Projet TuxML :
Manuel de reprise de code

Corentin Chédotal Gwendal Didot Dorian Dumanget
Antonin Garret Erwan Le Flem Pierre Le Luron
Alexis Le Masle Mickaël Lebreton Fahim Merzouk

Encadrés par Mathieu Acher

4 Mai 2018

Remerciements

L'équipe du projet TuxML souhaiterait remercier monsieur Olivier Barais pour avoir pris le temps de présenter Docker et son utilisation ainsi que pour les pistes de travail concernant son emploi dans le cadre de notre projet.

Le logo du projet utilise des *emojis* modifiés du projet Noto mis à disposition gracieusement sous la licence Apache. Seul l'emoji `1f473 1f3fb` a été modifié avant son inclusion dans le logo afin d'en extraire et recolorer son turban. Puis l'ensemble du logo a été recoloré pour donner un aspect "plan de construction".
Le logo Linux Tux ainsi que le fond font partie du domaine public.

Table des matières

1 TuxML

Les scripts et programmes sont présentés dans leur ordre d'exécution normal quand cela est possible.

1.1 *Machine Learning Food* : `MLfood.py`

1.1.1 Description

Ce programme ne contient en réalité qu'une unique fonction `mlfood()` qui est appelée en fin de fichier. Ce programme était juste une suite d'instructions, mais dans un souci de lisibilité de la documentation générée, le code a été placé dans une fonction. Cette dernière est découpée en sections, les sections sont reconnaissables car séparées par une nouvelle ligne. Un marqueur `"Section #"` marque la séparation entre deux sections de code où # est le numéro de section.

Au début se situe les codes terminaux permettant d'afficher des couleurs sur la sortie standard de manière à rendre plus significative l'output et de retrouver les lignes intéressantes, certaines couleurs étant associées à un type de message tel que le rouge pour les messages d'erreurs ou le vert pour les succès.

1.1.2 Sections

Section 1 : Contient l'initialisation des arguments du programme à travers «argparse». On inclut ici les paramètres tels que «nbcompil», le nombre de compilations à faire, et «incremental», le paramètre incrémental. Viennent ensuite les options comme `--dev`, `--no-clean` et `--reset-logs`.

Section 2 : On vérifie la présence de l'option `--reset-logs`, dans ce cas une demande à l'utilisateur est faite pour confirmer la suppression de tout le contenu du dossier Logs avant de terminer l'exécution du script.

Section 3 : On vérifie l'existence de l'option `--dev` de manière à configurer l'image docker à utiliser, si elle s'y trouve alors l'image utilisée sera `tuxml/tuxmldebian:dev`, si cette option n'est pas présente alors l'image par défaut est l'image stable appelée *«prod»*. On affiche un message d'avertissement lorsque le programme est lancé sans l'option `--dev` afin de s'assurer que c'est bien la version stable que l'on veut utiliser et non pas la version `dev` en cours de développement. L'utilisateur est donc prié de répondre par *«y»* pour valider son choix ou par *«n»* ou n'importe quelle autre touche pour annuler l'exécution.

Section 4 : Cette section a été commentée pour facilité l'utilisation à grande échelle répétée, il s'agit d'une vérification au près de l'utilisateur du nombre de compilations demandée, si cette partie est dé-commentée alors le programme demandera à l'utilisateur si c'est bien son intention de lancer le nombre de compilation précisé. (L'avertissement n'apparaît qu'à partir de 50 compilations ou plus demandée).

Section 5 : Cette section ne concerne que de la vérification telle que celle du nombre de compilations demandées doit être supérieur à 0, on vérifie que l'on a bien une image sur laquelle on va commencer le calcul. On affiche aussi une précision à l'utilisateur sur pourquoi le programme va demander les privilèges de super utilisateur (pour Docker en l'occurrence).

Section 6 : Il s'agit de la plus grosse section, mais pour l'explication du code, les sous-sections se trouvant dans la section 6 sont comptées comme des sections à part entière. La section 6 contient la boucle principale du programme qui lancera les différents conteneurs Docker pour la compilation.

Section 7 : Cette micro-section permet d'exécuter la commande de vérification de mise-à-jour de l'image Docker. `sudo docker pull` suivi de l'image `tuxml:tuxmldebian:dev` ou *prod* permet de vérifier notre version de l'image sur la machine exécutant `MLfood.py`, si l'image est à jour on passe à la suite, sinon la commande télécharge la dernière version de l'image correspondant au tag (soit *dev* ou *prod*).

Section 8 : Ici nous créons le dossier `Logs/` dans le cas où il n'existe pas déjà et nous paramétrons la variable 'logsFolder' qui correspond au nom du dossier de logs associé à la compilation en cours, le nom est la date et l'heure de la compilation courante. Cette variable est utilisée en section 10.

Section 9 : Voici la section d'exécution principale de l'arborescence finale du projet pour un conteneur docker. Cette commande permet de lancer un nouveau conteneur Docker (`run`) avec une commande terminal au démarrage (`-t`) en précisant l'image à lancer qui, ici, se trouve dans la variable 'image' configurée en section 2 (`tuxml:tuxmldebian:dev ou` *`prod`*) suivi de la commande à lancer (`/TuxML/runandlog.py` est le chemin absolu dans le conteneur Docker).

Section 10 : La récupération des logs se fait ici, cela permet d'avoir une trace des logs dans le dossier local Logs/ en plus de ceux envoyés sur la base de données. La commande lancée par `os.popen(«sudo docker ps -lq», 'r')` permet de rouvrir un conteneur ayant fini sont exécution afin de l'ouvrir en lecture et de récupérer des fichiers. La récupération de fichiers se fait par les quatre lignes "`sudo docker cp ...` " qui fonctionnent comme la commande `cp` des systèmes Unix en indiquant le chemin de la source (chemin dans le conteneur Docker) et le chemin de la destination (sur la machine Hôte) ici dans le dossier Logs dans le dossier pointé par la variable logsFolder.

Section 11 : Par défaut, les conteneurs Docker sont gardés en mémoire et non supprimés, dans le programme le fonctionnement a été inversé par `MLfood.py` pour supprimer les conteneurs à la fin de l'exécution, il est toute fois possible de les garder en ajoutant l'option −−`no-clean` qui empêchera la suppression de ces conteneurs (afin de les manipuler manuellement par exemple).

Section 12 : Section à seul but d'être un affichage d'informations destinés à l'utilisateur tel que le nombre de conteneurs utilisés, le nombre de compilations par

conteneurs et par déduction le nombre total de compilations (qui est le nombre de compilations demandées multiplié par le paramètre incrémental + 1).

Section 13 : Il s'agit seulement de l'appel à la fonction a effectuer. C'est la première ligne de code qui s'exécute.

1.2 Exécution et récupération de logs : runandlog.py

1.2.1 Bibliothèques

Les bibliothèques utilisées ici sont des bibliothèques devant être pré-installées dans l'image du conteneur Docker utilisé.

- `MySQLdb` afin de pouvoir interagir avec une base de données.
- `tuxml_common et tuxml_settings` les bibliothèques personnelles permettant d'accéder aux paramètres de la base de données. (identifiants, hôte, ...)
- `bz2` la bibliothèque de compression permettant d'envoyer un fichier sur la base de données.
- `re` la bibliothèque d'expressions régulières, afin de trouver et récupérer le "*cid*" de l'entrée de la base de données à modifier en cherchant dans le fichier output.log. ("*cid*" : explication en section "*Fonctions*")

1.2.2 Description

Ce programme est appelé par `MLfood.py`, il est utilisé comme la première des deux étapes intermédiaires entre la commande de base `MLfood.py` et la commande de compilation `tuxml.py`. `runandlog.py` à été créé afin de récupérer la sortie standard de l'exécution de la/des compilations demandées dans un fichier `output.log` et qui permet de l'envoyer dans la base de donnée à l'entrée correspondant à la compilation effectuée.

1.2.3 Fonctions

Ce script commence par créer le parser d'arguments grâce à `argparse` afin de préciser les paramètres utiles, ici le paramètre *incremental*.

S'en suit la définition de la fonction d'envoi du fichier output à la base de données à l'entrée d'indice "cid". L'indice "*cid*" correspond à "Compilation ID" et est calculé en parcourant le fichier `output.log` de fin d'exécution. En lisant ce fichier lignes par lignes, on cherche grâce à la bibliothèque *re* à matcher sur la ligne comprenant "DATABASE CONFIGURATION ID=(\d+)", lorsque cette ligne a été trouvée alors "*cid*" prends la valeur contenue à l'emplacement du "*d*" . On utilise ensuite une connexion Socket afin d'accéder à la base de données, le fichier *output.log* à envoyer est ensuite compressé grâce à bz2.

L'exécution de la requête SQL `UPDATE` se fait par `execute(query, data)`. Le script commence par lancer l'exécution de `tuxLogs.py` afin de faire la/les compilations demandées en formatant la commande d'appel et en intégrant le paramètre

incrémental. Une fois terminée on va chercher dans le fichier *output.log* pour chaque ligne, celle correspondant à l'ID de l'entrée de la base de données.

Pour finir on appelle finalement la fonction de mise-à-jour de la base de données : `send_outputlog(cid, outputfilename, databasename)`.

1.3 Lanceur de l'exécution de tuxml : tuxLogs.py

1.3.1 Description

Ce script a pour but de lancer l'exécution de `tuxml.py` en lui transférant les paramètres récupérés de `runandlog.py` et donc de `MLfood.py`. Il s'agit du second script intermédiaire entre la commande `MLfood.py` et la commande `tuxml.py` ayant pour but d'obtenir la sortie standard des erreurs possibles d'exécution. Si `tuxml.py` s'arrête brusquement pour une quelconque raison, `tuxLog.py` s'arrêtera en affichant la *"stacktrace"*. Cet output sera alors récupéré par `runandlog.py` qui pourra donc écrire la *"stacktrace"* à la suite de l'output normal de `tuxLogs.py` (qui est celui de `tuxml.py`) afin de l'envoyer à la base de données.

`tuxLogs.py` est donc le script permettant de récupérer une éventuelle *"stacktrace"*.

1.3.2 Script

Comme chacun des premiers scripts, on commence par créer le "parser" d'arguments afin d'indiquer les paramètres nécessaires au fonctionnement. Dans le cas de `tuxLogs.py` il n'y a besoin de transférer que le paramètre incrémental.

Ce script se contente d'exécuter la commande de lancement `tuxml.py` en mode verbosité maximale (4) dans un but de débug et de suivi d'exécution, tout en précisant en argument le paramètre incrémental avec l'option de `tuxml.py` "–incrémental".

1.4 TuxML core

Les fichiers minimums et nécessaires au fonctionnement de TuxML sont situés dans le dossier core (voir https://github.com/TuxML/ProjetIrma/tree/master/core) :

- `tuxml.py` : Fichier principal avec la fonction de lancement de la compilation, d'analyse de logs, de reprise de compilation etc...
- `tuxml_argshandler.py` : Fichier permettant de gérer les arguments passés en paramètre à TuxML.
- `tuxml_bootCheck.py` : Contient les fonctions utilisées pour lancer des tests sur le noyau.
- `tuxml_common.py` : Ce fichier contient toutes les fonctions communes à tous les fichiers `tuxml_*.py`, telle que la fonction d'affichage.
- `tuxml_depLog.py` : Module chargé de logger la résolution des paquets manquants et d'exporter ces informations en CSV (logging de quel paquet(s) installé(s) pour résoudre la dépendance vers tel fichier manquant.
- `tuxml_depman.py` : Ce fichier contient toutes les fonctions utilisées pour installer des paquets, mettre à jour le système etc...

— `tuxml_depML.py` : Script chargé de lancer des compilations avec un nombre minimaliste de paquets pré-installés pour obtenir des données permettant la corrélation config<->environnement<->paquets requis. Gère aussi l'exportation de cette base au format CSV.
— `tuxml_environment.py` : Module permettant de retrouver l'environnement de compilation : information sur le hardware, sur le système etc
— `tuxml_sendDB.py` : Contient les fonctions permettant d'envoyer les résultats de la compilation et des tests du noyau à la base de donnée.
— `tuxml_settings.py` : Contient toutes les variables globales utilisées par TuxML.

Dans la suite de cette section nous détaillerons seulement les fichiers "core" les plus importants. Pour plus d'informations vous pouvez vous référer à la documentation Doxygen.

1.4.1 tuxml.py

Le fichier tuxml.py est le script principal dans le sens où c'est lui qui va lancer la compilation, s'occuper de l'analyse des logs, la correction d'erreurs etc... Il contient les fonctions suivantes :

— `install_missing_packages(...)` : construit d'abord la liste des paquets manquants grâce à la fonction `tuxml_depman.build_dependencies(...)` puis les installe grâce à la fonction `tuxml_common.install_packages(...)`.
— `log_analysis(...)` : recherche dans le fichier err.log des schémas ("Command not found", "Fatal error" etc...) indiquant un paquet ou un fichier manquant, puis construit la liste de paquets/fichiers manquants.
— `compilation()` : permet de lancer une compilation et de rediriger la sortie standard et la sortie d'erreurs vers les fichiers appropriés (respectivement std.log et err.log).
— `init_launcher()` : récupère les informations liées à l'environnement avec `tuxml_environment.get_environment_details()`, récupère le gestionnaire de paquets du système avec `tuxml_common.get_package_manager()`, puis met à jour le système avec `tuxml_common.update_system()` et installe les dépendances par défaut avec `tuxml_depman.install_defaut_dependencies()`.
— `gen_config(...)` : permet de générer aléatoirement un `.config` ou d'utiliser le Kconfig passé en paramètre, sous la forme d'un seed (e.g. 0xdeadbeef) ou d'un chemin vers un `.config` existant.
— `launcher()` : fonction contenant la boucle principale permettant de compiler un noyau, d'analyser les logs en cas d'erreur, d'installer les paquets manquants et de reprendre la compilation. Un chronomètre est démarré au début de la fonction pour connaître le temps d'installation. À chaque installation de paquets manquants, TuxML soustrait le temps d'installation au temps de compilation. Si la compilation a été un succès on lance un test de démarrage sur le noyau avec `tuxml_bootCheck.boot_try()`. À ce moment là on démarre un deuxième chronomètre pour connaître le temps de démarrage du noyau. Enfin on envoie les résultats des tests à la base de données avec `tuxml_sendData.send_data(...)`. On retourne l'identifiant de la compilation dans la base de données (`cid`).

1.4.2 `tuxml_bootCheck.py`

Comme indiqué précédemment ce fichier contient le code permettant d'effectuer des tests de bon fonctionnement sur les noyaux une fois ceux-ci compilés. En l'état il ne contient qu'une seule fonction mais il serait intéressant à terme de *refactorer* le code afin d'en extraire les tests en eux-même dans leurs propres fonctions. Pour le moment seul un test est fonctionnel, celui vérifiant si

Dépendances La vérification du bon lancement du noyau se fait principalement à l'aide de l'émulateur QEMU. Ainsi on utilise sa commande `qemu-system-x86_64` ici puisque pour le moment la compilation de noyaux se fait uniquement en 64 bits. Actuellement le programme utilise aussi une autre dépendance avec la commande `mkinitramfs` qui permet la création d'un environnement système permettant de faire tourner le noyau.

Fonctionnement Tout d'abord on fait un appel à `mkinitramfs` dans un sous-processus à part afin de mettre en place un système de fichier initial ou "initramfs" dans lequel se placera le noyau testé. Une fois ceci fait on va exécuter QEMU dans un autre sous-processus afin de faire tourner le noyau en même temps que la fonction de test. Les paramètres utilisés à ce moment là sont les suivants :

- `-kernel tset.PATH + "/arch/x86_64/boot/bzImage"` : paramètre indiquant que l'on souhaite utiliser un noyau en particulier suivi de l'emplacement de celui-ci (ici celui fraîchement compilé)
- `-initrd tset.PATH + "/arch/x86_64/boot/initrd.img-4.13.3"` : paramètre indiquant que l'on souhaite utiliser un système de fichier particulier suivi de l'emplacement de celui-ci (ici celui venant juste d'être créé par `mkinitramfs`)
- `-m 1G` : paramètre indiquant que l'on donne 1 Gio de mémoire à la machine virtuelle
- `-append console=ttyS0,38400` : paramètre indiquant que les sorties se font par le terminal (essentiel pour une compatibilité avec le plus de noyaux possibles et pour la récupération de la sortie)
- `-serial file:serial.out` : paramètre indiquant que l'on récupère la sortie série dans le fichier `serial.out`

Une fois la machine virtuelle lancée elle va commencer à écrire en temps réel tout ce qui s'affiche sur son terminal dans le fichier `serial.out`. On va alors à intervalle régulier (ici toutes les 10 secondes) *parser* ce fichier à la recherche soit d'un *kernel panic* qui est annonciateur d'un échec de lancement ou soit de la bonne ouverture d'un shell indiquant que le noyau semble fonctionner. Après un certain nombre de tentatives de lectures (imposé arbitrairement) on considère qu'il y a *timeout* et que le noyau ne fonctionne pas. En effet on a pu trouver empiriquement que parfois un noyau ne fonctionne pas mais peut se bloquer dans une boucle infinie au lancement qui ne renverra jamais de messages d'erreurs. **Attention : une trop petite valeur pour le timeout causera des faux négatifs car certains noyaux peuvent être très lents au démarrage (plusieurs minutes par exemple).**
A l'issue on va forcer l'arrêt du sous-processus de la machine virtuelle et on renvoie le code correspondant à ce qui est arrivé.

Retours La fonction `boot_try()` retourne un code indiquant le fonctionnement ou non du noyau et d'éventuelles raisons de non-fonctionnement le cas échéant. Ceux-ci sont spécifiés dans la documentation *Doxygen* ainsi que dans le tableau 1 en annexe.
Dans son utilisation normale la fonction ne doit pas renvoyer d'exception. Elle peut éventuellement en lever une en cas d'échec de la commande `mkinitramfs` ou en cas d'impossibilité d'ouverture du fichier de sortie de QEMU mais ces exceptions sont traitées en interne.

Développements en cours et futurs

— En réalité l'emploi de `mkinitramfs` est déprécié. En effet son utilisation semble incompatible avec les grilles de calculs. Il apparaît alors nécessaire de le remplacer par un autre système. Une branche de développement spécifique est dédiée à ce problème et une proposition de *roadmap* pour son remplacement a été soumise sur GitHub et rapportée en 2 en annexe. Ce changement de fonctionnement implique cependant de changer aussi le test d'accès au shell car la piste actuellement explorée emplie un *Busybox* quasi-nu.
— Un test de temps de lancement en plus de celui de bon fonctionnement est aussi en cours d'essai sur une branche séparée mais est encore très expérimental et l'authenticité de ses résultats est encore à confirmer.

1.4.3 tuxml_depman.py

Module chargé de l'installation des paquets. Il est aussi utilisé pour la résolution des dépendances manquantes : à partir d'une liste de fichiers manquants pour pouvoir compléter la compilation, la fonction `build_dependencies(...)` permet de découvrir la liste des paquets contenant ces fichiers.
Bien qu'avant tout prévu pour Debian, il est conçu pour pouvoir ajouter facilement le support d'autres gestionnaires de paquets et est déjà compatible avec les gestionnaires *pacman* (*ArchLinux*) et *dnf* (*Red Hat*, *Fedora*...)

1.4.4 tuxml_environment.py

Module chargé de récupérer et logger les détails de l'environnement sur lequel on exécute Tuxml.

Cet environnement est découpé en trois parties :

— Fonction `get_compilation_details()` : L'environnement de compilation (Par exemple la version de GCC et de la LIBC utilisée, nombre de cœurs utilisés pour compiler...).
— Fonction `get_hardware_details()` : L'environnement matériel (Processeur, RAM, type de stockage).
— Fonction `get_os_details()` : Le système d'exploitation (distribution, version de la distribution, version du noyau).

On veut pouvoir prendre en compte l'environnement lors de la phase de learning car certaines mesures effectuées en sont dépendantes. Par exemple, le temps de compilation dépend fortement du CPU, le temps de boot dépend du type de disque. Récupérer l'environnement répond donc à trois objectifs :

1. D'éviter que des différences entre les environnements de différentes compilations ne faussent les corrélations.
2. Pour le learning et l'influence des options sur la compilation : déterminer à quel point les détails d'environnement influent sur chaque mesure et pour quelle(s) mesure(s) ils influent.
3. Pour le learning des packages, certaines dépendances peuvent varier pas seulement en fonction des options de compilation du noyau mais aussi en fonction de l'environnement d'exécution (exemple : les pilotes pour du matériel spécifique).

Pour chaque compilation, on log donc ces détails d'environnement dans la base de données et sous forme CSV. Cela favorise également le debug, un problème de résolution de paquet manquant étant souvent étroitement lié à ces détails.

1.4.5 tuxml_depLog.py

tuxml_depLog est chargé de logger toute la procédure d'installation des paquets manquants. Pour chaque fichier manquant, on log les packages candidats possibles, les packages installés, et si la résolution a été possible. L'ordre de logging des paquets alors installés est important, car Tuxml installe un par un les paquets candidats, jusqu'à ce que la compilation fonctionne.
C'est important à noter pour la partie learning où l'on veut pouvoir déterminer quel(s) package(s) résout quelle dépendance(s) : typiquement, c'est le dernier package le plus intéressant car on sait que si la dépendance a été résolue, on est au moins sûr que ce package en particulier est à installer. Mais d'autre paquets précédents dans la liste peuvent avoir étés utiles la pour résoudre aussi, mais il est moins trivial de déterminer lesquels.

1.4.6 tuxml_sendDB.py

Ce module est appelé à la fin de la compilation. Il permet d'envoyer les résultats de la compilation et du test à la base de données. Il est composé des fonctions suivantes :

- `get_kernel_size()` : permet de récupérer la taille du kernel compilé.
- `send_data(...)` : permet d'envoyer les résultats sur les différentes tables de la base de données (Compilations, Incremental_compilations, Tests). D'abord on récupère toutes les informations contenues dans l'environnement ainsi que la date de la compilation, le temps de compilation, la taille du noyau, la liste des dépendances et les fichiers qui seront envoyés dans la base de données : .config, err.log et std.log. Toutes ces informations seront stockées dans la table "Compilations". Si le mode incrémentale est utilisé, on sauvegarde cette information dans la table Incremental_compilations avec l'identifiant de la nouvelle compilation et la compilation sur laquelle se base le mode incrémental (BASE_CONFIG). Ensuite on crée une nouvelle entrée dans la table Tests avec la date du test, le temps de démarrage et l'identifiant de la compilation testée. Ce script renvoie l'identifiant (cid) de la compilation qu'il vient de sauvegarder.

2 Scripts auxiliaires

2.1 TuxML Project Docker Image Manager : TPDIM.py

2.1.1 Description

TPDIM est un script visant à faciliter l'utilisation des commandes Docker permettant la création et gestion d'image. Ce script possède 3 fonctions, chacune pouvant être utilisées indépendamment les unes des autres ou de façon combinées grâce à l'utilisation de `argparse`.

2.1.2 Fonctions

Cette section va traiter du fonctionnement de chacune des fonctions de TPDIM :

- `mkGenerate` : Cette fonction permet de vérifier s'il y a des DockerFile déjà présent dans le projet, et laisse le choix à l'utilisateur de les supprimer ou non. Puis appelle la fonction `docker_generate` en vérifiant si l'utilisateur souhaite utiliser le fichier de dépendances par défaut ou un de son choix (cela n'est pas fonctionnel pour le moment).
- `docker_generate` : Chaque image utilisée dans MLfood.py est en réalité composé de deux images, une intermédiaire et une finale. Cette fonction permet de générer les deux de façon automatique, en écrivant simplement dans des fichier appelés DockerFile. Pour le moment la fonction utilise `dependences.txt` dans tous les cas, il faudra changer pour permettre de choisir quel fichier de dépendances on veut utiliser.
- `mkBuild` : Cette fonction vérifie qu'un DockerFile soit bien présent dans le dossier donné en paramètre au script (ou le dossier courant sinon), puis appelle la fonction `docker_build`
- `docker_build` : Cette fonction lance la commande docker permettant de construire une image.
- `mkPush` : Cette fonction lance la fonction `docker_push`.
- `docker_push` : Cette fonction permet de mettre l'image sur le *repository* docker hub distant du projet TuxML en lançant la commande docker permettant cela.

2.2 Génération de CSV pour le Machine Learning : csvgen

CSVgen permet d'extraire la base de données pour remplir un fichier CSV contenant les entrées et sorties de la compilation, notamment les options sélectionnées ainsi que la taille du kernel résultant et la durée de la compilation. Les *features* sont placées en colonne et les *samples* en ligne pour faciliter l'import pour le machine learning.
Le script fait d'abord un appel à la base de données pour connaître le nombre et le types des options de compilation afin de renseigner leurs valeurs par défaut dans le CSV. Ensuite, plusieurs mêmes requêtes sont effectuées afin de réduire la taille de données à envoyer dans une seule requête et pouvoir construire le fichier de façon incrémentale.
Comme les fichiers `.config` sont récupérés directement (après décompression `bzip2`), l'ordre des options n'est pas garanti et il faut donc les trier manuellement pour que

les colonnes du fichier CSV correspondent. La class `DictWriter` de Python aurait pu être utilisée pour contourner ce problème mais son implémentation est très lente (le nom des colonnes est trié pour chaque ligne, et non pas une seule fois dès que les noms sont connus) et prenait 45 minutes pour 1500 compilations. Une implémentation personnalisée règle ce problème et ce même fichier est généré en une dizaine de secondes.

2.3 Exploitation des données : stats.ipynb

2.3.1 Bibliothèques

— `«sklearn»` est utilisé afin de mettre en forme ainsi que d'analyser les données récupérées auparavant. Il est utile aussi bien lors de la mise en forme initiale des données avec `sklearn.preprocessing`, que lors de l'apprentissage sur les données avec `sklearn.tree` ou `sklearn.ensemble` permettant la création de diverses méthodes d'apprentissages expliquées par la suite.

2.3.2 Description

Le but de cette partie est d'utiliser le *Machine Learning* afin de prédire l'aspect (ici notamment la taille) d'un noyau quelconque à partir d'observations sur un ensemble de noyaux dont nous connaissons les caractéristiques.
Les caractéristiques que l'on connaît pour chaque noyau de l'ensemble sont l'ensemble des options contenus dans le `.config` ainsi que sa taille une fois compilé et son temps de compilation. Toutes ces informations sont contenues dans un CSV, que l'on parcourt afin de les récupérer.
Pour cette partie nous avons utiliser un `Notebook Python` afin de pouvoir mettre plus facilement en forme les différentes parties du code. En effet ce format utilise un système de "cases" dans lequel on peut écrire directement du code python qui peut être exécuté. Ainsi le résultat de chaque case est conservée sous celle-ci ce qui permet une lecture facile et rapide des résultats qui nous intéresse.
De plus ce format est facilement exportable puisque, une fois exécuté, le notebook conserve les différents résultats directement dans la page de ce dernier permettant l'envoi direct du notebook, qui contient alors le code qui a été exécuté ainsi que les résultats associés à chaque partie. Il est donc simple de se déplacer et de comprendre chaque case.

2.3.3 Séparation

Ainsi ce programme est séparé de la façon suivante :
— Import des différents modules utilisés durant les phases suivantes.
— Analyse des données et analyse d'une corrélation.
— Preprocessing des données pour l'utilisation du `Machine Learning`.
— Utilisation de diverses méthodes de `Machine Learning`. On utilise ici les arbres de décisions (`Decision Trees`) et les forêts de décision aléatoire (`Random Forest Regressor`).

2.4 Etude des dépendances ML : tuxml_depML.py

2.4.1 Description

En étudiant les dépendances de compilation comme expliqué dans post−mortem, il a fallu coder ce script qui est très similaire à tuxml.py mais ne collecte pas et n'envoie pas la même chose à la base de données. Les étapes de ce script sont les suivantes :

- Un fichier de configuration*(.config)* est généré pour la compilation et est envoyé à la base de données.
- Les caractéristiques de l'environnement sont collectées puis envoyées dans la base de données.
- La compilation se lance avec des paquets dont on est sûr qu'ils sont indispensables (gcc, make, mysqlclient ..).
- A chaque fois qu'un fichier manquant est détecté, le paquet qui lui correspond est installé puis envoyé dans la base de données.

Entre deux compilations, il faut supprimer les paquets installés à la compilation qui la précède.

Une fonction `write_bdd_to_csv()` récupère tout le contenu de ces compilations (tables `depML_environnement` et `packages`) et l'exporte au format CSV, dans une forme plus adaptée pour la bibliothèque de learning.

Il reste donc à effectuer un nombre de compilations pour obtenir suffisamment de données et de les soumettre à la bibliothèque de learning.

Comme la bibliothèque fonctionne avec des données numériques, il va être nécessaire d'encoder les informations textuelles (nom de paquets par exemple) au format numérique via cette outil :
`http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html`

Mode opératoire

Objectif : A partir d'une configuration (environnement de compilation + options du Kernel), prédire automatiquement la liste des dépendances.
Du point de vue du Machine Learning c'est un problème de classification.

Pendant la compilation, nous générons un fichier de log qui, pour chaque fichier manquant requis, nous indique les paquets installés pour l'obtenir. On indique aussi si on a réussi à résoudre la dépendance ou pas.
Voici un exemple :

Missing files encountered	Missing packages installed	Resolution successfull
bversion.h	['cmake-doc', 'gcc-6-plugin-dev']	true

Ici on voit que pour le fichier requis *bversion.h*, on a pu résoudre cette dépendance en installant les paquets *cmake-doc* et *gcc-6-plugin-dev*.
Dans cette exemple, *cmake-doc* est installé pour rien et inutile.

Pour prédire quels paquets doivent être installés en fonction des options, l'idée est de désactiver pré-installation de paquets sauf sur ceux que les paquets sur lequel on sait qu'il est inutile d'investiguer car forcément requis quelque soit les options (gcc, make). On peut ainsi générer un ficher de log complet pour les dépendances.

On peut alors effectuer plein de compilation et obtenir une structure qui fait corrélation entre **configuration<-->options<-->paquets installés**.

Voici la forme simplifiée d'une entrée :

option1	...	option N	Configuration materielle et logicielle	Missing files encountered	Missing packages installed	Resolution successfull
val1	val2	val N	(CPU, disque dur, distribution, version GCC, etc...)	file1.h	['pck1', 'pck2']	true

A partir d'une liste d'entrées, on peut utiliser le machine learning pour tenter d'établir des corrélation entre des options et des paquets requis. On peut déterminer que tel ou tel paquet n'est requis que si telle ou telle option est activée. Même sans machine learning il est possible d'extraire de l'information utile : les paquets qui sont installés à chaque compilation sont facilement détectables et peuvent être ajoutés à la liste minimale de dépendances requises.

Pour plus de détailes dans la doc interne :
`https://github.com/TuxML/DocInterne/blob/master/Dependencies.md`

A Annexes

A.1 Tableaux

Code renvoyé	Signification
0	Le noyau a correctement été lancé (accès au shell possible)
-1	Détection d'un *kernel panic* (le boot a échoué)
-2	*Timeout* après une certaine attente (le boot a probablement échoué ou prend plus de temps que le délai choisi)
-3	Échec de fonctionnement de la fonction de test (un composant de la fonction a échoué, aucune information sur le noyau)

TABLE 1 – Codes d'erreur renvoyés par la fonction `boot_try()`

V0	Mise en place d'un initramfs construit au préalable de façon arbitraire en amont sur le Docker et modification du script de test de boot pour qu'il fonctionne
V1	Modification du script pour qu'il envoie le temps de boot aussi
V2	Mise en place d'un script sh pour générer le initramfs en temps réel

TABLE 2 – *Roadmap* de l'évolution du système d'`initramfs`

A.2 Documentation *Doxygen* du projet