6 "M27891_at"   4.639    9.877      10.148    2.249e-14    2.985e-11     22.284
7 "X95735_at"   3.046    9.638      9.596     4.566e-14    4.606e-11     21.638
8 "M11722_at"   -4.517   10.089     -9.507    4.860e-14    4.606e-11     21.539
```

FIGURE 6.5 – Exemple de tableau contenant les features les plus importantes pour Golub. Seules les colonnes ID – indiquant le nom des gènes – et B – trié du plus grand au plus petit – sont utilisées.

Une autre approche similaire aurait été d'utiliser SAM ou *Significance Analysis of Microarrays* qui est est un package R[47]. Limma lui a été préféré car on a pu le faire fonctionner et parce qu'il y avait plus de documentation.

6.4 Implémentation

Dans cette section, on s'intéresse à l'implémentation des algorithmes, aux librairies utilisées et aux différentes techniques testées durant le projet.

6.4.1 Librairies utilisées

Comme annoncé à la section 3.2, Python a été choisi conjointement à la librairie scikit-learn. Cette librairie permet de rapidement utiliser des algorithmes courants comme SVM, ExtraTrees ou encore des MLP (*Multilayer Perceptrons*). De plus, cette librairie offre des outils pour le pré-traitement des données comme l'encodage des classes en labels numériques. Cependant, elle dispose de peu d'algorithmes liés à la *feature selection*. On peut retrouver ici [12] un article dans la documentation de scikit-learn listant les possibilités proposées par la librairie.

On a donc complété scikit-learn avec une autre librairie appelée scikit-feature[25]. Cette dernière offre plusieurs implémentations d'algorithmes de *feature selection* telles que CFS, MRMR, F-Score, Gini-index, etc... La liste complète des algorithmes proposées se trouve ici [13].

6.4.2 Abstraction et hiérarchie de classes

Afin de pouvoir être utilisé de façon transparente — c'est-à-dire sans se préoccuper de l'implémentation en arrière-plan — une hiérarchie de classe a été mise en place. Ainsi tous les algorithmes développés ou importés de librairies possèdent la même structure. Il est également possible d'ajouter son propre algorithme relativement facilement.

12. `http://scikit-learn.org/stable/modules/feature_selection.html`
13. Liste de algorithmes proposé par scikit-feature : `http://featureselection.asu.edu/algorithms.php`

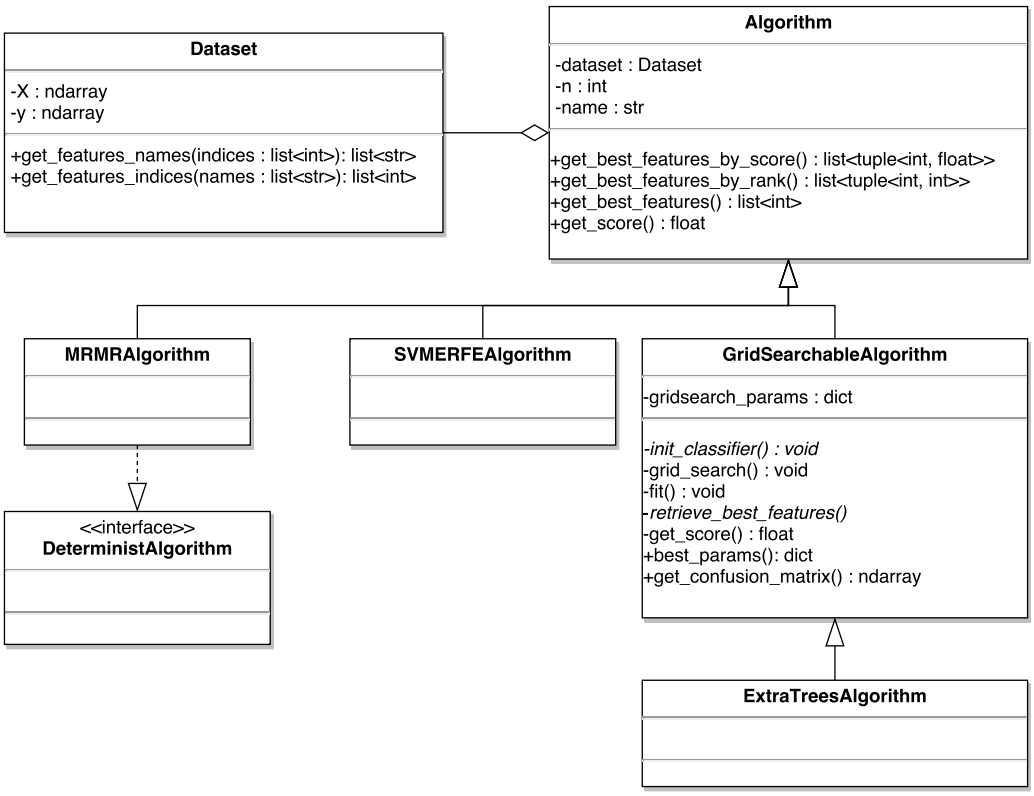

FIGURE 6.6 – Diagramme de classes non-exhaustif des algorithmes

Comme on peut le voir sur la figure ci-dessus, il y a la classe `Dataset` qui va contenir les données du dataset. Ces données ont été pré-traitées et séparées en *train* et *test* set comme expliqué au chapitre 5. C'est un objet de cette classe qui va nourrir un objet `Algorithm`.

La classe `Algorithm` est une classe abstraite qui fournit une structure et des méthodes communes aux algorithmes. Parmi ces méthodes, on note celles commençant par `get_best_featuresXXX()`. Elles correspondent aux différents types de listes de features évoqués à la section 6.1. La méthode `get_score()` permet, si l'algorithme le supporte, d'obtenir le score ou la performance de l'algorithme de manière générale, par exemple, le score obtenu par le *classifier*. Il peut être utilisé comme indice de confiance ou comme pondération quand il s'agira de choisir si on souhaite garder la liste fournie par l'algorithme en question.

L'interface `DeterministAlgorithm` indique si l'algorithme utilisé est déterministe, dans le sens où deux exécutions avec les mêmes paramètres donnent la même liste. Sur la figure 6.6, on voit, par exemple, que l'algorithme MRMR est déterministe. Cette information est utile car elle permet de ne pas devoir exécuter plusieurs fois l'algorithme pour obtenir une certaine stabilité dans les listes renvoyées par les algorithmes. Plus d'informations sont données à la section 6.4.3.

Certains algorithmes nécessitent des paramètres lors de leur initialisation. Le choix manuel de ces paramètres étant fastidieux, il est commun de recourir à une technique appelée *Grid Search*. Cette dernière consiste simplement à essayer l'ensemble des combinaisons des paramètres fournis par l'utilisateur de l'algorithme. Dans le cadre de ce projet, on note que l'algorithme `ExtraTreesAlgorithm` utilise cette technique de *Grid Search*. La classe abstraite `GridSearchAlgorithm` permet d'appliquer automatiquement un grid search à un algorithme (méthode `grid_search()`), si pertinent, mais laisse également la possibilité d'utiliser cet algorithme sans utiliser le grid search (méthode `fit()`).

Enfin, certains algorithmes comme `SVMRFEAlgorithm` ne sont pas déterministes et n'ont pas recours au grid search. En effet, dans le cas de `SVMRFEAlgorithm`, on considère que les itérations internes de l'algorithme (la partie RFE [14]) sont déjà bien assez coûteuses en terme de temps de calcul pour y ajouter une partie grid search.

6.4.3 Stabilité des listes données par les algorithmes

Pour les algorithmes non déterministes, il convient de s'assurer que les listes de features renvoyées soient relativement stables. On entend par stable une liste qui changerait peu d'un lancement de l'algorithme à un autre. Ainsi, pour atténuer l'effet aléatoire qui pourrait se dégager d'un algorithme, une stratégie de moyennage de listes a été mise en place.

Critères de sélection pour une méthode de moyennage de listes

1. On veut favoriser les features qui sont sélectionnées plusieurs fois par le même algorithme ;
2. Si deux features ont été sélectionnées le même nombre de fois, on veut pouvoir les départager (en utilisant le rang ou le score) ;
3. On préfère mettre en avant les features sélectionnées plusieurs fois (même si elles ont un moins bon rang/score) plutôt que celles sélectionnées qu'une seule fois mais ayant un grand score (ou plus petit rang).

Méthodes de moyennage utilisées

Pour obtenir cette moyenne, on lance l'algorithme plusieurs fois avec les mêmes données d'entrées (les mêmes *samples*). Puis on applique une formule permettant d'obtenir un score par feature (à ne pas confondre avec le score par feature renvoyé par la méthode `get_features_by_score()` qui renvoie le score de la feature selon l'algorithme utilisé). Cette formule ou plutôt ces formules — car ce ne sont pas les mêmes suivant les types de listes renvoyées (liste simple, liste classée, liste avec scores) — sont les suivantes :

Listes de features simples

il s'agit simplement d'un compteur d'occurrence. Le score de la feature est égal au nombre de fois où elle a été retenue par l'algorithme. F représente le nombre de features au total dans une liste et L représente le nombre de fois qu'on a relancé l'algorithme, c'est-à-dire le nombre de listes à moyenner.

$$^{15}score(f) = \sum_{l=1}^{L} \sum_{i=1}^{F} [f_i = f] \tag{6.1}$$

Visuellement, le processus est imagé à la figure 6.7. Au final, on dispose donc d'une liste de tuples (feature, #occurrences).

14. Voir `http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html`
15. Crochet d'Iverson : `https://en.wikipedia.org/wiki/Iverson_bracket`

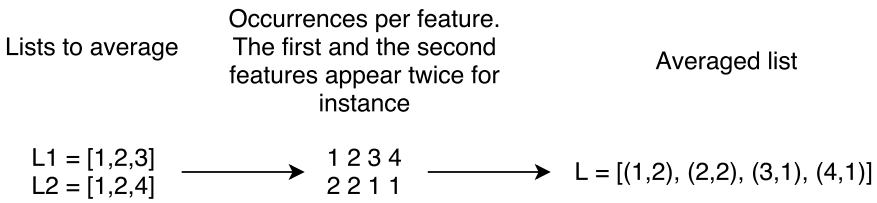

FIGURE 6.7 – Moyennage de listes simples

Listes de features par score

Pour les listes de features par score, on dispose de listes de tuples ordonnées, c'est-à-dire avec la feature ayant le plus grand score apparaissant en premier. On peut voir un exemple de ces listes à la figure 6.8. Par exemple, pour la liste `L1`, les features 1, 2, 6 et 3 ont été sélectionnées et leur score sont respectivement 0.9, 0.8, 0.6, 0.4.

Rappel : Il ne faut pas confondre les scores par feature renvoyés par un algorithme – normalisé entre 0 et 1 – et le score moyenné des features à partir des listes d'un même algorithme qui lui n'est pas normalisé.

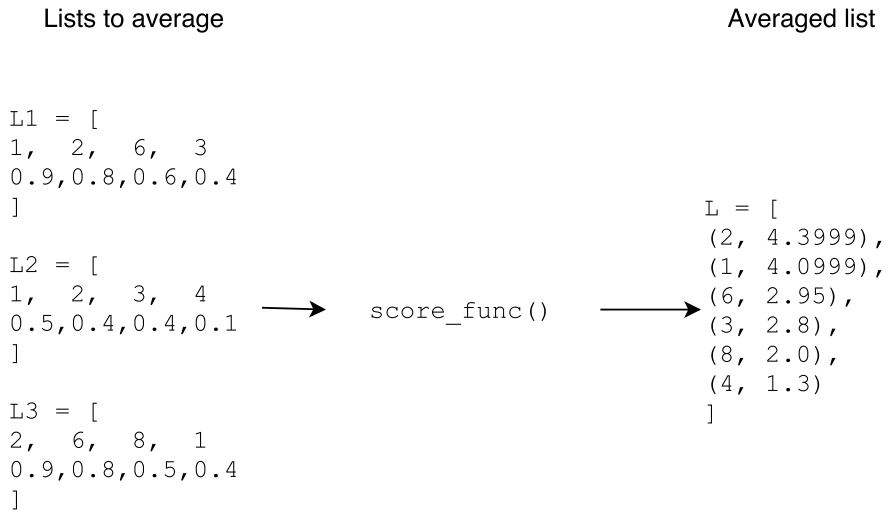

FIGURE 6.8 – Moyennage de listes classées (*rank*)

La méthode `score_func()` est utilisée pour calculer le score moyenné des listes afin d'en n'obtenir qu'une seule. La formule employée pour cette méthode est la suivante :

$$score(f) = 1.2 * occurrence(f) + median(f) \tag{6.2}$$

La méthode est scindée en deux parties. La première, composée de la fonction $occurrence(f)$ retourne simplement le nombre de fois que f, une feature donnée, apparaît dans les listes, de manière semblable à ce qui est expliqué à l'équation 6.1. Cette première partie permet de satisfaire le point n°1 des critères de sélection évoqués à la section 6.4.3.

La deuxième partie de la méthode permet de respecter le deuxième critère de sélection. En effet, on utilise le score médian de la feature dans les listes à moyenner. On utilise la médiane car elle est moins sensible aux valeurs extrêmes.

Enfin, on utilise un coefficient d'occurrence — le $*1.2$ dans l'équation — afin de respecter le point n°3 des critères de sélection. Ceci permet, en cas de médiane égale, de favoriser la feature qui apparaît le plus souvent. Par exemple, si on prend les listes `L5 = [(1, 0.4), (6, 0.3), (2, 0.3)]`, `L6 = [(2, 0.9), (6, 0.5), (4, 0.2)]` et `L7 = [(5, 0.9), (6, 0.4), (4, 0.2)]` et qu'on compare les features 1 et 6, on note que $median(f_1) = median(0.4) = 0.4$ et $median(f_6) = median(0.3, 0.5, 0.4) = 0.4$. La feature 1 et 6 aurait un score égal si on avait pas pondéré le nombre d'occurrences alors qu'on cherche à favoriser, en cas d'égalité, la feature la plus récurrente.

Après l'application de la méthode `score_func()`, on obtient une liste de tuple (feature, score) avec la feature la plus importante en première position.

Listes de features classées (*rank*)

Pour les listes de features classées, on dispose de listes ordonnées, c'est-à-dire avec la feature la plus importante apparaissant en premier dans la liste. Ces listes peuvent être vues comme des listes de tuples (feature, rang) comme illustré à la figure 6.9. Par exemple, pour la liste `L1`, les features 1, 2, 6 et 3 ont été sélectionnées et leur rang est respectivement 0, 1, 2, 3.

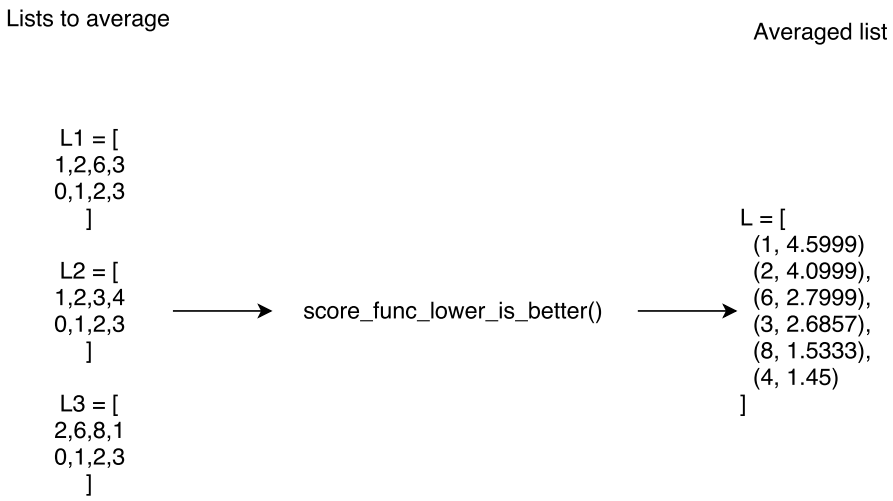

FIGURE 6.9 – Moyennage de listes classées (*rank*)

La méthode `score_func_lower_is_better()` est utilisée pour calculer le score moyenné des listes afin d'en n'obtenir qu'une seule. La formule utilisée pour cette méthode, qui est similaire à celle utilisée pour les listes à score, est la suivante :

$$score(f) = 1.2 * occurrence(f) + \frac{1}{1 + median(f)} \tag{6.3}$$

La différence principale concerne la deuxième partie de l'équation. En effet, on utilise $\frac{1}{1+median(f)}$ car c'est une fonction décroissante ce qui permet d'avoir les features avec les rangs les plus petits (donc les meilleurs) en premier. $1 + median(f)$ permet d'éviter une division par zéro dans le cas où la médiane renverrait 0.

Après l'application de la méthode `score_func_lower_is_better()`, on obtient une liste de tuple (feature, score) avec la feature la plus importante en première position.

Conclusion sur le moyennage des listes

Comme on peut le voir, cet algorithme respecte les critères cités à la section 6.4.3. En effet, les features qui reviennent le plus souvent sont mises en avant grâce au comptage du nombre d'occurrences (point n°1) ; le rang ou le score est utilisé pour départager deux features qui apparaissent le même nombre de fois (point n°2) ; et on favorise le nombre d'occurrences au rang/score grâce à une pondération supérieure du nombre d'occurrence par rapport au score "moyenné" (point n°3). On tient tout de même à signaler que les méthodes de moyennage proposées ne sont pas nécessairement les meilleures mais qu'elles respectent les critères de sélection et permettent d'avoir un indicateur de la stabilité des listes.

Le module `AlgorithmListsUtils.py` est commenté et peut apporter des précisions en terme d'implémentation.

6.4.4 Autres remarques concernant l'implémentation des algorithmes

Dans cette section, on aborde quelques points pour lesquels une section entière ne se justifie pas mais qui pour autant méritent d'être mentionnés.

Implémentation multi-coeur Certains algorithmes n'utilisent, par défaut, qu'un seul coeur processeur. Il est souvent possible dans la librairie scikit-learn d'utiliser un paramètre `n_jobs` qui permet de spécifier le nombre de coeurs à utiliser. Cependant, il faut veiller à ne pas saturer la mémoire RAM en utilisant ce paramètre. En effet, en Python le multi-thread est compliqué à utiliser [16], notamment à cause du GIL [17] qui séquentialise les threads pour garantir un déroulement sûr du programme. Il existe néanmoins un moyen pour éviter l'effet du GIL. Il ne faut pas utiliser des threads mais des processus, notamment avec le module `multiprocessing` [18]. Cependant, il y a un inconvénient. Les données utilisées par les processus ne sont pas partagées, ce qui implique qu'elles doivent être dupliquées pour chaque processus. Quand on utilise un grand dataset comme MILE, la mémoire vive vient à manquer rapidement. Il peut donc soit abandonner l'idée d'utiliser une approche multi-thread, soit d'utiliser un nombre réduit de threads de façon à ne pas trop impacter la mémoire RAM. Plusieurs approches d'optimisation non documentées ont été tentées pour l'algorithme GAANN.

Métrique de performance des algorithmes Pour les algorithmes qui le supportent, on dispose d'une méthode `get_score()` qui permet d'obtenir le score de l'algorithme après l'apprentissage. Ce score peut être utilisé comme indice de confiance et permet d'évincer un algorithme qui ne donnerait pas une performance convaincante. Le calcul de ce score est basé sur la précision qui est définie comme $précision = \frac{Élements\ correctement\ classifiés}{Élements\ au\ total}$. Cependant, il convient de noter que cette façon de faire possède un biais. Dans le cas où les classes ne sont pas équilibrées et qu'une classe serait largement représentée par rapport aux autres, il suffirait de classifier correctement cette classe pour obtenir une bonne précision. Par exemple, si dans une école il y a une classe avec 9 garçons et 1 fille, il suffit que l'algorithme renvoie toujours la réponse "garçon" pour obtenir 90% de précision. Pour éviter ce problème, on peut commencer par équilibrer les classes comme cela a été fait dans la partie pré-traitement du chapitre 5. Un autre moyen, plus visuel, a été utilisé, on affiche une matrice de confusion permettant de voir quelles classes ont été mal classifiées.

16. Lire les articles : `https://www.jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/` et `https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/`
17. `https://wiki.python.org/moin/GlobalInterpreterLock`
18. `https://docs.python.org/2/library/multiprocessing.html`

7. Sélection des features

Ce chapitre présente la partie sélection de features, c'est-à-dire l'analyse des listes données par les différents algorithmes.

Dans le *notebook* `features_selection.ipynb`, chaque algorithme donne une liste de 1000 features au maximum. Au maximum car certains algorithmes, comme CFS, fournissent une liste de features de taille variable. Le nombre de features à conserver est un choix qui permet d'en avoir suffisamment pour les mélanger dans la partie features merging et en avoir, idéalement, quelques centaines au final.

Pour comparer les listes entre elles, plusieurs techniques ont été utilisées. L'utilisation de matrices de similarité permet de visualiser, sur la forme d'un tableau, la similarité – à savoir le nombre de features partagées – entre deux listes. Un dendrogramme a également été utilisé pour représenter, sous la forme d'une hiérarchie, la distance entre les listes.

On précise que seule la présence ou l'absence de features a été prise en compte pour ces deux visualisations. On ne s'intéresse pas au rang, ni au score par features renvoyées par certaines listes. Cependant, on a tout de même tenté de savoir si les features sélectionnées par algorithmes donnaient, dans une certaine mesure, les mêmes features pour les premiers rangs. On a donc naïvement affiché les 15 premières features de chaque liste sur les 1000 fournies.

7.1 Visualisation naïve

Sur les figures ci-dessous, est affiché le top 15 des features par rang pour Golub, c'est-à-dire les 15 features avec **le plus haut rang**. La deuxième image *contient les mêmes features* mais triées de la plus petite à la plus grande de manière à pouvoir plus facilement comparer les listes entre elles.

7.1.1 Golub

```
MRMR           (1760, 1243, 6993, 2912, 1128, 3470, 243, 5197, 1139, 6114, 1213, 5285, 3305, 6483, 2101)
ExtraTrees     (4210, 2641, 5253, 3251, 2287, 1881, 6040, 4195, 759, 6224, 6802, 3319, 4488, 6184, 5500)
F Value        (2287, 4195, 4376, 4846, 6918, 3251, 2019, 1833, 5771, 6224, 1597, 3319, 6805, 2353, 4166)
SVM RFE        (18, 38, 39, 40, 41, 42, 43, 44, 45, 46, 50, 51, 53, 87, 94)
SVM            (1778, 1881, 6208, 5709, 2344, 2401, 5951, 1673, 2796, 6669, 5101, 6802, 6178, 4195, 6805)
Fisher Score   (2287, 4195, 4376, 4846, 6918, 3251, 2019, 1833, 5771, 6224, 1597, 3319, 6805, 2353, 4166)
ReliefF        (2401, 1778, 1881, 5709, 4195, 5710, 4016, 2287, 2185, 6805, 6802, 5228, 1673, 6167, 6208)
Limma          (1833, 6854, 3251, 759, 1881, 4846, 1684, 2120, 2353, 6280, 4372, 803, 6040, 4327, 2401)
```

FIGURE 7.1 – Top 15 des features par rang - Golub

```
MRMR           [243, 1128, 1139, 1213, 1243, 1760, 2101, 2912, 3305, 3470, 5197, 5285, 6114, 6483, 6993]
ExtraTrees     [759, 1881, 2287, 2641, 3251, 3319, 4195, 4210, 4488, 5253, 5500, 6040, 6184, 6224, 6802]
F Value        [1597, 1833, 2019, 2287, 2353, 3251, 3319, 4166, 4195, 4376, 4846, 5771, 6224, 6805, 6918]
SVM RFE        [18, 38, 39, 40, 41, 42, 43, 44, 45, 46, 50, 51, 53, 87, 94]
SVM            [1673, 1778, 1881, 2344, 2401, 2796, 4195, 5101, 5709, 5951, 6178, 6208, 6669, 6802, 6805]
Fisher Score   [1597, 1833, 2019, 2287, 2353, 3251, 3319, 4166, 4195, 4376, 4846, 5771, 6224, 6805, 6918]
ReliefF        [1673, 1778, 1881, 2185, 2287, 2401, 4016, 4195, 5228, 5709, 5710, 6167, 6208, 6802, 6805]
Limma          [759, 803, 1684, 1833, 1881, 2120, 2353, 2401, 3251, 4327, 4372, 4846, 6040, 6280, 6854]
```

FIGURE 7.2 – Top 15 des features par rang triées - Golub

En analysant ces différentes listes, on peut remarquer que les algorithmes F Value et Fisher Score, provenant de librairies différentes, renvoient le même top 15. Il s'agit très probablement du même algorithme qui est utilisé en arrière-plan et on verra, avec la matrice de similarité, qu'en réalité ce sont strictement les mêmes listes qui sont renvoyées.

SVM RFE possède un top 15 complètement différent des autres listes, y compris de SVM. Ceci s'explique car, parmi les 1000 features sélectionnées, toutes ou presque ont le même score. En effet, l'algorithme SVM RFE n'étant pas déterministe, il a été lancé plusieurs fois puis moyenné comme expliqué à la section 6.4.3. Les 15 premières features étant probablement chaque fois différentes – ce qui ne veut pas dire que cela soit le cas pour les 985 restantes – elles ont chacune obtenu le score de 4.6 (voir figure 7.3). Pour comprendre ce score et donc les features affichées ici, il est nécessaire d'expliquer comment fonctionne la fonction `get_score()` de SVM RFE. Une fois le RFE terminé – c'est-à-dire quand il a retiré suffisamment de features pour arriver aux 1000 features demandées – SVM RFE fournit un attribut `self.ranking_` qui contient une liste de nombres. La valeur de `self.ranking_[i]` contient le rang de la i-ème feature. Les rangs commencent à 1 et plusieurs features peuvent avoir le même rang. Attention, il ne faut pas confondre ce rang avec le rang de la liste des features retournée par `get_features_by_rank()` de la classe `Algorithm`. Le fait que plusieurs features possèdent le même rang est sûrement lié au nombre d'itérations avant que la feature ne soit éliminée. Ainsi, si les features 34 et 36 sont exclues à l'itération 4 et qu'il y a, en tout, 10 itérations, alors on pourrait leur attribuer le rang 6 ($6 = 10 - 4$) mais ceci n'est qu'une hypothèse. Par conséquent, si deux features ont le même rang, alors c'est malheureusement le numéro de la feature qui sera pris en compte lors du tri des features par score. De plus, les scores de 4.6 s'expliquent par le fait que la formule utilisée pour le moyennage de listes soit $score = 1.2 * occurrences(f) + median(1)$. Grâce à cette équation, on trouve que le nombre d'occurrence vaut 3, soit autant de fois que l'algorithme a été lancé par le "moyenneneur" et la médiane à 1 s'explique par le fait que SVM RFE attribue le score de 1 au meilleur rang, qui, encore une fois, peut être partagé par plusieurs features. C'est pour cette raison que la liste du top 15 de SVM RFE semble triée par numéro de feature car toutes ces features possèdent le même score. On peut prouver ceci avec la figure 7.3. Les précédentes explications montrent pourquoi le top 15 renvoyé par SVM RFE semble si différent des autres listes. On verra cependant qu'il y a tout de même des features en commun avec d'autres listes avec les matrices de similarités.

On remarque également que Limma est relativement proche de F Value/Fisher Score. On retrouve les features 1881, 2353, 3251 et 4846 pour les 15 meilleures features. On devrait donc retrouver cette proximité dans la liste de similarité et dans le classement hiérarchique avec le dendrogramme.

```
1 SVM RFE [(18, 4.6), (38, 4.6), (39, 4.6), (40, 4.6), (41, 4.6), (42, 4.6),
  ↪  (43, 4.6), (44, 4.6), (45, 4.6), (46, 4.6), (50, 4.6), (51, 4.6), (53,
  ↪  4.6), (87, 4.6), (94, 4.6)]
```

FIGURE 7.3 – Scores du top 15 des features données par SVM RFE pour Golub

7.1.2 MILE

```
MRMR          (7170, 5557, 16564, 8345, 470, 40611, 19215, 12483, 8336, 801, 973, 20796, 54014, 52848, 11792)
ExtraTrees    (7339, 6869, 5100, 51442, 7717, 1830, 26963, 18712, 25615, 7153, 33081, 46113, 6923, 31882, 11017)
F Value       (25971, 39104, 19498, 3218, 2133, 29388, 2047, 6302, 4341, 10034, 16885, 28925, 1434, 3150, 26549)
SVM RFE       (150, 219, 559, 602, 691, 729, 824, 869, 872, 873, 916, 941, 946, 960, 961)
SVM           (14153, 2977, 20728, 2978, 9978, 20937, 44097, 5526, 30315, 52979, 32361, 23409, 47859, 19448, 21309)
Fisher Score  (25971, 39104, 19498, 3218, 2133, 29388, 2047, 6302, 4341, 10034, 16885, 28925, 1434, 3150, 26549)
ReliefF       (24415, 9232, 13613, 24417, 4543, 9978, 3952, 7339, 27493, 21104, 8989, 13227, 12169, 9497, 52979)
Limma         (25971, 39104, 19498, 2133, 4341, 960, 29388, 6302, 10034, 961, 16885, 3218, 2047, 25480, 24237)
```

FIGURE 7.4 – Top 15 des features par rang - MILE

```
MRMR          [470, 801, 973, 5557, 7170, 8336, 8345, 11792, 12483, 16564, 19215, 20796, 40611, 52848, 54014]
ExtraTrees    [1830, 5100, 6869, 6923, 7153, 7339, 7717, 11017, 18712, 25615, 26963, 31882, 33081, 46113, 51442]
F Value       [1434, 2047, 2133, 3150, 3218, 4341, 6302, 10034, 16885, 19498, 25971, 26549, 28925, 29388, 39104]
SVM RFE       [150, 219, 559, 602, 691, 729, 824, 869, 872, 873, 916, 941, 946, 960, 961]
SVM           [2977, 2978, 5526, 9978, 14153, 19448, 20728, 20937, 21309, 23409, 30315, 32361, 44097, 47859, 52979]
Fisher Score  [1434, 2047, 2133, 3150, 3218, 4341, 6302, 10034, 16885, 19498, 25971, 26549, 28925, 29388, 39104]
ReliefF       [3952, 4543, 7339, 8989, 9232, 9497, 9978, 12169, 13227, 13613, 21104, 24415, 24417, 27493, 52979]
Limma         [960, 961, 2047, 2133, 3218, 4341, 6302, 10034, 16885, 19498, 24237, 25480, 25971, 29388, 39104]
```

FIGURE 7.5 – Top 15 des features par rang triées - MILE

Pour MILE, on peut observer les mêmes phénomènes. Par exemple, on retrouve cette proximité des listes entre F Value/Fisher Score et Limma avec les features 2133, 3218, 4341, 6302, 10034, 16885, 19498, 29388 et 39104 en commun pour les 15 meilleurs rangs.

7.2 Visualisation du score moyenné des listes

Le but des visualisations de cette section est de montrer le diminution des contributions des features sélectionnées. Une ligne horizontale signifierait que toutes les features sélectionnées sont équivalentes en terme d'importance, ce qui se produit avec des listes plates non-moyennées par exemple.

Dans un souci de lisibilité, seuls quelques graphes sont montrés ici. Les graphes suivants concerne uniquement le dataset Golub. L'intégralité des graphes est cependant disponible en annexe.

7.2.1 Listes plates

Dans le cas des algorithmes déterministes, toutes les features sélectionnées par un algorithme ont un score à 1. En effet, par définition, pour les listes plates, une feature n'est pas plus importante qu'une autre. Dans le cas des algorithmes non-déterministes, on a effectué un moyennage pour ne conserver que les 1000 features les plus souvent renvoyées par l'algorithme. On note `#runs`, le nombre de fois que l'algorithme a été lancé pour être moyenné. Pour ce projet, `#runs` a été fixé à 3. Comme on l'a signalé en 6.4.3, le score d'une feature moyennée d'une liste plate est simplement le nombre de fois que cette feature a été sélectionnée parmi les `#runs`. On a donc un score qui peut varier entre 1 – la feature n'a été sélectionnée qu'une seule fois par l'algorithme – et 3 – la feature a été sélectionnée à chaque run car `#runs = 3`.

Les figures ci-dessous montrent trois exemples différents :

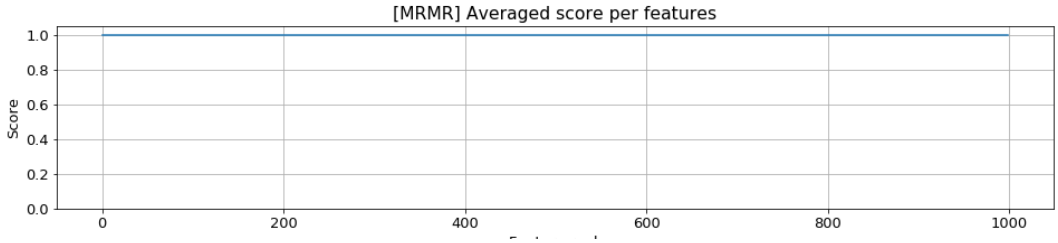

FIGURE 7.6 – MRMR en liste simple moyennée - Golub

On remarque bien que toutes les features ont un score de 1 pour cette liste simple provenant d'un algorithme déterministe.

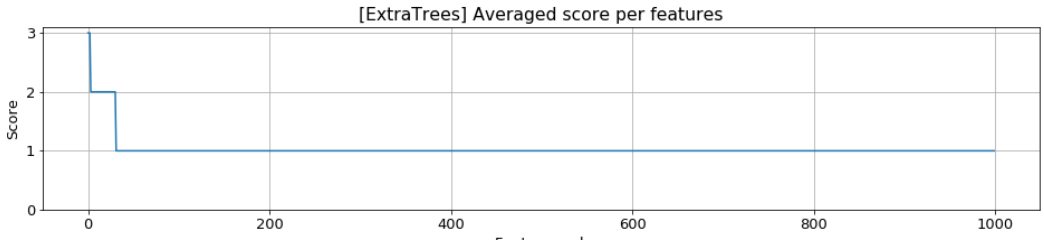

FIGURE 7.7 – ExtraTrees en liste simple moyennée - Golub

Pour ExtraTrees, un algorithme non-déterministe, on observe que seules certaines features ont été conservées pour tous les runs et que les autres n'ont été retenues par l'algorithme que 2 ou 1 seule fois. Bien que ces 1000 features aient été sélectionnées, on peut se demander s'il est vraiment nécessaire d'en garder autant, surtout pour celles n'ayant été retenues qu'une seule fois.

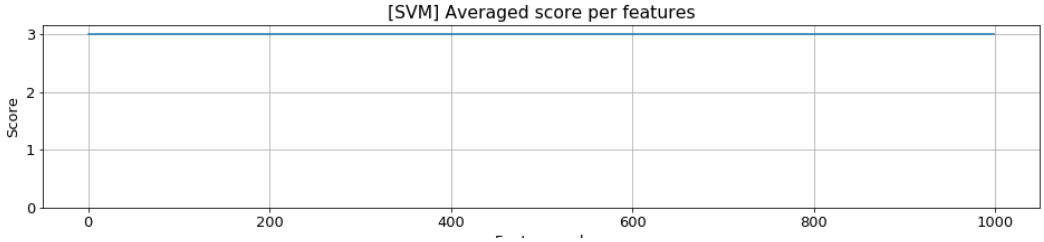

FIGURE 7.8 – SVM en liste simple moyennée - Golub

Enfin, pour les listes simples, on constate que SVM a conservé les mêmes features pour tous les runs. Bien qu'il s'agisse de la même liste de features, on peut se poser la question si le score attribué par SVM est toujours le même.

7.2.2 Listes par rang

Les algorithmes déterministes affichent leur rang avec la fonction $rang_{new} = 1/(1 + rang)$ pour avoir une fonction décroissante. Avec du recul, on se rend compte que l'utilisation d'une fonction décroissante reste pertinente pour avoir les features avec le plus petit rang apparaissant en premier. Toutefois, la formule elle-même peut être remise en question car le domaine de définition n'est pas le même que cette fonction $1.2 * \#occurrences + 1/(1 + median(f))$ utilisée pour moyenner les rangs des algorithmes non-déterministes. Si cela était à refaire, on proposerait plutôt une fonction proche de celle-ci : $1.2 * \#runs + 1/(1 + rang)$.

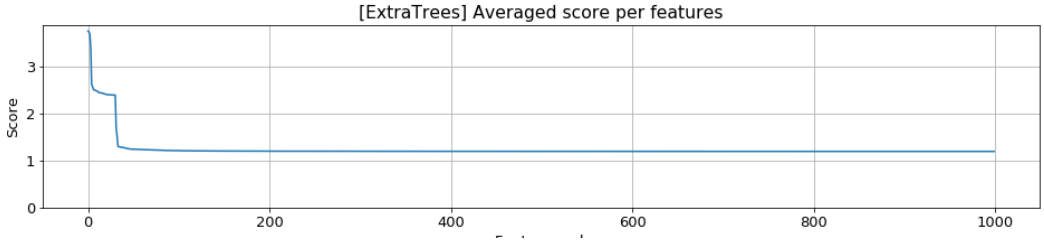

FIGURE 7.9 – ExtraTrees en liste par rang moyennée - Golub

La figure ci-dessus montre l'importance des features par rang pour ExtraTrees. On note qu'en ajoutant le rang dans le calcul du score pondéré, on peut distinguer plusieurs features qui auraient été sélectionnées le même nombre de fois par l'algorithme. Cette figure permet de visualiser l'effet de la formule du moyennage décrite à l'équation 6.3.

7.2.3 Listes par score

Remarque similaire à la sous-section précédente, il faudrait modifier la fonction de score pour les algorithmes déterministes. Actuellement, aucune modification n'est effectuée sur le score hormis une normalisation min-max (score borné entre 0 à 1). Il semble pertinent de multiplier le score obtenu de chaque feature par $\#runs$ pour avoir la même fourchette de valeurs que pour les algorithmes non-déterministes (voir équation 6.2).

Sur les trois figures ci-dessous, on constate que Fisher Score, ReflefF et Limma – qui sont des algorithmes déterministes – renvoient des features dont l'importance décroit progressivement.

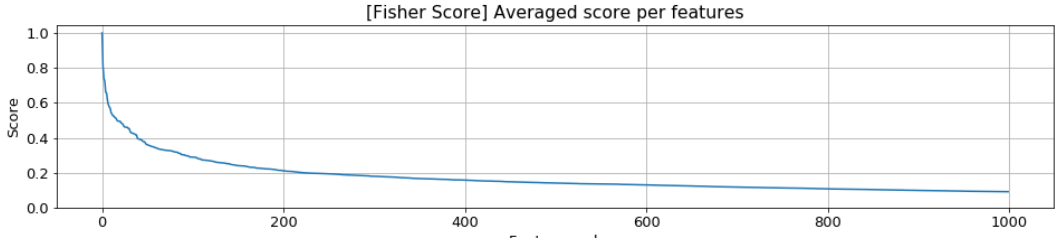

FIGURE 7.10 – Fisher Score en liste par score - Golub

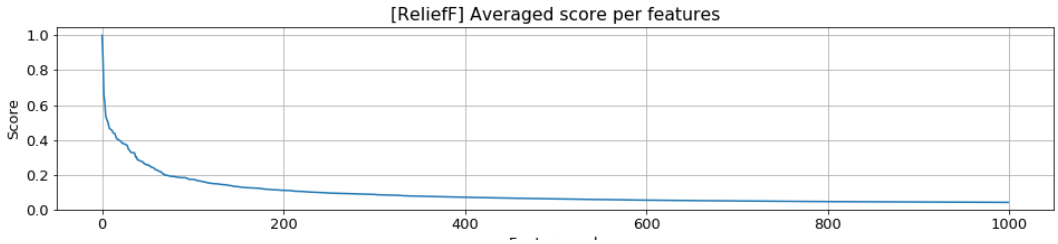

FIGURE 7.11 – ReliefF en liste par score - Golub

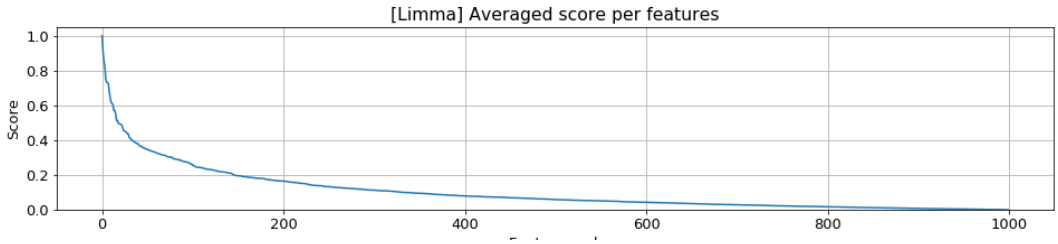

FIGURE 7.12 – Limma en liste par score - Golub

Dans la partie "Visualisation naïve", on énonçait que SVM RFE proposait des listes différentes des autres méthodes car les 15 meilleures features avaient toutes le même score de 4.6. On remarque, grâce à la figure ci-dessous, que ce phénomène s'étend à l'ensemble des features sélectionnées. Cela ne veut pas nécessairement dire que les features sont de mauvaise qualité mais qu'il faut probablement ne pas utiliser ces scores pour cet algorithme.

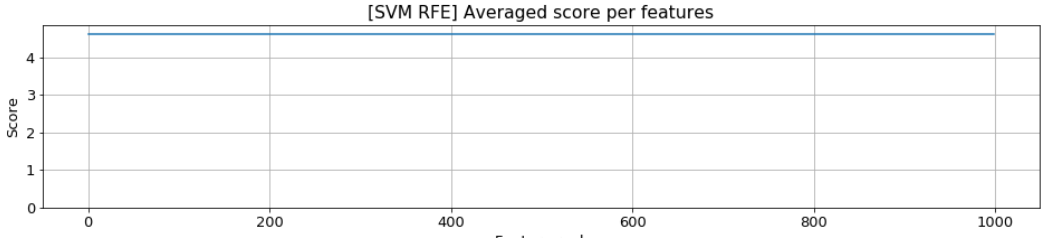

FIGURE 7.13 – SVM RFE en liste par score - Golub

En conlusion, pour ces graphes de listes moyennées, on observe que demander autant de features n'est pas forcément nécessaire. En effet, l'importance des features s'essouffle bien avant d'avoir atteint les 1000 features demandées. Une amélioration possible serait de mettre un seuil et de ne conserver que les features les plus importantes de ces listes. Néanmoins, ce seuil serait probablement fixé manuellement pour chaque algorithme et pour chaque dataset. De plus, on note que l'utilisation d'un moyenneur est nécessaire pour les algorithmes non-déterministes. Celui proposé n'est probablement pas le meilleur, mais il a le mérite de fournir une première implémentation.

7.3 Matrices de similarité

Comme il a été annoncé, des matrices de similarité ont été utilisées pour visualiser les features communes entre les listes. Pour précision, les matrices de similarité utilisent les listes plates et on entend par matrice de similarité, une matrice qui confronte les listes, une contre une autre et renvoie une mesure de similarité. Dans ce projet, deux mesures de similarité ont été utilisées, la première emploie l'intersection entre deux listes et renvoie donc le nombre de features présentes dans ces listes et la seconde, emploie la similarité de Jaccard [1] qui renvoie un nombre entre 0 (les listes sont complètement différentes) et 1 (les listes sont identiques). Bien que ces deux méthodes utilisent l'intersection, il a été jugé pertinent de conserver les deux. En effet, l'utilisation de la matrice de similarité utilisant l'intersection seule permet de voir, concrètement, le nombre de features partagées entre deux listes tandis que l'utilisation de l'indice de Jaccard permet de rapidement comparer plusieurs éléments de la matrice entre eux.

1. Indice de Jaccard : `https://fr.wikipedia.org/wiki/Indice_et_distance_de_Jaccard`

7.3.1 Golub

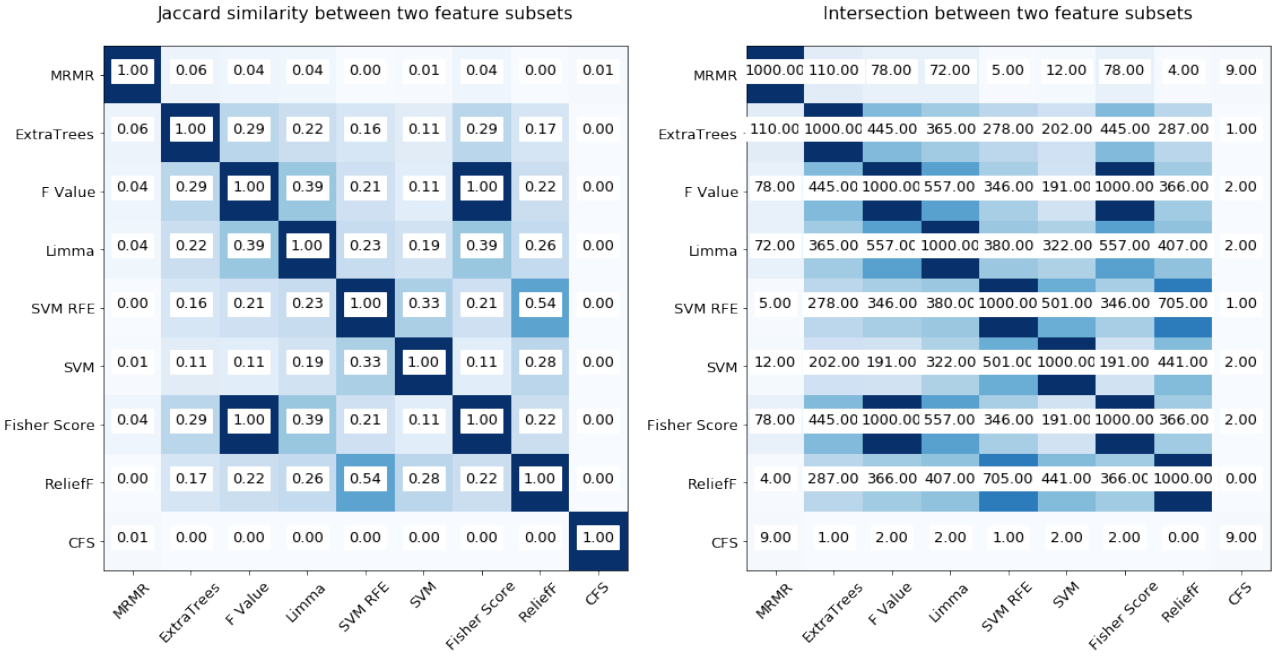

FIGURE 7.14 – Matrices de similarité - Golub

On peut remarquer que les matrices sont symétriques et que leur diagonale sont au maximum de la mesure (le nombre de features rapportées par l'algorithme ou 1) car les listes sont comparées à elle-même.

On constate également que F Value et Fisher Score renvoient bien les mêmes listes comme évoqué plus haut.

De plus, la proximité entre Limma et F Value/Fisher Score décrite aux paragraphes précédents se confirme ici. Sur la matrice de gauche, on voit un indice de Jaccard à 0.39. Tout aussi intéressant, on note que la liste de SVM RFE est relativement proche de celle proposée par ReliefF avec un indice à 0.54 bien que le top 15 des meilleures features laissait penser le contraire.

L'utilité d'avoir les deux matrices se justifie en comparant CFS et MRMR. En effet, quand on regarde la matrice de gauche – celle utilisant Jaccard – on remarque une très faible similarité entre CFS et les autres listes avec des indices proches de 0. En revanche, en regardant la matrice de droite – celle utilisant uniquement l'intersection – on constate en lisant la diagonale que CFS ne renvoie que 9 features. Or, ces 9 features sont également présentes chez l'algorithme MRMR. Les features de CFS sont donc toutes présentes chez MRMR. Grâce à ces matrices, on peut donc évincer l'algorithme CFS pour la suite du projet car il renvoie un nombre trop peu important de features, d'autant plus que ces features sont toutes contenues dans MRMR.

7.3.2 MILE

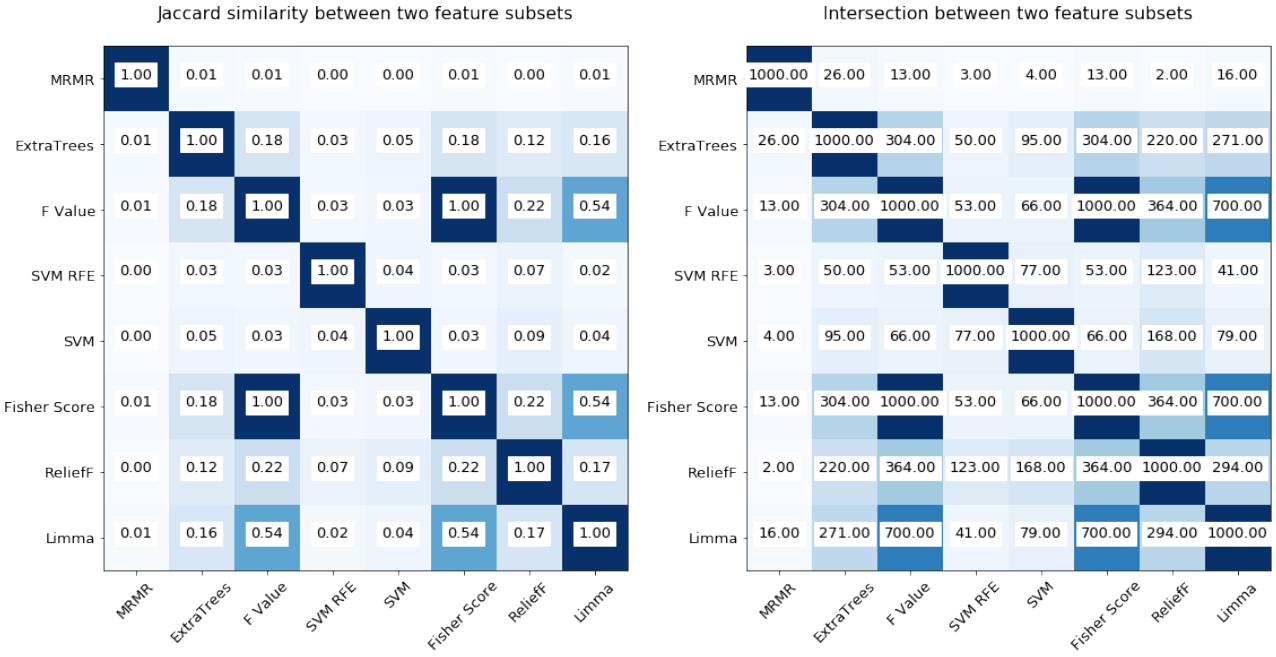

FIGURE 7.15 – Matrices de similarité - MILE

Concernant le dataset MILE, on remarque que CFS a été retiré pour les raisons décrites plus haut, d'autant plus que cet algorithme est coûteux en temps de calcul.

F Score/Fisher Score est proche de Limma avec un indice de Jaccard de 0.54 et, dans une moindre mesure, proche de toutes les autres listes. ReliefF et Limma partagent 294 features, soit presque un tiers de l'ensemble des features. MRMR propose par contre une liste très différente des autres avec moins de 30 features en commun au maximum. Avoir des listes différentes n'est pas pour autant critique car on peut profiter de cette différence. En effet, on pourrait obtenir une liste composée de features plus robustes qui auraient échappées à certains algorithmes.

7.4 Dendrogramme

Une autre façon de représenter la distance entre plusieurs listes est de les regrouper à l'aide d'un dendrogramme [2]. Ce dernier requiert une *linkage matrix* appelée Z qui indique une notion de distance entre les listes et comment ces dernières sont embranchées pour former un arbre. Pour calculer cette matrice, on a utilisé la fonction `linkage` de Scipy [3]. Cette fonction prend en paramètre une mesure de distance et une liste de listes de booléens où chaque liste représente un algorithme (F Value, SVM,...) et où chaque booléen de cette dernière indique si la feature a été sélectionnée ou non par l'algorithme en question. Les listes possédées pour l'instant ne répondant pas à ce critère, il a été nécessaire de créer un masque de booléens pour chaque algorithme. Pour créer ce masque, on utilise ce morceau de code de la classe `Dendrogram` :

2. Dendrogramme : `https://fr.wikipedia.org/wiki/Dendrogramme`
3. Scipy : `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.cluster.hierarchy.linkage.html`

```python
def _compute_masks(self):
    # Use the union of all the selected features in all the lists to create the mask.
    # Using the union allows us to reduce the size of the mask by ignoring the features that are not in any lists
    # The mask is simply a list of booleans. True if the feature is in the list, False otherwise
    # Warning: we lose the sorting with the union
    lists_union = list(reduce(lambda a, b: set(a).union(set(b)), self._lists))

    l_mask = [self._get_mask_of_features(f, lists_union) for f in self._lists]
    l_mask = np.array(l_mask)

    return l_mask

@staticmethod
def _get_mask_of_features(features, union):
    return [u_i in features for u_i in union]
```

FIGURE 7.16 – Code permettant de créer les listes de booléens exigés par la *linkage matrix*

Une fois les listes de booléens à disposition, il reste à fournir une mesure de distance entre ces dernières. La librairie Scipy en propose plusieurs – Jaccard, Hamming, Rogers-Tanimoto, Dice,.. [4] – et après plusieurs tests la mesure Dice a été conservée. On a choisi cette mesure car on souhaite, entre autre, avoir une distance comprise entre 0 et 1 – cela sera un pré-requis pour la technique de fusion pondérée des listes en 9.5 – et on désire que des listes strictement identiques obtiennent une distance minimale et que des listes complètement différentes obtiennent une distance maximale. Les différents essais de dendrogrammes se trouvent dans le notebook `features_merging.ipynb`.

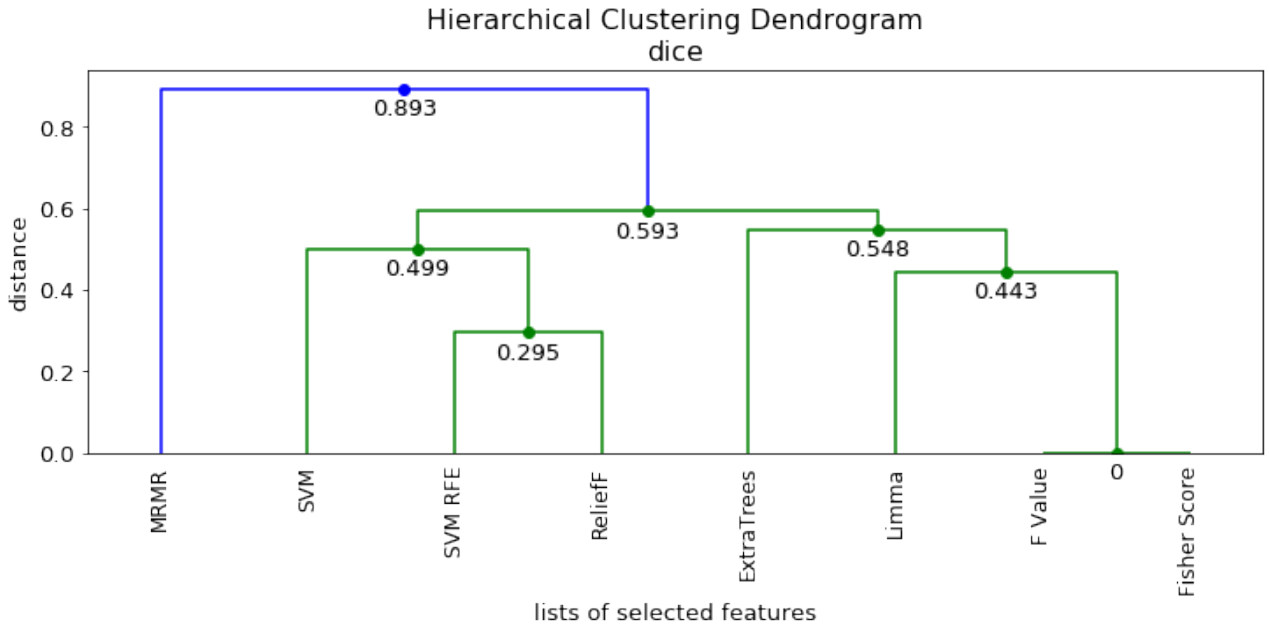

FIGURE 7.17 – Dendrogramme Golub

4. Mesures de distance proposées par Scipy : `https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.html`

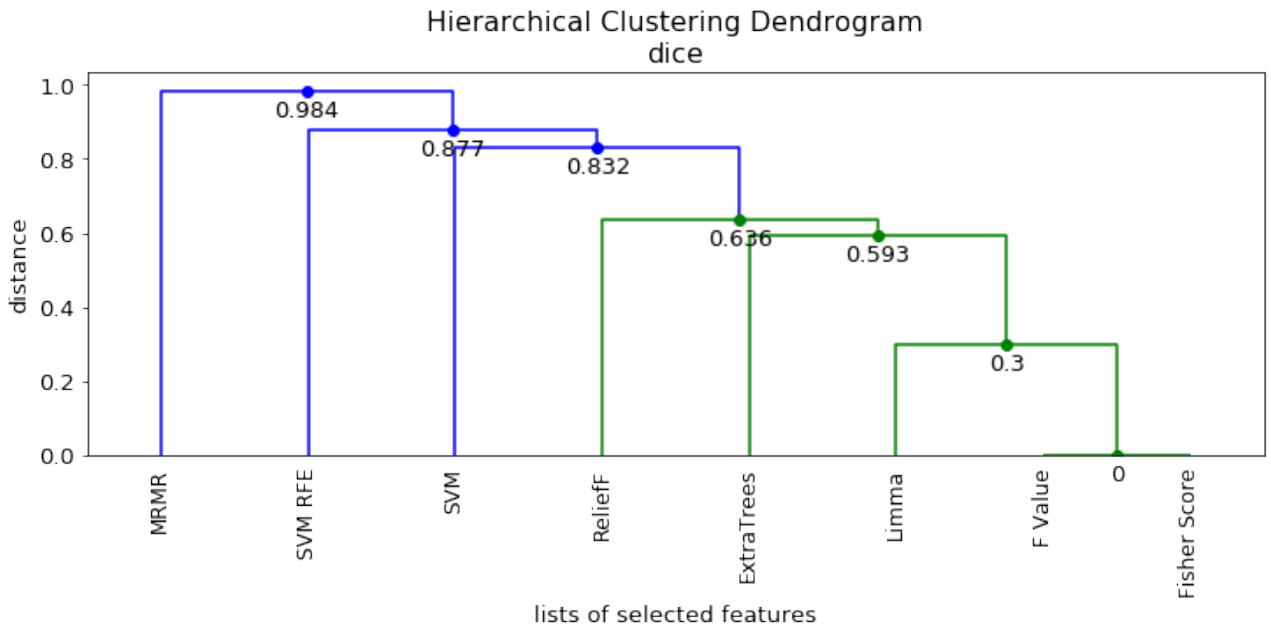

FIGURE 7.18 – Dendrogramme MILE

Les mêmes remarques s'appliquent à ces dendrogrammes : F Value/Fisher Score identiques et donc avec une distance à 0, Limma est proche de ces derniers autant avec Golub qu'avec MILE et MRMR propose la liste la moins proche. Ces dendrogrammes font parties de la visualisation des listes proposées par les algorithmes, cependant leur usage est plutôt destiné à la technique de merging des listes pondérées présentée en 9.5.

7.5 Analyse des résultats pour GAANN

Comme on peut le remarquer dans les listes de la section précédente, on ne retrouve pas une liste de features proposée par GAANN car il a été jugé que cet algorithme – bien qu'il semble fonctionner correctement, ce qu'on montrera plus bas – ne permet pas de fournir une liste de qualité.

On rappelle que cet algorithme combine un réseau de neurones et un algorithme génétique pour générer une population composée de listes de features. Quand on analyse la figure 7.19, on remarque bien une convergence et donc que la population se compose d'individus de plus de plus adaptés au problème qu'est Golub. Ceci conforte l'idée que l'algorithme fonctionne correctement.

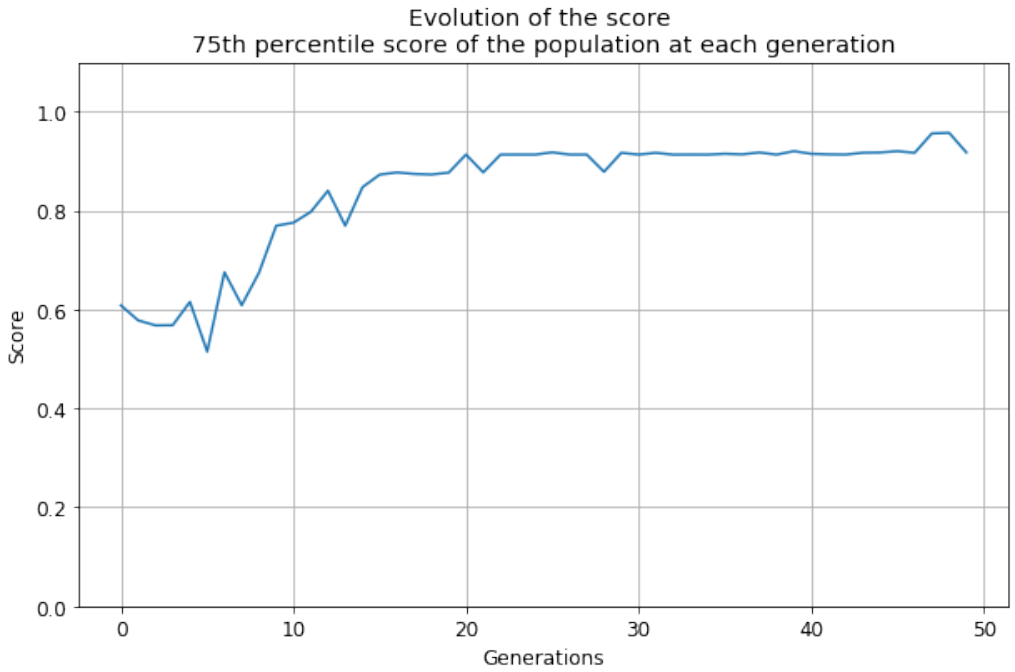

FIGURE 7.19 – Evolution du score de la population produite par l'algorithme GAANN pour Golub

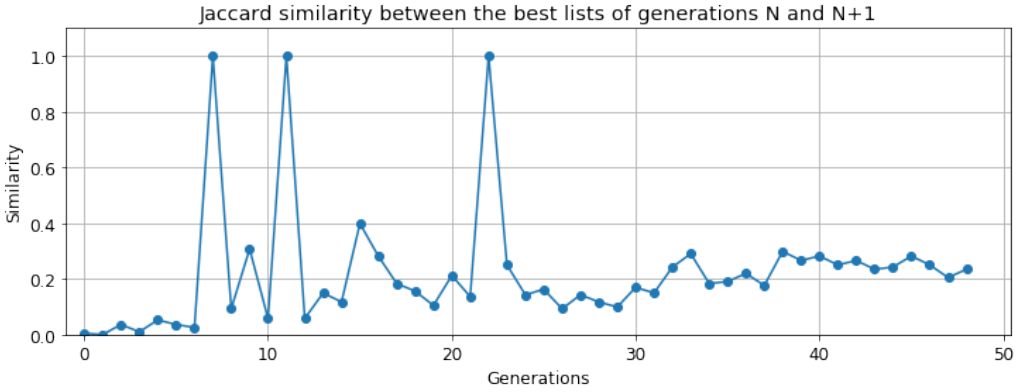

FIGURE 7.20 – Evolution de la similarité entre le meilleur individu de la génération n et n+1 pour Golub

Sur l'image ci-dessus, on a représenté la similarité entre le meilleur individu de la génération n et n+1. Ceci a pour objectif de montrer une certaine convergence et que le meilleur individu devient de plus en plus proche de son meilleur ancêtre au fil des générations. On remarque également certains pics à 1 indiquant que le meilleur individu et son meilleur ancêtre possèdent strictement la même liste. Dans la pratique, on se rend compte que cette évolution de la similarité est faible.

Bien que l'algorithme semble apporter une plus-value avec Golub – dans le sens où les générations améliorent le score de la population – ce n'est pas le cas avec le dataset MILE, voir figure 7.21, où le score de la population stagne et est déjà relativement bon (environ 0.9) et par conséquent GAANN n'apporte rien de plus que de tirer un individu au hasard (génération 0).

En cas de reprise du projet, il serait peut-être intéressant de montrer, pour chaque génération, l'écart type du score de la population. On aurait pu ainsi voir si cette valeur aurait tendance à diminuer au fil des générations. On montrerait ainsi que, bien que les scores soient égaux, ils seraient plus stables dans les dernières générations. Ceci reste une hypothèse. Par contre, cet ajout risquerait d'augmenter considérablement le temps de calcul, déjà élevé, de cet algorithme.

FIGURE 7.21 – Evolution du score de la population produite par l'algorithme GAANN pour MILE

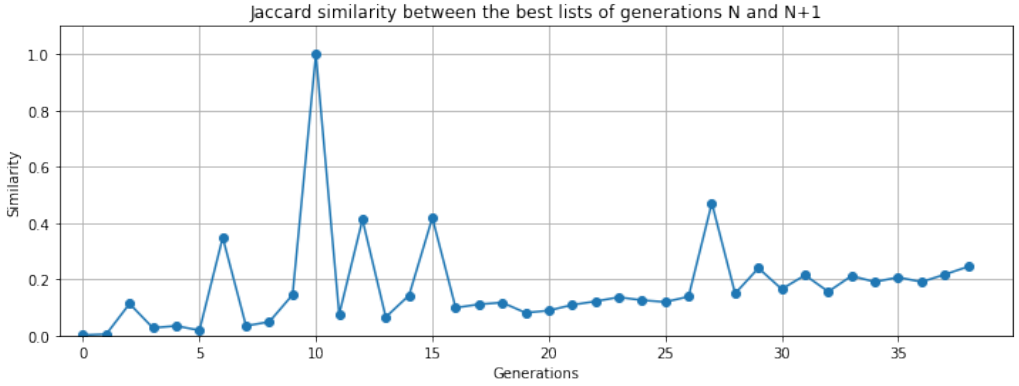

FIGURE 7.22 – Evolution de la similarité entre le meilleur individu de la génération n et n+1 pour MILE

Ici aussi, on note une légère convergence, non significative, entre le meilleur individu de la génération courante et son meilleur ancêtre.

On pourrait expliquer ce problème par le nombre de features redondantes ou corrélées que l'algorithme n'arriverait pas à identifier. Par exemple, par la présence d'un paysage de features plat où la majorité de ces dernières décrivent une petite partie de la classe rendant complexe l'extraction de features. Même si cela est peu probable, on pourrait remettre en question une implémentation douteuse du réseau de neurones de scikit-learn. Cette dernière hypothèse n'est, en réalité, non valable car on a tenté de remplacer cette librairie par une autre (Keras[60]) et les résultats étaient similaires.

Inconvénients de GAANN

De plus, l'algorithme souffre des inconvénients suivants :

Lent à l'exécution A chaque génération – entre 40 et 50 – l'ensemble des individus doit être évalué. Plusieurs tentatives d'optimisation ont été tentées. L'implémentation du MLP par scikit-learn étant monocore, on a tenté de paralléliser l'évaluation de la population en utilisant des processus grâce au module `multiprocessing` de Python, mais également en utilisant IParallel avec le module `IPython.parallel` ou encore en utilisant Keras. Le tableau ci-dessous compare les performances de ces techniques (sauf Keras). On a évalué (*benchmark* en anglais) la performance des méthodes internes de l'algorithme en mesurant le temps d'exécution de ces dernières et la consommation mémoire.

La première colonne est la version non-optimisée (séquentielle) de GAANN et la deuxième est l'implémentation utilisant le module `multiprocessing`. La dernière emploie IParallel qui utilise une architecture distribuée pour augmenter le nombre d'instances Python exécutées en même temps. Dans le cas de ce projet, ces instances ont été distribuées localement et non sur un cluster distant. On voit clairement que la méthode `_grade_pop()` prend la majorité du temps, c'est donc elle qu'on a cherché à optimiser. La table montre uniquement les temps d'exécution des deux premières générations pour des raisons de lisibilités et parce que le temps d'exécution d'une génération à une autre est globalement le même.

Python map (sequential) – ram : 5-6 GB	Multiprocessing – ram: 9-12 GB	Iparallel – ram: 12-16 GB
'_create_population' 0.07 sec	'_create_population' 0.05 sec	'_create_population' 0.05 sec
'_crossover' 0.10 sec	'_crossover' 0.11 sec	'_crossover' 0.09 sec
'_mutate' 0.00 sec	'_mutate' 0.00 sec	'_mutate' 0.00 sec
'_evolve' 0.10 sec	'_evolve' 0.11 sec	'_evolve' 0.09 sec
'_grade_pop' 112.70 sec	'_grade_pop' 54.70 sec	'_grade_pop' 51.51 sec
'_crossover' 0.12 sec	'_crossover' 0.13 sec	'_crossover' 0.09 sec
'_mutate' 0.00 sec	'_mutate' 0.00 sec	'_mutate' 0.00 sec
'_evolve' 0.12 sec	'_evolve' 0.13 sec	'_evolve' 0.09 sec
'_grade_pop' 86.51 sec	'_grade_pop' 56.95 sec	'_grade_pop' 57.31 sec
'_crossover' 0.11 sec	'_crossover' 0.12 sec	'_crossover' 0.10 sec
'_mutate' 0.00 sec	'_mutate' 0.00 sec	'_mutate' 0.00 sec
'_evolve' 0.11 sec	'_evolve' 0.12 sec	'_evolve' 0.10 sec

FIGURE 7.23 – Tableau comparatif des performances obtenues lors de la tentative d'optimisation de GAANN avec Golub

A la lecture de ce tableau, on constate que la version non-optimisée (séquentielle) est la plus lente avec un temps variant entre 86 sec et presque 2 min par génération. La consommation mémoire est d'environ 5 à 6 GB. Dans les versions parallélisées, on note une diminution du temps de calcul pour l'évaluation de la population d'un facteur 1.5 à 3 au détriment d'une consommation mémoire au moins doublée. Comme on l'expliquait à la section 6.4.4, ceci est dû au GIL qui ne permet pas de partager les ressources obligeant à recharger le dataset en mémoire. L'utilisation de Keras avait été motivée pour contourner ce problème étant donné que les algorithmes de ce dernier ne sont pas implémentés en Python.

Liste plate uniquement Dans son implémentation actuelle, cet algorithme fournit une liste de features plate uniquement. La liste n'est pas classée et ces dernières n'ont pas de score individuel. On pourrait cependant calculer un score en comptant le nombre de générations pour lesquelles la feature a été conservée. On pourrait également imaginer complexifier ce comptage en pondérant les générations. Par exemple, une feature conservée pendant les cinq premières générations aurait moins de valeur

qu'une feature introduite dans la 20^{e}génération et conservée jusqu'à la 25^{e}car les premières générations pourraient être plus sensibles au bruit. En effet, on imagine qu'au début de l'algorithme, un grand nombre de features soient remplacées mais que ce nombre diminue au fil des générations.

Individu idéal Il peut arriver que le meilleur individu d'une génération obtienne un score de 1 car il a réussi à classifier correctement chaque cas qu'on lui a soumis. On peut se demander dès lors pourquoi continuer à itérer et ne pas garder que cet individu ?

Nombreux paramètres L'utilisation combinée d'un réseau de neurones et d'un algorithme génétique multiplie le nombre d'hyper paramètres à gérer. On peut citer le nombre de couches du réseau de neurones, le *learning rate*, l'algorithme de descente de gradient,... Pour l'algorithme génétique, on peut citer les différentes techniques de sélection (roulette, par tournoi,...), de *crossover* et de mutation. Il faudrait dans l'idéal pouvoir tester plusieurs combinaisons de ces paramètres pour connaître celui ou ceux qui offre la meilleure performance.

Algorithme enveloppé non-déterministe Le réseau de neurones utilisé n'étant pas déterministe, on peut se poser la question si son utilisation conjointe à un algorithme génétique pourrait affecter la performance de ce dernier. En effet, si la performance d'un individu varie significativement entre plusieurs exécutions, alors on peut se demander si le score calculé est digne de confiance. On rappelle que pour mitiger ce problème, les individus sont évalués plusieurs fois et la médiane de leurs scores est prise en compte. Dans cette optique, on pourrait tenter d'utiliser un autre algorithme à la place d'un réseau de neurones comme SVM utilisé par [65]. Pour rappel, l'idée de base était d'utiliser un réseau de neurones. Puis, on a joint un algorithme génétique pour en extraire les features les plus importantes. Si on venait à changer le *classifier* employé par l'algorithme génétique par le futur, la classe `GA` imbriquée dans `GAANNAlgorithm` pourrait être réutilisée. Il suffirait de modifier la fonction de *fitness*.

Pour conclure, cet algorithme n'a pas apporté de plus-value espérée et ces nombreux inconvénients n'ont fait que conforter le choix de ne pas continuer à l'utiliser pour la suite du projet. On tient de tout même à signaler que le développement de cet algorithme a permis d'approfondir les connaissances vis-à-vis des algorithmes génétiques et a permis de se poser des questions utiles pour la continuité du travail.

Les fichiers `GAANN_01022017_Golub.html` et `GAANN_01022017_MILE_2.html`, en annexe sont les résultats des *notebooks* Jupyter pour respectivement Golub et MILE de cet algorithme.

7.6 Bilan à mi-chemin de la pipeline

Pour résumé cette partie, 10 algorithmes ont été utilisés. Grâce à la visualisation des listes renvoyées, CFS et GAANN ont été écartés car ils n'ont pas apporté de bonnes listes de features. Dans le cas de CFS, la liste renvoyée était de taille insuffisante et était redondante avec MRMR. Concernant GAANN, on a vu que les listes renvoyées obtenaient un score proche de celui d'une liste aléatoire, du moins pour MILE.

Chaque algorithme non-déterministe a été lancé plusieurs fois et les listes renvoyées ont été moyennées pour capturer certaines features qui sont choisies plusieurs fois par le même algorithme.

On possède donc 9 (GAANN ignoré) listes de features pour Golub et 8 (GAANN et CFS ignorés) pour MILE.

8. Transmission des listes de features

Une fois la partie *feature selection* effectuée, on peut passer au deuxième grand objectif de ce projet, la combinaison des listes. On se situe ici à mi-chemin de la pipeline. Le projet étant séparé en deux *notebooks* — `features_selection.ipynb` et `features_merging.ipynb` — il convient donc de transmettre les listes obtenues d'un *notebook* à l'autre.

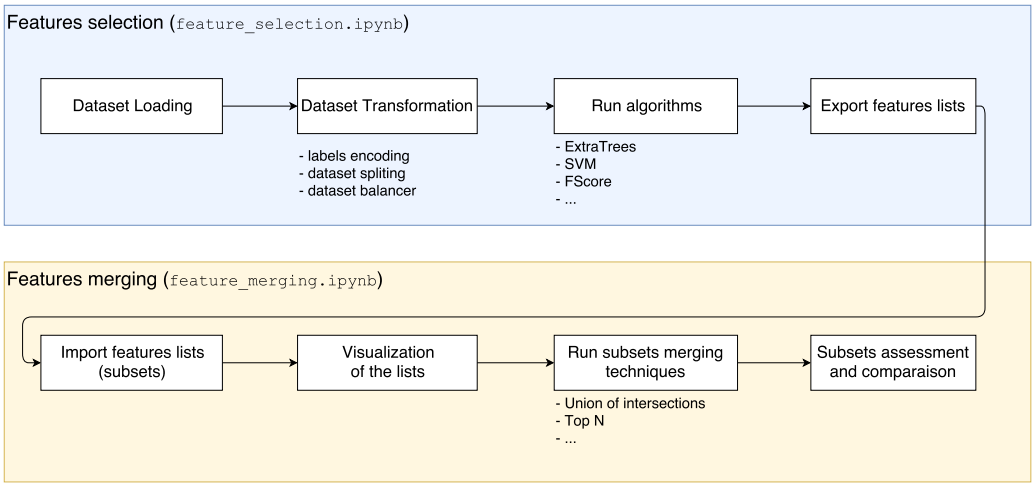

FIGURE 8.1 – *Rappel* : Pipeline du projet

Pour ce faire, il a été choisi de transmettre les listes de features à travers des fichiers CSV. Ce format a été préféré à d'autres (notamment à Pickle [1]) car les fichiers CSV sont des fichiers interopérables. Ceci permet de reprendre ces listes pour un autre projet ou un autre programme.

8.1 Exemple de listes

Comme on peut le voir si dessous, les listes exportées suivent toutes la même structure à savoir : `nomAlgorithme;feature1,score1;feature2,score2;[...]`. On note que les algorithmes qui ne fournissent pas de listes classées ou à score sont tout de même présentes mais avec une liste vide. Par exemple, l'algorithme CFS ne renvoie qu'une liste simple et s'affiche donc `CFS;` dans les listes classées et à scores. De plus, les listes sont triées par ordre décroissant de score, moyenné pour les algorithmes non-déterministes, en utilisant la technique expliquée à la section 6.4.3.

Listes simples :

```
1 MRMR;1760,1.00000;1243,1.00000;6993,1.00000;[...]
2 ExtraTrees;171,3.00000;234,3.00000;342,3.00000;[...]
3 F Value;4846,1.00000;6854,1.00000;4195,1.00000;[...]
4 CFS;6683,1.00000;6356,1.00000;1450,1.00000;[...]
5 [...]
```

1. `https://docs.python.org/2/library/pickle.html`

Listes classées :

```
MRMR;1760,1.00000;1243,0.50000;6993,0.33333;[...]
ExtraTrees;3251,4.50000;4846,2.25000;6854,1.50000;[...]
CFS;
[...]
```

Listes à scores :

```
MRMR;
ExtraTrees;3251,4.50000;4846,2.95205;2353,2.75077;[...]
F Value;4846,1.00000;6854,0.73751;4195,0.71070;[...]
CFS;
[...]
```

8.2 Implémentation

Ce sont les classes `CSVFeaturesExporter` et `CSVFeaturesImporter` qui sont responsables de l'export et de l'import des listes. Lors de l'import ou de l'export des listes, on demande de fournir un `group_name`. Ce dernier définit un préfixe pour les listes afin de pouvoir garder plusieurs groupes de listes et de pouvoir les comparer. Comme on peut le voir sur les exemples ci-dessus, les features sont séparées avec des points-virgules et le score est séparé de sa feature avec une virgule.

9. Combinaison des listes de features

Avec ce chapitre commence la seconde partie du projet, la combinaison de features (*features merging* en anglais). Le *notebook* `features_merging.ipynb` gère cette partie conformément à la pipeline évoquée en 3.1.

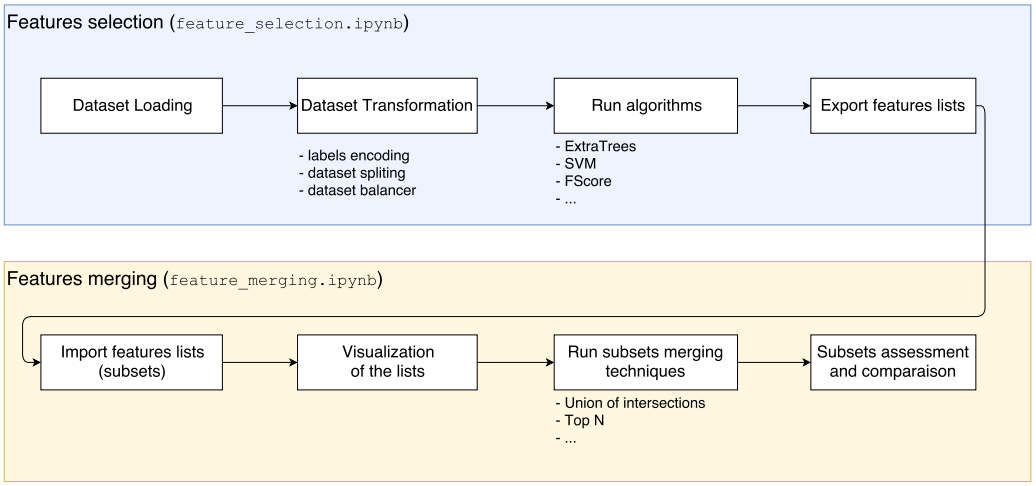

FIGURE 9.1 – Rappel : Pipeline du projet

Le but de ce *notebook* est de fusionner les listes en utilisant plusieurs techniques mais également d'évaluer la qualité de ces dernières. Premièrement, les techniques de fusion des listes sont présentées puis le protocole de mesure de ces techniques sera détaillé.

Les techniques de fusion des listes peuvent se baser sur les listes plates, par rang ou par score. Voici l'ensemble des techniques de fusion utilisées dans ce projet :

1. Union d'intersections ;
2. Union de toutes les listes ;
3. Top N features ;
4. Intersections deux à deux ;
5. Listes pondérées.

On signale qu'il est possible d'ajouter d'autres techniques de fusion des listes en héritant de la classe `SubsetMerger`. Plus d'informations sont disponibles à la section 11.2.4.

9.1 Technique Union d'intersections

Cette technique prend l'union de l'ensemble des intersections des listes prises deux à deux par une fenêtre glissante. La figure 9.2 illustre ce principe.

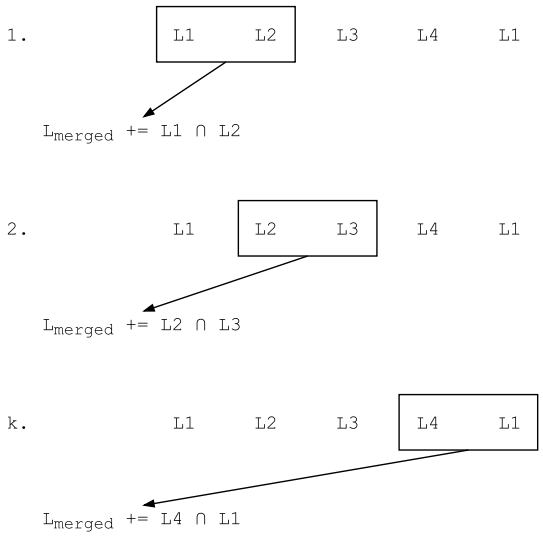

FIGURE 9.2 – Union d'intersections

La liste L_{merged} contient les features combinées. A chaque itération – donc à chaque fois que la fenêtre glisse – on ajoute à cette liste l'intersection de L_i et L_{i+1}. On note que la première liste est répliquée à la fin de la chaîne de manière à pouvoir prendre l'intersection de la première et de la dernière liste. Afin d'avoir un maximum de features, les listes sont triées par taille décroissante. On diminue ainsi le risque d'avoir une intersection d'une petite liste avec une grande ce qui empêcherait d'exploiter la plus grande liste. Dans le cas de ce projet, toutes les listes ont la même taille ce qui limite l'intérêt de cette astuce. Cependant, si cela venait à changer, il n'y aurait pas de modifications à faire dans la classe `UnionOfIntersectionsMerger`.

9.2 Technique Union de toutes les listes

Cette technique consiste simplement à prendre l'union de toutes les listes. Elle utilise les listes plates afin de n'exclure aucun algorithme qui ne pourrait pas fournir une liste à rang ou à score.

9.3 Technique Top N features

Top N est une technique qui crée une nouvelle liste à partir de toutes les autres, la trie par score descendant et en prend N. N a été fixé à 100 pour ne garder que les meilleurs.

Un des inconvénients de cette technique est qu'on peut privilégier un algorithme plutôt qu'un autre si celui-ci affirme que toutes ces features ont un grand score. Pour mitiger ceci, les scores des features ont été normalisés avant d'être triés. Cependant, cette normalisation est inefficace si un algorithme prétend que toutes ses features ont un score de 1.

9.4 Technique Intersections deux à deux

Cette technique prend l'intersection de deux listes, puis le résultat est intersecté avec la prochaine et ainsi de suite jusqu'à qu'il n'y plus de listes. Ce principe est illustré avec la figure ci-dessous.

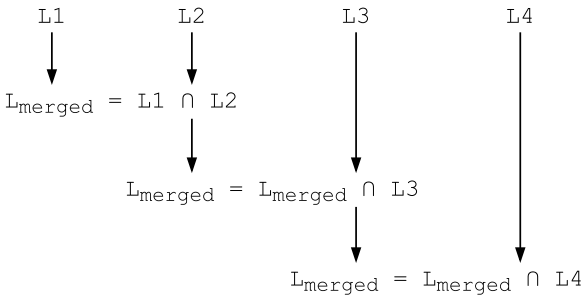

FIGURE 9.3 – Intersections deux à deux

9.5 Technique Listes pondérées

Cette technique, plus élaborée, utilise la distance entre les listes pour leur donner un poids. Ce poids est utilisé pour sélectionner des features de ces listes. La technique des listes pondérées emploie une *linkage matrix* similaire à ce qui est décrit à la section 7.4. Cette distance est utilisée pour mettre en avant les listes peu similaires par rapport aux autres. On souhaite également éviter que deux listes proches monopolisent l'ensemble des features sélectonnées.

La technique originale est le travail de A. Neves[76]. Cependant, après l'avoir implémentée et vérifiée à la main, il semblerait que cette technique ne couvre pas tous les cas. On a donc entrepris de créer une nouvelle technique en se basant sur les travaux de A. Neves.

L'explication et un pseudo-code de cette nouvelle technique sont disponibles en annexe. L'implémentation Python utilisée dans ce projet est matérialisé par la classe `MariglianoWeightedLists`.

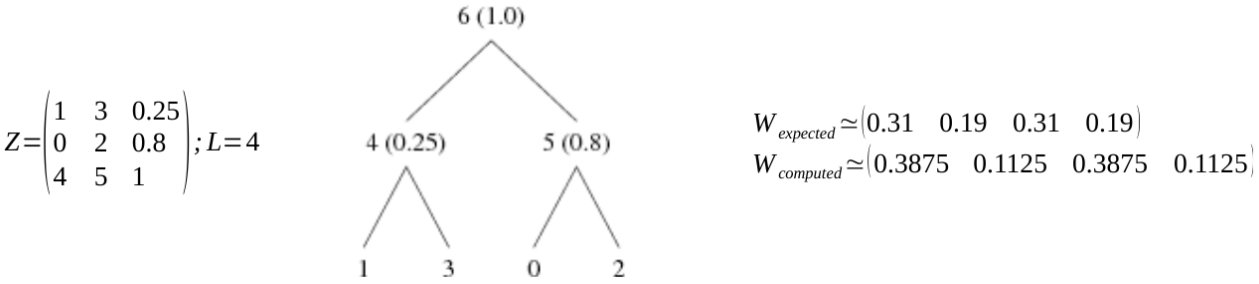

FIGURE 9.4 – Exemple MariglianoWeigtedLists

La figure, ci-dessus, montre un exemple de l'utilisation de la technique des listes pondérées. La *linkage matrix* est représentée par Z et $W_{computed}$ correspond aux poids des listes calculés par la technique. $W_{expected}$ correspond aux poids retournés par la technique originale.

9.5.1 En pratique

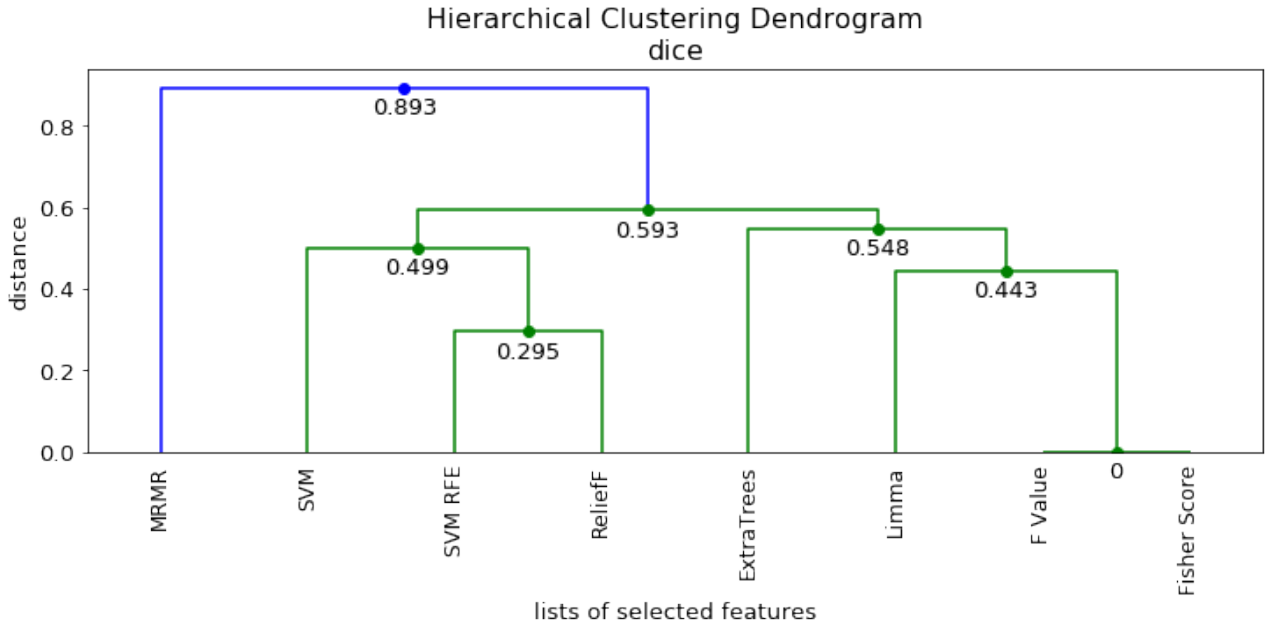

FIGURE 9.5 – Dendrogramme Golub

En pratique, on obtient la *linkage matrix* affichée par la figure 9.5 pour laquelle la technique attribue les poids suivants :

- MRMR : 0.333 ;
- ExtraTrees : 0.078 ;
- F Value : 0.045 ;
- SVM RFE : 0.100 ;
- SVM : 0.118 ;
- Fisher Score : 0.045 ;
- ReliefF : 0.100 ;
- Limma : 0.181.

On constate que :

- F Value et Fisher ont un poids égal entre elles mais plus faible que les autres listes car elles sont identiques ;
- SVM RFE et ReliefF ont également un poids égal mais cependant plus important que le couple F Value/Fisher Score car le distance n'est pas totalement nulle ;
- MRMR étant relativement différent des autres listes, il obtient un poids élevé
- SVM obtient un poids plus élevé que ExtraTrees situé au même niveau d'embranchement. Ceci s'explique parce que, bien qu'ExtraTrees ait une distance plus grande que SVM, sa distance avec ses listes frères et soeurs – Limma, F Value, Fisher Score – est plus proche que SVM avec les siens (SVM RFE et ReliefF).

Pour le dataset MILE, on observe le dendrogramme ci-dessous :

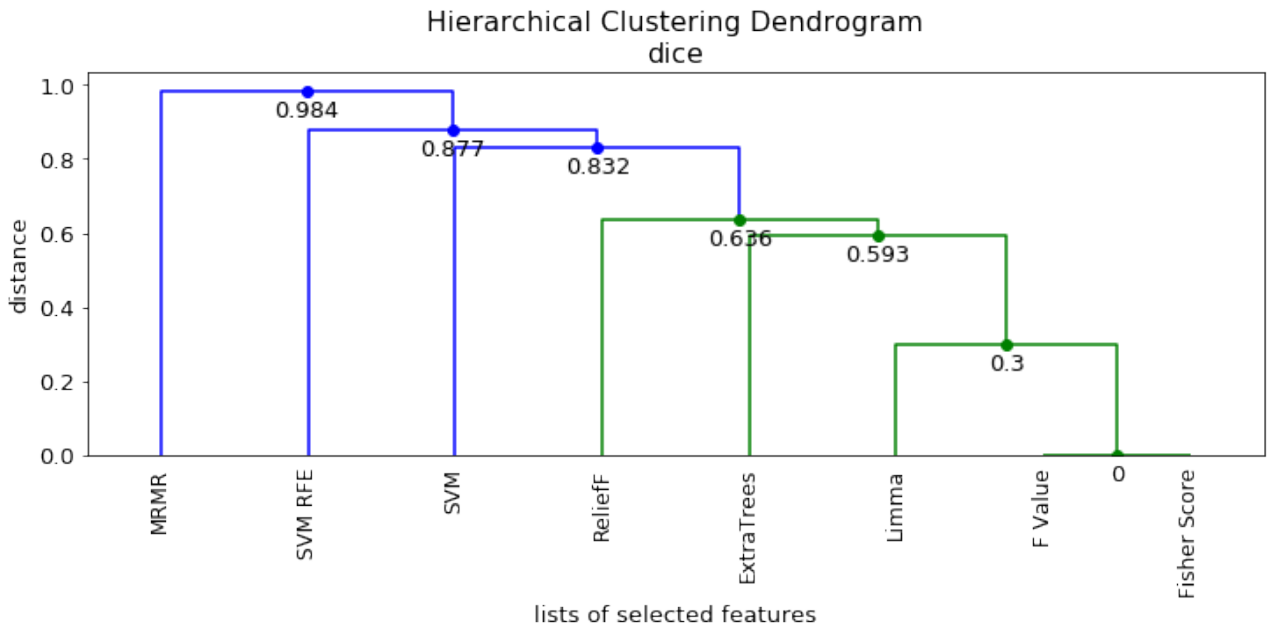

FIGURE 9.6 – Dendrogramme MILE

Et les poids suivants :

- MRMR : 0.333 ;
- ExtraTrees : 0.185 ;
- F Value : 0.065 ;
- SVM RFE : 0.069 ;
- SVM : 0.018 ;
- Fisher Score : 0.065 ;
- ReliefF : 0.003 ;
- Limma : 0.261.

On constate que les poids sont tous relativement faibles à l'exception de MRMR (0.333) et Limma (0.261). A elles seules, un poids de 0.591 est déjà attribué alors que visuellement, ces listes ne semblent pas si différentes des autres. Il y a certainement un problème dans la technique de pondération des listes développée. Serait-ce dû à la forme du dendrogramme qui embranche ses fils droits uniquement avec des noeuds ?

9.6 Analyse des résultats des techniques de fusion

Une façon d'évaluer la pertinence de ces listes – et donc de ces techniques de fusion – serait de les tester en laboratoire. Comme il est impossible de faire ceci à l'heure actuelle, on s'est rabattu sur une approche basée *machine learning*.

On a donc créé une classe `SubsetAssessor` dont le but est d'évaluer la performance de ces listes fusionnées en utilisant le score renvoyé par divers *classifiers*. Ces *classifiers* devraient – pour éviter tout favoritisme – être différents de ceux utilisés pour la production des listes. Les *classifiers* utilisés sont : un KNN, un réseau de neurones et ExtraTrees. En se limitant à un *classifier*, on risquerait que ce dernier favorise une liste plutôt qu'une autre. Si on en utilise plusieurs, ce risque diminue. De plus, ces algorithmes étant non-déterministes, il est nécessaire de les exécuter plusieurs fois et de calculer un score médian et un écart-type. Enfin, on rappelle que l'équilibrage des classes n'est effectué que pour le *training set*. F1 score a donc été utilisé pour éviter le problème de l'*accuracy paradox*[1].

1. Accuracy paradox : `https://en.wikipedia.org/wiki/Accuracy_paradox`

Avant de comparer les résultats obtenus, on souhaite faire une parenthèse et aborder le cas des listes aléatoires.

9.7 Analyse du cas des listes aléatoires

Dans ce projet, on a appris qu'un des problèmes principaux lors de la manipulation de *microarrays* concerne la corrélation, voire la redondance, entre les features. Si bien qu'il n'y a probablement pas qu'un seul groupe de features qui arrive à exprimer les différentes classes mais qu'il peut y en avoir plusieurs.

Partant de cette hypothèse, on a cherché à voir comment varie la performance à mesure qu'on augmente le nombre de features de listes aléatoires. Pour ce faire, on a utilisé la même classe `SubsetAssessor` que pour l'évaluation des listes fusionnées.

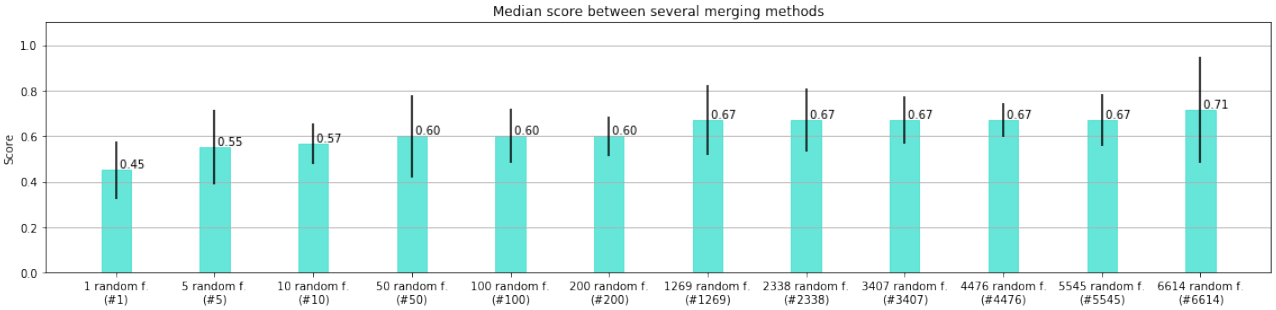

FIGURE 9.7 – Evolution du score médian de listes aléatoires à mesure que la taille augmente - Golub

Pour Golub, figure ci-dessus, on remarque que le score augmente légèrement à mesure que le nombre de features fournies croît. Le score, compris entre 0 et 1, n'est cependant pas très élevé.

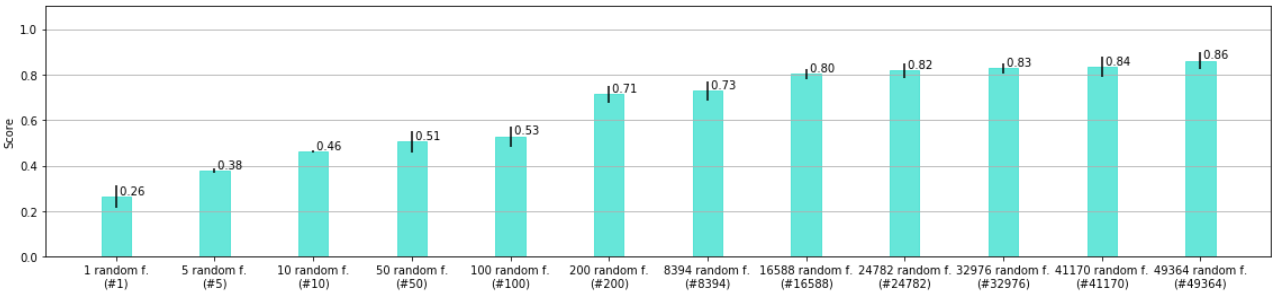

FIGURE 9.8 – Evolution du score médian de listes aléatoires à mesure que la taille augmente - MILE

Concernant MILE, on note avec l'image ci-dessus, que l'ajout de features permet d'augmenter le score significativement. C'est probablement dû au fait qu'il y ait plus de features dans ce dataset que dans Golub et qu'il serait donc plus simple de sélectionner de bonnes features car il y aurait plus de redondance. On observe que la variance est également plus faible que pour Golub. On voit que sélectionner 200 features ou environ 8000 renvoie un score proche et qu'il faut plus que doubler le nombre de features (> 16000) pour avoir une amélioration.

9.7.1 Comparaison des performances

En plus de comparer les techniques de fusion entre elles, on a choisi d'ajouter des listes composées de features tirées au hasard et d'une liste composée de l'ensemble de features. Ceci permet d'avoir une sorte de référence vis-à-vis des scores obtenus par les autres techniques.

Pour Golub, le *test set* est composé de 17 et 12 *samples* pour les deux classes. Le dataset contenant un nombre de *samples* limités, son score peut fortement varier.

La figure 9.9 montre que :

- Aucune technique surpasse les autres ;
- L'intersection deux à deux en utilisant les listes plates est composée d'aucune feature en commun ;
- L'intersection deux à deux en utilisant les listes par score renvoie une liste composée de 81 features sur les 3683 features uniques de toutes les listes (union) ;
- L'utilisation de toutes les features conduit à un grande variance.

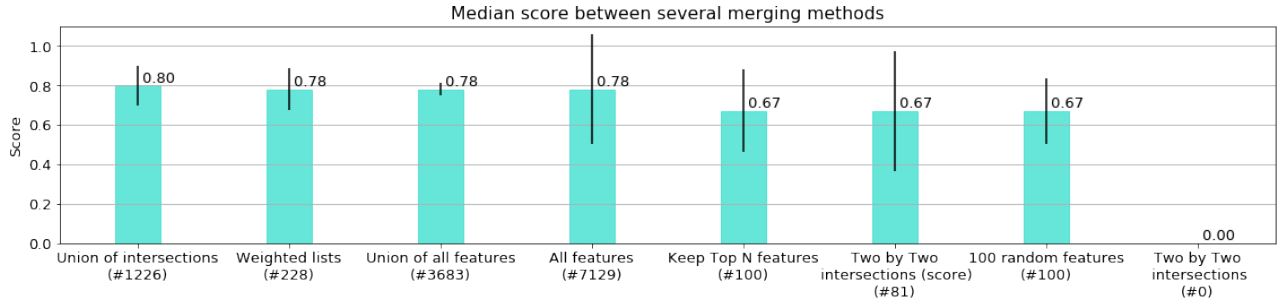

FIGURE 9.9 – Comparaison du score obtenu par les différentes techniques de fusion - Golub

Pour MILE, les classes contiennent respectivement 290, 223, 188, 81, 32 et 25 *samples*. On constate que :

- Prendre 100 features au hasard donne un score plus faible que certaines techniques comme l'union des intersections ou l'union de toutes les features. Par rapport à la figure 9.8, on remarque que même en augmentant le nombre de features prises au hasard à 8000, on obtient un score inférieur aux techniques mentionnées ;
- L'utilisation de toutes les features entraîne un grand écart type. Cela confirme la nécessité de réduire le nombre de features à utiliser ;
- Malgré les poids discutables calculés par la technique des listes pondérés pour MILE (voir 9.6), on constate qu'elle obtient un bon score. MRMR et Limma ayant la majorité – à elles seules – du poids, on peut se dire que ce sont elles qui ont contribué le plus à construire cette liste fusionnée. Par conséquent, cela confirmerait que ce sont de bonnes méthodes pour extraire des listes de features. Ceci n'est pas étonnant pour Limma qui est souvent utilisé avec les *microarrays* ;
- Ici aussi, l'intersection deux à deux de toutes les listes a été trop restrictive et n'a renvoyée aucune feature en commun.

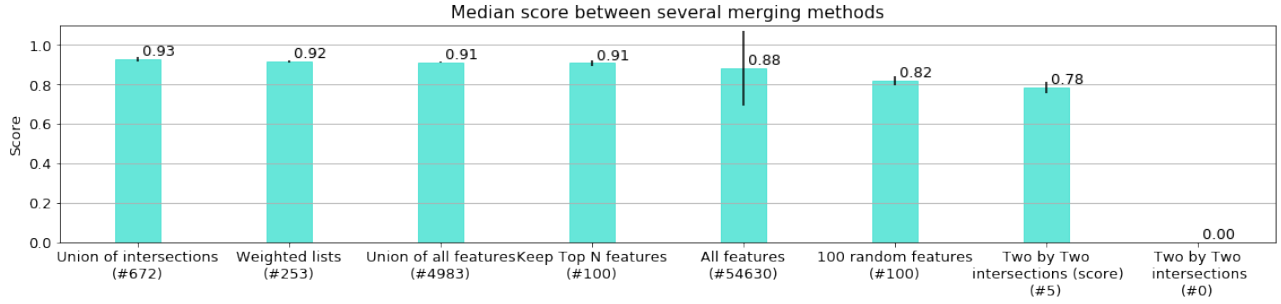

FIGURE 9.10 – Comparaison du score obtenu par les différentes techniques de fusion - MILE

10. Déroulement du projet

Ce chapitre présente visuellement le déroulement du projet. Une planification initiale, voir figure 10.1, a été modélisée sur la forme d'un diagramme de Gantt. Comme on peut le voir, cette planification est bien définie jusqu'au bilan M1. Ceci avait été un choix qui devait permettre de faire le point à mi-projet. Cependant, après plusieurs séances hebdomadaires, il a été jugé qu'il serait plus adapté de réaliser et de rectifier les objectifs par incréments à mesure de l'avancement du projet. En effet, de semaine en semaine, des difficultés sont rencontrées et on pense à de nouvelles façons de faire progresser le projet.

Planning initial

	Predecessors	Name	Duration	Start	Finish
1		Lectures	5 days	9/20/16 8:00 AM	9/26/16 5:00 PM
2	1	Prise en main des datasets	2 days	9/27/16 8:00 AM	9/28/16 5:00 PM
3	2	Choix des algorithmes	5 days	9/29/16 8:00 AM	10/5/16 5:00 PM
4	**3**	**Objectif 1**	**30 days**	**10/6/16 8:00 AM**	**11/16/16 5:00 PM**
5		Implémentation des algorithmes	25 days	10/6/16 8:00 AM	11/9/16 5:00 PM
6	5	Pipeline de combinaison	5 days	11/10/16 8:00 AM	11/16/16 5:00 PM
7		M1Bilan	0 days	10/31/16 9:00 AM	10/31/16 9:00 AM
8	**6**	**Objectif 2**	**25 days**	**11/17/16 8:00 AM**	**12/21/16 5:00 PM**
9		A définir	25 days	11/17/16 8:00 AM	12/21/16 5:00 PM
10		M2 Bilan	0 days	12/23/16 9:00 AM	12/23/16 9:00 AM
11	8	Tests et déploiement	5 days	1/27/17 9:00 AM	2/3/17 9:00 AM
12	11	Rédaction	5 days	2/3/17 9:00 AM	2/10/17 9:00 AM
13		M3 Fin du projet	0 days	2/10/17 9:00 AM	2/10/17 9:00 AM

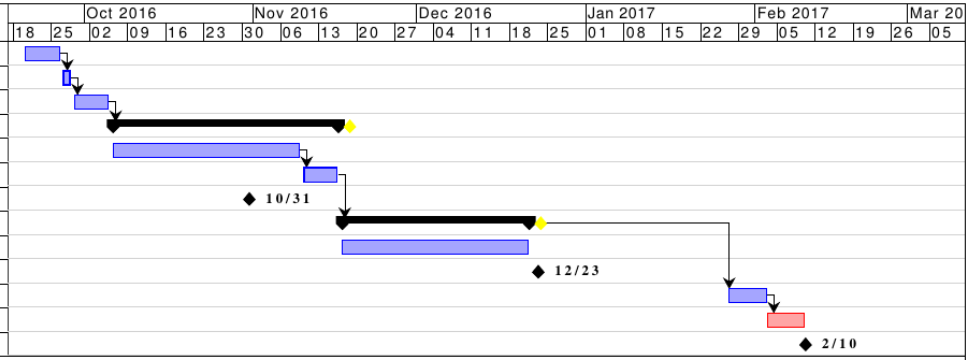

FIGURE 10.1 – Planning initial - 03.10.2016

Ainsi, on remarque pour la planification finale, voir figure 10.2, qu'il y a eu deux grandes tâches. La première concerne la feature selection et la seconde du feature merging conformément à la pipeline présentée en 3.1. Cette planification finale a été reconstruite à partir du journal de travail disponible en annexe. On remarque également que la partie feature selection a occupé environ la moitié du projet par rapport à la partie feature merging. On aurait souhaité pouvoir consacrer plus de temps à la seconde partie mais cela n'a pas été possible, car une bonne partie du projet s'est concentrée sur l'algorithme GAANN et sur différentes tentatives avec Limma.

Planning final

	Predecessors	Name	Duration	Start	Finish
1		Lectures	5 days	9/20/16 8:00 AM	9/26/16 5:00 PM
2	1	Prise en main des datasets	2 days	9/27/16 8:00 AM	9/28/16 5:00 PM
3	2	Choix des algorithmes	8 days	9/29/16 8:00 AM	10/10/16 5:00 PM
4	**3**	**Features selection**	**56 days?**	**10/11/16 8:00 AM**	**12/27/16 5:00 PM**
5		Implémentation des algorithmes	25 days	10/11/16 8:00 AM	11/14/16 5:00 PM
6	5	Implémentation de GAANN	14 days	11/15/16 8:00 AM	12/2/16 5:00 PM
7	6	Moyennage des listes	7 days?	12/5/16 8:00 AM	12/13/16 5:00 PM
8	7	Limma et R	10 days?	12/14/16 8:00 AM	12/27/16 5:00 PM
9	**4**	**Features merging**	**23 days?**	**12/28/16 8:00 AM**	**1/27/17 5:00 PM**
10		Impl. Neves weighted lists	3 days?	12/28/16 8:00 AM	12/30/16 5:00 PM
11	10	Impl. Marigliano weighted lists	10 days?	1/2/17 8:00 AM	1/13/17 5:00 PM
12	11	Impl. des techniques de merge	10 days?	1/16/17 8:00 AM	1/27/17 5:00 PM
13		Tests et déploiement	10 days	1/2/17 8:00 AM	1/13/17 5:00 PM
14		Rédaction	9 days	12/27/16 8:00 AM	1/6/17 5:00 PM
15	12	Rédaction	10 days	1/30/17 8:00 AM	2/10/17 5:00 PM

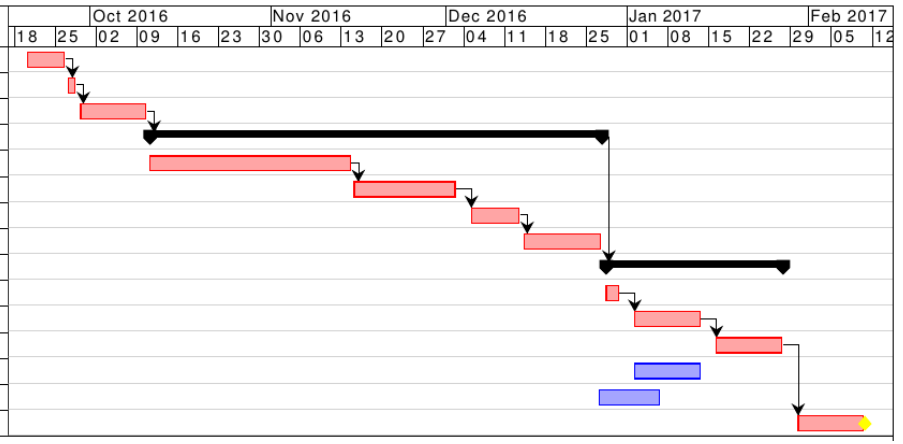

FIGURE 10.2 – Planning final - 05.02.2017

11. Reprise du projet

Ce chapitre montre la marche à suivre pour quiconque voudrait reprendre le projet. Afin de faciliter cette reprise, il a été choisi d'utiliser Docker car il permet d'encapsuler l'installation des dépendances du projet en une seule entité appelée container. Ainsi, le projet est séparé en deux parties. La première contient les sources du projet et la seconde contient un fichier `Dockerfile` qui décrit les étapes de création du container (choix de l'image de base, Ubuntu, Debian,..., installation des dépendances et paramétrage du projet). Il s'agit d'une alternative plus légère par rapport à la remise d'une machine virtuelle notamment. Une introduction rapide à Docker est disponible à l'appendice A.

11.1 Mise en route

Remarque : L'ensemble des commandes ci-dessous ont été testées avec une machine virtuelle Xubuntu 16.04. Il est recommandé d'utiliser une machine sous Linux malgré la potentielle compatibilité avec Windows grâce à l'utilisation de Docker.

Remarque : Voir aussi le README.md à la racine du projet.

11.1.1 Avec Docker (recommandé)

Pré-requis

- Une machine sous Linux capable d'exécuter Docker ;
- Si la machine utilisée n'est pas sous Linux, il va falloir utiliser une machine virtuelle. Il est recommandé d'utiliser la distribution Linux Xubuntu 16.04 (ou similaire).

Installation

- Installer Docker : `https://get.docker.com/`
  - Avec wget : `wget -qO- https://get.docker.com/ | sh`
  - Ou avec curl : `curl -sSL https://get.docker.com/ | sh`

A la fin de l'installation, il est recommandé de mettre l'utilisateur courant dans le groupe `docker` afin de pouvoir lancer des containers sans `sudo`. Pour ce faire, il suffit de faire les opérations suivantes et de se reconnecter à sa session :

```
1 sudo usermod -aG docker my_user
2
3 # demarrer Docker a l'ouverture de la session (Xubuntu et Debian-like)
4 sudo systemctl enable docker
5 sudo systemctl start docker
```

- Construire l'image Docker : `./build-docker.sh`
- Lancer Jupyter dans Docker : `./run-jupyter-docker.sh`

Ajouter les datasets

Normalement, il n'y a aucune manipulation supplémentaire à faire. Les datasets se trouvent déjà dans l'archive remise. Si les sources ont été récupérées depuis Github, alors il suffit de copier le dossier `data` et les fichiers `.pkl` à la racine du projet récupéré.

Ouvrez un navigateur web à l'adresse : `http://localhost:8888` et Jupyter devrait apparaître. Puis, ouvrez `DatasetVisualization.ipynb` et exécutez-le pour vérifier qu'il s'exécute sans problème.

Tests unitaires

Pour lancer les tests unitaires, il suffit de lancer le script `./run-unit-tests.sh`

Démarrer avec le projet

Le premier fichier qui devrait vous intéresser est `features_selection.ipynb`.

11.1.2 Sans Docker (non recommandé)

Remarque : L'exécution de Limma nécessite Docker mais le reste des notebooks peut fonctionner sans.

Pré-requis

— Une machine sous Linux ;
— git ;
— Python 2 et pip : `https://pip.pypa.io/en/stable/installing/`.

Récupération des sources

```
git clone https://github.com/krypty/BIO-SELECT.git
cd BIO-SELECT

# Si pip n'est pas installe
wget https://bootstrap.pypa.io/get-pip.py
sudo python2 get-pip.py
```

Création d'un environnement virtuel

```
sudo pip install virtualenv
virtualenv -p python2 bioselect
source bioselect/bin/activate
```

Installation des dépendances pour le projet

```
# Certaines librairies python utilisees ont besoin de packages supplementaires pour s'executer
sudo apt-get update && sudo apt-get install build-essential python-dev

# Installation des librairies requises
pip install -r requirements.txt
```

Ajouter les datasets

Normalement, il n'y a aucune manipulation supplémentaire à faire. Les datasets se trouvent déjà dans l'archive remise. Si les sources ont été récupérées depuis Github, alors il suffit de copier le dossier `data` et les fichiers `.pkl` à la racine du projet récupéré.

Lancement de Jupyter

Pour lancer Jupyter, il suffit d'exécuter la commande suivante : `jupyter-notebook`

Ouvrez `DatasetVisualization.ipynb` et exécutez-le pour vérifier qu'il s'exécute sans problème.

Tests unitaires

Pour lancer les tests unitaires, il faut lancer la commande :

```
1 source bioselect/bin/activate; pytest -v .
```

Démarrer avec le projet

Il est conseillé de commencer à parcourir les fichiers `DatasetVisualization.ipynb` et `features_selection.ipynb` en premier.

Le chargement des données et l'exécution des algorithmes peut être long malgré plusieurs tentatives d'optimisation.

11.2 Ajouts de fonctionnalités

Ce projet n'étant pas exhaustif, il peut être vu comme une plateforme de travail sur laquelle on peut ajouter et modifier ces fonctionnalités comme par exemple ajouter un algorithme de features selection ou une technique de fusion de listes. Cette section présente comment étendre les fonctionnalités du projet.

11.2.1 Ajouter un dataset

Les datasets sont gérés grâce à une paire de classes `Dataset` et `DatasetSampleParser`. La première permet de renseigner les *samples* au moyen d'une liste de fichiers et s'occupe de l'affectation d'un label aux *samples*. La seconde permet d'indiquer comment *parser* ces fichiers. `DatasetSampleParser` reçoit en argument un nom de fichier représentant un *sample*. Son rôle est de *parser* ce fichier et de renvoyer un vecteur contenant les features – donc les gènes – du sample. `Dataset` va, ainsi, créer une instance de `DatasetSampleParser` pour chaque fichier qu'il a reçu en argument. Au final, on se retrouve avec un attribut (`numpy array`) `X` où chaque ligne est un vecteur contenant les features d'un *sample* et un autre attribut `y` contenant les labels associés aux samples.

Pour ajouter un dataset, il suffit de créer, par héritage, une nouvelle paire de classes `Dataset` et `DatasetSampleParser` et d'implémenter les méthodes `_parse_labels()` pour la première classe et `_parse()` et `parse_features_names()` pour la seconde.

11.2.2 Ajouter un algorithme de sélection de features

Pour ajouter un algorithme, il suffit d'étendre la classe abstraire `Algorithm` et d'implémenter les méthodes suivantes :

- `get_best_features()`, obligatoire ;
- `get_best_features_by_rank()`, si supporté par l'algorithme ;
- `get_best_features_by_score_unnormed()`, si supporté par l'algorithme.

Remarque : Si l'algorithme est déterministe, il faut également hériter de `DeterministAlgorithm`. Cette interface n'a pas de méthodes mais est utilisée comme *flag* pour éviter d'effectuer le moyennage comme expliqué en 6.4.2.

Dans le cas où l'algorithme peut profiter d'un grid search, il convient d'hériter de la classe `GridSearchableAlgorithm` où il faut renseigner un dictionnaire avec comme clés le nom des paramètres à faire varier – par exemple `k` dans un KNN – et comme valeurs une liste de valeurs possibles pour le dit paramètre, par exemple `[3, 10, 20]`.

11.2.3 Ajouter une liste de features depuis une technique externe

Il est également possible d'utiliser une liste de features qui a été obtenue à partir d'une source externe, comme un script R par exemple. Pour ce faire, il faut procéder par les étapes suivantes :

1. Obtenir la liste de features ;
2. Convertir cette liste probablement composée des noms de gènes en liste d'indices utilisables pour la suite (*merging*). Cette opération peut être réalisée en utilisant la méthode `get_features_indices()` de la sous-classe `Dataset` appropriée ;
3. Remplir les listes simples, par rang et par score en utilisant la liste convertie du point 2
   - Les listes par rang et par score sont facultatives car sont peut-être non supportées par l'algorithme utilisé ;
   - Ces listes doivent être triées par score descendant/rang (facultatif pour la liste simple) ;
   - Si une liste de score est fournie, alors celle-ci doit être normalisée grâce à la méthode statique. `Algorithm.normalize_scores()`
4. Une fois les listes remplies, il faut les exporter en CSV grâce à la classe `CSVFeaturesExporter`.

Astuce : Pour l'exportation des features, il est possible d'utiliser plusieurs fichiers CSV séparés, dans un premier temps, si cela apporte plus de flexibilité (par exemple parce qu'on voudrait lancer plusieurs algorithmes en parallèle). Puis, de les concaténer – avec la commande Linux `cat` par exemple – pour avoir une seule liste utilisable pour la suite de la pipeline à savoir *features merging*.

La meilleure documentation reste tout de même le *notebook* `features_selection_limma.ipynb` qui applique ces étapes pour l'algorithme Limma.

11.2.4 Ajouter une technique de fusion de listes (merging)

Pour ajouter une technique de fusion des listes, il suffit d'étendre la classe `SubsetMerger` et d'implémenter la méthode `_merge()`. Toute classe héritant de `SubsetMerger` a à sa disposition une liste de listes de features. La nature de cette liste – simple/plate, par rang ou par score – est à préciser dans la *docstring* du constructeur du `SubsetMerger` nouvellement créé. Par exemple, `UnionSubsetMerger` ignore la notion de rang ou de score, il convient donc d'instancer cette classe en spécifiant la nature des listes ainsi : `UnionSubsetMerger(subsets["features"].values())`. Le *notebook* `features_merging` fournit d'autres exemples plus complets.

11.3 Qualité du code

Le code source respecte dans sa grande majorité la norme PEP8 [1] permettant une certaine clarté et constance dans le style de codage. Le code est commenté en utilisant les docstrings [2].

Afin d'éviter la régression de code, plusieurs tests unitaires ont été écrits. Ils ne couvrent pas l'ensemble du projet mais se concentrent sur les classes critiques comme le *parsing* de dataset ou certains algorithmes. De plus, les tests unitaires peuvent aussi faire office d'aide pour comprendre le fonctionnement d'une classe. Pour lancer les tests unitaires, voir la section 11.1.

Pour ce projet, on a été contraint d'utiliser Python 2 en lieu et place de Python 3 à cause de la librairie scikit-feature. Il aurait été préférable de démarrer ce projet en utilisant Python 3 car il est plus récent, plus rapide et la plupart des librairies ont tendance à préférer Python 3 également. Ce choix n'est néanmoins pas critique car la majorité du code — hormis celui de scikit-feature — a été écrit de façon à être portable en Python 3.

1. `https://www.python.org/dev/peps/pep-0008/`
2. `https://www.python.org/dev/peps/pep-0257/`

12. Conclusion

Note : Pour cette dernière partie, je m'autorise un ton plus personnel notamment pour l'autocritique.

Bref résumé

Les objectifs de ce projet étaient d'extraire des listes de features en utilisant plusieurs algorithmes. Puis d'utiliser ces listes en les combinant pour en obtenir d'autres, à priori, plus robustes. Pour parvenir à ces buts, on a d'abord lu plusieurs articles scientifiques [1] puis pensé à utiliser une structure de projet de type pipeline. Ensuite, plusieurs datasets ont été téléchargés et pré-traités. Puis, plusieurs algorithmes ont été utilisés pour extraire ces features. Enfin, ces listes ont été fusionnées et évaluées dans la dernière partie de ce projet.

Résultats obtenus

Il convient de distinguer deux types de résultats. On considère la première catégorie comme étant des résultats tangibles, directement utilisables, tel un produit. Pour cette catégorie, je pense que les résultats obtenus ne sont pas aussi satisfaisants qu'espéré. En effet, ce travail ne permet pas de répondre à la question "Pouvez-vous me donner une technique de combinaison de listes qui ne retient que les meilleures features ?".

Néanmoins, concernant la seconde catégorie de résultats, ceux indirectement utilisables ou annexes, on peut alors mettre en avant le travail effectué. En effet, il fournit une plateforme réutilisable pour poursuivre les objectifs demandés. On peut utiliser cette dernière pour ajouter d'autres datasets, algorithmes de *features selection* et de techniques de fusion. L'ensemble du code est commenté et des tests unitaires ont été mis en place pour éviter de la régression et peuvent être utilisés comme une forme de documentation.

Sur le plan personnel, j'ai énormément appris – un peu en bio-informatique, plus en *machine learning* – et c'est aussi un type de résultat.

Difficultés rencontrées

— Introduction aux *microarrays* : même si une compréhension totale de ces derniers n'est pas nécessaire d'un point de vue *machine learning*, il reste cependant indispensable, il semble, de savoir d'où proviennent les données qu'on utilise et quelles caractéristiques elles possèdent. Un cas concret serait, par exemple, la présence de sondes commençant par "AFFX-" dans les *samples* qu'il convient d'ignorer ;
— Limma et R : n'ayant aucune connaissance de R ni de Limma, il s'est avéré très difficile de prendre en main ces outils. Heureusement, Mme Zahra Mungloo- Dilmohamud a pu répondre à certaines de mes questions et on a tout de même pu utiliser Limma ;
— La taille des données : de manière plus générale, travailler avec autant de données mène rapidement à des problèmes. Par exemple, la consommation mémoire augmente rapidement tout comme les temps de calculs. Pour contourner ce problème, on a essayé de réduire le nombre de *samples* utilisés pour accélérer les calculs – au détriment de résultats moins précis – et on a tenté de paralléliser le plus possible les exécutions séquentielles (voir GAANN). Enfin, la séparation de la pipeline en deux parties a permis de ne pas à avoir à recalculer les listes à chaque lancement.

1. Aucune mention n'est faite dans le rapport, cependant de nombreux articles ont été lus et la plupart se trouvent dans la bibliographie

Pistes à explorer

Parmi les pistes à explorer, on peut citer :

- L'utilisation détournée de PCA pour sélectionner des features. On ne chercherait plus à réduire la dimension de l'espace d'entrée. Mais on pourrait utiliser les composantes principales pour trouver une pondération des features ;
- Dans la même idée, on pourrait aussi s'intéresser aux *autoencoders* et essayer de trouver un moyen d'utiliser les poids W et les entrées approximées $\hat{X}$ pour sélectionner des features. En effet, en "résumant" X (les features originales) en $\hat{X}$ (features approximées), l'*autoencoder* devrait se concentrer sur les features les plus importantes pour avoir la meilleure approximation globale. Donc, si on part de l'hypothèse que l'*autoencoder* arrive mieux à reconstruire des features discriminantes que des features redondantes ou bruitées, alors on pourrait sélectionner les features dont les différences $d_i = abs(X_i - \hat{X}_i)$ sont les plus faibles ;
- Au lieu de demander absolument un nombre exact de features par algorithme, on pourrait se limiter aux features les plus importantes par algorithme, quitte à utiliser plusieurs listes du même algorithme si celui-ci est non-déterministe.

Autocritique

Points positifs :

- On fournit une plateforme réutilisable pour le problème de la sélection de features ;
- Le projet est modulaire et extensible grâce à une hiérarchie de classes. On peut ajouter des datasets, des algorithmes de features selection et des techniques de fusion ;
- Le projet est documenté, respecte des conventions de codage Python et propose des tests unitaires ;
- La reprise de ce projet est détaillée dans ce rapport et l'utilisation d'outils comme Docker ou un environnement virtuel Python facilite sa mise en route ;
- Je pense que mon approche scientifique s'est améliorée durant ce projet. J'imagine m'être posé des questions pertinentes auxquelles j'ai tenté de répondre ;
- Sur le plan personnel, j'ai beaucoup appris et ne regrette pas d'avoir choisi ce sujet.

Points améliorables :

- Je pense que j'aurais dû consacrer plus de temps à visualiser les datasets avec lesquels j'ai travaillé. Cela aurait peut-être permis d'éliminer, à la main, certaines features que des algorithmes n'auraient pas fait ;
- Je pense qu'une petite formation à Limma, à R et une introduction aux *chips* de AffyMetrix – ou, du moins, une introduction à la lecture des *samples* des datasets – aurait pu être profitable pour éviter de dévouer plusieurs semaines à imaginer comment ceci fonctionne ;
- J'aurais dû écarter l'algorithme GAANN plus rapidement et commencer les techniques de fusion de features.

Remerciements

Je tiens à remercier toutes les personnes qui m'ont soutenu et aidé dans ce projet. Je souhaite remercier plus particulièrement M. Carlos Andrés Pena et Mme. Zahra Mungloo- Dilmohamud qui m'ont guidé et apporté de précieuses ressources à la réalisation de ce projet. Cela a été un plaisir sincère de collaborer avec eux.

Lausanne, février 2017

Gary Marigliano

Bibliographie

[1] 1.11. Ensemble methods — scikit-learn 0.18 documentation. http://scikit-learn.org/stable/modules/ensemble.html.

[2] 1.13. Feature selection — scikit-learn 0.18 documentation. http://scikit-learn.org/stable/modules/feature_selection.html.

[3] About Feature Scaling and Normalization. http://sebastianraschka.com/Articles/2014_about_feature_scaling.html.

[4] Algorithmes génétiques. http://www-igm.univ-mlv.fr/~dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html.

[5] Analyze your own microarray data in R/Bioconductor - BITS wiki. http://wiki.bits.vib.be/index.php/Analyze_your_own_microarray_data_in_R/Bioconductor#Install_the_required_R_packages.

[6] Artificial bee colony algorithm - Scholarpedia. http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm.

[7] Bioinfo-fr.net. http://bioinfo-fr.net/.

[8] BioInformatics Group Seville. http://eps.upo.es/bigs/datasets.html.

[9] Cancer Program Legacy Publication Resources. http://portals.broadinstitute.org/cgi-bin/cancer/publications/pub_paper.cgi?mode=view&paper_id=43.

[10] Classification of DNA microarrays using artificial neural networks and ABC algorithm. http://www.sciencedirect.com/science/article/pii/S1568494615006171.

[11] A comparative study of different machine learning methods on microarray gene expression data | BMC Genomics | Full Text. https://bmcgenomics.biomedcentral.com/articles/10.1186/1471-2164-9-S1-S13.

[12] Cross Validation. https://www.cs.cmu.edu/~schneide/tut5/node42.html.

[13] Crossover (genetic algorithm) - Wikipedia. https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).

[14] DNA microarray - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/DNA_microarray.

[15] DNA microarrays - YouTube. https://www.youtube.com/watch?v=VNsThMNjKhM.

[16] DNU_project/golub.csv at master · zoberg/DNU_project. https://github.com/zoberg/DNU_project/blob/master/datasets/golub.csv.

[17] E-GEOD-13159 < Browse < ArrayExpress < EMBL-EBI. http://www.ebi.ac.uk/arrayexpress/experiments/E-GEOD-13159/.

[18] E-GEOD-22619 - Genome wide expression data from discordant twins. http://www.ebi.ac.uk/arrayexpress/experiments/E-GEOD-22619/?query=1.+GSE22619.

[19] Feature Selection in Python with Scikit-Learn - Machine Learning Mastery. http://machinelearningmastery.com/feature-selection-in-python-with-scikit-learn/.

[20] Fitness proportionate selection - Wikipedia. https://en.wikipedia.org/wiki/Fitness_proportionate_selection.

[21] Gene - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Gene.

[22] GEO Accession viewer. https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSM329871.

[23] GitHub - biopython/biopython : Official git repository for Biopython (converted from CVS). https://github.com/biopython/biopython.

[24] GitHub - jundongl/scikit-feature : Oepn-source feature selection repository in python (DMML Lab@ASU). https://github.com/jundongl/scikit-feature.

[25] Home | Feature Selection @ ASU. http://featureselection.asu.edu/.

[26] Implementing a Weighted Majority Rule Ensemble Classifier. http://sebastianraschka.com/Articles/2014_ensemble_classifier.html.

[27] Introduction à la structure secondaire des ARN | bioinfo-fr.net. http://bioinfo-fr.net/introduction-a-la-structure-secondaire-des-arn.

[28] Introduction à matplotlib. http://www.python-simple.com/python-matplotlib/matplotlib-intro.php.

[29] Introduction to Machine Learning with Python and Scikit-Learn. http://kukuruku.co/hub/python/introduction-to-machine-learning-with-python-andscikit-learn.

[30] An Introduction to Supervised Learning via Scikit Learn | Bugra Akyildiz. http://bugra.github.io/work/notes/2014-11-22/an-introduction-to-supervised-learning-scikit-learn/.

[31] Leukemia data. https://web.stanford.edu/~hastie/CASI_files/DATA/leukemia.html.

[32] Leukemia : Types, Symptoms, & Treatment. http://www.healthline.com/health/leukemia.

[33] Limma. https://kasperdanielhansen.github.io/genbioconductor/html/limma.html.

[34] Matplotlib gallery – Python Tutorial. https://pythonspot.com/en/matplotlib-gallery/.

[35] Mutation (genetic algorithm) - Wikipedia. https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm).

[36] A primer on gene expression and microarrays for machine learning researchers. http://www.sciencedirect.com/science/article/pii/S1532046404000693.

[37] Project Jupyter. http://www.jupyter.org.

[38] Puce à ADN — Wikipédia. https://fr.wikipedia.org/wiki/Puce_%C3%A0_ADN.

[39] Quickstart tutorial — NumPy v1.12.dev0 Manual. https://docs.scipy.org/doc/numpy-dev/user/quickstart.html.

[40] Redundancy based feature selection for microarray data. http://dl.acm.org/citation.cfm?id=1014149.

[41] A Review of Feature Selection and Feature Extraction Methods Applied on Microarray Data. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4480804/.

[42] A review of microarray datasets and applied feature selection methods. http://www.sciencedirect.com/science/article/pii/S0020025514006021.

[43] Scikit-learn-contrib/imbalanced-learn : Python module to perform under sampling and over sampling with various techniques. https://github.com/scikit-learn-contrib/imbalanced-learn.

[44] se429900531p - 531.full.pdf. http://science.sciencemag.org/content/sci/286/5439/531.full.pdf.

[45] Selecting good features – Part IV : Stability selection, RFE and everything side by side | Diving into data. http://blog.datadive.net/selecting-good-features-part-iv-stability-selection-rfe-and-everything-side-by-side/.

[46] Selection (genetic algorithm) - Wikipedia. https://en.wikipedia.org/wiki/Selection_(genetic_algorithm).

[47] Significance Analysis of Microarrays. http://statweb.stanford.edu/~tibs/SAM/.

[48] A Simple and Efficient Artificial Bee Colony Algorithm. https://www.hindawi.com/journals/mpe/2013/526315/.

[49] sklearn.feature_selection.SelectKBest — scikit-learn 0.18 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest.

[50] A Survey on Filter Techniques for Feature Selection in Gene Expression Microarray Analysis. http://dl.acm.org/citation.cfm?id=2223923.

[51] Types of Leukemia : 4 Primary Types | CTCA. http://www.cancercenter.com/leukemia/types/.

[52] Docker 1.10 et les user namespace, mai 2016. http://linuxfr.org/users/w3blogfr/journaux/docker-1-10-et-les-user-namespace.

[53] Docker (software), mai 2016. https://en.wikipedia.org/wiki/Docker_%28software%29.

[54] Orly Alter, Patrick O. Brown, and David Botstein. Singular value decomposition for genome-wide expression data processing and modeling. *Proc. Natl. Acad. Sci.*, 97(18) :10101–10106, 2000.

[55] Naomi S. Altman. Differential Expression Analysis using LIMMA. 2013.

[56] V. Bolón-Canedo, N. Sánchez-Maroño, A. Alonso-Betanzos, J.M. Benítez, and F. Herrera. A review of microarray datasets and applied feature selection methods. *Information Sciences*, 282 :111 – 135, 2014.

[57] Jenna Carr. An introduction to genetic algorithms. *See Karczmarczuk Users Greyc FrTEACHIAD-GenDoccarrGenet Pdf*, 2014.

[58] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers and Electrical Engineering*, 40(1) :16 – 28, 2014. 40th-year commemorative issue.

[59] Sung-Bae Cho and Hong-Hee Won. Machine learning in DNA microarray analysis for cancer classification. In *Proceedings of the First Asia-Pacific Bioinformatics Conference on Bioinformatics 2003-Volume 19*, pages 189–198. Australian Computer Society, Inc., 2003.

[60] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[61] Ramón Díaz-Uriarte and Sara Alvarez De Andres. Gene selection and classification of microarray data using random forest. *BMC Bioinformatics*, 7(1) :1, 2006.

[62] Todd Golub. *golubEsets : exprSets for golub leukemia data*, 2016. R package version 1.16.0.

[63] Hanchuan Peng. Minimum Redundancy and Maximum Relevance Feature Selection and Its Applications. http://penglab.janelia.org/proj/mRMR/BIBM07_mRMR_071103_handout.pdf.

[64] Xiaofei He, Deng Cai, and Partha Niyogi. Laplacian score for feature selection. In *Advances in Neural Information Processing Systems*, pages 507–514, 2005.

[65] Edmundo Bonilla Huerta, Béatrice Duval, and Jin-Kao Hao. A hybrid GA/SVM approach for gene selection and classification of microarray data. In *Workshops on Applications of Evolutionary Computation*, pages 34–44. Springer, 2006.

[66] Docker Inc. Understand images, containers, and storage drivers, mai 2016. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

[67] Docker Inc. What is Docker ?, mai 2016. https://www.docker.com/what-docker.

[68] Jason Brownlee. 8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset - Machine Learning Mastery. http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/.

[69] Thanyaluk Jirapech-Umpai and Stuart Aitken. Feature selection and classification for microarray data analysis : Evolutionary methods for identifying predictive genes. *BMC Bioinformatics*, 6(1) :1, 2005.

[70] Kasper Daniel Hansen. Limma - Johns Hopkins University | Coursera. https://www.coursera.org/learn/bioconductor/lecture/aoNwW/limma, 2016.

[71] Ron Kohavi. Data Mining and Visualization Silicon Graphics, Inc. 2011 N. Shoreline Blvd Mountain View, CA 94043-1389. 1996.

[72] Ammu Prasanna Kumar and Preeja Valsala. Feature Selection for high Dimensional DNA Microarray data using hybrid approaches. *Bioinformation*, 9(16) :824, 2013.

[73] Lee Jacobson. Creating a genetic algorithm for beginners. http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3, 12th February 2012.

[74] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn : A python toolbox to tackle the curse of imbalanced datasets in machine learning. *CoRR*, abs/1609.06570, 2016.

[75] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. Feature selection : A data perspective. *arXiv preprint arXiv :1601.07996*, 2016.

[76] Aitana Neves. Ensemble combinaison rules. Technical report, Haute École d'ingénierie et de gestion du canton de Vaud, 2015.

[77] Katherine Noyes. Docker : A 'Shipping Container' for Linux Code, mai 2016. `https://www.linux.com/news/docker-shipping-container-linux-code`.

[78] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.

[79] Matthew E Ritchie, Belinda Phipson, Di Wu, Yifang Hu, Charity W Law, Wei Shi, and Gordon K Smyth. limma powers differential expression analyses for RNA-sequencing and microarray studies. *Nucleic Acids Research*, 43(7) :e47, 2015.

[80] Y. Saeys, I. Inza, and P. Larranaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19) :2507–2517, October 2007.

[81] Sebastian Raschka. An introduction to parallel programming using Python's multiprocessing module. `http://sebastianraschka.com/Articles/2014_multiprocessing.html`.

[82] Qiang Shen, Ren Diao, and Pan Su. Feature Selection Ensemble. *Turing-100*, 10 :289–306, 2012.

[83] John D. Storey, Wenzhong Xiao, Jeffrey T. Leek, Ronald G. Tompkins, and Ronald W. Davis. Significance analysis of time course microarray experiments. *Proc. Natl. Acad. Sci. U. S. A.*, 102(36) :12837–12842, 2005.

[84] A. H. Sung. Ranking importance of input parameters of neural networks. *Expert Syst. Appl.*, 15(3) :405–411, 1998.

[85] Tibshirani Robert and Bair Eric. Machine Learning Methods Applied to DNA Microarray Data Can Improve the Diagnosis of Cancer. `http://www-staff.it.uts.edu.au/~paulk/teaching/dma12/papers/m06-bair.pdf`.

[86] Warren S. Sarle. How to measure importance of inputs ? `ftp://ftp.sas.com/pub/neural/importance.html#intro`, June 2000.

[87] Will Larson. Genetic Algorithms : Cool Name & Damn Simple. `http://lethain.com/genetic-algorithms-cool-name-damn-simple/`.

[88] Limsoon Wong. Lecture 4 : Gene Expression Analysis. 2012.

[89] Eric P. Xing, Michael I. Jordan, Richard M. Karp, and others. Feature selection for high-dimensional genomic microarray data. In *ICML*, volume 1, pages 601–608. Citeseer, 2001.

[90] Zheng Zhao, Fred Morstatter, Shashvata Sharma, Salem Alelyani, Aneeth Anand, and Huan Liu. Advancing feature selection research. *ASU Feature Sel. Repos.*, pages 1–28, 2010.

Appendices

A. Introduction rapide à Docker

Note : Ce document a été adapté de mon PA du semestre passé. La lecture de cette annexe est facultative dans le sens où les explications données au chapitre 11 devraient suffire à utiliser les *notebooks* Jupyter sans pour autant comprendre les mécanismes de Docker.

A.1 Introduction

Le but de cette annexe est d'apporter un contexte quant à l'utilisation de Docker pour ce projet notamment en cas de reprise. Si le lecteur souhaite connaître les tréfonds de Docker, il est lui sera nécessaire de lire d'avantage de documentation sur Internet, par exemple.

Docker est un outil qui permet d'empaqueter une application et ses dépendances dans un container léger, autosuffisant, isolé et portable. En intégrant leur application dans un container, les développeurs s'assurent que celle-ci va tourner sur n'importe quel environnement GNU/Linux. Ainsi, le temps passé à configurer les différents environnements (développement, test et production typiquement) est réduit, voire même unifié[77][53][67].

Les caractéristiques principales des containers Docker sont les suivantes :

- *légers* : les containers partagent le même kernel et librairies que le système d'exploitation hôte permettant ainsi un démarrage rapide des containers et une utilisation mémoire contenue ;
- *isolés* : les containers sont isolés grâce à des mécanismes offerts par le kernel. Voir section A.5 pour plus de détails ;
- *éphémères et maintenables* : les containers sont conçus pour être créés et détruits régulièrement contrairement à un serveur ou une machine virtuelle pour lesquels un arrêt est souvent critique. Ils sont maintenables dans le sens où il est possible de revenir à une version précédente facilement et qu'il est aisé de déployer une nouvelle version.

Les sections qui suivent abordent un peu plus en profondeur certains points clés de Docker qui ont été jugés importants.

A.2 Containers vs machines virtuelles

Suite à la lecture de la section précédente, il serait normal de se dire que ces containers partagent certaines caractéristiques avec les machines virtuelles.

Voici les différences majeures qu'il existe entre les containers et les machines virtuelles[67].

Les machines virtuelles possèdent leur propre OS qui embarque ses propres binaires et librairies. Ceci engendre une perte d'espace disque importante surtout si les binaires ou libraires sont communes à plusieurs machines virtuelles. De plus, démarrer une machine virtuelle prend du temps (jusqu'à plusieurs minutes). En outre, les machines virtuelles doivent installer leurs propres drivers afin de communiquer avec l'hyperviseur (logiciel s'exécutant à l'intérieur d'un OS hôte qui gère les machines virtuelles). Un avantage cependant est l'isolation complète d'une machine virtuelle qui ne peut communiquer avec les autres par défaut.

Les containers s'exécutent de manière isolée par-dessus l'OS hôte qui partage ses ressources (kernel, binaires, librairies, périphériques...). Plus légers, les containers démarrent en quelques secondes seulement. Sur une machine, il est tout à fait possible de lancer des milliers de containers similaires, car l'empreinte mémoire est réduite et l'espace disque occupé est partagé si les containers sont semblables. Ceci est expliqué plus en détail à la section A.4. Les containers sont isolés, mais ils peuvent aussi communiquer entre eux si on leur a explicitement donné l'autorisation.

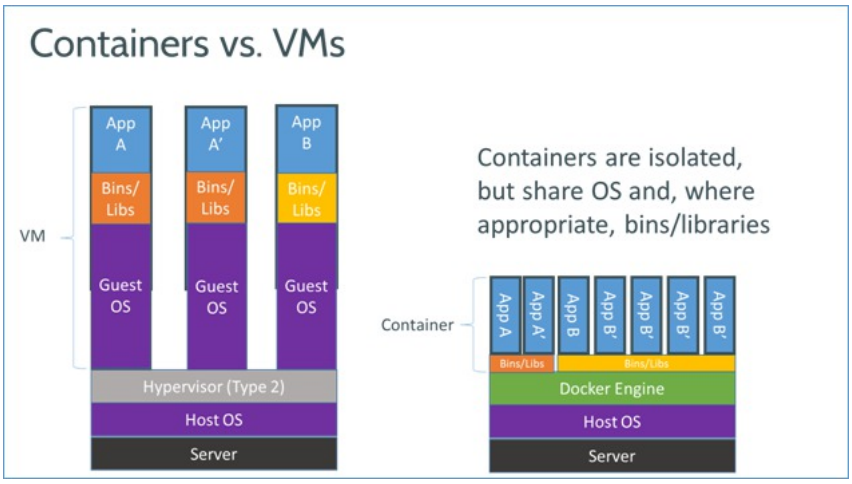

FIGURE A.1 – Machines virtuelles vs Containers

A.3 Docker images et Docker containers

Avec Docker, une application est encapsulée avec toutes ses dépendances et sa configuration dans une **image**.

Pour construire cette image, on utilise un Dockerfile. Il s'agit d'un fichier qui décrit les étapes de création et de configuration nécessaires à l'obtention de l'application configurée. C'est dans ce fichier qu'on retrouve l'OS à utiliser, les dépendances à installer et toutes autres configurations utiles au bon fonctionnement de l'application à déployer.

Typiquement un Dockerfile permettant de lancer un serveur web Nginx qui affiche un "hello world" ressemble à ceci :

```
1  FROM alpine  # image de depart
2  MAINTAINER support@tutum.co  # mainteneur du Dockerfile
3  RUN apk --update add nginx php-fpm && \  # installation des dependances
4      mkdir -p /var/log/nginx && \
5      touch /var/log/nginx/access.log && \
6      mkdir -p /tmp/nginx && \
7      echo "clear_env = no" >> /etc/php/php-fpm.conf
8  ADD www /www  # ajout des sources de l'application
9  ADD nginx.conf /etc/nginx/  # ajout d'un fichier de configuration
10 EXPOSE 80  # ouverture du port 80
11 CMD php-fpm -d variables_order="EGPCS" && (tail -F /var/log/nginx/access.log
   ↪  &) && exec nginx -g "daemon off;" # commande a lancer au lancement du
   ↪  container
```

Source : `https://github.com/tutumcloud/hello-world/blob/master/Dockerfile`

Un Dockerfile est en quelque sorte la recette de cuisine qui permet de construire une image Docker.

Une fois l'image construite, on peut exécuter l'application dans un container. Un container Docker est donc une instance de l'image fraîchement créée. La figure A.2 montre les relations entre un Dockerfile, une image et un container.

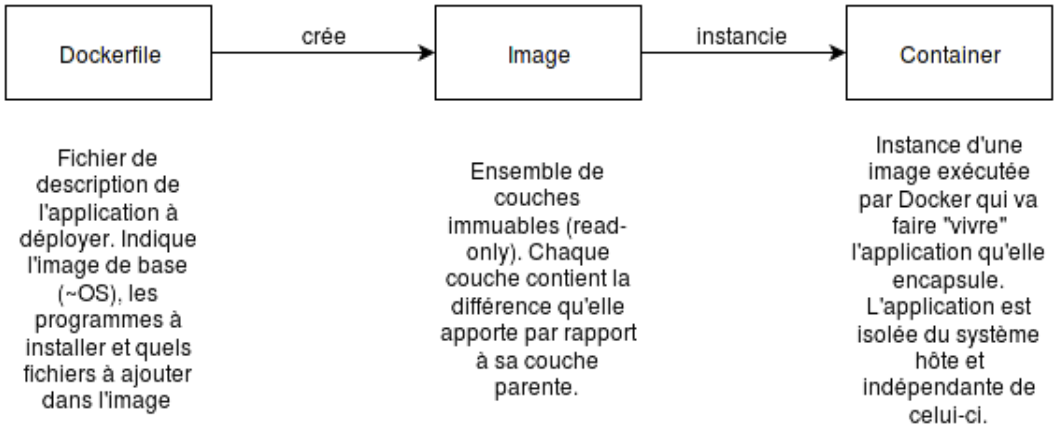

FIGURE A.2 – Dockerfile, image et container

A.4 Système de fichiers en couche

Chaque image Docker est composée d'une liste de couches (*layers*) superposées en lecture seule[66]. Chaque couche représente la différence du système de fichiers par rapport à la couche précédente. Sur la figure A.3, on peut voir 4 couches (identifiables par une chaîne de caractères unique, par exemple `91e54dfb1179`) et leur taille respective.

FIGURE A.3 – Couches d'une image Docker

À la création d'un container, une nouvelle couche fine est ajoutée. Cette couche, appelée "container layer" est accessible en écriture durant l'exécution du container. La figure A.4 le montre clairement.

Un mot supplémentaire sur une nouvelle caractéristique arrivée avec Docker 1.10 (mars 2016) ; avant cette version, Docker attribuait des UUID [1] générés aléatoirement pour identifier les couches d'une image. Désormais, ces UUID sont remplacés par des hash appelés *secure content hash*.

Les différences principales entre un UUID et un hash sont :

- Un UUID est généré aléatoirement, donc deux images exactement identiques auront deux UUID différents alors qu'en utilisant un hash, le résultat sera identique ;
- Avec les UUID, même si la probabilité est rare [2], il est possible de générer deux fois le même UUID ce qui peut poser des problèmes lors de la construction des images ;

1. UUID : `https://fr.wikipedia.org/wiki/Universal_Unique_Identifier`
2. Probabilité de doublon : `https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates`

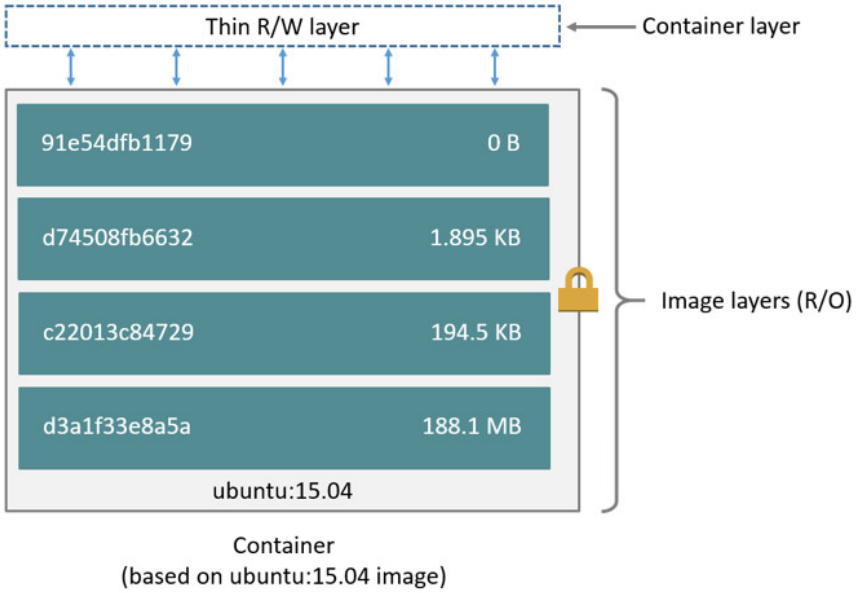

FIGURE A.4 – Couches d'un container Docker

— Une image téléchargée chez une personne A aura un UUID différent que la même image téléchargée chez une personne B. Impossible de s'assurer de l'intégrité de l'image téléchargée en se basant sur l'UUID. En utilisant le hash, on s'assure du même résultat si l'image est identique.

Par conséquent, Docker avance les avantages suivants :

— Intégrité des images téléchargées et envoyées (sur Docker Hub par exemple) ;
— Évite les collisions lors de l'identification des images et des couches ;
— Permet de partager des couches identiques qui proviendraient de *build* différents.

Le dernier point est relativement intéressant. En effet, si deux images de base (Ubuntu et Debian) sont différentes, mais qu'une couche supérieure est identique (par exemple l'ajout d'un même fichier texte) alors cette couche supérieure peut être partagée entre les deux images (puisqu'elle possède le même hash). Ceci peut potentiellement offrir un gain d'espace disque conséquent si plusieurs images partagent plusieurs couches identiques. Ceci n'aurait pas pu être possible avec les UUID, car les deux couches auraient produit deux UUID différents. Un exemple est visible à la figure A.5

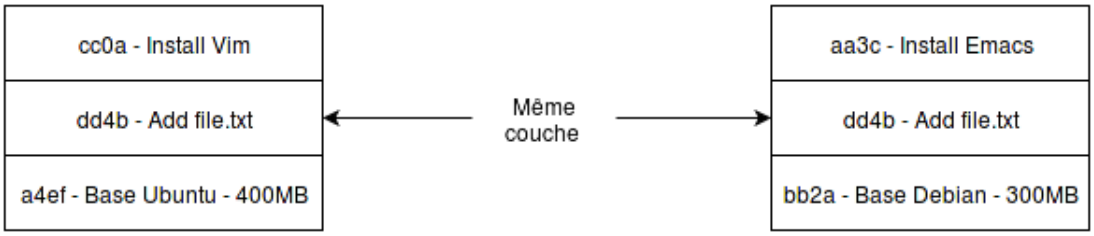

FIGURE A.5 – Couche partagée entre deux images Docker

A.5 Isolation

Docker met en avant le fait que ses containers soient isolés du système hôte. Pour ce faire, Docker utilise des mécanismes fournis par le kernel. On peut citer parmi ces mécanismes les *namespaces* et *cgroups*.

A.5.1 Les namespaces

Les namespaces permettent d'isoler certaines fonctionnalités d'un système d'exploitation utilisant Linux. Comme chroot permet aux processus de voir comme racine un dossier isolé du système et non pas la "vraie" racine, les namespaces isolent certains aspects du système comme les processus, les interfaces réseaux, les points de montage, etc.

Jusqu'à très récemment (docker $<$ 1.10.0), Docker supportait les namespaces suivants[52] :

- PID namespace, chaque conteneur a ses propres id de processus ;
- UTS namespace, pour avoir son propre hostname ;
- IPC namespace, qui permet d'isoler les Communications Inter-Processus ;
- Network namespace, chaque conteneur peut avoir sa propre interface réseau, son IP, ses règles de filtrage.

Docker a désormais ajouté le support d'un nouveau namespace : user namespace. Celui-ci permet à un processus d'avoir les droits root au sein d'un namespace, mais pas en dehors. Avant, Docker lançait les containers en root ce qui pouvait poser des problèmes de sécurité si un processus dans le container venait à en sortir ; il se retrouverait root sur le système hôte. Avec la prise en charge de ce namespace, un container Docker a l'impression d'être root alors qu'il n'est, en réalité, qu'un utilisateur normal sur le système hôte.

A.5.2 cgroups - Control Groups

Cgroups (control groups) est une fonctionnalité du kernel pour limiter, prioriser, isoler et contrôler l'utilisation des ressources (CPU, RAM, utilisation disque, utilisation réseau...). Pour limiter les ressources, cgroups propose de créer un groupe (profil) qui décrit les limitations à respecter. Par exemple, si on crée un groupe appelé "groupe 1" et qu'on exige de lui qu'il n'utilise qu'au maximum 25% de la charge CPU et n'utilise qu'au maximum 100 MB de RAM. Alors, il devient possible de lancer des programmes qui appartiennent à ce groupe et qui respectent les limites fixées.

Lorsqu'on utilise la commande `docker run` de Docker pour lancer un container, Docker peut utiliser cgroups et ainsi limiter les ressources du container [3].

3. Docker - Runtime constraints on resources : `https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources`

B. Script de conversion AARF vers CSV pour Golub

```
import arff
import os
import pandas as pd


def main(input_file, output_folder):
    """
    Original data have been downloaded here ("Leukemia" section):
    http://eps.upo.es/bigs/datasets.html

    Convert Golub arff dataset into N (where N is the numbers of samples) csv
    files.
    CSV files are formatted with 2 columns and M lines (number of features):
        ID_REF  VALUE
        feat1   val1
        feat2   val2
        ...     ...
        featM   valM

    Requirements : Pandas, LIAC-ARFF

    :param input_file: original source file in *.arff format
    :param output_folder: output folder where the csv files will be saved
    :return: nothing
    """
    decoder = arff.ArffDecoder()
    f = open(input_file)
    data = decoder.decode(f, encode_nominal=True)

    labels = [d[0] for d in data["attributes"][:-1]]

    labels_lookup_table = ["ALL", "AML"]

    for sample in range(len(data["data"])):
        class_name = labels_lookup_table[data["data"][sample][-1]]
        sample_filename = "sample_%s_%s.csv" % (sample, class_name)
        sample_filename = output_folder + os.sep + sample_filename
        print(sample_filename)

        write_sample(filename=sample_filename, data=data["data"][sample][:-1],
        labels=labels)


def write_sample(filename, data, labels):
    df = pd.Series(data, index=labels)
    df.to_csv(filename, sep='\t', encoding='utf-8', header=True,
    index_label=["ID_REF", "VALUE"])


if __name__ == '__main__':
    """
    Example of usage
    The original data in *.arff have been saved in a 'data/golub' folder
    """

    # Generate train files
```

```
53      input_file = r'./data/golub99/leukemia_train_38x7129.arff'
54      output_folder = r'./data/golub99/processed/train'
55      main(input_file=input_file, output_folder=output_folder)
56
57      # Generate test files
58      input_file = r'./data/golub99/leukemia_test_34x7129.arff'
59      output_folder = r'./data/golub99/processed/test'
60      main(input_file=input_file, output_folder=output_folder)
```