MASTER EN SCIENCES INFORMATIQUES

Configuration assistée par machine learning
études empiriques et application à la prédiction de défauts dans l'impression 3D

Amand, Benoit

Award date:
2018

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2017–2018

Configuration assistée par machine learning: études empiriques et application à la prédiction de défauts dans l'impression 3D

Benoit Amand

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Patrick Heymans

Co-promoteur : Maxime Cordy

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Remerciements

A l'issue de ce mémoire, je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à sa réalisation.

Je tiens à remercier mon promoteur, Monsieur P. Heymans et mon co-promoteur, Monsieur M. Cordy, pour le soutien et la guidance que j'ai reçue. Je remercie aussi Monsieur M. Acher, professeur associé à l'université de Rennes 1, pour ses conseils et son apport à mes recherches.

Je tiens aussi à remercier mon frère Julien, pour sa relecture attentive et ses critiques constructives sur le présent document, ainsi que Vincent Henrotin, pour les mêmes raisons.

Enfin, merci à mes parents, amis et collègues pour leur aide et leur soutien tout au long de la réalisation ce mémoire et de mon master.

Résumé

La modélisation de la variabilité permet de décrire les caractéristiques, paramètres et contraintes d'une ligne de produits. Un usage commun des modèles de variabilité est l'assistance à la configuration et la validation.

La modélisation manuelle de la variabilité étant une tâche longue et ardue, il existe un grand nombre de techniques visant à construire automatiquement les *feature models*. Ces techniques sont généralement spécifiques à chaque ligne de produits.

Dans ce travail, nous proposons une technique de modélisation de la variabilité générique, basée sur le *machine learning* : nous utiliserons un oracle pour prédire la validité d'échantillons, qui seront ensuite utilisés pour entrainer un classificateur. Notre technique nécessitera donc la conception d'un tel oracle par ligne de produit.

Nous validerons notre méthode en évaluant ses performances sur deux types de modèles : d'une part les modèles S.P.L.O.T., qui fournissent des exemples génériques de *feature models* et d'autre part les objets 3D configurables au format OpenSCAD du site Thingiverse, qui sont un cas pratique où une telle modélisation est nécessaire.

Mots-clés : modélisation variabilité, modèle de caractéristiques, apprentissage automatique, oracle, S.P.L.O.T., Impression 3D, Thingiverse, OpenSCAD.

Abstract

Variability modeling allows to describe features, parameters and constraints in a product line. A common use of such model is to assist product configuration.

Manually modeling variability being a long and tedious task, a growing number of techniques have been proposed to build feature models. Such techniques are generally specific to a given product line.

In this paper, we propose a more general modeling technique based on machine learning: we will train a classifier, using an oracle to predict the validity of samples. Our technique will thus require the making of such an oracle for each product line.

We will validate our method by evaluating its performances on two types of models: first we will use S.P.L.O.T. samples, which provide generic examples of feature models and second we will use OpenSCAD 3D objects available on Thingiverse as a more practical use case, where variability modeling was deemed necessary.

Key-words : variability modeling, feature model, machine learning, oracle, S.P.L.O.T., 3D printing, Thingiverse, OpenSCAD.

Table des matières

Introduction

La modélisation de la variabilité permet de décrire un ensemble de produits et de fournir des outils pour les utiliser. Elle trouve entre autres son intérêt dans les lignes de produits logiciels ou les chaines de production, en aidant l'utilisateur humain à configurer un produit. L'introduction de formalismes de modélisation permet aussi la création d'outils automatique pour la validation de ces produits.

Les modèles de variabilité permettent de décrire les caractéristiques et paramètres d'un produit. Différentes configurations de produit peuvent ensuite être obtenues en sélectionnant certaines caractéristiques et en assignant des valeurs aux paramètres. Souvent, le modèle intègre des contraintes empêchant certaines combinaisons de paramètres.

Un configurateur offre une interface utilisateur permettant la configuration d'un produit, généralement par le biais de formulaires dynamiques, basés sur le modèle de variabilité sous-jacent. Dans ces formulaires, on retrouve les paramètres et caractéristiques du modèle. Les contraintes limitent aussi les possibilités de l'utilisateur, pour s'assurer qu'il parvienne aisément à obtenir la variation désirée du produit. Les contraintes jouent un rôle très important car elles permettent d'éviter que des configurations invalides d'un modèle ne soient utilisées.

Un formalisme largement utilisé est celui des *feature models*. Ceux-ci permettent de décrire un modèle de variabilité sous forme d'une série de caractéristiques (*features*) et de contraintes. Ils sont principalement utilisés dans les lignes de produits logicielles (*Software product line* - SPL), notamment pour la validation [1], la configuration automatique [2] ou comme outil de communication [3]. Une extension, les *attributed features models* [4], introduit la notion d'*attributs*, qui peuvent prendre différentes valeurs.

La modélisation manuelle de la variabilité est une tâche difficile, demandant du temps et une expertise technique pouvant prendre énormément de temps. Différentes techniques ont été proposées pour permettre une modélisation automatique des *features models* [5] [6]. Ces techniques se basent généralement sur une connaissance du produit à modéliser pour construire le *feature model* et ses contraintes.

Dans ce travail, nous aborderons la modélisation de la variabilité comme un problème d'apprentissage automatique : nous entrainerons un classificateur capable de distinguer les configurations valides et invalides ; celui-ci fera office des contraintes du modèle. Afin d'entrainer un classificateur, nous aurons besoin d'un ensemble de configurations, ainsi qu'une connaissance de leur validité. Pour ce faire, nous utiliserons un *oracle*, capable de prédire la validité d'une configuration donnée.

Nous générerons un échantillon de paramètres et utiliserons l'oracle pour prédire leur validité. Ce faisant, nous obtiendrons les données nécessaires pour entrainer notre classificateur. La Figure 1 résume ce processus.

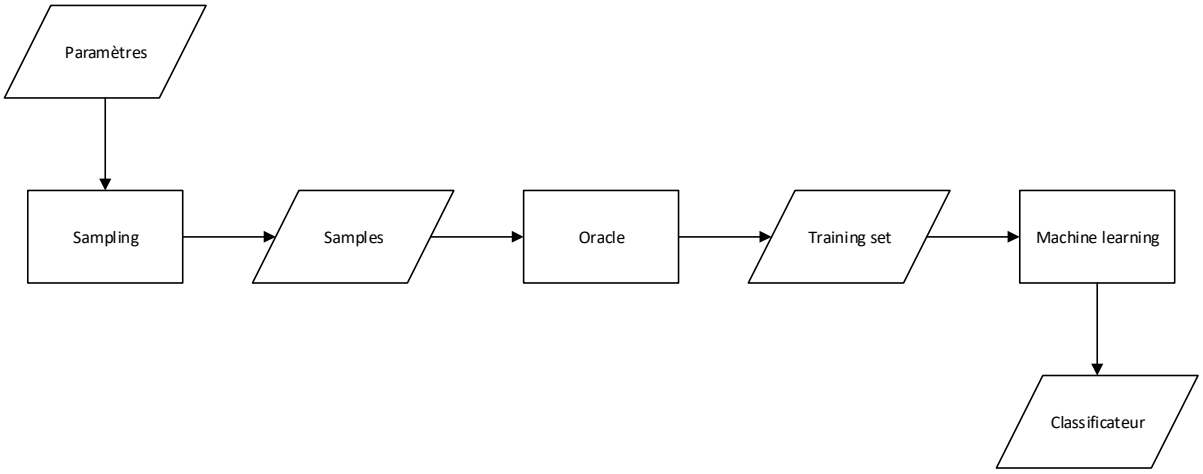


FIGURE 1: SCHEMA DE LA MODELISATION

Après quoi nous introduirons une méthode générique pour modéliser la variabilité, sous réserve de posséder un oracle spécifique au modèle à étudier et de connaitre les paramètres utilisés.

Dans un premier temps, nous étudierons l'efficacité de cette méthode avec des *feature models* existants, issus du projet S.P.L.O.T. Ce projet fournit un ensemble de modèles complets, avec contraintes, issus de problèmes réels. Nous utiliserons, pour construire un oracle, la validité du *feature model*, obtenant ainsi des prédictions parfaites. Dès lors, nous pourrons évaluer différents échantillons et classificateurs, sans nous occuper des erreurs que pourrait introduire la prédiction. Nous étudierons aussi l'intérêt de combiner cette technique à l'écriture manuelle de contraintes.

Dans un second temps, nous nous intéresserons à un cas plus concret : celui des modèles 3D paramétrables et de Thingiverse. Il s'agit d'un site communautaire regroupant des créateurs qui modélisent des objets 3D et des imprimeurs qui utilisent ces modèles. Il existe différents formats de modèles 3D, qui sont construits dans différents outils de modélisation spatiale. Parmi ceux-ci, il existe des modèles paramétriques, dont la particularité est de pouvoir produire différents objets, en fonction des valeurs données à leurs paramètres.

L'intérêt des objets paramétriques est qu'ils permettent aux utilisateurs d'adapter un modèle à leur besoin sans devoir utiliser d'outils de modélisation, qui demandent du temps et des compétences précises. Ces objets sont plus largement réutilisables que les objets non-paramétriques. Afin d'utiliser efficacement ces objets paramétriques, il est cependant nécessaire de posséder un outil facilitant leur configuration.

Sur le site de Thingiverse, un configurateur existe, mais sans contraintes sur les objets. L'impression 3D étant un procédé chronophage, un outil permettant d'éliminer les objets défectueux avant de débuter l'impression est le bienvenu, d'où l'intérêt de modéliser les contraintes des objets. Étant donné qu'il existe déjà un grand nombre de modèles existants et la difficulté que présente l'écriture manuelle des contraintes, notre technique présente ici un intérêt certain.

Nous construirons un oracle permettant de prédire la validité des objets 3D paramétrables et évaluerons ses performances avec des objets 3D issus de Thingiverse. Contrairement au cas précédent, nous ne posséderons pas d'information sur la véracité de l'oracle –pour cela, il faudrait imprimer les configurations échantillonnées, ce qui n'est pas envisageable. Nous validerons donc l'efficacité de notre méthode à restituer des résultats similaires à ceux de l'oracle.

Afin de valider l'intérêt de notre méthode, nous discuterons des quatre critères suivants, qui décrivent les caractéristiques attendues d'un modèle de variabilité :

- ses performances à l'exécution ;
- sa complexité de modélisation ;
- son exactitude ;
- sa compréhensibilité par un humain.

Dans le premier chapitre, nous discuterons plus en détail des formalismes et de notre méthode. Dans le deuxième, nous introduirons la classification et en particulier les algorithmes qui seront utilisés. Les quatrième et cinquième chapitres présenteront respectivement l'application de la méthode aux modèles S.P.L.O.T. et aux objets 3D. Ils expliqueront la mise en place des tests et détaillerons leurs résultats.

1 Définitions et méthode

Dans ce chapitre, nous définirons les termes et formalismes utilisés pour modéliser la variabilité et présenterons la méthode étudiée dans ce travail. Dans un premier temps, nous verrons les notions de variabilité et les formalismes de *feature models* et dans un second temps, nous parlerons plus en détail de la notion de validité et de contraintes et décrirons notre méthode pour les modéliser et les aspects qu'il nous faudra évaluer.

1.1 Variabilité

Par modèles de variabilité, on désigne un ensemble d'éléments étant des variations d'un modèle commun. La modélisation de la variabilité entend fournir un modèle paramétrique, permettant d'obtenir les différentes variations, ou configurations, possibles en affectant différentes valeurs aux paramètres.

Les modèles de variabilité sont notamment utilisés dans la validation automatique, ou pour assister un utilisateur. Dans la suite, nous allons étudier différents modèles de variabilité : les *feature models* et des objets 3D paramétrables.

1.2 Feature Models

Issu de l'étude des Software Product Lines (SPL), un *feature model* [7] [4]est un modèle représentant un ensemble de caractéristiques ou *features* formant un produit et les relations et contraintes qui s'appliquent entre ces *features*. De tels modèles peuvent être utilisés pour générer des configurateurs [2], ou valider des lignes de produits [8]. Les différentes variations d'un *feature model* sont obtenues en sélectionnant différentes *features*.

Les *features models* sont représentés sous forme d'un *feature diagram* [7]. Un tel diagramme représente les *features* sous forme d'un arbre dont les branches forment les contraintes du modèle. Il existe différents formalismes pour représenter les *feature models,* proposant différents types de relations possibles entre les *features* [7] [4] [9], mais nous ne nous intéresserons pas à ceux-ci pour l'instant (un formalisme sera introduit dans le chapitre S.P.L.O.T.).

De manière générale, l'arbre des *features* définit des relations « parent-enfant » et des « groupes de features ». Les premières contraignent la présence de la *feature* enfant à celle du parent - l'enfant pouvant être facultatif ou obligatoire. Les groupes définissent une contrainte sur la présence des enfants d'un même parent (par exemple : « *ou exclusif* », « *et* », « *minimum deux* »).

D'autres contraintes, qui ne sont pas représentables dans cet arbre, sont décrites séparément (généralement sous forme de propositions logiques). On les appelle contraintes non-hiérarchiques (*cross-tree constraints*).

1.2.1 Configurations

Une variation obtenue à partir d'un *feature* model est appelée configuration. Une configuration d'un *feature model* est une sélection de *features* de celui-ci. En d'autres termes, c'est une association de chaque *feature* à une valeur « présente » ou « absente ».

Une *configuration* d'un *feature model* est *valide* si l'ensemble des contraintes du *feature model* est respecté.

1.2.1.1 Exemple

La Figure 2 montre un des *feature models* issu de S.P.L.O.T., celui-ci comporte onze *features*. La *feature* racine est « telecom », elle est toujours obligatoire ; elle possède trois *features* enfants : « ipvoice », « rack » et « messaging ». Des trois, seul « rack » est obligatoire, ce qui signifie qu'une configuration n'est valide que si « rack » y est présente, et que « ipvoice » et « messaging » peuvent être absents.

Les deux *features* « swpack1 » et « swpack2 » forment un groupe d'alternatives, c'est-à-dire un groupe formé de tous les enfants d'un même parent et dont une et une seule *feature* peut être sélectionnée[1] simultanément. Un autre exemple de groupe d'alternatives est donné par les enfants de « size ».

Le modèle possède aussi deux contraintes non-hiérarchiques :

- ¬swpack1 ∨ ¬upgrade_swpack
- ¬ipvoice ∨ ¬messaging ∨ upgrade_swpack

Dans S.P.L.O.T., toutes les contraintes sont des disjonctions sur la présence ou l'absence d'une *feature*. La première contrainte impose que « swpack1 » et « upgrade_swpack » ne peuvent être choisis simultanément et la seconde que si « ipvoice » et « messaging » sont sélectionnés, on doit aussi sélectionner « upgrade_swpack ».

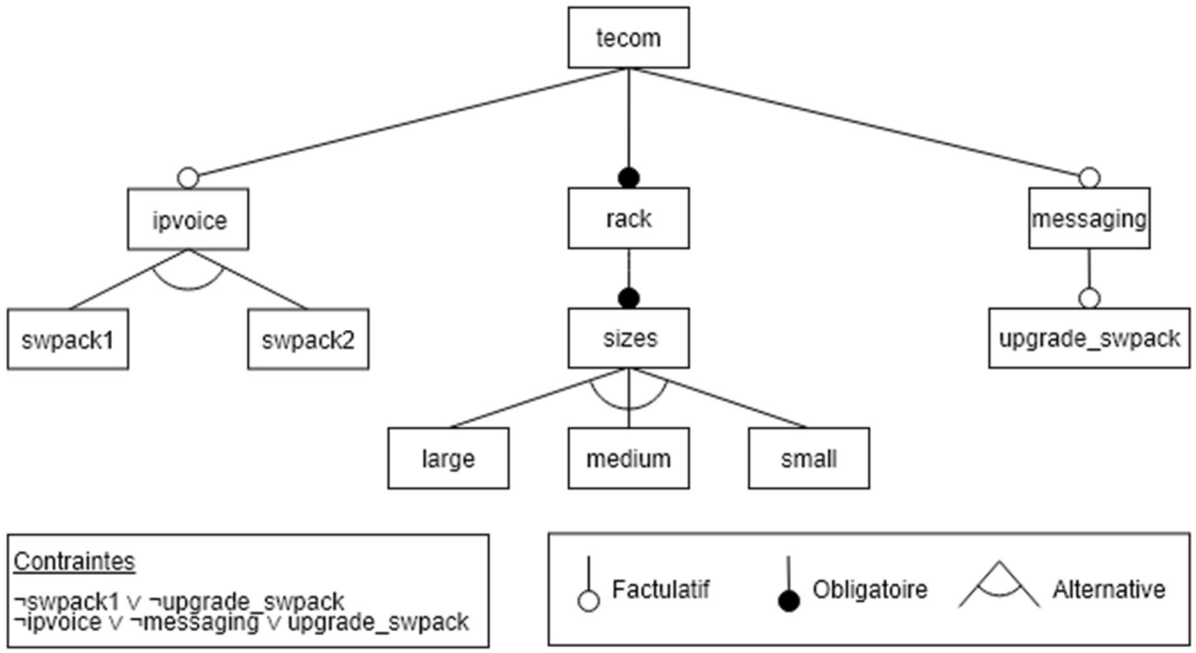


FIGURE 2: REPRESENTATION DU MODELE S.P.L.O.T "TELECOM"

Le Tableau 1 montre certaines configurations du modèle précédent, ainsi que leur validité.

TABLEAU 1: EXEMPLES DE CONFIGURATIONS POSSIBLES DU MODELE « TELECOM »

Features sélectionnées	*Validé*	*Raison*
telecom, ipvoice, swpack1, rack, sizes, large, messaging	Oui	
telecom, ipvoice, swpack1, rack	Non	"sizes" est obligatoire

[1] Notons que, pour qu'une configuration soit valide, la contrainte de groupe doit être respectée seulement si leur parent est présent. Si le parent est absent, aucun enfant ne peut être présent.

ipvoice, swpack1, rack, sizes, large, messaging	Non	“telecom” est obligatoire
telecom, rack, sizes, large, upgrade_swpack	Non	“upgrade_swpack” a son parent non-sélectionné

1.2.2 Attributed Feature Models

Une extension commune des *feature models* est l’association *d’attributs* [4] à certaines *features*. Un attribut est associé à un domaine de valeurs possibles. Un *attributed feature model* est constitué d’un ensemble de *features*, d’*attributs* qui leur sont liés, d’un domaine associé à chaque attribut, de contraintes sur le *feature model*, ainsi que de contraintes sur les *attributs.*

La définition d’une *configuration* d’un *attributed feature model* étend la définition d’une configuration d’un *feature model* en associant à chaque attribut d’une *feature* présente[2] une valeur du domaine de cet attribut.

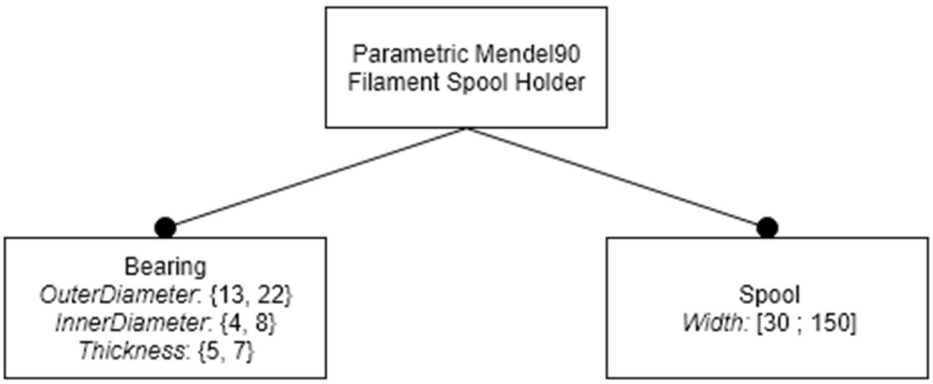
Une *configuration* d’un *attributed feature model* est valide si toutes les contraintes de l’*attributed feature model* sont vérifiées.

Notons qu’un *feature model* est isomorphe à un *attributed feature model* sans attribut. Nous pouvons donc considérer les *attributed feature models* comme une généralisation des *features models*. Nous considérerons ensuite les modèles S.P.L.O.T. et les objets 3D paramétrables comme deux cas particuliers d’*attributed feature models*.

1.2.2.1 Exemple

La Figure 3 montre la modélisation d’un objet 3D paramétrable dans un *attributed feature model*. Ce modèle ne permet qu’une seule configuration de ses *features* –comme c’est généralement le cas pour les modèles OpenSCAD– mais possède une série d’attributs. Nous avons introduits ici deux *features*, « Bearing » et « Spool », pour regrouper les attributs par thème.

FIGURE 3: ATTRIBUTED FEATURE MODEL POUR LE MODÈLE OPENSCAD "PARAMETRIC MENDEL90 FILAMENT SPOOL HOLDER" [10]

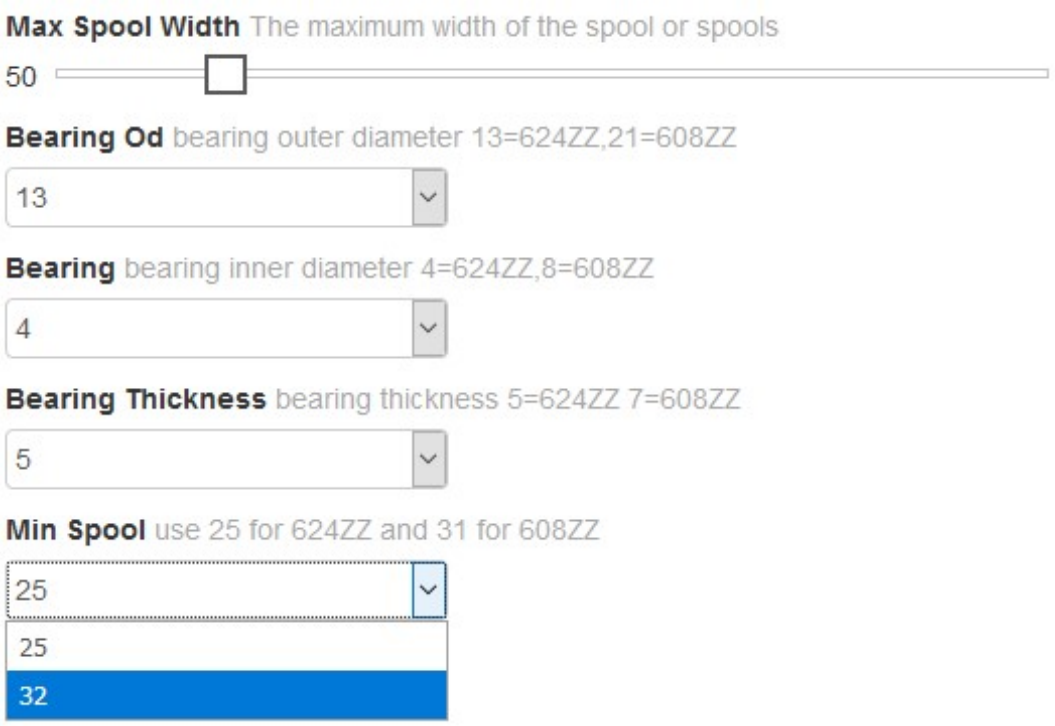


La *feature* « Bearing » possède trois attributs : « OuterDiameter », « InnerDiameter » et « Thickness ». Chacun de ses attributs possède un domaine finit qui n’accepte que deux valeurs possibles (respectivement 13 ou 22, 4 ou 8 et 5 ou 7). L’attribut « Width » de la *feature* « Spool » est un intervalle réel qui admet des valeurs entre 30 à 150.

[2] Dans [4], chaque attribut possède une *valeur nulle*. Par définition, une configuration associe cette valeur à tout attribut dont la *feature* n’est pas présente.

Notons qu'aucune contrainte n'est présente dans ce modèle et qu'il accepte donc n'importe quelle configuration. De fait, nous n'avons pas encore définit la notion de validité d'un modèle 3D ni modélisé les contraintes pour que la validité des configurations du modèle corresponde à cette notion. Le configurateur de *Thingiverse* met en place ce genre de configuration, sans contrainte sur les configurations. La Figure 4 montre le formulaire généré de ce configurateur pour le modèle précédent.

FIGURE 4: CUSTOMIZER THINGIVERSE POUR LE MODÈLE OPENSCAD "PARAMETRIC MENDEL90 FILAMENT SPOOL HOLDER" [10]



1.3 Généralisation

Dans un *attributed feature model*, la validité d'une configuration est modélisée à l'aide d'un ensemble de *contraintes* qui doivent toutes être satisfaites afin que la configuration soit valide. Ces contraintes sont typiquement formées de relations entre les *features* (implication, exclusion mutuelle, etc.) et d'opérateurs logiques, mais peuvent être étendues.

Par exemple, dans [4], un *attributed feature model* est caractérisé par :

- un ensemble de *features* ;
- un arbre qui est décrit par l'ensemble de ces branches, l'ensemble des *features* obligatoires et l'ensemble des groupes de *features* ;
- un ensemble *attributs*, et l'association d'un *domaine* et d'une *feature* à chaque *attribut*,
- des contraintes non-hiérarchiques « lisible par un être humain » sur les *features* et les *attributs*[3] ;
- une contrainte arbitraire sur les *features* et les *attributs*, reprenant l'ensemble des contraintes non-exprimables par les précédentes.

[3] Ces contraintes prennent toutes la forme d'une implication entre deux facteurs « booléens ». Ceux-ci étant soit la présence ou l'absence d'une *feature*, *soit* une comparaison entre la valeur d'un attribut et une constante.

De manière plus générale, nous pouvons définir le **modèle de variabilité** comme étant la donnée d'un ensemble de *features* et *d'attributs*, ainsi que d'une *fonction de validité*. Cette dernière est un prédicat prenant pour valeur une configuration du modèle et lui attribuant vrai si et seulement si cette configuration est valide.

Pour un *attributed feature model*, celui-ci est la conjonction des contraintes hiérarchiques, non-hiérarchiques et de la contrainte arbitraire.

Cette définition nous permet d'ignorer l'aspect hiérarchique et de nous concentrer sur la modélisation de la *validité* d'un modèle de validité.

1.3.1 Fonction de validité

La fonction de validité peut prendre plusieurs formes, telles qu'une conjonction de prédicats en logique du premier ordre, un arbre « parent-enfant », etc. Plusieurs aspects sont à prendre en compte dans le cadre de la modélisation d'une telle fonction :

1. ses performances (à l'exécution) ;
2. sa complexité de modélisation ;
3. son exactitude ;
4. qu'elle soit compréhensible par un humain.

Le premier point est nécessaire pour permettre une configuration interactive d'un modèle.

Le second signifie que le coût d'obtention de la fonction n'est pas anodin. Une fonction peut présenter les autres caractéristiques demandées mais demander un effort important (par exemple en temps de calcul, ou en effort humain).

Le troisième point est évident : la fiabilité de la fonction dépend de la justesse des résultats fournis.

Le quatrième point est utile lorsque l'expertise humaine doit intervenir pour corriger ou compléter le travail qu'un algorithme a fait pour extraire ou générer une telle fonction.

1.4 Modélisation

Les méthodes de modélisation d'une fonction de validité sont généralement l'écriture manuelle des contraintes ou l'usage d'un processus spécifique à certains modèles de variabilité, possédant des informations supplémentaires sur l'entité paramétrée.

Dans l'étude des *features models*, la modélisation est souvent un élément préalable à l'étude [4], effectuée par un expert du domaine. Cette modélisation peut être automatisée par la synthèse d'un *feature model*, à partir de l'ensemble des *features* et d'une connaissance complète des configurations valides[4] [4] ou de propositions logiques [11] qui peuvent elles-mêmes provenir de différentes sources, soit manuelles, soit minées [12].

L'approche manuelle nécessite qu'un expert du domaine analyse et modélise le système. Elle nécessite donc en travail conséquent qui augmente avec la complexité du système. De plus, cette approche est

[4] Sous forme d'une *matrice de configurations*, qui reprend l'ensemble des configurations valides d'un ensemble de *features*.

difficilement généralisable. L'approche par minage permet de réduire la quantité de travail manuel, mais elle reste peut reproductible.

Dans ce travail, nous proposons d'éviter la nécessité de l'intervention d'un expert lors de la modélisation grâce à une approche générique. Idéalement, notre approche devrait permettre de pouvoir modéliser la variabilité sans nécessiter l'intervention d'un expert pour le minage ou l'écriture de contraintes.

Pour ce faire, nous utiliserons l'apprentissage automatisé (*machine learning*) pour entraîner un classificateur. Le classificateur sera entraîné à l'aide d'un *oracle*. Un oracle est une fonction permettant de déterminer la validé d'une configuration. Grâce à celui-ci, nous pourrons entraîner un classificateur différenciant les modèles valides et non-valide.

L'approche est similaire à celle réalisée par [13], qui utilisait un oracle et un classificateur (de type « arbre de décision ») pour extraite des contraintes supplémentaires pour un *feature model* particulier. Dans ce travail, nous étendrons la recherche à d'autres types de classificateurs et avec un plus grand nombre de modèles. Nous varierons aussi différents paramètres d'apprentissage, afin

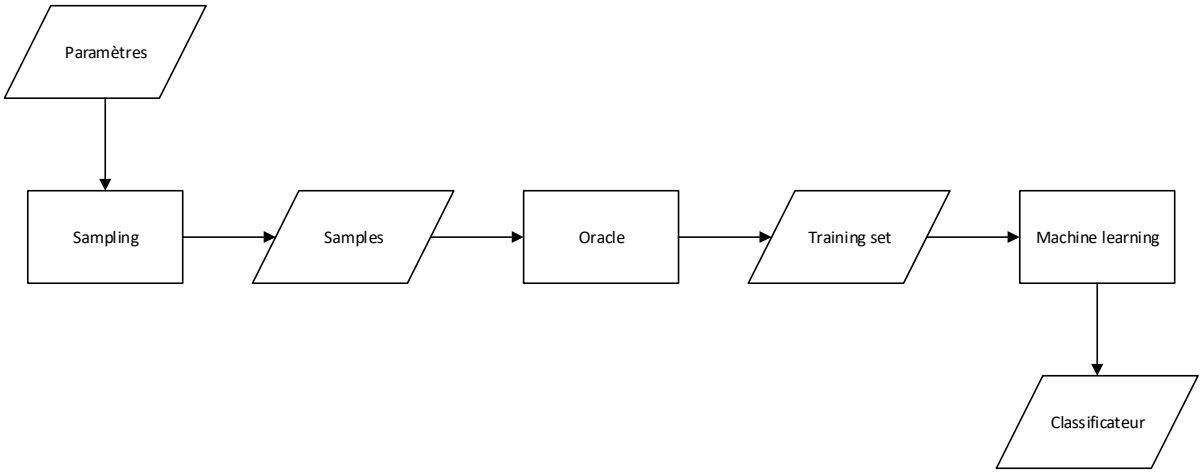


FIGURE 5: CONSTRUCTION D'UN CLASSIFICATEUR

Comme montre la Figure 5, le processus se passe en trois étapes : d'abord, un échantillonnage (*sampling)* est effectué pour générer un certain nombre de configurations du modèle. Ensuite, l'oracle est utilisé pour associé une *validité* aux configurations générées. Ces configurations sont alors utilisées pour entraîner, via un algorithme de *machine learning*, un classificateur. Un tel classificateur permettra de classifier toute configuration du modèle comme étant valide ou non.

Ce classificateur sert alors de fonction de validité. Le grand intérêt de cette méthode est que seul l'oracle est spécifique au modèle de variabilité actuel ; l'échantillonnage et l'entraînement sont des processus génériques. De ce fait, dès lors qu'on est capable de fournir un oracle, on peut obtenir un classificateur correspondant.

Notons qu'en fait, l'oracle est lui-même une fonction de validité du modèle. Notre méthode a donc besoin d'une fonction existante pour en obtenir une nouvelle. Cela étant, l'oracle ne possède pas nécessairement les caractéristiques attendues (par exemple en terme rapidité d'exécution). L'intérêt de notre méthode est de créer une fonction qui présente de meilleures caractéristiques que l'oracle.

1.5 Évaluation

Afin de valider l'intérêt de la méthode, il faudra déterminer ses gains et ses défauts. Le but recherché est de permettre une fonction de validité offrant de bonnes performances d'exécutions, des résultats corrects (i.e. correspondant à la réalité du modèle) et dont le coût d'obtention n'est pas excessif.

L'usage d'un classificateur offre les performances d'exécution suffisantes[5]. La suite de ce mémoire va donc se concentrer sur la détermination de l'exactitude de résultats et du coût de la génération. Il est à noter que ce coût dépend de deux facteurs : celui de la conception de l'oracle, et celui de l'entraînement du classificateur. Le premier étant spécifique à chaque domaine, nous nous concentrerons sur le deuxième.

1.6 Conclusion

Dans ce chapitre, nous avons introduit les notions de variabilité et de validité, les formalismes de *feature model* et d'*attributed feature model* et définit la notion de *fonction de validité*. Nous avons vu comment cette notion généralisait ces modèles et présenté notre méthode de travail pour modéliser cette fonction.

Dans la suite, nous allons introduire plus longuement la classification et donner des outils de mesure pour son évaluation.

[5] Lors des évaluations, le temps de classification d'une configuration n'a jamais dépassé 0,1ms.

2 Apprentissage automatique et classification

Dans ce chapitre, nous introduirons le concept d'apprentissage automatisé et son utilisation dans le cadre des problèmes de classification. Nous présenterons, sans entrer dans le détail, plusieurs algorithmes et structures utilisés dans la construction de classificateurs. Nous définirons ensuite des métriques essentielles à l'évaluation d'un classificateur.

2.1 Apprentissage automatique

L'apprentissage automatique (*machine learning*) réfère à la capacité d'un système de modifier son fonctionnement à partir d'informations externes [14]. Cette technologie s'est largement répandue ces dernières années et est omniprésente dans l'informatique actuelle : moteurs de recherches, filtres anti-spam, reconnaissance faciale, traitement du langage naturel, etc.

Contrairement à un programme « classique », un programme doté d'apprentissage automatique n'est pas conçu à partir d'une spécification complète des tâches qu'il devrait exécuter [15] ; au lieu de cela, ces programmes sont capables de s'adapter en fonction des données qu'ils reçoivent et des résultats attendus.

L'apprentissage automatique est utilisé pour résoudre des problèmes pour lesquels l'implémentation d'une solution classique n'est pas envisageable. Il peut s'agir de problèmes dont la spécification n'admet pas de solution algorithmique ou que l'on ne peut spécifier, de problèmes qui sont exprimés uniquement par des exemples ou de problèmes où l'on ne connait pas la forme de la solution.

On distingue deux formes d'apprentissage automatique : *supervisé* et *non-supervisé*. Dans le premier, le programme possède une connaissance préalable du résultat attendu pour certaines entrées, et tente de créer un modèle qui généralise ces données. L'apprentissage non-supervisé ne dispose quant à lui d'aucune expérience préalable.

Ce type d'apprentissage regroupe entre autre le partitionnement (*clustering*), qui tente plutôt de détecter des similarités entre les données d'entrées et de les regrouper et l'apprentissage par renforcement (*reinforcement learing*), qui apprend en tentant différentes actions et en mesurant l'impact de celles-ci.

Nous nous intéresserons ici à l'apprentissage supervisé.

2.2 Apprentissage supervisé

L'apprentissage supervisé se passe en en deux temps : d'abord, on apprend un ensemble de données à l'algorithme, en lui fournissant le résultat attendu pour chaque entrée. Ces données forment l'*ensemble d'entraînement* (training set). On appelle un élément de cet ensemble un *exemple*.

L'algorithme analyse les connaissances entrées et les résultats attendus, et *apprend* à prédire des résultats. Il peut ensuite prédire le résultat pour de nouvelles entrées, en utilisant le modèle appris.

Nous pouvons définir [16] la tâche (théorique) que l'on cherche à apprendre comme une fonction f. Pour une entrée $\boldsymbol{X}$, la sortie $f(\boldsymbol{X})$ représente le résultat théorique parfait. L'objectif de l'apprentissage est d'obtenir une fonction h, de la même forme que f (i.e. possédant le même domaine et une image compatible). Selon l'efficacité de l'apprentissage, les sorties de h se rapprocheront[6] de celles de f.

[6] Nous définirons par la suite des notions permettant de mesurer l'intérêt d'une fonction apprise.

Dans le cadre de l'apprentissage supervisé, nous possédons aussi un ensemble d'entraînement (noté Ξ) et les valeurs associées $f(\Xi)$. On suppose que si on trouve une fonction h suffisamment proche de f pour les valeurs de Ξ, alors on possède une bonne approximation de f [16].

Il est à noter qu'obtenir exactement la fonction f est généralement compliqué et pas recherché par l'apprentissage automatique [17]. En fait, tenter d'obtenir une fonction h se rapprochant trop de f pour l'ensemble d'entraînement n'est pas toujours bénéfique, comme nous le verrons plus tard avec le sur-apprentissage.

Selon le type des sorties de f, il existe différentes formes d'apprentissage supervisé. La *régression* s'intéresse aux fonctions réelles et continues et la *classification,* aux fonctions à image finie.

2.3 Classification

La classification est dont un apprentissage supervisé dont l'ensemble de sorties fini. On appelle l'apprenant d'un tel système un *classificateur.* Les valeurs de sortie d'un classificateur sont appelées *classes*.

Nous distinguons deux types de classificateurs : *binaires* et *multinomiaux* (ou multi-classe). Les premiers sont capables uniquement de distinguer deux classes, tandis que les suivants peuvent en distinguer un plus grand nombre. En pratique, il est cependant possible de construire un classificateur multinomial à partir de n'importe quel classificateur binaire.

Dans la pratique, les classificateurs ne fonctionnent pas nécessairement bien avec des entrées « brutes », surtout lorsque chaque entrée possède un grand nombre d'informations (comme des documents, des images, etc.). Les classificateurs travaillent sur un espace vectoriel, dont les éléments sont des réels ou des naturels [7]. Par la suite, nous considérerons une entrée comme un vecteur d'éléments réels ou dénombrables. Dans la littérature, ces éléments sont appelés *features* ou *attributs*.

L'*extraction de connaissance* (*feature extraction*) est un processus préalable par lequel les données brutes sont transformées en données utilisables par l'algorithme d'apprentissage. Pour des données complexes, l'extraction est une étape importante, et la qualité des connaissances extraites affectera fortement la justesse de la classification. Dans notre cas, cette étape sera triviale, car nos attributs sont déjà des nombres réels ou nominaux.

La Figure 6 présente le principe général d'un classificateur. Dans un premier temps, un classificateur h est entraîné à partir des connaissances extraites de l'ensemble d'entraînement. Celui-ci est ensuite utilisé pour classifier d'autres données. Notons qu'en pratique, plutôt qu'une fonction, l'entraînement d'un classificateur produit un ensemble de données, qui est fourni à un algorithme de prédiction. Ainsi, on peut considérer un classificateur comme la combinaison d'un algorithme d'apprentissage et d'un algorithme de prédiction. La sortie du premier étant utilisée par le second pour classifier les données.

[7] Tout ensemble fini ou dénombrable pouvant être associé à un naturel, les attributs nominaux entrent dans cette catégorie.

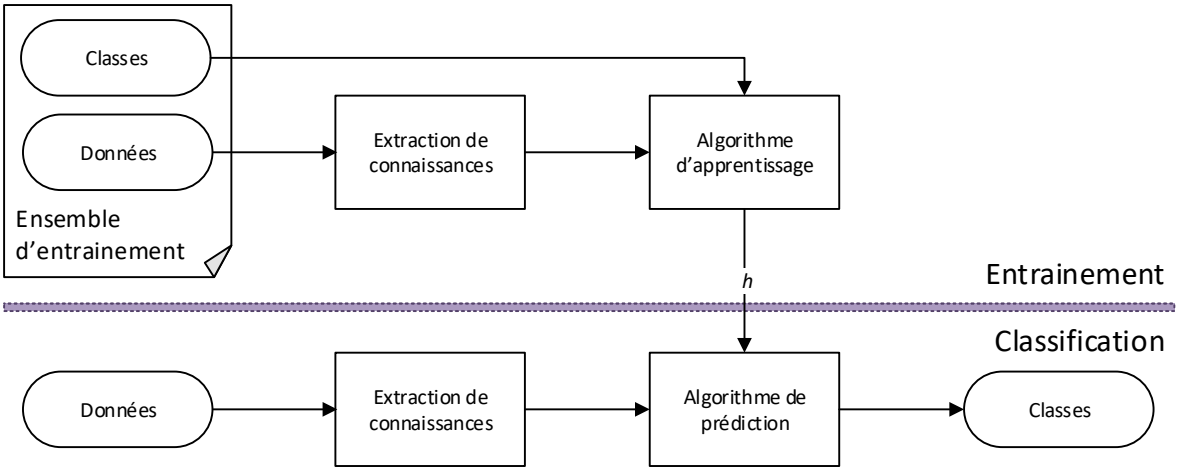


FIGURE 6:PRINCIPE D'UN CLASSIFICATEUR

Il existe différentes familles d'algorithmes de classifications, selon leur modèle sous-jacent. Nous étudierons ici les principaux : réseaux de neurones, arbres de décision, naive Bayes, classification par règles, méthodes des plus proche voisins, classificateurs ensemblistes et machines à vecteur de support.

2.3.1 Les arbres de décision

Ces classificateurs [18] sont une famille d'algorithmes utilisant un *arbre de décision*. Ceux-ci sont des arbres dont les feuilles sont des classes. Chaque nœud est associé à un **ensemble**[8] d'entrées : pour la racine, il s'agit du domaine d'entrée ; pour les autres nœuds, les ensembles de tous les enfants d'un même parent forment une partition de l'ensemble du parent.

Étant donné une entrée, on peut parcourir l'arbre en sélectionnant à chaque fois l'enfant qui contient l'entrée. Dans la pratique, le partitionnement ce fait via un critère de choix : une fonction *de choix* est associée à chaque nœud, et à chaque valeur de l'image de cette fonction est associé un enfant. Souvent, le critère de choix est :

- Un prédicat sur un élément du vecteur d'entrée (par exemple une comparaison avec une constante), avec deux enfants séparant ;
- La sélection d'un élément fini du vecteur d'entrée, avec un enfant pour chaque valeur possible.

La Figure 7 montre un exemple d'arbre de décision. Cet arbre correspond à un ensemble d'entrée dont les éléments sont de vecteurs $(\text{shape}, \text{inside\_diameter}) \in \{\text{cyinder}, \text{cube}, \text{triangle}\} \times \mathbb{R}^+$ et dont les classes sont VALID et INVALID. Par exemple, cet arbre classifie $(\text{cylinder}, 35)$ comme INVALID, tandis que $(\text{cylinder}, 50)$ et $(\text{cube}, 10)$ sont VALID.

[8] Nous dirons qu'une entrée appartient au nœud si elle appartient à l'ensemble associé

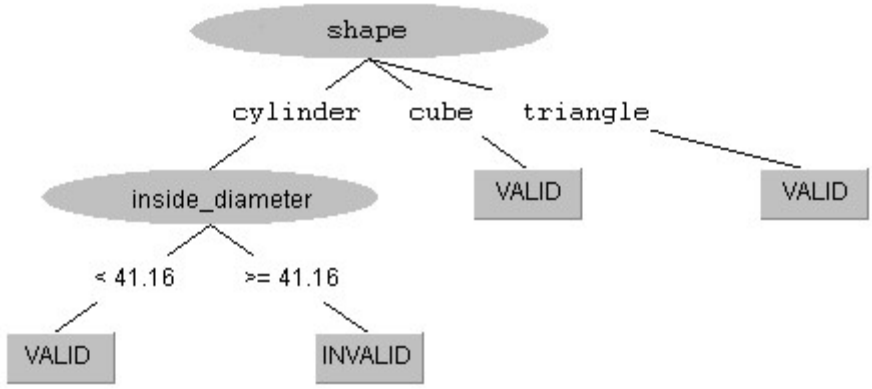


FIGURE 7: EXEMPLE D'ARBRE DE DECISION

Tous les classificateurs de type « arbre de décision » se basent sur cette structure. La principale différence entre eux est l'algorithme d'entraînement qui doit construire un tel arbre. En fait, construire un arbre de décision est un problème complexe [19].

Une manière simple de construire un arbre de décision sur des éléments fini est de manière récursive [20] : en partant d'un arbre d'un seul élément, on découpe celui-ci en plusieurs enfants, selon un *critère de division*. Pour chaque enfant, on choisit un nouveau critère de division, produisant de nouveaux enfants, jusqu'à remplir une *condition d'arrêt*.

L'algorithme *ID3* [20] fonctionne sur ce principe, sélectionnant comme critère de division un attribut, de manière à maximiser le gain d'*information*. L'information est la mesure de la quantité de données nécessaire pour déterminer la classe d'un élément appartenant au nœud. L'algorithme s'arrête lorsque le gain d'information n'est pas assez important.

Cet algorithme est très simple et comporte de nombreuses limitations[9], mais il sert de base pour des algorithmes plus complexes.

Sans entrer dans les détails, présentons brièvement les algorithmes utilisés par la suite : « Hoeffding Tree », « C4.5 (J48) », « REP Tree » et « Random Tree ».

2.3.1.1 REP Tree

Reduced Error Pruning (REP) Tree est un algorithme d'apprentissage d'arbre de décision basé sur le principe du calcul du gain d'information. REP Tree génère plusieurs arbres de décision de régression et sélectionne le meilleur, en utilisant la variance et le gain d'information et élague[10] l'arbre en utilisant la méthode par erreur réduite (*reduced-error pruning*) [21].

[9] Par exemple, il ne gère pas cas de valeurs manquantes dans l'ensemble d'entraînement, ou les paramètres réels.

[10] L'élagage [19] est une méthode consistant à retirer (simplifier) des branches d'un arbre qui fournissent peu d'information, afin de réduire la taille de l'arbre et diminuer le risque de sur-apprentissage.

2.3.1.2 C4.5 (J48)

Il s'agit d'un algorithme basé sur *ID3*, mais apportant des améliorations permettant de gérer le bruit dans les données, les valeurs manquantes, les attributs numériques et l'élagage [20]. J48 est le nom de l'implémentation libre de cet algorithme dans Weka, dont nous reparlons par la suite.

2.3.1.3 Hoeffding Tree

Hoeffding Tree (aussi appelé VFDT) est un algorithme incrémental et inductif d'apprentissage d'arbre de décision [22] [23], ce qui signifie qu'il peut être construit et mis à jour par l'ajout successif de données et ne nécessite pas de posséder l'entièreté des données d'entraînement en un seul ensemble. Cette caractéristique permet de travailler avec de larges ensembles de données, même avec des ressources limitées.

2.3.1.4 Random Tree

Pour chaque nœud, cet algorithme sélectionne aléatoirement k attributs, sur lesquels il exécute sa découpe.

2.3.2 Les règles

Les règles forment un type de classificateur similaires aux arbres de décision : la structure de donnée est un ensemble de règles, chacune composée d'un antécédent –un prédicat sur les entrées– et d'un conséquent –la classe associée. Tout élément passant le prédicat d'une règle est classifié selon sa classe [24]. L'ensemble des prédicats est supposé complet et non-conflictuel[11].

Les règles d'un tel classificateur peuvent prendre la forme d'une liste de décision (*decision list*). Celle-ci est aussi formée de prédicats et de classes, mais contrairement aux règles, elle est ordonnée et chaque prédicat nie implicitement tous les prédicats précédents ; de cette manière, pour classifier un élément, on parcourt les règles dans l'ordre et sélectionne la première règle dont le prédicat est passé. La classe de la règle sélectionne est retournée.

La Figure 8 montre un exemple de règles, sous forme de liste de décision. Le classificateur représenté est équivalent à celui de la Figure 7.

`shape = cylinder and inside_diameter >=41.16`	`INVALID`
`true`	`VALID`

FIGURE 8: EXEMPLE DE REGLES

2.3.2.1 RIPPER

Repeated Incremental Pruning to Produce Error Reduction (RIPPER) est un algorithme d'extraction de règles, fonctionnant en générant des règles par ajout successif de condition, suivi d'un élagage pour retirer les éléments utilisés. La génération est suivie par une étape d'optimisation, visant à minimiser la taille des règles. Générations et optimisations successives se déroulent jusqu'à épuiser les exemples ou les possibilités de génération [25].

2.3.2.2 PART decision list

Cet algorithme crée une liste de décision en générant un arbre de décision utilisant C4.5, puis en sélectionnant une feuille de l'arbre couvrant le plus d'éléments de l'ensemble d'entraînement. Une règle est construite à partir de la conjonction des prédicats des nœuds successifs menant à la feuille sélectionnée,

[11] C'est-à-dire qu'un élément du domaine est toujours sélectionné par au moins une règle et qu'il n'est pas sélectionné par plusieurs règles ayant des classes différentes.

et de la classe de celle-ci. Les éléments couverts sont retirés de l'ensemble d'entraînement et l'étape est répétée [26].

2.3.3 Naïve Bayes

Naïve Bayes [27] est un classificateur probabiliste, basé sur le théorème de Bayes [28]. Celui-ci dit que, étant donnée une hypothèse A et un fait B, on a[12]

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)},$$

C'est-à-dire que la probabilité qu'un événement A se produise, sachant que B c'est produit. Utilisant ce théorème, nous pouvons écrire la probabilité d'obtenir une classe C_k pour une valeur d'entrée $\boldsymbol{X}$

$$P(C_k|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C).P(C_k)}{P(\boldsymbol{X})}.$$

Naïve Bayes fait l'hypothèse –naïve– que les différents éléments de $\boldsymbol{X}$ sont des événements **indépendants**. Ainsi, si $\boldsymbol{X} = (x_1, \dots, x_n)$, alors

$$P(C_k|\boldsymbol{X}) = \frac{P(C_k)}{P(\boldsymbol{X})} \prod_{i=1}^{n} P(C_k|x_i).$$

Dès lors, si nous connaissons la probabilité de chaque classe en fonction de chaque valeur d'un attribut (c'est-à-dire tous les $P(C_k|x_i)$) et les probabilités de chaque classe (c'est-à-dire $P(C_k)$), nous pouvons définir un classificateur grâce au critère de vraisemblance maximum : nous sélectionnons la classe qui a le plus grand $P(C|\boldsymbol{X})$[13].

Pour construire ce classificateur, on peut calculer les $P(C_k)$ via leurs fréquences dans l'ensemble d'entraînement, ou les considérer comme équiprobable. Pour obtenir les $P(C_k|x_i)$, il faut modéliser une distribution de cette probabilité à partir des données d'entraînement. Il existe différentes techniques (Gaussian, Multinomial, Benouilli, etc) pour gérer les attributs discrets ou numériques.

2.3.4 Modèle linéaire

Un modèle linéaire [29] utilise la somme pondérée des éléments du vecteur d'entrée. Dans le cadre de la classification, un modèle linéaire prend la forme d'une équation suivante, étant donné un vecteur d'entrée $\boldsymbol{X} = (x_1, \dots, x_n)$:

$$a_0 + \sum_{i=1}^{n} a_i x_i \geq 0,$$

Avec $a_1, \dots, a_n, b$ les paramètres du modèle. Nous noterons que $a_0 + \sum_{i=1}^{n} a_i x_i = 0$ forme un hyperplan de l'espace vectoriel $\mathbb{R}^n$; il s'agit de la frontière de décision (*decision boundary*) qui sépare les éléments des deux classes. La Figure 9 montre un exemple d'un tel modèle linéaire.

[12] $P(A)$ est la probabilité qu'un événement A ce produise, et $P(A|B)$ la probabilité conditionnelle qu'un événement A se produise, sachant que B c'est produit.

[13] Comme $P(\boldsymbol{X})$ est indépendant de C_k, nous pouvons le simplifier pour les comparaisons

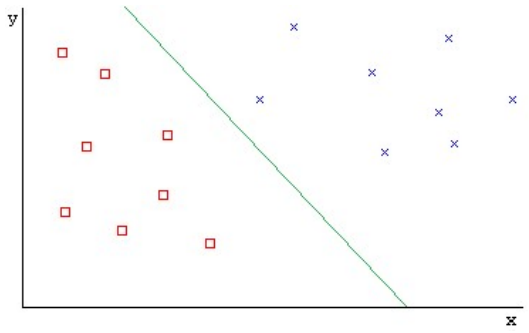


FIGURE 9:EXEMPLE DEN MODELE LINEAIRE SUR UN ESPACE VECTORIEL A DEUX DIMENSIONS. EN ROUGE ET EN BLEU : LES ELEMENTS DE L'ENSEMBLE D'ENTRAINEMENT, AVEC LA COULEUR SELON LEUR CLASSE ;EN VERT : LA FRONTIERE DE DECISION.

Le modèle linéaire est utilisé dans le cadre de la *régression linéaire*. Comme dit précédemment, la régression est une forme d'apprentissage supervisé qui construit une fonction réelle. La régression linéaire permet d'obtenir une fonction linéaire, c'est-à-dire du type $f(\boldsymbol{X}) = a_0 + \sum_{i=1}^{n} a_i x_i$.

2.3.4.1 Régression logistique

Pour obtenir un classificateur à partir de la régression linéaire, nous pouvons l'appliquer à la probabilité d'obtenir la classe C_k en fonction d'une entrée $\boldsymbol{X}$. La classification revient alors à calculer la probabilité de chaque classe et à sélectionner la plus probable.

Cela étant, la régression linéaire retourne des valeurs entre $-\infty$ et $+\infty$, alors qu'une probabilité est entre 0 et 1. Pour remédier à cela, on applique à $P(C_k|\boldsymbol{X})$ la transformation « logit », qui transforme les valeurs entre 0 et 1 en valeurs allant de $-\infty$ à $+\infty$, comme le montre la Figure 10. On peut approximer cette fonction par une fonction linéaire, puis appliquer la transformation inverse. Nous obtenons ainsi une fonction paramétrée par $a_0, \dots, a_n$.

Dans la régression linéaire, ces valeurs sont calculées en minimisant le carré de l'erreur entre les résultats attendus et calculés pour l'ensemble des exemples. Dans ce cas, le critère utilisé est la fonction de vraisemblance logarithmique (*log-likelihood*).

Le classificateur obtenu est la régression logistique.

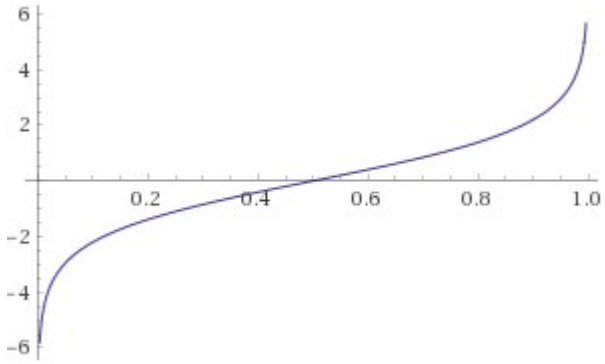


FIGURE 10: TRANSFORMATION « LOGIT »

2.3.5 Classificateurs « instance-based »

Ce type de classificateur [30] utilise directement les entrées de l'ensemble d'entraînement, ainsi qu'une mesure de similarité. Pour classifier un élément, on détecte les exemples les plus similaires à celui-ci et on sélectionne la classe en fonction de la sélection.

Selon l'algorithme, il peut s'agit de la classe la plus représentée, en tenant compte ou non de la similarité des éléments.

2.3.5.1 kNN

L'algorithme « k Nearest Neighbours » (k plus proche voisins – kNN) utilise la distance comme mesure de similarité : étant donné une entrée, on sélectionne la classe la plus représentée dans ses k plus proches voisins (k étant une valeur fournie).

2.3.5.2 K*

Cet algorithme est similaire à *kNN*, mais utilise l'entropie comme distance. Celle-ci est décrite comme la complexité pour transformer un élément en un autre [31].

2.3.6 Les algorithmes ensemblistes

Ces algorithmes sont une sorte de méta-classificateurs : ils entraînent une série d'autres classificateurs et combinent leurs résultats. La *randomization* [32] est un type de classificateur qui entraîne différentes instances de classificateurs en faisant varier certains de leurs paramètres ; *Random Forest* et *Random Committee* en sont deux exemples.

2.3.6.1 Random Committee

Cet algorithme [33] utilise un ensemble de classificateurs « *randomizables* ». Il s'agit de tout classificateur utilisant des valeurs aléatoires lors de son entraînement ; en faisant varier la *seed* [14] d'un tel classificateur, il génère des valeurs différentes. *Random Committee* instancie plusieurs classificateurs avec des *seed* différentes, et les entraîne avec le même ensemble. Pour classifier, il prend simplement la moyenne des résultats de ses classificateurs.

Nous utiliserons *Random* Committee avec comme algorithme sous-jacent *Random Tree*.

2.3.6.2 Random Forest

Cet algorithme construit un ensemble d'arbre de décisions, entraînés par des parties de l'ensemble d'entrainement et utilise la moyenne des réponses. Cette façon d'utiliser plusieurs classificateurs identiques avec des ensembles d'entrainement différents s'appelle *bagging*. En utilisant des *Random Tree* comme classificateurs sous-jacent, *Random Forest* combine *bagging* et *randomization*.

2.3.6.3 Logistic Model Tree

Cet algorithme [34] est un autre type d'algorithme ensembliste. Il combine deux types de classificateurs : le modèle linéaire et les arbres de décision. Il s'agit en fait d'un arbre de décision pour lequel chaque feuille possède un classificateur plutôt qu'une classe. Ce classificateur est de type *logistic regression*.

La classification s'effectue en parcourant l'arbre de décision normalement, puis en utilisant le classificateur de la feuille atteinte. La construction de l'arbre de décision s'effectue de manière à scinder les nœuds tant

[14] Paramètre utilisé par les générateurs de nombre aléatoire.

qu'il n'est pas possible d'obtenir une régression logistique pertinence. L'algorithme utilisé pour la génération de l'arbre est C4.5, avec un critère de division adapté.

2.3.7 SVM

Le modèle linéaire, que nous avons introduit plus haut, possède un inconvénient majeur : il ne peut représenter que des frontières de décision linéaires (i.e. des hyperplans), ce qui est très limité. Par exemple, dans la Figure 11, il est impossible de construire une frontière linéaire séparant complétement les éléments des deux classes.

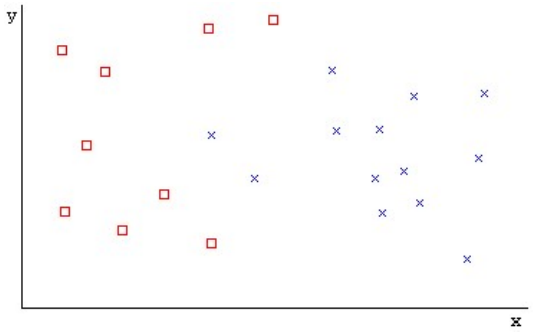


FIGURE 11: EXEMPLE DE MODELE LINEAIRE SUR UN ESPACE VECTORIEL A DEUX DIMENSIONS. EN ROUGE ET EN BLEU : LES ELEMENTS DE L'ENSEMBLE D'ENTRAINEMENT, AVEC LA COULEUR SELON LEUR CLASSE.

Les *Support Vector Machines* (Machines à vecteur de support – SVM) [35] apportent une solution. Ils combinent un modèle linéaire à une transformation non-linéaire : leur *noyau* (*kernel*). Ce noyau transforme les valeurs d'entrée dans un espace vectoriel de même dimension. En appliquant un modèle vectoriel à cet espace, on obtient des frontières de décision. Celles-ci sont linéaires dans cet espaces, mais pas dans l'espace des valeurs d'entrées.

La Figure 12 montre un exemple de la transformation accomplie par un tel noyau. Dans l'exemple, cette transformation a permis de placer les éléments dans un espace vectoriel dans lequel une frontière de séparation linéaire existe. Pour classifier ces éléments, il suffit donc de les transformer, via le noyau, puis de les classifier via un classificateur linéaire.

L'algorithme des SVM tente de sélectionner la meilleure frontière de séparation, c'est-à-dire la plus éloignée de tout élément[15]. On nomme les points les plus proches de cette frontière *les vecteurs de support* (*support vector*). La Figure 12 montre une telle frontière, où les points touchant les lignes en pointillé sont les vecteurs de support.

[15] Aussi appelée *maximum-margin hyperplane*

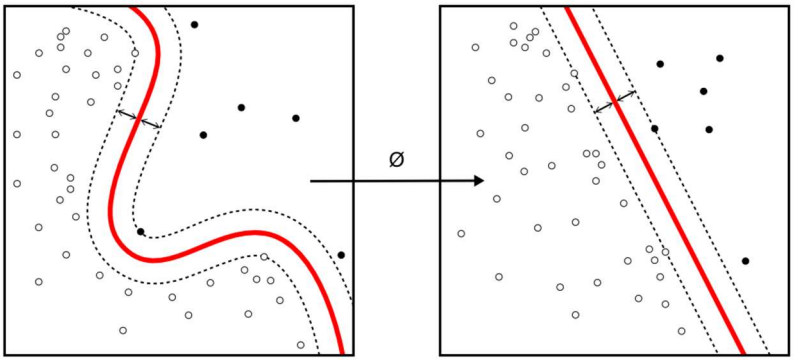


FIGURE 12: EXEMPLE [36] DE LA TRANSFORMATION ACCOMPLIE PAR UN NOYAU. À GAUCHE, L'ESPACE D'ENTREE ET A DROITE, L'ESPACE LINEARISE. LA FRONTIERE EST DENOTEE EN ROUGE ET LES POINT-TILLES INDIQUENT LA DISTANCE MINIMALE.

Il existe plusieurs types de noyaux utilisés avec les SVM ; les plus communs sont les noyaux polynomiaux, les *Radius Based Function* (RBF) et les sigmoïdes.

2.3.8 Les réseaux de neurones

Les réseaux de neurones artificiels [37] sont des algorithmes imitant le fonctionnement des neurones. Un neurone artificiel est constitué d'un certain nombre d'entrées, ayant chacune un poids, d'une fonction d'activation (aussi appelée fonction de transfert) et d'une sortie.

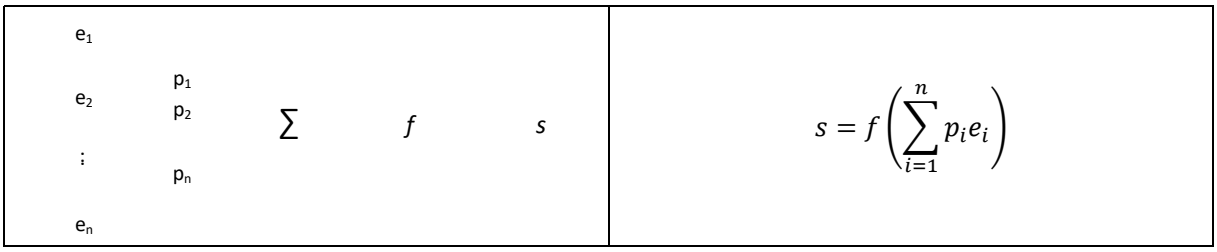
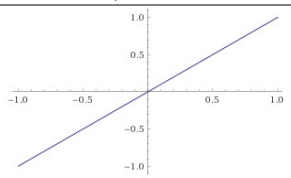
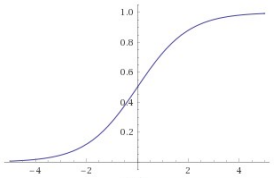
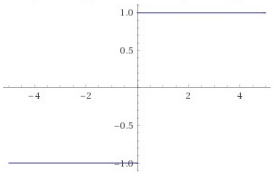


FIGURE 13: SCHEMA D'UN NEURONE ARTIFICIEL (A GAUCHE) ET SA VALEUR DE SORTIE (A DROITE)

La Figure 13 montre un exemple de neurones, avec ses entrées $e_1, \dots, e_n$, leurs poids $p_1, \dots, p_n$, une fonction d'activation f et une sortie s. La relation entre ces valeurs est donnée par l'équation à droite. Le Tableau 2 donne la liste des fonctions d'activation les plus courantes.

TABLEAU 2: PRINCIPALES FONCTIONS D'ACTIVATION

FONCTION D'ACTIVATION	FORMULE	GRAPHIQUE
LINEAIRE	$f(x) = x$	
SIGMOÏDE	$f(x) = \frac{1}{1+e^{-x}}$	
SEUIL	$f(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$	

Un réseau de neurones artificiels est constitué d'un ensemble de neurones artificiels, dont les sorties sont reliées aux entrées d'autres neurones. Le réseau de neurone possède des entrées, qui sont connectées aux entrées de certaines neurones et des sorties, qui sont la sortie d'un neurone.

Un réseau de neurones à propagation avant (*feed forward neural network* – FFNN) est un réseau de neurones où les nœuds ne forment pas de boucle. Généralement, ces réseaux sont organisés en couches : les sorties des neurones d'une couche sont reliées aux entrées de la suivante. La dernière couche, dont les neurones sont les sorties du réseau, est appelée couche de sortie ; toutes les couches intermédiaires sont appelées les couches cachées. Nous appelons la couche d'entrée l'ensemble des entrées du réseau de neurones.

2.3.8.1 Multilayer Perceptron

Cet algorithme [38] entraîne un réseau de neurones à propagation avant. La couche d'entrées associe une entrée à chaque attribut. La couche de sortie possède un neurone par classe. Les sorties d'une couche sont toujours toutes connectées à chaque neurone de la couche suivante. Les fonctions d'activations sont des sigmoïdes.

La classification s'opère en entrant les valeurs des attributs en entrée et en récupérant la valeur de chaque sortie. La classe associée à sortie avec la plus grande valeur est sélectionnée.

L'entrainement d'un perceptron multicouche consiste à calculer les poids des entrées des neurones de manière à minimiser le carré de l'erreur entre les valeurs attendues et calculées. Ce processus se fait de manière itérative via un algorithme de rétro-propagation (*backpropagation*).

2.4 Compréhensibilité des classificateurs

Nous avions déjà mentionné qu'une caractéristique importante de la fonction de validité est sa compréhensibilité. L'intérêt d'une fonction compréhensible est qu'elle peut être appréhendée et altérée par un utilisateur humain. Ceci pour permettre de rectifier des erreurs introduites par la génération de la fonction ou l'inexactitude de l'oracle.

Lorsque notre fonction est un classificateur, sa compréhensibilité dépend fortement du modèle utilisé. Pour la suite, sera utilisé comme exemple le modèle S.P.L.O.T. fourni dans l'Annexe B : Modèle S.P.L.O.T. « télécom ».

2.4.1 Règles

Les algorithmes d'apprentissage comme RIPPER et PART se basent sur l'extraction de règles. Ils génèrent une liste de règles qui peuvent valider ou invalider certaines configurations. Une configuration est classifiée par la première règle qu'elle remplit (les règles sont ordonnées). Chaque règle associe un prédicat logique (généralement une conjonction de comparaisons entre une valeur et un attribut) à une classe. Dans le cas de la classification des modèles S.P.L.O.T., les prédicats testent toujours la présence ou l'absence d'une *feature*, et y associent la validité ou l'invalidité du modèle. Les Figure 46 et Figure 47 (voir Annexe B : Modèle S.P.L.O.T. « télécom ») montrent un exemple de règles générées par ces algorithmes.

Ces règles sont facilement compréhensibles et sont presque immédiatement utilisables. Attention toutefois que ces règles sont ordonnées et que chaque règle nie donc implicitement toutes les précédentes (c'est-à-dire qu'une règle ne se vérifie que si toutes les précédentes sont fausses).

2.4.2 Arbres de décisions

Les algorithmes tels que *C4.5 (J48)*, *Hoeffding Tree*, *Logistic Model Tree* et *REP Tree* génèrent un arbre de décision. Cet arbre est constitué d'un ensemble de nœuds. Les nœuds parents ont une condition sur un attribut du modèle. Les enfants d'un nœud correspondent aux différentes valeurs de vérités de la condition ; les feuilles de l'arbre sont associées à une classe.

Dans le cadre des modèles S.P.L.OT, cela signifie que chaque nœud intermédiaire est associé à une *feature*, et qu'il possède deux enfants, représentant la présence ou l'absence de la *feature*. La Figure 48 (Annexe B : Modèle S.P.L.O.T. « télécom ») montre le résultat obtenu par l'algorithme *REP Tree* sur le modèle d'exemple.

Les arbres de décision sont, comme les règles, facilement compréhensibles par un utilisateur non-expert. Toutefois, la taille d'un arbre de décision peut être très importante.

2.4.3 Algorithmes ensemblistes

Random Committee et *Random Forest* génèrent plusieurs arbres de décisions sur des sous-ensembles d'attributs ou des variations de l'ensemble d'entraînement et classifient en sélectionnant une classe parmi les résultats des classifications individuelles faites par leurs sous arbres. Cette sélection s'effectue en prenant la classe la plus représentée dans les résultats individuels.

Comme les sous arbres ne représentent pas le modèle, mais des variations de celui-ci, leur lecture individuelle n'offre pas d'information complète sur la validité du modèle. Ces classificateurs sont donc difficilement interprétables par un humain.

2.4.4 Autres algorithmes

Les autres algorithmes ne génèrent pas de modèle facilement interprétable ou manipulable :

- Les *SVM* génèrent une combinaison de termes pondérés dans l'espace vectoriel défini par leur noyau. Cette combinaison est utilisée comme discriminant pour la classification.
- *kNN* et *K** se basent sur le calcul de la similarité avec les éléments connus.
- *Naive Bayes* et *Logistic Regression* construisent un modèle statistique en fonction de l'ensemble d'entraînement et évaluent ensuite la probabilité d'un élément d'appartenir à une classe.

Dans l'absolu, tous ces algorithmes sont basés sur des espaces vectoriels de grande dimension (généralement le nombre d'attributs du classificateur) et classifient en effectuant des opérations mathématiques plus ou moins complexes sur le vecteur correspondant à l'instance à classifier dans cet espace vectoriel. Le résultat de ces opérations est utilisé pour décider de la classe à sélectionner.

Enfin, *Multilayer Perceptron* construit un réseau de neurones. Les résultats de l'entraînement sont les poids attribués aux nœuds et le seuil d'activation.

2.5 WEKA

L'ensemble des algorithmes que nous avons présenté est largement utilisés dans la recherche et l'industrie dans de nombreux problèmes de classifications. Un certain nombre d'outils [39] [40] fournissent une implémentation clé-en-main de de ces algorithmes.

Nous utiliserons dans la suite de ce travail WEKA [40]. IL s'agit d'un projet libre, développé en Java, fournissant un grand nombre d'algorithmes de classification et de régression, ainsi que des outils graphique pour l'analyse des résultats.

2.6 Évaluation

L'évaluation d'un algorithme de classification ce fait par la validation d'un *ensemble de test* (test set). Il s'agit d'un ensemble similaire à celui d'entrainement, contenant des données et leur valeur attendue. L'ensemble de test est entré dans le classifier, qui retourne un ensemble de résultats. Ces résultats obtenus sont comparés aux résultats attendus, afin d'obtenir une évaluation du classificateur.

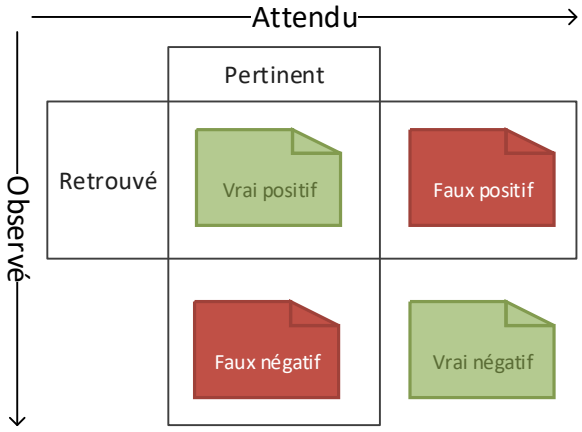
Il est clair que la taille de l'ensemble de test jouera sur la qualité de l'évaluation. Un ensemble trop petit fournira des résultats biaisés. Pareillement, si l'ensemble de test n'est pas représentatif des documents classifiés, l'évaluation ne sera pas probante.

2.6.1 Erreurs

Pour chaque donnée classifiée de l'ensemble de test, le résultat peut donc être vrai ou faux. De même, le résultat attendu dans l'ensemble de test peut aussi être vrai ou faux. Dès lors, on distingue quatre cas possibles :

- vrai positif : le résultat attendu est vrai et le résultat observé aussi,
- faux négatif : le résultat attendu est faux et le résultat observé aussi,
- faux positif : le résultat attendu est faux, mais le résultat observé est vrai (erreur de type 1),
- faux négatif : le résultat attendu est vrai, mais le résultat observé est faux (erreur de type 2).

TABLEAU 3 : RESULTATS POSSIBLES D'UNE COMPARAISON



2.6.2 Métriques

Avec ces résultats, nous pouvons définir les métriques suivantes :

- La *précision* est la proportion de documents retrouvés qui sont pertinents :

$$\text{precision} = \frac{\text{vrais positifs}}{\text{retrouvés}}$$

- Le *rappel* (*recall*) et proportion de documents pertinents qui ont été retrouvés :

$$\text{rappel} = \frac{\text{vrais positifs}}{\text{pertinents}}$$

- Le F_1 Score (aussi appelé *F-measure* ou *F-Score*) est la moyenne harmonique de la précision et du rappel :

$$F_1\text{Score} = \frac{2\ \text{precision}\ .\ \text{rappel}}{\text{precision} + \text{rappel}}$$

La précision nous donne une idée de la fiabilité des résultats retrouvés, le rappel mesure la proportion d'oublis commis par le classificateur. Le F_1 Score prend les fournit un résultat combinant les deux informations.

Les métriques suivantes sont moins utilisées, mais sont

- L'exactitude (*accuracy*) est la proportion de résultats où l'observé égale l'attendu :

$$\text{exactitude} = \frac{\text{vrai positifs} + \text{vrai négatifs}}{\text{total}}$$

- La spécificité est la proportion de de résultats négatifs parmi les négatifs :

$$\text{spécificité} = \frac{\text{vrai négatifs}}{\text{vrai négatifs} + \text{faux positifs}}$$

2.6.3 Évaluation croisée

Le principal inconvénient de l'évaluation est qu'il faut posséder un ensemble de test. Lorsqu'on travaille sur des données sans référence, cela signifie qu'une personne doit faire le travail de classification préalable manuellement.

Comme c'est aussi le cas de l'ensemble d'entrainement, et comme les deux ensembles doivent être disjoints (sinon l'évaluation sera biaisée), cela signifie qu'il faut trier le double de résultats. Pour éviter cela, on procède à une *évaluation croisée*.

Pour procéder à une telle évaluation, nous utilisons un unique ensemble d'entrainement et de test Tr. Nous divisons cet ensemble en n parties $Tr_1, \dots, Tr_n$, autant que possible « bien réparties ». Soit notre fonction d'évaluation $E(C, T)$ d'un classificateur C et d'un ensemble de test T, et soit C_{Tr} notre classificateur entrainé avec l'ensemble Tr.

Nous définissons l'évaluation croisée de C et Tr telle que

$$CE(C_\circ, Tr) = \sum_{i=1}^{n} E\left(C_{Tr \setminus Tr_i}, Tr_i\right)$$

C'est-à-dire que nous entrainons l'ensemble d'entrainement avec $n-1$ parties de l'ensemble d'entrainement et testons avec la partie restante. Nous effectuons l'opération pour chaque partie, et sommons les résultats (vrai positifs, faux positifs, vrai négatifs et faux négatifs).

Nous pouvons ensuite calculer les métriques définies dans sur le résultat final.

2.7 Conclusion

Dans ce chapitre, nous avons présenté les notions d'apprentissage automatisé et de classification. Nous avons présenté un certain nombre d'algorithmes de classification : réseaux de neurones, arbres de décision, algorithme de naive Bayes, classification par règles, méthodes des plus proche voisins, classificateurs ensemblistes et machines à vecteur de support. Nous avons aussi discuté de la lisibilité des modèles obtenus.

Nous avons ensuite défini la notion d'erreur dans un problème de classification et des métriques permettant d'évaluer les performances d'un classificateur. Nous avons aussi introduit WEKA. Dans la suite,

nous utiliserons WEKA et les algorithmes présentés pour construire des classificateurs et utiliserons les métriques définies pour évaluer leurs performances.

3 S.P.L.O.T.

Dans ce chapitre, nous évaluerons notre méthode dans le cadre des *features model* S.P.L.O.T. Nous commencerons par introduire ce projet, puis discuterons du protocole et de la mise en place de l'évaluation. Nous présenterons et analyserons ensuite en détails des évaluations et des résultats obtenus.

3.1 Le projet S.P.L.O.T.

Le projet S.P.L.O.T. offre un certain nombre de *features models* modélisant divers problèmes de configuration réels, ou tout du moins réalistes, à des fins de recherche et d'éducation. Nous utiliserons ceux-ci afin de pouvoir expérimenter rapidement sur un grand nombre de cas réels.

Comme les modèles fournissent leurs contraintes, on peut les solutionner pour une configuration donnée pour en déterminer sa validité. Nous avons donc une connaissance complète de la vérité du modèle, ce qui permettra de valider l'ensemble des configurations possibles et ainsi évaluer complétement la justesse des classificateurs entraînés.

3.2 Les *feature model* et S.P.L.O.T

Le projet *Software Product Line Online Tools* (S.P.L.O.T.) [9] est un ensemble d'outils permettant la création et l'analyse de *features models*, sous forme d'un site Web. Celui-ci propose entre-autre un éditeur visuel, un outil d'analyse et un répertoire de modèles existants et utilisables à des fins de recherche et éducative. Les modèles proviennent de la littérature scientifique ou ont étés publiés par la communauté des utilisateurs.

Actuellement, le site propose plus de 900 modèles créés par des humains, et plusieurs milliers de modèles générés. Ces derniers ne seront cependant pas utilisés, les modèles manuels offrant un ensemble suffisamment important pour nous besoins.

3.2.1 Format des *feature models* S.P.L.O.T.

Un *feature model* S.P.L.O.T. est définit [41] par un arbre de *features*, et des contraintes non-hiérarchiques. En outre, chaque feature possède un identifiant, un label et un type, ainsi qu'un parent, sauf la *feature* racine. Les types des *features* définissent les relations entre elles ; il en existe quatre : la *racine*, les obligatoires, les facultatives et les groupes.

Les modèles sont décrits par un fichier SXFM (Simple XML Feature Model). Ce fichier permet de décrire un arbre de *features*, des contraintes non-hiérarchiques, ainsi qu'un certain nombre de métadonnées descriptives du modèle. Un exemple de modèle est donné par la Figure 44 (voir Annexe B : Modèle S.P.L.O.T. « télécom »).

Notons que la relation parent-enfant implique qu'un enfant ne peut jamais être sélectionné lorsque son parent ne l'est pas. Une configuration violant cette contrainte est invalide. En outre, les différents types définissent d'autres contraintes.

3.2.1.1 La *feature* racine

Cette *feature* est unique et obligatoire. C'est l'unique *feature* qui n'a pas de parent. Étant donné une *feature* racine R, sa contrainte est

$$R$$

3.2.1.2 Les *features* facultatives

Ces *features* peuvent être omises. Étant donné une *feature* facultative F et son parent P, sa contrainte est

$$P \Rightarrow F$$

3.2.1.3 Les *features* obligatoires

Ces *features* doivent être présentes si leur parent l'est aussi. Étant donné une *feature* obligatoire O et son parent P, sa contrainte est

$$P \Leftrightarrow O$$

3.2.1.4 Les groupes de *feature*

Ce type de *feature* ajoute une contrainte sur ses enfants. Un groupe de *feature* possède une cardinalité minimale et optionnellement une maximale. Ces cardinalités contraignent le nombre de *features* enfant pouvant être sélectionnées dans une même configuration.

Par exemple, les pour les cardinalités minimales et maximale valant 1 (on note `[1,1]`), un seul des enfants peut être sélectionné (le groupe est une alternative). Si seule la cardinalité minimale est définie et vaut 1 (on note `[1,*]`), au moins un enfant doit être sélectionné, mais plusieurs peuvent l'être (il s'agit d'un ou inclusif).

Étant donné un groupe de *features* G, son parent P, ses cardinalités $[a, b]$ (en supposant $b = +\infty$ si la cardinalité maximale n'est pas définie) et ses enfants $E_1, \dots, E_n$, sa contrainte est

$$P \Leftrightarrow G \wedge G \Rightarrow a \leq |\{i : F_i\}| \leq b$$

3.2.1.5 Les contraintes non-hiérarchiques

Dans un modèle S.P.L.O.T., les contraintes non-hiérarchiques prennent la forme de disjonctions de *features* ou de négation de *features*. Ces contraintes sont souvent utilisées pour représentées des exclusions mutuelles ou des implications entre *features* de différentes branches.

3.3 Protocole d'évaluation

L'évaluation se passe en deux étapes. D'abord, la génération un ensemble d'entraînement (*training set*) avec pour oracle les contraintes du modèle S.P.L.O.T. et son utilisation pour entraîner un classificateur.

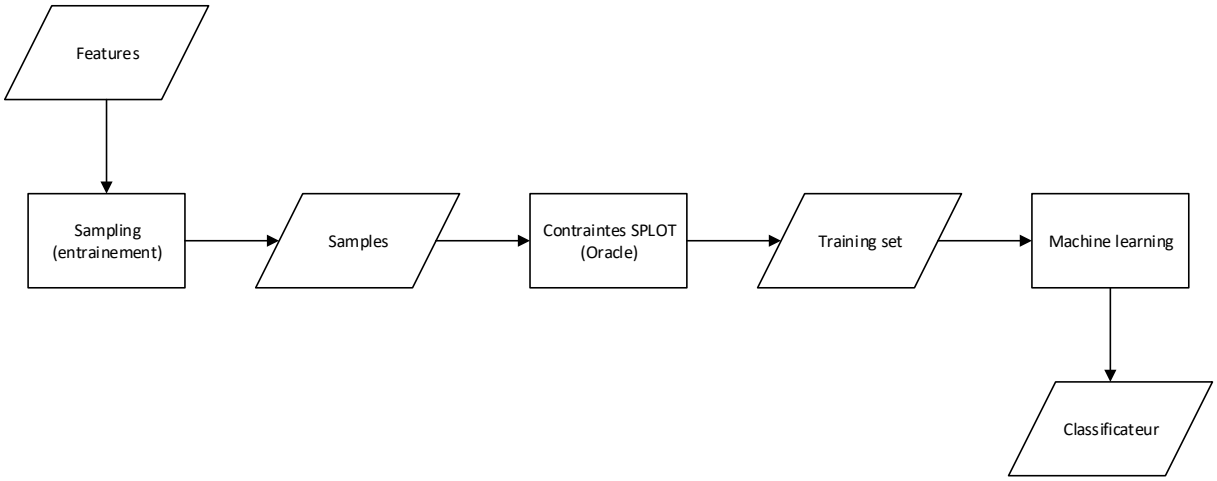


FIGURE 14: ENTRAINEMENT D'UN CLASSIFICATEUR

Ensuite, la génération d'un second ensemble, de validation, dont la validité sera vérifiée, toujours via les contraintes du modèle. La validation se fait toujours sur un échantillonnage exhaustif : l'entièreté des configurations possibles sera générée afin de valider le classificateur sur l'ensemble du modèle. Le classificateur précédemment entraîné sera testé avec chaque élément de l'ensemble de validation (*validation set*). Le résultat obtenu par le classificateur est comparé au résultat attendu (obtenu par l'oracle).

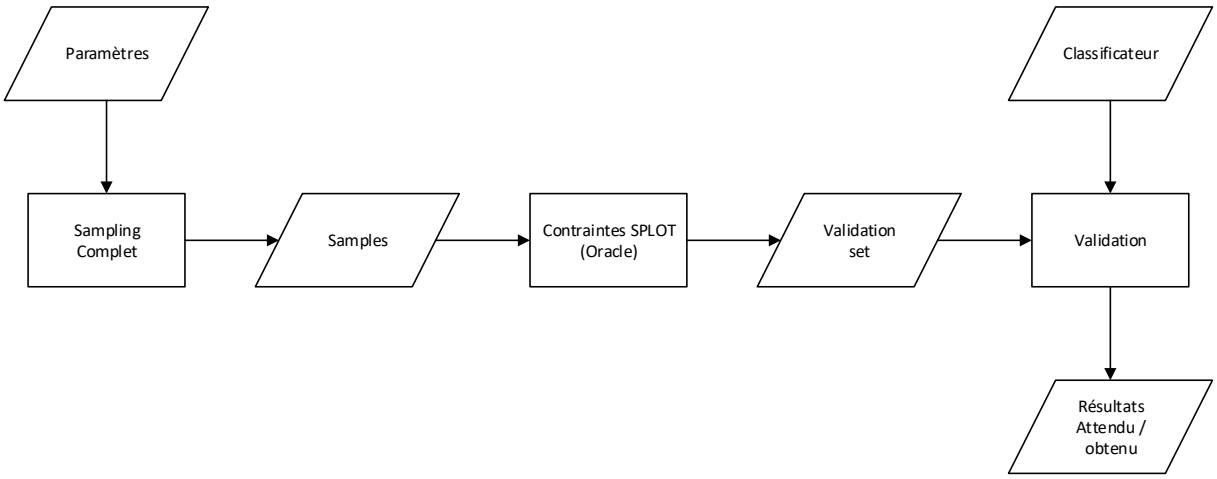


FIGURE 15: VALIDATION D'UN CLASSIFICATEUR

L'évaluation sera exécutée sur un certain nombre de modèles, avec pour critères principaux la précision, le rappel et le F_1-score obtenu pour chaque modèle.

3.3.1 Échantillonnage exhaustif

La première étape de l'évaluation sera de déterminer si un classificateur est capable de modéliser la validité de modèles S.P.L.O.T. de manière complète ou partielle. Pour ce faire, différents algorithmes de classification seront testés avec un échantillonnage exhaustif et le score obtenu par chaque algorithme sera mesuré.

3.3.2 Échantillonnage partiel

Ensuite, la capacité des algorithmes de classification à généraliser un modèle à partir d'un échantillonnage incomplet sera évaluée. Pour ce faire, un échantillonnage aléatoire sera mis en place. Celui-ci sélectionnera une partie des configurations possibles, de manière arbitraire, sans répétition.

Afin d'éviter des résultats biaisés dû à la sélection aléatoire des sous-échantillons, les tests seront répétés en utilisant plusieurs fois le même modèle et en générant des échantillons différents.

Plusieurs validations auront lieu, avec différents taux d'échantillonnage, de 90% à 10% par tranche de 10%, afin d'étudier la fiabilité des algorithmes avec des échantillonnages de plus en plus réduits, et donc de moins en moins représentatifs de la réalité du modèle.

3.3.3 Ajout de contraintes manuelles

L'approche proposée permet de modéliser la validité d'un modèle de manière générique et automatique. Cependant, il est possible que le modèle obtenu ne soit pas suffisamment précis ou sensible pour répondre à un besoin particulier. Lorsque l'approche automatique n'est pas suffisante, une intervention manuelle d'un expert est nécessaire pour écrire les contraintes du modèle. Afin de limiter la nécessité de cette intervention, nous proposons une approche hybride, combinant l'entraînement préalable d'un classificateur, et l'ajout manuel de contraintes a posteriori.

Afin de valider la viabilité de cette approche, les contraintes existantes des modèles S.P.L.O.T. seront utilisées : un classificateur sera entraîné comme précédemment, puis une partie des contraintes extraites du modèle S.P.L.O.T. lui sera adjointe. La fonction de validité ainsi obtenue sera la conjonction du classificateur et des contraintes extraites.

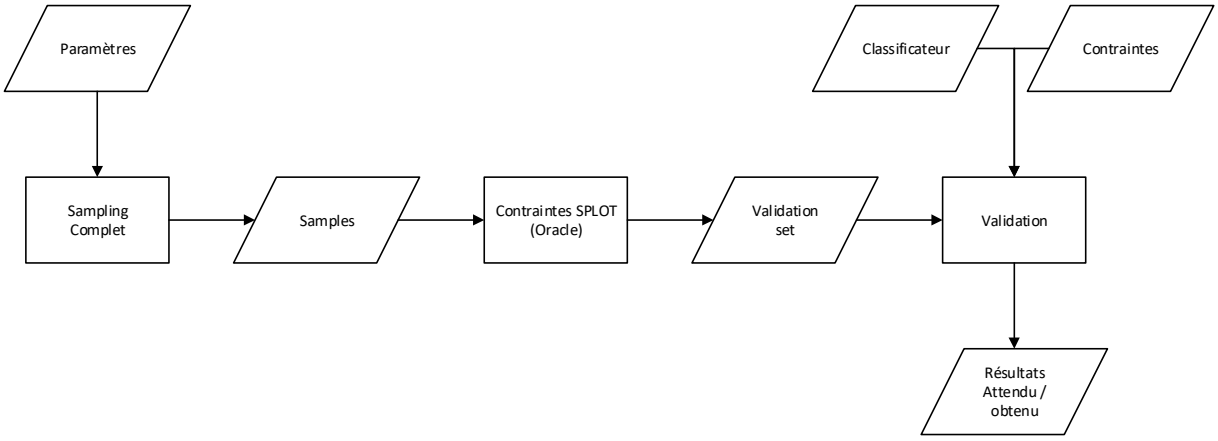


FIGURE 16:VALIDATION D'UN CLASSIFICATEUR ADJOINT DE CONTRAINTES

Nous validerons cette nouvelle fonction comme précédemment, avec un échantillonnage exhaustif puis partiel, en variant la quantité de contraintes extraites de chaque modèle, de 10% à 100% par tranche de 10%. Pour chaque évaluation, plusieurs tirages des contraintes extraites seront effectués.

3.4 Mise en place

Afin de réaliser les différentes évaluations, un programme a été mis en place permettant le chargement des modèles S.P.L.O.T. et l'exécution du protocole de test, en utilisant Weka pour effectuer la classification. La

librairie S.P.L.O.T. a été utilisée pour analyser les fichiers au format *SXFM* utilisé par ces modèles. Les contraintes des modèles ont été intégrées à un solveur afin de permettre leur usage comme oracle.

Ensuite, les modèles ont étés utilisés avec les différents générateurs d'échantillons et les différents classificateurs, avec notre évaluateur.

3.4.1 Limitations

Les outils utilisés possèdent certaines limitations. La première est le nombre de contraintes utilisables. En effet, il faut noter qu'un échantillonnage exhaustif d'un modèle à n features comprend 2^n configurations. Si dans l'absolu, la limite imposée par nos outils est de 30 features, car WEKA utilise des entiers signés de 32 bits16 pour indexer les ensembles d'entraînement, dans la pratique, un échantillonnage de cette taille utiliserait plusieurs gigaoctet mémoire vive. De plus, même en utilisant un échantillonnage partiel, il faut considérer les limites des différents algorithmes, en consommation mémoire comme en temps de calcul.

Lors de nos tests, aucun algorithme n'a pu dépasser 26 *features* sans rencontrer de plantage. La cause de ces plantages était chaque fois un manque de mémoire de travail (*out of memory*) ou une surcharge du ramasse miettes Java (*GC overhead limit exceeded*). Notons que la configuration pour l'exécution de nos tests accordait 4Go de mémoire RAM à la machine virtuelle Java (JVM).

Aussi, étant donné que le temps d'exécuter l'entraînement et validation d'un seul modèle de cette taille se compte en heures, le nombre de *features* a dû être encore limité. Afin de pouvoir effectuer les évaluations dans un temps raisonnable, nous nous sommes limités à 15 ou 18 *features*.

3.4.2 Statistiques des modèles

Avant de passer aux résultats des évaluations, il est intéressant de placer quelques grandeurs sur les modèles utilisés dans les tests. Parmi les 900 modèles S.P.L.O.T., nous nous limiterons aux modèles possédant un maximum de 20 *features*. La Figure 17reprend le nombre de modèles et la répartition du nombre de contraintes[17] en fonction du nombre de *feature*.

[16] Dont la valeur maximale est 2^{31}-1

[17] Dans le comptage des contraintes, nous considérons que l'ensemble des restrictions hiérarchiques appliquées à un nœud constitue une seule contrainte.

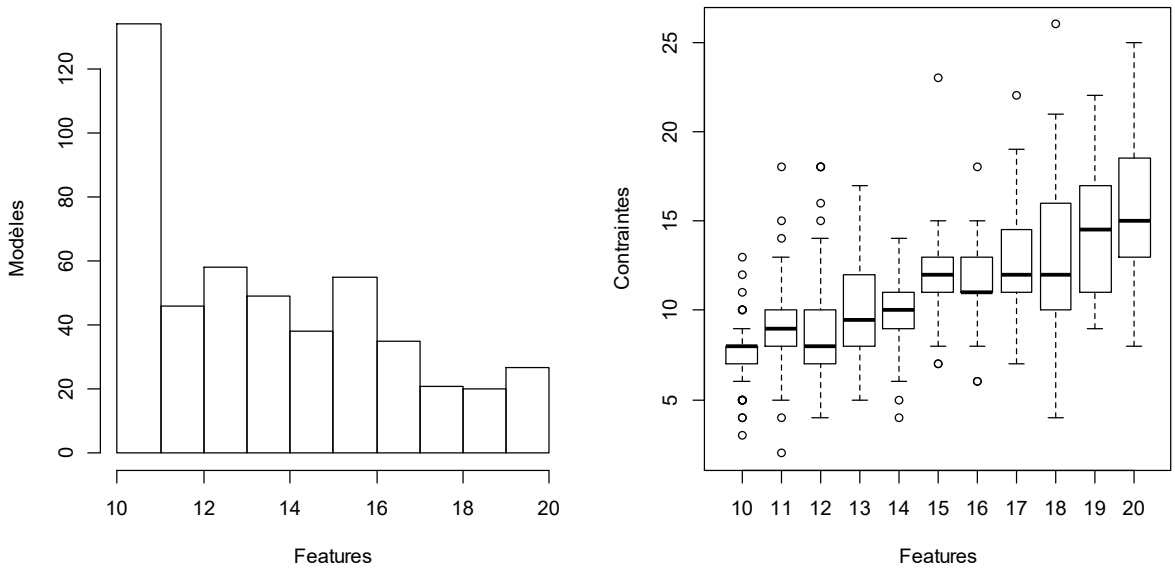


FIGURE 17: (A GAUCHE) NOMBRE DE MODELE PAR NOMBRE DE FEATURES ET (A DROITE) LA REPARTITION DU NOMBRE DE CONTRAINTES PAR NOMBRE DE FEATURES

3.5 Évaluations

3.5.1 Échantillonnage exhaustif

Nous avons entraîné et validé un classificateur avec un échantillonnage complet sur différents algorithmes. Les évaluations ont été effectuées sur les modèles S.P.L.O.T. ayant moins de 20 *features*[18]. Le tableau suivant reprend la moyenne sur tous les modèles sélectionnés, des critères d'évaluation pour chaque classificateur.

TABLEAU 4: RESULTATS DE L'EVALUATION DE DIFFERENTS CLASSIFICATEURS AVEC UN ECHANTILLONNAGE EXHAUSTIF

Classificateur	*Exactitude*	*Précision*	*Rappel*	*Spécificité*	*F_1-Score*
Hoeffding Tree	0.6905	0.1067	0.2410	0.6941	0.0743
Naive Bayes	0.9916	0.1665	0.0685	0.9998	0.0857
C4.5 (J48)	0.9956	0.3196	0.2973	0.9997	0.3017
SVM (Linear)	0.9893	0.3757	0.3502	0.9981	0.3592
Logistic Model Tree	0.9959	0.4258	0.3769	0.9995	0.3881
Logistic Regression	0.9919	0.4918	0.3894	0.9982	0.4130
*K**	0.9962	0.5600	0.4538	1.0000	0.4864
REP Tree	0.9968	0.6123	0.4991	0.9993	0.5276
PART Decision List	0.9953	0.6017	0.5310	0.9994	0.5496
RIPPER (JRip)	0.9965	0.6921	0.5883	0.9993	0.6064
Random Committee	0.9976	0.7371	0.7251	0.9988	0.7065
SVM (RBF γ=2)	0.9990	0.8600	0.7976	1.0000	0.8193
Multilayer Perceptron	0.9997	0.9656	0.9215	1.0000	0.9395
SVM (PUK ω=1, σ=0.1)	1.0000	0.9800	0.9800	1.0000	0.9800
Random Forest	1.0000	1.0000	1.0000	1.0000	1.0000

[18] Pour certains configurateurs, le nombre de *features* fut limité à 15, vu le temps nécessaire à l'entraînement d'un tel modèle (voir Comparaison des performances)

SVM (Poly. d=3,γ=2,c_0=0.5)	1.0000	1.0000	1.0000	1.0000	1.0000
kNN (IBk)	1.0000	1.0000	1.0000	1.0000	1.0000
SVM (RBF γ=5)	1.0000	1.0000	1.0000	1.0000	1.0000

Parmi les algorithmes, certains sont capables de modéliser correctement et entièrement tous les modèles. Ces algorithmes obtiennent une précision et un rappel de 100%. C'est le cas des *k Nearest Neighbours*, de *Random Forest* et de *SVM* pour certains *kernels* particuliers (en l'occurrence RBF avec γ=5 et polynomial de degré 3 avec γ=2 et c_0=0.5).

D'autres algorithmes se situent juste derrière : *Multilayer Perceptron* et *SVM* avec un *kernel PUK (ω=1 et σ=0.1)*. Ces algorithmes ont une précision et un rappel au-dessus de 95%, ce qui signifie que plus de 95% des configurations qu'ils classifient comme valide le sont réellement et qu'ils trouvent plus de 95% des configurations valides.

Les autres algorithmes obtiennent des résultats nettement plus bas. Par exemple PART et RIPPER, les deux algorithmes d'extraction de règles, on un F_1-Score avoisinant 0.55 et 0.6. Les arbres de décisions sont particulièrement mauvais dans cette évaluation : le meilleur est *REP Tree* avec un F_1-score de 0.52.

Les résultats obtenus par certains algorithmes en termes de précision et de rappel sont très bas, même si l'exactitude et la spécificité sont généralement très hautes (au-dessus de 90% à une exception près). Cela s'explique facilement par le fait que la plupart des configurations des modèles sont négatives. Pour les modèles utilisés lors de l'évaluation, le ratio de validité (défini comme la fraction de configurations valides sur le nombre total de configurations) est très faible, comme le montre le graphique suivant.

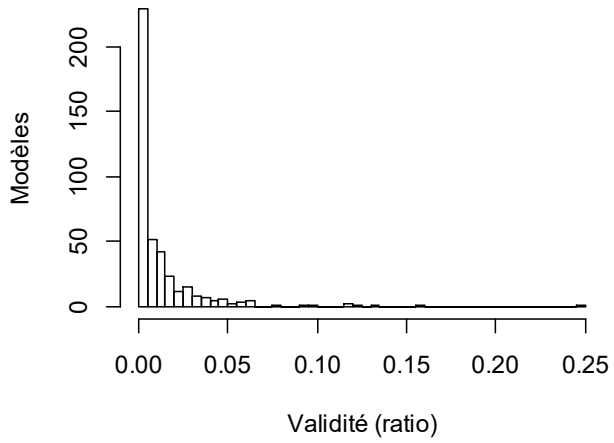


FIGURE 18: REPARTITION DU RATIO DE VALIDITE DES MODELES S.P.L.O.T. AYANT AU PLUS 18 *FEATURES*

Afin d'expliquer ces résultats, regardons plus en détails la répartition des résultats des algorithmes par modèle. Le graphique suivant montre la répartition du F_1-Score obtenu par modèle pour chaque algorithme.

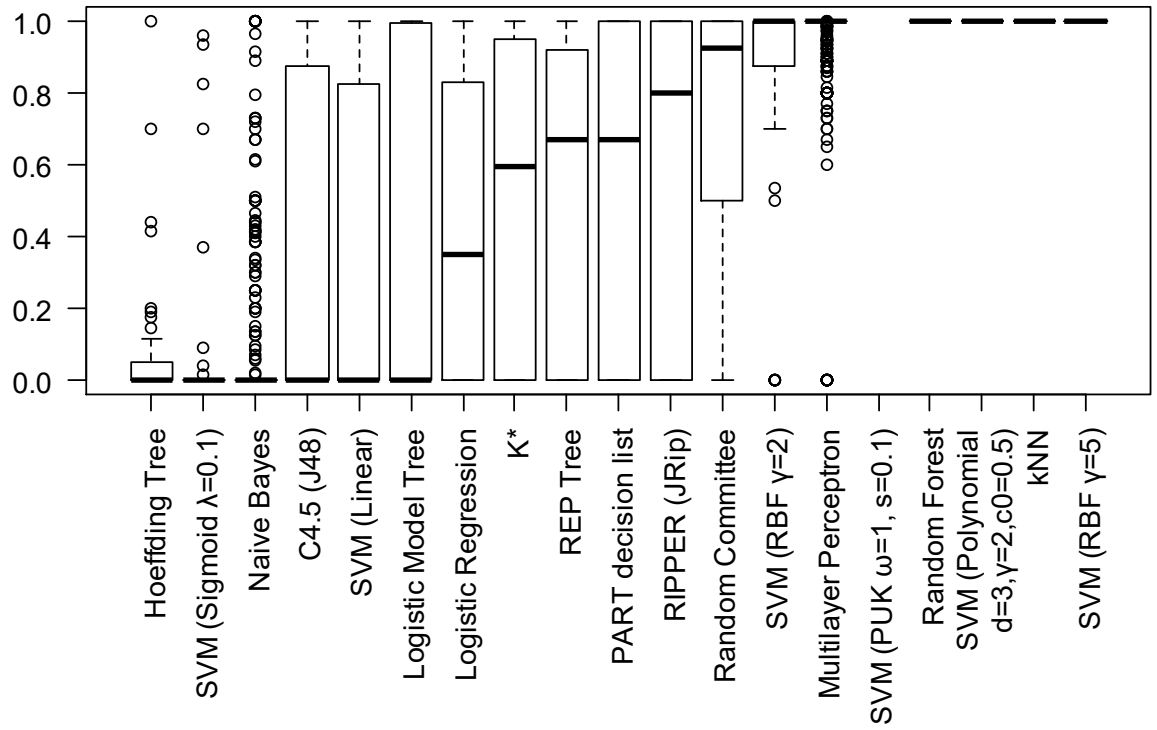


FIGURE 19: F_1-SCORE OBTENU A L'EVALUATION DE DIFFERENTS CLASSIFICATEURS AVEC UN ECHANTILLONNAGE EXHAUSTIF

Nous constatons de grandes disparités dans les F_1-Scores de plusieurs algorithmes. Certains, comme RIPPER ou PART, varient d'une extrême à l'autre en fonction du modèle : ils sont capable de modéliser entièrement certains, et ce trouvent complétement incapable d'en représenter d'autres.

Intéressons-nous au cas spécifique où l'algorithme ne classifie pas un seul résultat comme valide, alors que le modèle S.P.L.O.T. possède bel et bien des configurations valides (auquel cas la précision et le F_1-Score sont nuls). Ce cas est responsable en grande partie des mauvais résultats des classificateurs. Si on le met de côté, nous constatons que le F_1-Score de certains algorithmes augmente fortement. C'est par exemple le cas de *K*, Random Committee* et *C4.5*. Le Tableau 5 montre la différence entre les deux.

TABLEAU 5: COMPARAISON DES F_1-SCORE EN RETIRANT LES MODELES DONT LE CLASSIFICATEUR OBTIENT TOUJOURS "INVALIDE" ; LES COLONNES REPRESENTENT : LA MOYENNE DES F_1-SCORE POUR TOUS LES MODELE, LA MOYENNE DU F_1-SCORE DES MODELES RESTANT ET LA PROPORTION DE MODELES RETIRE

Classificateur	*F_1-Score*	*F_1-Score'*	*Retirés (taux)*
Hoeffding Tree	0.0743	0.2184	0.6600
Naive Bayes	0.0857	0.4854	0.8234
Logistic Regression	0.4130	0.6422	0.3569
REP Tree	0.5276	0.7878	0.3303
RIPPER (JRip)	0.6064	0.8178	0.2585
Logistic Model Tree	0.3881	0.8254	0.5298
PART decision list	0.5496	0.8426	0.3477
SVM (Linear)	0.3592	0.8551	0.5800
Random Committee	0.7065	0.8629	0.1812

*K**	0.4864	0.8686	0.4400
C4.5 (J48)	0.3017	0.9009	0.6651
SVM (RBF γ=2)	0.8193	0.9527	0.1400
Multilayer Perceptron	0.9395	0.9729	0.0344
SVM (PUK ω=1, σ=0.1)	0.9800	1.0000	0.0200

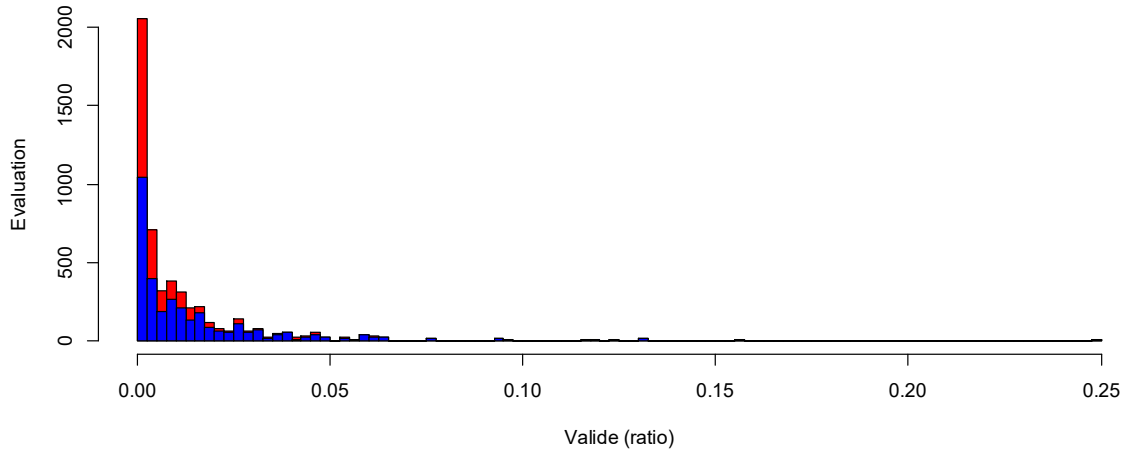


FIGURE 20: HISTOGRAMME DU TAUX DE VALIDITE D'UN MODELE, DISTINGUANT LES MODELES RETIRES (ROUGE) ET RESTANTS (BLEU)

Comme on le voit dans l'histogramme Figure 20, les modèles pour lesquels les classificateurs sont complétement invalides sont des modèles où la proportion de résultats valide est la plus faible.

Il y a deux explications principales au fait que les algorithmes ne produisent pas des classificateurs parfait, même en présence d'un échantillonnage exhaustif. De manière générale, un algorithme de classification utilise un modèle sous-jacent et effectue son apprentissage en ajustant ce modèle afin qu'il corresponde au mieux aux données d'entraînement. Les deux problèmes qui peuvent survenir sont que d'une part, le modèle sélectionné ne peut pas s'ajuster aux données d'entrainement, et que d'autre part, l'algorithme n'ajuste pas de manière optimale les données.

Le premier cas est flagrant, par exemple, avec naive Bayes. Cet algorithme se base sur l'hypothèse que les *features* sont toutes indépendantes. Dans les modèles S.P.L.O.T., toute *feature*, sauf la racine, a au moins une dépendance avec son parent.

Le deuxième cas est présent par exemple avec les arbres de décision. Intuitivement, n'importe quel modèle S.P.L.O.T. pourrait être représenté par un arbre de décision : on peut construire un tel arbre en construisant un arbre de *n* étages (avec *n* le nombre de *features*) et qui associe à tous les nœuds de l'étage *j* à la présence de la $j^{ème}$ *feature*. Cela étant, un tel arbre occupe un large espace mémoire (complexité exponentielle) et ne pourrait pas être construit à partir d'un échantillonnage incomplet. Ce faisant, il perdrait le principal objectif des algorithmes de classification : leur capacité de généraliser un modèle pour prédire de nouveaux résultats.

Leur objectif étant la généralisation, les algorithmes ne tentent pas nécessairement d'obtenir un classificateur ayant une *erreur d'entraînement* nulle (il s'agit de l'erreur obtenue en classifiant l'ensemble

d'entraînement). Par exemple, des algorithmes utilisant des arbres de décision comme *C4.5* ou *REP Tree* se basent sur la mesure du gain d'information pour décider d'ajouter ou de retirer une décision sur une *feature* particulière. Si aucune *feature* n'offre un gain d'information suffisant, l'algorithme conclut qu'il n'est pas possible de généraliser.

3.5.2 Échantillonnage partiel

Les évaluations ont été reproduites avec un échantillonnage partiel. Les évaluations ont été effectuées sur les modèles S.P.L.O.T. ayant moins de 15 *features*. Le graphique suivant reprend la moyenne du F_1-Score par classificateur sur chacun des modèles sélectionnés avec les différents échantillonnages.

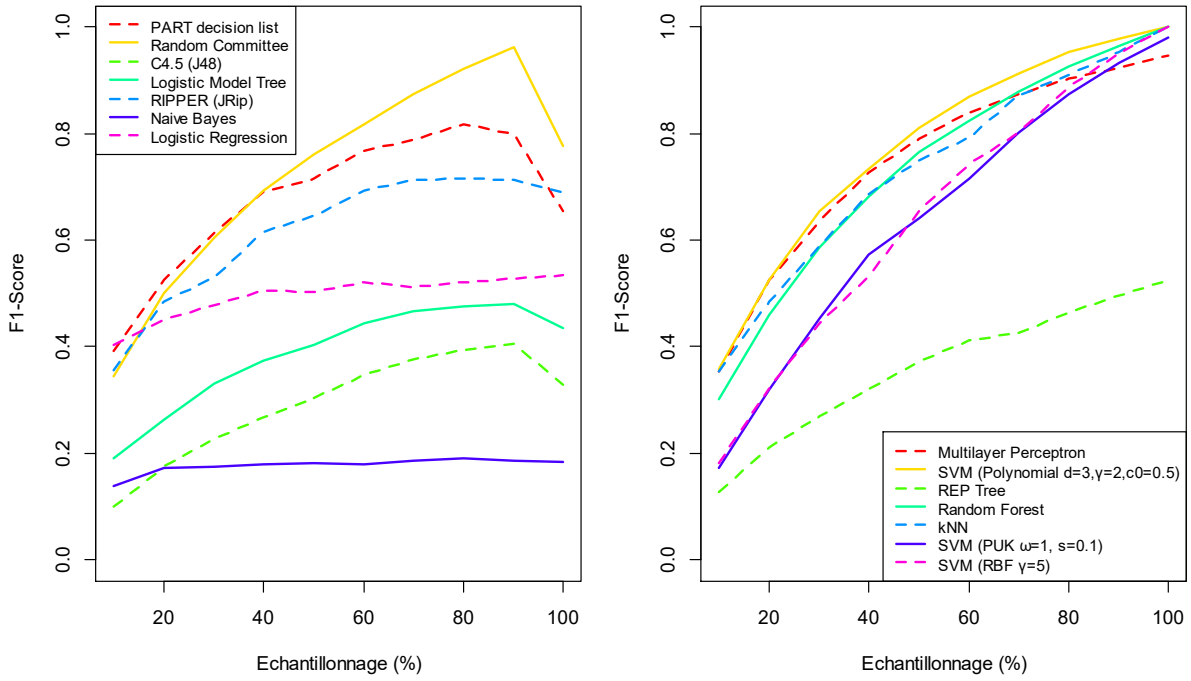


FIGURE 21: EVOLUTION DU F_1-SCORE AVEC UN ECHANTILLONNAGE PARTIEL

Nous constatons que, globalement, le F_1-Score croit avec la complétude de l'échantillonnage, à deux exceptions près : *naive Bayes* et *Logistic Regression,* qui ne varient que très peu. Pour les autres algorithmes, deux types de comportements de remarquent : certains montrent une croissance monotone sur l'ensemble des échantillonnages (ceux situés dans le graphique de droite), et d'autres (à gauche) ont leur maximum aux alentours de 90%, puis décroissent avec un échantillonnage complet.

Il s'agit d'un cas particulier de sur-apprentissage : l'algorithme d'apprentissage, en générant certaines valeurs particulières, produit un classificateur moins fidèle au modèle original.

Les algorithmes offrant les meilleurs résultats sont les *SVM*, *Multilayer Perceptron*, *kNN* et les algorithmes ensemblistes (*Random Forest* et *Random Committee*). Ensuite viennent *PART* et *RIPPER* : les algorithmes d'extraction de règles. Les autres algorithmes, *naive Bayes*, L*ogistic Regression* et les arbres de décision (*C4.5* et *REP Tree*) ne dépassent pas un F_1-Score de 0,5.

Les graphiques suivants montrent l'évolution de la précision et du rappel dans les mêmes conditions. Nous constatons des comportements.

FIGURE 22: EVOLUTION DE LA PRECISION AVEC UN ECHANTILLONNAGE PARTIEL

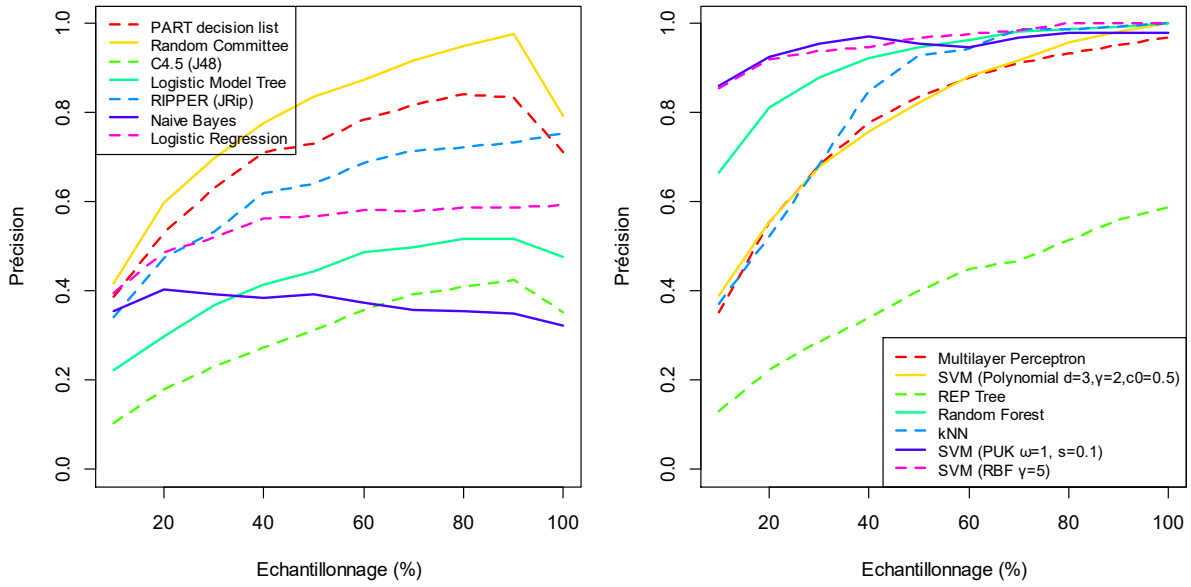
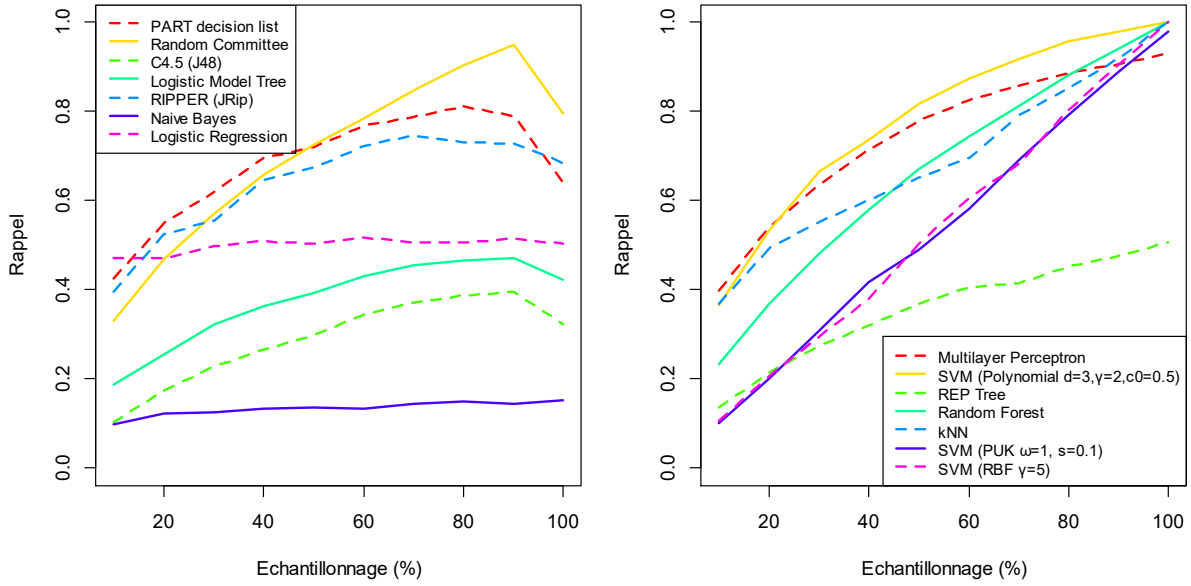


FIGURE 23: EVOLUTION DU RAPPEL AVEC UN ECHANTILLONNAGE PARTIEL



Nous noterons que la précision est généralement plus haute que le rappel, particulièrement pour les algorithmes obtenant de bons résultats (*SVM, Random Forest* et *kNN*). Ces algorithmes sont capables d'obtenir un classificateur précis (même si peu sélectif) même avec un échantillonnage réduit.

Notons aussi qu'un résultat de 0,5 en précision signifie que le classificateur se trompe pour la moitié des modèles qu'il classifie comme valides (faux positifs), tandis qu'un rappel de 0,5 signifie qu'il ne classifie correctement que la moitié des modèles réellement valides (faux négatifs). Ainsi, les résultats dans la zone en dessous de 0,5 sont ceux des classificateurs étant plus souvent incorrects que corrects.

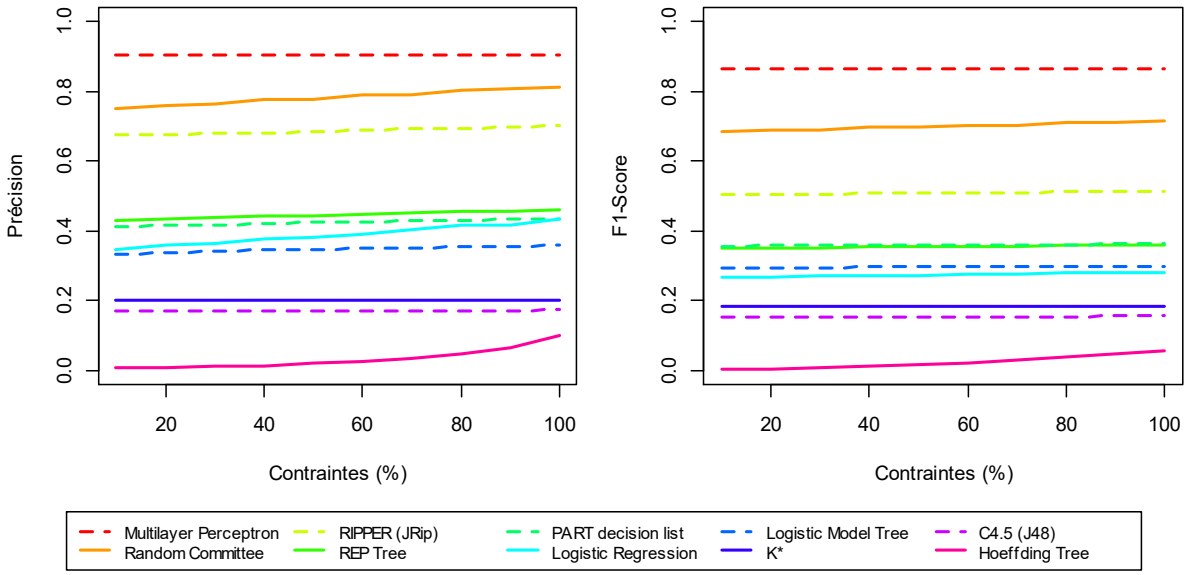
3.5.3 Ajout de contraintes manuelles

3.5.3.1 Échantillonnage exhaustif

Dans un premier temps, nous avons évalué les résultats des différents algorithmes avec un échantillonnage complet. Notons d'abord que le rappel précédemment mesuré ne sera pas affecté par l'ajout de contraintes : étant donné que les contraintes ne font qu'éliminer des configurations, les seules erreurs de classification qu'elles élimineront sont les faux négatifs. Les algorithmes ayant précédemment obtenu une précision de 1 ont donc étés omis lors de ce test.

Nous avons évalué les modèles ayant moins de 18 *features*, et avons effectué dix tirages pour les contraintes à extraire sur chaque modèle. Le graphique suivant reprend la moyenne de la précision et du F_1-Score obtenu par algorithme, pour les différentes proportions de contraintes extraites.

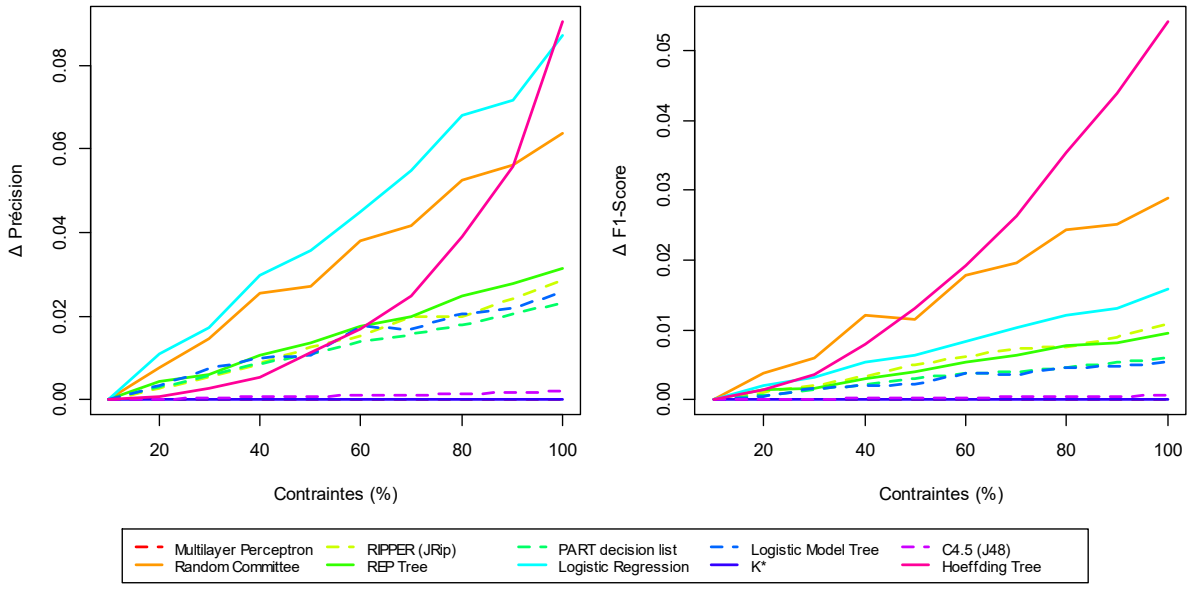
FIGURE 24: EVOLUTION DE LA PRECISION ET DU F1-SCORE AVEC L'AJOUT DE CONTRAINTES



Nous constatons premièrement que certains classificateurs offrent des résultats similaires sur l'ensemble des tests (*Multilayer Perception*, *K** et *C4.5*). Ceci peut s'expliquer par le fait que les classificateurs obtenus sont soit complétement invalides (cas discuté pour l'échantillonnage exhaustif), ou ont une précision de 1. Dans les deux cas, le nombre de faux négatifs est 0 et donc la valeur n'est pas affectée.

Pour les autres algorithmes, il y a une augmentation de la précision et du F_1-Score. Cette augmentation n'est toutefois pas significative : le gain maximum en précision est de 9% pour les *Hoeffing Tree* et la *Logistic Regression*, mais dans l'absolue, leurs F_1-Scores restent très faibles. L'autre résultat notable est celui du *Random Committee*, dont la précision passe de 74% à 81%. Le graphique suivant montre la variation de la précision et du F_1-Score dans les résultats.

FIGURE 25: VARIATION DE LA PRECISION ET DU F_1-SCORE AVEC L'AJOUT DE CONTRAINTES



3.5.3.2 Échantillonnage partiel

Le protocole précédent a ensuite été repris, en utilisant cette fois un échantillonnage partiel. Les évaluations ont été effectuées sur 40 modèles S.P.L.O.T. ayant moins de 15 *features*, avec un échantillonnage aléatoire variant de 10% à 90% des configurations, par tranche de 10%.

Les Figure 38 et Figure 39 (voir Annexe A : Résultats S.P.L.O.T. détaillés) à représentent la précision moyenne obtenue en fonction du pourcentage de contraintes ajoutées, par échantillonnage, et les Figure 40 et Figure 41, le F_1-Score. Enfin, les Figure 42 et Figure 43 montrent la variation du F_1-Score.

Comme précédemment, l'effet de l'ajout de contraintes est en général d'augmenter la précision. Les cas qui font exception (*Logistic Model Tree*, *K** et *SVM* avec de haut taux d'échantillonnage) sont similaires à ceux déjà discutés.

On constate que l'effet de l'ajout de contraintes sur la précision et le F_1-Score est plus important pour de plus faibles taux d'échantillonnage. À 10%, certains algorithmes (*SVM*, *kNN* et *PART*) voient leur F_1-Score augmenter de 0,15 avec l'ajout des contraintes. À 50%, la variation maximale du F_1-Score est d'environ 0,05. En fait, plus on augmente l'échantillonnage, plus le F_1-Score de base (sans contraintes) augmente, et moins l'ajout de contraintes impacte le résultat global.

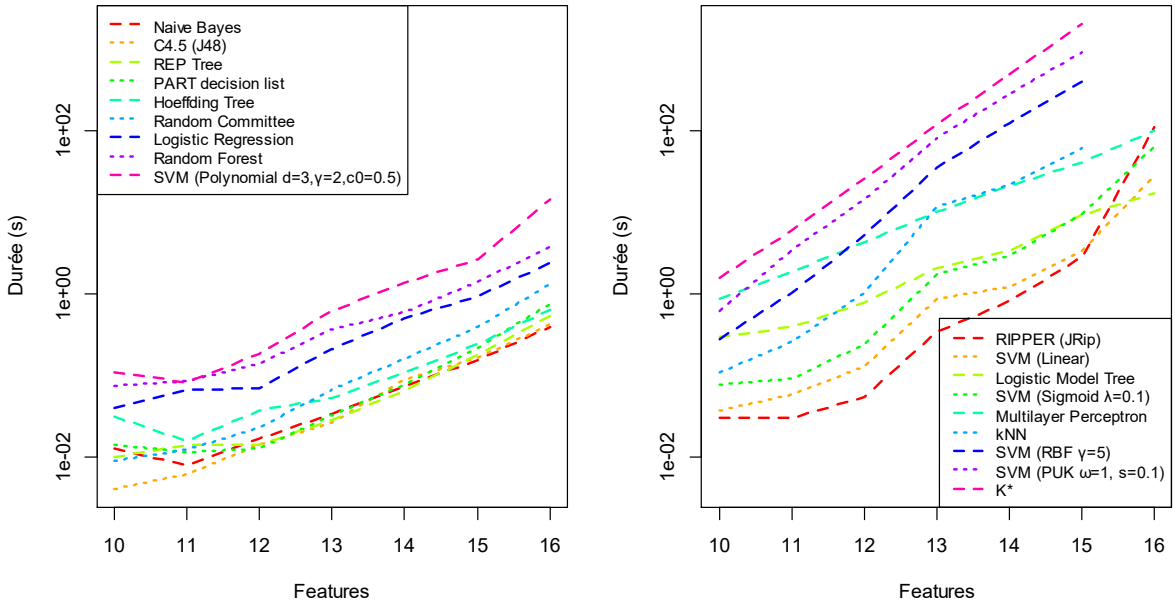
Comparaison des performances

Afin de compléter nos évaluations, il est intéressant de comparer les performances respectives des différents modèles et leurs variations en fonction du nombre de *features*. En effet, le temps de l'entraînement des modèles n'est pas nécessairement négligeable, surtout quand le nombre de *features* augmente. Certains algorithmes ne seront utilisables qu'avec un nombre limité de *features*, vu la croissance exponentielle du nombre de modèles.

Le graphique suivant reprend le temps d'entraînement et de validation moyen sur 10 modèles, en fonction du nombre de *features*. La validation du modèle est incluse dans la mesure, afin de prendre en compte les

étapes d'optimisations effectuées par certains algorithmes après la fin de l'entraînement et pendant les premières classifications.

FIGURE 26: DUREE D'ENTRAINEMENT ET D'EVALUATION EN FONCTION DU NOMBRE DE *FEATURES* PAR CLASSIFICATEUR



Ce graphique, dont l'échelle de temps est logarithmique, montre une nette séparation entre trois groupes d'algorithmes. D'une part, les arbres de décision et les algorithmes ensemblistes, ainsi que *Logistic Regression*, *naive Bayes et PART*, qui ont une progression plus faible. Ce sont les algorithmes les plus rapides, et en ce sens les plus adaptés aux modèles plus grands. Ensuite viennent le *SVM polynomial* et *linéaire*, *RIPPER*, *Logistic Model Tree* et *Multilayer Perceptron*. En dernier viennent *kNN, K** et les *SVM RBF* et *PUK*.

3.6 Conclusion

Dans ce chapitre, nous avons utilisé les modèles S.P.L.O.T. pour évaluer l'usage de l'apprentissage automatique d'un classificateur afin de modéliser la validité de modèles de variabilité, en utilisant comme oracle un solveur de contraintes alimenté par les informations des modèles.

Des classificateurs ont étés entraînés avec un échantillonnage exhaustif, puis partiel. Les tests ont révélé que les *SVM*, *kNN*, *Random Forest* et *Multilayer Perceptron* obtiennent un F_1-Score satisfaisant pour des échantillonnages suffisamment importants. *PART,* et dans une moindre mesure *Random Committee* et *RIPPER*, fournissent aussi de bons résultats, mais sont sensible au surentraînement.

D'autres algorithmes, comme *naive Bayes*, *Logistic Regression*, ainsi que les quatre algorithmes de type « arbre de décision » (*Hoeffding Tree*, *C4.5*, *Logistic Model Tree* et *REP Tree*) semblent inadaptés à notre usage. Il est à noter que le principal défaut de ces derniers est qu'ils classifient certains modèles comme « totalement invalides ».

Ensuite fut simulé l'effet de l'ajout de contraintes écrites par un expert, en utilisant les contraintes du modèle, ou une partie de celles-ci. L'évaluation comprenait différents échantillonnages avec ajouts plus ou moins conséquents. Nous avons déterminé que l'ajout de contraintes influençait positivement la précision

et le F_1-Score, en corrigeant les faux-positifs. Cela étant, les faux-négatifs ne sont pas affectés et donc le score de rappel ne change pas.

De plus, comme l'ajout de 100% des contraintes signifie qu'un expert à modélisé l'entièreté des contraintes du modèle, ce cas rend le classificateur effectivement inutile. Pour que la technique soit intéressante, il faudrait pouvoir obtenir une augmentation substantielle du F_1-Score avec peu de contraintes ajoutées, ce qui ne fut pas le cas lors de nos évaluations.

4 Thingiverse et OpenSCAD

Dans ce chapitre, nous allons évaluer les performances de notre technique pour un cas pratique : la configuration de modèles 3D issus de Thingiverse. Dans un premier temps, nous présenterons celui-ci, ainsi que le format des fichiers d'objets 3D paramétriques : OpenSCAD. Ensuite nous discuterons de la récupération des informations sur les fichiers et de la conception de l'oracle.

Nous présenterons ensuite les résultats de nos évaluations et analyserons ces informations. Nous fournirons des exemples d'objets et de configurateurs obtenus et discuterons des limitations de la méthode et de l'oracle.

4.1 Thingiverse

Thingiverse[19] est un site communautaire de partage de modèles 3D imprimables. Il propose de télécharger des modèles publiés par les utilisateurs. Parmi ces modèles, certains sont paramétriques. Le site propose un configurateur pour générer les configurations de ces modèles, mais celui-ci ne comporte aucune forme de validation.

Même si le configurateur proposait une forme de validation, la communauté du site, principalement composée de hobbyistes, n'a pas nécessairement le temps ou les connaissances requises pour écrire des contraintes sur la validité des modèles 3D. De plus, une grande quantité de modèles existe déjà sur ce site. Notre méthode se veut permettre d'offrir une validation interactive lors de la configuration des modèles, sans nécessiter un travail manuel spécifique à chaque modèle.

Dans ce chapitre, les modèles configurables de ce site seront utilisés pour mettre en œuvre notre méthode. Un oracle sera construit pour valider des configurations, et il sera utilisé pour entraîner une fonction de validation des modèles 3D.

4.1.1 Les objets 3D sur Thingiverse

Sur ce site, un modèle 3D (appelé *Thing*) peut être publié dans plusieurs formats. Parmi ceux-ci, les fichiers *.stl* et *.scad* sont majoritairement utilisés, et les plus supportés. Chaque *thing* présente une description, une liste de fichiers et une page de commentaire. D'autres informations sont disponibles, comme la License du modèle ou le nombre de téléchargements. Il est aussi possible aux utilisateurs ayant imprimé le modèle de publier des photos, ou d'ajouter le modèle à des collections.

Le format *.stl* (STereo Lithography) est un format largement répandu dans l'impression 3D et supporté par de nombreux logiciels de CAD utilisés pour la modélisation. Ces fichiers ne sont pas paramétrables. Un fichier *.stl* décrit un ensemble de solides, chacun constitué d'un ensemble de polygones, généralement des triangles. Ce format est considéré comme un standard « de facto », reconnu par tous les logiciels d'impression 3D et utilisé comme format d'échange. Le format *.scad* est spécifique au logiciel *OpenSCAD*. Contrairement aux précédents, ces fichiers sont paramétrables.

Sur *Thingiverse*, les *things* possèdent un certain nombre de tags. Un modèle .scad taggué du mot-clé « customisable » est considéré comme paramétrique. Thingiverse permet d'ouvrir ces modèles dans une application de configuration (le *customizer*).

[19] https://www.thingiverse.com

Ce configurateur prend la forme d'un formulaire, permettant de remplir les valeurs des paramètres du modèle. Une prévisualisation de l'objet configuré est affichée et mise à jour à chaque modification des paramètres. Une fois la configuration effectuée, l'utilisateur télécharge un fichier *.stl* généré à partir du fichier *.scad*, avec les paramètres entrés.

4.2 OpenSCAD

Le format *.scad* est spécifique au logiciel *OpenSCAD*. Les fichiers *.scad* sont des fichiers texte, décris dans un langage spécifique. Le format n'est reconnu que par ce programme, mais celui-ci permet la génération d'un *.stl,* pour des valeurs de paramètres données.

Le langage *OpenSCAD* est un langage descriptif permettant de construire un objet 3D. Les objets SCAD sont construits par programmation et non par modélisation graphique, comme c'est généralement le cas des logiciels de modélisation.

Le langage lui-même n'offre pas de formalisme pour déclarer des paramètres. Cela étant, le logiciel permet de remplacer les valeurs des constantes déclarées dans un modèle lors de la génération d'un *.stl*. En interprétant les constantes déclarées dans un fichier *.scad* comme paramètres, un fichier *.scad* est paramétrique. Le configurateur de *Thingiverse* se base sur les constantes et les commentaires qui les accompagnent pour extraire les paramètres du modèle.

Le langage Open*SCAD* est un langage de programmation complexe, et les constantes peuvent prendre avec différents types de données. Les types de base sont *number* (nombre à virgule flottante) et *string* ; il existe des types de données plus complexes, comme les tableaux, les images ou les maillages, mais ils ne sont généralement pas utilisés comme paramètres. Aussi, cette étude se cantonnera aux types de données de base.

4.3 Mise en place

Avant de pouvoir entraîner un classificateur, trois étapes préalables sont nécessaires : l'extraction des paramètres d'un fichier OpenSCAD et de leurs domaines, et la création d'un oracle. Il aura aussi fallu récupérer un grand nombre de fichiers depuis *Thingiverse*. Pour ce faire, un *scraping* du site a été mis en œuvre afin de récupérer l'ensemble des fichiers OpenSCAD configurables et disponibles en ligne.

4.3.1 Extraction de paramètres des fichiers SCAD

La première étape de l'extraction est de *parser* un fichier SCAD. Le *parsing* d'un langage de programmation consiste à la décomposition du texte source en une représentation structurée, exploitable par un programme. Ce processus se base sur un ensemble de règles définissant la syntaxe du langage : sa grammaire formelle.

4.3.1.1 Parsing des fichiers OpenSCAD

Le *parsing* comporte généralement deux étapes : l'analyse lexicale et l'analyse syntaxique. La première transforme le texte source en une séquence de termes du langage, ou lexèmes ; la seconde construit un arbre des lexèmes du langage (aussi appelé *concrete syntax tree*). Cet arbre peut ensuite être parcouru pour générer un arbre syntaxique (*abstract syntax tree* – AST).

Il existe de nombreux outils permettant de générer un analyseur lexical et syntaxique. Il s'agit généralement de décrire les lexèmes et les règles grammaticales dans un langage particulier, et programmer une série d'opérations afin d'associer à chacun des informations. Durant l'analyse lexicale, on peut ainsi récupérer

les valeurs des différents lexèmes. Durant l'analyse syntaxique, les valeurs associées aux lexèmes seront utilisées pour construire l'arbre syntaxique.

Dans notre cas, l'analyse des fichiers OpenSCAD a été développée en Java. Nous avons choisis ANTLR pour développer le parseur. ANTLR4 permet la génération d'un analyseur lexical et syntaxique de type « ALL(*) » (Adaptative LL). Il s'agit d'une évolution des parseurs traditionnels « LL(k) ».

Le parseur généré est un « *tree-walker* » : chaque règle de la grammaire du langage est associée à une séquence d'instructions, et l'analyseur effectue un parcours de l'arbre syntaxique concret, en appelant les instructions correspondant à la règle générant chaque nœud. La description des lexèmes et des règles se font dans un langage dédié. Celui-ci se base sur la forme de Backus-Naur étendue (EBNF), augmenté de blocs de code Java.

4.3.1.2 Grammaire formelle OpenSCAD

OpenSCAD ne possède de de spécification formelle ; nous devrons donc déduire la grammaire avec les informations disponibles. Étant un projet communautaire actif, le langage évolue de manière incrémentale et continue. De plus, la documentation ne suit pas toujours les dernières fonctionnalités. Pour construire la grammaire, nous disposons comme principales sources :

- du code source du programme OpenSCAD, qui est l'implémentation de référence du langage ;
- de la documentation sur le wiki du site OpenSCAD ;
- en dernier recours, du fonctionnement observé du programme étant donné un code source spécifique.

Le code source d'OpenSCAD est écris en C++ et comprend un analyseur écrit à l'aide de Flex et Bison. Flex est un outil permettant de générer un analyseur lexical, tandis que Bison génère un analyseur syntaxique. Ces deux outils possèdent leur propre langage dédié, aussi dérivé d'EBNF, que nous utiliserons comme base pour décrire la grammaire avec ANTLR.

4.3.1.3 Récupération des commentaires dans OpenSCAD

Les commentaires dans les fichiers sources sont généralement inutiles au programme interprétant le fichier. Généralement, l'analyseur lexical ne génère pas de lexème pour un commentaire, et la grammaire du langage ne les considère pas. Dans notre cas, les commentaires peuvent s'avérer des sources d'information pertinente, car les utilisateurs y ajoutent des informations concernant les paramètres.

Intégrer les commentaires dans l'analyse syntaxique est problématique, car un commentaire peut apparaître à n'importe quelle position dans une séquence de lexèmes. Il faudrait donc créer une grammaire ajoutant des commentaires potentiels entre chaque pair de lexèmes, ainsi que les intégrer à n'importe quel endroit de la structure de l'arbre syntaxique.

Pour simplifier, nous avons opéré à la récupération des commentaires de manière séparée. Lorsqu'il rencontre un commentaire, notre analyseur lexical ne générera aucun lexème, mais ajoutera le commentaire dans une table séparée.

Un autre problème est l'association entre les commentaires et les instructions qu'ils décrivent. Les commentaires peuvent se trouver avant ou après une instruction, sur la même ligne ou à plusieurs lignes d'écart. Contrairement aux lexèmes de la grammaire, le positionnement « visuel » créé par les espacements et les passages à la ligne sont importants pour l'association des commentaires.

Il est donc nécessaire d'associer une information de position (i.e. la ligne et la colonne) aux commentaires, ainsi qu'aux nœuds de l'arbre syntaxique.

4.3.1.4 Extraction des paramètres

Une fois l'arbre syntaxique construit, nous pouvons extraire les paramètres et leurs associer les commentaires. Dans notre arbre, les paramètres sont des nœuds, situés directement à la racine, qui sont de type « définition de variable », et dont la valeur affectée est une valeur littérale (i.e. une valeur constante non calculée).

Après avoir listé tous les paramètres, nous parcourons la liste les commentaires. Pour chaque commentaire, nous récupérons les nœuds précédant et suivant le commentaire à la racine de l'arbre syntaxique.

Suivant la syntaxe *Thingiverse (*voir Annexe C : format pour le configurateur Thingiverse), nous associons les commentaires du code aux paramètres, de la manière suivante :

- un commentaire sur ligne précédent un paramètre devient sa description ;
- un commentaire en fin de la ligne du paramètre représente les valeurs possibles paramètre.

Les commentaires de section sont aussi reconnus et les paramètres appartenant à la section « hidden » sont éliminés.

4.3.1.5 Domaine des paramètres

Contrairement au cas d'étude précédent, où tous les paramètres étaient booléens, les paramètres peuvent ici prendre plusieurs types. Nous pouvons ramener les différents types à deux cas : les domaines nominaux et numériques.

Les domaines nominaux sont des énumérations d'un ensemble fini de valeurs, toutes connues, tandis que les domaines numériques sont des sous-ensembles des réels, possédant potentiellement des valeurs minimales et maximales. Les paramètres dont les domaines ne sont pas représentables par ces deux catégories (comme par exemple les paramètres textuels libres) ne seront pas étudiés ici.

Si une description des valeurs possibles est présente pour un paramètre, elle est utilisée pour déduire le domaine. Dans le cas contraire, la valeur par défaut sera utilisée pour voir si le domaine est numérique ou non.

4.3.2 Construction d'un oracle

Valider un modèle 3D est un problème complexe. Il existe un grand nombre de raisons qui peuvent faire qu'un modèle donné peut être considéré invalide. En fait, la notion de validité pour un modèle 3D manque de rigueur pour les raisons exposées aux paragraphes suivants.

4.3.2.1 Validité d'un modèle 3D

Un modèle peut très bien être imprimé avec succès par une personne sur une imprimante A, alors que l'impression échoue systématiquement pour une autre personne avec l'imprimante B. Un validateur de modèle 3D parfait dépendrait donc d'autres paramètres que ceux de génération du modèles 3D (par exemple l'imprimante utilisée). Cela étant, il devrait être possible de valider si un modèle 3D est « imprimable », auquel cas son impression devrait se dérouler correctement dans des conditions habituelles d'impression.

Outre les problèmes d'une imprimante particulière, il existe différentes causes qui rendent un modèle 3D non imprimable :

- le modèle n'est pas une surface fermée orientable - son intérieur n'est pas clairement défini[20] ;
- l'épaisseur du modèle est trop faible à certains endroits ;
- le modèle n'est pas connexe ;
- etc.

4.3.2.2 Implémentation de l'oracle

Construire un tel outil de validation est un problème complexe, que nous n'aborderons pas ici. Nous utiliserons des outils existants pour effectuer la validation. Nous procéderons en deux étapes :

Tout d'abord, le logiciel OpenSCAD sera utilisé pour générer un modèle *.stl* avec les paramètres correspondants. La réussite ou l'échec de la génération servira de première validation. Ensuite, nous utiliserons le logiciel *slic3r* pour valider le modèle.

Scil3r est un logiciel libre de découpe de modèles 3D pour l'impression. Il transforme des modèles 3D (dont le format *.stl*) en une série d'instructions de bas niveau pour une imprimante 3D. Ce logiciel possède aussi un mode diagnostique, qui analyse un objet 3D. Nous utiliserons ce mode pour récupérer des informations sur le modèle précédemment généré. Slic3r retourne les informations suivantes :

- le volume ;
- le nombre d'éléments ;
- les dimensions ;
- si des réparations sont nécessaires.

Les modèles dont le volume ou les dimensions sont inférieurs à un seuil, ou nécessitant des réparations, seront considérés comme invalides.

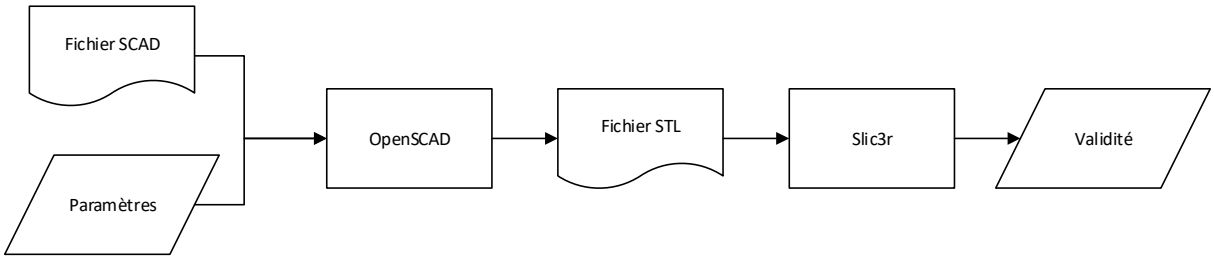


FIGURE 27: PROTOCOLE DE VALIDATION D'UNE CONFIGURATION D'UN FICHIER OPENSCAD

4.3.3 Échantillonnage

Contrairement au cas des modèles S.P.L.O.T., où le nombre de configurations possibles est fini, les paramètres des modèles OpenSCAD peuvent avoir un domaine infini. Pour de tels modèles, un échantillonnage exhaustif est évidemment impossible. Il existe de nombreuses méthodes pour générer un échantillon sur des valeurs numériques, de la génération complétement aléatoire à la génération homogène.

[20] Spécifiquement, un objet 3D imprimable doit être une variété de l'espace (*2-manifold*).

Nous avons choisi cette seconde approche : pour chaque paramètre de domaine numérique, nous générons n valeurs équidistantes entre la valeur minimale et maximale de l'intervalle :

$$\left\{V_{min} + \frac{x}{n-1}(V_{max} - V_{min}) \middle| x \in \{0, \dots, n-1\}\right\}$$

Pour générer un échantillon de configuration avec cette méthode, nous prenons l'ensemble des valeurs des paramètres nominaux, et générons l'ensemble ci-dessus pour les valeurs numériques. L'échantillon est obtenu en prenant toutes les combinaisons possibles de ces valeurs.

Afin d'obtenir un ensemble d'entraînement et un ensemble de validation différents (pour les paramètres numériques), mais de taille similaire, nous prendrons les échantillons générés avec n et $n+1$ valeurs.

4.3.4 Limitations

Les performances de l'oracle se sont avérées extrêmement variées selon les modèles. Pour certains, valider une seule configuration peut prendre plusieurs heures, tandis que pour d'autres, le processus prend moins d'une seconde. Ce temps est principalement dû à la complexité de génération de certains modèles .scad en .stl.

Disposant d'un grand nombre de modèles, mais d'aucun moyen préalable d'estimer la complexité d'un modèle, nous avons lancé la génération d'un ensemble d'entraînement et d'un ensemble de validation sur l'entièreté des modèles dont la taille estimée de l'ensemble d'entraînement était inférieure au milliard de configurations, en limitant à une heure le temps alloué à chaque modèle.

Seuls les modèles pour lesquels la génération fut complétée dans les temps seront considérés dans l'évaluation. Nous avons généré un ensemble d'entraînement avec $n = 20$ et d'un ensemble de validation avec $n = 21$. Sur 5057 modèles, 1580 correspondaient au critère de taille. Parmi ceux-ci, 201 modèles ont terminé la génération dans les temps.

Parmi les échantillons obtenus, 35 modèles n'ont généré que des configurations invalides (généralement à cause d'une erreur de syntaxe) et 123 uniquement des configurations valides (ces modèles simples ne produisant aucun cas dégradé). Il reste donc 33 modèles présentant des configurations valides et invalides.

4.3.5 Statistiques des modèles

La Figure 28 montre la répartition du nombre de paramètres dans les modèles. Ceux-ci ont entre un et trois paramètres numériques, et jusque six paramètres nominaux. Notons que, contrairement aux *features* S.P.L.O.T., les paramètres nominaux peuvent avoir plus de deux valeurs possibles. La Figure 29 montre la taille de l'ensemble d'entraînement généré.

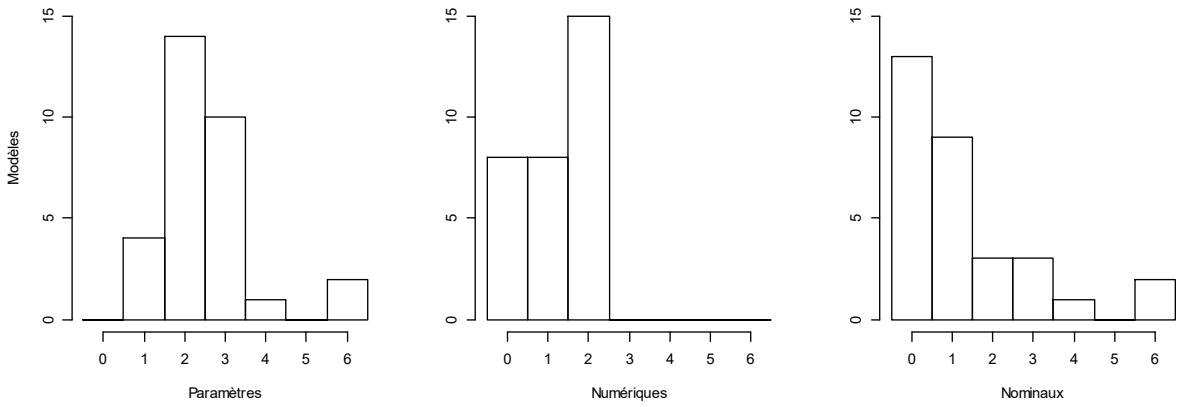


FIGURE 28: REPARTITION DU NOMBRE DE PARAMETRES TOTAUX (GAUCHE), NUMERIQUES (CENTRE) ET NOMINAUX (DROITE)

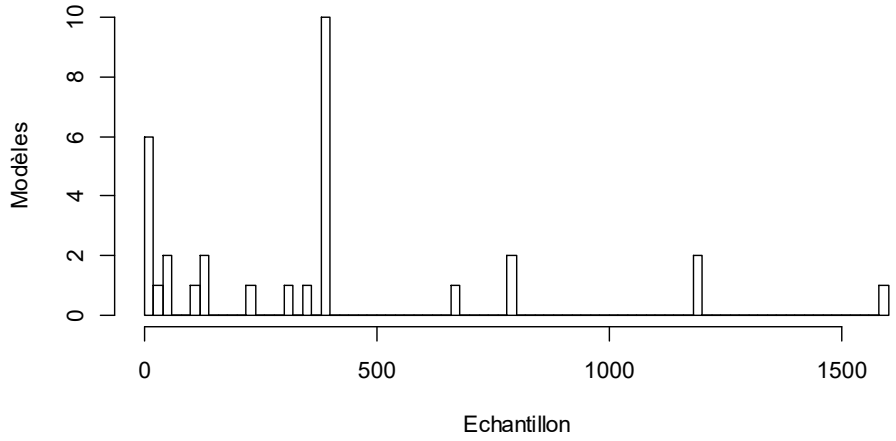


FIGURE 29: REPARTITION DE LA TAILLE DE L'ECHANTILLON UTILISE COMME ENSEMBLE D'ENTRAINEMENT

4.4 Évaluation

Nous avons effectué les évaluations sur les 31 modèles possédant au moins une configuration valide et une, invalide (pour l'ensemble des configurations des deux échantillons). Le Tableau 6 montre les résultats obtenus.

TABLEAU 6: MOYENNE DES RESULTATS DE LA CLASSIFICATION DES MODELES OPENSCAD PAR CLASSIFICATEUR

Classificateur	*Exactitude*	*Précision*	*Rappel*	*F_1-Score*
Hoeffding Tree	0.9384	0.9313	0.9874	0.9567
Naive Bayes	0.9507	0.9473	0.9852	0.9642
C4.5 (J48)	0.9562	0.9531	0.9824	0.9665
SVM (Linear)	0.9477	0.9565	0.9722	0.9523
Logistic Model Tree	0.9633	0.9593	0.9843	0.9708
Logistic Regression	0.9623	0.9568	0.9902	0.9715
*K**	0.9584	0.9527	0.9898	0.9696

REP Tree	0.9612	0.9566	0.9826	0.9688
PART Decision List	0.9614	0.9595	0.9810	0.9692
RIPPER (JRip)	0.9628	0.9578	0.9835	0.9696
Random Committee	0.9680	0.9692	0.9793	0.9735
Multilayer Perceptron	0.9684	0.9627	0.9902	0.9753
SVM (PUK ω=1, σ=0.1)	0.9688	0.9635	0.9890	0.9749
Random Forest	0.9673	0.9692	0.9780	0.9729
SVM (Poly. d=3,γ=2,c0=0.5)	0.9368	0.9555	0.9639	0.9543
kNN (IBk)	0.9673	0.9692	0.9780	0.9729
SVM (RBF γ=5)	0.9506	0.9715	0.9412	0.9304

Les résultats obtenus sont tous assez haut, le F_1-Score étant partout supérieur à 0,95. Notons que, contrairement aux évaluations des modèles S.P.L.O.T., nous travaillons avec un ensemble d'entraînement qui diffère de celui de validation, ce qui explique qu'aucun algorithme n'obtienne un F_1-Score de 1.

Les graphiques suivant (Figure 30 et Figure 31) montrent les détails de la répartition des résultats. Le premier montre la répartition pour l'ensemble des modèles, tandis que le deuxième reprend uniquement les modèles n'ayant que des paramètres nominaux (énumérés). Pour ceux-ci, nous constatons qu'une partie des algorithmes, notamment les arbres de décision et les algorithmes basés sur les règles, ont une variation du F_1-Score bien plus importante.

FIGURE 30: REPARTITION DU F_1-SCORE PAR CLASSIFICATEUR POUR LA CLASSIFICATION DES MODELES OPENSCAD

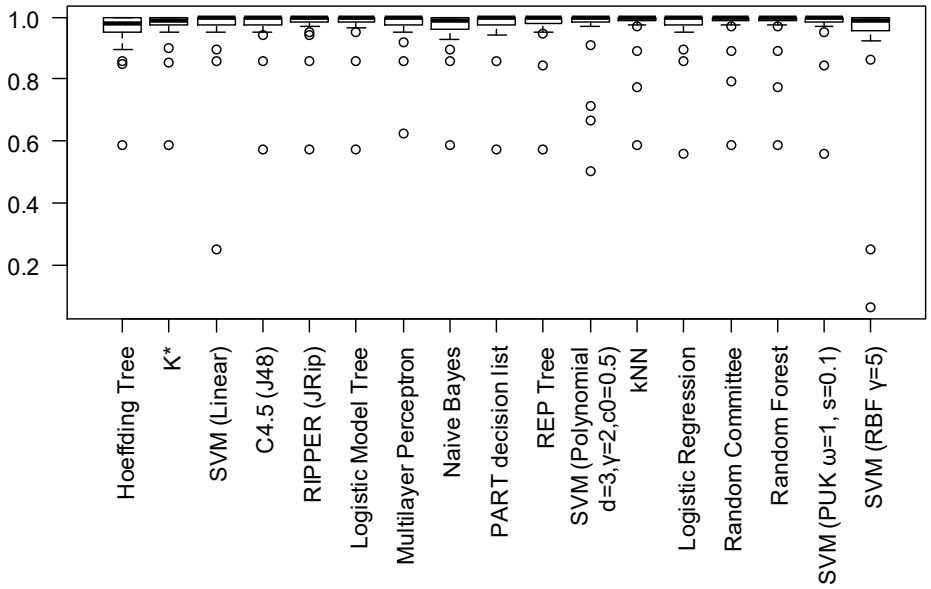
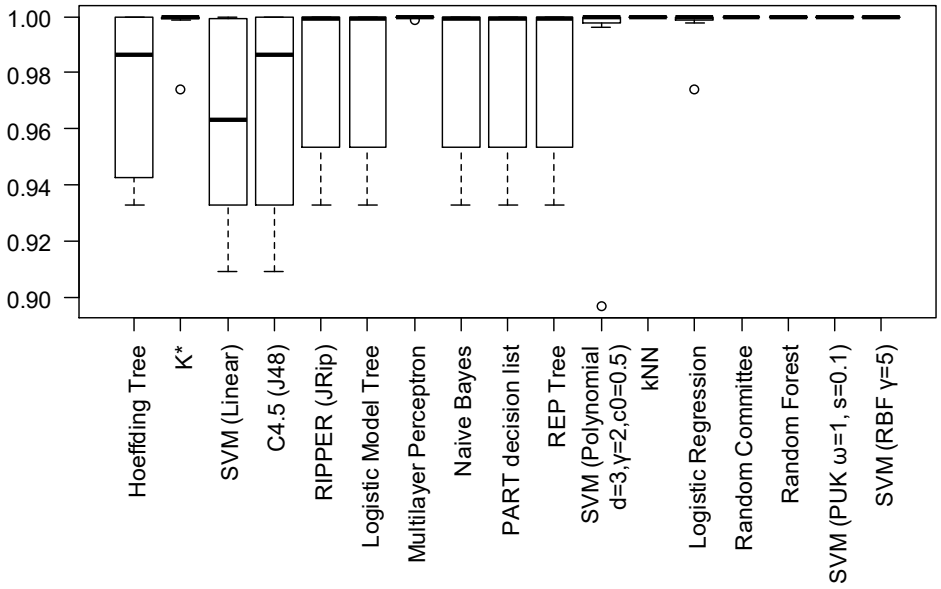


FIGURE 31: REPARTITION DU F_1-SCORE PAR CLASSIFICATEUR POUR LA CLASSIFICATION DES MODELES OPENSCAD N'AYANT QUE DES PARAMETRES NOMINAUX



Ce comportement est similaire à celui constaté sur les modèles S.P.L.O.T., où ces mêmes modèles obtiennent des scores assez bas, même avec un échantillonnage complet. Il semble apparaître que ces algorithmes ont plus de mal à généraliser des paramètres nominaux que numériques.

4.4.1 Exemples

4.4.1.1 « Hopper Upgrade Extension »

Le premier exemple [42] est un modèle OpenSCAD simple, cylindrique, avec un seul paramètre numérique : « coupling inner dimension » dont le domaine est $[30\,;40]$. Ce modèle génère en fait un cylindre dont le diamètre interne est 32 et le diamètre externe est « coupling inner dimension ».

Le Tableau 7 montre les échantillons générés pour le modèle.

TABLEAU 7: ECHANTILLON D'ENTRAINEMENT (A GAUCHE) ET DE VALIDATION (A DROITE) GENERES POUR LE MODELE « HOPPER UPGRADE EXTENSION ».

Validité	dimension
INVALID	30
INVALID	30.526316
INVALID	31.052632
INVALID	31.578947
VALID	32.105263
VALID	32.631579
VALID	33.157895
VALID	33.684211
VALID	34.210526
VALID	34.736842
VALID	35.263158
VALID	35.789474

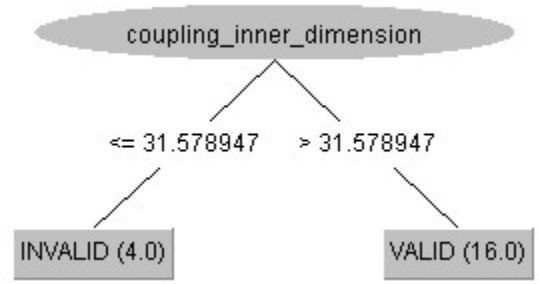
Validité	dimension
INVALID	30
INVALID	30.5
INVALID	31
INVALID	31.5
INVALID	32
VALID	32.5
VALID	33
VALID	33.5
VALID	34
VALID	34.5
VALID	35
VALID	35.5

VALID	36.315789
VALID	36.842105
VALID	37.368421
VALID	37.894737
VALID	38.421053
VALID	38.947368
VALID	39.473684
VALID	40

VALID	36
VALID	36.5
VALID	37
VALID	37.5
VALID	38
VALID	38.5
VALID	39
VALID	39.5
VALID	40

Pour ce modèle simple, tous les algorithmes ont obtenu le même résultat : 16 vrai positifs, 1 faux positif et 4 vrai négatifs, soit un F_1-Score de 0,97. Ce faux-positif est dû au fait que, dans l'ensemble d'entraînement, tous les résultats inférieurs ou égaux à 31,578947 sont invalides, tandis que tous ceux supérieurs ou égaux à 32,105263 sont valides. Pour la valeur de l'ensemble d'évaluation 32, qui se situe entre ces deux valeurs, le résultat inféré par les algorithmes n'est pas correct.

Les algorithmes construisent tous un classificateur correct pour les valeurs inférieures à 31,578947 et supérieure à 32,105263 ; mais entre ces valeurs, ils se comportent différemment. Les algorithmes de type « règles » ou « arbre » utilisent généralement une valeur de l'ensemble d'entraînement comme délimiteur d'une comparaison, comme le montre la Figure 32.




```
(coupling_inner_dimension <= 31.578947)
  => validity=INVALID
()
  => validity=VALID
```

FIGURE 32: REPRESENTATION DES CONFIGURATEURS GENERES PAR C4.5 (A GAUCHE) ET RIPPER (A DROITE)

4.4.1.2 « Holesphere »

Ce second modèle, [43], possède deux paramètres numériques : « sphere radius » et « hole radius », dont les domaines sont respectivement $[2; 100]$ et $[1; 100]$. Ce modèle représente une sphère percée de trois trous cylindriques, parallèles aux axes et passant par son centre. Les deux paramètres représentent le rayon de la sphère et des cylindres. Comme le montre la Figure 33, le rayon des trous devrait rester inférieur au rayon de la sphère, avec une certaine marge.

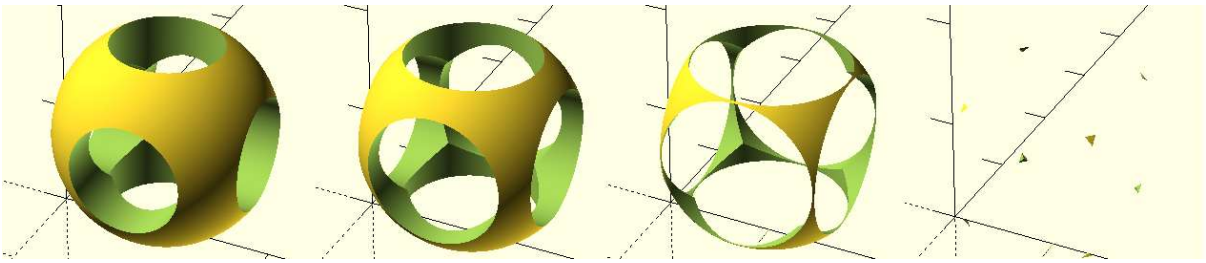


FIGURE 33: DIFFERENTS RENDUS DE « HOLESPHERE » AVEC *SPHERE_RADIUS*=10 ET *HOLE_RADIUS* PRENNANT LES VALEURS (DE GAUCHE A DROITE) 5, 6, 7 ET 8

La Figure 34 montre l'ensemble de validation, placé sur un graphique avec les paramètres pour abscisse et ordonnée. On constate que les configurations valides forment un demi-plan, qui pourrait être représenté par une contrainte du $\mathrm{hole\_radius} < a.\,\mathrm{sphere\_radius} + b$ (avec a et b des constantes à déterminer). Cela étant, les algorithmes de types « règles » ou « arbre » ici étudiés ne peuvent générer de telles contraintes, et les configurateurs obtenus ont un plus grand nombre de contraintes (voir Figure 49 et Figure 50).

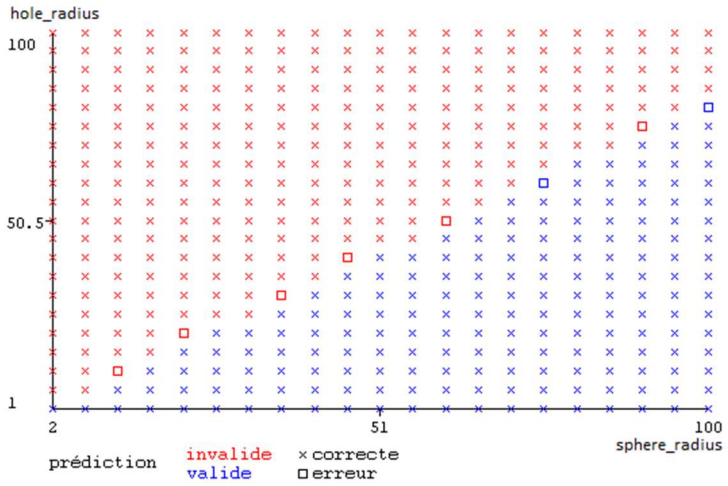


FIGURE 34: CLASSIFICATION EFFECTUEE SUR L'ENSEMBLE D'EVALUATION PAR LE CLASSIFICATEUR C4.5 (J48)

4.4.1.3 « BLTouch mounting adapter for remixed Micromake effector »

Le troisième exemple, [44], est une pièce détachée pour imprimante 3D. Cette pièce possède quatre paramètres : « horizontal_offset », « vertical_offset », « base_height » et « hole_diameter ». Leurs domaines respectifs sont $[0;5]$, $[0;50]$, $\{1,2,3\}$ et $[0;3]$.

Avec ces paramètres et domaines, une grande majorité des configurations s'avère valide. En fait, le seul cas où des configurations sont invalides est lorsque $\mathrm{horizontal\_offset} = 2{,}5$: comme le montre la Figure 35, pour cette valeur, les deux trous cylindriques verticaux du modèle sont tangents, et le modèle résultant n'est pas une variété (*non-manifold*).

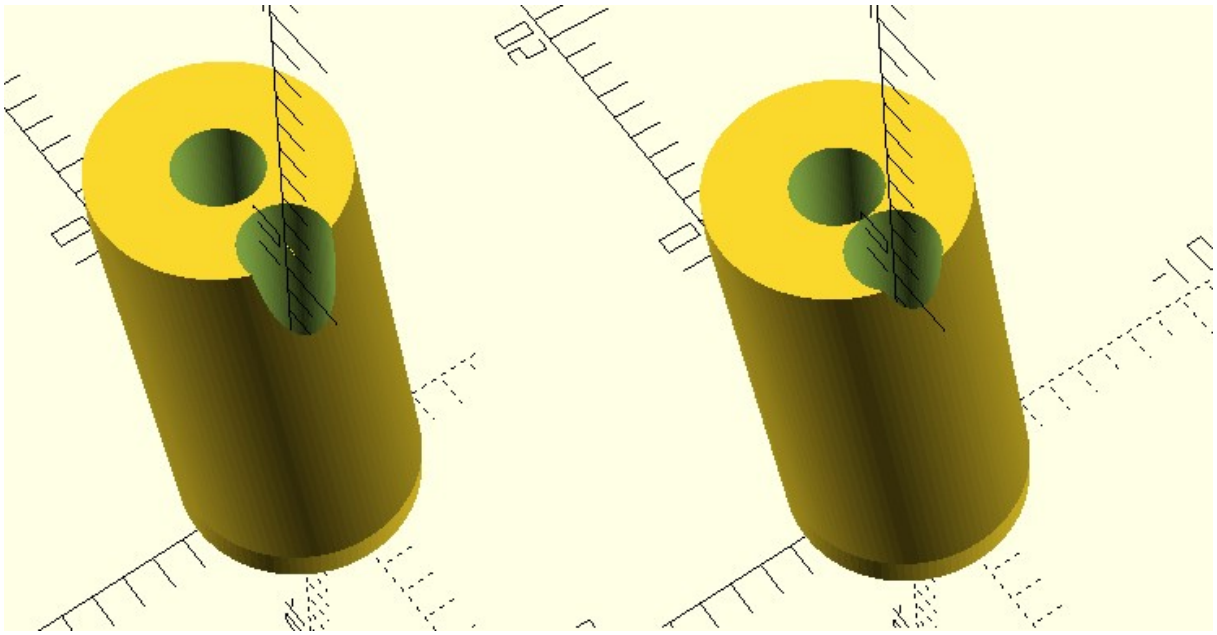


FIGURE 35: RENDUS DE « BLTOUCH MOUNTING ADAPTER FOR REMIXED MICROMAKE EFFECTOR » AVEC HORIZONTAL_OFFSET = 4 (A GAUCHE) ET 2.5 (A DROITE)

Cela étant, la condition est une valeur précise, dans un domaine continu. Comme nous échantillonnons les domaines continus avec un nombre fini de valeurs, il est tout à fait possible que ce cas particulier ne soit pas du tout représenté dans nos ensembles d'entraînement et de validation.

Dans ce cas particulier, la valeur $2{,}5$ se retrouve dans notre ensemble de validation, mais pas dans l'ensemble d'entraînement. Celui-ci ne comporte en fait que des configurations valides, et nous obtenons donc des classificateurs donnant toujours le résultat « valide ». Dès lors, tous les configurateurs obtiennent le même résultat à la validation, comme le montre le Tableau 8.

TABLEAU 8 : RESULTAT DE LA VALIDATION D'UN CLASSIFICATEUR ENTRAINE SUR LE MODELE « BLTOUCH MOUNTING ADAPTER FOR REMIXED MICROMAKE EFFECTOR »

Vrai positif	*Faux positif*	*Faux négatif*	*Vrai négatif*	*Précision*	*Rappel*	*F_1-Score*
1260	63	0	0	0.952	1	0.976

4.4.1.4 « Blizzard of Unique Snowflakes »

Ce modèle, [45], possède un unique paramètre numérique, « seed », dont le domaine est $[0; 1000000]$. Ce paramètre est en fait utilisé comme « graine » pour le générateur de nombre aléatoires de la fonction OpenSCAD « rands ». Cette fonction est prévue pour donner des résultats complétement différent pour des valeurs de « seed » différentes, ainsi, la plus petite variation possible[21] du paramètre donne des résultats très différents, comme le montre la Figure 36.

[21] Pour un nombre a virgule flottante 64 bit IEEE

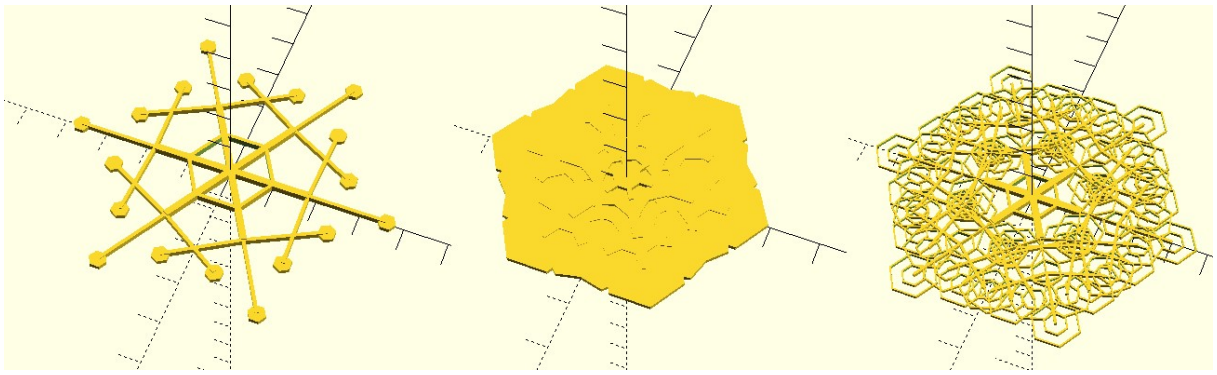


FIGURE 36: RENDUS DE « BLIZZARD OF UNIQUE SNOWFLAKES »AVEC POUR « SEED » 1 (A GAUCHE), 1,0000000000000002 (AU CENTRE) ET 1,0000000000000004 (A DROITE).

Pour ce modèle, les résultats ont été très mauvais, comme le montre le Tableau 9.

TABLEAU 9: RESULTATS DE L'EVALUATION DES CLASSIFICATEURS POUR « BLIZZARD OF UNIQUE SNOWFLAKES »

Classificateur	*Précision*	*Rappel*	*F_1-Score*
SVM (Poly. d=3,γ=2,c0=0.5)	0.353	0.857	0.5
Random Forest	0.5	0.714	0.588
Random Committee	0.5	0.714	0.588
REP Tree	0.571	0.571	0.571
Logistic Model Tree	0.571	0.571	0.571
Multilayer Perceptron	0.556	0.714	0.625
Logistic Regression	0.455	0.714	0.556
C4.5 (J48)	0.571	0.571	0.571
Naive Bayes	0.5	0.714	0.588
SVM (RBF γ=5)	1	0.143	0.25
SVM (Linear)	1	0.143	0.25
SVM (PUK ω=1, σ=0.1)	0.455	0.714	0.556
Hoeffding Tree	0.5	0.714	0.588
kNN (IBk)	0.5	0.714	0.588
*K**	0.5	0.714	0.588
RIPPER (JRip)	0.571	0.571	0.571
PART Decision List	0.571	0.571	0.571

4.5 Limitations

En observant les résultats des évaluations, nous constatons que notre technique comporte plusieurs limitations, dont nous allons à présent discuter. Leurs trois types de limitations rencontrées proviennent de l'échantillonnage, de l'oracle et des algorithmes d'apprentissage.

4.5.1 Échantillonnage

Comme dis précédemment, le grand intérêt d'échantillonner les configurations est que nous pouvons tester un nombre fini de configurations et inférer un résultat global pour ces configurations. Ce faisant, nous faisons l'hypothèse que la variation du résultat « entre » les échantillons est contrôlée : on suppose que, si tous les échantillons autour[22] d'une configuration sont valides, elle le sera aussi.

[22] C'est-à-dire tous les échantillons dont les valeurs des paramètres nominaux sont égales et celles des paramètres numériques sont les plus proches (inférieurs et supérieurs) des valeurs de la configuration.

Lorsque cette hypothèse est respectée, les changements de validité se produisent toujours « entre » des échantillons valides et invalides. Si cette hypothèse est globalement vraisemblable, nous constatons dans la pratique deux cas où son non-respect cause de mauvais résultats.

Le premier apparaît lorsque le changement de validité s'effectue sur un intervalle assez petit, ou une valeur ponctuelle (comme c'est le cas dans l'exemple précédent). La variation de validité pourra aussi bien apparaître uniquement dans l'ensemble d'entraînement, de validation ou dans aucun des deux.

Dès lors, la validation peut très bien obtenir un F_1-Score élevé, tout en « oubliant » certaines valeurs de configurations. Pour les intervalles ainsi oubliés, on peut potentiellement solutionner le problème en augmentant la résolution de l'échantillonnage, mais pour les valeurs ponctuelles, un échantillonnage fini n'est jamais certain de les découvrir.

Le second problème constaté est lorsque le modèle est totalement discontinu par rapport à un de ses paramètres numériques. C'est par exemple le cas pour les modèles possédant un paramètre « seed » utilisé pour générer des nombres aléatoires (comme le modèle « Blizzard of Unique Snowflakes »). Dans ces modèles paramétriques, deux valeurs arbitrairement proches peuvent générer des modèles complétement différents (système chaotique), comme le montre la Figure 36 et il sera très difficile de modéliser correctement leur validité.

4.5.2 Oracle

La seconde limitation rencontrée est la précision de l'oracle. Notre méthode tente en fait de modéliser la variabilité de l'oracle. Dès lors, nos classificateurs représentent seulement la validité du point de vue « oracle ». La correspondance avec une vérité de terrain dépend de la justesse de l'oracle à représenter cette vérité.

Dans le cas des modèles 3D, nous avons choisis des règles assez simples, utilisant des outils OpenSource, pour définir notre oracle, qui ne reflètent pas nécessairement la réalité terrain. L'usage d'outils d'analyse plus sophistiqués permettrait d'obtenir un oracle plus précis.

De plus, nous noterons que notre Oracle ne gère que les paramètres énumérés et numériques. Nous ne gérons pas les paramètres spéciaux du configurateur Thingiverse (les images et les polygones) ou les chaînes de caractères.

4.5.3 Algorithmes

Comme nous l'ont montrés les exemples précédents (en particulier « Holesphere »), certains algorithmes, tels les types « arbres » et « règles », gèrent mal les relations entre paramètres. En outre, les résultats obtenus par les différents algorithmes ne permettent pas de désigner un algorithme particulier comme « meilleur ». La Figure 37 montre les performances comparées des différents algorithmes sur certains modèles. Comme on le constate, chaque algorithme se classe parmi les meilleurs pour certain modèles, et parmi les pire pour d'autre.

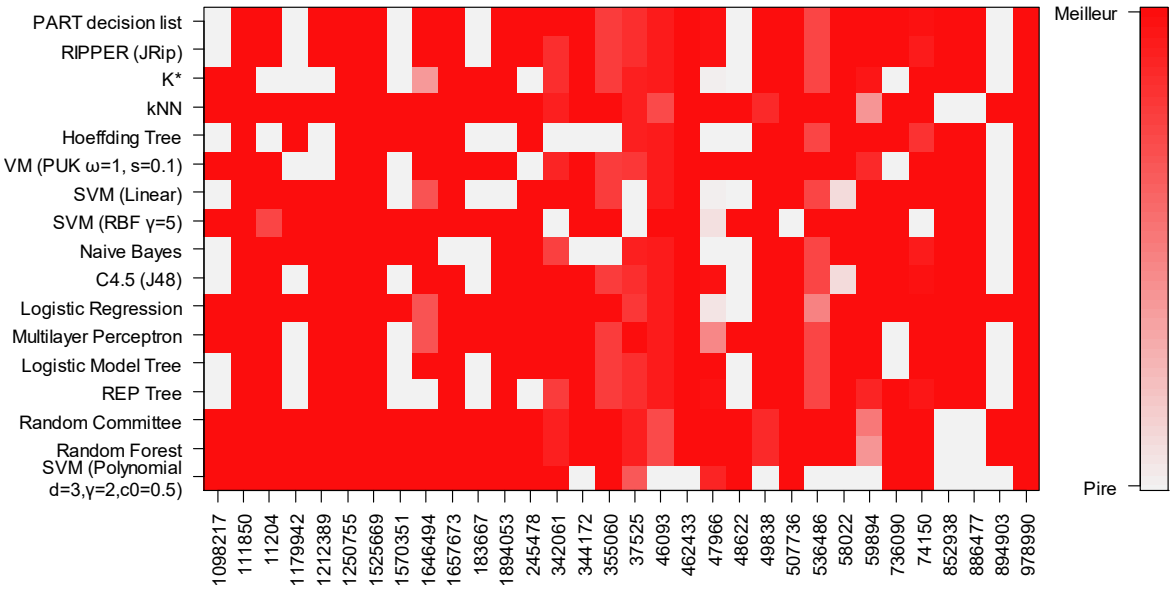


FIGURE 37: RESULTATS COMPARES PAR MODELE (BLANC = F_1-SCORE MINIMUM ET ROUGE = F_1-SCORE MAXIMUM) POUR LES DIFFERENTS ALGORITHMES

4.6 Conclusion

Dans ce chapitre, nous avons utilisé les modèles 3D configurables du site Thingiverse comme modèles de variabilité pour valider notre méthode. Nous avons récupéré un grand nombre de modèles et mis en place des outils afin d'extraire leurs paramètres et de les intégrer à nos tests. Un oracle a ensuite été développé à partir de logiciels spécialisés.

À partir de là, nous avons tenté de générer des échantillons pour ces objets 3D et les avons validés avec notre oracle. Celui-ci s'est avéré très lent pour la validation de nombreux modèles, et nous avons réduit notre analyse à environ 200 modèles. La lenteur de l'oracle justifie en soit la nécessité de modéliser un validateur plus rapide pour la configuration interactive.

Notre méthode permet de générer un classificateur, qu'il est possible d'utiliser interactivement et dont les résultats sont globalement corrects, comparés à ceux de l'oracle. De ce fait, notre validateur offre une valeur ajoutée à l'expérience de l'utilisateur.

Toutefois, notre solution n'est pas exempte de défauts. Les limitations principales sont la trop grande simplicité de l'oracle, l'inconstance des algorithmes et l'inaptitude à gérer des valeurs singulières ou les paramètres irréguliers.

Des outils plus avancés pour l'analyse des fichiers .stl produits par OpenSCAD permettraient d'obtenir un oracle plus sélectif, tandis que le choix d'un algorithme pourrait être fait dynamiquement, après entraînement et validation de plusieurs algorithmes, sur base du F_1-Score obtenu.

Le dernier point peut être amélioré par l'augmentation de la taille de l'échantillon, mais au prix de temps de calcul supplémentaire. Dans notre étude, nous avons opté pour un échantillonnage régulier ; d'autres techniques d'échantillonnage pourraient permettre de découvrir des valeurs singulières en empêchant l'explosion du nombre d'échantillons. Une autre possibilité serait d'intégrer des connaissances métier à la génération d'échantillons (comme par exemple une analyse statique du fichier OpenSCAD pour déterminer les valeurs particulières).

Conclusion

Dans ce travail, nous avons proposé une approche pour la modélisation des contraintes de modèles de variabilité. Cette approche combine l'usage d'algorithmes de classification génériques à un oracle spécifique capable de prédire les valeurs d'un échantillon de configurations.

Dans un premier temps, nous avons introduit les notions de variabilité et de validité, les formalismes de *feature model* et d'*attributed feature model* et définit la notion de *fonction de validité*. Nous avons vu comment cette notion généralisait ces modèles et présenté notre méthode de travail pour modéliser cette fonction.

Ensuite, nous avons présenté les notions de classification. Nous avons présenté un certain nombre d'algorithmes, puis avons discuté de leur lisibilité. Nous avons définit la notion d'erreur et les métriques associées.

Nous avons ensuite mis en place un protocole pour évaluer l'usage de la classification afin de modéliser la validité de modèles de S.P.L.O.T., en utilisant comme oracle un solveur de contraintes. Nous avons d'abord évalué ceux-ci sur des échantillonnages exhaustifs, puis partiels.

Les évaluations ont révélé que les certains algorithmes fournissaient des résultats satisfaisants, pour un échantillonnage suffisamment complet. Il s'agissait de *SVM*, *kNN*, *Random Forest* et *Multilayer Perceptron*. *PART* ainsi que, dans une moindre mesure, *Random Committee* et *RIPPER* offrent aussi de bons résultats, quoi qu'inférieurs, mais sont sensibles au surentraînement. Les algorithmes d'arbre de décision se sont révélés inadaptés à cet usage.

Nous avons ensuite simulé l'effet de l'ajout de contraintes écrites par un expert, en utilisant des contraintes existantes dans les modèles. Cette technique ne s'est pas avérée augmenter substantiellement les performances des algorithmes.

Dans le dernier chapitre, nous avons développé un oracle pour des modèles OpenSCAD configurables, afin d'évaluer notre approche pour la configuration d'objets 3D configurables issus du site Thingiverse. En partant d'un grand nombre de modèles, récupérés depuis le site, nous avons extrait leurs paramètres et généré des échantillons. Nous avons obtenu un ensemble plus réduit de modèles pour lesquels nous avons pu construire un échantillon en un temps raisonnable.

Le classificateur obtenu par notre méthode offre un intérêt pour la configuration d'objets 3D, mais souffre de plusieurs limitations : l'oracle n'est pas assez spécifique, les paramètres de type plus complexe ne sont pas gérés et les comportements chaotiques diminuent sa précision.

Pour mesurer l'intérêt de notre méthode dans les deux cas étudiés, reprenons nos critères initiaux pour une modélisation de variabilité :

1. ses performances à l'exécution ;
2. sa complexité de modélisation ;
3. son exactitude ;
4. sa compréhensibilité par un humain.

Le temps d'exécution d'un classificateur étant négligeable, nous avons considéré le premier point comme acquis. Pour ce qui est de complexité de modélisation, il nous est impossible d'en juger dans le cas de

S.P.L.O.T., où la modélisation était préexistante. Dans le cas des objets 3D, il faut considérer deux points : le travail humain et le traitement automatisé.

Si l'on compare notre méthode à l'écriture manuelle de contraintes, il est clair que le travail humain est moins important, quoique plus pointu, pour la conception d'un oracle unique, par opposition à l'étude et la modélisation manuelle de chaque objet –surtout lorsque que de nouveaux objets sont publiés chaque jour. Pour notre oracle, nous avons vu que le temps de traitement est non-négligeable ; un oracle plus optimisé permettrait de réduire ce coût.

Les deux derniers points, l'exactitude et la compréhensibilité, sont liés au choix du type de classificateur. Comme nous l'avons vu, les seuls classificateurs compréhensibles sont ceux basés sur des règles et sur des arbres de décision. Malheureusement, les arbres sont parmi les algorithmes obtenant les plus mauvais résultats dans le cas des modèles S.P.L.O.T. Les règles sont donc le meilleur choix lorsque la compréhensibilité est requise.

Pour ce qui est de l'exactitude des contraintes, elle dépend fortement de l'algorithme choisi et de la taille de l'échantillonnage. Dans le cas de S.P.L.O.T., *SVM*, *kNN*, *Random Forest* et *Multilayer Perceptron* ont obtenu les meilleurs résultats. Pour les objets 3D, un classificateur ne s'est pas distingué sur l'ensemble des modèles. Dans ce cas, une approche dynamique peut être envisagée : après génération d'ensembles d'entrainement, utiliser la validation croisée pour sélectionner le meilleur classificateur pour un objet 3D particulier.

Nous noterons, dans le cas de S.P.L.O.T. et des *feature models* en général, que les classificateurs sont rarement exacts. Dès lors, si l'on dispose d'un échantillonnage complet du modèle, un classificateur sera inférieur, en termes de justesse, à un modèle obtenu grâce à des méthodes « classiques » de synthèse [4] appliquées aux résultats de l'oracle. Dès lors, la méthode n'est intéressante que dans le cas où nous ne pouvons obtenir un échantillonnage complet –par exemple lorsque le coût pour échantillonner l'espace entier des solutions est trop important.

L'approche s'avère plus intéressante en présence d'attributs numériques, comme pour les objets 3D. Dans ce cas, l'usage de l'apprentissage automatique pour généraliser un échantillon partiel prend tout son sens.

Perspectives

Nous avons confirmé l'intérêt de la méthode dans le cadre des modèles 3D, mais plusieurs améliorations peuvent néanmoins être appliquées à celle-ci. La plus importante concerne l'amélioration de l'oracle, notamment par l'usage d'outils plus avancés pour l'analyse des fichiers .stl produits par OpenSCAD.

Une autre amélioration possible à rapport à l'échantillonnage. Nous pourrions bien sûr augmenter le nombre d'échantillons, mais cela augmenterais en conséquence le temps d'exécution de la prédiction. Afin de maintenir une balance entre ce temps et la précision, une meilleure sélection des échantillons est à envisager.

S'il existe de nombreuses techniques génériques d'échantillonnage [46], nous pouvons aussi envisager une forme d'échantillonnage dynamique, inspirée de méthodes d'analyse numérique comme la dichotomie, afin de déterminer plus précisément les limites entres les classes de validité.

Une autre possibilité est d'effectuer un d'échantillonnage intelligent basé sur le modèle 3D. L'analyse du fichier OpenSCAD peut permettre de déterminer des valeurs singulières, telles des divisions par zéro, des

intersections vides, des dimensions nulles ou négatives, etc. Typiquement, ces valeurs singulières se situent à la limite entre différentes classes.

Bibliographie

[1] Q. Boucher, A. Classen, P. Faber et P. Heymans, «Introducing TVL, a text-based feature modeling language,» chez *VaMoS'10*, 2010, p. 159–162..

[2] Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher et P. Heymans, «Towards more reliable configurators: A re-engineering perspective,» chez *PLEASE 2012*, 2012.

[3] D. Benavides, S. Segura et A. Ruiz-Cortes, «Automated analysis of feature models 20 years later: a literature review,» *Information Systems,* vol. 35, n° %16, 2016.

[4] G. Bécan, R. Behjati, A. Gotlieb et M. Acher, «Synthesis of Attributed Feature Models From Product Descriptions: Foundations,» chez *SPLC'15*, 2015.

[5] K. Czarnecki et A. Wasowski, «Feature diagrams and logics: There and back again,» chez *SPLC'07*, 2007.

[6] M. Acher, P. Heymans, A. Cleve, J.-L. Hainaut et B. Baudry, «Support for reverse engineering and maintaining feature models,» chez *VaMoS'13*, ACM, 2013.

[7] K. Kang, S. Cohen, J. Hess, W. Novak et A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[8] T. Thüm, S. Apel, C. Kästner, I. Schaefer et G. Saake, «A classification and survey of analysis strategies for software product lines,» *ACM Comput. Surv.,* vol. 47, n° %11, p. 45, 2014.

[9] M. Mendonca, M. Branco et D. Cowan, « S.P.L.O.T. - Software Product Lines Online Tools,» chez *OOPSLA'09 (companion)*, 2009.

[10] «Parametric Mendel90 Filament Spool Holder by Think3dPrint3d - Thingiverse,» [En ligne]. https://www.thingiverse.com/thing:172484/files. [Accès le 22 Aout 2018].

[11] N. Andersen, K. Czarnecki, S. She et A. Wąsowski, «Efficient synthesis of feature models,» chez *Proceedings of SPLC'12*.

[12] T. Berger, S. She, R. Lotufo, K. Czarnecki et A. Wąsowski, «Feature-to-code mapping in two large product lines,» chez *SPLC*, 2010.

[13] P. Temple, J. A. G. Duarte, M. Acher et J.-M. Jézéquel, «Using Machine Learning to Infer Constraints for Product Lines,» chez *SPLC*, Beijing, China, 2016.

[14] S. Shalev-Shwartz et S. Ben-David, «Preface,» chez *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.

[15] N. J. Nilsson, «1.1 What is Machine Learning?,» chez *Introduction to Machine Learning (An Early Draft to a proposed textbook)*, Stanford University, 1998.

[16] N. J. Nilsson, «1.2 Learning Input-Output Functions,» chez *Introduction to Machine Learning (An Early Draft to a proposed textbook)*, Stanford University, 1998.

[17] N. J. Nilsson, «1.3 Learning Requires Bias,» chez *Introduction to Machine Learning (Draft of Incomplete Notes)*, Stanford University, 1998.

[18] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «3.3 Trees,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[19] S. Shalev-Shwartz et S. Ben-David, «18 Decision Trees,» chez *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.

[20] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «4.3 Divide-and-conquer: constructing decision trees,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[21] S. Jayanthi et S.Sasikala, «REPTree Classifier for indentifying Link Spam In Web Search Engines,» *IJSC,* vol. 3, n° %12, pp. 498-505, 2013.

[22] «HoefflingTree - Weka documentation,» [En ligne]. http://weka.sourceforge.net/doc.stable-3-8/weka/classifiers/trees/HoeffdingTree.html. [Accès le 22 Aout 2018].

[23] G. Hulten, L. Spencer et P. Domingos, «Mining time-changing data streams,» chez *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, ACM Press, 2001, pp. 97-106.

[24] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «3.4 Rules,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[25] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «6.2 Classification rules,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[26] E. Frank et I. H. Witten, «Generating Accurate Rule Sets Without Global Optimization,» chez *Fifteenth International Conference on Machine Learning*, Morgan Kaufmann, 1998, pp. 144-151.

[27] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «4.2 Statistical modeling,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[28] S. Russel et P. Norvig, «13.5 Bayes' Rule and its uses,» chez *Artificial Intelligence: A Modern Approach (3rd Edition)*, Prentice Hall, 2009.

[29] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «4.6 Linear models,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[30] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «4.7 Instance-based nearling,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[31] J. G. Cleary et L. E. Trigg, «K*: An Instance-based Learner Using an Entropic Distance Measur,» chez *12th International Conference on Machine Learning*, 1995, pp. 108-114.

[32] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[33] «RandomCommittee - Weka Documentation,» [En ligne]. http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/RandomCommittee.html. [Accès le 22 Aout 2018].

[34] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «8.6 Interpretable Ensembles,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[35] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «6.4 Extending linear models,» chez *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*, Morgan Kaufmann, 2011.

[36] Alisneaky, «File:Kernel Machine.png,» 17 Avril 2011. [En ligne]. https://commons.wikimedia.org/wiki/File:Kernel_Machine.png. [Accès le Aout 2018].

[37] D. J. C. MacKay, «38. Introduction to Neural Networks,» chez *Information Theory, Inference and Learning Algorithms*, University of Cambridge, 2003.

[38] D. J. MacKay, «44.1 Multilayer perceptrons,» chez *Information Theory , Inference, and Learning Algorithms*, University of Cambridge, 2003.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, V, J. erplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot et E. Duchesnay, «Scikit-learn: Machine Learning in Python,» *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[40] I. H. Witten, E. Frank, M. A. Hall et C. J. Pal, «1.1 What's in WEKA,» chez *The WEKA Workbench (Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques" Morgan Kaufmann, Fourth Edition, 2016)*, 2016.

[41] «SXFM Format - SPLOT,» [En ligne]. http://52.32.1.180:8080/SPLOT/sxfm.html. [Accès le 22 Aout 2018].

[42] «Customizer (Hopper Upgrade Extension),» [En ligne]. https://www.thingiverse.com/apps/customizer/run?thing_id=111850. [Accès le 22 Aout 2018].

[43] «Customizer (Holesphere),» [En ligne]. https://www.thingiverse.com/apps/customizer/run?thing_id=74150. [Accès le 22 Aout 2018].

[44] «Customizer (BLTouch mounting adapter for remixed Micromake effector),» [En ligne]. https://www.thingiverse.com/apps/customizer/run?thing_id=1657673. [Accès le 22 Aout 2018].

[45] «Customizer (Blizzard of Unique Snowflakes),» [En ligne]. https://www.thingiverse.com/apps/customizer/run?thing_id=37525. [Accès le 22 Aout 2018].

[46] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi et S. Apel, A Comparison of 10 Sampling Algorithms for Configurable Systems, 2016.

[47] «Developer Documentation - Thingiverse,» [En ligne]. http://customizer.makerbot.com/docs. [Accès le 22 Aout 2018].

Annexe A : Résultats S.P.L.O.T. détaillés

FIGURE 38: EVOLUTION DE LA PRECISION EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (PREMIERE PARTIE)

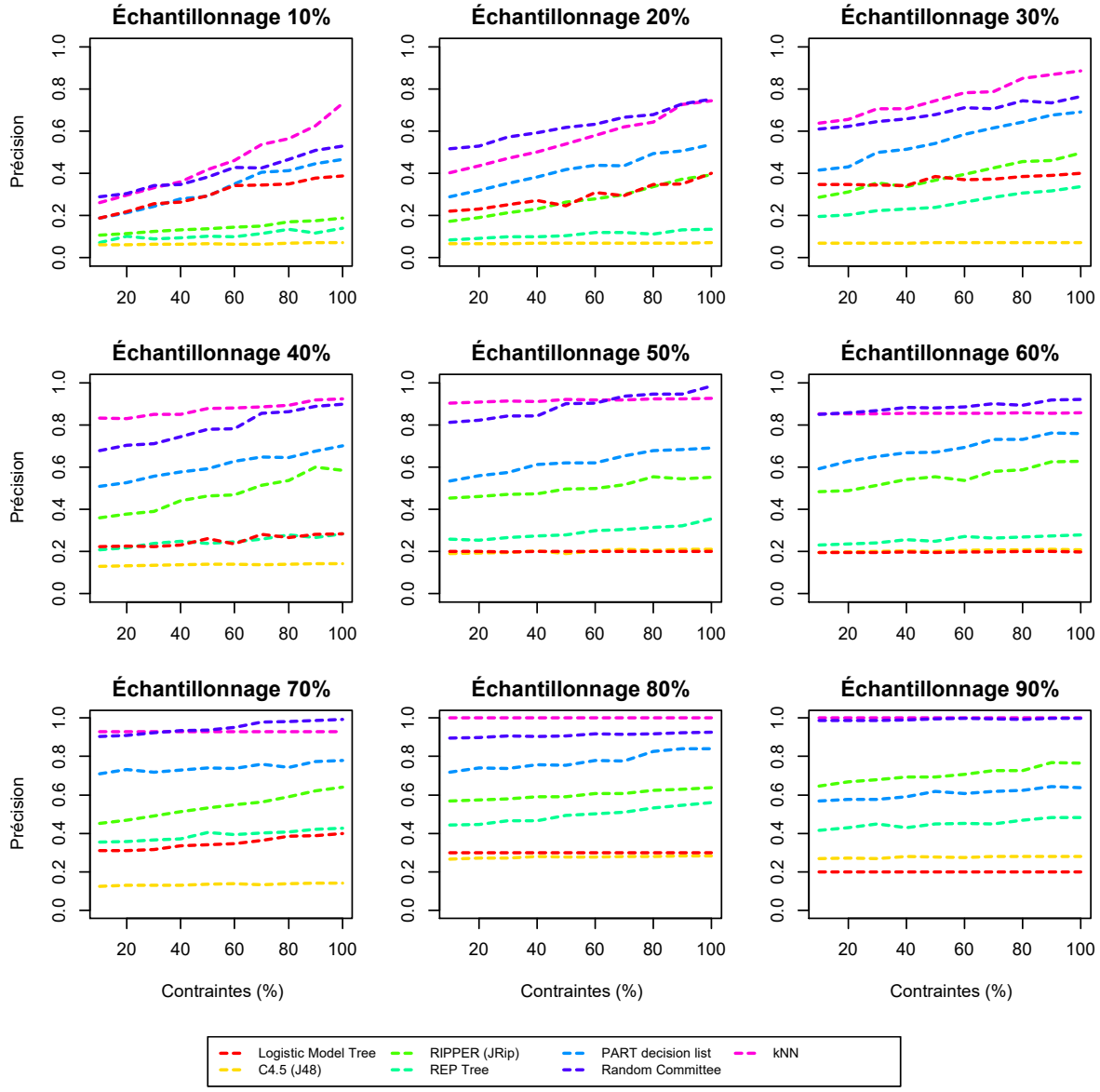


FIGURE 39: EVOLUTION DU F_1-SCORE EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (SECONDE PARTIE)

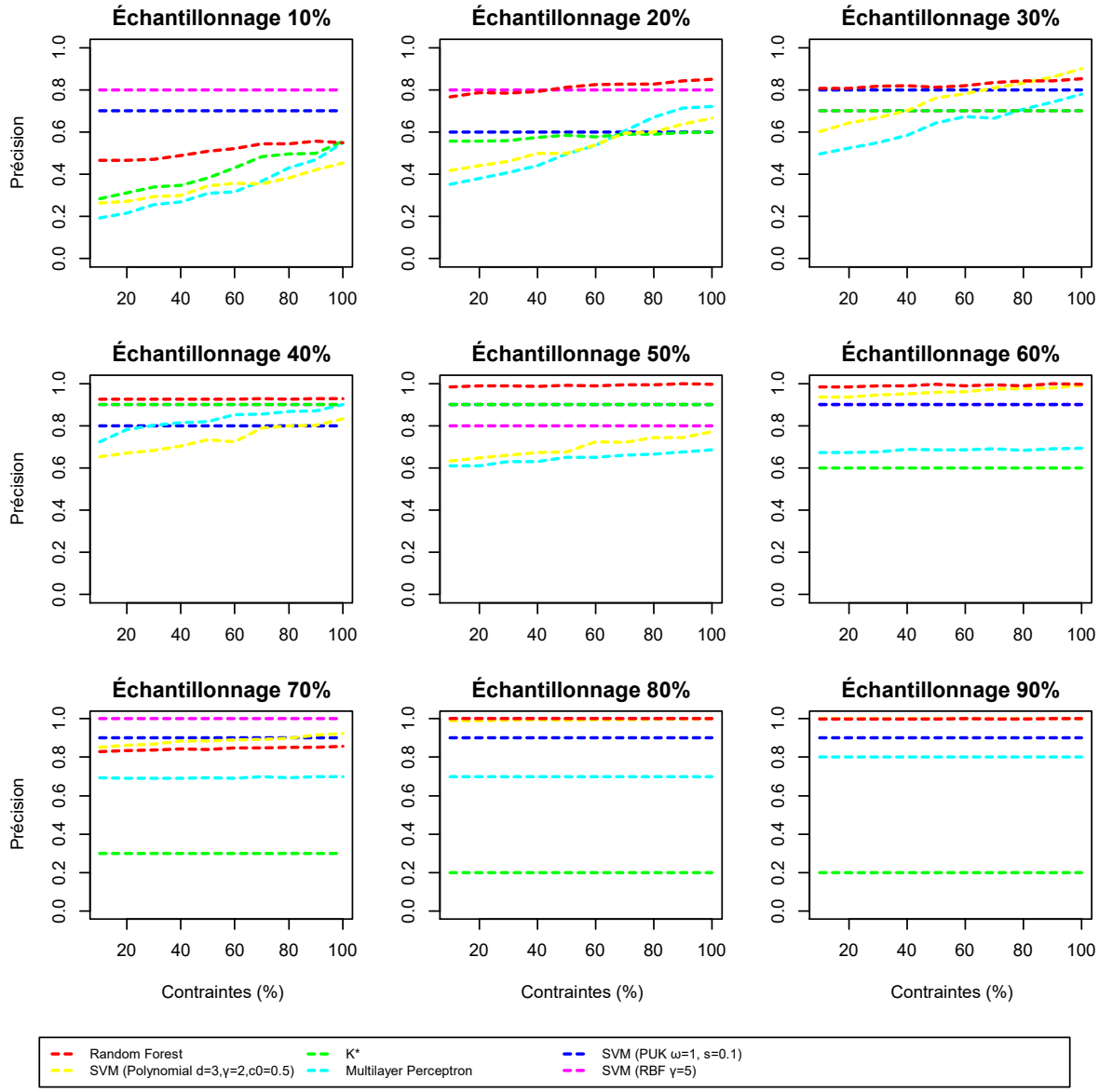


FIGURE 40: EVOLUTION DU F_1-SCORE EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (PREMIERE PARTIE)

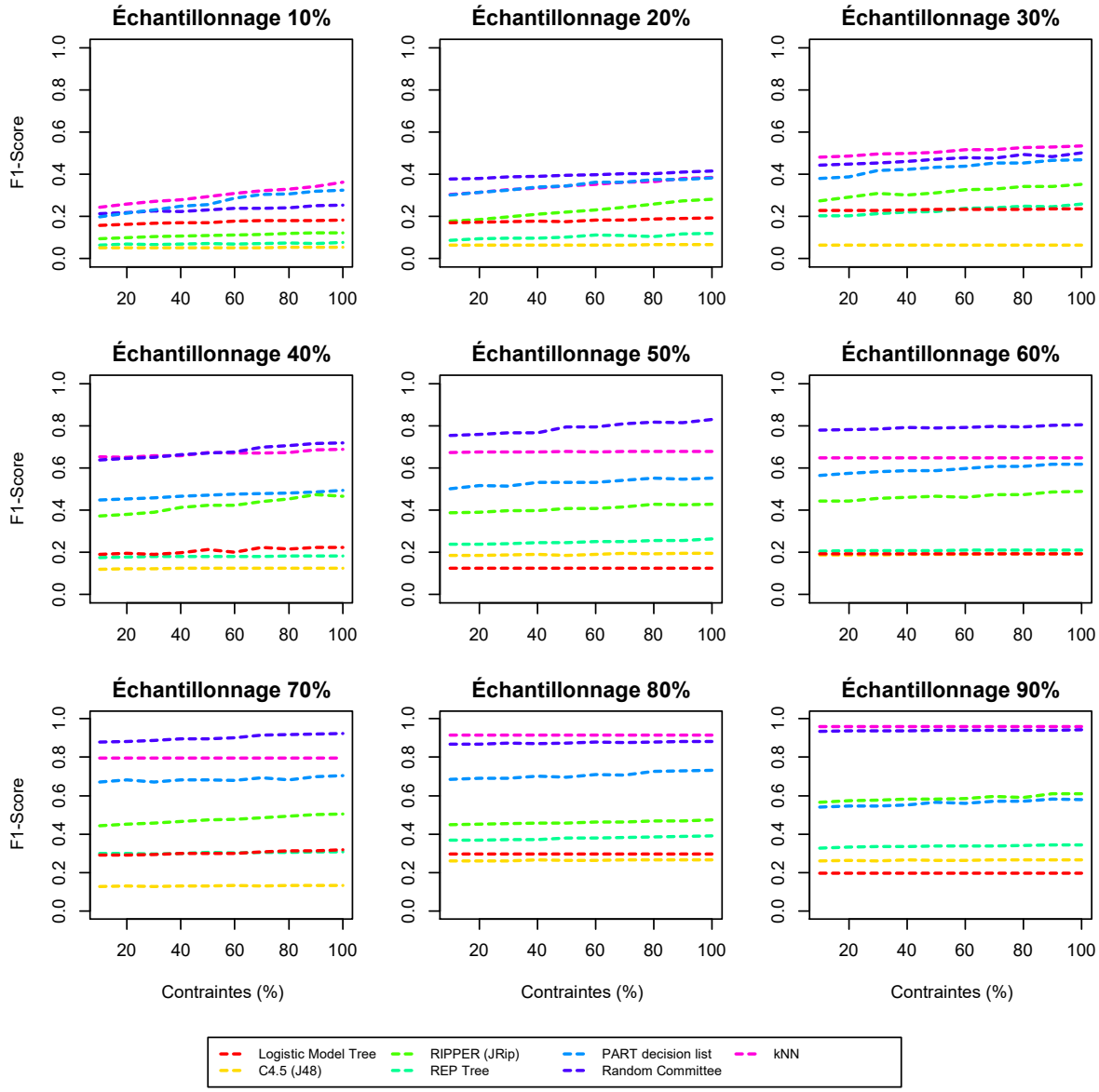


FIGURE 41: EVOLUTION DU F_1-SCORE EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (SECONDE PARTIE)

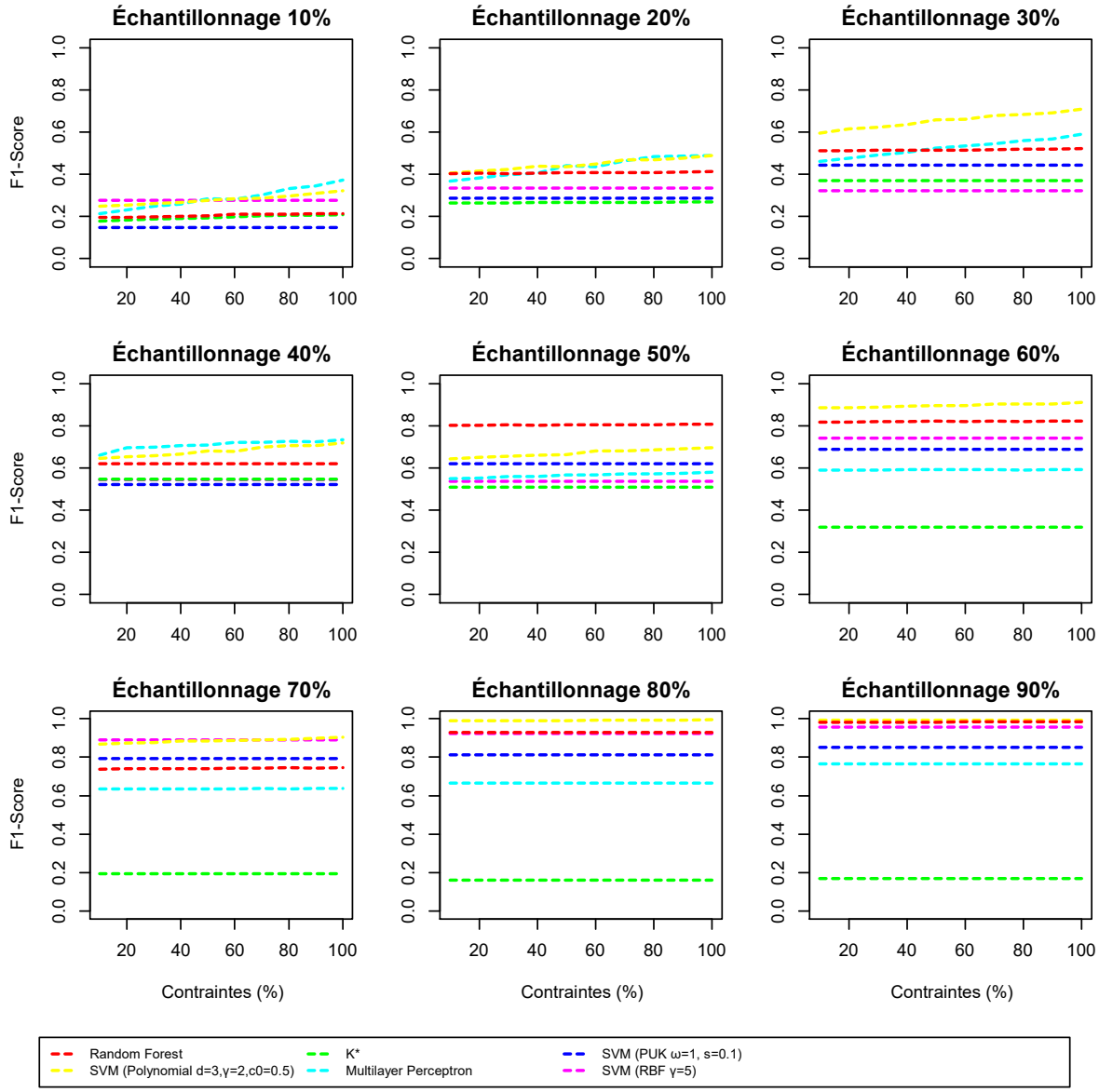


FIGURE 42: VARIATION DU F_1-SCORE EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (PREMIERE PARTIE)

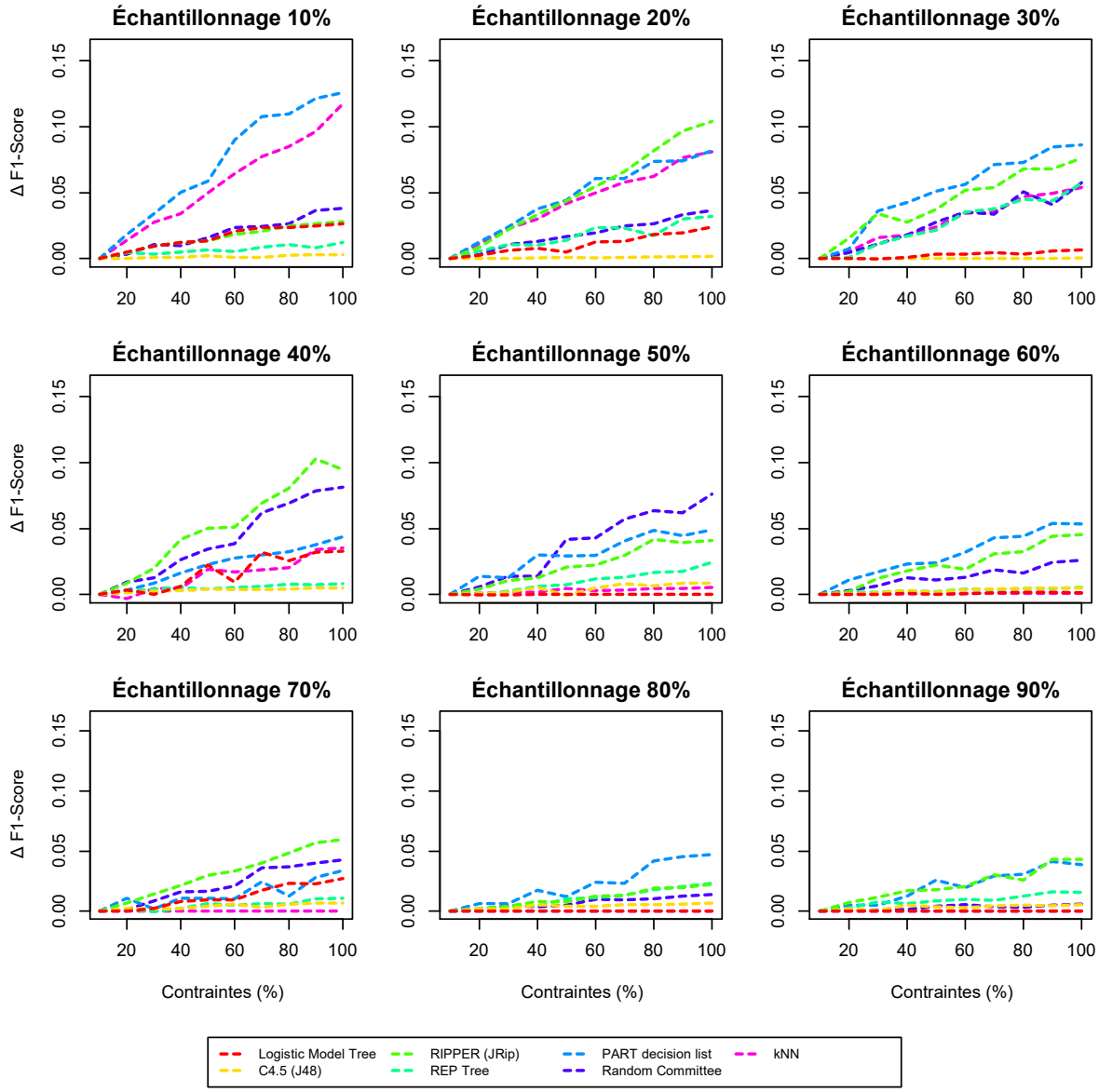
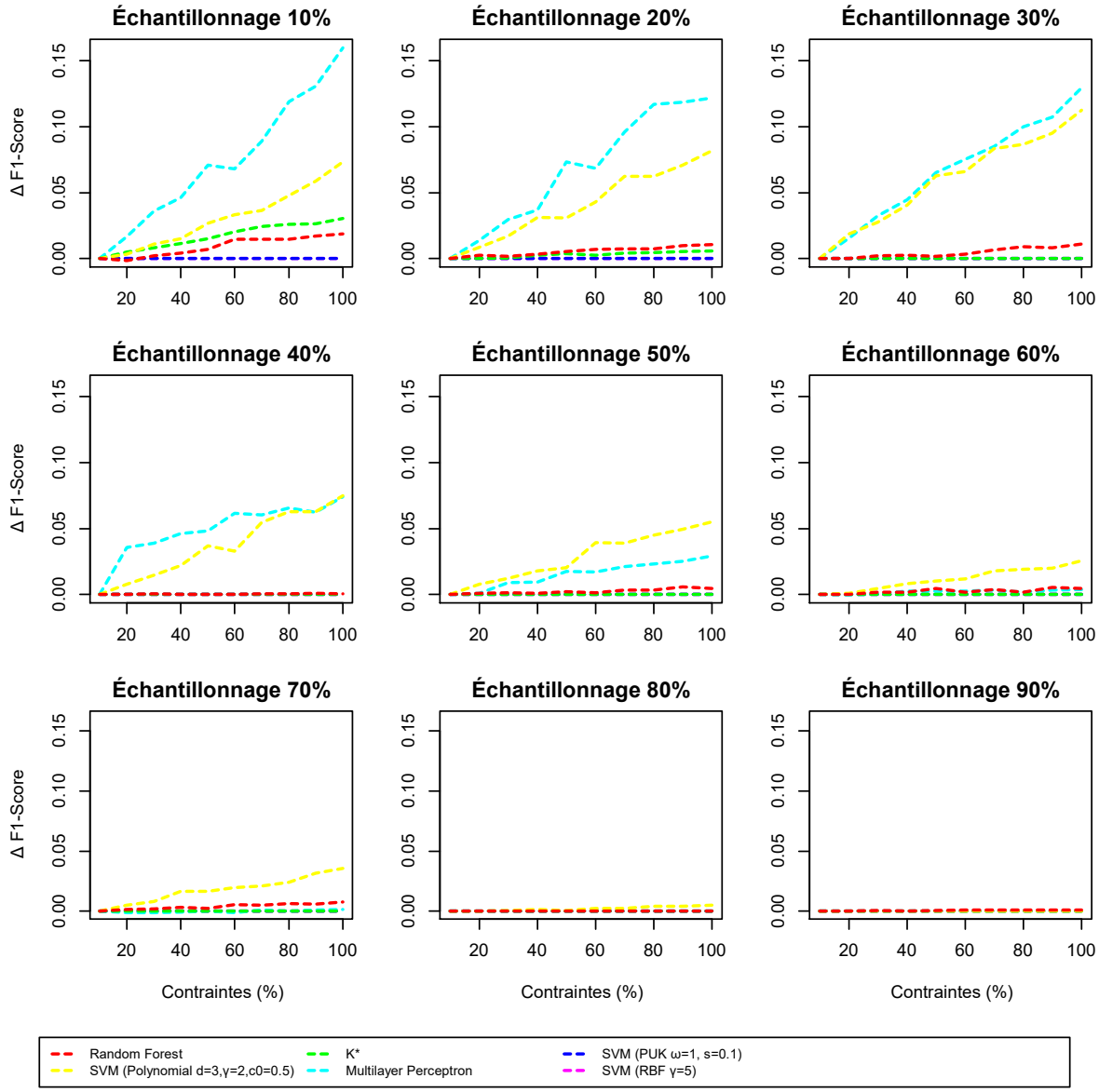


FIGURE 43: VARIATION DU F_1-SCORE EN FONCTION DES CONTRAINTES AJOUTEES, POUR DIFFERENTS ECHANTILLONNAGES PARTIELS (SECONDE PARTIE)

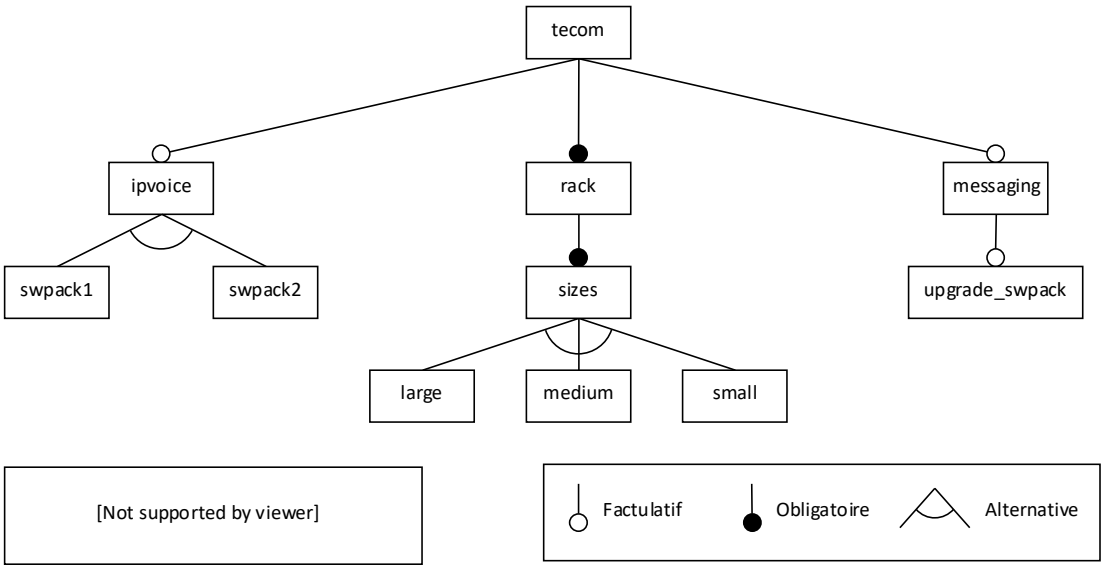


Annexe B : Modèle S.P.L.O.T. « télécom »

FIGURE 44: MODELE S.L.P.O.T "TELECOM", AU FORMAT SXFM

```
<feature_model name="Telecommunication_System">
<meta>
<data name="description">This feature models describes the features of a simple
telecommunication system</data>
<data name="creator">Alexander Felfernig</data>
<data name="email"></data>
<data name="date"></data>
<data name="department"></data>
<data name="organization"></data>
<data name="address"></data>
<data name="phone"></data>
<data name="website"></data>
<data name="reference">Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach,
and Markus Zanker. Towards distributed configuration. In KI '01: Proceedings of
the Joint Ger- man/Austrian Conference on AI, pages 198–212, London, UK, 2001.
Springer-Verlag</data>
</meta>
<feature_tree>
:r tecom
   :o ipvoice(ipvoice)
      :m software
         :g [1,1]
            : swpack1 (swpack1)
            : swpack2
   :m rack
      :m sizes
         :g [1,1]
            : large
            : medium
            : small
   :o messaging (messaging)
      :o upgrade_swpack(upgrade_swpack)
</feature_tree>
<constraints>
c1: ~swpack1 or ~upgrade_swpack
c2: ~ipvoice or ~messaging or upgrade_swpack
</constraints>
</feature_model>
```

FIGURE 45: REPRESENTATION DU MODELE S.P.L.O.T "TELECOM"



Résultats de la classification du modèle « telecom »

Note : les classificateurs obtenus ne représentent pas nécessairement la réalité terrain du modèle.

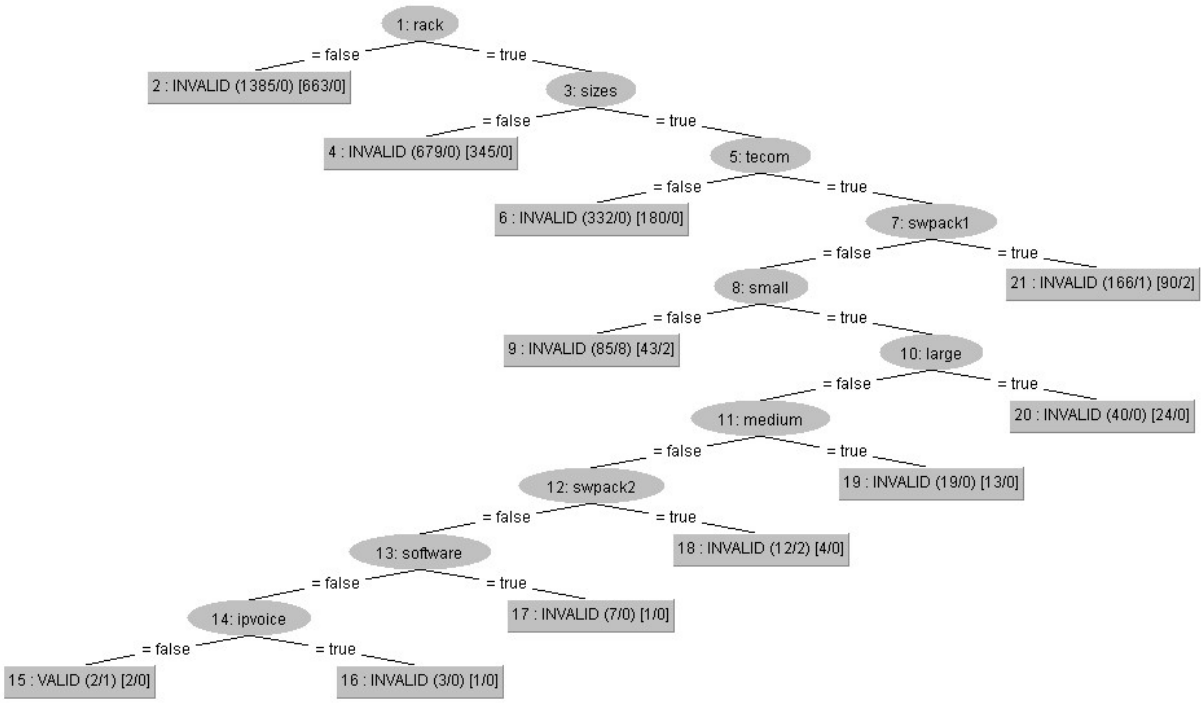
FIGURE 46: REGLES GENEREES PAR L'ALGORITHME RIPPER POUR UN ECHANTILLONNAGE EXHAUSTIF

```
(tecom = true) and (rack = true) and (sizes = true) and (swpack1 = false)
  and (large = false) and (ipvoice = false) and (software = false) and (swpack2 = false)
  and (messaging = true) and (medium = false) and (small = true)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (swpack1 = false)
  and (small = false) and (ipvoice = false) and (software = false) and (swpack2 = false)
  and (messaging = true) and (large = false) and (medium = true)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (swpack1 = false)
  and (medium = false) and (upgrade_swpack = false) and (messaging = false)
  and (ipvoice = false) and (software = false) and (swpack2 = false)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (swpack1 = false)
  and (small = false) and (upgrade_swpack = false) and (messaging = false)
  and (large = false) and (medium = true)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (medium = false)
  and (swpack1 = false) and (large = true) and (small = false) and (messaging = true)
  and (ipvoice = false) and (software = false) and (swpack2 = false)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (ipvoice = true)
  and (software = true) and (medium = false) and (swpack1 = false) and (swpack2 = true)
  and (large = false) and (small = true)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (ipvoice = true)
  and (software = true) and (small = false) and (swpack1 = false) and (swpack2 = true)
  and (large = true) and (medium = false)
  => VALID
(tecom = true) and (rack = true) and (sizes = true) and (ipvoice = true)
  and (software = true) and (large = false) and (messaging = false)
  and (upgrade_swpack = false) and (swpack2 = false) and (swpack1 = true)
  => VALID
=> INVALID
```

FIGURE 47: REGLES GENEREES PAR L'ALGORITHME PART POUR UN ECHANTILLONNAGE EXHAUSTIF

```
tecom = false: INVALID
rack = false: INVALID
sizes = false: INVALID
swpack1 = true AND ipvoice = false: INVALID
swpack1 = true AND software = false: INVALID
swpack2 = true AND ipvoice = false: INVALID
ipvoice = true AND software = false: INVALID
software = true AND ipvoice = false: INVALID
ipvoice = true AND swpack1 = true AND swpack2 = true: INVALID
ipvoice = true AND swpack2 = false AND swpack1 = false: INVALID
swpack1 = true AND messaging = true: INVALID
large = true AND medium = true: INVALID
small = true AND large = false AND medium = true: INVALID
medium = false AND upgrade_swpack = true AND messaging = false: INVALID
swpack2 = true AND messaging = true AND upgrade_swpack = false: INVALID
medium = true AND upgrade_swpack = false: VALID
messaging = false AND medium = false: VALID
messaging = true AND medium = false: VALID
messaging = false: INVALID
: VALID
```

FIGURE 48: ARBRE DE DECISION GENERE PAR L'ALGORITHME REPTREE POUR UN ECHANTILLONNAGE EXHAUSTIF



Annexe C : format pour le configurateur Thingiverse

Format des commentaires

Le configurateur Thingiverse utilise une syntaxe spécifique de commentaires dans le code source des fichiers OpenSCAD [47].

Paramètres

```
// description
nom_de_variable = valeur_par_defaut; // valeurs possibles
```

Le commentaire sur la ligne précédant une variable sera sa description, et le commentaire sur la même ligne désigne les valeurs possibles. Parmi les valeurs possibles, on retrouve :

- Les énumérations, par exemple
  - numériques : `[0, 1, 2, 3]`
  - de *string* : `[foo, bar, baz]`
  - labélisées : `[10:Small, 20:Medium, 30:Large]`
- Les intervalles, par exemple
  - `[10:100]`
  - `[0:0.5:100]`
- Des types plus complexes, tels que les images, les polygones, etc.

Sections

Les sections permettent d'organiser les différents paramètres. Une définition de section s'applique à tous les paramètres suivants, jusqu'à la prochaine définition de section.

```
/* [Section Name] */
```

En particulier, « hidden » désigne des constantes qui ne seront pas utilisées comme des paramètres.

Annexe D : résultats supplémentaires pour les modèles OpenSCAD

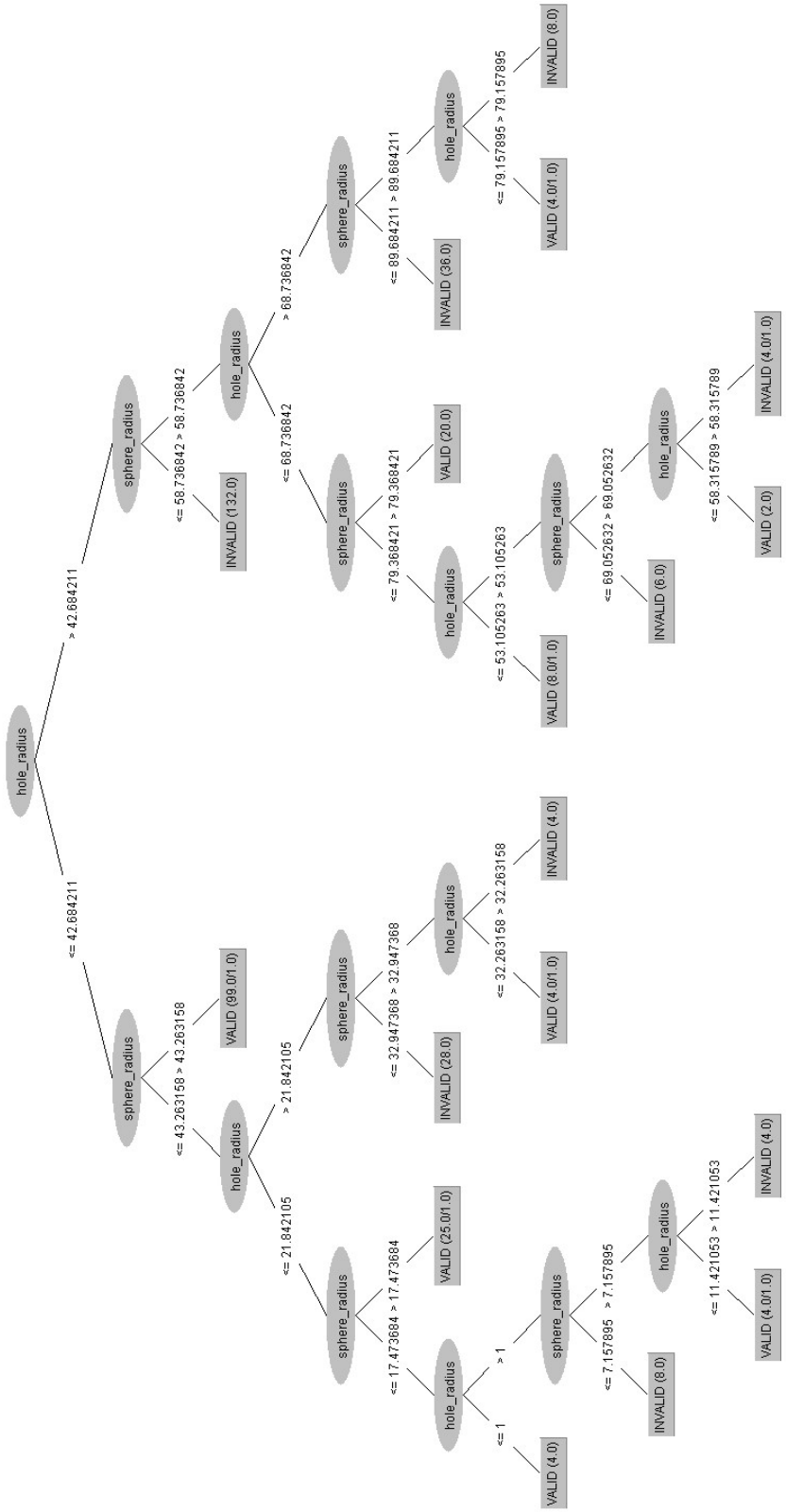


FIGURE 49: CLASSIFICATEUR C4.5 (J48) POUR LE MODELE « HOLESPHERE »

```
hole_radius > 42.684211 AND
sphere_radius <= 58.736842: INVALID (132.0)

hole_radius > 68.736842 AND
sphere_radius <= 89.684211: INVALID (36.0)

sphere_radius > 32.947368 AND
hole_radius <= 53.105263 AND
sphere_radius > 48.421053: VALID (106.0/1.0)

hole_radius <= 16.631579 AND
sphere_radius > 17.473684: VALID (24.0)

sphere_radius <= 32.947368 AND
hole_radius > 6.210526 AND
hole_radius > 21.842105: INVALID (28.0)

hole_radius <= 79.157895 AND
sphere_radius > 79.368421: VALID (16.0/1.0)

hole_radius <= 63.526316 AND
hole_radius > 1 AND
sphere_radius > 7.157895 AND
hole_radius > 32.263158 AND
sphere_radius <= 74.210526: INVALID (12.0/2.0)

hole_radius <= 63.526316 AND
sphere_radius > 22.631579: VALID (13.0/1.0)

hole_radius > 11.421053: INVALID (21.0)

hole_radius <= 1: VALID (4.0)

sphere_radius <= 7.157895: INVALID (4.0)

: VALID (4.0/1.0)
```

FIGURE 50: CLASSIFICATEUR PART POUR LE MODELE « HOLESPHERE »