PGROU

Rapport final de projet

Sujet 6 :
Vérification de propriétés dynamiques de modèles inférés par LFIT à l'aide de la bibliothèque CoLoMoTo

intégration de pyLFIT dans les notebooks de CoLoMoTo

Morgan MAGNIN, Maxime FOLSCHETTE, Tony RIBEIRO

29/03/2021
(version amendée du 30/03/2021)

SOMMAIRE

Remerciements

Introduction

I. Contexte du projet
1) pyLFIT
2) CoLoMoTo
3) Intégration de pyFLIT au sein du consortium CoLoMoTo

II. Structuration des pipelines
1) Mise en place d'un pipeline simple intégrant pyLFIT avec les outils de CoLoMoTo
2) Traitement et analyse de données à l'aide d'un pipeline mettant en jeu pyLFIT
3) Création d'un système de comparaison de modèles

III. Mise en place des pipelines
1) Mise en place d'un pipeline simple intégrant pyLFIT avec les outils de CoLoMoTo
2) Traitement et comparaison de données
3) Vision critique sur le travail réalisé
4) Limitations du projet

Estimation des coûts

Conclusion

Annexes

Remerciements

Nous souhaitons remercier nos trois encadrants Morgan MAGNIN, Maxime FOLSCHETTE et Tony RIBEIRO pour nous avoir proposé ce Projet de Groupe. Nous les remercions également pour leur accompagnement tout au long de ce projet et pour leurs réponses à nos différentes questions.

Introduction

Le sujet 10 de Projet de Groupe a été réalisé par les étudiants Guillaume CHEVRON, Yuri FIGUEIREDO BORRMANN, Fatma HAMILA, Mathieu JUNG-MULLER et Gabin SCHIEFFER, en deuxième année à l'Ecole Centrale de Nantes, suivant l'option Informatique.

Intitulé « Vérification de propriétés dynamiques de modèles inférés par LFIT à l'aide de la bibliothèque CoLoMoTo, intégration de pyLFIT dans les notebooks de CoLoMoTo », ce projet avait pour encadrants et destinataires Morgan MAGNIN, Maxime FOLSCHETTE et Tony RIBEIRO.

I) Contexte du projet

Dans ce projet, nous cherchons à analyser la dynamique de systèmes complexes. Ces systèmes peuvent être de plusieurs nature, comme par exemple des systèmes biologiques. Plus généralement, nous nous intéressons à des systèmes dont l'évolution dynamique permet d'obtenir des données sous forme de séries temporelles. Dans le cas de modèles biologiques, cela peut par exemple correspondre à une expression génétique au fil du temps.

1) pyLFIT[1]

pyLFIT est une bibliothèque python issue de la recherche permettant d'apprendre la dynamique d'un système, en observant uniquement les entrées et sorties de celui-ci. Cela permet de connaître certains mécanismes internes d'un système complexe alors qu'il est normalement impossible d'accéder au fonctionnement réel du système, cela est notamment le cas pour des systèmes biologiques pour lesquels on ignore les mécanismes internes.

Cette bibliothèque s'appuie sur les algorithmes LFIT (Learning From Interpretation Transitions). Ces algorithmes exploitent en entrée un ensemble de transitions, et produisent en sortie un programme logique. Le but des algorithmes de pyLFIT est de produire un programme qui permette de simuler au mieux le système étudié, dans l'objectif d'obtenir la réponse du système pour des cas non observés.

En outre, le fait que l'on puisse modéliser un système par des règles logiques permet de mieux comprendre et analyser le fonctionnement d'un système. En effet, la présentation sous forme de règles logique d'un modèle permet de comprendre certains mécanismes du système complexe sous-jacent. Cela justifie dès lors l'utilisation de pyLFIT dans le domaine de la bio-informatique.

Cadre de pyLFIT. pyLFIT se restreint à l'étude de systèmes discrets, aussi, les entrées et sorties des systèmes étudiés sont des vecteurs de même taille. Chaque composante de ces vecteurs est appelée variable et est une valeur discrète. Les systèmes étudiés sont dynamiques : les données en entrée et en sortie représentent l'état du système, et la sortie obtenue à un instant constitue l'entrée à l'instant suivant.

Règles en sortie de pyLFIT. Les règles en sortie de pyLFIT permettent de déterminer l'état du système à un instant t à partir d'un état à l'instant $t\text{-}1$. On dispose ainsi de règles logiques dont la tête correspond à un prédicat à l'instant t, et dont le corps correspond aux prédicats à l'instant $t\text{-}1$, par exemple :

```
X_t(1) :- X_t-1(0), ~Y_t-1(0)
```

Les prédicats à l'instant t sont nommés targets (cible), tandis que les prédicats à l'instant $t\text{-}1$ sont nommés features. Ces règles logiques permettent donc de déterminer la valeur de chacun des prédicats à l'instant t en fonction de la valeur des prédicats à l'instant $t\text{-}1$. Ces prédicats sont nommés variables, et l'ensemble des valeurs des variables constitue l'état du système. Il est donc possible grâce à l'ensemble des règles de prédire l'état du système à partir de l'état précédent.

Dans l'exemple ci-dessus, à l'instant t la variable X vaudra 1 dès que $X\ vaut\ 0$ et $Y\ ne\ vaut\ pas\ 0$ à l'instant $t\text{-}1$.

1 : cette sous-section est tirée du site web de Tony Ribeiro, http://www.tonyribeiro.fr/research_main.html

Le format d'entrée de l'API de pyLFIT est une liste de liste, que l'on appelle binôme, où chaque élément de cette sous-liste symbolise un état et le suivant en détaillant les valeurs de différentes features par état. Voici un exemple d'entrée pour pyLFIT :

```
data = [ \
    ([0,0,0],[0,0,1]), \
    ([0,0,0],[1,0,0]), \
    ([1,0,0],[0,0,0]), \
    ([0,1,0],[1,0,1]), \
    ([0,0,1],[0,0,1]), \
    ([1,1,0],[1,0,0]), \
    ([1,0,1],[0,1,0]), \
    ([0,1,1],[1,0,1]), \
    ([1,1,1],[1,1,0])]
```

2) CoLoMoTo

Le consortium CoLoMoTo (Consortium for Logical Models and Tools) regroupe des chercheurs dont les travaux de recherche s'axent autour de la modélisation logique. Ces chercheurs ont développé des outils et méthodes permettant de répondre à des problématiques issues de ce domaine. Le but de ce consortium est de fédérer ces acteurs, et de fournir un cadre de travail accessible pour le développement et l'utilisation de ces outils, afin notamment d'améliorer leur interopérabilité, particulièrement en ce qui concerne la comparaison de modèles, de méthodes et d'outils.

L'ensemble de ces outils est disponible au sein d'une image Docker proposée par le consortium. Cette image propose un environnement python incluant ces outils, permettant d'interfacer facilement plusieurs outils entre eux. Cet environnement est proposé sous forme de serveur Jupyter, sur lequel il est possible d'exécuter du code sous forme de fichiers incluant du texte et des illustrations, nommés notebooks. Ces notebooks permettent – par leur simplicité de compréhension – d'exposer des exemples d'utilisation. Ainsi, de tels exemples sont fournis au sein de l'image Docker, ils exposent des utilisations possibles des outils du consortium.

Nous proposons dans ce projet d'interfacer pyLFIT avec plusieurs de ces outils, au sein de notebooks utilisables dans l'environnement proposé par CoLoMoTo. Nous avons pour cela sélectionné un ensemble d'outils du consortium, avec lesquels il est intéressant d'interfacer pyLFIT.

a– PyBoolNet

PyBoolNet est une bibliothèque python permettant de manipuler des réseaux booléens, cette bibliothèque est incluse dans l'environnement CoLoMoTo. Cet outil propose des fonctionnalités d'analyse de réseaux booléens. Ainsi, il pourrait permettre d'effectuer certaines analyses sur les règles logiques produites en sortie de pyLFIT. Cet outil est capable d'interpréter des règles logiques fournies sous forme de chaîne de caractère, une traduction des règles en sortie de pyLFIT est donc nécessaire pour pouvoir faire un usage conjoint de ces deux outils.

b– GINsim

GINsim est un programme permettant de modéliser et simuler des réseaux de régulation génétiques. Cet outil est principalement proposé sous forme d'interface graphique utilisable en

l'état. En outre, une interface de programmation (API) est accessible depuis un environnement python. Cette interface permet d'utiliser certaines fonctionnalités proposées par GINsim au sein de scripts python.

Dans GINsim, les réseaux de régulation génétiques sont représentés sous forme de graphes, nommés graphes logiques d'influence (Logical Regulatory Graph - LRG). Les sommets représentent des gènes, tandis que les arrêtes représentent l'influence d'un gène (sommet) sur un autre. Les interactions mettent en jeu un gène source et un gène cible (target).

GINsim propose en outre une base de données de modèles issus de la littérature scientifique. Cette base de données fait référence dans le domaine de la bio-informatique, et propose des modèles dans des formats utilisables dans GINsim.

c– BoolNet

BoolNet est un module initialement disponible pour le langage R. Il est proposé au sein de CoLoMoTo, et une passerelle permet de faire usage de cet outil dans le notebook CoLoMoTo. Cet outil permet de construire et d'analyser des réseaux booléens. Nous pouvons dès lors en faire usage pour analyser les ensembles de règles logiques produites en sortie de pyLFIT.

3) Intégration de pyLFIT au sein du consortium CoLoMoTo

Nous cherchons à interfacer pyLFIT avec les outils présentés précédemment. Le but de notre projet est de montrer qu'il est possible d'utiliser facilement pyLFIT au sein de notebooks utilisables dans l'environnement CoLoMoTo.

Pour cela, nous proposons de développer plusieurs processus sous forme de notebooks Jupyter, chacun intégrant différents outils de CoLoMoTo conjointement avec pyLFIT. Le but de chacun de ces notebooks est de montrer qu'à partir d'une entrée il est possible de chaîner plusieurs outils de CoLoMoTo avec pyLFIT pour produire une sortie utilisable dans un ou plusieurs outils de CoLoMoTo, ceci à des fins d'analyse notamment. Dans ce rapport, ces processus sont nommés pipelines.

II) Structuration des pipelines

Comme décrit précédemment, il s'agit dans ce projet d'intégrer les algorithmes pyLFIT dans les notebooks de CoLoMoTo, afin de permettre aux chercheurs d'utiliser les fonctionnalités intéressantes proposées par pyLFIT. Il est donc nécessaire de mettre en place des interfaces au niveau des entrées et sorties, pour que le chercheur utilisant pyLFIT reste dans un environnement qui lui est familier : des pipelines permettront au chercheur d'utiliser le format souhaité en entrée et en sortie, sans qu'il n'ait à interagir directement avec pyLFIT.

Il s'agit également de mettre en place un système d'analyse des données. La modélisation de certaines formes de bruit dans les données en entrées entraîne des divergences au niveau de la sortie qu'il s'agira d'étudier.

Pour ce faire, nous avons isolé trois parties principales du projet que nous détaillons ci-dessous.

1) Mise en place d'un pipeline simple intégrant pyLFIT avec les outils de CoLoMoTo

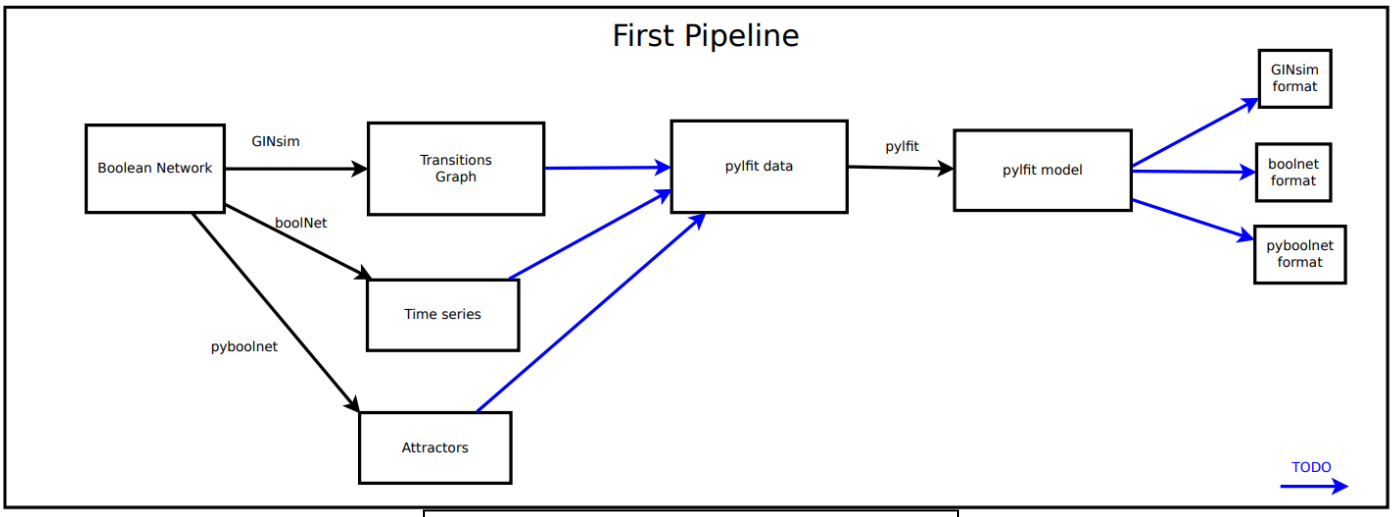

Figure 1 : premier pipeline à réaliser

Cette première partie vise à interfacer pyLFIT avec d'autres outils. Pour cela, il est nécessaire d'effectuer des traductions entre différents formats de données. Cette traduction aura lieu depuis ou vers un des formats de données utilisés dans pyLFIT. Les formats de ce projet sont pyBoolNet, BoolNet et GINsim.

Formats à traduire, en entrée de pyLFIT :
– Graphe de transitions (GINsim)
– Séries temporelles (BoolNet)
– Attracteurs (pyBoolNet)

Traduction de programmes logiques en sortie de pyLFIT vers les formats :
– GINsim
– BoolNet
– pyBoolNet

Ces méthodes de traduction seront implémentées en python, et devront être intégrées au système existant (pyLFIT et CoLoMoTo).

2) Traitement et analyse de données à l'aide d'un pipeline mettant en jeu pyLFIT

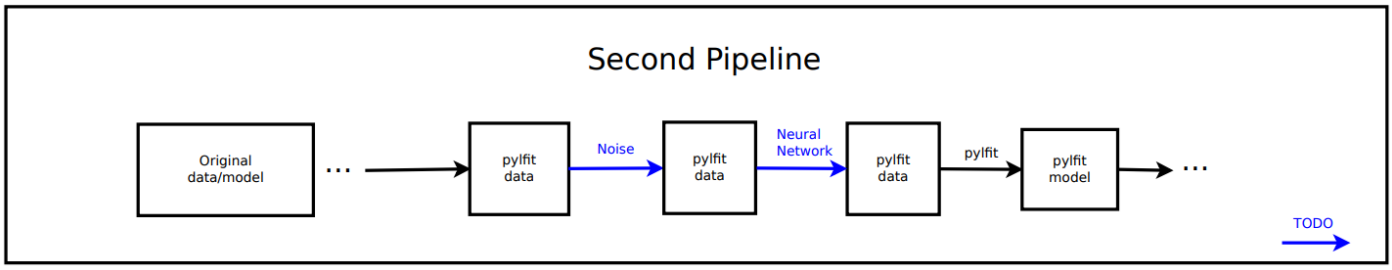

Figure 2 : deuxième pipeline à réaliser

Cette deuxième partie vise à altérer les données en entrée de pyLFIT, afin d'observer l'impact sur les modèles en sortie de pyLFIT. Cette altération se fera selon le schéma suivant, à partir de données au format pyLFIT :
1– Ajout de bruit, selon plusieurs méthodes : suppression/ajout de transitions, suppression d'une variable
2– Suppression du bruit à l'aide d'un réseau de neurones - données nettoyées
3– Exécution de pyLFIT sur les données nettoyées

3) Création d'un système de comparaison de modèles

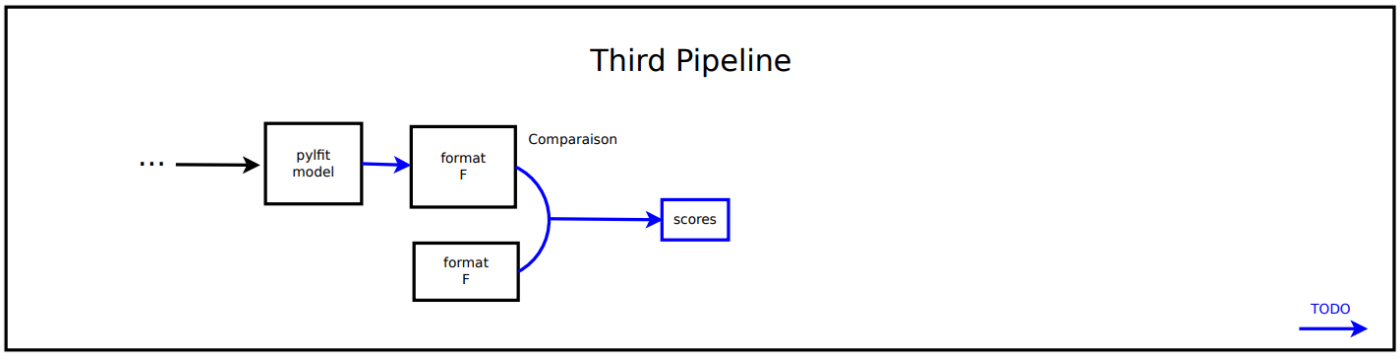

Figure 3 : troisième pipeline à réaliser

Cette dernière partie vise à comparer différents modèles stockés dans un même format (pyLFIT, pyBoolNet, BoolNet ou GINsim).

Nous développerons des méthodes permettant de comparer deux modèles. Ces méthodes devront, à partir de deux modèles distincts, fournir un ou plusieurs scores. Ce score devra permettre d'évaluer comparativement la qualité des données de sortie en fonction des traitements effectués dans le pipeline, en particulier :
– selon la méthode d'ajout de bruit, comparativement avec les données non modifiées
– selon l'algorithme de pyLFIT choisi.

Schéma général explicatif :

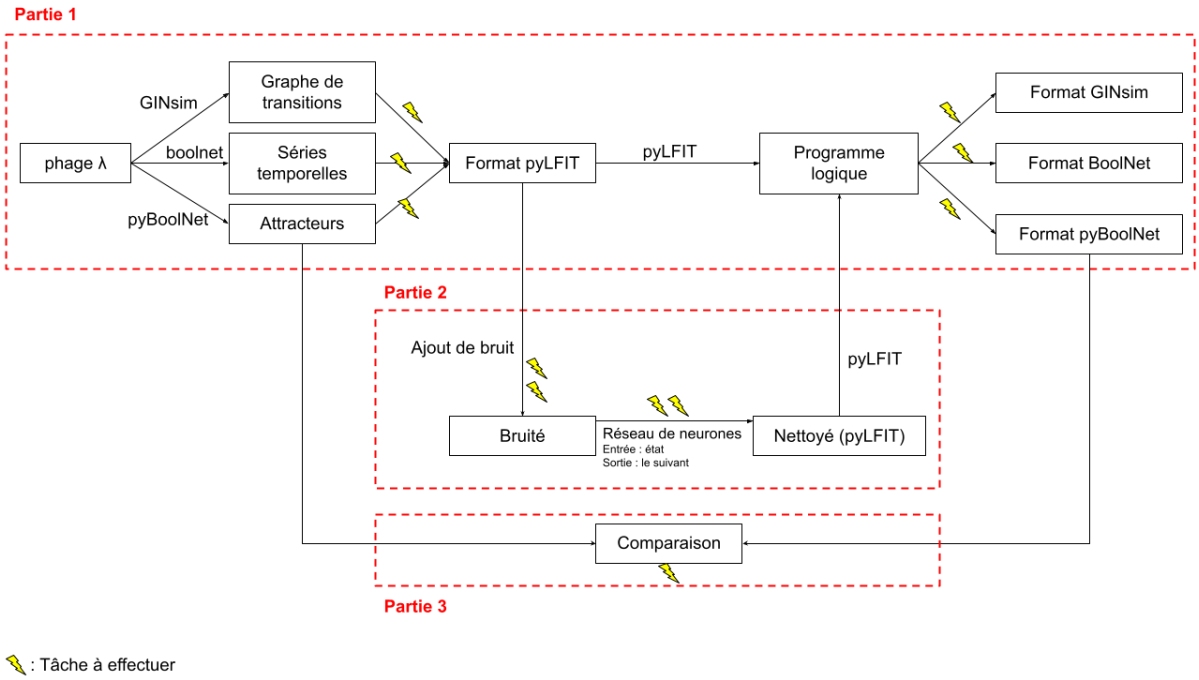

Figure 4 : schéma général du cahier des charges

NB : Ce schéma a été repris tel qu'utilisé lors de la rédaction du cahier des charges. Il a subi des modifications par la suite, ainsi qu'une refonte de la présentation, tel que cela sera présenté dans la partie suivante.

III) Mise en place des pipelines

Retour sur le cahier des charges :

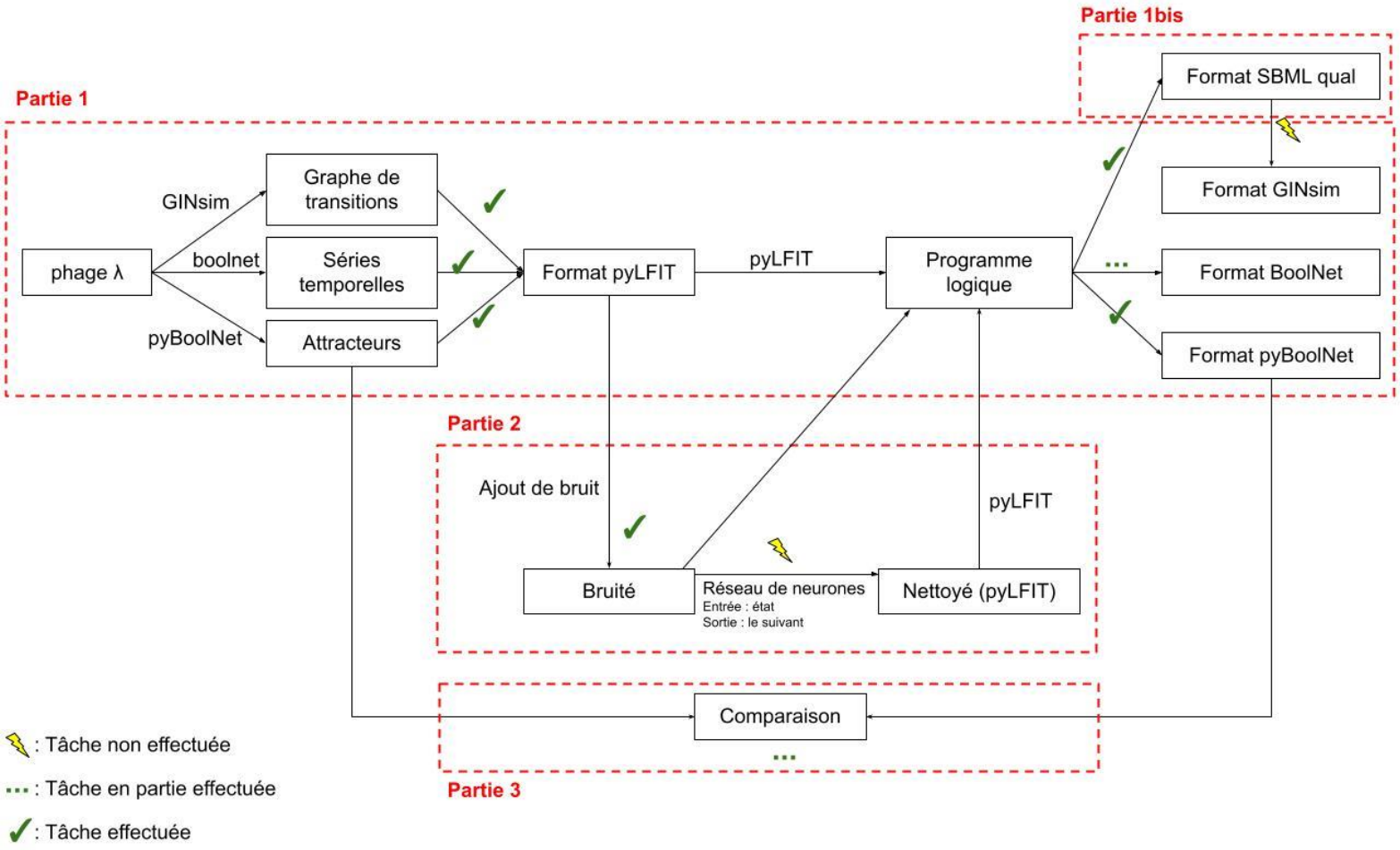

Figure 5 : schéma reprenant les attentes du cahier des charges

Ce schéma reprend celui du cahier des charges et présente notre avancement dans les différentes tâches à la fin du projet de groupe.

Schéma d'explication général :

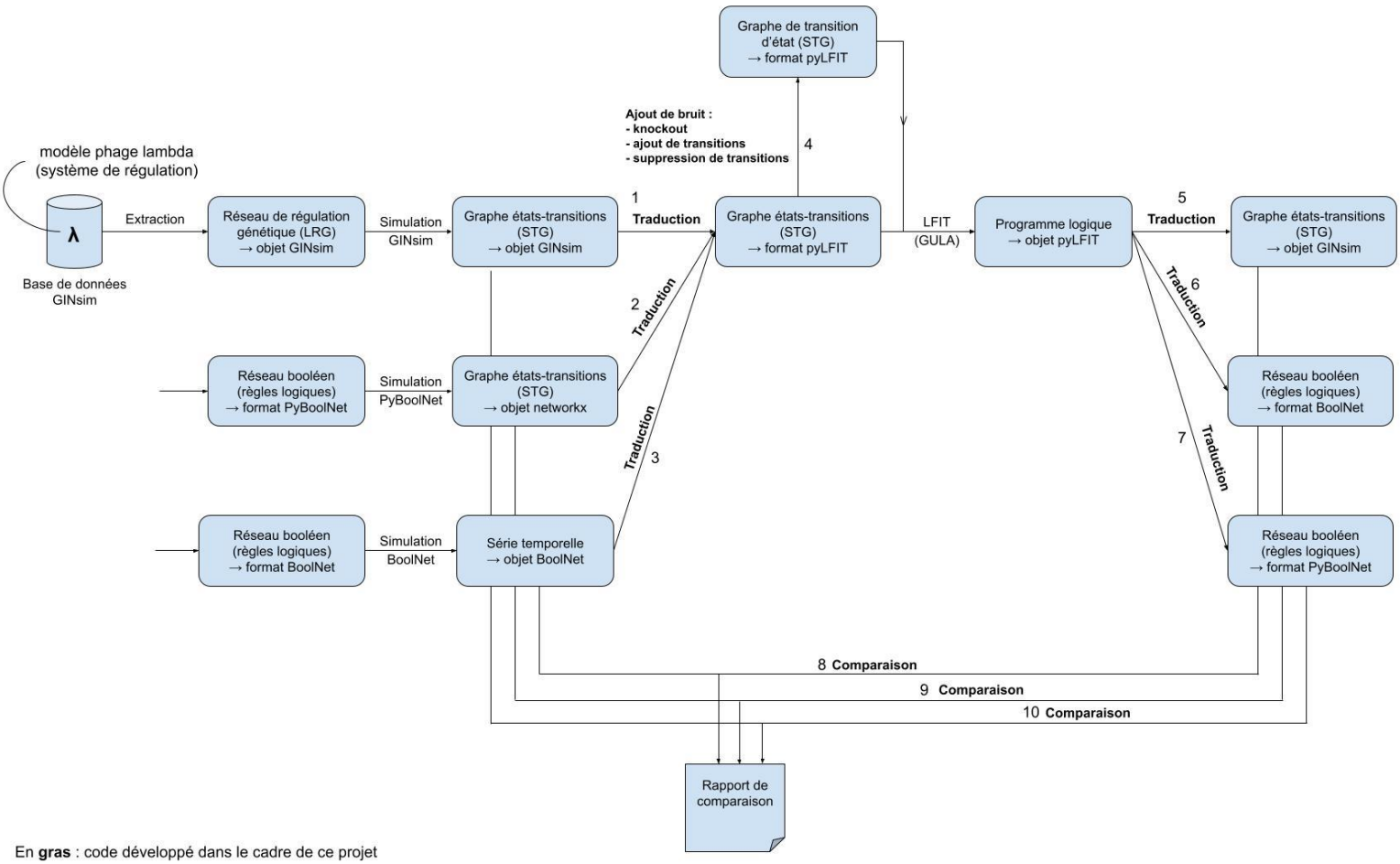

Figure 6 : schéma général reprenant le travail réalisé

Ce schéma a pour vocation de faire écho à celui présenté juste au-dessus. Il a été modifié afin de mettre mieux en lumière les éléments réalisés. Il regroupe également mieux les informations par blocs de similarité.

Le détail des différentes parties que nous avons traitées et implémentées est fourni ci-dessous. Elles seront à chaque fois référencées par leur numéro correspondant à un arc du graphe ci-dessus.

1) Mise en place d'un pipeline simple intégrant pyLFIT avec les outils de CoLoMoTo

a– Les formats en entrée de pyLFIT

Utilisation de la bibliothèque GINsim (arc 1)

GINsim propose de manipuler deux types de graphes : les graphes de régulation (LRG - Logical Regulatory Graph) - qui modélisent des réseaux de régulation - et les graphes états-transitions (STG - State Transition Graph) qui modélisent le comportement dynamique de ceux-ci [1].

Dans la mesure où pyLFIT permet d'analyser le comportement dynamique d'un système, nous cherchons à placer un graphe états-transitions fourni par GINsim en entrée de pyLFIT.

GINsim propose une base de données de modèles issus de la littérature scientifique. Nous proposons d'extraire un modèle simple de cette base de données sous forme de fichier importable dans GINsim, afin de l'utiliser en entrée de pyLFIT, pour tester notre pipeline. Pour cela, nous utilisons le modèle phage lambda, qui présente quatre variables [2].

Une fois ce modèle chargé dans GINsim sous forme de graphe de régulation (LRG), nous utilisons les fonctionnalités de simulation exposées par l'API GINsim pour obtenir un graphe états-transitions (STG). Nous convertissons alors directement le graphe obtenu sous forme d'objet python en un format utilisable en entrée de pyLFIT. Ainsi, il est possible d'exécuter les algorithmes de pyLFIT sur de telles données.

Références :

[1] http://ginsim.org/node/9#Introduction
[2] http://ginsim.org/taxonomy/term/29

Utilisation de la bibliothèque PyBoolNet (arc 2)

Dans cette partie du pipeline, nous nous intéressons à la partie qui concerne l'utilisation de la bibliothèque PyBoolNet de python. Le format d'entrée de ce pipeline est une chaîne de caractères qui résume les formules booléennes implémentées dans le réseau booléen sur lequel on va travailler. Il s'écrit de cette façon :

```
"""
v1, !v1
v2, 1
v3, v2 & (!v1 | v3)
"""
```

Pour faire le parallèle avec le format des règles que l'on manipule dans pyLFIT, ce format en est très proche si l'on considère les trois variables p, q et r utilisées en pyLFIT, cette entrée PyBoolNet peut être traduite de cette façon :

```
p_t(1) : -p_t-1(0).
q_t(1) : -.
r_t(1) : - q_t-1(1), p-t-1(0).
r_t(1) : - q_t-1(1), r_t-1(1).
```

Les règles pyLFIT sont définies de cette façon avec la syntaxe de - avant chaque élément de droite de manière arbitraire, ce - n'a pas de sens mathématique. De plus, dans les règles de pyLFIT on peut exprimer le temps qui passe de t-1 à t selon la target ou la feature. Nous avions dans le processus de traduction d'abord penser à établir un format de formules booléennes universel dont on voulait ensuite adapter la forme selon la librairie utilisée. Mais il s'avère que l'on peut simplement garder le format accepté par PyBoolNet avec les v1, v2 et v3.

La première étape de conversion consiste à générer le graphe des transitions de ce réseau booléen pour pouvoir en extraire les attracteurs sous forme de liste. Ensuite il faut modifier cette liste d'états pour qu'elle soit conforme à l'entrée de pyLFIT. Ce formatage est alors la dernière étape de la conversion et on finit par obtenir la liste de listes souhaitée qui va arriver en entrée de pyLFIT.

Utilisation de la bibliothèque BoolNet (arc 3)

On s'intéresse ici à la dernière partie du pipeline qui concerne la dernière traduction à réaliser, celle concernant l'utilisation de la bibliothèque BoolNet. Avec cette branche, nous pouvons gérer deux types d'entrées, une entrée de réseau booléen sous le format SBML qual où une entrée de réseau booléen manuelle comme pour PyBoolNet. Dans le cas de la partie saisie manuelle, nous avons, pour notre étude, utilisé la fonction BoolNet de génération d'un réseau booléen aléatoire où l'on contrôle le nombre de gènes – donc de variables – et le nombre de transitions que l'on veut implémenter.

Dans les deux cas, en deuxième étape, nous disposons alors d'une matrice qui a le format d'un objet R, à partir de laquelle nous allons générer des séries temporelles à l'aide de la fonction generateTimeSeries, où l'on décide du nombre de séries que l'on veut générer et du nombre de mesures que l'on veut exécuter.

Après cette étape, nous avons les données finales prêtes pour leur migration vers pyLFIT, mais il faut encore changer leur formatage pour que ce dernier corresponde au format d'entrée de pyLFIT. Pour cela, il faut d'abord convertir cet objet R en objet python, et ensuite

dupliquer certains états afin d'obtenir cette liste de couples d'états que l'on souhaite pour l'entrée de pyLFIT.

Dans le cas d'un fichier SBML qual en entrée, on utilise la fonction intégrée au package BoolNet à savoir loadSBML qui permet de convertir le fichier SBML en réseau booléen conforme au format R demandé par les fonctions BoolNet.

b– Les formats en sortie de pyLFIT

Utilisation de la bibliothèque PyBoolNet (arc 7)

Pour le processus inverse en utilisant la bibliothèque PyBoolNet, on va extraire la partie du model.summary() généré par pyLFIT qui nous intéresse. Ici, on précise, même si cela n'a pas un grand impact, que nous avons fait la conversion en utilisant l'algorithme GULA de pyLFIT. Cette conversion fonctionne cependant aussi avec les autres algorithmes, étant donné que la sortie reste sous la même forme.

Tout d'abord, nous avons extrait les règles générées par pyLFIT sous forme de chaîne de caractères qui prend en compte toutes les règles obtenues. Celles-ci sont ensuite extraites grâce à la commande de récupération de sous-chaîne de model.logic_form(), où l'on prend toute la sous-chaîne après le séparateur qui est un retour à la ligne. Voici un exemple de sortie obtenue :

```
FEATURE p_t-1 0 1
FEATURE q_t-1 0 1
FEATURE r_t-1 0 1
TARGET p_t 0 1
TARGET q_t 0 1
TARGET r_t 0 1

p_t(0) :- q_t-1(0).
p_t(1) :- q_t-1(1).
p_t(1) :- p_t-1(0), r_t-1(0).
q_t(0) :- p_t-1(0).
q_t(0) :- r_t-1(0).
q_t(1) :- p_t-1(1), r_t-1(1).
r_t(0) :- p_t-1(1).
r_t(0) :- q_t-1(0), r_t-1(0).
r_t(1) :- p_t-1(0).
```

On veut par exemple extraire la partie commençant après p_t(0).

On va alors créer des chunks de cette chaîne de caractères qui sont les différentes lignes où il y a une règle par ligne. Puis chacun de ces chunks, seront stockées dans un tableau de chaînes de caractères. Ensuite, en utilisant la fonction replace, on formate les noms pour adopter une dénomination plus générale et adaptée, par exemple en écrivant les targets avec des t et les features avec des f. On obtient alors un tableau de règles ainsi créé :

```
['t1(0) = f2(0)', 't1(1) = f2(1)', 't1(1) = f1(0), f3(0)', 't2(0) = f1(0)', 't2(0) = f3(0)', 't2(1) = f1(1), f3(1)', 't3(0) = f1(1)', 't3(0) = f2(0), f3(0)', 't3(1) = f1(0)']
```

Enfin, on filtre les règles qui ont des targets dont l'argument est 0 comme t1(0) par exemple : en effet, dans un programme logique, il n'est pas nécessaire de les implémenter car elles peuvent être déduites – par exemple en faisant le complémentaire de t1(1). Il n'y a plus qu'à regrouper les règles par target en ajoutant les « ou inclusif », ce qui nous donne dans notre cas trois chaînes qui vont décrire les règles pour nos trois targets différents. Pour finir, on reformate encore afin de coller au formatage programme logique PyBoolnet avec les « et » : '&' et les « ou » : '|' entre autres. On formate les règles à partir de cette fonction :

```
#Format to the final form
str1 = "t1 = "
str2 = "t2 = "
str3 = "t3 = "
for s in strarray :
    if s[3] == "1" :
        if s[0:2] == "t1" :
            str1 = str1 + " & " + delim(substring_after(s, "="), "")
        if s[0:2] == "t2" :
            str2 = str2 + " & " + delim(substring_after(s, "="), "")
        if s[0:2] == "t3" :
            str3 = str3 + " & " + delim(substring_after(s, "="), "")
str1 = str1.replace("=  &", "=")
str2 = str2.replace("=  &", "=")
str3 = str3.replace("=  &", "=")
```

Ce code produit le format que l'on souhaite prêt à être implémenter suivant :

```
v1 , v2 & !v1 | !v3
v2 , v1 | v3
v3 , !v1
```

Ici, nous nous sommes concentrés sur trois chaînes de caractères car il n'y avait que trois targets mais nous pouvons modifier légèrement le code en implémentant des boucles et des tableaux de chaînes de caractères afin de rendre cette partie dynamique et lui permettre de gérer la traduction peu importe le nombre de targets.

Utilisation de la bibliothèque BoolNet (arc 6)

Une fois que l'on a réussi à transformer les règles en sortie de pyLFIT en formules booléennes dont on a standardisé le format, cela rend plus simple la conversion en entrée pour les utiliser avec les autres librairies comme BoolNet.

Le format d'entrée de BoolNet étant aussi sous forme d'un réseau booléen comme PyBoolNet, on peut convertir les formules dans ce format qui est une R-matrice avec les colonnes gènes et les transitions, un peu comme un graphe de transition.

Une fois converti en matrice R, on peut générer les séries temporelles dessus de la même façon que pour le chemin BoolNet vers l'entrée de pyLFIT et ensuite on peut comparer les 2 séries temporelles obtenues, celle en entrée du pipeline et celle que l'on vient d'obtenir.

Mise au format SBML qual

Pour exporter en format SBML qual, il faut réaliser deux étapes de plus que le processus réalisé à la partie précédente. Tout d'abord, on fait appel à la fonction reconstructNetwork qui va reconstruire le réseau à partir des séries temporelles obtenues et les mettre sous une forme convenable pour les exporter par la suite.

La seconde étape consiste à utiliser la fonction toSBML() qui va créer un fichier contenant le réseau booléen sous le format SBML qual que l'on pourra comparer avec le fichier SBML qual pris en entrée.

2) Traitement et comparaison de données

a– Ajout de bruit aux données en entrée de pyLFIT (arc 4)

L'ajout de bruit à un système est fait pour simuler des conditions réelles. Autrement dit, lorsqu'on a des signaux ou des données collectés lors d'expériences scientifiques ou dans des situations quotidiennes, on collecte également des signaux indésirables et aléatoires. Afin de se rapprocher des signaux réels ou d'avoir un système qui généralise mieux les données d'entrées, on ajoute du bruit.

Dans le cadre de notre projet, le bruit est ajouté aux données en entrée de pyLFIT. Tout d'abord, nous avons ajouté trois types de bruit différents : supprimer une transition, ajouter une nouvelle transition, remplacer les zéros par un et vice versa.

L'utilisateur peut contrôler le taux de bruit à l'aide d'une fonction d'ajout de bruit, ainsi que le type du bruit à ajouter. Nous avons implémenté deux types de bruits différents : le bruit ajouté selon une distribution uniforme, et celui selon une distribution gaussienne.

Concernant la première méthode, on utilise une fonction de distribution uniforme (0,1) et un threshold qui indiquera si le bruit doit être ajouté ou non. Si la fonction uniforme renvoie une valeur inférieure au threshold, alors on ajoute le bruit. De cette manière, un threshold égal à zéro indique un taux de bruit égal à zéro et un threshold égal à un indique un taux de bruit égal à un (le signal n'est que du bruit).

Pour illustrer le fonctionnement de la fonction d'ajout de bruit uniforme, on considère l'entrée suivante :

```
data = [(["0","0","0"],["0","0","1"]),
            (["1","0","0"],["0","0","0"]),
            (["0","1","0"],["1","0","1"]),
            (["0","0","1"],["0","0","1"]),
            (["1","1","0"],["1","0","0"]),
            (["1","0","1"],["0","1","0"]),
            (["0","1","1"],["1","0","1"]),
            (["1","1","1"],["1","1","0"])]
```

Pour définir les différents bruits et leurs thresholds, on utilise la fonction *creat_noise* comme ci-dessous, en passant des arguments Vrai ou Faux pour chaque bruit et des chiffres pour les threshold. Par défaut, on utilise tous les bruits et les thresholds sont définis à 0.1.

creat_noise (**data**, knockout = **True**, knockout_threshold=**0.1**, remove_transitions = **True**, rvtr_threshold = **0.1**, add_transitions = **True**, addtr_threshold = **0.1**)

En utilisant l'entrée au-dessus comme exemple, on obtient le résultat suivant :

```
[(['1', '0', '0'], ['1', '0', '0']),
 (['0', '1', '0'], ['1', '0', '1']),
 (['0', '0', '1'], ['0', '0', '1']),
 (['1', '1', '0'], ['1', '0', '0']),
 (['1', '0', '1'], ['0', '1', '0']),
 (['0', '1', '1'], ['1', '0', '1']),
 (['1', '0', '1'], ['1', '1', '0']),
 (['0', '1', '0'], ['1', '0', '1'])]
```

On remarque que plusieurs états ont été touchés par le bruit *knockout*, la première transition a été supprimée et la dernière transition a été ajoutée.

Concernant la seconde méthode, on part du même principe que celui de la distribution uniforme (threshold et un seuil à partir duquel on peut modifier notre entrée). La particularité consiste dans le fait qu'on génère aléatoirement, et d'une façon gaussienne, un signal bruité qu'on additionne à notre donnée en entrée. Suivant la comparaison de chaque valeur de ce nouveau signal à notre threshold_knockout, on procède ou pas à la modification des valeurs des données en entrée.

Pour un exemple concernant le bruit gaussien, on considère l'entrée suivante :

```
data = [ \
    (["0","0","0"],["0","0","1"]), \
    #(["0","0","0"],["1","0","0"]), \
    (["1","0","0"],["0","0","0"]), \
    (["0","1","0"],["1","0","1"]), \
    (["0","0","1"],["0","0","1"]), \
    (["1","1","0"],["1","0","0"]), \
    (["1","0","1"],["0","1","0"]), \
    (["0","1","1"],["1","0","1"]), \
    (["1","1","1"],["1","1","0"])]
```

On cherche à modifier le contenu de cette donnée en se basant sur un modèle généré aléatoirement selon une distribution normale. Dans l'input de la fonction, on dispose d'un threshold réglable à True ou à False selon lequel on induit ou pas le bruit, ainsi que d'un knokout_threshold qui servira de seuil.

D'autre part, après conversion de notre data en une matrice d'entiers plus facile à manier, on génère une matrice aléatoire d'une façon gaussienne qui constitue le bruit (les paramètres constituant le centre et la largeur de la distribution peuvent être modifiés). On additionne ce bruit à notre signal original.

```
            original = np.array(data, int) #converts our data to a matrix of int
            noise = np.random.normal(0.5, .5, original.shape) #generates a matrix of the same shape than our original
data containing normally-distributed noise
            new_signal = original + noise #adds the noise to our original data
```

Finalement, on compare chaque valeur de ce signal bruité à knokout_threshold (passé en input) et selon le résultat de la comparaison, on procède ou pas à la modification des valeurs de notre data :

```
if math.ceil(math.fabs(new_signal[line][state][value])) > knockout_threshold :
    data[line][state][value] = "1"
else :
    data[line][state][value] = "0"
```

Dans les deux fonctions d'ajout du bruit, on a implémenté deux fonctions pour la suppression et l'ajout aléatoire de transitions de l'input de PyLFIT. En effet, suivant un nombre saisi en entrée : addtr_threshold (respectivement rvtr_threshold), on génère aléatoirement le nombre de transitions à ajouter (respectivement à supprimer) puis on génère avec le module numpy les transitions à ajouter (respectivement à supprimer) aléatoirement. Finalement, elles seront soit retirées soit ajoutées à notre donnée d'entrée.

On va alors dans cet exemple avoir comme sortie :

```
Out[10]: [(['1', '0', '1'], ['1', '1', '0']),
          (['1', '1', '1'], ['1', '1', '1']),
          (['0', '1', '1'], ['1', '1', '1']),
          (['1', '1', '1'], ['1', '0', '1']),
          (['1', '1', '1'], ['1', '1', '1']),
          (['1', '1', '1'], ['1', '0', '1']),
          (['0', '1', '1'], ['1', '1', '1'])]
```

Dans les deux fonctions d'ajout du bruit, on a implémenté deux fonctions pour la suppression et l'ajout aléatoire de transitions de l'input de PyLFIT. En effet, suivant un nombre saisi en entrée : addtr_threshold (respectivement rvtr_threshold), on génère aléatoirement le nombre de transitions à ajouter (respectivement à supprimer) puis on génère avec le module numpy les transitions à ajouter (respectivement à supprimer) aléatoirement. Finalement, elles seront soit retirées soit ajoutées à notre donnée d'entrée.

b– Mise en place d'un réseau de neurones pour corriger le bruit

Pour des raisons de manque de temps, et au vu d'importantes difficultés rencontrées lors de la mise en place du réseau de neurones et son application à notre situation, nous avons abandonné cette partie afin de nous concentrer en priorité sur les éléments de comparaison de modèles.

c– Mise en place du modèle de comparaison (arc 10)

Nous avons mis en place un système de comparaison des données pour PyBoolNet. Ce système prend en entrée deux ensembles de règles logiques à comparer, et fournit en sortie trois métriques de comparaison, décrites ci-après. Cela permet de comparer les données produites en sortie de pyLFIT, une fois celles-ci converties au format PyBoolNet.

Les règles logiques fournies en sortie de pyLFIT et traduites dans le format PyBoolNet peuvent être représentées par un diagramme états-transitions (STG), cela à l'aide d'outils fournis par PyBoolNet. Ainsi, le problème initial de comparaison de deux ensembles de règles logiques est transposable en un problème de comparaison de deux graphes orientés. Il existe alors des algorithmes pour résoudre ce problème.

Nous proposons trois métriques pour évaluer la différence entre deux ensembles de règles logiques. D'abord, la modification des règles logiques se reflète directement dans le graphe états-transitions sous forme d'ajouts ou de suppressions d'états et de transitions. Nous proposons donc de compter le nombre d'ajouts et de suppressions pour les états et les transitions. Cela fournit deux métriques qui permettent de comparer deux ensembles de règles, obtenus en sortie de pyLFIT.

En outre, nous proposons de comparer deux ensembles de règles en vérifiant les propriétés d'isomorphisme des deux graphes les représentant. Dans un premier temps nous évaluons l'isomorphisme des deux graphes complets, puis nous déterminons si un des deux graphes est inclus dans l'autre sous forme d'un graphe isomorphe. Ces deux comparaisons sont réalisées à l'aide d'algorithmes, disponibles dans la bibliothèque networkx qui est utilisée par PyBoolNet pour représenter les graphes états-transitions.

Un notebook présentant un exemple de ces comparaisons est disponible dans le dépôt du projet. Nous ne fournissons pas d'exemple dans ce rapport, point pour lequel nous vous prions de nous excuser. Nous comptons cependant le réaliser pour la soutenance.

d– Objectifs de la comparaison

Le système de comparaison présenté précédemment est un outil d'évaluation des pipelines que nous avons développé. Cette évaluation est composée de deux parties.

D'un part, elle doit permettre de vérifier la correction du code, pour cela, il s'agit de comparer les données en entrée avec celles produites en sortie de pyLFIT. En absence de modification, elles doivent être strictement exactes.

D'autre part, la comparaison permet d'évaluer l'effet de l'ajout de bruit dans les données placées en entrée de pyLFIT, et de mesurer les différences de comportement des algorithmes de pyLFIT en fonction du type de bruit utilisé.

3) Vision critique sur le travail réalisé

Nous n'avons pas réalisé ce projet dans son intégralité, ce qui constitue bien sûr un point important pour lequel nous demandons à nos encadrants de nous excuser.

Nous avons eu du mal tout au long du projet à bien saisir les attentes et les tâches à réaliser. Concernant par exemple la partie en sortie de pyLFIT vers les différents formats rattachés aux trois bibliothèques utilisées, nous avions compris que la sortie à traduire résidait dans les prédictions d'états situés en sortie de l'API pyLFIT fournie par Tony, alors qu'il était en fait demandé de traduire les règles logiques.

Cela nous a beaucoup ralenti sur certains points : nous avons passé un volume horaire plus important à chercher des informations et comprendre les concepts qu'à effectivement implémenter le code en question. En particulier, nous n'avons pas été suffisamment capables de nous répartir les tâches de manière efficace, rendant notamment le début du projet assez confus : plusieurs personnes travaillaient sur la même chose sans pour autant mettre leurs résultats en commun, ce qui nous a fait perdre beaucoup de temps.

Nous avions à l'origine distingué trois rôles différents au sein du groupe : un chef de projet, un responsable de la communication et une personne chargée de la rédaction des documents. Cependant, il aurait sans doute été plus efficace de combiner les rôles communication/rédaction, car cela a surtout engendré des délais supplémentaires et de la communication superflue, par exemple entre la fin de la rédaction d'un rapport d'avancement et son envoi effectif.

Nous n'avons par ailleurs pas communiqué de manière optimale entre nous : nous n'avons pas organisé assez de réunions d'avancement entre nous. Par ailleurs, il était presque toujours impossible d'avoir l'ensemble des étudiants connectés à une réunion interne, ce qui a constitué un frein notable…

Nous avons réussi à rectifier le tir dans la deuxième moitié du projet et à ensuite avancer de manière bien plus significative, mais le temps perdu était déjà considérable.

4) Limitations du projet

Une étape supplémentaire à mettre en place pour compléter ce projet serait d'ajouter un réseau de neurones pour supprimer de bruit dans les données d'entrée de pyLFIT. Cette suppression peut corriger les signaux collectés en conditions réelles, par conséquent, cela peut augmenter les performances de certains modèles.

Il serait ensuite judicieux d'appliquer une comparaison de données plus fournie et complète, en élargissant la gamme de métriques utilisées et en l'appliquant à l'ensemble des formats d'entrée et de sortie. Ces métriques de comparaison doivent pouvoir rendre compte de la perte d'information, tout autant que de l'ajout d'informations parasites.

Finalement, une fois les conversions d'entrées et sorties et l'analyse des résultats terminées, on peut intégrer le modèle pyLFIT dans le docker de CoLoMoTo. Cette intégration peut aider différents groupes de recherches dans la modélisation logique.

Estimation des coûts

Nous avons passé sur ce projet un total de 215 heures.

Si l'on considère une consommation électrique approximative de 80Wh pour nos PC (estimation que l'on base en partie sur le fait que la majorité des machines utilisées étaient des ordinateurs de type portables et non fixes), on obtient alors ensuite une consommation totale de 16 kWh. En prenant un coût au kWh de 0.1765€, on obtient donc un coût total lié à la consommation électrique de quelques euros seulement, que nous avons choisi de négliger.

En outre, les outils et logiciels que nous avons été amenés à utiliser pour ce projet n'ont engendré aucun surcoût supplémentaire : discord et les différents services de messagerie sont gratuits d'utilisation, de même que GitHub, les bibliothèques Python utilisées, etc. Nous avons utilisé Zoom pour ce projet, en utilisant la licence payante fournie par Centrale Nantes. Cependant, comme il s'agit d'un projet de recherche public dont les codes sont de toute façon open source et qui ne contient pas de données confidentielles, il aurait été possible d'utiliser à la place un autre outil gratuit du même style.

Il faudrait ajouter à cela un coût correspondant à l'usure de nos machines liée à leur utilisation pour ce projet. Au vu de la quantité de temps passée devant nos écrans en Option Informatique et pour des activités personnelles, et en particulier lors de la crise sanitaire, l'usure liée au PGROU nous a paru négligeable.

Si l'on considère que cette mission est confiée dans le cadre d'un stage d'une durée de plus de deux mois, en prenant donc 3,9€/h comme montant de rémunération, cela fait donc une gratification de 838,50€, c'est-à-dire un coût effectif pour un éventuel employeur (supposé public dans le cadre de ce projet) égal à 838,50€. Il n'y a pas de cotisation supplémentaire pour les stagiaires payés au minimum légal, d'après ce que nous avons compris.

Si l'on suppose qu'il s'agit d'une équipe de développeurs débutants en entreprise, pour un salaire estimé à 10€ de l'heure, il faut encore ajouter les charges payées par l'employeur liées au paiement du salaire. D'après nos recherches, elles s'élèvent à environ 42% du salaire brut, ce qui fait un coût pour l'employeur approximativement égal à 14,20€ par heure. Cela donne donc un coût total du projet de 3053€. On constate une très nette différence par rapport à l'emploi d'une équipe de stagiaires.

Au vu des difficultés de communication évoquées plus haut dans ce rapport, il aurait sans doute été plus efficace, et donc plus rentable, de confier ce projet à une ou deux personnes travaillant dessus sur un volume horaire hebdomadaire plus élevé que ce que nous avons fait individuellement.

Il serait également intéressant de se pencher sur d'éventuels coûts écologiques induits par ce projet.

En effet, les réunions Zoom à distance ont un plus fort impact environnemental que les réunions en présentiel. Il est probable que ces réunions se seraient tenues sur Zoom avec les encadrants, même sans la situation sanitaire exceptionnelle, du fait de l'éloignement géographique (Nantes, Saumur, Lille). Des trajets en train Saumur-Nantes et Lille-Nantes auraient cependant été sans doute nécessaires pour que les intervenants puissent assister à la soutenance, dans le cas d'une situation sanitaire normale où elle aurait été en présentiel.

Par ailleurs, l'utilisation de machines virtuelles pour ceux d'entre nous qui étaient sur Windows est là encore un élément à prendre en compte.

Conclusion

La partie gauche du pipeline concernant les différents formats à traduire en entrée de pyLFIT a été assez bien réalisée, notamment grâce au support de Tony. Cela a aussi été intéressant à mettre en place, en particulier avec une phase d'adaptation et de recherche d'outils sur une bibliothèque précise, puis une phase d'implémentation de ces fonctions pour avoir le résultat souhaité.

La partie traduction en sortie de pyLFIT vers formule logique a été aussi bien réalisée car elle était assez guidée, on savait précisément vers quelle forme on devait aller et comment traiter les règles qui avaient un format particulier. De plus c'est la partie où il y a eu pas mal d'échanges pour être sûr que nous allions dans la bonne direction sans empiéter les uns sur les autres. Le reste des formats en sortie n'as pas pu être finalisé.

Nous avons réalisé une mise en place de bruit pour les données en entrée, en utilisant deux méthodes bien distinctes permettant de doser le bruit s'appliquant aux données. Cette partie a été réalisée en entier.

Nous avons proposé des métriques de comparaison pour la dernière partie du projet, mais n'avons cependant pas pu étendre cela à d'autres formats que les relations booléennes de PyBoolNet, puisque les parties de traduction des règles logiques n'ont pas pu être finalisées dans les temps.

Annexes

Dépôt GitHub créé pour ce projet : https://github.com/MathieuJM/pgrou6ecn.
CoLoMoTo : https://colomoto.github.io.
pyLFIT : https://github.com/Tony-sama/pylfit.
GINsim : http://ginsim.org.
PyBoolNet : https://pyboolnet.readthedocs.io/en/latest.
BoolNet : http://www.colomoto.org/software/boolnet.html.

Diagramme de Gantt final du projet :

N°	Nom de la tâche
1	**Partie 1 - Intégration**
2	**Conversion pour l'entrée de pyLFIT**
3	depuis GinSIM (graphe de transition)
4	depuis BoolNet (séries temporelles)
5	depuis pyBoolNet (attracteurs)
6	**Conversion en sortie de pyLFIT**
7	vers GinSIM
8	vers BoolNet
9	vers PyBoolNet
10	Conversion SBML qual (E/S)
11	**Partie 2 - Altération des données intermédiaire**
12	Ajout de bruit sur les données en entrée de pyLFIT
13	Mise en place d'un réseau de neuronne pour réduire le bruit
14	**Partie 3 - Analyse des résultats**
15	Création d'un système de comparaison des modèles
16	Mise en place de scores issues de la comparaison
17	Étude des scores obtenus sur différents modèles