Rapport de stage

Sujet : Étude et mise en œuvre de techniques d'explicabilité pour les modèles de machine learning

Nom et prénom : Manon Davion

Tuteur de stage : Jean-Marie Lagniez

Établissement : Université d'Artois

Entreprise d'accueil : Centre de Recherche en Informatique de Lens

Date : Mars – Mai 2025

Table des matières

1 Introduction

Aujourd'hui, l'intelligence artificielle est partout, autour de nous. Elle fait partie intégrante de notre vie quotidienne. Mais actuellement, l'un des enjeux majeurs de ce domaine est de réussir à expliquer, de manière claire et concise, comment une intelligence artificielle parvient à produire un résultat correct, et sur quels éléments elle s'appuie pour effectuer ses choix.

C'est dans le but d'en apprendre plus sur cette thématique, sur l'intelligence artificielle et sur le métier de chercheur que j'ai réalisé mon stage au sein du **Centre de Recherche en Informatique de Lens** (CRIL). Entourée de chercheurs, d'enseignants-chercheurs et de doctorants passionnés par l'intelligence artificielle, j'ai pu, avec leur aide, concevoir un modèle mathématique capable d'expliquer un modèle d'apprentissage automatique. Par la suite, j'ai également contribué à l'amélioration du framework développé par M. Lagniez et son équipe pour l'inclure dans la bibliothèque Python **PyXAI**.

Dans un premier temps, je présenterai le contexte de mon stage : comment je l'ai obtenu, la structure d'accueil et le sujet qui m'a été proposé. Dans un deuxième temps, je détaillerai le travail que j'ai réalisé, les choix de conception effectués et les problèmes rencontrés. Pour finir, j'expliquerai le déroulement de mon stage, c'est-à-dire l'intégration dans la structure, ce que m'a apporté le CRIL, ce que ma formation m'a apporté pour le stage, et ce que j'ai pu apporter à l'entreprise.

2 Contexte du stage

2.1 Comment j'ai obtenu le stage

Après de nombreuses candidatures et plusieurs refus, principalement dus au fait que je ne possédais pas encore les compétences nécessaires pour effectuer un stage en intelligence artificielle dans une entreprise, M. Lagniez a proposé à l'ensemble de la promotion un stage axé sur le machine learning.

Très motivée à l'idée d'en apprendre davantage sur ce sujet, j'ai échangé avec M. Lagniez afin de lui faire part de mon intérêt. C'est ainsi que j'ai pu intégrer le Centre de Recherche en Informatique de Lens (CRIL) pour y effectuer mon stage.

2.2 La structure d'accueil

J'ai effectué mon stage au sein du Centre de Recherche en Informatique de Lens (CRIL). Il s'agit d'un laboratoire de **l'Université d'Artois** et du **Centre National de la Recherche Scientifique** (CNRS), dont la majeure partie des recherches porte sur l'intelligence artificielle, et plus particulièrement sur les techniques de raisonnement automatique, d'explicabilité des modèles et de résolution de problèmes complexes.

Le CRIL regroupe près de 70 membres : chercheurs, enseignants-chercheurs, doctorants, ainsi que du personnel administratif et technique.

2.3 Le sujet proposé

Le sujet de mon stage était de concevoir un modèle mathématique capable d'expliquer les prédictions d'un modèle d'apprentissage automatique, afin de mieux comprendre ses décisions. Ce travail s'inscrivait dans une démarche plus large menée par l'équipe du CRIL, qui développe des outils pour améliorer l'interprétabilité et la transparence des systèmes d'IA.

J'ai également contribué à l'amélioration d'un framework existant pour l'intégrer à la bibliothèque PyXAI, développée par mon tuteur et plusieurs autres chercheurs, destinée à faciliter l'analyse et l'explication des modèles.

3 Travail effectué

3.1 Présentation détaillé du projet auquel vous avez participé et mise en perspective

Le projet sur lequel j'ai travaillé s'inscrit dans le domaine de **l'eXplainable Artificial Intelligence** (XAI), c'est-à-dire l'explicabilité des modèles d'intelligence artificielle. Plus précisément, il vise à fournir des explications sur les décisions prises par un modèle de machine learning, à l'aide de **la programmation par contrainte** (CP). Ces explications sont agnostiques, cela signifie que les explications générées ne dépendent pas de la nature du modèle utilisé (réseau de neurones, arbre de décision, etc.), mais seulement de son comportement.

L'idée centrale est la suivante : étant donné une instance d'un jeu de données et la prédiction effectuée par un modèle, peut-on identifier quelles sont **les caractéristiques essentielles** (features) qui ont conduit à cette prédiction ? Et surtout, si l'on sélectionne uniquement ces features essentielles et que l'on prend d'autres instances qui partagent exactement **les mêmes valeurs sur ces features**, obtient-on **la même classification** ? Si oui, cela indiquerait que ces features expliquent effectivement bien la décision.

Pour mieux illustrer cela, prenons un exemple simple avec le célèbre jeu de données des **Iris**. Ce jeu de données comprend 150 échantillons, répartis équitablement entre trois espèces d'iris : *Setosa*, *Versicolor* et *Virginica*. Pour chaque échantillon, quatre caractéristiques ont été mesurées : la longueur et la largeur des sépales, ainsi que la longueur et la largeur des pétales. Ce dataset a été introduit par Ronald Fisher en 1936. Supposons maintenant que l'explication produite soit : "Si la longueur des sépales est supérieure à 6 et la longueur des pétales est supérieure à 5, alors la fleur est classée comme étant de type Virginica." On cherche ensuite à vérifier cette règle sur d'autres exemples similaires, et à calculer un **taux d'erreur** associé. Si ce taux est en dessous d'un certain seuil, alors l'explication est considérée comme valide.

Dans ce cadre, j'ai été chargé de développer différentes modélisations mathématiques du problème. L'objectif principal était d'identifier automatiquement les colonnes (ou features) qui permettent de justifier au mieux une prédiction donnée, en sélectionnant les valeurs discriminantes. La figure 6.1

illustre les différentes étapes du processus de génération d'explication. J'ai, par la suite, codé ces modélisations dans un environnement Python, amélioré et réorganisé le code existant, en le rendant plus modulaire et plus facilement maintenable, ajouté plusieurs nouvelles fonctionnalités, que je détaillerai plus loin dans ce rapport, et préparé le code à son intégration future dans la bibliothèque PyXAI. Cette expérience m'a permis de travailler à la croisée entre modélisation mathématique, programmation et explicabilité, dans une démarche rigoureuse de recherche appliquée. Elle m'a également permis de contribuer concrètement à un outil open-source ayant vocation à être utilisé dans la communauté XAI.

3.2 Choix de conceptions

Dans le cadre du développement du framework, plusieurs décisions de conception ont été prises afin d'assurer sa modularité, sa maintenabilité et son extensibilité. Cette section présente les choix structurants que j'ai opérés au cours de ce travail : la gestion de la récupération des données, l'usage de patrons de conception comme Factory et Null Object, la refonte de la hiérarchie des explainers, l'intégration souple des solveurs, ainsi que la refonte de l'interface utilisateur.

Gestion des données sans classe abstraite intermédiaire

Initialement, j'envisageais d'introduire une classe abstraite **DataLoader** pour imposer une interface commune aux classes **DataLoaderML** et **DataLoaderFile**, à travers la méthode *getData()*. Toutefois, après une réflexion sur les particularités du langage Python, notamment l'absence d'interfaces strictes comme en Java, j'ai estimé qu'une telle abstraction était inutilement complexe. J'ai donc supprimé cette couche intermédiaire pour ne conserver que les classes concrètes, qui assurent chacune la récupération des données selon leur propre logique. Cette approche allège la hiérarchie, évite la duplication inutile et offre davantage de souplesse. Par ailleurs, la classe **DataLoaderFactory** se charge d'instancier dynamiquement la bonne classe de chargement (**DataLoaderML** ou **DataLoaderFile**) en fonction des paramètres fournis, jouant ainsi un rôle équivalent à celui qu'aurait eu une classe abstraite **DataLoader** dans une architecture plus rigide.

Utilisation du patron de conception Factory

Dans une logique de simplification et d'ouverture à l'extension, j'ai largement appliqué **le patron de conception Factory** à différents composants du projet. Ce choix permet de déléguer l'instanciation des objets à des classes dédiées, selon les paramètres fournis en entrée. Cette démarche s'inscrit pleinement dans **le principe Open/Closed du SOLID**, favorisant l'ajout de nouveaux comportements sans modifier le code existant. Voici quelques exemples concrets :

- **DataLoaderFactory** : sélectionne dynamiquement le bon chargeur de données (fichier ou identifiant).
- **ExperimentFactory** : centralise la création et la configuration des expériences.
- **SamplerFactory** : facilite l'introduction de nouvelles stratégies d'échantillonnage.
- **ExplainerFactory** : instancie dynamiquement les différents types d'explainers

L'utilisation de ces factories améliore considérablement la modularité du code, tout en facilitant sa maintenance et son évolution.

Refactorisation et hiérarchisation des explainers

Une autre amélioration majeure concerne la restructuration des classes d'explication. À l'origine, chaque explainer (**SatExplainer**, **CplexExplainer**, etc.) était conçu indépendamment, avec des redondances importantes dans leur structure et leurs méthodes (*___str___*, *toDataFrame*, *test*, etc.). Pour remédier à cette situation, j'ai introduit une classe mère abstraite **Explainer**, regroupant les comportements communs à tous les explainers. J'ai ensuite distingué deux sous-classes intermédiaires :

- **SymbolicExplainer** : pour les approches basées sur des modèles formels (SAT, CPLEX, PySAT, Gurobi, etc.).
- **LocalExplainer** : pour les méthodes d'explicabilité issues de bibliothèques Python (SHAP, Anchor, etc.).

Ce découpage clarifie la hiérarchie, réduit la duplication de code et renforce la cohérence de l'architecture.

Intégration souple des solveurs via le paramètre solvers

En analysant la classe **Experiment**, j'ai identifié une limitation dans la manière dont les solveurs étaient activés : deux booléens (usingShap,

usingAnchor) contrôlaient l'exécution de **Shap** et **Anchor**, rendant l'ajout de nouveaux solveurs difficile et le code verbeux. Pour résoudre ce problème, j'ai introduit un paramètre solvers, passé sous forme de liste de chaînes de caractères, par exemple :

```
solvers = ["sat", "cplex", "pySat", "anchor", "shap", "gurobi"]
```

Cette approche permet à l'utilisateur de choisir dynamiquement les solveurs à utiliser, dans l'ordre de son choix. Les solveurs mal orthographiés sont simplement ignorés, sans générer d'erreur. Ce mécanisme unifie la logique de sélection, facilite l'extension du système, et rend le code plus lisible et maintenable.

Refonte de l'interface utilisateur avec argparse

Afin de centraliser et de rendre plus flexible la configuration des expériences, j'ai également restructuré le fichier **main.py**. La version initiale se limitait à trois paramètres obligatoires en ligne de commande, ce qui ne répondait plus aux besoins croissants du framework. J'ai donc introduit la bibliothèque *argparse*, qui permet de définir des options souples, avec des valeurs par défaut. Cette évolution a rendu le programme :

- plus ergonomique pour les utilisateurs,
- plus clair à utiliser,
- et surtout, plus adaptable à l'évolution des fonctionnalités (ajout de solveurs, choix du fichier de données, nombre d'expériences, etc.).

Implémentation du patron Null Object pour le rééquilibrage

Enfin, dans un souci de lisibilité et de modularité, j'ai implémenté une version adaptée du **patron Null Object** pour la gestion du rééquilibrage des données. J'ai ainsi créé la classe **NoneSampling**, héritant de **DataBalancer**, dont la méthode *balanced()* retourne simplement les données d'origine, sans les modifier. Cette classe remplace avantageusement une logique conditionnelle du type if **use_balancing**, en fournissant une implémentation neutre et explicite. Elle présente plusieurs avantages :

- Élimine les conditions spécifiques et les if redondants.
- Uniformise l'appel à la méthode *balanced()*, quel que soit le contexte.
- S'intègre naturellement dans la **DataBalancedFactory**, qui choisit dynamiquement la stratégie appropriée.

3.3 Travail effectué

Dans un premier temps, afin de me familiariser avec la modélisation mathématique, j'ai étudié le problème classique du coloriage de graphe, que j'ai formalisé sous trois approches différentes (CNF, CSP, MIP). Cette étape préliminaire m'a permis d'appréhender les outils et formalismes que j'allais mobiliser par la suite. Dans un second temps, j'ai pris en main le framework développé par M. Lagniez. Après avoir compris son fonctionnement, je l'ai enrichi en profondeur : refactorisation de l'architecture, ajout de nouvelles fonctionnalités, amélioration de la modularité, et intégration de nouveaux explainers. Par la suite, j'ai introduit une nouvelle méthode d'explication s'appuyant sur le solveur Gurobi, en reformulant le critère d'optimisation sous la forme d'un ratio $\frac{\text{neg}}{\text{neg + pos}}$ plus robuste que les approches précédentes. Enfin, j'ai réalisé une campagne d'expérimentations à grande échelle sur le cluster du CRIL, ce qui m'a permis d'évaluer la performance et la scalabilité du framework dans des conditions proches d'un usage réel.

Modélisation du problème de graphe sous trois formes mathématiques

Avant de m'attaquer pleinement au cœur de mon sujet de stage, j'ai dû me familiariser avec la notion de modélisation mathématique sur un problème bien connu qui est celui de **colorier un graphe**. C'est-à-dire colorier les nœuds du graphe de telle sorte que les nœuds reliés entre eux par des arcs ne soient pas coloriés de la même couleur. J'ai modélisé ce problème de trois façons différentes :

- sous la forme normale conjonctive. Une **forme normale conjonctive** (CNF de l'anglais *Conjunctive Normal Form*) est une formule de calcul propositionnel sous la forme d'une conjonction de clauses. Une **clause** est une disjonction de littéraux. Un **littéral** est une variable propositionnelle de la forme v_j ou $\neg v_j$.

Exemple : Soit l'ensemble des variables $\{v_1, v_2, v_3\}$ et la formule $f = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_3)$. Les clauses $(v_1 \vee v_2)$ et $(\neg v_1 \vee v_3)$ sont des clauses de deux littéraux, et f est sous forme normale conjonctive.

Dans le cas du problème de coloriage de graphe, j'ai créé une variable pour chaque nœud associée à une couleur. C'est-à-dire que si x_1 représente le premier nœud et que le nombre de couleurs autorisé pour résoudre ce problème est de trois, alors j'ai créé des variables nommées x_{11}, x_{12}, x_{13}, qui représentent le nœud 1 avec une couleur différente.

J'ai ensuite ajouté la contrainte qu'un nœud ne peut avoir qu'une couleur, ce qui se modélise comme suit : si on reprend le cas du nœud 1 avec trois couleurs possibles :

$$(\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{12} \vee \neg x_{13}) \wedge (\neg x_{11} \vee \neg x_{13})$$

Cette contrainte signifie que le noeud 1 peut prendre une couleur ou zéro, on rajoute donc une contrainte pour dire que le noeud doit prendre au moins une couleur :

$$x_{11} \vee x_{12} \vee x_{13}$$

Enfin, il faut ajouter la règle qui fait que deux nœuds reliés par un arc ne peuvent pas avoir la même couleur. Cela se modélise comme suit si x_1 est en relation avec x_2 :

$$\neg x_{11} \vee \neg x_{21}$$

Et on répète cela pour toutes les couleurs que peuvent prendre ces deux nœuds.

- sous la forme de contrainte en utilisant le formalisme du **Problème de Satisfaction de Contraintes** (CSP de l'anglais *Constraint Satisfaction Problem*). On définit un problème de satisfaction de contrainte par un triplet (X, D, C) où :

— $X = \{x_1, x_2, \ldots, x_n\}$ est l'ensemble des variables du problème,

— $D = \{D_1, D_2, \ldots, D_n\}$ qui associe à chaque variable x_i son domaine D_j, c'est-à-dire l'ensemble des valeurs que peut prendre x_i,

— $C = \{c_1, c_2, \ldots, c_n\}$ est l'ensemble des contraintes. Ces contraintes s'appliquent sur les variables du problème et permettent de restreindre les valeurs pouvant être prises par ses variables.

Exemple : Si nous prenons l'exemple du sudoku :

— X est l'ensemble des cases x_{ij},

— D est soit la valeur que contient déjà la case ou soit $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

— C est l'ensemble des contraintes qui définissent les règles du sudoku :

  — $x_{ij} \neq x_{ik}$, pour tout $k \neq j$,

  — $x_{ij} \neq x_{kj}$, pour tout $k \neq i$,

  — $x_{i,j} \neq x_{k,l}$, pour tout $(k, l) \neq (i, j)$ dans le même carré 3×3.

Dans le cas de notre problème de coloriage de graphe, on représente ce problème comme suit :

— X est l'ensemble des nœuds du graphe,

— D est l'ensemble des couleurs que peut prendre un nœud,

— C est l'ensemble des contraintes tel que $x_i \neq x_j$, pour tout x_i qui est en relation avec x_j.

- sous la forme d'un problème de **programmation linéaire en nombres mixtes** (MIP en anglais pour *Mixed Integer Programming*). La programmation linéaire en nombres mixtes est une extension de la Programmation Linéaire (PL) où certaines variables sont entières, tandis que d'autres peuvent être continues. Un problème MIP est un problème d'optimisation où l'on essaie de maximiser ou de minimiser une fonction objective avec certaines contraintes.

Exemple : On définit une fonction objective à maximiser, par exemple $5x_0 + 3x_1 + 2x_2$. Puis, on définit des contraintes, par exemple $x_1 + x_2 \geq 3$ et $2x_0 + x_1 + x_2 \geq 10$.

Dans le cas de notre problème de coloriage de graphes, la façon de modéliser ce problème se rapproche de la manière de le faire en forme normale conjonctive. Il faut interpréter les plus comme des "ou" et chaque nœud est représenté par des variables en fonction de la couleur prise. Ces variables sont à 1 si le nœud est de cette couleur-là et à 0 si le nœud ne prend pas cette couleur. On rajoute la contrainte de couleur qui se modélise comme suit :

$$x_{11} + x_{12} + x_{13} = 1$$

Ce qui revient à dire qu'une seule des variables x_{11}, x_{12}, x_{13} doit être à 1.

Pour la contrainte que deux nœuds reliés ne doivent pas avoir la même couleur, il faut la modéliser comme suit :

$$x_{11} + x_{21} \leq 1$$

Si x_1 et x_2 sont reliés par un arc. Cela signifie qu'un seul des deux nœuds, au maximum, peut prendre la première couleur.

Refactorisation et amélioration du framework d'explicabilité

Après avoir modélisé le problème du coloriage de graphe sous trois formes différentes, j'ai pu commencer à m'approprier le framework sur lequel avait travaillé M. Lagniez. Cette première étape m'a permis de mieux cerner la logique générale du projet et d'identifier les points d'entrée du code.

Dans un premier temps, j'ai enrichi le framework existant en ajoutant deux nouveaux explainers : **ExplainerCplex** et **ExplainerPySat**. Chacun modélise le problème de l'explication à l'aide de formalismes mathématiques différents, permettant d'élargir les approches disponibles au sein du framework. Pour améliorer la clarté et l'évolutivité du code, j'ai ensuite entrepris une restructuration complète de l'architecture logicielle. J'ai commencé par créer deux classes distinctes pour la gestion des données :

- **DataLoaderML**, qui permet de charger des jeux de données depuis la plateforme OpenML,
- et **DataLoaderFile**, dédiée à la lecture de fichiers locaux aux formats CSV, JSON ou Parquet.

Afin de faciliter l'instanciation dynamique des objets tout en respectant les bonnes pratiques de conception logicielle, j'ai ensuite introduit **le patron de conception Factory** à plusieurs niveaux du projet. Concrètement, cela s'est traduit par :

- **DataLoaderFactory** pour déléguer le choix de la source de données,
- **ExperimentFactory** pour la configuration des expériences,
- **SamplerFactory** pour la création flexible de méthodes d'échantillonnage.

Ces améliorations ont jeté les bases d'un code plus modulaire, plus maintenable et ouvert à l'extension.

J'ai ensuite entamé un travail de refactorisation plus global, visant à éliminer les redondances et à mieux structurer le code selon différents niveaux d'abstraction. Un bon exemple est l'organisation mise en place autour des explainers, que j'ai réorganisés comme suit :

- **Explainer** : classe abstraite définissant l'interface commune à tous les explainers.
- **LocalExplainer** : héritée de Explainer, cette classe regroupe les explainers utilisant des bibliothèques d'explicabilité connues comme SHAP ou Anchor. Elle centralise des méthodes communes comme *getExplanation*.
- **SymbolicExplainer** : également héritée de Explainer, elle regroupe les explainers reposant sur la résolution de contraintes (SAT, CPLEX, PySAT, etc.), et permet de partager certaines logiques spécifiques à ces approches.

Cette hiérarchie permet non seulement de réduire considérablement les duplications, mais elle facilite également l'ajout de nouveaux explainers dans le futur, tout en garantissant une architecture cohérente.

Une fois les mécanismes de génération d'explications bien établis, il restait une étape cruciale : vérifier la pertinence de ces explications. J'ai donc intégré une méthode *check()* directement dans chaque classe explainer (comme SatExplainer, CplexExplainer, etc.). Cette méthode repose sur un principe simple : une bonne explication doit permettre de distinguer correctement les instances qui partagent les mêmes caractéristiques que l'instance expliquée sur les colonnes sélectionnées. Concrètement, je parcours l'ensemble du dataset pour compter **le nombre de contre-exemples** (neg), c'est-à-dire les instances similaires mais classées différemment. Une fois la solution optimale trouvée par le solveur (en minimisant neg), je compare ce nombre de négatifs au total détecté dans le dataset. Si les deux coïncident, l'explication est considérée comme valide. Cette méthode permet donc d'avoir une vérification locale, cohérente et intégrée directement à chaque explainer, sans avoir à passer par un module externe.

Pour enrichir les fonctionnalités du framework, deux évolutions majeures ont été introduites :

Le rééquilibrage des données : une classe **DataBalancer** a été développée afin de rééquilibrer un jeu de données selon différentes stratégies (*undersampling*, *oversampling*). Cette fonctionnalité permet de corriger les déséquilibres de classes fréquemment observés dans les jeux de données réels.

De nouvelles méthodes de sampling : le mécanisme d'échantillonnage initial reposait sur un tirage aléatoire classique. Deux nouvelles méthodes ont été ajoutées :

- *L'importance sampling*, qui attribue des poids aux différentes classes afin d'orienter l'échantillonnage de manière plus ciblée.
- *Le stratified sampling*, qui sélectionne des instances proches de celle à expliquer, en tenant compte de la structure locale du dataset.

Enfin, dans un souci de qualité logicielle, j'ai pris soin d'améliorer plusieurs aspects pratiques du projet :

- J'ai documenté l'intégralité du projet avec des balises **Doxygen** pour pouvoir générer le code en **HTML** et en **LaTeX** automatiquement.
- Un fichier **requirements.txt** a été généré via pip freeze pour référencer précisément toutes les dépendances du projet. Cela permet à

tout utilisateur de reproduire l'environnement d'exécution en une seule commande (*pip install -r requirements.txt*).

- Enfin, le fichier **README.md** a été progressivement enrichi afin de documenter le fonctionnement du framework, expliquer comment l'utiliser, et décrire les différentes options disponibles. Cela constitue une porte d'entrée claire pour tout nouveau contributeur.

Nouvelle approche d'explication basée sur un ratio

Une autre amélioration importante a concerné le cœur même de la méthode d'explication. Jusqu'ici, toutes les approches étaient basées sur trois contraintes :

1. *Une contrainte d'exclusion basée sur la classe* : si une instance appartient à une classe différente de celle à expliquer mais possède les mêmes caractéristiques (sur les colonnes de l'explication), cela remet en cause la validité de l'explication.
2. *Une contrainte de taille* : l'explication ne doit pas dépasser un seuil fixé par l'utilisateur.
3. *Une contrainte d'optimisation* : on cherchait à minimiser le nombre de négatifs, c'est-à-dire les instances similaires mal classées.

Dans cette nouvelle version, j'ai remplacé cette dernière contrainte par **une minimisation du ratio** :

$$\text{ratio} = \frac{\text{neg}}{\text{neg} + \text{pos}}$$

où :

— neg désigne le nombre d'instances ayant les mêmes caractéristiques mais étant classées différemment,

— pos désigne celles qui ont la même prédiction que celle expliquée.

Cette approche, plus robuste, permet de prendre en compte à la fois les cas positifs et négatifs. Cependant, comme **Gurobi** ne permet pas de minimiser un ratio directement, j'ai dû modéliser cette contrainte à l'aide de **la programmation quadratique**. En partant de :

$$z = \frac{\text{neg}}{\text{neg} + \text{pos}}$$

on peut écrire :

$$z \cdot (\text{neg} + \text{pos}) \geq \text{neg}$$

Ce qui revient à dire :

$$z \geq \frac{\text{neg}}{\text{neg} + \text{pos}}$$

Minimiser z revient alors à minimiser le ratio initial. Cette formulation est compatible avec les capacités de Gurobi et permet d'approximer le comportement souhaité tout en restant solvable.

Utilisation du cluster de calcul

Pour tester l'intégralité de mon travail et mener une étude statistique sur les performances des différents explainers, j'ai eu accès **au cluster de calcul** du CRIL, un environnement haute performance bien plus puissant que mon ordinateur personnel. Cet accès m'a permis d'exécuter mon programme sur de grands jeux de données, difficilement traitables en local, et ainsi de valider l'efficacité sur des données de grande taille et la robustesse de mon framework. J'ai profité de cette opportunité pour apprendre à :

- Transférer mon programme et ses dépendances sur le cluster (via *scp* et des environnements Python isolés),
- Configurer l'environnement d'exécution
- Et rédiger des scripts de lancement (*batch*) adaptés à l'architecture du cluster (fichiers .slurm), permettant l'exécution automatique et reproductible d'expériences.

Cette étape a été essentielle pour tester le système dans un contexte réel d'utilisation et m'initier à l'usage d'infrastructures de calcul avancées, souvent utilisées en recherche ou en entreprise.

Une partie des résultats expérimentaux, incluant notamment les taux d'erreur obtenus par chaque explainer sur différents jeux de données, est présentée en Annexe 6.

3.4 Problèmes rencontrés, solutions éventuellement apportées

Au cours de mon projet, j'ai été confronté à plusieurs difficultés techniques et conceptuelles. Voici les principales, accompagnées des solutions que j'ai pu mettre en œuvre.

Modélisation du problème du coloriage de graphe en MIP

La première difficulté notable fut la modélisation du problème de coloriage de graphe sous forme de **programme linéaire en nombres entiers mixtes** (MIP). L'objectif était de garantir que deux sommets reliés par une arête ne puissent recevoir la même couleur, ce qui devait être exprimé de manière linéaire.

Initialement, j'ai envisagé d'imposer la contrainte $|x_1 - x_2| \geq 1$, mais l'utilisation de la valeur absolue n'est pas autorisée en programmation linéaire standard. J'ai ensuite tenté des reformulations comme $x_1 - x_2 + x_2 - x_1 \geq 1$, ou encore $x_1 - x_2 \geq 1$ ou $x_2 - x_1 \geq 1$, mais ces expressions non linéaires ou contenant un opérateur « ou » ne sont pas compatibles avec les contraintes MIP.

La solution a été de reformuler le problème en introduisant une variable binaire pour chaque couple (sommet, couleur), de sorte qu'un sommet ne puisse prendre qu'une couleur et que deux sommets adjacents ne puissent partager la même. Ce raisonnement est similaire à celui utilisé dans la formulation **CNF**.

Programmation orientée objet en Python

Une autre difficulté majeure a été l'adaptation à la programmation orientée objet en Python, qui diffère sensiblement de celle de **Java**. Le projet reposait sur de nombreux fichiers interconnectés, faisant appel à des bibliothèques que je connaissais peu (*pandas*, *numpy*, *sklearn*), et dont le code était dense et peu documenté.

Dans un premier temps, j'ai donc pris le temps de comprendre l'architecture globale et les rôles de chaque composant. Cela m'a permis d'acquérir de solides bases en Python orienté objet et de mieux cerner les attentes du projet. Un point bloquant a été l'absence d'interfaces strictes comme en Java. Je souhaitais initialement créer **une interface DataLoader** implémentée par **DataLoaderFile** et **DataLoaderML**. En Python, j'ai envisagé l'utilisation de la bibliothèque *abc* pour créer une classe abstraite, mais l'absence de vérification stricte à l'exécution m'a amené à revoir cette approche. Finalement, j'ai simplifié la conception en laissant la **DataLoaderFactory** instancier directement l'objet approprié en fonction du contexte.

Un autre obstacle a été **l'impossibilité de surcharger les méthodes** (fonctionnalité disponible en Java). Pour contourner ce manque, j'ai opté pour des paramètres optionnels dans les constructeurs ou fonctions, ce qui complexifie légèrement la lecture du code mais reste une solution viable. Au

fil du temps, j'ai su m'adapter aux spécificités de Python tout en conservant une approche propre et maintenable.

Transfert des données sur le cluster

Le transfert du projet sur le cluster du CRIL a soulevé plusieurs défis, notamment liés à la gestion des environnements Python. Mon projet étant initialement développé sous **Python 3.10**, j'ai tenté de reproduire un environnement compatible (Python 3.9) via *pyenv* pour l'importer sur le cluster. Cependant, cette solution a échoué car de nombreuses bibliothèques se basaient sur des chemins spécifiques à ma machine locale.

Pour surmonter ce problème, j'ai utilisé *pip download* afin de récupérer l'ensemble des dépendances dans un dossier, puis les installer sur le cluster avec la commande *pip install –no-index –find-links*.

Enfin, j'ai rencontré des difficultés lors de l'installation de la bibliothèque *anchor_exp*, qui repose sur certains outils de compilation (comme *Cython*) absents du fichier **requirements.txt**. L'installation manuelle des dépendances manquantes a permis de rétablir le fonctionnement de l'outil et de finaliser l'exécution de mes expériences.

4 Bilan personnel

4.1 Intégration dans la structure

Dès le premier jour, j'ai fait la rencontre de plusieurs doctorants et personnels de l'université d'Artois. Je me suis assez facilement intégrée dans l'équipe de doctorants qui ont gentiment partagé leur bureau avec moi. J'ai pu demander de l'aide et être toujours accueilli avec le sourire le matin. Même si parler avec les doctorants qui ne parlaient pas français en anglais m'a posé quelque difficulté dans mon intégration.

J'ai été autonome très rapidement, n'étant pas dans le même bureau que mon tuteur et mon tuteur étant parti trois semaines à l'étranger. J'ai su me mettre au travail en autonomie dès le départ et apprendre à chercher par moi-même. Bien que mon tuteur soit parti à l'étranger, il a toujours été là quand je sollicitais son aide.

J'ai donc été très bien accueilli, j'ai pu parler avec de nombreux doctorants et personnels de l'université.

4.2 Apports personnels

Ce que la structure d'accueil m'a apporté

Au cours de mon stage au CRIL, j'ai eu l'opportunité d'élargir mes compétences tant sur le plan technique que personnel.

- **Utilisation de nouvelles bibliothèques Python** : j'ai pu approfondir ma connaissance de diverses bibliothèques Python (numpy, pandas, sklearn) centrées sur l'IA et le traitement de données.
- **Faire de la programmation orientée objet sur un gros projet en python** : cela m'a permis de voir que dans un langage tel que Python qui n'est pas purement objet comme Java, certaines des règles tel que les interfaces, la surcharge de méthode etc sont absente. Cela m'a obligé à m'adapter et à penser différemment, tout en essayant de coller le plus possible aux principes de cette méthode de programmation.
- **Compréhension de l'explication des résultats d'une IA** : j'ai acquis une meilleure compréhension des mécanismes permettant d'expliquer les décisions prises par les modèles d'intelligence artificielle, une compétence essentielle dans la transparence des algorithmes.

- **Modélisation de problèmes sous différentes formes mathématiques** : j'ai appris à modéliser des problèmes complexes sous différentes formes mathématiques, telles que la forme normale conjonctive, les problèmes linéaires en nombres mixtes, ainsi que le formalisme du problème de satisfaction de contraintes (CSP).
- **Utilisation d'un cluster de calcul haute performance** : j'ai découvert comment exploiter un environnement de calcul puissant pour exécuter des expériences à grande échelle. Cela m'a appris à transférer mon code et ses dépendances, à adapter mes scripts pour cet environnement spécifique, et à gérer des contraintes liées aux versions de Python ou aux bibliothèques utilisées.
- **Développement de soft skills** : mon expérience au CRIL m'a également permis de renforcer mes compétences interpersonnelles, notamment l'adaptabilité, la persévérance dans la résolution de problèmes complexes, ainsi que l'autonomie dans l'exécution de tâches de manière indépendante.

Ce que ma formation m'a apporté

Ma formation en informatique m'a donné de solides bases théoriques et pratiques que j'ai pu mobiliser de manière concrète tout au long de mon stage.

En particulier, j'ai beaucoup utilisé les principes de la **programmation orientée objet** (POO), notamment l'héritage, l'encapsulation et la délégation. Ces concepts m'ont permis de structurer le code de manière modulaire et réutilisable, facilitant ainsi sa lisibilité et sa maintenabilité.

J'ai également eu l'occasion de mettre en œuvre plusieurs patrons de conception vu en cours de **Conception Orientée Objet** : comme le design pattern "Factory". Ce patron m'a permis de rendre la création d'objets plus flexible, en centralisant la logique d'instanciation dans une structure dédiée.

D'un point de vue plus général, j'ai aussi mis en application les bonnes pratiques de factorisation du code : réduction des redondances, regroupement logique des fonctionnalités, séparation des responsabilités entre classes et fonctions.

Ce que j'ai pu apporté à l'entreprise

Au cours de mon stage, j'ai contribué de manière significative au projet qui m'a été confié :

- J'ai tout d'abord résolu des problèmes liés à l'exécution du code, qui fonctionnait bien sur le cluster du CRIL, mais rencontré des difficultés à s'exécuter sur d'autres machines.
- J'ai contribué à restructurer et à factoriser le code écrit par Mr Lagniez et son équipe pour pouvoir l'intégrer dans la bibliothèque PyXAI.
- J'ai amélioré le framework en n'y ajoutant plusieurs nouvelles fonctionnalités. Par exemple, le fait de pouvoir vérifier l'explication avec la méthode check, mais aussi par exemple le fait d'avoir ajouté trois nouveaux explainers (PySat, Cplex et Gurobi).
- J'ai documenté l'entièreté du code avec des balises Doxygen pour que les utilisateurs puissent générer la documentation entière de notre programme. De plus, j'ai maintenu le Read.me à jour.

5 Conclusion

Ce stage au CRIL a été une expérience enrichissante qui m'a permis d'appliquer et de développer mes compétences techniques, notamment en Python, en conception orientée objet, et en modélisation mathématique. Travailler sur des projets d'IA et de programmation par contraintes m'a permis de mieux comprendre les défis techniques et les enjeux de la recherche en intelligence artificielle, tout en renforçant mes capacités à résoudre des problèmes complexes.

Les compétences humaines, telles que l'adaptabilité, la persévérance et l'autonomie, ont également été des atouts majeurs dans l'achèvement de mes missions. En contribuant à améliorer la compatibilité du code et à développer une bibliothèque Python, j'ai pu apporter une réelle valeur ajoutée au sein de la structure.

Ce stage m'a permis de mieux cerner mes aspirations professionnelles et m'a conforté dans mon projet de carrière dans le domaine de l'IA. Il a été un pas important vers la consolidation de mes compétences et vers la poursuite de mon parcours dans ce domaine passionnant.

Remerciements

Je tiens à remercier sincèrement Monsieur Lagniez pour m'avoir offert l'opportunité d'approfondir mes connaissances en intelligence artificielle, pour la confiance qu'il m'a accordée, et pour m'avoir plongé dans le monde passionnant de la recherche. Cette expérience a été particulièrement enrichissante, et je n'aurais pas pu rêver mieux. Merci, Monsieur, de m'avoir fait découvrir ce si bel univers qu'est la recherche, d'avoir pris le temps de me le transmettre, et de m'avoir inspiré.

Je tiens à remercier le CRIL pour m'avoir accueilli dans ses locaux et permis d'évoluer dans un environnement de recherche stimulant et enrichissant. Travailler au sein de ce laboratoire a été une opportunité précieuse dans mon parcours académique.

Je remercie également chaleureusement les doctorants pour leur accueil, leur aide précieuse tout au long du stage, et pour m'avoir fait sentir comme un membre à part entière de leur équipe.

6 Annexe

Voici quelques ressources pertinentes consultées pendant le stage.

— Wikipédia – Coloration de graphe
— Wikipédia - Problème de satisfaction de contrainte
— Cours - Problème de satisfaction de contrainte
— Cours - Mixed Integer Programming
— Wikipédia - Forme normale Conjonctive
— Cours - Forme normale conjonctive
— Cours - Installation de Cplex
— Documentation Cplex
— Pour installer Cplex
— Documentation Ortools
— Documentation PySat
— PyXAI
— Documentation PyXAI
— Travaux réalisé par M.Lagniez et ses collègues - Learning Model Agnostic Explanations via Constraint Programming
— Cours de Conception Orientée Objet
— Patron de conception Factory
— Patron de conception Null Object
— Documentation de Shap
— SHAP
— Anchor
— Programmation Quadratique
— Gurobi Documentation

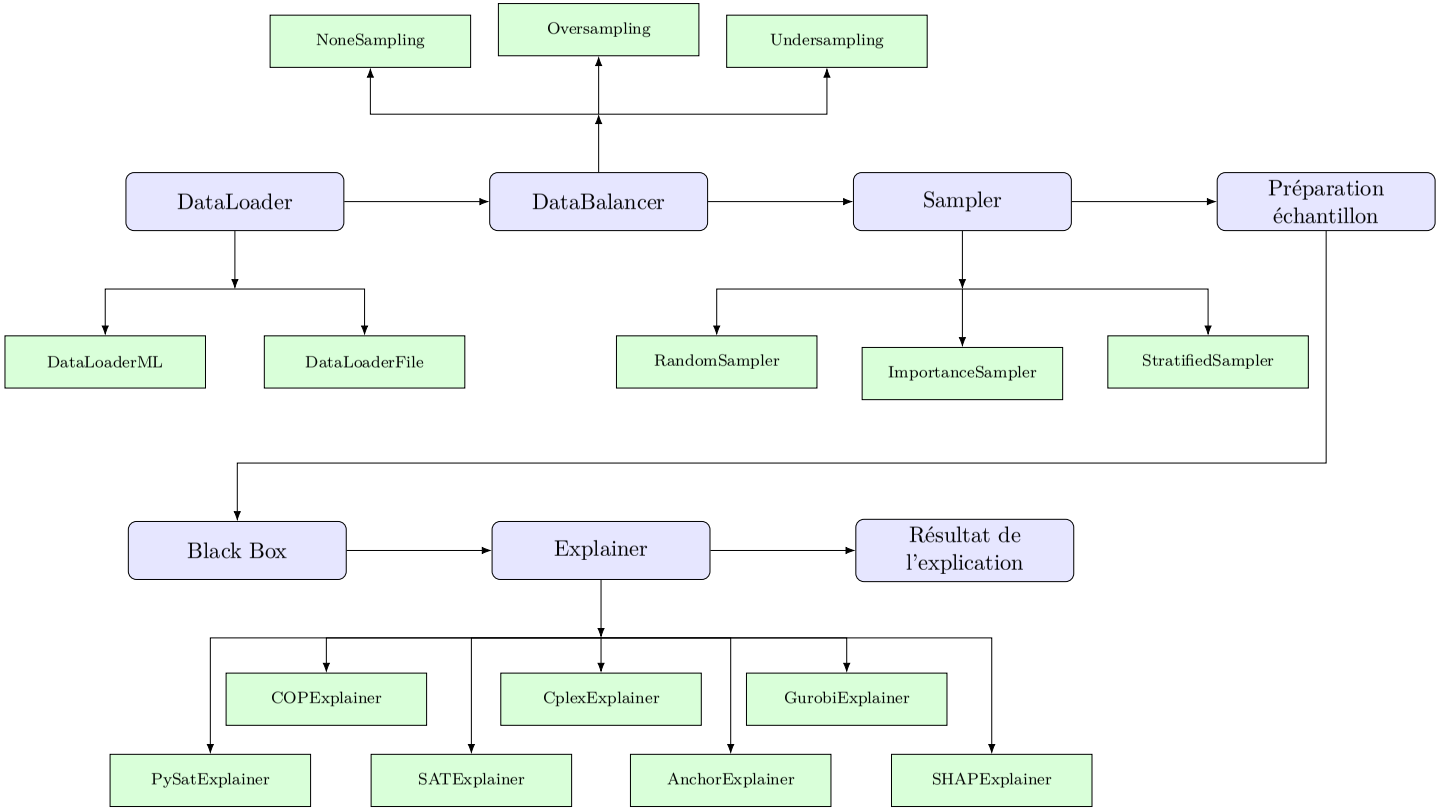

FIGURE 6.1 – Génération d'une explication

Résultats expérimentaux sur le cluster

Les résultats présentés ci-dessous comparent les taux d'erreur obtenus par chaque explainer sur différents jeux de données. Chaque expérience a été répétée **10** fois, avec **1000** échantillons d'entraînement à chaque fois. La taille maximale des explications était fixée à **5**, et un temps limite de **60** secondes était imposé pour chaque explication.

Données		**Taux d'erreur par explainer**					
Nom	**Id**	**SAT**	**PySat**	**Anchor**	**Shap**	**Gurobi**	**COP**
iris	61	0.03(±0.05)	0.2(±0.2)	0.65(±0.13)	0.65(±0.13)	0.02(±0.03)	0.02(±0.06)
wine	187	0.18(±0.17)	0.47(±0.17)	0.51(±0.13)	0.51(±0.13)	0.19(±0.16)	0.18(±0.12)
bank-marketing	1461	0.1(±0.06)	0.13(±0.9)	0.19(±0.008)	0.19(±0.008)	0.06(±0.05)	0.095(±0.07)
heart_disease	43944	0.19(±0.1)	0.5(±0.2)	0.5(±0.2)	0.5(±0.2)	0.17(±0.1)	0.15(±0.1)
vehicule	54	0.35(±0.17)	0.75(±0.06)	0.75(±0.06)	0.75(±0.06)	0.43(±0.16)	0.34(±0.13)
covertype	150	0.45(±0.1)	0.63(±0.1)	0.63(±0.1)	0.63(±0.1)	0.45(±0.1)	0.4(±0.13)
first-order-theorem-proving	1475	0.45(±0.34)	0.49(±0.36)	0.49(±0.36)	0.49(±0.36)	0.48(±0.35)	0.4(±0.34)
satimage	182	0.77(±0.13)	0.89(±0.06)	0.89(±0.06)	0.89(±0.06)	0.79(±0.1)	0.74(±0.12)

FIGURE 2 – Taux d'erreur moyens obtenus par chaque explainer pour différents jeux de données.

Dans la Figure 2, qui présente les taux d'erreur moyens obtenus par chaque explainer sur les différents jeux de données, il ressort que **COP** est l'explainer avec les meilleurs résultats (les taux d'erreur les plus faibles) sur l'ensemble des datasets. **Gurobi**, utilisant la minimisation du ratio, affiche un taux d'erreur particulièrement bas sur le jeu de données *heart_disease*, et il obtient des résultats proches de COP pour des datasets comme *iris* ou *wine*. Toutefois, Gurobi ne parvient pas à obtenir de bonnes performances sur des jeux de données comme *vehicule*.

De son côté, **SAT** réalise de meilleurs résultats que Gurobi dans la majorité des cas, et approche presque les performances de COP. Cependant, pour chaque jeu de données, le taux d'erreur de SAT reste légèrement plus élevé que celui de COP.

Les explainers **Shap**, **Anchor**, et **PySat** présentent des taux d'erreur bien plus élevés. **PySat**, en particulier, est bien moins performant que **SAT** pour trouver des explications correctes et efficaces.

Données		Temps par explainer en seconde					
Nom du dataset	**Id du dataset**	**SAT**	**PySat**	**Anchor**	**Shap**	**Gurobi**	**COP**
iris	61	0.53(±0.06)	0.01(±0.03)	0.00007	0.007	3.81(±0.6)	15.3(±2)
wine	187	9.87(±19.7)	0.07(±0.04)	0.000064	0.0023	15.85(±22)	60
bank-marketing	1461	0.2(±0.01)	0.009(±0.02)	0.00064	0.008(±0.004)	2.7(±0.09)	60
heart_disease	43944	12.7(±15)	0.07(±0.05)	0.00007	0.005(±0.005)	28(±27)	60
vehicule	54	43.5(±27)	0.1(±0.002)	0.000061	0.008	51(±16)	60
covertype	150	20(±26)	0.12(±0.002)	0.000089	0.001	41(±23)	60
first-order-theorem-proving	1475	25(±30)	0.12(±0.007)	0.0008	0.01	26(±27)	60
satimage	182	60	0.12	0.0009	8e-07	60	60

FIGURE 3 – Temps moyen (en secondes) mis par chaque explainer pour produire une explication.

Dans la Figure 3, nous pouvons observer le temps moyen nécessaire pour chaque explainer afin de produire une explication. Il est notable que **COP**, à l'exception du jeu de données *iris*, atteint systématiquement le timeout de 60 secondes, que ce soit sur des jeux de données de grande taille (comme *heart_disease* et *vehicule*) ou plus petits. En revanche, **SAT** et **Gurobi** atteignent ce timeout uniquement pour le dataset *satimage*, ce qui montre que ces explainers sont relativement plus rapides que **COP** sur les autres datasets. Quant à **Anchor** et **Shap**, ils sont extrêmement rapides, nécessitant seulement quelques millisecondes à quelques secondes pour produire une explication, même sur les jeux de données les plus grands.

Dans les diagrammes ci-dessous, on observe que, pour la plupart des explainers, le taux d'erreur diminue généralement à mesure que la valeur de k augmente. Cela indique que l'augmentation de la taille de l'échantillon d'explications a tendance à améliorer la qualité de l'explication, ce qui est particulièrement vrai pour les explainers comme **SAT**, **Gurobi**, et **COP**.

Cependant, pour **Anchor**, on remarque une tendance différente : peu importe la taille de k, le taux d'erreur reste relativement élevé et ne varie que très peu. Cela suggère que Anchor ne profite pas autant de l'augmentation de la taille de k, ce qui pourrait être dû à la nature spécifique de son approche d'explication.

De plus, bien que **COP** ait tendance à obtenir le taux d'erreur le plus bas dans presque tous les cas, il y a des exceptions notables. Par exemple, pour $k=4$ sur le dataset iris, COP ne présente pas le meilleur taux d'erreur, avec **SAT** et **Gurobi** étant plus performants dans ce cas précis. Cela montre que bien que COP est généralement performant, mais il n'est pas

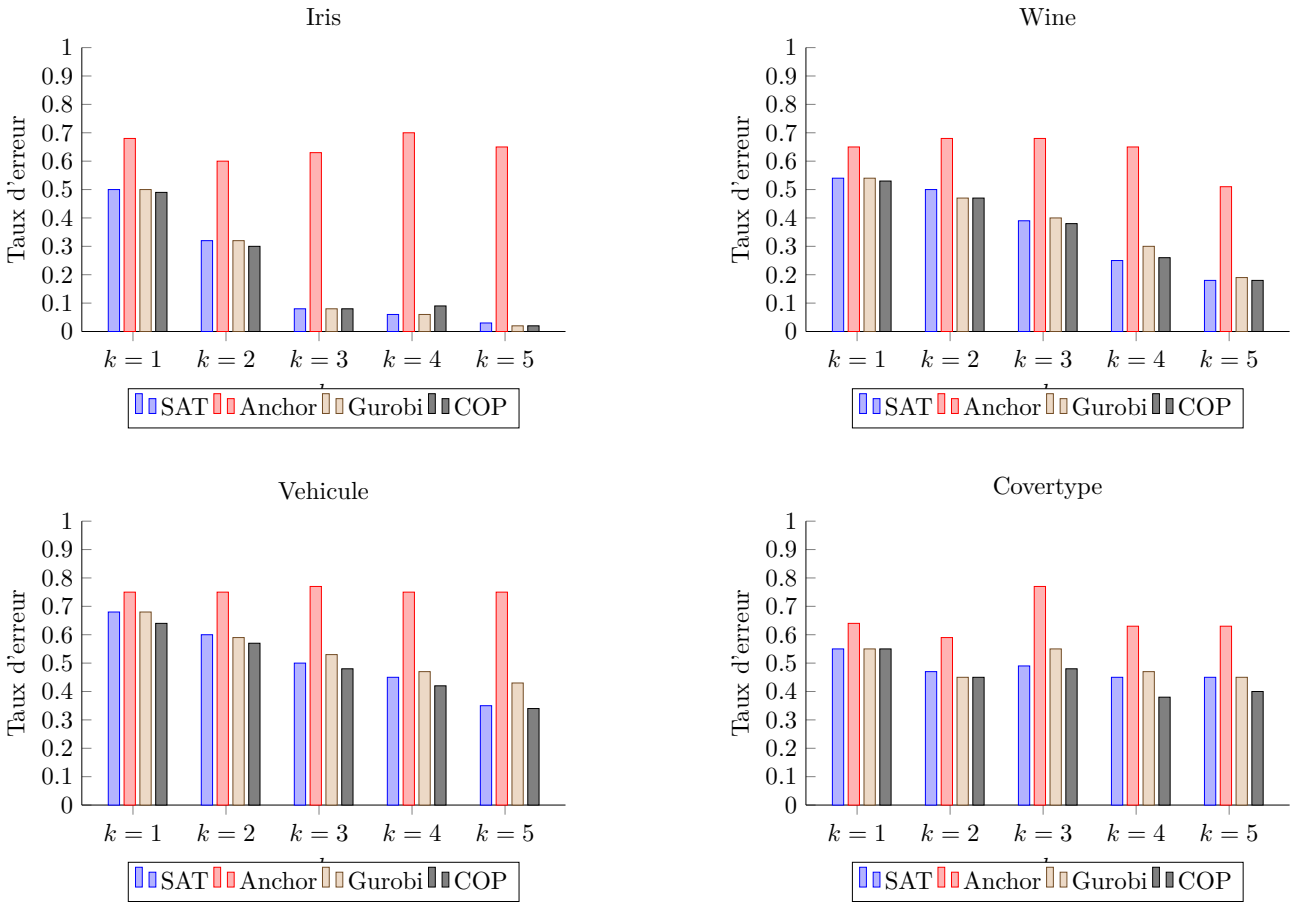

FIGURE 4 – Taux d'erreur en fonction de k pour différents explainers sur 4 jeux de données

systématiquement le meilleur dans toutes les situations.