L'apprentissage automatique dans la vraie vie

Petit guide pour la mise en production

Nastasia Saby

L’apprentissage automatique dans la vraie vie

Petit guide pour la mise en production

Nastasia Saby

Ce livre est en vente à http://leanpub.com/machinelearningenproduction

Cette version a été publiée le 2024-08-29

Ce livre est publié par Leanpub. Leanpub permet aux auteurs et aux éditeurs de bénéficier du processus Lean Publishing. Lean Publishing consiste à publier à l’aide d’outils très simples de nombreuses itérations d’un livre ebook en cours de rédaction, d’obtenir des retours et des commentaires des lecteurs afin d’améliorer le livre.

Table des matières

Introduction

Un modèle de machine learning aussi puissant soit il n'a aucune valeur s'il n'est pas en production.

Le temps où l'apprentissage automatique était uniquement présenté comme quelque chose de magique est dépassé. Ce qu'on veut aujourd'hui, ce sont des modèles qui partent en production et apportent une plus-value. Or cela reste compliqué. *L'article Continuous Delivery for machine learning (intégration continue pour le machine learning)* de Danilo Sato, Arif Wider et Christophe Windheuser explique que beaucoup de projets de data science ne partent pas en production ou s'ils sont mis en production restent difficiles à maintenir.

Le présent écrit se veut un guide pour mettre en production des modèles de machine learning. Il se spécialise sur l'apprentissage supervisé en mode batch pour la partie entraînement. Il ne s'agit pas de parler d'apprentissage par renforcement (« reinforcement learning »). Il ne s'agit pas non plus d'apprentissage non supervisé. Il y aura un petit chapitre qui évoquera ces deux méthodes sans entrer dans les détails. Cet écrit ne s'intéresse pas non plus à l'exploration des données ou la partie modélisation sauf pour spécifier l'intérêt qu'ils ont dans la mise en production. Ainsi, nous supposons que nous avons déjà un modèle dont nous sommes satisfaits, sur lequel une première phase de validation, quelle que soit sa forme, a été mise en place et dont nous voulons rentabiliser l'existence.

Nous aborderons les notions d'automatisation, de déploiement, de monitoring, de tests ou encore de versioning. Celles-ci s'abordent d'une manière particulière quand on fait de la data science. Certaines choses sont tout à fait semblables à ce qui est aujourd'hui réalisé en programmation classique. D'autres choses sont très différentes. Nous verrons aussi comment organiser une équipe autour de ces problématiques et discuterons davantage de techniques de collaboration.

À la fin de ce guide, vous devriez avoir une vision plus complète de ce qu'est une API web qui dessert une prédiction, un pipeline de données, du versionning de data, du monitoring de dérive de modèle (« model drift » en anglais), des containeurs ou encore les méthodes d'organisation d'équipes qu'on peut utiliser pour développer de l'apprentissage automatique. Ces quelques éléments sont là pour vous mettre l'eau à la bouche. Il y a beaucoup à dire quand il s'agit de rendre opérationnel un système de machine learning. Cela ne doit pourtant pas vous effrayer. Chaque projet a ses spécificités et ce guide présente une dimension large de la problématique. De plus, il est possible de mettre en place tous les méthodes et processus qui seront évoqués au fur et à mesure. Certains sont plus critiques que d'autres. Certains sont primordiaux pour tous les projets. D'autres n'ont d'intérêt que dans certains cas.

Ce travail est lié à mes expériences, échanges et réflexions. Anciennement développeur back-end, je suis ingénieur machine learning. Ma spécialité est la mise en production de modèles de machine learning. Je pense avoir encore beaucoup de choses à apprendre dans ce domaine qui est assez récent. Cependant, j'ai aussi eu l'occasion de voir et mettre en place plusieurs projets en production. Cela

m'a permis de traverser tout un champ des possibles. Je voudrais aujourd'hui partager cela avec vous.
Je voudrais que cette lecture soit pour vous une source d'inspiration pour vos projets courants ou à venir. Elle représente cependant ma vision du machine learning en production à un instant donné. J'espère que cette vision sera inspirante, mais ne la souhaite en aucun cas dogmatique. Les contextes peuvent différer.

Remerciements

Je voudrais remercier Fabien de Saint pern, Thomas Bracher, Megan Saby, Mathieu Ecalard, Sylvain Coudert, Laurent Tardif et Jonathan Winandy. Toutes ces personnes m'ont aidé dans la folle aventure de ce guide. Elles m'ont apporté des relectures et soutiens utiles et chaleureux. Sans elles, cet écrit aurait été bien plus brouillon et moins pertinent. Je n'ai pas pu appliquer l'exactitude de tous les conseils judicieux qu'elles m'ont délivrés, mais je garde ceux-ci pour de nouvelles aventures.

Pour qui ?

Ce livre est à l'adresse des data scientistes, ingénieurs machine learning, ingénieurs data et développeurs data amenés à mettre en production des modèles. En fonction des équipes, cela peut être à la charge d'un développeur, d'un data scientiste ou encore d'un ingénieur data. La dénomination n'est pas ce qui importe. Ce guide s'adresse à toute personne susceptible de près ou de loin d'être en charge de la mise en production de modèles. Si vous créez des modèles, mais ne les mettez pas en production, ce guide reste intéressant pour vous aussi. Il vous permettra de mieux comprendre toute la chaîne et les impacts qu'ont vos choix dans la maintenance du projet.
Cet ouvrage s'adresse aussi en second lieu aux managers, product owners ou encore aux business owners. Par l'intermédiaire de cet écrit, ces personnes-là comprendront davantage les grands principes pour pérenniser un modèle.

L'écrit ne donnera pas d'explication sur ce qu'est la data science ou l'intelligence artificielle. Le principe ici est que vous avez déjà une idée de ce dont il s'agit et que votre intérêt actuel est la mise en production de systèmes de machine learning. Une courte définition de certains termes sera donnée. Son but est de faire de la clarification lexicale. Il s'agit par-là d'éviter les confusions, mais non pas d'expliquer ce dont il s'agit réellement. L'intention n'est pas d'en lister toutes les spécificités. Les concepts, méthodes et outils sont toujours évoqués dans le contexte de la mise en production.

Pourquoi ?

J'avais à cœur de discuter du sujet de la mise en production de modèles de machine learning. C'est un thème sur lequel il y a beaucoup à dire et sur lequel peu de choses ont encore été dites. On a longuement parlé des algorithmes de machine learning. Je reconnais l'avoir fait aussi. On a débattu

sur l'apprentissage ensembliste ou profond, sur le traitement du langage naturel ou les meilleures manières d'encoder un texte, mais assez peu encore sur les meilleures manières de déployer des modèles. Cela est sans doute dû au fait que la discipline est jeune et tâtonne encore sur plusieurs points. Cependant, plusieurs d'entre nous commencent à pouvoir rassembler un peu d'expérience qu'il serait bon de partager. Je voulais ainsi sortir des buzzwords tels qu'apprentissage profond ou réseaux de neurones dont on entend souvent parler. J'ai plus d'une fois contribué à l'expansion de ces buzzwords lors d'articles de blog ou de conférence. Je ne dis pas qu'ils ne sont pas nécessaires. Mais ils représentent une partie bien moindre du travail réel d'un data scientiste. En réalité, les algorithmes que nous entendons beaucoup citer représentent généralement 10% du travail d'un data scientiste, le reste est consacré à la préparation de données, validation, monitoring, etc.

On parle donc encore peu de la production en machine learning. Je tiens ainsi à remédier en partie à ce problème avec ce guide. Bienvenue dans le monde réel de l'apprentissage automatique.

Comment ?

Je m'excuse pour le franglais sur lequel vous avez déjà peut-être commencé à vous heurter. Vous le trouverez de manière récurrente tout au long de cette lecture. Nos métiers sont influencés par la culture anglo-saxonne. Certains termes sont ainsi difficiles à traduire. J'ai tenté de choisir les mots les plus appropriés en fonction du contexte.

Je n'ai pas pour but de vous amener à découvrir ou maîtriser tel ou tel outil. Vous ne trouverez pas d'exemple de code. Je n'ai inclus qu'un peu de pseudo code. Je n'évoquerai que peu de librairies ou frameworks. Je ne les citerai qu'à titre d'exemples. L'idée est davantage d'avoir une vue logique de ce que veut dire mettre en production un système de machine learning. Je me concentre sur les éléments auxquels il faut penser ou mettre en place quand on veut pérenniser l'utilisation d'un modèle. J'ai pris ce parti pour trois raisons principales. Premièrement, je veux que cette lecture puisse être cursive, que vous puissiez le faire dans le métro quand il est bondé ou le soir en allant vous coucher. Je ne souhaitais pas réaliser un livre qui ressemble davantage à un tutoriel et pour lequel il faut se munir d'une grande concentration. Ensuite, les outils sont multiples. Dans certains cas, certains se démarquent, mais pas toujours. Ce serait donc dommage de donner la préférence à l'un plutôt qu'à un autre. Enfin, au-delà des outils, je souhaite attirer votre attention sur les concepts qu'il y a derrière et les méthodes d'une manière générale que vous pouvez choisir pour partir en production sereinement. Les outils vieillissent bien plus vite que les concepts qui les font naître.

Garder l'esprit ouvert

En data science, il s'agit de garder l'esprit ouvert. C'est le cas pour bien d'autres domaines, mais ce présent ouvrage s'intéresse à celui-ci. Beaucoup d'outils et de techniques n'existaient pas il y a deux ans. Notre monde évolue vite et cela est aussi dû au fait que nous sommes dans un secteur d'innovation. Cela est stimulant. Mais cela veut aussi dire que tout peut se périmer plus vite qu'on ne le croit. Il ne faut donc pas rester sur des positions dogmatiques, mais continuer à avancer et regarder

autour de soi pour tirer le meilleur de ce qui se fait. Il ne s'agit bien sûr pas de s'engouffrer dans toutes les nouvelles idées qui apparaissent. Il n'y a pas de honte à vouloir tester de nouvelles choses et cela devrait même être encouragé. Pour autant, il ne s'agit pas de vouloir mettre en production tout et n'importe quoi. Cependant, il est normal de mettre des choses nouvelles en production. Nous sommes un secteur d'innovation. La veille que certains employeurs prennent parfois pour de l'amusement de développeurs ou de data scientistes est cruciale. Il ne faut pas s'attendre à avoir une équipe réellement performante quand on ne l'encourage pas et ne l'aide pas à faire de la veille. Si on veut réellement être compétitifs ou juste sereins en production, nous n'avons pas le choix. C'est un monde mouvant loin d'être complètement mature. C'est pourquoi il faut rester attentif à notre champ qui évolue. Cela nous permet d'en tirer le meilleur. Les articles, papiers et livres sur le sujet sont utiles et souvent inspirants, mais en aucune façon ils ne sauraient déceler une vérité unique pour tous pour tout moment donné.

Que signifie mettre un modèle en production ?

Afin de rentrer plus avant dans le sujet, il s'agit de clarifier cela. Que signifie « mettre en production » ? Cela renvoie-t-il au fait de donner aux utilisateurs finaux la possibilité d'utiliser le modèle ? Est-ce que ce concept ne comprendrait pas aussi le fait d'évaluer au fur et à mesure les impacts business du modèle ? Souvent, quand on parle de mise en production, on signifie aussi qu'on veut automatiser toutes les tâches que nous avions réalisées de manière manuelle dans les phases d'exploration et de modélisation. On cherche à passer du prototype à la production.

Mettre en production un modèle, c'est en fait déjà tout cela et d'autres choses encore.
Mettre en production, c'est en effet mettre le système de machine learning à disposition des utilisateurs finaux. Pour ce faire, il faut penser à plusieurs éléments inhérents à la data science. L'apprentissage supervisé a besoin de données pour exister. C'est par là que tout commence. Il s'agit de les penser de manière régulière et pérenne pour assurer une production continuelle.
Ensuite, il y a deux éléments à penser : l'entraînement et l'inférence. L'inférence renvoie au fait d'utiliser le modèle pour délivrer les prédictions. Cela peut se faire de différentes manières que nous approfondirons.
Notons que mettre en production ne signifie pas uniquement servir le modèle. Il s'agit d'avoir confiance en celui-ci et de pouvoir assurer sa pérennité dans le temps. C'est aussi être capable de le maintenir, de le débuguer et de le faire évoluer facilement. C'est pourquoi le versioning, les tests automatisés ou le monitoring sont si importants.

Pour donner une structure de ce guide, vous vous trouverez dans le premier chapitre qui constitue l'introduction.
Le chapitre 2 est consacré à l'automatisation d'un système prédictif. Il y sera présenté les trois éléments clefs que sont la mise à disposition des données, la nourriture du machine learning, l'automatisation de l'entraînement et de l'inférence.
Le chapitre 3 se concentre le fait de tirer profit d'un projet de machine learning. Le monitoring et le passage à l'échelle y sont expliqués.
Le chapitre 4 s'intéresse dans un sens plus large à la conduite du succès d'un projet de système

prédictif. Donner confiance aux utilisateurs dans le modèle et organiser son équipe y sont deux éléments clefs.
Le dernier chapitre constitue la conclusion dans lequel il y aura une ouverture sur d'autres formes de machine learning.

Dans quelle mesure doit-on s'inspirer du développement logiciel classique ?

Cette question revient finalement à se demander quels parallèles nous pouvons établir entre le développement logiciel classique et le développement de solutions d'apprentissage automatique. Je viens du monde du développement back-end. Je ne peux donc m'empêcher personnellement de faire des ponts et ceux-ci reviendront tout au long du guide. Cependant, au-delà de mon expérience personnelle, on est en droit de se demander comment les deux champs sont connectés et tout du moins comment le champ du développement logiciel peut servir à celui du machine learning. Il y a beaucoup de choses à prendre du monde de la programmation traditionnelle. C'est un univers où les solutions de tests, déploiement et monitoring sont matures. Elles le sont beaucoup plus qu'en data science. Ainsi, s'inspirer de ces éléments-là est une bonne chose.

Une fois cela dit, rester attaché au monde du développement logiciel classique sans prendre en compte les spécificités de la data science est dangereux. Beaucoup expliquent que la data science est plus complexe car en plus du code, il y a le modèle et les données à gérer. C'est oublier que le code en programmation classique est plus compliqué qu'en data science. Pourtant, il y a quelque chose d'intéressant dans ces remarques. La data science est un champ relativement nouveau. Il est en tout cas différent de la programmation traditionnelle. J'ai connu des personnes qui n'aimaient pas cette idée. Pour elles, cela revenait à dire qu'on n'était pas obligés de tester son code ou simplement d'en prendre. En résumé, ce serait prendre cette idée pour justifier le fait d'être plus permissif sur la qualité d'une manière générale. Ce n'est pas mon but, bien au contraire. Je crois fermement que le machine learning a des spécificités dont il faut tenir compte pour des productions sereines. Ce n'est que ma conviction, mais beaucoup de projets d'apprentissage automatique échouent parce qu'ils ne tiennent pas compte de cela. Négliger ces éléments et considérer qu'il s'agit de mettre en production n'importe quelle autre forme de projets, c'est prendre un grand risque. S'il y a matière à rédiger un guide sur le sujet, on peut affirmer que la mise en production de machine learning requiert des spécificités qui sont cruciales. Le reste de cet écrit porte finalement là-dessus. Nous allons revenir plus en détail sur tous ces éléments. Je tenais simplement à le signaler d'entrée de jeu et expliquer pourquoi j'établirai de nombreux parallèles avec le développement classique.

Clarifications lexicales

Comme précédemment expliqué, il ne s'agit pas définir réellement les termes d'intelligence artificielle, apprentissage automatique ou autre. Il s'agit simplement de faire une mise au point pour que nous parlions tous de la même chose au sein de ce guide.

Intelligence artificielle

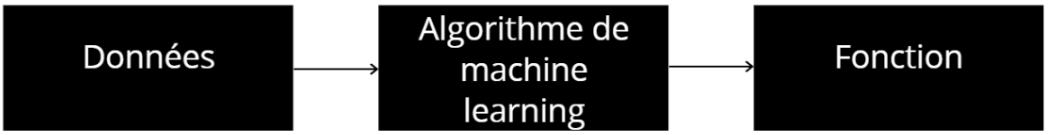
Depuis longtemps, on cherche à concevoir une machine qui nous ressemble. L'article *Computing machinery and intelligence (Machine de calcul et intelligence)* écrit en 1950 par Alan Turing sur le sujet est édifiant. Sa solution est simple : une machine est intelligente si on la confond avec un humain. Cependant, au-delà de toutes ces considérations presque philosophiques, une définition plus simple de l'intelligence artificielle est la suivante : l'intelligence artificielle, c'est le fait d'automatiser un processus cognitif. Par exemple, estimer une tâche est un processus cognitif. Si un algorithme est capable d'automatiser cela, on peut parler d' « intelligence artificielle ».
Pour la petite histoire, dans son article, Alan Turing parle des « learning machines » (machines apprenantes). Ce sont des ordinateurs avec des capacités d'enfant, donc capables d'apprendre et donc de grandir. C'est un peu l'idée qu'on retrouve aujourd'hui dans le machine learning.

Intelligence artificielle spécialisée et générale

On désigne à ce jour deux formes d'intelligence artificielle. L'une est dite forte et générale. Celle-ci relève pour l'instant de la science-fiction. Elle renvoie à l'idée d'une machine capable de simuler entièrement un être humain. Ce n'est pour l'instant pas d'actualité. Jusqu'ici, seuls les écrivains et scénaristes y sont parvenus au sein de leurs œuvres.
Il y a aussi l'intelligence artificielle dite faible et spécialisée. Le terme « faible » pourrait être pris dans un sens négatif. Mais cette intelligence artificielle faible est la seule aujourd'hui qui a une réalité, qui génère des revenus et améliorent nos processus et peut-être, je l'espère, un peu nos vies. Notre cas d'estimation de tâche est un cas d'intelligence artificielle faible. En effet, nous sommes loin du but d'Alan Turing. L'algorithme ne sait pas faire grand-chose. Il ne sait faire qu'une chose. Mais il vaut mieux savoir faire une seule chose et la réaliser efficacement.

Apprentissage automatique

L'une des manières de parvenir à l'intelligence artificielle, c'est l'apprentissage automatique, ce qu'on appelle donc aussi en anglais le « machine learning ». Il s'agit de la capacité pour une machine à réaliser des choses sans qu'on les lui ait explicitement indiquées. On ne programme pas directement un comportement. La machine apprend et s'adapte. On a des données auxquelles on applique un algorithme. Cela nous retourne alors une fonction.



Supervisé et non supervisé

L'apprentissage automatique est possible de plusieurs manières. On définit généralement deux types d'apprentissage. Il y a d'abord ce qu'on appelle l' « apprentissage supervisé ». Le guide traite de ce dernier. Pour celui-ci, nous avons à notre disposition des données qui ont une cible (une « target »). C'est ce qu'on cherche à prédire. La cible est étiquetée. On dit aussi « labelisée ». Cet étiquetage peut être naturel si la chance est de notre côté. Sinon, il est réalisé manuellement par des équipes dédiées. Il y a aussi des entreprises où la tâche est réalisée par les data scientistes eux-mêmes. Cette tâche étant fastidieuse, il existe des solutions pour automatiser au moins en partie ce processus. Ce sujet sera abordé dans les grandes lignes plus loin. Je vous propose de revenir à la définition de l' « apprentissage supervisé ». Supposons que nous cherchions à prédire la température du jour. Pour ce faire, nous avons des données. Celles-ci représentent la cible « température ». Elles ont aussi d'autres informations qui permettent d'expliquer la cible. Cela peut ressembler à cela :

Température de la veille	Mois	Couverture nuageuse	Température
12	Mars	40%	15
5	Décembre	50%	6

L'idée est de prédire la cible en arrivant à établir une corrélation entre des informations telles que « température de la veille », « mois », « couverture nuageuse » et la cible « température ». Les éléments qui permettent de prédire la cible sont appelés « features ».

De cette manière-là, lorsqu'on souhaite prédire la « température », on pourra la deviner avec une justesse intéressante. Dans le cas suivant, on n'a que les champs « Température de la veille », « Mois » et « Couverture nuageuse ».

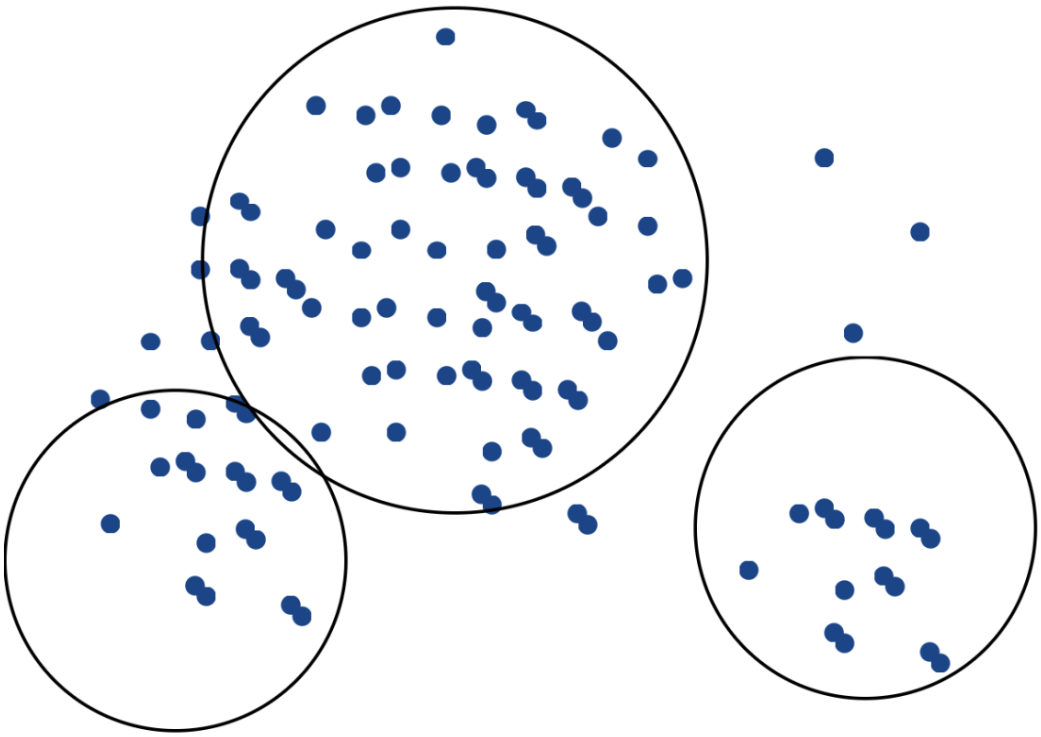
Température de la veille	Mois	Couverture nuageuse
11	Mars	20%
7	Décembre	90%

Grâce à notre algorithme d'apprentissage supervisé, on pourra induire la température :

Température de la veille	Mois	Couverture nuageuse	Température prédite
11	Mars	20%	15

Température de la veille	Mois	Couverture nuageuse	Température prédite
7	Décembre	90%	5

Il existe aussi ce qu'on appelle « l'apprentissage non-supervisé ». Ce dernier n'essaye pas d'établir de corrélation entre les différentes informations qu'on appelle « features » et une « target ». Ce genre de machine learning tente plutôt de repérer des structures au sein même des données. Cela est utile par exemple pour faire du clustering, c'est-à-dire regrouper des données similaires entre elles.



Il existe aussi ce qu'on appelle l'apprentissage « semi-supervisé ». Nous n'en parlerons pas vraiment dans la suite. Cependant, il est de plus en plus plébiscité.

Apprentissage profond et peu profond

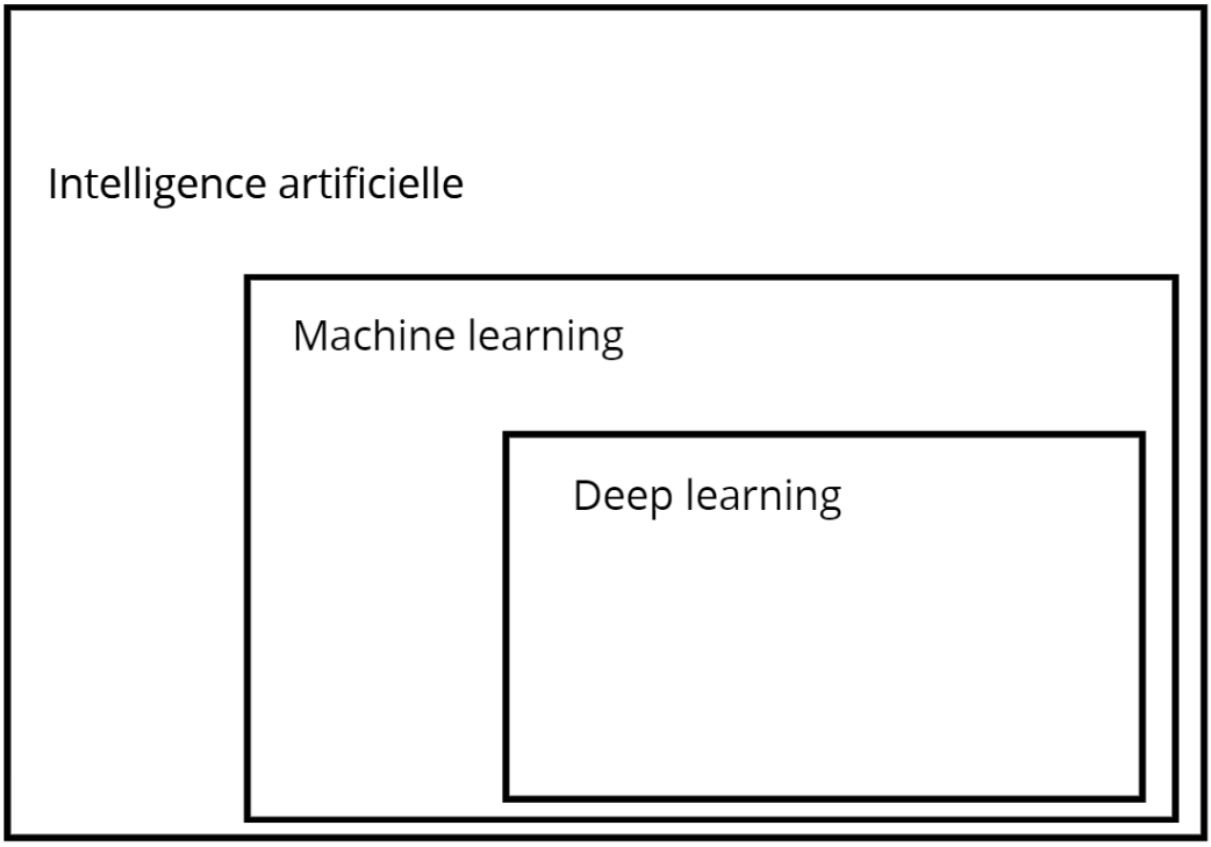
L'apprentissage peu profond fait référence à tout ce qui est régression linéaire, logistique, SVM, etc. Cela fait aussi référence aux théories ensemblistes avec RandomForest ou encore GradientBoosting Tree. Ce sont des modèles généralement plus simples à comprendre et qui ne multiplient pas les couches de calcul. On a des entrées, un algorithme de machine learning et une sortie immédiate.

L'apprentissage profond fait référence principalement aux réseaux de neurones quand ils accumulent plusieurs couches de calcul. En effet, si un réseau de neurones n'a qu'une couche, ce n'est pas un modèle profond. Le stacking est une technique qui se rapproche de l'apprentissage profond. Il s'agit d'avoir plusieurs étapes de prédiction en passant par différents modèles. On a par exemple des données qui entrent dans une régression linéaire. La sortie sert alors d'entrée à un arbre de décision.

Je traiterai indifféremment ici d'apprentissage profond ou peu profond. Je n'ai pas l'intention de dédier une partie à l'un ou l'autre. Les différences sont inintéressantes lorsqu'il s'agit d'aller en production.

Imbrication apprentissage profond, machine learning et intelligence artificielle

Afin de clarifier la place de chacun, je vous propose un schéma. L'apprentissage profond (« deep learning » en anglais) est un type de machine learning. Le machine learning est une manière d'arriver à l'intelligence artificielle.



Entraînement, réentraînement

La première fois qu'on lance son algorithme d'apprentissage, on dit qu'on l'entraîne. Par la suite, on le réentraînera avec de nouvelles données et parfois de nouveaux hyperparamètres. Si on voulait être puriste, à partir de ce moment-là, on devrait tout le temps dire « réentraîner » puisque le modèle a déjà été entraîné une fois au tout début. Pour ma part, j'ai tendance à confondre les deux. La différence n'est pas importante dans le contexte de la mise en production et ne nécessite pas qu'on s'y attarde.

Feature engineering

Il s'agit du fait de prendre des données brutes et d'en faire des « features » exploitables. Par simplification, je l'inclus dans la partie « entraînement ».

Inférence, scoring, prédiction

Inférence, scoring et prédiction représentent trois termes qui renvoient à la même chose. Il s'agit du fait de délivrer la ou les prédictions finales à l'utilisateur. Cela comprend le fait de calculer les prédictions. Cela peut être fait de différentes manières : batch ou à la demande principalement. Cela inclut aussi le fait de donner à voir ces prédictions par l'intermédiaire d'API, de dashboards ou de tout autre moyen jugé adéquat.

MLOps et DataOps

Ces deux concepts sont souvent confondus. C'est la raison pour laquelle je voudrais m'arrêter dessus quelques instants.

Wikipedia écrit : « DataOps est une méthodologie automatisée et orientée processus, utilisée par les équipes d'analyse et de données, pour améliorer la qualité et réduire le temps de cycle de l'analyse des données. ». Finalement ce qu'on trouve à l'intérieur de ce concept, ce sont des méthodes DevOps pour déployer des pipelines de données comme de l'extraction de données et de la transformation de données. Il s'agit aussi de monitorer ces pipelines. Le DataOps est plutôt appliqué dans les équipes de data ingénierie et de data analyse.
Le MLOps renvoie davantage à tout ce qui gravite autour du machine learning. Ce sont toujours des méthodes DevOps appliquées cette fois-ci au déploiement et monitoring de modèles de machine learning. L'idée est d'assurer une reproductibilité et de pouvoir aussi mettre ses modèles à l'échelle. La cible de ce concept est davantage les équipes de data science, même si elles peuvent collaborer avec d'autres équipes pour atteindre ce résultat.
Finalement, le DataOps est davantage utilisé pour déployer et monitorer des pipelines de données. Le concept de MLOps se concentre sur le machine learning. On peut avoir du DataOps sans MLOps parce qu'on peut faire de l'extraction et de la transformation de données sans machine learning. Le contraire est rarement vrai. J'évoquerai un peu les concepts de DataOps quand je parlerai des extractions de données. Cependant, ce guide est surtout orienté MLOps.

Pipelines

Le terme de « pipelines » est un mot très utilisé en data. Cela prête à confusion. Ce terme renvoie à plusieurs choses. Un pipeline d'une manière générale est un enchaînement d'actions qui prend une entrée et la transforme jusqu'à obtenir un résultat.
On parle de « data pipeline ». Il s'agit du fait de transformer de la donnée. Par exemple, quand on collecte de la donnée et qu'on la transforme pour lui donner du sens ou en extraire des « features

», on est face à une « data pipeline » ou « pipeline de données ». L'entrée est de la donnée brute. Le résultat est de la donnée enrichie. On peut réaliser cela avec du code. Certains outils comme Denodo ou Talend permettent aussi de faire cela de manière semi-automatique.

On parle aussi de « pipeline de machine learning ». On commence par prendre des données en entrées. Les features n'ont pas toujours été extraites à ce stade-là de l'opération. Cela peut se faire au sein même du pipeline. En sortie, on a le modèle. De nouveau, on peut réaliser cela avec Python ou R. Java semble un peu compliqué à utiliser ici étant donné qu'il y a peu de librairies de machine learning dans ce langage. Il existe aussi des outils pour réaliser cela de manière graphique. Dataïku par exemple est assez connu dans le domaine.

Enfin, il y a aussi les « pipelines de déploiement ». Ceux-ci permettent, comme leur nom l'indique, de déployer. Cela signifie qu'on déplace le système prédictif sur le bon serveur de manière automatique pour qu'il puisse être déployé. Cela s'accompagne généralement d'une phase de validation. C'est là par exemple qu'on lance les tests de code. Finalement, on peut réaliser cela avec des scripts Bash ou Python. Aujourd'hui, beaucoup d'outils nous simplifient la vie et nous permettent d'écrire une grande partie avec des fichiers yaml par exemple. On déclare ce qu'on veut faire pour chaque étape.

Une fois, ces explications données, on est en droit de vouloir revenir à notre sujet principal, à savoir la mise en production de machine learning. La question sous-jacente qui nous intéresse dès à présent est la suivante : quand faut-il commencer à se projeter en production ?

L'idée de mise en production doit être présente dès le début du projet

Il faut pouvoir envisager la production dès qu'on commence un projet. Cela ne signifie pas qu'il faut s'interdire des expérimentations pendant la phase de prototypage. Mais il s'agit d'avoir en tête la finalité de ce que l'on fait pour se guider dans ses choix.

Avant même que le projet commence, l'idée doit être là

Il s'agit ainsi de s'interroger sur la pertinence de nos choix et du projet même.
L'impact du machine learning est grand quand il remplace une partie complexe du business ou qu'on est d'accord avec le fait d'avoir des prédictions peu coûteuses, mais imparfaites. Dans d'autres cas, cela est moins intéressant.

Il s'agit de s'interroger sur les différents facteurs qui peuvent influencer les coûts d'un projet de machine learning. C'est pourquoi il s'agit de se poser certaines questions. Quelle est la difficulté du problème qu'on cherche à résoudre ? Combien coûte le fait d'acquérir et maintenir les données ? Quelles sont les performances requises pour le modèle ? Apporter des réponses à ces questions permet d'évaluer la pertinence de notre projet.

Il est à noter qu'il est cependant assez difficile d'estimer réellement un projet de machine learning. Il y a beaucoup d'incertitudes. Par exemple, on ignore le nombre de données dont nous aurons besoin.

On ignore le nombre de features et lesquelles seront pertinentes. Finalement, on ignore même s'il sera possible d'atteindre la performance désirée. Si celle-ci est atteignable, on ne sait pas les conséquences que cela aura en termes de temps de calcul. On ne peut répondre à ces questions qu'en se plongeant dans le projet.
De plus, il est à noter qu'en data science, les progrès sont non linéaires. Au début, les performances du modèle augmentent rapidement, puis stagnent. Les avancées sont alors moins spectaculaires et plus lentes.
Une solution pour prendre de moins grands risques est de simplifier le problème. Au lieu de prédire mille classes par exemple, on peut commencer par essayer d'en prédire dix.

Au moment de la collecte des données, l'idée de mise en production est toujours présente

D'autres questions se posent alors dans cette phase du projet. Est-ce que la donnée est accessible ? A quel prix en termes de solutions techniques ? Combien de temps mettra-t-on pour y parvenir ?
On ne peut pas répondre à ces questions avant de se plonger dans les données dont nous avons besoin. Mais avoir ces choses en tête permet d'évaluer tout le long du projet la faisabilité réelle de celui-ci.
Il y a des problèmes récurrents avec les données dont il faut se méfier. Elles ont parfois peu de valeurs prédictives, comportent des biais ou représentent des exemples dépassés.
De bonnes données sont faciles d'accès, ont un pouvoir prédictif fort et sont capables de généraliser. Il faut de prime abord éviter d'avoir un jeu d'entraînement qui n'aurait pas la même distribution que les données dans la réalité. On peut avoir recours à du suréchantillonnage ou sous échantillonnage pour pallier ce problème.

Au moment du feature engineering, l'idée de mise en production est toujours présente

De bonnes « features » ont un pouvoir prédictif fort, se calculent rapidement, sont non corrélées et sont fiables. Elles sont unitaires, faciles à comprendre et à maintenir. L'extraction de « features » est l'une des parties les plus importantes d'un projet de machine learning. Sa maintenance en production peut se révéler lourde. C'est pourquoi elle doit être intensément et systématiquement testée.

Au moment du choix du modèle, l'idée de mise en production est toujours présente

De nouveau, de nouvelles questions permettent d'évaluer la pertinence de nos choix. En premier lieu, il s'agit de se demander si mon modèle a besoin d'être facilement interprété et compris. Même s'il existe des manières pour parer à l'opacité des modèles, certaines parties prenantes cherchent parfois à avoir un fort contrôle sur le modèle. Cela arrive notamment quand on commence à mettre en place de l'intelligence artificielle dans une entreprise. Il faut donner confiance aux personnes qui

n'en ont pas l'habitude ou s'en méfient. Cela prend parfois du temps. Ainsi, si on a besoin de fortes explications autour du modèle, il vaut mieux partir sur des modèles simples comme des modèles linéaires, KNN ou des arbres de décisions.
Une autre question est celle du temps de calcul. Avons-nous besoin d'un modèle qui se calcule très rapidement ou sommes-nous libres à ce sujet ? C'est une question dont la réponse influence aussi notre choix du modèle. Effectuer des calculs avec un réseau de neurones peut s'avérer coûteux en termes de temps d'exécution.

Au moment de l'évaluation du modèle, l'idée de mise en production est toujours présente

Quand on pense à cette partie-là du projet, souvent, on se voit évaluer la capacité du modèle à effectuer de bonnes prédictions. On sépare ses données en deux et on fait de la validation croisée. Ce n'est pas suffisant dans l'objectif de la production. Il s'agit dans cette étape de se demander s'il y a un risque financier ou légal à mettre le modèle à destination d'utilisateurs finaux. On essaye en fait de se projeter pour connaître d'une manière générale les impacts potentiels et négatifs que pourrait avoir le système prédictif. La plupart du temps, on envisage les retombées positives, mais peu les retombées négatives.
C'est aussi dans certains cas le moment de se demander si le modèle est éthique et robuste. En combien de temps est-il capable de répondre ? Est-il discriminant ? Évaluer son modèle, c'est donc plus que vérifier la justesse de celui-ci.
Il existe ce qu'on peut appeler une validation hors ligne. On vérifie les résultats du modèle par rapport à d'autres données, mais sans le mettre à destination des utilisateurs. Il y a aussi une validation en ligne où on dessert le modèle à une partie de la cible prévue. Dans les grandes entreprises, cela se fait par l'intermédiaire de solutions d'AB testing complètement automatisées par exemple. Dans les industries plus petites, on peut penser à des phases de pilotage où on sélectionne des utilisateurs pilotes.

Finalement, l'idée de production court tout du long d'un projet. Il est vrai qu'il n'est pas nécessaire de mettre en production tous les prototypes. Certains sont des tests qui n'aboutissent à rien. Il ne faut pas insister. En fait, plus justement, il s'agit d'évaluer les prototypes constamment par rapport à la production en imaginant tout ce que cela peut impliquer : coût des données, du monitoring, bénéfices versus investissement, éthique, etc.

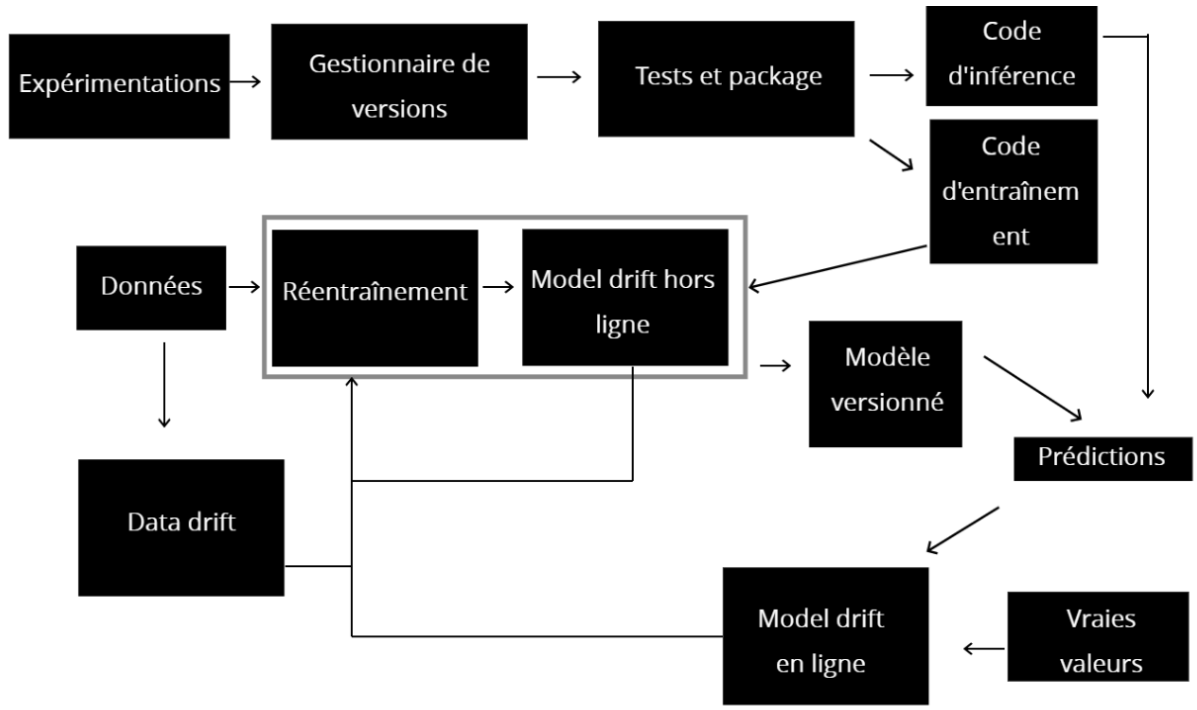
Pour avoir une idée de comment intégrer plus avant cet esprit tout au long du projet, dès le début du prototype, je recommande le livre *Machine learning engineering* de Andriy Burkov. Vous retrouvez plus en détail plusieurs idées évoquées ci-dessus.

La mise en production de machine learning est donc un sujet vaste qui court tout au long du projet. La première étape pour pouvoir rentabiliser un système prédictif est en fait l'automatisation.

Automatiser un système prédictif

Pour tirer profit d'un système prédictif, il s'agit d'abord de l'automatiser, de passer de la phase de prototypage qui peut contenir des parties manuelles à une solution automatisée.

Avant d'entrer plus avant dans ce qui va suivre, je voudrais d'abord vous partager un schéma d'un processus qu'on peut envisager concernant la collecte des données, l'automatisation de l'entraînement, l'inférence et quelque peu le monitoring de cet ensemble. Il ne sera peut-être pas aisé à comprendre du premier coup d'œil, mais j'y reviendrai. L'idée générale est de réaliser des expérimentations, de pouvoir versionner ce code dans un gestionnaire de versions afin de pouvoir lancer des tests et déployer le tout. À la suite de cela, on peut programmer un réentraînement qui à partir de données produit un modèle qu'on versionne. Le code déployé d'inférence récupère ce modèle et permet de générer des prédictions. Il y a plusieurs étapes de monitoring. On vérifie l'état des données qu'on reçoit, la performance du modèle pendant l'entraînement et la performance du modèle une fois qu'il est introduit en production.



Quelle part d'automatisation est souhaitée et acceptable dans le machine learning ?

L'automatisation en machine learning nous simplifie la vie. Quand on parle d'automatisation, on fait souvent référence à des outils ou des techniques. Ceux-ci pourraient nous amener à tout automatiser. Mais cela est-il vraiment toujours pertinent ? On sait que notre système peut faire des erreurs. Dans quelle mesure alors l'automatisation est souhaitée et acceptable dans le machine learning ? Pour répondre à cette question, j'aimerais ici évoquer trois techniques qui facilitent le travail des data scientistes mais sont aussi à prendre avec précaution. Il s'agit du cloud, de l'auto-ml et de l'intervention humaine. Évaluer ces trois éléments nous permettra d'avoir une meilleure idée du degré d'automatisation nécessaire en machine learning.

L'intérêt du cloud

Nous sommes entrés dans une nouvelle ère, celle du cloud. Il est partout ou presque. Il y a encore des entreprises qui sont réfractaires pourtant. Plutôt que de se demander qui a tort ou raison, voyons l'intérêt en data science d'utiliser une solution cloud. Il y en a plusieurs. La première c'est que cette solution facilite la mise à l'échelle. On peut augmenter rapidement et simplement ses ressources. Ce n'est pas négligeable quand on sait qu'il y a beaucoup de projets en data science qui sont des projets de big data où il faut augmenter le nombre de machines sur lesquelles on effectue le calcul. C'est donc pratique d'avoir un système comme celui-ci. Il n'est pas besoin d'avoir à payer pour un nombre permanent de machines. Les coûts s'ajustent en fonction des ressources nécessaires à tel ou tel traitement.
Le deuxième avantage au cloud, c'est que derrière ce terme, se cachent des plateformes protéiformes qui offrent beaucoup de facilités. Databricks que je connais bien par exemple permet de faire du Spark, du Tensorflow, du Scikit-Learn, intègre un outil de data versioning, un outil de versioning de modèle, de quoi paramétrer l'exécution de ces travaux, etc. Azure Machine Learning Studio aussi propose ce genre de choses. Finalement, on se retrouve avec bien plus qu'une solution cloud. On a à sa disposition de nombreux services qui peuvent nous aider au quotidien. Il est plus facile de démarrer avec ce genre de choses. On n'a pas besoin de mettre en place du monitoring, pas besoin d'installer toutes les librairies nécessaires, etc. On peut commencer à mettre en place rapidement notre système.

L'inconvénient majeur au cloud c'est le fait que chaque plateforme a ses spécificités. Aujourd'hui, on n'entend pas encore d'histoires d'entreprises qui ont migré d'une plateforme à une autre, mais cela arrivera pour sûr. Certaines plateformes finiront par être dépréciées ou se spécialiser. L'intérêt de migrer sera alors présent. Si on a développé son projet selon les spécificités de la plateforme, ce sera compliqué de migrer. C'est pourquoi malgré le confort qu'apportent les plateformes de Cloud, il s'agit de s'en méfier un peu. C'est un service pratique mais qu'il ne faut pas voir comme une solution qui perdura à jamais. On a beaucoup parlé de la mode des frameworks. On a vu des migrations de toute part. Les développeurs ont alors avancé l'idée d'avoir un code agnostique qui ne sert qu'a minima des fonctions des frameworks. Je pense que c'est le genre de démarche qu'il faut avoir pour

les plateformes cloud. Il vaut mieux garder en tête que le framework actuellement à la mode pourrait demain être déprécié. En informatique, cela arrive tout le temps. Il ne faut pas s'en étonner et se tenir prêt. Une solution est aussi de se baser sur des plateformes reposant elles-mêmes sur des outils open-source.

L'intérêt de l'auto-ml

J'ignore quelle place prendra l'auto-ml dans nos vies de data scientistes. Je ne le vois pas comme un danger. En effet, une bonne partie du travail du data scientiste consiste à comprendre les données, formuler le besoin et trouver des solutions pour y répondre. L'auto-ml a pour vocation d'aider à définir quel modèle est le plus pertinent et avec quels hyperparamètres. On voit aussi de plus en plus des solutions d'auto-feature-engineering. Ces solutions peuvent avoir un impact sur la manière dont on met un modèle en production. Cela dépend de la façon dont on les utilise. Si on se sert de l'auto-ml et de l'auto-feature-engineering uniquement dans les phases de modélisation avec une intervention humaine, cela est sans conséquence. En revanche, il y a des cas où lors du réentraînement, on lance une phase d'auto-ml pour définir quel modèle utiliser. Dans ce cas-là, c'est plus compliqué. Cela veut dire qu'on laisse la machine s'adapter au meilleur modèle. Ce n'est pas toujours pertinent. Il s'agit de se demander si on a besoin d'une forte interprétabilité et de comprendre les détails du modèle. Si oui, l'auto-ml n'est sans doute pas une solution à envisager. L'auto-ml pose aussi la question du temps d'exécution. Avons-nous besoin d'un entraînement rapide ? Si oui, cela signifie aussi que l'auto-ml est à écarter. Il y a des méthodes d'auto-ml qui s'appuient sur des algorithmes génétiques et accélèrent le temps de calcul. Malgré tout, cela reste coûteux en termes de temps. La même question se pose pour l'inférence. Avons-nous besoin d'une réponse rapide en inférence ? Si oui, on ne pourra peut-être pas utiliser tous les modèles disponibles dans la base de notre outil d'auto-ml. Certains sont plus rapides que d'autres pour rendre un résultat. Ce sont autant de choses qu'il faut prendre en considération.

L'intérêt d'une intervention humaine dans le système prédictif

En programmation traditionnelle, cela paraîtrait aberrant de ne pas tout automatiser. On déploie le code automatiquement du début à la fin. En data science, il y a parfois des interventions manuelles dans le processus de déploiement. La raison la plus évidente, c'est par exemple pour faire contrôler par un humain l'éthique du modèle. Même si de plus en plus de logiciels se développent pour nous aider, l'éthique est un concept beaucoup trop humain pour qu'une machine comprenne exactement de quoi il s'agit. Ainsi donc, un humain a toute sa place ici. On peut envisager qu'après un réentraînement, un humain vérifie manuellement les métriques et résultats d'une manière générale. Il choisit alors de promouvoir ou non le nouveau modèle. Cela semble plutôt intéressant comme stratégie.

Il s'agit cependant de ne pas tomber dans l'effet où toutes les actions prises sont humaines. Si on réentraîne tous les jours une vingtaine de modèles et qu'on a des actions manuelles pour chacun, cela risque vite de devenir fastidieux et coûteux. Une manière de pallier cela est d'évaluer quels modèles sont critiques et nécessitent une intervention humaine. Sans doute que tous les modèles

n'ont pas besoin de validation humaine. On peut aussi envisager des interactions humaines au début du démarrage d'un projet, mais tendre vers une totale automatisation. Quand on n'a que quelques projets en production, avoir quelques actions manuelles semble jouable. Mais quand le nombre de modèles commence à se développer, cela devient un goulot d'étranglement qu'il faut éviter. Dans l'article *Continuous Delivery for Machine Learning*, les auteurs expliquent que le problème en data science n'est pas uniquement que beaucoup de projets ne partent jamais en production. En fait, certains y vont tout de même mais avec beaucoup d'actions humaines qui rendent la mise à jour et la maintenance compliquée. C'est pourquoi il faut tendre vers l'industrialisation totale et réserver un humain dans la boucle uniquement de manière ponctuelle uniquement sur des sujets critiques. Bien sûr, seules des personnes peuvent améliorer et réparer les problèmes quand ils arrivent en production. Seules des personnes peuvent comprendre les alertes qui sont remontées. Ne nous méprenons pas. Il ne faut pas bannir toute intervention humaine. Seulement, tout ce qui peut être automatisé doit être considéré comme tel pour éviter de perdre du temps et de faire des erreurs. En effet, l'erreur est humaine. Même quand la machine fait une erreur, son origine est humaine. C'est celle du développeur.

Les systèmes prédictifs doivent être largement automatisés

C'est en automatisant dans une grande mesure les systèmes prédictifs qu'on les conduit au succès. Cependant, tout ne peut pas toujours être automatisé.
Le but ici était de présenter les avantages à utiliser une plateforme cloud, de l'auto-ml et un humain dans les processus que nous cherchons à automatiser.
Les principaux intérêts du cloud sont la scalabilité qui peut être pratique avec de fortes volumétries de données et le fait que la plateforme intègre une panoplie d'outils. Les dangers sont que tout n'est pas toujours open source ou faciles à exporter vers d'autres environnements. Dans le futur des migrations auront lieu.
L'auto-ml a comme avantage de desservir le meilleur algorithme sans intervention humaine. Cependant, il faut être sûr de ne pas avoir de contraintes trop fortes d'interprétabilité ni de contraintes trop fortes temps de calcul en entraînement et en inférence.
L'humain bien entendu est la personne qui crée le système prédictif. On peut envisager qu'à chaque entraînement, un humain valide si on dessert le nouveau modèle. Cela permet de vérifier des éléments comme l'éthique, difficile à appréhender pour une machine. Cependant, quand on a une multitude de projets qui ont un réentraînement par jour, garder un humain dans la boucle est coûteux. Il s'agit donc de penser le plus possible à l'automatisation et de réserver l'intervention humaine pour quelques cas précis.

Nous parlons d'automatisation dans ce chapitre. Il était donc important de voir un peu les limites de cela. D'un côté, on peut vouloir tout automatiser avec le cloud et l'auto-ml, mais cela a des limites. De l'autre côté, on peut vouloir limiter cette automatisation en remettant l'humain au cœur de tout cela. Cela comprend aussi des risques si on le fait trop souvent.
Ainsi, il faut automatiser. Il y a trois grandes parties qui sont concernées par cela : la mise à jour des données, l'entraînement et l'inférence.

Construire un socle solide avec des données pérennes

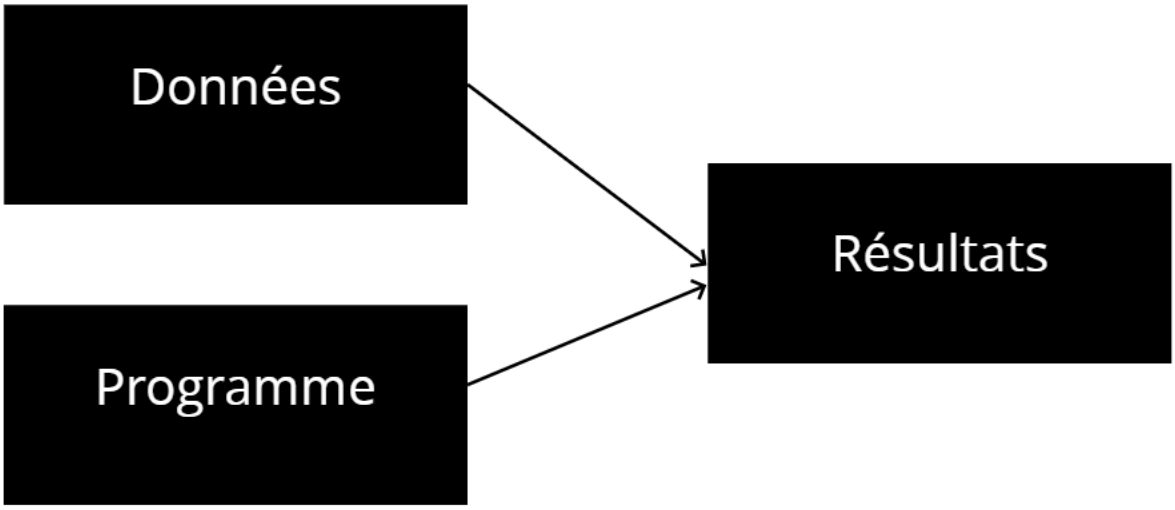
La première chose à faire est d'automatiser la mise à jour des données et de la rendre pérenne.

L'importance des données en machine learning

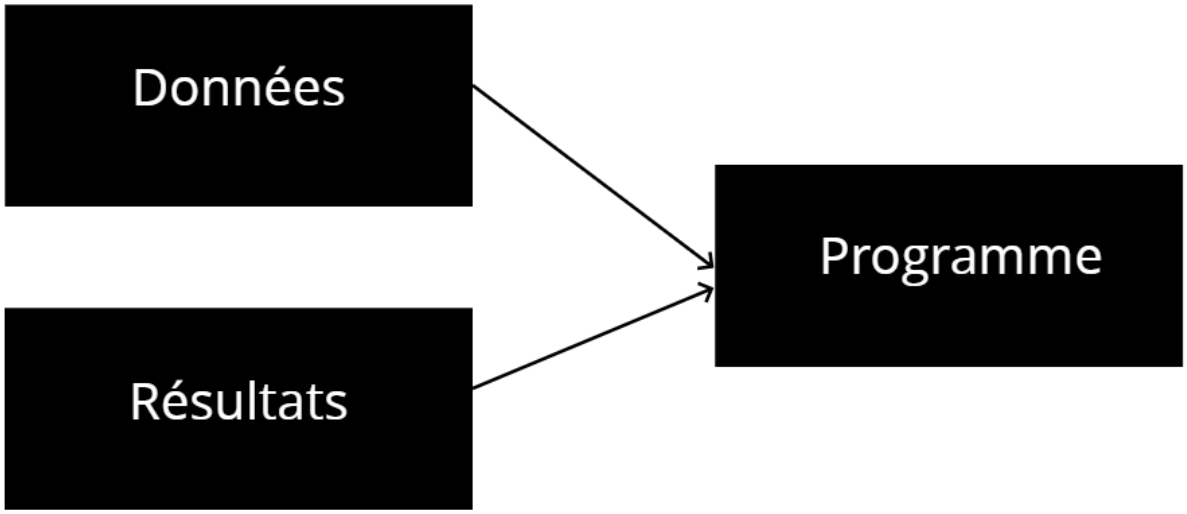
Les données sont la matière première du machine learning. Le terme « data » est un buzzword qui a fait couler beaucoup d'encre. De nombreux ouvrages y ont été consacrés. Le « big data » était le terme à avoir sur son CV. Les foules étaient en délire autour de ces concepts. Les choses se sont aujourd'hui un peu apaisées. Il serait possible que ce soit d'ailleurs pour donner place à d'autres buzzwords comme « intelligence artificielle ».
Il est à noter que les termes de « data scientiste », « data ingénieur », « data analyste », « data architecte » et « développeur data » sont restés. Cela vient finalement du fait que la donnée est ce qui nous permet de construire nos algorithmes. Elle est au cœur de ceux-ci. Nous cherchons et découvrons des comportements dans différentes données. Nous apprenons des structures. Cela nous donne la possibilité de les prédire.

Dans le développement logiciel plus classique, la qualité dont on a besoin pour un programme réussi réside en grande partie dans le code. Il doit être à même d'exprimer le métier. Il le faut lisible et facilement modifiable. C'est là le principal enjeu.
Dans la data science, les choses sont un peu différentes. De mauvaises données peuvent ruiner un système de machine learning bien construit et écrit. Supposons un modèle très performant. Il peut arriver de nouvelles données biaisées, erronées ou simplement différentes de ce que à quoi le modèle s'attend. Les performances de celui-ci peuvent se retrouver anéanties.
Au-delà de cet aspect, les données sont en fait la nourriture de nos algorithmes. Dans le développement plus classique, nous avons des données. Nous construisons un programme et obtenons un résultat.
C'est bien différent avec la data science. Dans ce genre de projets, nous avons des données et un résultat que nous voulons obtenir. Le but du jeu consiste à construire un programme à partir de là. Nous ne connaissons pas celui-ci en avance. Et d'ailleurs, une partie est stochastique : le modèle lui-même.
Dans la programmation plus classique, on cherche à implémenter une spécification. Celle-ci est parfois difficile à définir. On itère souvent pour parvenir à une solution convenable. En data science, il n'y a pas cette notion de spécification. On cherche à optimiser une métrique. Pour cela, on s'appuie sur des données auxquelles on applique un modèle. Le processus est itératif et s'apparente beaucoup à de la recherche. On peut considérer que la data science est pour partie du développement logiciel, pour partie de la recherche. C'est la partie scientifique du terme « data science ».
Matei Zaharia résume ces points dans une conférence donnée en octobre 2019. Matei Zaharia est le créateur d'Apache Spark. Ce dernier est un logiciel open source qui permet de réaliser du calcul distribué. Il contient notamment de nombreux algorithmes d'apprentissage automatique. Matei Zaharia a aussi cofondé Databricks et est à l'origine de MLFlow, un outil qui permet de versionner des modèles. La conférence que j'évoquais a été réalisée en collaboration avec Corey

Zumar. Elle a pour titre *Simplifying Model Management with Mlflow (Simplifier le la gestion de modèles avec Mlflow)*. Matei Zaharia y explique les difficultés du machine learning par rapport au logiciel traditionnel. Dans ce dernier, le but est de répondre à une spécification. Comme nous l'avons déjà évoqué, la qualité dépend du code. De plus, on peut choisir une et une seule stack technique, par exemple Java Spring avec Vue.js. Il y a plusieurs frameworks et langages dans l'exemple que je prends. Mais on n'est point obligé de jongler tantôt avec Tensorflow, tantôt avec Apache Spark pour trouver le meilleur algorithme qui réponde à nos besoins. En data science, il ne faut pas avoir peur de changer de stack technique. On réalise cela continuellement.
Dans le machine learning, en cherchant à optimiser une métrique, on expérimente constamment. La qualité dépend aussi des hyperparamètres et de la donnée.
Dans la manière de faire de la data science, il y a comme un renversement. C'est ce qu'explique Pierre Delort dans son livre *Big data, que sais-je*. Dans l'informatique traditionnelle, on a en entrée des données et un programme. En sortie, on obtient un résultat.



C'est un peu différent avec le machine learning. Dans celui-ci, on a des données et un résultat en entrée. Cela nous permet d'obtenir un programme.



On voit donc l'intérêt des données dans ce processus. Elles ont une forte importance. Pour construire un système de machine learning efficace, elles doivent être de qualité. Bien souvent, on ajuste de ci et de là les défauts de la donnée. On peut combler les éléments vides, les filtrer, etc. Il arrive que la donnée soit jugée trop mauvaise pour continuer. Dans tous les cas, on comprend qu'elles sont la base de nos systèmes. Une partie non négligeable d'une mise en production sereine repose sur elles. Elles conduisent le programme. C'est aussi leur volume qui permet de choisir entre des modèles distribués ou non.

Dans la phase exploratoire, on s'accommode souvent de données temporaires tirées de ci et de là. On commence parfois à travailler sur un fichier CSV. Celui-ci représente un certain historique et est déjà suffisant pour se donner une idée d'un modèle, commencer la partie feature engineering, etc.
Pour passer en production, il s'agit d'apporter davantage de soin à tout cela. Il s'établit une nouvelle phase d'exploration qui devrait avoir lieu le plus tôt possible : mais d'où viennent ces données ? Il arrive que les données soient le fruit d'un traitement fait entre différentes tables de bases de données. Autrement dit, il faut généralement commencer par répertorier les sources des données et le chemin que les données ont pris pour arriver dans l'état dans lequel elles sont.
Penser à comment exploiter les données sereinement au-delà du prototype, en dehors du fichier CSV qu'on possède, peut nous économiser du temps. Cela nous permet de nous projeter. Il s'agit d'avoir une meilleure idée de la solution finale. On peut aussi déjà envisager si elle est faisable et plus ou moins facile à réaliser. C'est après tout l'un des buts du prototypage : étudier la faisabilité.

Solutions pour des données pérennes

Pour avoir un socle solide, il faut des données régulières, versionnées et de confiance

Gérer les dépendances

Les données sont les principales dépendances que nous avons dans un projet de machine learning. Cela signifie que si quelque chose se passe mal dans celles-ci, la suite risque d'être problématique aussi. Les dépendances dans un projet sont toujours compliquées. Dans notre cas, il s'agit de penser celles-ci et d'envisager les potentiels échecs. Supposons une équipe data censée recevoir d'un service externe de la donnée tous les jours à 4h du matin. Si celles-ci n'arrivent pas, il s'agit de savoir comment répondre à ce problème. Plusieurs options s'offrent à eux. On peut décider de ne rien faire. Le système est trop critique. Il vaut mieux ne rien délivrer que délivrer quelque chose d'erroné ou avec des éléments manquant. On peut aussi décider de garder le dernier modèle, celui de la veille donc. Enfin, on peut aussi essayer de régler le problème en se donnant une heure maximale avant de considérer une autre solution. Les réponses sont variées. Cela dépend de la criticité du système et des moyens de l'entreprise. Il s'agit d'avoir une idée de comment répondre au problème avant que celui-ci n'arrive. Pour sûr, ce genre de problème arrivera. Autant s'y préparer.

La régularité des données

Malgré le fait que ce soit compliqué, il s'agit d'essayer d'avoir un flux de données régulier. C'est là un enjeu. Il s'agit de trouver des solutions pour que les données parviennent jusqu'au système d'apprentissage automatique de manière régulière. Cela peut paraître aisé dit comme cela. Mais ce n'est souvent pas le cas. Nombreuses sont les équipes qui butent sur cette première étape. Comme nous l'avons dit, il faut répondre à des questions légales. En effet, la RGPD (Règlement Général sur la Protection des Données) ne permet plus aux entreprises de faire tout ce qu'elles veulent des données qu'elles ont en leur possession. C'est une victoire pour l'utilisateur. C'est aussi un casse-tête pour les data scientistes. Il faut s'assurer de n'avoir que les données dont on en a besoin pour réaliser l'apprentissage. On se sera au préalable assuré que l'apprentissage même ne contredit pas la loi. Il s'agit aussi de répondre à des questions logiques : D'où vient la donnée qui a servi à l'exploration ? Quel chemin a-t-elle emprunté pour arriver là ? Une fois, ces questions répondues, il s'agit de mettre en place ce processus de manière automatique.
Cela signifie plusieurs choses. Il faut savoir comment se connecter aux différentes sources de données. On ne se connecte pas à une base de données, comme à un datalake par exemple. De plus, on agit avec des services externes qui ont souvent chacun leur règle d'accès. Il s'agit aussi de veiller à ne pas surcharger les appels à une base de données par exemple et l'empêcher ainsi de tourner. De nombreux outils peuvent être utilisés pour réaliser cela. Il existe de nombreux connecteurs à de nombreuses sources de données. Il s'agit d'en choisir un qui correspond à nos besoins, répond aux exigences de notre entreprise et avec lequel nous sommes à l'aise.
Ensuite, il faut retracer automatiquement le chemin qui a permis aux données d'arriver dans l'état dans lequel elles sont. Il faut être sûr d'avoir toutes les pièces afin de reconstruire le puzzle, puis réaliser celui-ci. On peut alors être amenés à réaliser des schémas comme suit :

Table 1 température de MySQL avec champs degré et jour + Fichier météo du Datalake avec champs intensité nuageuse, pluie et jour = donnée finale avec les champs degré, intensité nuageuse et pluie.

Il y a de nombreux outils qui permettent ensuite de collecter automatiquement les différentes sources de données, de les joindre et de déposer le tout au bon endroit. On est parfois contraints de coder

nous-mêmes certaines parties pour gérer certaines spécificités.
Enfin, non seulement cette extraction, manipulation des données doit être automatique, mais elle doit avoir lieu de manière régulière. Il faut connaître la fréquence que l'on désire. Souvent, on s'adapte aux besoins du business. Parfois, il est nécessaire de faire un entraînement ou des prédictions tous les jours, parfois tous les mois. Cela peut varier. Cela dépend aussi de la sensibilité du modèle à des données plus ou moins fraîches. Si le modèle n'est pertinent qu'avec les toutes dernières données, cela signifie qu'il faut l'entraîner souvent.
Notons d'ailleurs que l'automatisation de l'extraction des données que nous mettons en place peut servir à l'entraînement et/ou aux prédictions. Il faut parfois envisager deux automatisations d'extraction de données.
Il faut donc paramétrer de manière automatique cette récupération des données. Encore une fois, on trouve de nombreux outils pour réaliser cela de manière automatique et régulière.
Dernière partie, mais pas des moindres : le monitoring. De nombreux problèmes peuvent survenir entre la récupération des données, son traitement et sa mise à disposition. Il faut repérer les erreurs et s'y préparer. Souhaitons-nous relancer le traitement quand nous rencontrons un problème ? Ou mettre en place une solution B ? Quel système d'alerte mettre en place ? Qui alerter ?
Il existe des outils haut niveau qui permettent de construire ce qu'on appelle des pipelines. C'est là qu'on parle alors de pipelines de données. Elles permettent de réaliser toute la chaîne, souvent en faisant appel à des logiciels externes pour le traitement des données par exemple. Elles apportent du monitoring, des dashboards et un système d'alerting. À titre d'exemple, nous pouvons citer Airflow (qui fait bien plus que du traitement de données cela dit), Denodo (qui est uniquement orienté traitement des données) ou encore Azure Data Factory. Ce dernier aussi ne fait pas que du traitement de données, mais peut connecter tout un ensemble de choses. À part Denodo, les deux outils que je viens de citer, à savoir Airflow et Azure Data Factory servent surtout à réaliser des pipelines d'actions d'une manière générale. Ils font appel à d'autres services pour réaliser l'ingestion.
Cette mise en place étant déjà assez complexe, il ne s'agit pas se lancer dans la construction d'un datalake qui durerait par exemple une année et se dire qu'après cela on est fin prêt à faire de l'apprentissage automatique. La solution est toujours la même depuis que l'Agile a pris le pas dans notre monde : penser itératif. L'idée est de collecter la donnée dont on a besoin au moment où on en a besoin Il s'agit d'avancer un pas après l'autre, mais d'avancer continuellement.

Versionner ses données

Une fois que les données nous arrivent de manière régulière, nous allons pouvoir nous en servir pour réaliser l'entraînement et/ou des prédictions. Cependant, nous ne sommes pas à l'abri de rencontrer un problème dans ces deux autres étapes. Supposons que nous notions quelque chose d'inattendu dans les prédictions. Avec un peu d'investigation, nous voyons que ces dernières ont commencé à être différentes il y a deux jours. Il serait bon de pouvoir savoir l'état des données sur lesquelles les prédictions étaient faites avant ces deux jours et celui sur lequel elles sont réalisées aujourd'hui. Il s'agit de pouvoir reproduire la prédiction. C'est le fameux problème de reproductibilité dont on parle beaucoup en data science. Il faut être en mesure à un moment donné ou un autre de pouvoir reproduire ce qui s'est passé dans le temps. En data science pour y parvenir, il faut être en mesure de savoir dans quel état étaient les données. Ainsi, si dans le logiciel traditionnel, versionner le code

peut suffire, dans le machine learning, on cherche souvent à versionner le code, le modèle et la donnée.
Une manière relativement simple de faire du versionning de données est de partitionner ces données par rapport à leur fréquence de mise à jour. Supposons que nous recevions des nouvelles données tous les jours. Nous mettons chaque jour ces nouvelles données dans un dossier correspondant à la journée de réception. La donnée reçue le 10 janvier se retrouve dans le dossier du 10 janvier, celle du 11 janvier dans celui du 11 janvier, etc. Nous avons quelque chose comme suit :

```
- données
        - mois=01
                - jour = 10
                        - donnée1
                - jour = 11
                        - donnée2
```

Les dossiers une fois écrits sont immutables. De cette manière-là, nous savons avec quelles données nous avons réalisé un entraînement ou des prédictions. On peut aisément rejouer ce qui s'est passé il y a quelque temps. La reproductibilité est critique pour débuguer.
Cette technique a l'avantage de n'avoir besoin d'aucun outil externe. Elle ne nécessite qu'un peu de logique et de code. L'inconvénient est qu'il faut maintenir cela. Il faut aussi parfois débuguer ce système. Le meilleur code est toujours celui qu'on n'écrit pas.
C'est pourquoi il existe de plus en plus d'outils de versionning de données. Ceux-ci peuvent aussi être d'ailleurs très utiles pour l'exploration de donnée et/ou la modélisation. Il suffit d'activer la fonction et les données sont versionnées. On peut revenir à un état précédent ou effectuer une requête sur celui-ci. On peut demander à voir tout l'historique. De cette manière-là on sait ce qu'il s'est passé (insertion, suppression ou mise à jour). Avec l'aide d'un timestamp, on sait quand cela a eu lieu. On a aussi des informations sur l'utilisateur. C'est une mine de trésors quand un problème survient.
Certains outils permettent de versionner l'ensemble d'un datalake. Pour d'autres, il s'agit d'activer par dossier le versionning. Ces logiciels sont assez récents, mais il y a fort à parier qu'ils deviennent de plus en plus nombreux.
L'avantage d'un outil comme ces derniers dont nous venons de parler, c'est qu'il n'y a pas de code à maintenir ou à débuguer. C'est donc une option qui paraît préférable. Cependant, dans certains cas le producteur de la donnée ne la fournit que jour par jour, semaine par semaine. Dans ce cas-là, elle est de base versionnée par son arrivée. Cela n'empêche pas de combiner les deux systèmes. Pour donner un exemple de ces outils, DeltaLake est une surcouche à un datalake qui permet de voyager dans le temps. En allant lire un fichier, on peut demander son historique comme ceci :

```
fichier.history().show(truncate = False)
```

On obtiendra ce genre de résultats :

Version	Timestamp	UserName	Operation	Type	Cluster
0	2020-02-10 15:30:39	nastasia.saby	WRITE	ErrorIfExists	234
1	2020-02-10 16:45:54	nastasia.saby	WRITE	Append	5666

Dans ces colonnes (qui ne sont pas complètes pour des raisons de simplicité), on note plusieurs informations. Nous avons un numéro de version, un timestamp associé, l'opération qui a été effectuée sur le fichier, qui l'a effectué, dans quelles conditions et le cluster à l'origine de la requête.

On pourrait remonter dans le temps en chargeant les données à un instant T du passé comme ceci :

```
1 old_data = read.option("timestamp", " 2020-02-10 15:30:39").load(path)
```

Versionner ses données d'entraînement et de prédiction est un pas certain vers la pérennité d'une production. Nous l'avons dit, les données sont la nourriture de nos systèmes. Autant en prendre soin et les maîtriser.

Avoir confiance en ses données

On entend souvent des plaintes à propos de la qualité des données. C'est un élément important du machine learning. Une partie du travail d'un data scientiste consiste à analyser les données, identifier les manques et failles, les combler quand c'est possible et savoir renoncer au projet quand un vrai problème est identifié.
Quand on est en production, on a alors des données qui nous arrivent de manière régulière. Nous mettons un système de versioning en place. Cependant, ce n'est pas suffisant. Nous savons que les données peuvent évoluer. La seule chose qui ne change pas, c'est le changement. Comment les données vont-elles diverger de ce dont nous avions l'habitude de traiter ?
Même si cela est difficile à dire, il existe des solutions. Il est important de se donner les moyens de s'en rendre compte. Ainsi, on met en place une forme de monitoring des données. Dans ce cas-là, on peut aussi parler d'observabilité. On observe les données qu'on reçoit en entrée. On bâtit des statistiques descriptives à partir de celles-ci. On crée de la donnée à partir de ces données afin de mieux maîtriser l'ensemble du processus. On lance aussi des alertes quand la donnée ne ressemble plus à ce dont nous avions l'habitude. Cela peut ensuite éviter des problèmes en aval.
Par exemple, supposons que nous soyons dans une phase d'exploration/modélisation. On a des données provenant d'une table quelconque. Cela constitue nos features. On réalise que dans 2% des cas, un champ est manquant. En supprimant les lignes concernées, on remarque qu'on a quand même un modèle efficace. On choisit donc de les filtrer pour construire notre modèle. Aura-t-on toujours 2% de lignes problématiques ? Comment en être sûrs ? Si pour une raison quelconque, on voit ce chiffre passer à 20%, ne va-t-on pas avoir un modèle moins performant ? Bien sûr, le meilleur moyen de le savoir est de tester et de vérifier les performances du modèle à chaque entraînement.
On peut aussi mettre en place un système qui regarde le contenu de la donnée. On logue ainsi le fait qu'on passe de 2% de lignes problématiques à 20%. Cela donne des informations précieuses pour débuguer et comprendre ce qui se passe dans le système.

On voit alors l'intérêt d'évaluer les données d'entrée. Pour réaliser cela, nous nous aidons de librairies et/ou frameworks pour dresser des statistiques descriptives de nos données. Nous évaluons la distribution de ce que nous recevons. Par exemple, nous cherchons à prédire si une proposition commerciale sera acceptée. Jusqu'ici pour notre entraînement, nous avions en entrée 50% de propositions commerciales acceptées et 50% non acceptées. Il faut être à même de noter quand une différence trop importante dans les données apparaît.
Enfin, un système d'alerte s'avère efficace afin de repérer quand la donnée est jugée trop divergente. Cela signifie qu'il faut s'entendre sur ce qu'on accepte de la donnée ou pas.
Le monitoring et l'observabilité sont des concepts très utilisés dans la programmation classique. Dans la data science, ils peuvent aussi s'appliquer à la donnée même, la source de tous nos systèmes.
Mettre en production un modèle, c'est aussi être à même de le maintenir et de le débuguer quand un problème survient. C'est à cet objectif que répondent le monitoring et l'observabilité.

Nouveaux problèmes

Une fois que l'on comprend qu'il faut mettre en place un socle solide de données, on peut se retrouver face à de nouveaux problèmes.

La jungle dans les pipelines de données

Ce terme de « jungle » est assez fort mais décrit assez bien un concept inhérent aux pipelines de données. On ne mesure pas toujours tout de suite les problèmes que peuvent révéler les pipelines de données. En effet, au début, nous avons souvent peu de données et un système relativement petit. Au fur et à mesure, on ajoute des éléments. On peut se retrouver face à une « jungle » difficile à maintenir.
C'est un « code smell » évoqué dans le papier *Hidden Technical Debt in Machine Learning Systems (Dette technique cachée dans les systèmes de machine learning)* de D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, ToddPhilli, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo et Dan Dennison. Le concept de « smell » a été créé par Martin Fowler et Kent Beck. Ces derniers sont deux développeurs signataires du manifeste Agile. Kent Beck est à l'origine de l'Extrême Programming, une méthode Agile qui met l'accent notamment sur les pratiques de développement. Il est aussi l'auteur du Test Driven Development et a rédigé un livre sur le sujet. C'est un concept qui prône l'écriture des tests avant le code afin de se laisser guider par eux. Quant à Martin Fowler, il s'intéresse à des sujets comme la programmation orientée objet, le réusinage (refactoring) ou encore les patrons de conception (design patterns). Dans son livre *Refactoring, Improving the Design of Existing Code (Réusinage, améliorer le design du code existant)*, il dresse une liste de « smells » et donne des méthodes pour y remédier. Il s'agit d'identifier des structures dans le code qui mèneront à des soucis de maintenance. Cela peut être des difficultés à debugger ou faire évoluer le programme. Le concept lui vient d'une conversation avec Kent Beck qu'il relate dans le livre. Ils cherchaient à nommer les structures de code qui sont problématiques. Il ne voulait pas utiliser de concept esthétique en parlant de code beau ou laid. L'idée d'odeur leur est venue pour décrire le fait qu'il y a des éléments dans un programme qui ne sentent pas bon et qui présagent de mauvaises choses. C'est encore une image, mais elle est peut-être plus parlante que celles qui sont dans le

registre de la beauté. On peut bien entendu remettre en question certains codes jugés comme des « smells ». Puis certains éléments sont volontairement flous. Une classe trop longue est un « smell ». Mais qu'est-ce qu'une classe trop longue concrètement ? Comment mesure-t-on cela ? Est-ce le nombre de méthodes ou de lignes qui comptent ? Martin Fowler nous incite à découvrir par nous-mêmes ce genre de choses en fonction du projet et du contexte dans lesquels nous évoluons. L'intérêt du travail de Martin Fowler, c'est qu'il a clarifié la notion de « smell » et tenté une systématisation de celle-ci en établissant une liste et des remèdes.

La data science est un champ relativement nouveau du développement. Naturellement, elle apporte alors de nouveaux « smells ». La particularité du machine learning, c'est que ces « smells » ne sont pas forcément directement liés au code. Les « smells » qu'on peut trouver en data science sont un des sujets que traitent les auteurs de l'article *Hidden Technical Debt in Machine Learning*. Ils évoquent notamment les jungles de pipeline. Afin d'éviter cela, ils incitent à penser à la mise en production dès le début et à mélanger dans une même équipe des personnes plus orientées recherche et d'autres plus orientées ingénierie.
Notons que les pipelines peuvent comme un code être refactorés (réusinés) tout au long de leur vie afin d'être améliorés. On peut aussi mettre en place des tests automatisés sur les pipelines. C'est cependant très coûteux. Une balance entre l'effort et les bienfaits est à trouver.

La maintenance de multitude de dépendances

Les données sont la matière première de nos algorithmes. Elles représentent aussi nos premières et peut-être nos plus lourdes dépendances au monde extérieur. Il est parfois difficile de les maintenir. Automatiser leur suivi, alerter quand il y a des problèmes et avoir des solutions de repli (systèmes de fallback) aide à y voir plus clair.
Les auteurs de *Hidden Technical Debt in Machine Learning* notent aussi que les dépendances de données peu utilisées peuvent être problématiques. Il s'agit d'éléments qui apportent peu de plus-value mais nécessitent beaucoup de maintenance. Il est important d'évaluer alors ce qui peut être omis dans le modèle. Parfois, certaines features ne sont pas aussi nécessaires que ce qu'on imaginait. C'est une manière de réduire ses dépendances.

Pourquoi doit-on monitorer des données qui ne changent pas ?

On peut avoir l'impression que les données n'évoluent pas. La meilleure manière d'être sûr de ce présupposé est de mettre en place du monitoring. Des données inattendues peuvent ruiner le meilleur des modèles. Il s'agit de se prémunir de ce genre de situations périlleuses.

Peut-on allier versionning automatique et personnalisé ?

Les deux formes de monitoring ne sont pas incompatibles. Versionner avec son propre système permet d'avoir exactement le versioning que l'on souhaite. Je conseillerais de garder dans tous les cas un versionning automatique comme filet de sécurité.

Pour conclure, les données représentent ce sur quoi reposent les systèmes de machine learning. C'en est la matière première. Il faut des mises à jour de données sereines. Celles-ci doivent correspondre à

la fréquence nécessaire. Les données alimentent l'entraînement et peuvent alimenter les prédictions. Il s'agit de mettre en place des pipelines de données maîtrisables. Cette maîtrise passe par du monitoring et du réusinage. Versionner les données comme le code sur lequel le modèle est construit est un gage de production sereine.

Face à un projet, je résumerais les éléments à suivre ainsi :

- Identifier les données nécessaires (source et chemin de transformation)
- Identifier pourquoi ces données sont nécessaires (entraînement et/ou prédiction)
- Identifier la fréquence de mise à jour
- Monitorer, loguer les erreurs et alerter quand il y a un problème ou une divergence
- Observer les données pour mieux en évaluer l'évolution

Cela sera traité plus en profondeur, mais notons dès à présent qu'il s'agit de tester la qualité de ses données et être sûrs de pouvoir s'appuyer dessus en évaluant son comportement général. C'est d'elles que part tout notre système. Elles sont cruciales et représentent la source de vérité. C'est d'ailleurs la raison pour laquelle, il ne faut travailler qu'avec des données de production. Les données factices, reproduites ou réalisées à l'aide de tests de développeurs n'ont aucune valeur en data science. Dans ce domaine, les données sont à la data science ce que le code est à la programmation classique. On ne traite pas avec du faux code. En machine learning, on ne traite pas avec de fausses données.

Une fois le socle des données établi, il reste l'automatisation de l'entraînement et de l'inférence.

Automatiser l'entraînement

Dans ses débuts en phase d'exploration et de modélisation, l'entraînement est souvent réalisé manuellement. Une des étapes de la mise en production est donc d'automatiser cela. Il s'agit de pérenniser le système. Cela passe par plusieurs éléments que nous allons détailler.

Versionner le code et le modèle

Ainsi, il y a le code et le modèle qu'on veut versionner pour pouvoir remonter dans le temps, reproduire un bug ou une prédiction étrange.

Comme dit précédemment, dans la programmation classique, on cherche à versionner le code. En machine learning, on veut aussi versionner le modèle et les données. Nous avons déjà parlé de versioning de données au dernier chapitre. Parlons un peu cette fois-ci de versionner le code et le modèle.
Au fond, qu'est-ce que cela veut dire « versionner » ? Cela signifie garder des versions (on peut aussi parler d'instantanés) de quelque chose. Par exemple, quand on rédige un document, on le modifie à différents moments. L'idée est de garder ces modifications et de pouvoir y revenir. Assez souvent, on fait cela de manière naturelle sans outil. On trouve parfois dans un même dossier : doc_version$_1$.txt, doc_version$_2$.txt, etc. Il y a généralement un besoin assez naturel de garder les

différentes étapes de la création d'un document. Il en est de même pour les modèles et le code. On essaye généralement différentes choses et on apprécie de garder certains éléments. Le problème, c'est que quand on le fait manuellement, on prend le risque d'oublier de versionner certaines étapes ou d'avoir un dossier rempli de différents documents auxquels on ne comprend rien avec des choses comme doc_version_3_versionB_(1).txt par exemple.

Pourquoi versionner ?

Finalement, nous voyons assez bien les raisons à cela. Elles sont multiples. Il s'agit de comprendre les étapes qui nous ont amenés à l'état où nous sommes, débuguer s'il y a un problème ou revenir à une version antérieure.

Pourquoi versionner automatiquement ?

L'erreur est humaine. Elle l'est aussi dans le versionning. C'est pour cela qu'on cherche à l'automatiser. Cela signifie qu'il faut maîtriser un outil de versioning. Ce n'est pas toujours aisé au début. La charge cognitive d'apprendre un outil de versioning est forte au début quand on découvre le système automatique, mais le devient de moins en moins avec le temps.
Entrer dans de l'automatisation, c'est aussi se donner des moyens de déployer plus aisément. En effet, pour générer automatiquement un nouveau changement, une solution est d'avoir un système de versioning. Chaque nouvelle version déploie le système de manière automatique. Je reviendrai là-dessus et nous verrons de quelle manière cela simplifie le travail.

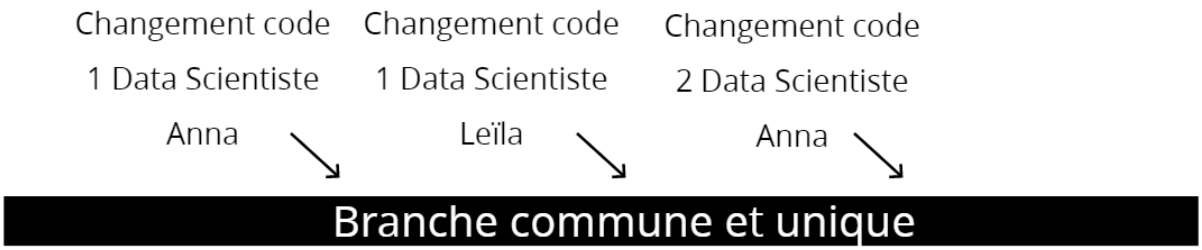
Une des raisons de versionner automatiquement, c'est aussi pour pouvoir partager. Il est vrai qu'on peut versionner uniquement sur son poste. Cependant, tous les outils automatiques de versionning permettent de mettre en commun le code, le modèle ou les données. De cette manière-là, toutes les personnes de l'équipe savent ce qui se passe dans tel ou tel projet. Cela permet aussi de collaborer. On sait ce qui est développé par son collègue. On peut lui proposer de nouvelles versions.

Versionner automatiquement le code

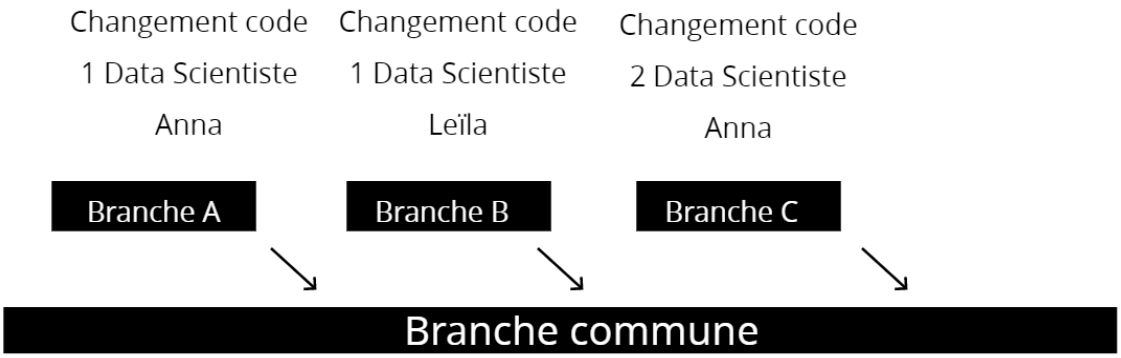
Les développeurs des logiciels plus classiques ont la plupart du temps recours au versioning automatique. Dans le monde de la data science, c'est moins le cas notamment dans les phases d'exploration et de modeling. Pourtant, tester un nouvel algorithme ou un nouvel hyperparamètre revient finalement à créer une nouvelle version du modèle. On ne la gardera peut-être pas. Mais on garde justement comme trace le fait que ce code a été expérimenté et qu'on n'en veut pas.
Il existe des outils qui versionnent automatiquement le code sans avoir à y penser. Certaines plateformes de notebooks par exemple enregistrent pendant que l'on y travaille les différentes évolutions du code. C'est le cas de beaucoup d'environnements de développement aussi. Ces outils sont confortables. Ils sont faciles d'usage. Il n'est pas besoin de penser à sauvegarder des instantanés de son code. La courbe d'apprentissage est facile. Cependant, il n'apporte pas toutes les fonctionnalités d'un outil plus classique de versionning tels que Git ou Mercurial par exemple. Ces deux derniers logiciels sont en effet plus difficiles à prendre en main. Ils sont aussi plus flexibles.

On choisit quel nom donner à cette sauvegarde. C'est plus pratique que d'avoir à rechercher un instantané par date et heure. De plus, on peut choisir quelle sauvegarde on veut faire et ce qu'on veut y inclure.
Ces outils permettent de partager le code. On peut versionner en local sur sa machine, mais aussi partager cela dans ce qu'on appelle un dépôt (repository) partagé. De cette manière-là, les personnes de l'équipe voient les évolutions du code. On peut soumettre une version en approbation à ses collègues. Ils peuvent la commenter, la refuser et l'accepter.
Enfin, ces logiciels sont pratiques pour évaluer quand une nouvelle version de code a été partagée. On peut y connecter un système de déploiement automatique. Celui-ci met en place les machines de production et met à jour le code sur celles-ci. Chaque nouvelle version est alors déployée automatiquement en production. On peut bien entendu choisir de n'envoyer que certaines versions en production. On peut avoir différents codes en parallèle et n'en garder qu'un comme la version finale voulue pour la production. Dans Git, on réalise souvent cela avec le système de branches. Il y a une branche « main ». C'est celle qui va en production. À partir de là, on peut créer de nouvelles branches. On y réalise du code. C'est uniquement après soumission et approbation que ce code est fusionné dans « main » et que cela lance un déploiement automatique en production.
Versionner son code peut paraître abscons la première fois qu'on rentre dedans. S'il est vrai qu'il est difficile de tout comprendre tout de suite, on peut généralement s'en sortir avec deux ou trois concepts. Il est à noter que versionner avec un outil comme Git ou Mercurial ne veut pas dire qu'on ne peut pas continuer le versionning automatique présent dans son notebook ou environnement de développement. On a alors différentes formes de copies de ce qu'on fait et plusieurs manières d'y revenir s'il y a un problème.

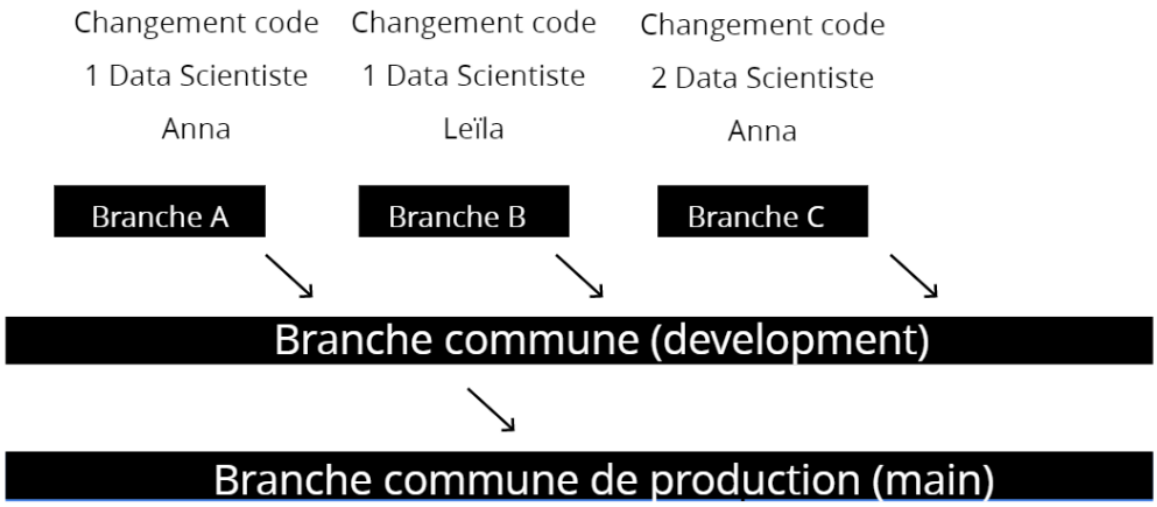
Versionner son code signifie aussi avoir un « workflow ». Il y a plusieurs solutions. On peut faire du « single branch development ». Cela signifie qu'il n'y a qu'une branche qui représente à la fois la production et le développement en cours. Certains pourraient trouver cette technique trop aventureuse. Pour ma part, je vois surtout un avantage à celle-ci : elle permet le développement continu de manière assez poussée. Cela signifie que toute modification de code dans l'espace commun (« main ») part directement en production. J'ai plusieurs fois testé cette technique et je sais qu'elle peut très bien marcher dans certaines équipes. Il s'agit de s'adapter au cas par cas. En data science, nous sommes parfois seuls sur un projet. On peut le regretter. En effet, on ne peut pas partager et s'enrichir des autres. Mais dans un cas comme celui-ci, le « single branch development » peut être assez adéquat. Il n'est pas besoin de se rajouter de la complexité pour rien. Ce workflow est aussi adapté quand les équipes travaillent à plusieurs sur le code. La vérification est alors continue. Elle se fait au moment même de l'écriture du code. On n'a donc pas nécessairement besoin d'une autre phase de validation. Dans le schéma, ci-dessous, on voit qu'il n'y a qu'une branche de code et que tous les data scientistes la font évoluer au fur et à mesure du temps.



Un autre workflow, c'est celui nommé « trunk based development ». Celui-ci gère deux versions du code : la production (la branche commune) et des branches sur lesquelles se trouvent des fonctionnalités en attente. L'idée est de rassembler ces branches fonctionnelles sur la branche commune. Ce workflow permet aussi d'aller rapidement en production. Il est assez adapté quand on travaille à plusieurs sur un projet et qu'on veut faire une validation en différé avant de fusionner sa branche (celle où on a ajouté une nouvelle fonctionnalité) et la branche commune.
Dans le schéma ci-dessous, on voit qu'il y a toujours une branche tampon avant d'intégrer du code à la branche commune :



Il existe aussi le workflow « gitFlow ». Dans celui-ci, pour résumer, nous avons une branche « main ». Il représente la production. Nous avons aussi une branche « development » qui représente l'état de travail courant. Enfin, on peut créer de nouvelles branches de cette dernière afin de développer les fonctionnalités. Il y a aussi tout un système de branches pour les corrections et les releases.
Dans le schéma ci-dessous, on voit l'idée générale. Il y a une couche supplémentaire. Il y a deux branches : l'une pour la mise en commun, l'autre pour partir en production. Je vous épargne les détails. Il y a en réalité bien d'autres branches. Nous avons ici davantage à faire à un « gitFlow » simplifié, ce processus étant en réalité plus complexe.



Ce workflow devient souvent un frein pour la mise en production. Il est un frein au fait de déployer souvent, voire continuellement. L'auteur même de ce workflow ne le conseille pas dans le cas où on souhaite partir en production souvent et rapidement. Il est de plus complexe. Dans des équipes peu habituées à versionner, son usage peut être rédhibitoire. Ce workflow n'a en fait tout simplement pas été pensé pour notre cas d'usage. Cependant, il reste fort utilisé. Je me devais donc de le citer. Il n'a pas beaucoup d'intérêt dans notre contexte, mais il aurait été injuste de ma part de ne pas l'évoquer. L'un des objectifs de ce guide est de vous donner des conseils sans dogmatisme. Vous entendrez probablement parler un jour ou l'autre de « gitFlow ». Je tiens à vous mettre en garde contre ses inconvénients et à vous le déconseiller en rappelant que l'auteur lui-même le préconise pour un cas qui n'est pas le nôtre, à savoir du déploiement non continu pour des logiciels lourds.

Versionner automatiquement le modèle

Versionner le code peut être un concept que tous les data scientistes ne maitrisent pas. Cependant, c'est quelque chose d'assez classique dans le monde du développement back-end ou front-end par exemple. Il n'en est pas de même pour le versioning de modèles qui est une spécificité à la data science.
On pourrait être d'abord surpris que les data scientistes ne versionnent pas tous leur code. Mais c'est peut-être aussi parce que le versioning de code est un type de versioning parmi d'autres dans le monde de la data science. Nous avons un peu parlé du versioning de données. Nous avons vu que nous pouvons réaliser cela nous-mêmes avec un système de partition paramétré par nos soins. Nous avons aussi évoqué le fait qu'il existe des outils automatiques pour réaliser cela.

Évoquons un peu plus en détail à présent le versioning de modèle. Dans la phase de modélisation, on essaye plusieurs algorithmes : régression logistique, random forest, réseau de neurones, etc. On essaye plusieurs hyperparamètres : nombre d'arbres pour un random forest, nombre de couches pour un réseau de neurones, fonction d'activation, learning rate, etc. Généralement, on apprécie de garder

la trace de tout cela. Ces différentes tentatives sont en fait des expérimentations. Chaque expérience mérite d'être retenue. Même les moins performantes sont intéressantes. Il s'agit de savoir qu'elles ont existé pour ne pas les reproduire de nouveau. Comme nous le voyons, garder les algorithmes et hyperparamètres des expériences qu'on réalise apporte des bienfaits dans la phase de modélisation. Mais c'est aussi très utile en phase de production. On pourrait croire qu'un modèle une fois mis en production ne va plus évoluer. C'est assez rarement le cas. Même si on ne change pas les hyperparamètres de notre modèle tous les jours, nous savons que les données vont évoluer et les performances du modèle décroître. Nous avons des chances d'être amenés à repasser dessus et le faire évoluer. Puisque celui-ci va être modifié dans le temps, il vaut mieux savoir avec quel modèle une prédiction a été réalisée. Pour cela, autant versionner ses modèles.
On peut aussi enregistrer les résultats des performances du modèle lors de sa phase d'entraînement. De cette manière-là, on voit aussi les résultats fluctuer. On choisit de sauvegarder les métriques pertinentes : accuracy, precision, recall, F1 score, AUC, RMSE, etc. Ces éléments-là servent d'ailleurs au monitoring. On observe les résultats de notre modèle dans le temps. Il est à noter qu'on peut aussi choisir de construire une métrique business qui serait complètement personnalisée et suivre celle-ci. Pour ce faire, on logue soi-même les traces qu'on veut garder de l'expérimentation. Mais il existe aujourd'hui des outils open source qui permettent de faire cela. C'est le cas par exemple de MLFlow. Ce dernier a plusieurs composantes. L'une d'entre elles est d'enregistrer (tracker) les éléments qui composent une expérience. On choisit de loguer l'algorithme qu'on utilise, les hyperparamètres, les métriques ou même des commentaires. Ces métriques peuvent être diverses. Elles dépendent aussi du type de modèle : précision, accuracy, AUC. De plus, nous pouvons loguer n'importe quelle autre métrique personnalisée.

Le versioning de modèle a aussi un intérêt quand on fait de l'auto-ml. Le concept renvoie au fait de laisser plus ou moins en partie un outil choisir le meilleur modèle à appliquer sur les données. Cette automatisation peut s'appliquer sur l'ensemble du modèle et décider si GradientBoosted Tree ou une régression linéaire est plus performant. Elle peut aussi porter uniquement sur les hyperparamètres. En versionnant ces différents tests, on a une vue plus complète de ce que le système a essayé de faire. Ces outils peuvent présenter une méthodologie différente. Certains vont tester un ensemble de modèles possibles. D'autres sont basés sur des algorithmes génétiques. Cela signifie qu'ils agissent par itération. La première étape est de générer plusieurs versions de modèles. À partir de là commence le processus itératif. Ils effectuent de légères modifications sur les modèles générés, évaluent la performance de ceux-ci et ne gardent que les meilleurs. Ce sont ces derniers qui servent à l'itération d'après. C'est un peu le principe de la théorie darwinienne. Les plus forts survivent et engendrent des enfants auxquels ils donnent leur force. Les meilleurs parmi ces derniers se reproduisent en transmettant leurs gènes à leur progéniture. Cela continue ainsi de suite. Dans tous les cas, avec l'auto-ml, on génère différents modèles ou différentes formes de modèles avec différents hyperparamètres. Cela constitue un lot d'expérimentations qu'on versionne. On ira ensuite chercher dans ce lot d'expériences laquelle est la plus intéressante et la choisir pour réaliser les prédictions.
Enfin, ces outils permettent de versionner le modèle même sous une forme sérialisée. On a alors en main tout pour comprendre notre modèle à un instant ou à un autre.
Versionner son modèle apporte plus d'un avantage. Cela permet de garder des traces des expériences. On sait ce qui a plus ou moins bien marché. On sait où on en est dans sa recherche ou debug. Cela

permet de garder les réentraînements en production. La reproductibilité est alors plus aisée. On sait alors ce qui s'est passé et à quel moment. C'est aussi un premier pas pour monitorer les performances de son modèle. Si on ne logue rien, on ne peut pas évaluer cela. Versionner un modèle est donc utile à plus d'un égard. Il s'agit de combiner cela au versioning classique de code et au versioning moins classique de données. Les trois réunis permettent d'obtenir la reproductibilité.

Automatiser les tests

Ensuite, il faut automatiser les tests. D'abord quand on parle de tests, il s'agit de clarifier ce dont il s'agit ? Dans le monde du web, il n'y a peut-être que le code à tester. On veut vérifier que ce que l'application fait est bien conforme à ce qu'on souhaite. Il y a différentes manières de le faire bien entendu. Mais la cible c'est le code. Il est assez compréhensible que toute l'attention soit portée là. En effet, il s'agit souvent d'un méandre de fonctionnalités que l'utilisateur peut décider d'activer à sa guise.
En data science, le code est plus simple. On ne cherche pas à reproduire des actions utilisateurs, mais davantage à reproduire un processus cognitif. Ainsi, en data science, quand on parle de tests, on peut faire référence à d'autres choses. Nous l'avons dit, certains envisagent la data science comme un domaine étant pour partie de l'ingénierie pour partie de la recherche. Et c'est avec ce dernier élément qu'on donne son côté scientifique à la data science. Or, une expérimentation a besoin de validation scientifique. C'est le but des métriques et de l'évaluation du modèle. Ces éléments servent à évaluer les performances du système. On pourrait considérer cela comme du test. Pour moi, cela relève davantage du monitoring. En tous cas, c'est dans cette section qu'il a été classé. Ce chapitre traite du fait de réaliser des tests sur le code. Il me semble important de le préciser parce que la question est souvent posée. C'est tout à fait légitime. Les données et le code sont très importants en data science. On a d'abord envie de tester ces éléments avant de tester son code qui est souvent plus simple.
Ainsi au début, quand on crée du code, qu'on le modifie en ajoutant ou supprimant un élément, assez naturellement, on effectue des tests manuels pour vérifier qu'aucune régression n'a été apportée. On ne s'en rend pas toujours compte, mais c'est un processus long et fastidieux. On ne réalise pas toujours qu'il y a des manières d'automatiser cela. Il existe de nombreux outils dans tous les langages du monde. On peut d'ailleurs aussi créer soi-même assez aisément des moyens de réaliser ses tests.

Quelle forme prend un test de code ?

Afin de comprendre davantage de quoi il s'agit, je vous propose un peu de pseudo code. Supposons une fonction de division comme suit :

```
1 def divise(premierNombre, deuxièmeNombre) :
2         return diviser(premierNombre).par(deuxièmeNombre)
```

Afin de tester cette division, on exécute la fonction et vérifie le résultat comme suit :

```
resultat = divise(8, 4)
try { resultat == 2
} catch {
        raise Exception(« Le test échoue »)
}
```

Dans ce cas, si le résultat n'est pas égal à deux, cela lève une exception. On peut utiliser autre chose qu'une exception. On a la possibilité de prendre une fonction d'une librairie ou d'utiliser la fonction « assert » assez courante dans beaucoup de langages. Le mieux est d'encapsuler cela dans une fonction qui sera exécutée par un outil de test automatisé.

```
def testQue8DiviséPar4EstEgalA2() :
        resultat = divise(8, 4)
        assert(resultat == 2)
```

Nous venons de faire ce qu'on appelle un test unitaire. Les définitions sur ce concept varient. Une définition simple serait de dire qu'il s'agit de tester un élément de notre système de manière isolée. Il n'y a pas d'appel à une base de données, un élément externe ni même une autre classe dans ce cas-là.
Il s'agit aussi d'un test d'acceptance. Nous avons uniquement testé le cas passant, c'est-à-dire le cas que nous espérons le plus fréquent et qui ne renvoie pas d'erreur. Nous pourrions aller plus loin et tester les cas aux limites. En effet, il est intéressant de tester aussi les cas plus rares qui peuvent renvoyer une erreur. Cela nous permet de connaître notre résilience face aux erreurs. Ainsi, on écrit un second test comme suit :

```
def testQue8DiviséPar0RenvoieUneErreur() :
        resultat = divise(8, 0)
        assert(resultat == « Erreur : la division par zéro n'est pas possible »)
```

On ordonne ces éléments dans une suite de tests dans une même classe :

```
class DivisionTest() :
        def testQue8DiviséPar4EstEgalA2() :
        resultat = divise(8, 4)
        assert(resultat == 2)

        def testQue8DiviséPar0RenvoieUneErreur() :
        resultat = divise(8, 0)
        assert(resultat == « Erreur : la division par zéro n'est pas possible »)
```

L'idée est d'avoir plusieurs classes correspondant aux différents éléments de notre système. Cela permet alors de tester l'ensemble de notre code et d'être sûr quand on exécute tous ces tests qu'il

n'y a pas de régression. Généralement, on a un voyant vert qui s'allume quand tous les tests passent et un voyant rouge quand un test échoue.

L'exemple de la division d'un nombre par un autre nombre n'est peut-être pas le plus convaincant. On peut imaginer de tester une transformation de données. On pourrait avoir une fonction comme suit :

```
def supprimeLesValeursExtremes(données) :
        valeursExtremes = calculeLesValeursExtremes(données)
        return resultat.supprime(valeursExtremes)
```

On veut alors tester une entrée et une sortie. L'entrée est la suivante :

Champ
100
101
10
1400
103

On veut la sortie suivante :

Champ
100
101
103

On peut alors réaliser un test comme suit :

```
def testQueLesValeursExtremesSontBienSupprimées() :
        entrée = créerDonnées(100, 101, 10, 1400, 103)
        sortieAttendue = créerDonnées(100, 101, 103)
        sortie =  supprimeLesValeursExtremes(entrée)

        assert(sortie == sortieAttendue)
```

Avec ce genre de tests, on voit davantage l'utilité des tests automatiques. Nous faisons très souvent ce genre de manipulation de données en data science. Nous réalisons toujours des tests. Seulement, nous les réalisons parfois à la main. Avec des tests automatisés, cela n'est plus nécessaire. Il s'agit en effet d'écrire les tests, mais une fois que c'est fait, il suffit de lancer l'ensemble des tests pour savoir s'il y a un problème. C'est un gain de temps certain. Il s'agit de plus d'ajouter le lancement de ces tests de manière automatique dans le processus. Ainsi, dès qu'on envoie le code sur le serveur de versioning commun, les tests se lancent pour vérifier qu'il n'y a pas d'erreur.

Enfin en tant que data scientistes, on s'attend à pouvoir tester le code qui crée le modèle. Ce n'est pas toujours évident car celui-ci est stochastique par définition. Il existe dans beaucoup de librairies

des manières de pallier ce côté aléatoire. L'argument « seed » est souvent utilisé par exemple pour y remédier.

La pyramide des tests

En réalité, il existe plusieurs types de tests. Souvent, on parle surtout de tests unitaires et de tests d'intégration. Tout le monde n'établit pas exactement les mêmes séparations, mais on peut envisager les choses ainsi. Les tests unitaires correspondent davantage au fait de tester des petites parties de son application de manière unitaire. Parfois, on dit qu'un test unitaire teste une méthode. C'est rarement vrai. Il y a un article à ce sujet que j'aime beaucoup qui s'appelle *Faire des tests unitaires est impossible*. Il a été écrit par Xavier Detant. Quand on réalise un test, on teste au moins deux méthodes : celle qu'on veut vraiment tester et le constructeur. Peut-être qu'une meilleure définition d'un test unitaire serait comme le donne l'article celle-ci : « Un test est unitaire si le code sous test ne dépend que du langage (pas de framework, pas de base de donnée, …) ».
Il m'apparaissait important de souligner que les définitions ne sont pas aussi claires que ce qui peut apparaître quand on parle de tests.
Les tests d'intégration testent à un plus haut niveau. Tester son pipeline de données dans son ensemble est davantage un test d'intégration qu'un test unitaire.

Il existe ensuite d'autres formes de tests : les tests de performance ou le profiling pour savoir ce que fait exactement l'application à chaque appel. Tous ces tests testent le code d'une manière différente. Ils n'apportent pas tous le même gain. Les tests bas niveau, ceux dit unitaires, sont sans doute les plus simples et sont rapides à rédiger. Ce sont aussi ceux qui ont le plus de valeur.

Des tests rapides

L'idée est de jouer les tests unitaires plusieurs fois par heure quand on code et de jouer les tests d'intégration au moins une fois par jour. Pour ce faire, il s'agit donc d'avoir des tests rapides qu'on a envie de lancer souvent. Cela nous permet d'avoir un retour rapide. En data science, un test peut être long si on ne fait pas attention. Cela peut être dû à un trop fort volume de données, un algorithme itératif fort gourmand ou encore au fait d'utiliser un framework optimisé pour le traitement distribué, mais pas pour le calcul en local. Pour la volumétrie des données, il ne sert à rien de tester sur beaucoup de données. Il suffit de tester sur quelques-unes bien particulières et/ou représentatives de ce que l'on a en production. Limiter le nombre de données permet d'accélérer les algorithmes gourmands. Ceux-ci peuvent pour les tests être lancés avec des paramètres différents de la production. On peut envisager moins d'époques ou poser des limites plus importantes à la convergence par exemple. Il est à noter qu'on ne cherche pas à évaluer la performance globale du modèle. Ce dernier point relève davantage du monitoring. Ce qu'on cherche à faire dans notre cas, c'est à tester le code. On veut éviter les régressions. Enfin pour le dernier point lié au choix du framework, on sait que certains frameworks ont des performances assez mauvaises pour les tests. Je pense notamment à Spark. Il y a plusieurs manières de contourner cela. D'une manière générale, il s'agit d'en avoir conscience, d'y veiller et d'optimiser ses tests dans cet objectif. Par exemple, avec Spark, on sait que certaines opérations sont plus coûteuses que d'autres. On peut préférer les moins

gourmandes dans les tests. Il existe aussi d'ailleurs des librairies spécialisées dans ses frameworks. Ils permettent d'abstraire ce travail et sont une solution.

Des tests indépendants

Les tests sont finalement du code qu'il faut maintenir. Pour avoir des tests robustes qui nécessitent peu d'entretien, il vaut mieux que ceux-ci soient indépendants. Cela signifie qu'on doit pouvoir lancer chaque test séparément. De cette manière, on peut exécuter les tests un à un. Quand il y a un problème, l'investigation est plus simple. Qu'un test échoue ou réussisse ne doit pas avoir de conséquence sur les précédents ou suivants.

Des tests déterministes

Pour une maintenance plus aisée des tests, une chose importante est d'avoir des tests qui s'effectuent de manière déterministe. Ils ne doivent pas être soumis à des aléas. Un test doit donner toujours le même résultat qu'il ait été exécuté le lundi, le mardi, etc. En data science où les statistiques sont reines, ce n'est pas forcément toujours évident. C'est pourquoi il y a des aides fournies par de grandes librairies comme Scikit-Learn.

Des tests pour tous les environnements

Les tests sont généralement lancés sur notre machine. Mais ils sont aussi souvent lancés sur les machines de nos collègues data scientistes ou même sur un serveur d'intégration. Le but de celui-ci est de centraliser le code de tous. Nous voulons vérifier qu'il n'y a pas d'erreur à chaque nouvelle modification de code. C'est pourquoi nous lançons les tests à chaque fois. Les tests ont donc la contrainte supplémentaire de devoir pouvoir se lancer sur différentes machines. Par exemple, une date ne doit pas être soumise au fuseau horaire de de la machine sur laquelle elle s'affiche.

Toutes ces contraintes peuvent paraître fastidieuses. Je reconnais que cela peut paraître effrayant de premier abord. Pourtant, les tests s'ils ne sauvent pas de vies, économisent du temps et promettent de la pérennité dans la mise en production. Les prérequis supplémentaires dont nous venons de parler, à savoir avoir des tests rapides, indépendants, stables et exécutables sur tous les environnements qu'ils desservent, guident pour rédiger une suite de tests efficace qui nous évitera du tracas.
Une fois qu'on prend l'habitude de tout cela, cela devient un automatisme sans effort.

Test first

L'idée avec les tests automatisés est qu'ils couvrent l'ensemble du code. Cela n'est pas évident à réaliser après coup. De plus, devoir rédiger tous les tests à la fin rend le travail long et pénible. On ne sait plus ce qu'on cherchait à tester. Il faut revenir sur nos intentions de départ. Cela demande un effort certain. C'est l'une des raisons qui fait qu'on peut avoir du mal à trouver séduisante l'idée de tester son code. J'entends souvent des complaintes à ce propos. L'une des manières d'y remédier, c'est de tester dès le début, voire avant d'avoir écrit son code. On peut faire ce qu'on appelle du Test

first. C'est un concept qui vient du Test Driven Development. Ce dernier provient de l'auteur Kent Beck et de la méthode Agile Extreme Programming. L'idée est de réaliser ses tests avant d'écrire son code et de se laisser conduire par eux. Le design du code est censé émerger des tests. Dans les faits, c'est un peu plus compliqué que cela.
Le Test first consiste à écrire les tests avant. C'est ce qui est expliqué dans le livre *TDD par l'exemple* de Kent Beck. Une autre référence que je trouve particulièrement intéressante pour expliquer ce concept, c'est le livre *Test Driven Development : A practical Guide* de David Astels. Nous allons résumer l'idée. Prenons cette fois-ci une fonction d'addition simple de deux nombres. Cette fois-ci, nous allons écrire le code après avoir écrit le test. Le premier test est le suivant :

```
def testQue1Plus2Egal3() :
        résultat = additionne(1, 3)

        assert(resultat == 4)
```

La fonction « additionne » n'existe pas. Cependant, nous pouvons la générer automatiquement avec un IDE. Nous pouvons aussi simplement l'écrire :

```
def addition(premierNombre, deuxièmeNombre) :
        //A compléter
```

Dans le corps de la fonction, nous avons plusieurs solutions. Nous pouvons décider d'écrire logiquement ce que nous attendons d'une fonction d'addition. Cependant, l'un des buts du Test first est de s'assurer de tout tester. Une des solutions est de n'écrire que le minimum de code nécessaire pour faire passer le test au vert, c'est-à-dire pour le faire fonctionner. Nous allons choisir cette deuxième option et renvoyer le minimum que nous pouvons avoir pour avoir un test correct :

```
        def addition(premierNombre, deuxièmeNombre) :
                return 4
```

À partir de là, nous pouvons ajouter un deuxième test pour aller plus loin et réellement écrire notre fonction d'addition :

```
def testQue7Plus4Egal11() :
        résultat = additionne(4, 7)

        assert(resultat == 12)
```

Notre test échoue puisque dans notre fonction d'addition, nous ne renvoyons que 4. Cela signifie qu'il nous faut adapter notre fonction. Ainsi, nous pouvons la modifier :

```
        def addition(premierNombre, deuxièmeNombre) :
                return premierNombre + deuxièmeNombre
```

Cette fois-ci, les tests n'échouent plus. L'idée du Test Driven Development dans son ensemble est de constituer une boucle. On commence par écrire un test. Celui-ci doit échouer. Il est rouge. On écrit ensuite le code, juste ce qu'il faut pour faire passer le test. On relance le test qui passe en vert. On prend un temps de pause pour se demander s'il ne faut pas réusiner le code et le faire si tel est le cas. À partir de là, on reprend. On écrit de nouveau un test qui échoue, un peu de code, etc. Cette boucle ne s'arrête qu'à la fin du développement, donc souvent à la fin du produit.

Test first en data science

La méthode Test first a plusieurs avantages. Elle permet de savoir où on en est quand on veut tester quelque chose. Les tests accompagnent les développements tout du long. Le retour est rapide. Dès qu'on écrit un morceau de code, on sait où on en est. Cela permet aussi de ne pas avoir à écrire tous les tests à la fin et de se demander par où commencer devant la montagne qui se dresse devant nous. On peut cependant ajouter qu'en data science, on ne sait pas toujours si un code va aller en production. De ce fait, on est souvent frileux à investir tout de suite dans les tests. Investir dans le Test first est ainsi encore plus compliqué. La data science c'est beaucoup de recherche et comme dans toute recherche une partie n'est pas gardé. Je comprends donc tout à fait cette opinion. Je reviendrai dessus, mais après quelques phases d'exploration et de modélisation, on sait davantage quoi attendre du projet. C'est à partir de ce moment-là qu'on peut se mettre à écrire des tests et à le faire avec la technique de Test first.

Il y a une autre bonne raison de se méfier de cette méthode en data science. C'est le fait que le principe même de la recherche c'est ne de pas savoir sur quoi on va aboutir. Il n'est pas possible de coder d'après une spécification en data science. Nous construisons celle-ci au fur et à mesure de notre recherche. Si on ne peut rédiger des contraintes avant, comment peut-on alors rédiger des tests avant le code ? Le Test first en data science ne peut être utilisée qu'avec parcimonie pour certains éléments où après recherche quand on arrive à être sûrs d'une solution.
Le Test first est aussi une technique qui peut être utilisé au moment où on réusine son code pour justement l'emmener en production plus sereinement. D'abord, on écrit le test. On le lance. Il est au vert. Il assure de ne pas perdre la fonctionnalité. On réusine le code. On lance le test. Tout est toujours vert. On a sans doute là quelque peu perdu les caractéristiques propres du Test first. Pourtant, on a gardé l'avantage d'écrire les tests au fur et à mesure du réusinage. C'est moins douloureux que tout à la fin.

Je suis assez partagée sur le Test Driven Development et le Test first en data science. D'un côté, je ne le pense possible qu'à certaines étapes. La dimension « recherche » de la data science en rend l'utilisation compliquée. Dans le même temps, quand je vois des équipes passer des semaines à trépigner parce qu'elles doivent écrire une montagne de tests avant de partir en production, je ne peux pas m'empêcher de me dire qu'il y a des avantages certains à penser Test first. Il y a au moins un gain à penser dès le début aux tests. L'un des avantages du Test first est aussi de rédiger un code testable de manière naturelle. Puisque celui-ci est écrit d'après un test, il est forcément testable.

Ainsi, on s'assure d'avoir un code plus facile à tester et donc mieux testé quand il part en production. En l'absence de Test first, avoir conscience qu'il faudra tester son code à un moment donné ou un autre m'apparaît important pour s'éviter des heures fastidieuses de réusinage.

Je voudrais parler d'un dernier point. Le Test first est une méthode qui peut parfois être vue comme dogmatique. Cette dernière a fait de nombreux fans. Beaucoup s'y retrouvent avec plaisir et se pensent plus efficaces avec. J'aimerais attirer votre attention cependant sur le fait qu'il n'existe pas en soi de bonnes pratiques ou très peu. Adeptes de science, nous savons qu'elle avance lentement. Or, aujourd'hui, s'il y a plusieurs études sur le sujet, il n'y a pas de consensus scientifique sur le fait que le Test first soit plus efficace. C'est un avis partagé par de nombreuses personnes dans le développement dont moi-même. Ce n'est pourtant pas une raison pour en faire un dogme. C'est une technique à essayer. Cela vaut la peine de se familiariser avec. Il reste qu'elle ne convient pas à tout le monde. Il vaut mieux utiliser ses propres méthodes si elles nous rendent efficaces et heureux que ce qu'une communauté considère comme de bonnes pratiques. La meilleure manière de ne pas être dogmatique, c'est en évitant de faire un absolu de cette technique et en ne la rejetant pas non plus sans avoir pris le temps de la tester.
Si vous voulez en savoir plus, je ne peux que vous recommander d'aller aux sources en lisant Kent Beck mais aussi Robert C. Martin qui ont contribué à l'expansion de cette pratique.

Je conclurai en disant que pour autant les tests automatiques sont primordiaux pour gagner du temps et ne pas faire de régression quand on fait évoluer un code. Ils aident aussi à débuguer et participent en partie à documenter une application de manière fiable. Si les tests passent, la documentation est à jour. Ce n'est pas toujours le cas des autres types de documentations parfois plus flexibles. Les tests sont un filet de sécurité importants quand on part en production. Ils méritent une attention particulière.
Nous parlons ici toujours de tests de code. Je reviendrai sur d'autres formes de tests en data science car les tests de code ne sont qu'une des multiples facettes des manières de tester son système de machine learning.

Des tests dans un notebook

Les notebooks sont fortement utilisés dans le monde de la data science. Ils ont un avantage certain pour faire de la data analyse et de la data visualisation par exemple. En ce qui concerne leur mise en production, les avis sont partagés. Pour ma part, je ne les apprécie guère. S'ils sont un outil que j'aime utiliser dans les premières phases d'un projet, j'ai plus de mal à les voir en production. Il y a à cela plusieurs raisons. La première est qu'ils sont durs à versionner. On peut en effet faire des snapshots de ceux-ci. Mais il est difficile de voir la différence entre deux d'entre eux. On peut alors versionner la version Python d'un notebook. Ainsi, le versioning est plus facile. On peut aisément voir la différence entre deux versions. Le problème est qu'un notebook ne s'exécute pas nécessairement comme un fichier Python. Il peut contenir ce qu'on appelle des commandes magiques en exécutant du SQL ou en affichant un graphique par exemple. L'import d'un fichier dans un autre ne se fait pas de la même manière non plus. On peut moins facilement exécuter le tout sur une plateforme de déploiement automatisé. De plus, même si les notebooks s'améliorent de jour en jour, ils n'ont pas les facilités d'un IDE en termes de réusinage par exemple. L'accès entre différents fichiers est difficile. On a

donc plutôt tendance à tout mettre dans le même notebook. Il est aussi plus compliqué de voir ce qu'on vient d'ajouter par rapport à la dernière version en cours. Pour toutes ces raisons, même si les notebooks présentent de nombreux intérêts, je les déconseille en production. Sergey Karayev liste ces éléments dans le cours *Full Stack Deep Learning*. Il cite le cas de Netflix qui ne déploie que des notebooks en production. Il présente l'infrastructure assez importante mise en place pour supporter tout cela. Le schéma est gigantesque. Pour Sergey Karayev, il est dommage d'en arriver là. Je n'ai jamais travaillé pour Netflix. Mais j'ai tendance à partager le même avis.
Une fois cela dit, on est parfois confrontés aux notebooks. Les choix sont faits en équipe. Il faut parfois se plier à l'avis du plus grand nombre même si on a des doutes quant à une méthode plus performante qu'une autre. Ainsi, donc on peut se retrouver à devoir écrire des tests pour un notebook. Cela est moins aisé que pour des fichiers. C'est encore là une raison de préférer se séparer des notebooks en production.

C'est tout de même possible dans un mode dégradé. Pour y parvenir, on va écrire des tests comme on le ferait normalement avec du code classique. Il s'agit de s'arranger pour lancer une exception quand il y a une erreur. Peu importe la librairie de framework qu'on choisit alors d'utiliser, l'important est de parvenir à gérer les erreurs quand on lance le notebook. Ce dernier doit être en erreur s'il y a un test qui échoue ou une autre forme d'erreur. De cette manière-là, on peut alors savoir si les choses se sont bien passées ou pas. Le processus est donc le suivant :
1. On déploie le/les notebooks quelque part dont le notebook de tests.
2. On lance le notebook de tests qui lance une exception s'il y a un problème.
3. S'il y a une exception, on remonte les erreurs.
Je reconnais que la solution que je propose ici ressemble davantage à du bricolage qu'à une vraie solution pérenne. Mais si vous êtes obligés de travailler avec des notebooks, il vaut mieux envisager des tests en mode dégradé que pas de test du tout. On est plus sur des tests d'intégration qu'on ne peut pas lancer souvent. Vous ne pourrez pas non plus faire de couverture de code ni évaluer d'aucune manière la pertinence de vos tests.

Automatiser le lancement des tests

L'idéal est de pouvoir lancer ses tests plusieurs fois par heure. On peut le faire avec une commande sur son environnement local. C'est déjà une première boucle de retour. S'il y a une erreur, on le voit tout de suite. Ensuite, il est nécessaire de pouvoir les lancer automatiquement à chaque fois qu'on envoie une modification de code sur le serveur commun. Cela permet plusieurs choses. D'abord, c'est un moyen de pallier le fait qu'on oubliera forcément à un moment donné ou autre de lancer ses tests sur sa machine. Ensuite, c'est aussi une manière d'être sûr qu'on n'a pas partagé du code non fonctionnel avec ses collègues.

Automatiser le déploiement

On entend souvent parler d'automatiser le « déploiement ». On est en droit de se demander ce qu'est le « déploiement ». Ici, je fais référence à la faculté de pouvoir à chaque fois qu'on ajoute une nouvelle ligne de code au serveur commun lancer des vérifications et un déploiement. Déployer

c'est pouvoir générer à partir des sources un environnement où les éléments sont utilisables. Cela peut être un environnement de recette ou alors la production. On met souvent en place des pipelines de déploiement qui suivent le schéma suivant :

1. Du nouveau code arrive sur le serveur commun où on gère les versions.
2. On fait quelques vérifications. Par exemple, on lance des tests.
3. On déploie.

Déployer peut en fait prendre plusieurs formes.

Déployer des scripts de code

En réalité, il peut s'agir de code de feature engineering, entraînement ou encore de prédiction. C'est la manière la plus simple de faire un déploiement. On prend les différents fichiers du dépôt (repository) et on les dépose sur le bon serveur au bon endroit. On lance généralement les éléments qui vont bien avec. Ainsi, par exemple on installe ou vérifie que les librairies nécessaires au fonctionnement des scripts sont bien présentes sur le serveur.
Pour ce chapitre, nous nous concentrons uniquement sur les parties feature engineering et entraînement. Ainsi, les étapes sont les suivantes :

1. Du nouveau code arrive sur le serveur commun où on gère les versions.
2. On fait quelques vérifications. Par exemple, on lance des tests.
3. On installe ou vérifie que les bonnes librairies sont présentes.
4. On copie les fichiers sur le bon serveur.

Déployer des notebooks

On peut aussi déployer des notebooks. Ce n'est encore une fois pas la manière la plus aisée. Mais c'est parfois celle que l'équipe préfère mettre en place. Les notebooks sont donc versionnées. La manière la plus simple de le faire est en transformant ceux-ci en leur version Python pure. Cela signifie que pour déployer ces notebooks, il faut les recréer à partir du Python. C'est parfois automatique sur certaines plateformes. Les tests inclus dans un notebook peuvent aussi être versionnés comme des fichiers de code classiques. Il s'agit aussi de les retransformer en notebook et de les exécuter. En fonction de la plateforme sur laquelle on déploie les notebooks, les librairies sont déjà présentes ou non. Ainsi, le processus est le suivant si les notebooks ont été versionnés comme des fichiers Python :

1. Du nouveau code arrive sur le serveur commun où on gère les versions.
2. On recrée les notebooks au bon endroit.
3. On installe potentiellement des librairies.
4. On exécute le notebook de tests pour vérifier que tout va bien

Si les notebooks sont versionnés comme des notebooks, il n'est pas besoin de les recréer. Cependant, il m'apparaît important de souligner encore que versionner des notebooks n'aide pas à se rendre compte des différences qu'il y a entre deux versions dans un outil plus adapté à du code tel que git. Il est aussi à noter que plus d'une plateforme est capable de lire un fichier Python et d'en faire un notebook de manière automatique. Je pense à Databricks par exemple.

Déployer des notebooks et des fichiers Python

En réalité, on peut aussi déployer les deux : des fichiers Python et des notebooks. D'un côté on peut avoir des packages Python qui contiennent la logique du code et de l'autre des notebooks qui forment la glue. De cette manière-là, on a peut-être le meilleur des deux mondes et c'est une manière plus aisée pour un data scientiste qui n'est pas forcément un développeur chevronné de démarrer. Dans ce cas-là, l'idée est d'avoir au fur et à mesure uniquement le minimum de code possible dans des notebooks. L'essentiel est contenu dans des packages testables, standardisables (si vous me permettez ce néologisme), manipulables à souhait et maintenables.

Les vérifications

Nous avons vu qu'à certains moments, nous faisons des vérifications de déploiement. En fait, nous lançons majoritairement des tests qui peuvent être unitaires ou plus globales. Ils peuvent aussi être davantage liés à la plateforme : y a-t-il assez de RAM, les librairies sont-elles là ?
Dans le cas des scripts, les vérifications, notamment les tests, peuvent avoir lieu tout de suite. Pour le déploiement des notebooks, les tests ne peuvent avoir lieu qu'après puisqu'il faut d'abord disposer du notebook pour les lancer. Cela signifie qu'il faut prévoir un retour en arrière en cas d'échec ou déployer le notebook de tests dans un endroit dédié.

À quoi ressemble un pipeline de déploiement ?

Il y a en fait plusieurs sortes de pipelines de déploiement en fonction de l'outil qu'on utilise. On peut très bien faire du déploiement avec un outil indépendant comme GitLab ou préférer une solution cloud. Assez souvent, nous avons affaire à du yaml qui décrit plusieurs étapes. Un pseudo exemple pourrait être le suivant :

```
étapes:

- étape : PythonVersion
    nom : Fixer la version de Python
    version: 3.7.8

- étape : Bash
    nom : Installer les librairies
    tâche : pip install -r requirements.txt

```

```
- étape : PythonTests
    nom : Lancer les tests
    dossier: tests

- étape : TestsPublication
   nom : Publier les résultats des tests
   dossier : tests

- étape: Bash
    nom : Copier les fichiers
    tâche : cp -R * $cibleFinale
```

Parfois, il y a des aides pour générer des morceaux de code. Finalement, c'est souvent du bash plus ou moins déguisé. On peut appeler cela des « wrappers ». Ce sont des fonctions toutes faites comme « PythonVersion ». Le pipeline est une manière de séquencer les éléments. Si une étape échoue, le pipeline s'arrête. Le déploiement n'a pas lieu. C'est une manière de s'assurer qu'il n'y a pas d'erreur avant d'envoyer quelque chose sur un environnement. Je reviens sur le fait qu'au final celui-ci peut être un environnement de recette pour faire des tests à plus grande échelle. Mais il peut aussi s'agir de la production. Nous faisons là en fait de l'intégration continue. Wikipédia définit ce concept ainsi : « L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée ».

Déploiement continu

Nous venons de parler d'intégration continue. C'est une arme redoutable pour une production stable et maintenable. Une autre est celle du déploiement continu. Cette pratique est fortement utilisée par des acteurs comme Netflix, Google ou Facebook. Cela signifie-t-il pour autant que c'est ce qu'il faut mettre en place dans un projet de machine learning et dans votre projet ? Le déploiement continu a plus d'un intérêt.
Le déploiement continu est le fait de mettre en production chaque élément valide, testé et considéré comme fonctionnel en production à l'usage des personnes utilisatrices du service. Cette validation se fait de manière automatique. On n'attend pas la fin d'une version pour déployer un bug, une fonctionnalité ou même un réusinage. Ainsi, il n'est pas besoin d'attendre la fin d'une semaine ou n'importe quelle date pour déployer. On déploie l'élément quand il est fini.
Par exemple, si on a réussi à faire une amélioration de notre modèle, il n'est pas besoin d'attendre une date qu'on s'est arbitrairement fixée pour déployer l'élément. On peut le faire immédiatement.

Le déploiement continu sert à la personne utilisatrice du système pour l'ajout de valeur constante et la correction de bugs. C'est la première personne actrice concernée par l'application qu'on crée. Plus vite l'amélioration d'un modèle lui arrive entre les mains, plus vite on lui apporte de la valeur. Le déploiement continu n'accélère pas le développement en soi, mais sa mise à disposition.
Du point de vue de la personne utilisatrice de l'application, il n'y a pas de raison pour qu'elle attende

que d'autres éléments soient finis pour avoir une amélioration.
Ainsi, si avec un nouveau paramètre ou une nouvelle donnée, le modèle est meilleur, il n'y a pas d'intérêt à attendre.

Il n'y a pas que l'ajout constant de valeur qui est important. Il y a aussi la correction de bugs. Si on note une erreur dans les données par exemple, il n'y a pas de raison d'attendre pour la corriger.

Le déploiement continu sert à ne plus avoir peur de mettre en production et à être plus stable. L'habitude nous permet de devenir performants. Ainsi, plus on déploie en production, plus on s'y habitue et on y devient performant. Cette performance se voit concrètement dans les tâches telles que l'automatisation, le monitoring ou la détection de bugs.

Quand on déploie souvent, on déploie peu de modifications de code. Il y a moins de risque d'échouer. Nous sommes donc dans une position plus stable.
Déployer peu de code, c'est se donner une chance d'en avoir une plus grande maîtrise.
Peu de choses arrivent en même temps en production. On en a un bon contrôle. Si problème il y a, on peut identifier l'erreur facilement.

Le déploiement continu sert à pouvoir effectuer un retour rapidement et de manière unitaire. Si ce qui est envoyé en production pose des problèmes, alors on peut retirer le code nouvellement ajouté. Il n'y a qu'un seul élément qui est supprimé. On n'est pas obligé de retirer tout un lot de fonctionnalités nouvellement ajoutées.
On n'hésite pas à le faire. On retire juste le code qui ne marche pas. On prend le temps de voir le problème. Puis on remet en production quand on a compris et corrigé le problème.

Le déploiement continu sert à éviter les processus lents, compliqués et source d'erreurs. On n'a pas différentes versions d'applications qui cohabitent, pas de multiples branches qu'il faut mettre à jour les unes sur les autres. On a moins de possibilité d'erreurs, de perte de temps et de bugs.
De plus, quand la personne qui développe annonce qu'un développement est fini, il est vraiment fini puisqu'il est en production.
Il ne faut pas le mettre dans une case temporaire et espérer que tout se passera bien lors du déploiement final.

Le déploiement continu sert à avoir un code fonctionnel tout le temps. Le code partagé par tous est constamment fonctionnel. C'est l'application à l'instant même. S'il n'est pas fonctionnel, tout le monde est déstabilisé. Cela se voit très vite. C'est prohibitif. Le code commun est très rarement non fonctionnel.

Pour faire du déploiement continu, il faut apprendre à penser en petits incréments Pour que le déploiement continu fonctionne, il faut penser en petits incréments. Il faut penser de manière très itérative. Avec cet exercice, on découvre que certaines fonctionnalités qu'on pensait constituer un bloc en représente souvent une plus petite série.
Par exemple, on peut envoyer une moyenne pour faire une première prédiction, quelque chose de très simple. Cela nous permet de mettre en place l'infrastructure en attendant.

Le déploiement continu est une méthode exigeante. Le code commun doit être stable à chaque instant. C'est plus sain. Mais il faut en effet y être préparé.

Un haut niveau d'automatisation est recommandé. Le déploiement continu est assez indissociable d'un haut niveau d'automatisation.
Celui-ci doit être élevé dans les tests unitaires, d'intégration, mais aussi dans le monitoring et l'alerting. C'est également plus sain pour emmener du machine learning en production d'une manière générale.

Il peut arriver que les utilisateurs ne soient pas favorables à cette méthode. En fonction de la criticité de l'application, il pourrait vouloir faire des vérifications et mettre un entraînement avant de se lancer dans les nouvelles fonctionnalités.

Il y a des difficultés nouvelles avec le déploiement continu en machine learning. Le machine learning réclame davantage de monitoring. Les erreurs sont souvent silencieuses. Il ne suffit pas de lancer des tests de code pour être sûr qu'il n'y a pas d'erreur. C'est la raison pour laquelle, en machine learning, le déploiement continu peut être plus compliqué. C'est loin d'être impossible, mais en effet, il faut s'armer avec une batterie de tests plus importante. En programmation traditionnelle, c'est plus simple de démarrer. En data science, il faut s'y préparer un peu avec des tests de données par exemple. Lancer un entraînement à chaque nouvelle modification est une bonne pratique. Cela évite quelques surprises. Ainsi, le déploiement continu en machine learning n'est pas impossible. Tout ce dont je parle dans ce guide - versioning et tests de données et de modèles - nous y aide.

En conclusion, le déploiement continu est une méthode qui comme beaucoup vient de l'ingénierie du logiciel. J'ai vu certaines équipes de data science la pratiquer naturellement sans même connaître le nom. D'autres au contraire ont des difficultés à la mettre en place pour plusieurs raisons plus ou moins justifiées. La moins justifiée, c'est celle où une personne venant du développement logiciel avec de vieux réflexes a interdit le déploiement continu parce que c'est ce qu'elle a toujours fait. C'est à mon avis la pire des raisons pour ne pas faire du déploiement continu et pour se fermer à une pratique d'une manière générale. Une raison plus justifiée est celle que nous venons juste d'évoquer. Le business et les utilisateurs ont l'habitude de passer par une phase de pilotage et ne veulent pas recevoir l'application avant, même si on le leur propose. C'est une raison justifiée. Après tout, si on n'aide pas les utilisateurs finaux à prendre le système en main, c'est dommage. On peut cependant essayer de montrer les avantages d'un système de mise en production plus rapide ou s'accorder sur ce qu'on peut déployer au fil de l'eau. Certains changements restent alors sous le contrôle d'une validation. On peut aussi utiliser la méthode du « feature flipping ». J'y reviendrai plus longuement, mais pour faire simple, c'est une manière d'intégrer son code dans le système sans le rendre actif. Ce n'est pas du déploiement continu stricto sensu, mais il nous permet de garder quelques avantages de cette méthode.
Enfin, la dernière raison, c'est qu'il peut être difficile de penser en incréments quand on met les éléments bout à bout d'un système de machine learning : nettoyage des données, préparation des données, modèle, prédiction, explicabilité. Il y a finalement quelque chose de procédural dans le machine learning. Le côté itératif est parfois difficile à voir. Il y a plusieurs solutions pour réussir tout de même à penser de manière itérative. Nous l'avons évoqué, nous pouvons mettre en production un système sans machine learning. Ainsi, toute l'infrastructure est déjà prête. C'est déjà une manière de s'assurer de la pérennité d'un élément important de notre système. Il arrive aussi parfois que le business soit intéressé par le nettoyage des données mêmes. On peut par exemple créer un dashboard pour présenter celles-là. Ainsi, on peut d'abord mettre en production cet élément avant le modèle.

Puis quand le système global est en production, si on trouve une amélioration, on peut la déployer naturellement dès qu'on la voit.
Il y a cependant des moments où ce n'est pas possible. Comme pour le Test first, il s'agit d'éviter le dogmatisme dans un sens ou dans l'autre. Dans le déploiement continu, ce qui est le plus important c'est de déployer souvent et de manière sûre. La méthode préconisée est de passer par un haut niveau d'automatisation.
Enfin, quelle que soit la méthode de déploiement utilisée, le plus important est peut-être d'avoir conscience des avantages et inconvénients que comportent nos choix. Le déploiement continu est une arme certaine pour une production pérenne.

Programmer le réentraînement

Il faut enfin que le réentraînement soit automatique. On peut programmer celui-ci de manière régulière. De cette manière-là, on est à jour avec les nouvelles données. Cela dit, ce n'est pas parce qu'on réentraîne continuellement qu'il n'y aura jamais d'erreur. On peut aussi programmer le réentraînement quand on voit un « model drift », le fait que la performance du modèle décroit avec le temps.

Réentraîner régulièrement ou en fonction du besoin

La solution la plus simple est de réentraîner de manière régulière. Cela permet d'être à jour par rapport aux données. Cela ne veut pas dire que cela suffit. Il ne faut pas réentraîner aveuglément. Il s'agit de monitorer le réentraînement pour savoir si celui-ci est performant et ne décroît pas. Du coup, c'est un avantage de réentraîner. Cela permet de collecter des métriques. C'est une première manière d'évaluer la pérennité de son modèle.
On peut définir la fréquence de réentraînement en fonction de l'arrivée des données. Si les données ne nous parviennent qu'une fois par semaine, on peut définir le réentraînement une fois par semaine. Ensuite, il faut mettre différentes formes de monitoring en place pour savoir si notre modèle est efficace. Par exemple, si on se rend compte que dans son environnement de production, les performances de notre modèle décroissent au bout de quelques jours, cela signifie peut-être qu'il faut réentraîner plus souvent. Même si un chapitre est consacré à cela, il me paraît important de souligner le « peut-être ». Réentraîner est une première manière de pallier le « model drift ». Mais c'est loin d'être suffisant. Il s'agit souvent de recalibrer le modèle pour s'adapter à un monde qui change continuellement.

Automatiser le réentraînement de manière régulière

Il faut ensuite mettre en place une forme de CRON, c'est-à-dire programmer un réentraînement régulier. Il faut dire quelque part « je veux lancer le réentraînement tous les mois ». Cela peut prendre plusieurs formes. S'il s'agit d'un script, on peut le lancer dans un CRON. S'il s'agit d'un notebook, on peut aussi programmer son exécution. Il est important quand on lance quelque chose comme cela de savoir qu'il n'y a pas eu d'erreur. Une manière de commencer est de savoir quand le

lancement a vraiment eu lieu, s'il y a une erreur ou un succès. Ensuite, ajouter des logs permet de savoir ce qui se passe. Dans le chapitre dédié au monitoring, nous verrons tout cela.
Souvent, on a besoin d'inclure ce script ou notebook dans un ensemble plus grand. On veut s'assurer que certaines choses en amont se passent bien avant de lancer le processus. On veut aussi souvent s'assurer que certaines choses en aval se passent bien. Ce qu'on cherche à faire dans ces cas-là, c'est à inclure cela dans ce qu'on appelle un pipeline. Comme nous l'avons vu, ce dernier terme est un polysème. Dans le contexte du déploiement, c'est le code souvent en bash ou yaml qui permet de déployer. Dans le contexte du réentraînement, c'est une autre forme de pipeline. On peut découper les choses ainsi :

1. Mise à jour des données
2. Réentraînement
3. Prédictions avec enregistrement des résultats
4. Mise à jour du dashboard à partir des résultats

C'est plus simple de réaliser tout cela dans un pipeline. Il y a différentes manières de faire cela. Il existe différents outils. Ils ne se valent pas tous. Certains comme Oozie sont sur le déclin. Airflow en est que j'apprécie particulièrement. Il arrive à allier l'avantage d'un pipeline écrit avec du code, du monitoring et une interface graphique. Le pipeline est écrit en Python avec des systèmes d'aides. Certaines fonctions nous aident à être plus efficients. Il existe bien sûr d'autres outils comme Luigi. Il y a aussi les plateformes de cloud qui proposent leur propre logiciel intégré comme Azure Data Factory.

La sérialisation

Parfois, on réalise les prédictions tout de suite après avoir réalisé l'entraînement. Beaucoup plus souvent, la partie prédiction se fait dans un second temps. Cela signifie qu'il faut sauvegarder d'une manière ou d'une autre le modèle. C'est là qu'intervient le concept de « sérialisation ». Wikipedia définit ce concept ainsi : « En informatique, la sérialisation (de l'anglais américain serialization) est le codage d'une information sous la forme d'une suite d'informations plus petites (dites atomiques, voir l'étymologie de atome) pour, par exemple, sa sauvegarde (persistance) ou son transport sur le réseau (proxy, RPC…). » C'est exactement ce que nous cherchons à réaliser. Nous voulons effectuer une sauvegarde de nos modèles dans une version plus petite pour des raisons de coût et de performance. Il existe plusieurs formats de sérialisation. C'est une chose à prendre en compte. Cela signifie qu'on ne peut pas enregistrer un modèle écrit avec Scikit-Learn dans un format de sérialisation spécialement et uniquement conçue pour Scikit-Learn et vouloir le lire ensuite avec Spark. Pickle est pratique mais n'est pas un format agnostique. Il y a une volonté d'unifier les formats de sérialisation et de proposer des formats qui marcheraient de manière agnostique. Dans tous les cas, la sérialisation est quelque chose à prendre en compte pour éviter des problèmes par la suite.

Feature engineering

J'ai inclus le feature engineering dans l'entraînement pour des raisons de commodité. Cependant, je ne pense pas qu'il serait honnête intellectuellement de ne pas se pencher davantage sur ce sujet. On pourrait considérer que le feature engineering est une étape préliminaire à l'entraînement. Nous voulons d'abord là extraire des fonctionnalités de données brutes. Cela peut se faire de différentes manières.

Différentes formes de feature engineering

On peut vouloir faire du feature engineering pour plusieurs raisons. L'une d'entre elles est purement technique. La plupart des algorithmes de machine learning n'acceptent que des valeurs numériques. Or, dans la vraie vie, on traite aussi avec des valeurs catégorielles comme par exemple le nom des pays. Il s'agit donc de transformer ces catégories en nombres.
On peut aussi vouloir faire du feature engineering pour réduire la dimensionnalité des features. C'est par exemple le cas avec l'âge comme entrée. Au lieu de garder toute la possibilité des âges, on peut en faire des tranches.
Le feature engineering peut aussi donner plus de sens aux données d'entrée. Le fait de vectoriser un texte par exemple lui confère davantage de sens. « Reine » et « Roi » sont des noms qui pris à part n'ont que peu de connexion. A part la première lettre, ils ne se ressemblent pas. Pourtant, sémantiquement, ils sont proches. Des algorithmes comme Bert vont remplacer les mots d'un texte par des vecteurs. Ceux qui sont proches ont un sens sémantique proche.
Le feature engineering a donc beaucoup d'utilité et peut être plus ou moins complexe.

Des librairies de feature engineering

De ce fait, il y a des librairies qui peuvent plus ou moins nous aider à construire nos features. Les frameworks comme Scikit-Learn ou Spark incluent de nombreuses fonctions de feature engineering prêtes à l'emploi : normalisation, standardisation, one-hot encoding, binarisation, etc. Cependant, ce n'est pas toujours suffisant. Il n'est pas rare qu'on doive réaliser nous-mêmes ces fonctions de feature engineering. Supposons que nous voulions prédire la consommation d'essence d'un véhicule. Nous avons les données par mois, mais voulons les convertir par heure. Ce genre de calcul ne peut pas se faire avec une fonction prête à l'emploi. J'ai d'ailleurs connu des contextes où le feature engineering était un projet à part entière pour imputer du sens à des données brutes. Le feature engineering est une étape de l'entraînement. Cependant, on a parfois besoin de reproduire ces mêmes étapes lors de l'étape de prédiction. On parle parfois aussi alors de « preprocessing ». On a besoin de transformer la donnée avant de s'en servir. Supposons par exemple que lors de l'inférence, la distance parcourue par les véhicules nous soit donnée en mètre mais que dans l'entraînement, nous traitions avec des valeurs en pied, elles-mêmes obtenues de valeurs initialement en mètre. Il nous faut faire une conversion. C'est quelque chose que nous réalisons alors de manière personnalisée.

Des étapes de feature engineering plus ou moins facilement sérialisables

On peut inclure dans la sérialisation les étapes de feature engineering. On ne se contente plus alors de sérialiser le modèle mais le pipeline de transformation des données et l'apprentissage. C'est pratique quand on a besoin du même pipeline lors de l'inférence. Ce n'est malheureusement pas toujours possible. Nous avons vu que certaines formes de feature engineering était plus complexe. Que faut-il faire alors ? S'interdire ces étapes ? Expliquer au business qu'on ne répondra pas à son besoin ? Copier le code deux fois dans l'entraînement et l'inférence ? On peut aussi mutualiser le code pour n'avoir à l'écrire qu'une fois, mais l'utiliser deux fois. Cela signifie qu'il faut écrire dans le même langage ou appeler un morceau de script extérieur à l'inférence. C'est cependant possible et cela évite d'avoir à maintenir deux éléments posés à différents endroits.

Beaucoup de code peut se cacher dans le feature engineering

Finalement, nous voyons que beaucoup de complexité peut se cacher dans le feature engineering. C'est donc une étape à tester comme les autres. Les tests unitaires de code sont plus évidents pour cette partie-là puisqu'il s'agit peut-être là de code plus standard. On fait de la transformation de données. On a une entrée, un programme, une sortie. Il s'agit de vérifier que cette dernière est bien celle attendue par rapport à l'entrée fournie.

Organiser et réutiliser ses features

Il commence à naître un nouveau concept qui s'appelle *les feature stores*. Je n'en ai testé aucun de ceux qui sont présents sur le marché. Cependant le principe d'organiser et de réutiliser ses features est une chose qui permet d'optimiser son travail. On calcule des features qu'on peut réutiliser. Cela permet finalement de ne pas avoir à calculer la même chose plusieurs fois. Accompagner ses features d'une description et les organiser dans un espace commun est une solution d'optimisation certaine.

La folie des notebooks

Je voulais encore une dernière fois revenir sur la question des notebooks. Nous avons vu qu'il est possible de les mettre en production, mais que cela n'est pas sans frais. Les notebooks sont une manière originale et nouvelle de faire du développement. Ils sont surtout utilisés en data science. Ils sont puissants, mais leur utilisation en production peut aussi être dramatique.
Les notebooks ont été rapidement adoptés par des personnes ayant peu de compétence en développement, mais pas seulement. Je les utilise pour ma part largement. Il est génial de pouvoir avoir une réponse tout de suite, visualiser des données rapidement et réaliser des graphiques en un rien de temps. C'est pratique pour faire de l'exploration de données. C'est aussi intéressant pour valider une idée et pourquoi pas commencer un prototype. Parfois même, je ne vois aucun inconvénient à ce qu'on pérennise un notebook en production. Par exemple, un « one-shot » ou la construction de dashboards est très pratique de cette manière-là. J'ai même tendance à préférer construire un dashboard avec un notebook qu'avec une application dédiée. En effet, au moins on peut versionner ce que l'on fait. C'est en fait plus professionnel de passer par là. On s'assure moins de régression. Je

peux même imaginer des notebooks avec peu de machine learning qui partirait en production. Par exemple, on pourrait avoir du clustering et présenter les clusters sous forme graphique.
En revanche, il est préjudiciable d'emmener de gros projets complexes qu'il faut maintenir en production s'ils sont sous la forme unique de notebooks. C'est de la folie en maintenance. Cela ne veut pas dire que cela ne se fait pas. Mais quand on a beaucoup de projets qui tournent comme cela, cela devient assez dur et coûteux en termes de temps. Pouvoir maintenir cet ensemble est difficile. Il faut ruser pour effectuer des tests, ruser pour les versionner, etc. Si vous me permettez un argument d'autorité, aujourd'hui, même une plateforme comme Databricks souhaite ne plus mettre en avant que les notebooks. Databricks est une interface construite autour de Spark. L'une de ces fonctionnalités phare sont ses notebooks. Cependant, cette interface a toujours permis de déployer du code Python classique ou même un JAR. Spark est après tout écrit en Scala. C'est donc a priori logique. Le problème qu'ils reconnaissent aujourd'hui, c'est que les notebooks ne conviennent pas aux projets trop complexes. Ce que je trouve vraiment intéressant, c'est qu'ils proposent aujourd'hui de plus en plus d'outils pour allier le meilleur des deux mondes : notebooks et packages Python.
Puisque tous les data scientistes ne sont pas à l'aise avec le code et que personne n'a jamais eu l'intention de les torturer avec cela, il existe des solutions pour faciliter ce travail. On peut construire un générateur de projet. Celui-ci contient tous les éléments pour démarrer et permet à chacun de s'atteler à la tâche. Il faut aussi faire du pair programming pour que les personnes puissent démarrer. Finalement, le plus compliqué est sans doute d'installer la bonne version de Python sur son ordinateur personnel et de se connecter au cluster. Pour cela aussi, il existe des services. Si, je prends le cas de Databricks, il existe « databricks-connect » qui permet de se connecter depuis l'extérieur à un cluster et à la plupart de ces facilités. Pour ce qui est de la génération de projet, il existe par exemple cookie-cutter qui permet d'avoir un template avec des fichiers déjà préremplis. Si on inclut là-dedans de quoi déployer, on est déjà pas mal avancé.
Je comprends la folie des notebooks. Cet outil est génial. Mais attention à la folie de vouloir l'utiliser partout et sans aucun filet de sécurité. Il existe des solutions pour soulager le travail des personnes qui ne sont pas à l'aise avec le développement. On n'est pas obligé de trafiquer sa mise en production ou mettre en danger la maintenance d'un projet pour régler ce problème.

Nouveaux problèmes

Automatiser l'entraînement représente beaucoup de méthodes et processus à prendre en compte. Il est vrai que tout ce dont nous venons de parler (automatiser le déploiement, tester, versionner, etc.) peut paraître effrayant de prime abord. Cependant ces éléments sont importants. De plus, il y a surtout beaucoup de décisions à prendre plus que de choses à faire. On se pose des questions pendant cette phase telle que « Quel outil choisir ? » ou « Quelle fréquence d'entraînement mettre en place ? ». Une fois, ces questions répondues, il reste à dérouler la mise en place.
Il est à noter que les méthodes une fois mises en place sont réutilisables pour d'autres projets.
Il existe aussi des personnes dont les compétences sont dédiées à la mise en production du machine learning. Ce peut être un moyen de commencer. Ce n'est pas nécessairement un recrutement à envisager de manière permanente. Son rôle peut être de mettre en place en place les choses et apprendre à ses collègues à être autonomes.

Pour conclure, automatiser l'entraînement est une étape importante en data science. Comme on a versionné les données, il s'agit de le faire avec le code et le modèle. Cela ne suffit pas de se concentrer sur le code. C'est une première étape. Pour des raisons de reproductibilité, il s'agit de versionner aussi le modèle. Cela signifie enregistrer celui-ci sous une forme miniaturisée. En langage professionnel, on dit « sérialisée ». Il s'agit aussi d'enregistrer les hyperparamètres utilisés et les résultats obtenus. Ces derniers dépendent de ce que l'on cherche à évaluer : justesse, mean square error, etc. Cela peut aussi être une métrique spécialement définie pour le business dans lequel on opère.
Puis, les tests de code sont cruciaux. Cela permet d'éviter des régressions inattendues. On gagne du temps en automatisant un travail manuel et répétitif.
Les sources étant testées et versionnées, on peut alors plus facilement les déployer, c'est-à-dire les déplacer sur un serveur approprié où elles pourront être lancées et données le meilleur d'elles-mêmes. Il ne s'agit pas de lancer l'entraînement à la main. Il faut définir quand le faire et mettre en place un système pour que cela se fasse automatiquement.
La sérialisation permet d'enregistrer le modèle. On peut enregistrer le modèle et les étapes de transformation de la donnée. Cependant, ce n'est pas toujours facile. Développer l'inférence et l'entraînement dans le même langage permettent alors d'éviter d'avoir des fonctions différentes à deux endroits différents. On peut aussi appeler un bout de script Python depuis Java. C'est aussi une solution.
J'ai parlé des notebooks. Certaines entreprises déploient des notebooks. D'autres s'y refusent. Si les notebooks ont de nombreuses fonctionnalités utiles pour l'exploration de données, ils sont durs à maintenir et représentent un effort pour la production. S'il faut déployer des notebooks, il y a cependant des solutions pour le faire. Il faut ruser pour tester par exemple, mais cela est possible. Il y a des limitations plus fortes comme l'intégration avec les IDEs qui est plus difficile. Cela dépend aussi du type de notebook utilisé.

Mes conseils pourraient être résumés ainsi :

- Versionner le code. Les développeurs le font depuis longtemps et en tirent grand bénéfice.
- Versionner aussi le modèle et les données
- Tester le modèle et les données
- Tester le code en soi. C'est un avantage certain.
- Pour aller plus vite, utiliser des outils pour faire des pipelines de déploiement et des pipelines de réentraînement.
- Préférez les packages Python aux notebooks. Ces derniers sont difficiles à maintenir en production.

L'entraînement ne représente qu'une partie de ce qu'il s'agit d'automatiser. Il nous manque une partie cruciale qu'on nomme l' « inférence ».

Automatiser l'inférence

L'inférence est le fait de donner à voir aux utilisateurs finaux les prédictions. Il y a plusieurs manières pour réaliser cela. Parfois, on a besoin de temps réel, parfois cela n'est pas nécessaire.

Parfois, on donne un ensemble de prédictions à la fois, parfois une seule. C'est un élément important d'un système de machine learning, parfois plus important qu'il n'y paraît. Il s'agit aussi de penser l'inférence dans le produit dans lequel elle s'interface. En effet, la prédiction peut être intégrée à une autre application. Elle peut aussi s'accompagner d'éléments pour la comprendre ou d'autres formes d'informations supplémentaires. L'intégration de la prédiction comme un produit n'est donc pas une chose négligeable. L'inférence est protéiforme, au moins tout autant que l'entraînement, peut-être plus. Elle demande parfois des compétences qui vont au-delà de la data science. Il arrive qu'on soit amené à réaliser une API web et/ou même une page web en elle-même. Dans ce cas-là, le déploiement prend une forme plus complexe. Certaines questions vont émerger telles que « Combien d'utilisateurs seront sur la plateforme ? » Même si on stocke les prédictions pour les utiliser dans un dashboard, la question se pose. « Les prédictions sont-elles volumineuses ? » Beaucoup d'interrogations qui amènent à différentes solutions.

Choisir le type d'inférence nécessaire à son contexte

Temps réel ou différé

A-t-on besoin d'avoir la prédiction immédiatement ? Ou cela n'est pas du tout nécessaire ? Les deux cas peuvent avoir lieu. Supposons une application qui prédit les offres commerciales qui déboucheront sur un accord. Dans ce cas-là, ce qu'on veut, c'est évaluer l'ensemble des offres et leur donner une note pour définir leur potentiel. On est sur du temps différé. Supposons un autre cas d'usage. Nous cherchons à évaluer le temps dont un technicien a besoin pour une intervention électrique. L'utilisateur reçoit une demande précise. Il a besoin de ces critères pour avoir une prédiction en particulier. Il souhaite celle-ci rapidement. Son intervention est prévue dans la journée et il doit répondre au client dès qu'il l'a au téléphone. Nous sommes plutôt dans du temps réel. Le technicien veut sa réponse dans l'immédiat.

Donner une prédiction en différé ou en temps réel ne se traite pas de la même manière. Pour du différé, la ou les prédictions sont calculées à l'avance en mode batch. Elles sont souvent enregistrées quelque part et réutiliser plus tard sous une forme ou une autre. Cela peut être un dashboard ou une autre forme d'intégration. C'est différent pour le temps réel. La réponse étant exigée immédiatement et ne sachant bien souvent pas les critères en avance, la ou les prédictions est/sont calculée(s) et donnée(s) en temps réel. On va souvent davantage se tourner vers une API web dans ce cas. Cela ne signifie pas pour autant qu'on ne stocke pas les prédictions quelque part. Mais on fait cela pour des raisons de monitoring.

Une prédiction ou un lot de prédictions

Avons-nous besoin de délivrer une prédiction ou un lot de prédictions ? C'est une solution qui oriente nos choix d'architecture. Dans le cas où on veut prédire les offres commerciales concluantes, on cherche à produire un ensemble de prédictions. On veut le statut de chaque offre, même celles non concluantes. On peut aussi imaginer qu'il y a différentes classes : les offres qui vont marcher à coup sûr, celles dont on ignore l'issue, celles plus périlleuses et enfin celles classées comme impossibles. Avec l'intervention du technicien, on a une seule prédiction à émettre. On cherche uniquement à

savoir le temps d'intervention.
Le fait d'émettre une prédiction ou un lot de prédictions n'est pas une question symétrique au fait de développer une solution de temps réel ou différé. On pourrait imaginer qu'un technicien nécessite d'évaluer systématiquement plusieurs interventions à la fois. Cela serait toujours en temps réel, mais les prédictions seraient multiples. Elles ne seraient sans doute pas aussi nombreuses que dans le cas des offres commerciales. Mais il pourrait y en avoir plusieurs. Il en va de même pour la prédiction en temps différé. Imaginons qu'il s'agisse de prédire si une épidémie nouvelle va avoir lieu. Le but du système serait de prédire uniquement cela. Il s'agit d'une seule prédiction, mais une importante prédiction avec de lourdes conséquences.

Il y a plusieurs problèmes qui peuvent se soulever dans l'inférence et plusieurs manières d'y répondre que nous allons détailler plus avant.

Les solutions

L'API web

Le cas de l'API web

En fonction de l'entreprise et du domaine dans lequel vous évoluez, ce cas peut être primordial comme négligeable. Il y a des compagnies qui y sont régulièrement confrontées. C'est par exemple le cas quand on fait de la prédiction pour une application web déjà existante. D'autres entreprises se retrouvent assez peu confrontées à cette situation.
L'API web répond au besoin de temps réel. On peut traiter le temps réel différemment cependant. En effet, si on connaît déjà les critères à l'avance, on peut envisager d'enregistrer les prédictions nécessaires et les servir en temps voulu. Mais quand on ignore les critères indispensables pour réaliser la prédiction, on peut passer par une API web qui répond en temps réel. Imaginons pour commencer le cas où un utilisateur envoie une requête à une API avec des critères. Ces derniers sont en fait des « features ». L'API prépare ces features, fait la prédiction, peut éventuellement ajouter d'autres informations et renvoie le tout à l'utilisateur. Une page web ou un autre système se charge alors de prendre la prédiction et de la mettre en forme.
Le cas de l'API web peut paraître challengeant en data science. Cela demande des compétences différentes qu'on n'attend pas forcément d'un data scientiste. Heureusement, il y a des logiciels pour faire des API web conçus pour répondre à un public de data scientistes.

Réaliser une API web

Ainsi, on peut citer à titre d'exemple FastAPI, outil en Python qui permet, comme son nom l'indique, de construire une API rapidement. En quelques lignes de codes, on obtient une API. On a de plus une documentation qui accompagne cette dernière. De cette manière-là, les personnes qui vont venir la consommer sauront comment s'y prendre. Cette documentation est générée automatiquement et sous différentes formes, à savoir du json, une interface graphique statique et une interface graphique interactive.

D'autres outils existent et verront sans doute le jour dans les mois et années à venir.
Même si FastAPI permet de générer des points d'entrée rapidement, cela ne donne pas le droit d'oublier qu'il faut très souvent sécuriser son API par une authentification. La documentation de l'outil présente comment réaliser cela.
FastAPI permet de générer des routes. Une route est une URL à laquelle on accède et qui dessert du contenu ou réalise une action. On dit qu'on fait une requête. Pour certaines routes, cela peut se faire via un navigateur. Quand on entre une URL, on fait une requête de type GET. L'URL nous donne une réponse. Celle-ci dans le cas de nos APIs sera souvent en JSON. On peut imaginer une route « monsysteme/predict ». Quand on y accède, on obtient quelque chose comme :

```
1 { « temps estimé en heure » : « 3.5 » }.
```

Pour donner des paramètres à cette URL (les features), on va passer par une requête de type POST. On ne peut pas y accéder depuis un navigateur, mais d'autres outils permettent d'y accéder.

Le modèle au sein de l'API web

L'une des premières choses que doit faire cette API, c'est donc renvoyer une ou plusieurs prédictions. Ainsi, on a par exemple une route qu'on peut nommer « prédiction ». Celle-ci est appelée via une requête POST en lui envoyant des informations. Celles-ci vont servir à construire les features et générer la prédiction.
Au cœur de cela, il y a donc un modèle qui est appelé pour renvoyer une prédiction. On peut faire cela de plusieurs manières. La manière peut-être la plus simple, c'est d'appeler le modèle là où il est stocké. Nous en avons déjà parlé, mais le stocker dans un système de versioning est une solution appropriée pour des raisons de monitoring et de reproductibilité. On a ce genre de systèmes avec MLFlow par exemple. On peut donc appeler le modèle depuis cet endroit. Le modèle est sérialisé. Cela signifie qu'il est dans une forme compressée.
Appeler le modèle depuis l'endroit où il est stocké n'est pas toujours possible. Il se peut que le modèle soit assez lourd et entraîne des lenteurs réseau. Pour éviter cela, il faut donc ruser. On peut alors décider de packager les modèles avec le code. Cela signifie qu'on va chercher les nouveaux modèles et qu'on redéploye à chaque fois qu'on a un nouvel entraînement. Le processus est ainsi le suivant :

1. On réentraîne. Cela produit un nouveau modèle.
2. On lance un packaging pour inclure le modèle dans le package avec le code.
3. On redéploye l'API avec le modèle.

De cette manière-là, on évite des lenteurs réseau. Ce n'est pas la solution la plus simple, mais elle a au moins le mérite de résoudre un problème. Puis, en réalité, on n'a pas toujours à faire à un seul modèle. Il arrive qu'on en ait plusieurs. Par exemple, quand on veut prédire un comportement utilisateur, on peut avoir différents modèles pour différentes tranches d'âge. Ainsi, le fait de devoir changer de modèle pour chaque requête entraîne des lenteurs sur le réseau. Packager le tout résout ce problème.

Le feature engineering au sein de l'API web

Souvent, on ne prédit pas directement à partir des entrées de l'utilisateur. On a souvent besoin de les retraiter. Par exemple, les entrées catégorielles doivent être transformées sous une forme numérique. Le modèle a appris avec les données dans un certain état. Il faut prendre les entrées de l'utilisateur pour les emmener dans ce même état.
Cela peut être un challenge si on prend des décisions hasardeuses. Par exemple, j'ai déjà vu des personnes écrire le modèle en Python avec des librairies spécifiques et préférer pour l'API partir sur du Java. Dans le cas auquel je pense, ce n'était pas très grave. La transformation des features était relativement simple. Les fonctions côté Python ont pu assez facilement être repassées en Java. On y a cependant perdu plusieurs choses. D'abord, il a fallu réécrire ces fonctions dans deux langages différents. Cela veut aussi dire qu'il faut les maintenir et potentiellement les débuguer deux fois. Elles sont plus difficiles à faire évoluer. D'une manière générale, le tout est plus difficile à faire évoluer. On a aussi dû passer par un format de sérialisation qui soit lisible dans plusieurs langages. En faisant cela, on n'a pas vraiment pu utiliser un système de versioning de modèle. On traque les métriques, paramètres et résultats de celui-ci mais dans un mode dégradé. Le modèle utilisé pour l'inférence n'est pas celui versionné. On perd alors le concept de reproductibilité.
Je n'ai rien contre Java ni aucun langage en soi. Je décris simplement un cas que je connais. Les personnes étaient motivées pour partir dans cette direction. La motivation d'un data scientiste est parfois plus importante que la meilleure des solutions. Je vous mets cependant en garde contre ce que cela peut apporter comme contrepartie. Je conseillerais d'utiliser un même langage pour l'entraînement et la prédiction. On n'a ainsi pas besoin d'écrire du code deux fois. On peut le mutualiser et le maintenir à un seul endroit. On peut avoir du feature engineering complexe qu'on n'aura pas de mal à reproduire côté prédiction. Une autre solution est d'appeler le code Python depuis l'application Java.

L'explainabilité au sein de l'API web

Nous aurons loisir d'évoquer plus avant ce sujet. Disons pour l'instant qu'on sait que beaucoup de systèmes de machine learning ont besoin de donner confiance aux utilisateurs pour être utilisés sereinement. On introduit alors une forme d'explication qui accompagne le modèle. Celle-ci peut être plus ou moins conséquente. Parfois, on renvoie juste les features qui ont le plus participé à la création du modèle. Cela peut être plus considérable. J'ai vu un cas où cela avait donné lieu à une application dédiée pour expliquer les détails de la prédiction. Finalement, celle-ci restait le cœur du système, mais beaucoup de choses gravitaient autour.
Ainsi, il faut parfois renvoyer dans l'API des informations sur la prédiction. Cette dernière est rarement la seule chose qui intéresse.

D'autres éléments au sein de l'API web

On peut donc être amenés à renvoyer plus que des prédictions. On peut aussi être amenés à avoir besoin d'interroger une autre API afin de compléter une information. L'utilisateur qui fait la requête n'a pas toujours en sa main tous les éléments nécessaires au système. C'est un élément à prendre en compte parce que cela peut générer du trafic et amener des lenteurs réseau.

Recueillir du feedback

Si on veut connaître la pertinence de sa prédiction, il faut d'abord la recueillir. De cette manière-là, on pourra la comparer avec la réalité et savoir où on en est. C'est là une manière de connaître son impact et d'évaluer le retour sur investissement. Les systèmes de machine learning sont parfois démunis quand il faut rendre compte de cela. L'une des raisons est qu'ils n'ont pas enregistré la prédiction. Les prédictions sont donc données aux utilisateurs sans qu'on sache ce qui a été délivré. Sans cette information, il est impossible de pouvoir évaluer l'impact business. Mettre en production un système d'apprentissage automatique ne s'arrête pas à sa capacité à délivrer des prédictions. Il faut encore connaître leur pertinence. Pour recueillir ce feedback, on a besoin des prédictions. On a aussi souvent besoin d'en savoir plus. Les prédictions sont-elles utilisées contre leur gré ? Sans conviction ? Ou pire en ayant l'impression qu'elles nous volent notre travail. Une prédiction qui marche est une prédiction pleinement accueillie par l'utilisateur. Il y a différentes manières de vérifier cela. Comme nous le disions, en premier lieu, on enregistre toutes les prédictions faites. De cette manière-là, on peut les comparer avec la réalité. Ensuite, des enquêtes orientées plus qualitatives permettent de savoir ce qu'il en est comme le fait que la prédiction est pertinente pour l'utilisateur. La meilleure des solutions est d'embarquer ce dernier dans le processus. C'est tout à fait possible dans certaines entreprises. Ainsi, si on peut le faire, on doit le faire. C'est lui l'acteur de notre système qui le reçoit. Il n'est pas toujours le commanditaire. Mais il est le premier acteur. Il vaut mieux ainsi l'embarquer avec nous pour être sûr que notre produit va marcher.

Le cas de la page web

Donc en tant que data scientiste, ingénieur machine learning, data ingénieur ou encore développeur data, on peut être amenés à construire une API web. Dans bien des cas, celle-ci est consommée par d'autres services. On n'a donc pas à s'occuper du rendu final. Il arrive cependant qu'on doive construire nous-mêmes l'interface web. Cela peut être de manière permanente ou non. Quand on sait que l'interface web sera compliquée, nécessitera beaucoup de graphiques, qu'on y passera du temps et que c'est une solution permanente, il vaut mieux choisir des outils adaptés à ce besoin. Il existe différentes librairies qui peuvent convenir.
Parfois, c'est un besoin temporaire ou nécessitant peu d'investissement. Dans ce cas, de nouveaux outils ont vu le jour. C'est le cas notamment de Streamlit. Ce logiciel est remarquable et adapté à un public de data scientistes qui n'a pas les compétences d'un développeur web et ne cherche pas à les acquérir dans l'immédiat. Streamlit est un outil qui permet d'écrire en Python des pages web. Il n'y pas ni HTML, ni CSS, ni JavaScript. On n'a besoin que de Python. Des fonctions écrites dans ce langage génèrent alors des pages. Certes, le design est assez restreint de fait. Mais on peut développer rapidement une solution sans être un expert en web. C'est pratique pour le prototypage. D'autres outils de même catégorie verront sans doute le jour. Il me semblait nécessaire de les évoquer. Ils facilitent grandement le travail. Ils permettent aux data scientistes de rester concentrer sur ce qui est davantage leur cœur de métier.

Containeuriser pour une reproductibilité système

Déployer une API web, c'est installer sur un serveur les sources et lancer l'application pour qu'elle tourne. Dans le cas de FastAPI, on déplace les sources sur le serveur et on peut utiliser uvicorn qui nous permet de lancer nos sources en ayant un point d'entrée pour y avoir accès.
Quand on déploie, on n'aime pas les mauvaises surprises. Par exemple, on ne veut pas voir que la version de Python a changé ou qu'une librairie n'est plus sur le serveur sur lequel on avait l'habitude de déployer. On veut pouvoir maîtriser son environnement et ne pas voir des déploiements échouer pour ces raisons. Cela tombe bien. Une solution existe : la containeurisation. Le grand outil pour réaliser cela est Docker. Ainsi, on ne va pas juste déployer ses sources. On va aussi le faire dans un conteneur simple dont on connaît les caractéristiques.

Déployer pour un grand nombre d'utilisateurs

Quand on commence à faire du web, on peut se retrouver face à des problématiques différentes en termes de déploiement. Si on n'a qu'une dizaine d'utilisateurs qui lancent quelques requêtes par jour, le déploiement est assez simple. On a une seule instance de notre API web. Les problèmes de performance ne peuvent être liés qu'à notre code ou des problématiques réseau en interne.
Cela se complique quand on a des milliers de requêtes par jour. Il y a aussi des cas où les requêtes se font par millions ou par milliards. On n'a pas forcément besoin d'être Google pour déjà devoir penser aux problématiques utilisateurs. Dans ces cas-là, on n'aura pas uniquement une instance de notre API Web, mais plusieurs. Cela peut paraître abscons, mais c'est le cas pour beaucoup de sites web sur lesquels on se rend tous les jours. Généralement, l'API web est copié plusieurs fois. Par exemple, on peut avoir trois instances. Les utilisateurs sont redirigés sur l'une des trois. Cela permet de gérer le trafic. Toutes les requêtes ne sont pas redirigées vers la même machine, ce qui créerait un embouteillage. Au lieu de cela, les requêtes sont distribuées sur les trois machines. Le grand outil pour faire cela est Kubernetes. On en entend beaucoup parler en ce moment. C'est justifié. Il répond à des problématiques de déploiement de manière efficace. Il permet avec quelques fichiers de configurer plusieurs instances qu'on appelle alors des replicas. On peut définir les prérequis de la machine sur laquelle on veut déployer. On peut définir des règles de routage. Cela nous permet de distribuer nos instances et d'avoir des déploiements transparents pour l'utilisateur.

Ainsi, on peut être confrontés au développement d'une API web.

Le dashboard

Le dashboard : l'outil du temps différé

Le deuxième cas assez répandu est celui du dashboard. Il fonctionne bien pour le temps différé. Il y a des cas où ce dont on a besoin, ce n'est pas d'une prédiction dans l'immédiat. On peut avoir besoin par exemple d'une prédiction qui est délivrée tous les jours ou toutes les semaines. Ce peut être le cas quand on veut prédire des pannes par exemple ou l'arrivée d'une nouvelle maladie. La prédiction n'est pas donnée immédiatement. Finalement, dans la prédiction en temps différé, c'est souvent la prédiction qui vient à nous alors que dans le temps réel, c'est nous qui venons la chercher.

C'est nous qui interrogeons. Prédire de manière différée ne signifie pas que la prédiction ne doit pas être rapide. On veut savoir qu'il va y avoir une panne avant qu'elle n'ait eu lieu. Autrement, cela ne sert à rien. Mais la prédiction est souvent donnée de manière plus régulière. Cela signifie qu'on a aussi une fréquence des prédictions. Cela vient s'ajouter à la fréquence du réentraînement. On va donc mettre en place un batch pour effectuer la prédiction. Celle-ci peut ensuite être affichée de différentes manières à l'utilisateur. En effet, souvent, on voit le résultat affiché dans un dashboard. Mais la ou les prédictions pourraient aussi être enregistrées dans une base de données qui serviraient à une autre application. En réalité, tous les scenarii sont possibles. Cela dépend de la manière dont l'utilisateur a besoin d'interagir avec le système prédictif.

Le dashboard : un outil intéressant pour les lots de prédictions

On peut être amenés à servir plusieurs prédictions. Reprenons nos deux exemples. On peut prédire des pannes ou des affaires commerciales concluantes. Dans ces deux-cas et d'autres où on donne une rangée de prédictions, le challenge est souvent de savoir quelle prédiction a compté. Lesquelles ont été vues ? C'est une question à laquelle on peut répondre avec un peu de monitoring. Lesquelles ont été utilisées ? C'est parfois compliqué. Je ne connais pas de solution parfaite à ce problème. J'ai vu plusieurs cas où les utilisateurs expliquaient ce qu'ils avaient vus et utilisés. Cela peut paraître fastidieux pour l'utilisateur. On peut faire en sorte que le processus de retour s'intègre à son travail. En effet, les techniciens dans le cas des pannes rédigent un rapport quoi qu'il arrive pour expliquer leur intervention. Il en est de même pour les commerciaux qui se saisissent d'une offre pour en suivre le déroulement et la mettre en ordre de marche si elle est concluante. Ces cas ne sont donc pas forcément les plus difficiles quand on les regarde d'un peu plus près.
Un cas plus compliqué, c'est par exemple quand on conseille une vidéo à la fin d'une première. Dans tous les cas, ce qu'on cherche à savoir, c'est si la ou les prédictions améliorent le système en général. On peut donc réaliser un AB test pour savoir si le fait de recommander une vidéo améliore le service, le détériore ou n'a aucun effet.

Réaliser un dashboard

Comme pour l'API web, réaliser un dashboard demande de nouvelles compétences. C'est une manière de présenter des prédictions qui est très utilisé. C'est dans ces moments-là que le data scientiste se met à faire de la data visualisation, surnommé « data viz ». La data visualisation consiste à mettre en images et en schéma de l'information. En réalité, cela peut prendre des formes très diverses. On pourrait considérer que la vidéo peut être un medium. Dans tous les cas, il s'agit de rendre compte de l'information. Cela doit être efficace. Cela signifie que l'information doit être rapidement saisie. Elle doit aussi être pertinente, c'est-à-dire présenter l'essentiel du contenu de manière claire, non biaisée et transparente sans noyer la personne qui regarde les schémas. Nous nous sommes peut-être habitués à la data visualisation pendant la pandémie de coronavirus. Pourtant la data visualisation était déjà partout avant cela. Elle nous permet de nous informer et même de prendre des décisions.
De même que pour le fait de développer une application web, elle peut décontenancer les data scientistes. Elle demande des compétences qu'on n'a pas toujours. Il est plus aisé de se faire aider

de personnes spécialisées en interface utilisateur et design. On n'a pas toujours ces professionnels à notre disposition. On peut se faire alors aider des premiers concernés par le système prédictif : les utilisateurs. Réaliser des expériences, questionnaires ou entretiens peut y aider. La data visualisation est donc pour une grand part une réflexion théorique qui se passe de technique.
Pour une autre part, il s'agit aussi de trouver le bon outil. Cela peut paraître négligeable. Mais au contraire c'est très important. J'ai eu à travailler avec un outil que je qualifierais d'assez peu professionnel : du « drag and drop » en apparence facile, mais sans système de versioning, des erreurs incompréhensibles données par un système fermé sans aucune possibilité d'améliorer l'outil ou de l'étendre. C'était la porte ouverte aux bugs et régressions. Ajoutez à cela peu d'expérience en data visualisation et vous aurez un cocktail magique peu apprécié par les financiers de l'entreprise, les utilisateurs, vos collègues et vous-mêmes. Je déconseillerais donc vivement de se lancer dans ce genre d'outils tête baissée. Certains sont plus stables et performants que d'autres. Quand on utilise une boîte noire, il vaut mieux s'assurer qu'elle ne nous lâchera pas en cours de route.
On peut aussi entièrement coder ses dashboards en Javascript par exemple, mais aussi en Python ou encore R. Je pense par exemple à D3.js, Dash ou R Shiny. Les possibilités sont ouvertes. Dans ce cas-là, on est très flexibles et libres de faire ce que l'on veut. On versionne. On code. On ne peut pas perdre ce qu'on a fait. Tout est enregistré. On peut conduire les dashboards dans le sens où on le souhaite. C'est plutôt séduisant. Cela ne va pas sans contrainte cependant. C'est forcément plus long à développer. Il faut tout faire soi-même. Ce n'est peut-être pas nécessaire pour tous les besoins. J'ai vu ces cas utiliser quand il y a un grand besoin de flexibilité et qu'on est donc prêt à coder et maintenir ce que l'on crée soi-même.
Dans d'autres cas, on peut utiliser des solutions intermédiaires comme Superset par exemple. Cet outil est un « drag and drop ». Mais c'est un logiciel open source auquel on peut avoir accès, qu'on peut comprendre et qu'on peut étendre.
Une autre solution à mon sens de plus en plus séduisante, c'est l'utilisation de notebooks comme dashboards. Certains systèmes de notebooks, comme celui de Databricks permettent d'ajouter des graphiques et d'en faire des dashboards. On a un peu le meilleur de tous les mondes : du code, du versioning, un peu de « drag and drop », etc. On peut cacher le code et ne donner à voir que le dashboard. Même si cette solution est séduisante sur le papier, elle comporte encore beaucoup d'inconvénients. Les notebooks n'offrent pas encore beaucoup d'options. Je ne l'envisagerais que pour des petits dashboards. L'export est difficile. Enfin, même si on peut versionner, on perd certaines informations qu'on ne peut mettre en place que par des interactions graphiques. C'est donc pour moi une manière de faire prometteuse, mais pas encore totalement au point.

Servir le modèle comme un produit

Le système d'apprentissage automatique s'intègre à un ensemble souvent plus large. Dans le cas d'une recommandation par exemple, il peut être associé à une application. Supposons un site web qui desserve de la musique. La recommandation de la nouvelle chanson à écouter cache en fait un système prédictif. Dans bien des cas, son succès n'est pas uniquement dû à sa pertinence. En effet, la recommandation est-elle placée au bon endroit ? Est-ce la meilleure location pour que l'utilisateur en tienne vraiment compte ? Est-ce utile pour l'application même d'avoir un système de recommandation ? Une recommandation aussi juste soit-elle n'aura aucune valeur si elle n'intéresse

pas l'utilisateur. On pourrait imaginer un site très spécialisé sur certains contenus. Les utilisateurs y viendraient dans un but précis. Dans ce genre de cas, est-ce qu'une recommandation aussi pertinente soit-elle est justifiée ? Ce n'est pas parce qu'un algorithme de machine learning a une très forte justesse qu'il remportera forcément le succès des foules une fois servi dans la vraie vie.
De la même manière, le système de recommandation n'est peut-être pas forcément à glisser sur n'importe quelle page. Il aura sans doute plus de pertinence à certains endroits que d'autres. Dans une page de contact, il est a priori assez inutile. Cela reste cependant à vérifier. Nous avons des préjugés, mais pouvons les remettre en question en évaluant le produit final.
Toujours dans le cas de la recommandation, on pourrait imaginer un système d'AB testing. Certains utilisateurs verraient la recommandation tandis que d'autres ne verraient rien. Il s'agit ensuite de savoir si cela fait une différence. La recommandation est-elle utilisée ? Influe-t-elle les écoutes ?
Une autre chose à considérer, c'est l'acceptation de l'outil prédictif. Pour que l'utilisateur s'en serve, il faut qu'il le reconnaisse comme utile ou intéressant a minima. Il s'agit d'avoir quelque chose qui ne soit pas envahissant ou effrayant. L'explainabilité aide l'utilisateur à prendre confiance en l'intelligence artificielle d'une manière générale. Le système prédictif doit aussi servir sa fonction première : aider. Le mieux serait qu'il ne soit pas trop disruptif, nouveau, mais pas trop envahissant dans l'application ou le processus. Ainsi, il s'agirait de lui donner une place physique à l'écran modérée. Il serait idéal qu'il s'intègre de manière discrète dans les outils quotidiens de l'utilisateur. Quant à la disruption dans le processus, il serait intéressant qu'il remplace une tâche par exemple. Il y a changement, mais le processus n'est pas plus long. Le temps est aussi une affaire primordiale. Si la prédiction est trop longue à être donnée, il y a des chances pour que la personne qui l'attendait s'en détourne et continue à faire comme avant.
Je dirais que le succès d'un algorithme de prédiction ne tient pas uniquement à ses qualités intrinsèques mais aussi à sa capacité à s'intégrer dans le produit ou le processus qu'elle améliore. Négliger cette partie, c'est prendre le risque d'avoir un système de machine learning très beau sur le papier, mais qui ne sert à personne. Le temps pour obtenir la prédiction, le temps pour en faire usage, sa place au sein de l'application et/ou du processus, la confiance qu'elle donne ou enlève, tous ces éléments sont importants. Le système de machine learning n'est souvent qu'un composant d'un ensemble plus grand. Je n'ai jamais travaillé pour une entreprise AI first. Mais j'imagine que certaines problématiques restent les mêmes. Il doit s'agir de faire accepter le produit pour qu'il réussisse. Il ne doit pas seulement être performant.

Nouveaux problèmes

Il y a beaucoup de compétences à avoir en inférence

Une seule personne peut-elle avoir toutes ces compétences ? Il y a peu de chance. Trouver quelqu'un qui serait expert à la fois en data science, en développement web et en data visualisation semble compliqué. Pour autant, ce n'est pas parce qu'on ne peut pas avoir toutes les compétences désirées dans une même personne qu'on ne peut pas les avoir dans une même équipe. C'est sans doute la meilleure approche.

Faut-il constituer une équipe dédiée à l'inférence ?

L'idée d'avoir des équipes par compétence n'est pas une bonne idée. Je sais que la mode est à la séparation des parties du processus en data science. Je vois parfois des personnes finaliser et mettre en production des choses qu'une toute autre personne a développé. Cela ne me paraît pas une bonne idée. Il vaut mieux avoir idée de ce que c'est que mettre en production quand on développe. On le fait différemment. De la même manière, les personnes qui mettent en production savent ce qu'elles activent, les détails de cela, comment le monitorer au mieux, etc.
Une séparation des tâches entre différentes équipes est périlleuse. Réunir les data ingénieurs et les data scientistes en une seule équipe est bien plus bénéfique. En effet, une même personne ne peut pas avoir toutes les compétences du monde et être à la fois un expert en web, data pipeline, data visualisation, statistiques, etc. Mais une équipe peut avoir ces compétences. En collaborant et en partageant ces différentes compétences, elle forme un tout qui peut affronter un projet de machine learning et le mener à bien du début à la fin.

Automatiser l'inférence revient à prendre beaucoup de décisions

En effet, ce ne sont pas que des compétences techniques. Il ne faut pas juste savoir utiliser un outil de « dataviz » ou avoir des compétences web. L'inférence, c'est aussi prendre des décisions. Temps différé ou temps réel ? Lots de prédictions ou une prédiction ? Généralement, ces questions vont de soi et répondent à la problématique de base.
Ce qui est parfois plus difficile, c'est comment intégrer au mieux le système prédictif pour qu'il soit apprécié des utilisateurs. Le faire participer au projet, explainabilité, discrétion, réflexion avec tous les acteurs sont autant de choses qui peuvent répondre à cela.

En conclusion, l'inférence, c'est le moment où on donne à voir la prédiction. Cela peut se faire de différentes manières. Il y a plusieurs questions à se poser afin d'arriver à une conclusion. Par exemple, sommes-nous davantage sur une solution en temps réel ou différé ? Avons-nous besoin de répondre par une prédiction ou sommes-nous davantage dans une situation où nous renvoyons un lot de prédictions ? Savons-nous quels paramètres vont être donnés pour faire la prédiction/les prédictions ? Si oui, on peut enregistrer des prédictions types en avance.

Une API web répond généralement au cas où on a besoin de temps réel et où on renvoie une ou quelques prédictions. Pour être réalisé, on a besoin d'avoir des compétences en développement web. Cependant, beaucoup d'outils facilitent aujourd'hui le travail pour les data scientistes. Aujourd'hui, on peut même réaliser une interface web uniquement avec du code Python. Ce n'est pas le plus flexible cependant.
Un dashboard répond généralement au cas où on veut visualiser un ensemble de prédictions sans avoir besoin de faire une requête sur demande. Cela nécessite des compétences en data visualisation. Il s'agit de compétences techniques, mais aussi théoriques. Il n'est pas toujours évident de savoir comment représenter au mieux l'information. Certaines entreprises vont avoir des personnes dédiées à cela. Dans d'autres, les data scientistes doivent s'en occuper endossant une nouvelle casquette.
Enfin, on peut aussi intégrer les prédictions dans un outil tiers. C'est sans doute la meilleure manière de toucher les utilisateurs que de s'interfacer dans les outils qu'ils utilisent déjà.

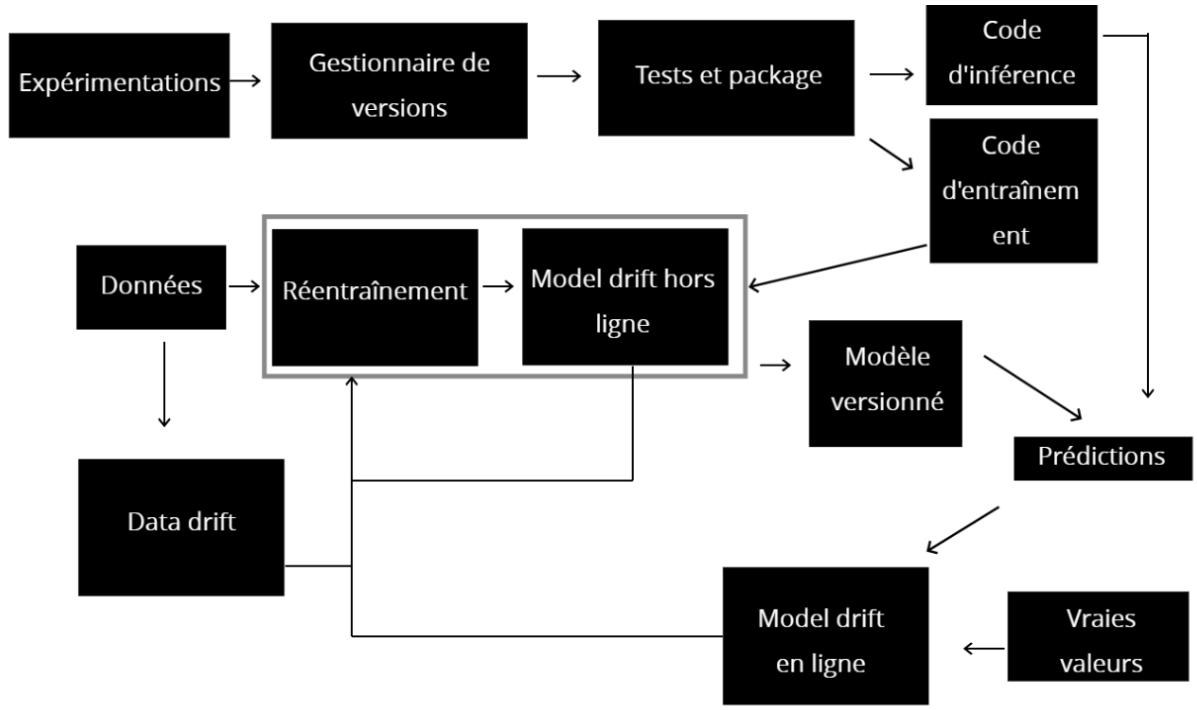
Dans tous les cas, l'inférence est le moment où on présente la ou les prédiction(s) aux utilisateurs. Il s'agit donc de susciter leur intérêt.

Mes conseils pour finir seraient les suivants :

- Répondre aux questions de base pour faire de l'inférence : Temps réel versus temps différé ? Une prédiction versus un lot de prédictions ?
- Répondre aux questions moins basiques mais tout aussi importantes : Où et comment intégrer le système prédictif ? Comment susciter l'intérêt des utilisateurs ? A-t-on besoin d'explainabilité ? Sous quelle forme ?
- Former une seule et même équipe ayant des compétences diverses : web, datavisualisation, déploiement avec des outils comme Docker et Kubernetes
- Réfléchir aux logiciels à utiliser et faire un compromis entre flexibilité et rapidité
- Penser au modèle comme à un produit qui doit se vendre

L'inférence est une partie non négligeable du système prédictif. C'est là où on dessert la prédiction. C'est là où on commence à avoir une action réelle sur le monde.

Nous avons vu comment automatiser un système de machine learning en passant par ses différentes étapes : un socle solide avec des données fiables, un entraînement robuste et reproductible et une inférence capable de susciter l'intérêt de l'utilisateur. Sur notre graphique, nous pouvons de nouveau visualiser ces différents éléments.



Tirer profit du machine learning en production

Maîtriser son impact et détecter des bugs avec le monitoring

Cette partie parle davantage de monitoring en général que de monitoring d'apprentissage automatisé. Ce dernier sujet sera développé dans le chapitre suivant.

L'importance du monitoring

Avant de s'intéresser à ce qu'il s'agit de monitorer et comment, demandons-nous pourquoi monitorer et voyons ensemble l'importance de ce sujet.

Les raisons du monitoring

Commençons peut-être par les raisons qui nous poussent à faire du monitoring. Mettre en production ne signifie pas seulement mettre quelque chose - un modèle dans notre cas - à disposition des utilisateurs. Il faut s'assurer au quotidien qu'il n'y a pas d'erreur et qu'on tire profit de ce qui est en production. C'est à ce moment-là qu'on commence à rentrer dans une phase de maintenance. Ce dernier terme, « maintenance » peut paraître négatif. Il n'en est rien. C'est là où le spectacle commence. Quand une application est en maintenance, cela veut dire qu'elle est en production, qu'elle est dans le monde réel et qu'elle lui donne un sens. Être en maintenance ne signifie pas que le modèle n'évolue plus. Là où l'idée de « maintenance » peut effrayer, c'est qu'à présent les bugs qu'on va rencontrer risquent de se trouver en production et d'impacter les utilisateurs finaux. L'enjeu est différent quand on développe sur sa machine. Il s'agit donc de se parer aux éventuelles pannes. Pour cela, il faut être attentif au système qui est en production. Finalement, le monitoring, c'est mettre sur écoute son système afin d'être sûr qu'il n'y a pas d'erreur. On peut ainsi réagir quand il y a un problème. On sait aussi la valeur qu'on apporte par ce biais-là. Ce sont les raisons du monitoring.

Différentes formes de monitoring

Il existe plusieurs formes de monitoring. Il y en a que j'aurais envie de qualifier de plus classiques. On parle souvent de monitoring quand on veut écouter l'état de santé de serveurs : CPU utilisé, état de la RAM, etc. Généralement, on peut s'appuyer sur des dashboards préétablis ou mieux encore des

outils déjà configurés. Les plateformes de cloud notamment facilitent cela.
Il y a aussi le monitoring d'application : en combien de temps ce traitement s'exécute-t-il ? Ai-je eu des réponses autres que des 200 dans le cas d'une API web ? Combien de données ai-je ingérées ? Ces éléments sont spécifiques à l'application et vont varier selon le contexte. Elles conviennent à toute forme d'application.
Il a aussi le monitoring spécifique aux applications de machine learning : Comment le modèle performe ? Comment la donnée est-elle distribuée ? Dans le chapitre suivant, nous allons nous concentrer sur les spécificités liées au monde de l'apprentissage automatique.

Solutions pour maîtriser son impact et détecter les bugs

Monitorer l'entraînement

Il s'agit ainsi de monitorer les différentes parties de l'application. Commençons par l'entraînement. Pléthore d'éléments peuvent nous intéresser. Nous pouvons observer le nombre de données ingérées. Nous pouvons ajouter à cela le nombre de données filtrées ou encore les features créées. Nous pouvons loguer tout cela pour y revenir. Nous avons à notre disposition des bibliothèques comme Logging en Python. Nous pouvons aussi construire des dashboards au-dessus de cela pour avoir un côté plus visuel. Grafana, DataDog ou d'autres formes de data visualisation spécialisées dans le monitoring peuvent être utiles. Enfin, on peut mettre en place des alertes. Par exemple, si trop peu de données ont été ingérées, on peut décider de lancer une alerte, c'est-à-dire qu'on peut recevoir un email ou une notification dans un chat par exemple. Ce seuil est à définir et dépend beaucoup du contexte business.
Dans le monitoring d'entraînement, il y a aussi le fait d'être sûr que le modèle performe au moins sur des données de validation. On peut loguer, grapher et envoyer une alerte pour cela aussi. Ce type de monitoring sera détaillé au prochain chapitre. Il m'apparaît important de noter que ce type de monitoring ne remplace pas le fait de placer des écoutes sur le système dans sa vie réelle. Ce n'est pas parce que le modèle est performant sur des données de validation qu'il l'ait vraiment dans la vraie vie. Cela constitue simplement un premier indicateur.

Monitorer l'inférence

L'autre grande partie de l'application, c'est l'inférence. Celle-ci aussi doit être monitoré. Il faut vérifier qu'il n'y ait pas au minimum d'erreur apparente. Ce qu'on logue est ensuite spécifique au contexte. Les exigences ne sont pas les mêmes non plus. Pour une application web, il s'agit de donner une réponse rapidement. Pour des prédictions calculées et enregistrées en avance, nous pouvons nous permettre davantage de latitude.

Les métriques

Un des enjeux est ne pas se retrouver envahi de métriques qui n'apporteraient pas ou peu d'information. Une métrique représente tout ce qu'on mesure, peu importe la forme que cela prend. On peut avoir beaucoup de métriques à condition de savoir comment les organiser et à qui les présenter. C'est pourquoi il faut imaginer plusieurs niveaux de métriques :

- Les logs : il peut y en avoir beaucoup tout au long du code. Ils sont consultés par les data scientistes. Ils sont peut-être davantage à consulter quand il y a un bug.
- Les dashboards : ceux-ci reprennent quelques logs d'une manière plus visuelle. Consultables plus facilement par n'importe qui, ils peuvent être à l'adresse de n'importe qui. On peut en imaginer plusieurs, mais moins que les logs. Ils peuvent être consultés tous les jours même s'il vaut mieux remonter une alerte quand il y a un problème.
- Les alertes : ces dernières remontent des points vraiment critiques. Elles sont peu nombreuses, mais doivent être choisies efficacement.

Les cibles

Il peut ainsi y avoir plusieurs cibles au monitoring :

- Les data scientistes en charge de maintenir l'application
- Le métier, plus éloigné, mais qui vérifie que le système continue à créer de la valeur

Les outils

Quant aux outils possibles pour réaliser du monitoring, ils sont nombreux. J'ai cité Logging pour loguer en Python. On peut ensuite construire des dashboards avec Grafana ou encore DataDog. Pour monitorer les performances du modèle lors de l'entraînement, Databricks recommande de passer par les notebooks et d'en faire des dashboards.
Les possibilités sont ouvertes. Notons aussi qu'on peut se servir de choses déjà faites comme des dashboards déjà faits ou préparamétrés. Cela est possible dans les plateformes de cloud. Dans Grafana, il existe des modèles de dashboard.

Les post-mortem

Il est difficile de tout prévoir en avance. Cela ne veut pas dire qu'il ne faut rien loguer et espérer qu'il n'y aura pas d'erreur quand on part en production. On peut poser les basiques et ce qu'on imagine comme allant de soi. Une fois le système lancé dans la vraie vie, il peut cependant être intéressant de noter les pannes. Celles-ci auront malheureusement lieu. Si on veut éviter à tout prix les bugs, il vaut mieux ne pas développer. L'idée est davantage de les limiter, de les noter, d'en mesurer l'impact et de savoir y remédier de manière que cela ne se reproduise pas. Pour ce faire, on peut employer la pratique des post-mortem. Il s'agit de traquer les erreurs et de leur donner du sens. Ainsi, quand un problème arrive en production, on peut enregistrer cela, noter l'impact et la cause. Par la suite, on peut catégoriser ces éléments. En faisant cela, on sait davantage où concentrer nos efforts.

Nouveaux problèmes

Le monitoring est protéiforme et omniprésent

Quand on fait du machine learning, il faut monitorer les performances système et les performances de l'application comme dans l'ingénierie du logiciel. Mais il faut aussi monitorer les éléments

spécifiques du machine learning. Finalement, cela fait beaucoup de choses auxquelles penser. Face à cette tâche, il y a plusieurs solutions :

- Utiliser des outils qui simplifient la tâche
- Réaliser des incréments. Mettre peu de chose en production et monitorer immédiatement fait paraître la tâche moins grande.
- Avoir des ressources dédiées à cela dans son équipe, au moins au début pour commencer

Le monitoring en machine learning est-il vraiment spécifique ?

Il l'est vraiment. Nous allons voir cela très rapidement. On peut résumer en disant que le monitoring en apprentissage automatique, c'est d'abord le monitoring auquel on a l'habitude de faire face en développement de logiciels. À cela s'ajoute un monitoring dédié au côté stochastique du système et à la complexité des données avec lesquelles on traite.

Pour conclure, comme dans une application classique, le monitoring est important en machine learning pour savoir s'il n'y a pas d'erreur et pour pouvoir réagir quand il y a un problème. C'est aussi ainsi qu'on sait si notre système apporte de la valeur. Les deux grandes parties de notre application, à savoir l'entraînement et l'inférence doivent être monitorées. Il peut y avoir plusieurs niveaux de monitoring plus ou moins détaillés avec des cibles plus ou moins techniques.
Il s'agit de définir des métriques adéquates qui nous permettent d'un seul coup d'œil de comprendre l'état du système.
Il y a différentes cibles au monitoring. Le business peut vouloir avoir quelques informations de surface pour comprendre rapidement si le système apporte toujours de la valeur. La personne qui débugue le système doit avoir beaucoup plus de détails.
Il existe beaucoup d'outils sur le marché qui peuvent nous aider. Avoir des logs est la première étape. On peut ensuite réaliser des systèmes d'alerting et avoir des dashboards.
Enfin, malheureusement, les bugs même s'ils doivent être évités, arriveront. On peut s'appuyer sur ceux-là pour savoir où investir du temps pour maintenir notre système de manière optimale. Réaliser des post-mortem peut nous y aider.

Mes conseils pour finir seraient les suivants :
- Commencer par monitorer le machine learning comme une application classique
- Ne rien délaisser : l'entraînement et l'inférence doivent être monitorés
- Définir des métriques pour le monitoring
- Définir différentes cibles et différents niveaux d'alertes
- Jauger les alertes pour qu'elles soient informatives et non pas envahissantes
- Avoir recours à des post-mortem quand quelque chose se passe mal

Les spécificités du monitoring en data science

L'importance du monitoring en data science

Le concept de « model drift »

Le model drift correspond au fait qu'avec le temps la performance d'un modèle diminue. C'est quelque chose d'assez fréquent et de dangereux. Un modèle très performant à un moment donné s'il est laissé sans surveillance peut décroître dans la justesse de ses prédictions. Il perd de son intérêt. Les utilisateurs peuvent ne plus avoir confiance. Le système peut être délaissé. Dans des cas de systèmes prédictifs critiques comme en médecine, cela peut-être vraiment nocif. Certaines manifestations de maladies évoluent aussi par exemple. C'est pourquoi, il faut monitorer le model drift qui peut arriver pour plusieurs raisons.

Concept Drift

Dans l'apprentissage automatique supervisé, on a des features qui permettent de prédire un label. Ce label peut évoluer. Le cas qui est le plus souvent cité est celui de la fraude. Supposons un système qui détecte les fraudes bancaires. Il peut être très performant à un moment donné. Mais cela ne suffit pas. La manière de faire des attaques évolue constamment. Ainsi, le label, donc le concept, va évoluer, ce qui conduit à un model drift.

Data Drift

La plupart du temps, ce sont les features qui évoluent. Un exemple classique est celui de la saisonnalité. Si l'une des features est la température et qu'on effectue un entraînement au mois de janvier, les performances du modèle vont décliner. C'est un modèle naïf. Cependant, les tendances évoluent d'une manière générale. Tout change continuellement. Seul le changement demeure. Un model drift n'arrive pas forcément le mois suivant. Cela peut parvenir au bout d'un an d'existence par exemple, voire plus.

Changements dans les données haut niveau

Les features peuvent changer pour des raisons différentes. Il se peut par exemple qu'une température qui était calculée en degré soit alors calculée en Farenheit. On peut même avoir des cas où une feature n'est plus alimentée. C'est une chose que Joel Thomas et Clemens Medwald expliquent assez bien dans leur article *Productionizing Machine Learning : From Deployment to Drift Detection (Mettre en production du machine learning : du déploiement à la détection de drift)*. Le fait même de parler de drift en général est intéressant et pertinent. Bien souvent, ce qu'on remarque en aval c'est le model drift. Pourtant, c'est généralement parce que quelque chose en amont a lieu.

Les données sont la matière première de nos algorithmes. Si le code n'a pas évolué, c'est l'origine du problème. En effet, excepté les erreurs de développement, nous pouvons remarquer que les trois

raisons d'un model drift sont liées aux données. Dans le premier cas, c'est le label qui varie. Dans le deuxième, il s'agit des features. Dans le troisième, c'est l'alimentation des features qui est en jeu.
Un modèle s'il n'est pas en production n'a aucune valeur. Mais un modèle qu'on laisse décroître avec le temps en apporte peu aussi. Dans un domaine comme le machine learning où on sait que les utilisateurs sont méfiants, il vaut mieux continuer à garder la confiance de ceux-ci. S'ils remarquent une baisse de performance dans le modèle avant nous, nous risquons de la perdre. C'est pourquoi le sujet du model drift est important. Il faut s'en prémunir, le comprendre, le monitorer et agir quand il est détecté.

Différentes formes de model drift

Il y a en réalité plusieurs formes de model drift. Oren Razon, CTO de superwise.ai en rend compte dans une interview pour le blog *ML in production (ML en production)*. Je suis admiratrice de ce blog tenu par Luigi Patruno. Le but de ce dernier est de donner des manières de mettre en production du machine learning. Dans cette interview, Oren Razon explique que la plupart des personnes imaginent le model drift comme quelque chose de lent, long et graduel. Ce n'est pas toujours le cas. Parfois, c'est abrupt. Il classe le model drift en quatre éléments.
Il y a la catégorie « graduelle » (gradual). C'est celle à laquelle on s'attend. Par exemple, les données évoluent lentement. Les choses changent continuellement. C'est pourquoi c'est celui-ci auquel on s'attend. Il y a la catégorie « soudaine » (sudden). Il y a une rupture. Cela arrive quand les données changent brutalement du jour au lendemain de forme. Par exemple, cela peut arriver quand une récession intervient brusquement. Les comportements d'achat sont modifiés du jour au lendemain. Il y a la catégorie « récurrent » (recurrent). Cela arrive quand on se rend compte que le changement intervient par période. Cela peut être dû à des saisons. On sait par exemple que les gens ne consomment pas la télévision de la même manière en été ou vers Noël qu'à d'autres périodes de l'année. Enfin, il y a la catégorie « écart » (blip). C'est quand on a une donnée différente un jour de manière isolée. Cela est souvent dû à des problèmes techniques. Une donnée a mal été générée.
Ce que Oren Razon explique c'est qu'il est pertinent de connaître ces différentes formes. En en ayant conscience, on peut mieux s'en prémunir. Un problème technique ne se traite pas comme un effet de saisonnalité par exemple.

Solutions pour gérer les spécificités du monitoring en data science

Il y a plusieurs manières de se protéger d'un modèle drift. On peut faire de la prévention. Mais cela n'est pas suffisant. Il s'agit d'une manière générale d'avoir différentes couches de monitoring.

Réentraîner

Réentraîner un modèle est déjà une première manière d'éviter une baisse de performance. Réentraîner signifie qu'à intervalle régulier, on réentraîne le modèle. On peut juger que cela est nécessaire tous les jours, mois ou encore semaines. Le faire souvent est intéressant pour monitorer son système.

Il y a cependant une balance à trouver. Certains réentraînements sont parfois très gourmands.
Mettre en place des réentraînements est déjà une manière d'éviter le model drift. On prend ainsi en compte les dernières données. Ce n'est cependant pas suffisant. Cela ne permet pas toujours d'éviter le model drift. Certaines features peuvent devenir obsolètes tandis que d'autres éléments qu'on n'avait pas pris en compte peuvent devenir très importants. C'est quelque chose qu'un réentraînement ne peut pas prendre en compte.
Cependant, réentraîner permet de connaître l'évolution du modèle dans le temps. Cela a lieu avec l'action conjointe du versioning de modèles. Les deux combinés permettent de connaître l'évolution d'un modèle dans le temps. En loguant des métriques telles que le R2 ou la précision, on connaît l'évolution de la performance du modèle dans le temps. C'est déjà une première étape.
Ainsi, réentraîner un modèle permet en partie d'éviter le model drift. En versionnant le modèle, on obtient un monitoring de la performance de celui-ci. On connaît son évolution.

Vérifier le réentraînement avant déploiement

On peut aller plus loin. Le processus consiste souvent à réentraîner, loguer le modèle et déployer. On peut vérifier que tout va bien avant de lancer cette dernière partie. On pourrait imaginer un système d'alerting. Quand la performance est jugée trop basse, on lance une alerte. En fonction de la criticité du système, on peut décider d'arrêter le réentraînement, continuer à utiliser le modèle de la veille, lancer du tuning d'hyperparamètres automatiquement, etc. Dans tous les cas a priori, une étape d'investigation est à envisager. Le model drift arrive souvent. Il est normal qu'un jour un réentraînement ne donne pas les résultats escomptés. Il vaut mieux le savoir et être prêt à passer un peu de temps pour mieux comprendre ce qui s'est passé. Nous allons voir qu'avec quelques formes de monitoring, ce travail peut être simplifié.
L'investigation sert aussi à éliminer le cas où le modèle n'est pas performant en raison d'un bug quelconque dans le pipeline. Nous ne sommes jamais à l'abri d'une erreur qui pourrait pervertir le système. Une fois cette possibilité écartée, on peut s'attaquer aux causes plus probables du model drift.

Vérifier dans le temps les performances du modèle

Au fur et à mesure des entraînements, il y a des fluctuations. Si par exemple, on a commencé avec une justesse de 82,897 % il est assez peu probable que ce chiffre exact ressorte la prochaine fois. On aura peut-être 83,989 % ou 81,989 %. Dans tous les cas, le nombre précis qui en résulte n'a pas beaucoup d'importance. En revanche, ce qui compte, c'est la tendance. Il s'agit de vérifier que les performances du modèle ne décroissent pas. Loguer les métriques de ce dernier n'est pas suffisant. Un peu de data visualisation complète cela. Avec quelques graphiques, on peut voir l'évolution dans le temps. Les courbes nous permettent de mieux nous rendre compte de celle-ci.
Dans le cas où on réentraîne tous les jours, on peut aussi vouloir faire un résumé des métriques. Ainsi, on peut afficher la valeur la plus haute, la plus basse, la moyenne et la médiane de la semaine.

Faire de la validation de données

Comme nous l'avons dit, les données sont toujours plus ou moins le déclencheur d'un model drift. C'est parfois dû au fait que les données pour une journée précise sont mauvaises. Il se peut qu'un champ non nullable soit devenu nullable. C'est pourquoi il est important de faire de la validation de données avant d'utiliser celles-ci. On peut le faire avec du code. On vérifie simplement que la donnée est bien celle attendue. Il existe des librairies dédiées à ce genre de tests. Pour en citer une et pour l'exemple, nous avons par exemple DeeQu. Il s'agit d'une librairie Spark qui fait de la validation de donnée. Ce qu'elle apporte, ce sont des fonctions déjà prêtes pour réaliser cela. Pour donner des exemples, nous avons des méthodes qui permettent de tester :
- Le fait qu'un champ ne contienne pas de valeur nulle
- L'unicité d'un champ
- Le rang des valeurs dans un champ
- Le nombre de lignes
- L'ordre et le nom des colonnes
- Le rang des valeurs de la médiane, moyenne, minimum, maximum ou percentiles d'un champ

L'inconvénient, c'est qu'il faut savoir à quelles données on s'attend. Souvent, on croit savoir. C'est là l'un des pièges de la donnée, surtout quand elle est volumineuse. Elle est souvent si peu conforme à ce à quoi on s'attendait. Il n'est pas rare d'avoir oublié une validation. Certains outils de validation de données comme DeeQu ou Great Expectations proposent aujourd'hui des systèmes de profilage. Vous indiquez un chemin à l'outil et il est capable de vous délivrer les grandes tendances de vos données. Ils sont même capables de générer les tests à dérouler pour un prochain extrait de données. Ils deviennent donc de plus en plus performants. Cela permet en partie de pallier les problèmes de la donnée fuyante qui passe son temps à nous surprendre. Cependant, s'appuyer uniquement là-dessus pour faire du monitoring de drift d'une manière générale est périlleux. Ce sont des éléments difficiles à maintenir. La donnée évolue. Il faudra faire évoluer les tests avec elle. Cela peut être fastidieux surtout quand on a un grand nombre de features. En fait, je recommande l'approche de tester ses données, mais davantage pour avoir en vue les grandes erreurs et incohérences. Que peut-on faire alors pour monitorer les changements de données dans le temps ? D'autres formes de monitoring et validation permettent de répondre à cela.

Monitorer le data drift

Avec une librairie comme DeeQu ou encore Great Expectations, on vérifie qu'un « id » est unique, qu'un champ n'a pas de nombre négatif, qu'un autre n'est jamais null, etc. Mais on ne monitore pas le changement qui peut avoir lieu dans nos données, ce changement qui peut avoir lieu avec les saisons, les tendances, etc.
On peut donc monitorer le data drift. On peut par exemple choisir des données de référence et comparer celles-ci avec les nouvelles qui nous parviennent. Il y a plusieurs manières de réaliser cela. On peut le faire manuellement. Mais il y a aussi des outils qui permettent de faire cela. L'idée est de comparer la moyenne des températures d'un ensemble de données par rapport à des données antérieures. On peut préférer une médiane ou toute autre métrique qui fait sens.
En monitorant le data drift, on peut prévoir un model drift. Au moins quand celui-ci parvient, on a

une idée de pourquoi. On sait où chercher.
Aujourd'hui, il y a en réalité peu d'outils sur le marché, même si je prévois que cela viendra dans les années à venir. Le problème actuellement est aussi qu'il y a plusieurs manières de détecter le data drift. Monitorer le data drift est critique. C'est ce qui entraîne la performance du modèle à décroître. Dans le même temps, il est dur d'obtenir une solution satisfaisante pour deux raisons principales :
- Développer une solution soi-même est challengeant
- Il est laborieux de savoir ce qu'il vaut mieux faire : des mesures de distances statistiques ou des tests d'hypothèses avec une p-valeur.

Finalement le data drift apparaît quand il y a un tel changement dans les données que cela impacte le modèle. Ce que je nomme « data drift » a en réalité plusieurs synonymes comme « data shift ». Le mot a aussi plusieurs sous-noms en fonction de ce à quoi il renvoie. On peut avoir du « data drift » graduel, du data drift abrupt, etc. Il est important de comprendre qu'il y a plusieurs sortes de data drift et qu'ils ne veulent pas dire la même chose. Un data drift court et abrupt peut être dû à un problème dans les données. Un data drift qui dure peut être dû à une récession. Un data drift saisonnier, comme son nom l'indique, est dû au changement des saisons. Vous pouvez voir toutes ces nuances dans le papier *Characterizing Concept Drift (Caractériser le concept drift)* par Geoffrey I Webb, Roy Hyde, Hong Cao, Hai-Long Nguyen et François Petitjean.
Il y a donc deux manières principales de détecter le data drift qu'on trouve dans la littérature et les outils proposés. J'ai l'impression que les distances statistiques sont les préférées pour détecter une différence trop importante dans les distributions. On peut utiliser la distance Wasserstein, la distance euclidienne, etc. Parmi toutes celles-là, il faut bien en choisir une. « Hellinger distance » est la préférée du papier *Survey of distance measures for quantifying concept drift and shift in numeric data* (Étude des mesures de distance pour quantifier le concept drift et shift dans les données numériques) écrit par Igor Goldenberg et Geoffrey I Webb. Ils ont construit plusieurs tests et ont trouvé la « Hellinger distance » la plus adéquate parce qu'elle est robuste et donne une valeur absolue bornée entre 0 et 1. Il est à noter que ce papier concerne les valeurs numériques. On ne compare pas une distribution de valeurs continues comme on le ferait avec des valeurs discrètes.
Martin Schmidtz, l'auteur de l'article *How to detect drifting models (comment détecter du modèle drift)* explique qu'il ne pense pas les tests d'hypothèses utiles pour le data drift :
"A low p-value only means that we cannot confirm it. This does not mean that we can prove that these distributions are different ! Not very useful… » (Une petite p-valeur seulement signifie seulement qu'on ne peut pas confirmer l'hypothèse. Cela ne veut pas dire que les distributions sont différentes. Ce n'est pas très utile.) « So I asked myself—isn't there something else to use ? And there is—distance measures for distributions ! « Alors je me suis demandé – n'y a-t-il pas quelque chose d'autre qu'on puisse utilise. Mais si – les mesures de distance pour les distributions.
Le grand vainqueur semble donc être les distances statistiques.

Cependant, un papier m'a récemment marqué : *Monitoring and explainability of models in production (Monitoring et explainabilité des modèles en production)* écrit par Janis Klaise, Arnaud Van Looveren, Clive Cox, Giovanni Vacanti et Alexandru Coca. Les auteurs parlent de tests d'hypothèse avec p-valeur. J'ai eu une discussion avec l'un des auteurs et j'ai trouvé ces explications enrichissantes. Une distance entre deux distributions ne vous dit pas si vous devez vous sentir concerné. Il faut que vous ajoutiez un seuil pour recevoir une alerte. Or c'est exactement ce que

fait un test statistique. La p-valeur n'est certainement pas la meilleure des métriques. Mais il est plus facile de comprendre une p-valeur qu'une distance Wasserstein.

Pour répondre à la question : que faut-il mieux faire ? Des tests d'hypothèses ou calculer des distances statistiques ? Je ne sais pas. Le « data drift » au moment où j'écris ces pages est un champ encore ouvert de la recherche. C'est donc normal de ne pas savoir. Ce que je crois en revanche, c'est que monitorer le data drift est important peu importe le chemin emprunté. Après avoir fait le tour des différentes plateformes, on trouve des outils pour monitorer le model drift. MLFlow dont nous avons déjà parlé est très utile dans ce domaine. C'est plus difficile de trouver des outils pour le data drift. Il y en a, mais ils ont parfois des limites qui ne conviennent pas à tous les projets. Azure Machine Learning Studio Data Drift par exemple offre une belle interface. J'y vois deux inconvénients majeurs. La première est que le produit n'est pas open source et donc fonctionne un peu comme une boîte noire. La deuxième, c'est qu'on ne peut pas calculer le drift de plus de 200 features. Cela ne convient pas à tous les projets. La meilleure option que j'ai trouvée pour l'instant sur le marché, c'est Alibi-detect. Il s'agit d'une librairie qui réalise de la détection d'outliers et du data drift avec des tests d'hypothèse. Il y a donc un parti pris. Cependant, c'est un début. Je prédis que dans quelques années, de plus en plus d'outils, de solutions et d'échanges auront vu le jour sur le sujet. Il est peut-être encore un peu tôt pour avoir une solution clef en main. On en est alors réduit à devoir réaliser la solution soi-même. Dans ce sens-là, Alibi-detect aide. Cette dernière est basée sur le papier *Failing Loudly : An Empirical Study of Methods for Detecting Dataset Shift (Echouer bruyamment : une étude empirique des méthodes de détection de data shift)* écrit par Stephan Rabanser, Stephan Gunneman et Zachary C.Lipton. Ils ont effectué des tests avec le dataset MNIST et utiliser des tests d'hypothèse pour détecter le data drift. Ils utilisent aussi de la réduction de dimension pour avoir une solution pratique. Après différents tests, ils ont trouvé que les deux solutions suivantes étaient les plus efficaces :

- BBSDs pour la réduction de dimension + de multiples tests univariés (des tests de Kolmogorov-Smirnov agrégés via la correction de Bonferroni)
- UAE pour la réduction de dimension + test multivarié (Maximum Mean Discrepancy)

À la fin, ils ouvrent sur la possibilité de faire de nouveaux tests. Ils ont utilisé un dataset assez classique de classification d'images. Il serait utile de réaliser d'autres tests sur d'autres types de dataset.

C'est sur ce papier que se base Alibi-detect. Étant donné qu'il s'agit aujourd'hui du seul outil open source et gratuit disponible sur le marché et capable de traiter de la donnée volumétrique, c'est certainement un début. Ce ne sera sans doute pas suffisant si vous voulez davantage explorer vos données. Il vous faudra sans doute ajouter quelques analyses et dashboards.

Comparer les prédictions avec la réalité

Une autre manière de monitorer le model drift, c'est tout simplement de comparer les prédictions qu'on réalise avec ce qu'il se passe vraiment dans la réalité. Ce que cette technique a de séduisant, c'est qu'elle permet d'évaluer l'impact business. Les prédictions que nous réalisons sont-elles pertinentes ? Cela va finalement au-delà de monitorer le model drift. C'est sans doute une solution très attractive. Elle permet de savoir ce que valent nos modèles. Elles nous donnent de précieuses indications. Nous sommes plus à même de savoir s'il faut réentraîner nos modèles plus tôt ou investiguer parce que le modèle ne donne plus rien d'intéressant.

Il est cependant nécessaire de noter que cette méthode bien que séduisante n'est pas toujours la plus évidente. C'est dommage parce que comme nous l'avons vu c'est celle qui permet de montrer le plus l'impact sur la réalité.
Il y a là aussi plusieurs manières de réaliser cela. Il s'agit d'abord de collecter les prédictions que l'on a faites. Il faut aussi s'assurer que ces prédictions sont bien utilisées. Enfin, nous collectons ce qu'il s'est vraiment passé. Par exemple, si nous prédisons des pannes, il nous faut collecter quelles pannes nous avons prédites. Il serait aussi bon de savoir quelles prédictions ont donné lieu à une intervention ou non. Et enfin, il s'agit de savoir si le technicien a bien repéré une panne potentielle ou si en l'absence d'intervention, la panne s'est bien déclenchée.
Ensuite, il y a plusieurs manières de réaliser cette comparaison et de sonner la sonnette d'alarme. Cela dépend du contexte, des données, de ce que l'on cherche à prédire et de la criticité du système.

Des dashboards et des alertes

Pour monitorer, on va souvent construire des dashboards. Ainsi, on représente la performance du modèle dans le temps par exemple. C'est très utile parce qu'agréable à prendre en main. Les graphiques ont cet avantage certain d'être attrayant. Les images ont un pouvoir attractif que n'ont pas toujours les mots. C'est pour cela que lorsque je pense au monitoring, je pense d'abord aux dashboards. C'est un bon début de penser aux dashboards. Mais ce n'est pas pour autant par là qu'il faut commencer. Tout du moins, si on le fait, il ne faut pas perdre de vue l'idée de construire un système d'alerte. En effet, il y a peu de chance pour que l'équipe s'évertue à regarder les détails des dashboards tous les jours pour vérifier qu'il n'y a pas eu d'erreur. En réalité, j'ai travaillé dans des équipes qui fonctionnaient ainsi. Cela marchait plutôt bien. Je ne peux cependant que me méfier de ce genre de situations. Le problème, c'est que cela demande de la rigueur. Celle-ci doit être déployée par des humains. Or on sait que l'humain est faillible. Il risque d'oublier un jour de faire son tour de vérifications habituel. Que peut-on faire alors pour pallier cela ? On peut mettre en place un système d'alerte. Il s'agit en fait de recevoir des notifications quand quelque chose ne se passe pas comme prévu. En réalité, je ne pense pas que cela permette de se passer de regarder les dashboards et de vérifier l'état du système d'une manière générale. Mais cela permet de pallier les plus gros problèmes.
Ces notifications peuvent prendre plusieurs formes. Souvent, il s'agit d'emails, d'envois dans les systèmes de messagerie tels que Slack ou Teams. Enfin, il peut aussi s'agir de SMS. Avec certains outils, on peut d'ailleurs mettre en place une granularité des alertes plus ou moins importantes et les desservir sur différents canaux en fonction de la criticité. Par exemple, si l'alerte est maximale, on envoie un SMS. Si elle est moyenne, ce sera un mail. Enfin les alertes mineures peuvent être reçues dans Slack, Teams ou autre.
Ces alertes doivent aussi être calibrées pour ne pas importuner, mais alerter. Il est parfois difficile de trouver le juste milieu. Cela dépend des contraintes business, de la criticité et disponibilité des personnes présentes. Généralement, ces deux derniers éléments vont de pair. Jauger les alertes ne se fera sans doute pas lors du premier jet. Il faudra peut-être plusieurs coups d'essai avant d'avoir des alertes justes et justifiées. Je conseillerais de commencer par trop d'alertes que pas assez. Bien sûr, il faut quand même juger le « trop d'alertes » pour éviter que tout le monde se détourne du système d'alerting très rapidement.
Enfin, les alertes peuvent être placées à différents endroits et mener à différentes réactions prévues

par l'équipe. Veut-on juste une solution de repli ? Dans ce cas, il faut la prévoir. On peut aussi la mettre en place automatiquement. Faut-il investiguer dès que le problème survient, peu importe l'heure du jour ou de la nuit ? Combien de temps pouvons-nous survivre avec le problème ? Toutes ces questions sont spécifiques aux équipes. Il est utile d'y penser avant que le problème arrive.

Nouveaux problèmes

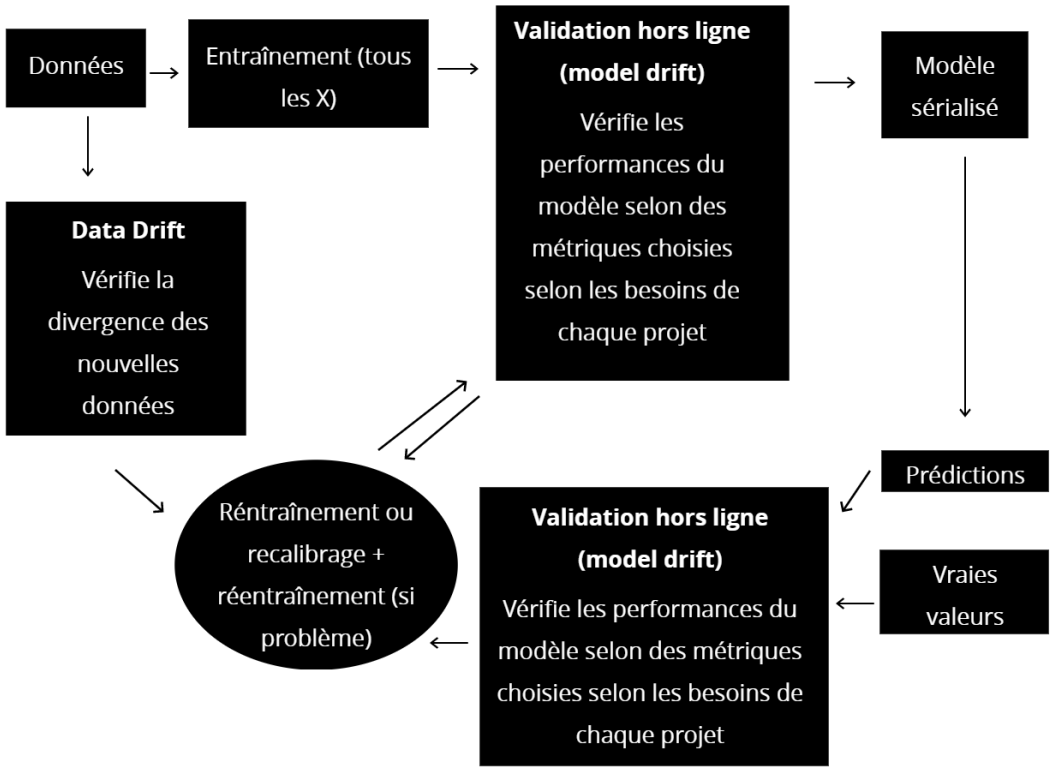
Le model drift existe-t-il vraiment ?

Le model drift est un phénomène assez courant qui arrive quel que soit le type de machine learning avec lequel on travaille. Il arrive plus au moins rapidement en fonction du système. Mais aucun modèle ne peut présenter les mêmes performances dans le temps. Un modèle lié à la mode évoluera plus vite qu'un modèle lié à des pannes électriques. Mais il y a fort à parier que ce dernier évoluera quand même. Le model drift n'est pas quelque chose qui apparaît dans l'heure qui suit. Cela peut arriver un an après. Si on ne le monitore pas, cela crée une erreur silencieuse. Il n'y a rien de pire pour un système que des erreurs silencieuses.
Le model drift est donc l'affaire de tous. Ce n'est pas un concept d'élite ou réservé à seulement quelques types de modèles. L'investigation une fois qu'un model drift est détecté n'est pas toujours longue. Il suffit parfois de réentraîner avec des données plus fraîches ou de changer un hyperparamètre.

Quel type de monitoring choisir pour vérifier le model drift ?

Nous avons vu qu'il y a plusieurs manières de monitorer le model drift : vérifier la qualité des données, vérifier la variation de celles-ci dans le temps, réentraîner, vérifier les résultats du réentraînement et comparer les prédictions avec la réalité. Toutes se complètent et ont leur intérêt. Certaines sont plus faciles que d'autres à mettre en place. Dans l'ordre, on pourrait citer : réentraîner, vérifier le résultat du réentraînement, vérifier la qualité des données, surveiller le data drift et comparer les prédictions avec la réalité. Certaines sont plus proches de l'impact réel qu'a le model sur la réalité. Dans l'ordre, on pourrait citer : comparaison avec la vie réelle, vérifier le data drift, vérifier le réentraînement, vérifier la qualité des données et réentraîner.
On peut en choisir une ou plusieurs en fonction de la criticité de son système, mais aussi de la maturité du projet et de l'équipe. Étant donné que pour beaucoup de systèmes, on sait que le model drift n'apparaît pas immédiatement, on peut commencer par les choses les plus simples à mettre en place pour une première mise en production et itérer jusqu'à avoir un système de monitoring satisfaisant.

On peut imaginer un système final comme celui-ci :



Les données sont continuellement monitorées avec du data drift. S'il y a un problème, on réentraîne ou on recalibre et on réentraîne. On réentraîne à intervalle régulier et on vérifie le résultat de ces entraînements. S'il y a un problème, on revient sur le modèle. Cela nous produit un modèle sérialisé qu'on utilise pour faire des prédictions. On compare ces prédictions avec les vraies valeurs afin de vérifier qu'il n'y a pas d'erreur. Si on rencontre un problème dans cette phase, on recalibre et réentraîne le modèle. On a ainsi différentes validations : du model drift hors ligne au moment de l'entraînement, du model drift en ligne en vérifiant dans la réalité les performances du modèle. Enfin, les données sont elles-mêmes soumises à du monitoring pour vérifier le data drift. Cela nous permet d'anticiper de mauvaises prédictions et aussi de comprendre la source d'un model drift.

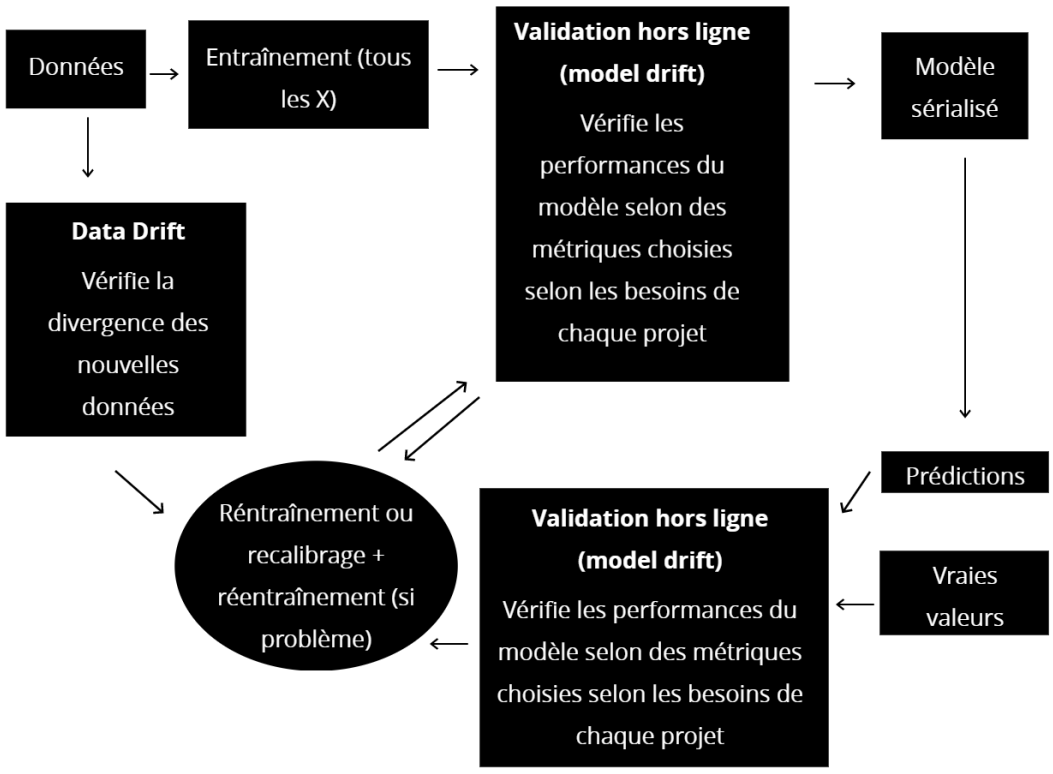
Pour conclure, le model drift arrive du fait d'un changement dans les données. Celui-ci peut intervenir parce que la cible ou les sources évoluent. Il y a fort à parier que tout système de machine learning change. Ainsi, le model drift est quelque chose d'assez courant. Mieux vaut s'en prémunir. Le monitoring peut avoir un côté rébarbatif parce qu'il semble ne pas apporter de plus-value directe. Pourtant, on sait que la maintenance d'un produit coûte cher. Le monitoring permet de réduire ses coûts. Il y a ensuite plusieurs manières de monitorer le model drift plus ou moins simples et plus ou moins proches de l'évaluation de l'impact business.
Il y a différentes formes de model drift. Celui-ci peut être abrupt et provenir d'une anomalie ou d'une soudaine pandémie. Il peut aussi être graduel quand les données évoluent doucement. Il peut être réparé en réentraînant ou en recalibrant le modèle. Parfois, c'est le moment de découvrir qu'une feature n'est plus intéressante ou qu'il faut en ajouter une autre.
Réentraîner est donc une première manière de se protéger du data drift, mais ce n'est pas suffisant.

Savoir détecter le data drift est aujourd'hui une question scientifique en suspens. Cependant, il vaut mieux essayer quelque chose qu'attendre de voir son modèle voler en éclats.

Je finirais cette partie par les conseils suivants :

- Commencer par les choses les plus simples à mettre en place pour une première mise en production
- Commencer par mettre en place des choses simples comme des logs, puis ajouter des alertes et dashboards
- Itérer jusqu'à obtenir un système de monitoring de model drift satisfaisant
- S'aider d'outils ou librairie existants pour accélérer cette mise en place
- Adapter les différentes techniques aux besoins de son produit

J'avais promis de revenir plusieurs fois sur le schéma de notre architecture général. De nouveau, je l'affiche. J'espère que cette fois-ci, il est plus clair. Nous voyons qu'à l'aide d'un gestionnaire de versions, nous lançons des codes de tests pour déployer l'inférence et l'entraînement. À partir de là, on peut générer des prédictions et mettre en place tout un système de monitoring.



Ce monitoring étant réalisé, on peut aller plus loin dans le profit qu'on peut tirer de notre système. C'est le moment de passer à l'échelle et de traiter de la forte volumétrie de données.

Passer à l'échelle

Concepts et raisons pour passer à l'échelle

Big Data

Il y a eu beaucoup d'améliorations dans le machine learning ces dernières années. Je pense notamment au transfer learning. Il s'agit du fait d'utiliser des modèles préentraînés. On peut alors finir l'entraînement sur de plus petites données. Ce genre de techniques a rendu moins primordiale l'idée de « big data ». Ce terme qui était un buzzword paraît parfois un peu dépassé. Pourtant, on continue à en avoir besoin pour entraîner un modèle sur de fortes volumétries. Dans ces cas-là, un simple algorithme avec Scikit-learn ne suffit pas. Il faut trouver des manières de digérer toute cette donnée sans faire exploser son serveur et dans un temps acceptable. Le big data retrouve alors ces lettres de noblesse.
Le big data est en fait un terme qui renferme bien plus de choses que le simple fait de se confronter à une forte volumétrie. On a pu définir le big data comme le fait de gérer des données volumineuses, variées et avec un besoin de rapidité. Ici, ce qui m'intéresse, c'est la volumétrie. La question pour ce chapitre est comment entraîner ou prédire sur de fortes volumétries ? Notons tout d'abord que tous les systèmes de machine learning n'ont pas affaire à ce problème. Mais quand celui-ci arrive, quelles sont les solutions pour le résoudre ?

Scalabilité verticale et scalabilité horizontale

Une première manière de réaliser cela est en passant par de la stabilité verticale. Cela signifie qu'on va augmenter les capacités de la machine sur laquelle on réalise les calculs. On peut ajouter de la RAM, du CPU, voire changer pour du GPU. On développe de manière classique quand on est en scalabilité verticale. C'est le gros point positif. Le problème avec la scalabilité verticale, c'est que quand on veut rajouter de la RAM, cela peut être difficile et long.
C'est pourquoi une réponse plus appropriée est la scalabilité horizontale. Il s'agit non pas d'ajouter des capacités à une machine seule, mais de faire travailler plusieurs machines entre elles. C'est ce qu'on appelle un cluster. On effectue alors des calculs sur plusieurs machines qui travaillent en parallèle et finissent par nous donner un résultat. On parle aussi de systèmes distribués. La personne qui développe avec un tel système ne se rend pas forcément compte des rouages de tout cela. Pourtant, c'est quelque chose à considérer quand on est face à ce genre de systèmes. En effet, quand on développe avec un tel système, on ne s'en rend pas toujours compte. C'est malheureusement moins vrai quand un bug apparaît.

Scalabilité horizontale des données

En réalité, il y a plusieurs choses qu'on peut rendre scalable. Pour commencer, il y a les données. Nous avons beaucoup entendu parler de data lake ces dernières années. Les data lakes sont très souvent construits sur des systèmes distribués. Leur but est de centraliser de la donnée. Il existe différents

outils pour distribuer la donnée. HDFS est sans doute le plus connu. C'est un outil qui appartient à la suite Hadoop. Hadoop dans l'ensemble a pour but de faire du calcul distribué avec un système de données distribuées. Mais Hadoop n'est plus l'outil préféré en data. HDFS reste beaucoup utilisé, mais il a de plus en plus de concurrents. Je pense par exemple à DBFS qui est le système de données distribué créé par Databricks.

Comment ces systèmes distribuent-ils la donnée ? Supposons un fichier data.csv qui a les colonnes animal, nom et âge. Ce fichier a les lignes suivantes :

```
chat, Snoopy, 7
poisson, Nemo, 1
manchot, Pinguee, 2
```

Ces lignes vont être distribuées sur le cluster, c'est-à-dire un ensemble de machines. Supposons que nous ayons trois machines en fonctionnement : machine A, B et C. Le système divise le fichier. Snoopy sera sur la machine A, Nemo sur la machine B et Pinguee sur la machine C. La donnée est distribuée. Du point de vue de l'utilisateur, c'est transparent. Quand il demande à avoir accès au fichier *data.csv*, on lui donne l'ensemble du fichier sans lui préciser d'où les éléments viennent.

Mais pourquoi distribuer la donnée ? L'un des avantages de séparer la donnée en plusieurs éléments qui va beaucoup nous intéresser, c'est alors qu'on peut effectuer des calculs en parallèle.

Scalabilité horizontale du calcul

La scalabilité horizontale du stockage est en fait liée à la scalabilité horizontale du calcul. Supposons que nous voulons faire un calcul qui nous donne la date de naissance de chaque animal. Nous sommes en 2020. Snoopy est donc né en 2013, Nemo en 2019 et Pinguee en 2018. Pour réaliser ce calcul, on peut le diviser. Un calcul aura lieu sur la machine A où se trouve Snoopy, un autre sur la machine B où est Nemo et un dernier sera sur la machine C où se trouve Pinguee. Cela se fait en parallèle et augmente donc la capacité de calcul. Là, nous n'avons que trois animaux. Ce n'est pas très intéressant. Il y a même fort à parier que c'est moins performant que de faire le calcul sur une seule machine. Il y a un coup dans le fait même de distribuer le calcul et d'agréger les données finales. Ce coût est négligeable quand on traite d nombreuses lignes. Avec des problématiques de big data, cela devient plus intéressant.

Les solutions pour passer à l'échelle

Une fois l'intérêt du passage à l'échelle compris, il y a plusieurs solutions pour réaliser cela.

Des frameworks dédiés au passage à l'échelle

Nous avons vu qu'il y a des logiciels dédiés pour distribuer les données : HDFS ou DBFS par exemple. Pour le calcul distribué, Hadoop MapReduce qui comme son nom l'indique appartient à la suite

Hadoop. Il a longtemps régné en maître. Mais Hadoop MapReduce n'avait aucune facilité pour le machine learning. Il permettait davantage de faire du data processing. Il n'était pas non plus capable de travailler en mémoire ou de faire du streaming. Puis est arrivée l'ère de Spark. Spark est un outil davantage spécialisé dans le data processing et le shallow learning, c'est-à-dire l'apprentissage peu profond. Autrement dit, il n'est pas très bien équipé en ce qui concerne le deep learning. TensorFlow est par exemple une bibliothèque spécialisée dans le deep learning qui intègre son propre moteur de distribution.
Il y a donc plusieurs frameworks de machine learning dédiés à la distribution. Ces frameworks, notamment Spark avec lequel j'ai beaucoup travaillé, sont pensés pour distribuer le calcul. Il faut parfois jongler entre plusieurs pour arriver à ses fins : un peu de Spark pour faire du data processing, puis un autre logiciel pour faire du deep learning.

Des logiciels pour distribuer n'importe quel calcul

Il existe aussi des logiciels qui proposent de distribuer n'importe quel calcul. Je pense notamment au logiciel Ray. Cet outil permet de distribuer du Python. Il a aussi des possibilités en Java, mais étant donné que Python est bien davantage utilisé en machine learning, ce n'est pas cette partie qui nous intéresse. Ray permet de distribuer n'importe quel code Python. Cela ne s'arrête donc pas à du Scikit-learn, mais va jusqu'à du Pandas, du numpy ou n'importe quel code Python. Cela ne signifie pas que ce soit évident et qu'il ne faille pas fournir un certain effort pour y parvenir. Ray est un framework qui offre de grandes possibilités dont il faut savoir se servir pour distribuer son code.

Les difficultés du passage à l'échelle

La difficulté des systèmes distribués

Les systèmes distribués offrent de grands avantages. En big data, le temps de calcul est réduit. Cela rend parfois juste possible le fait d'effectuer un calcul. Une scalabilité horizontale est plus simple à mettre en place qu'une scalabilité verticale. Cependant, ce n'est pas forcément évident. C'est pourquoi Ray ou Spark par exemple viennent avec des outils de monitoring inclus dans l'installation même. Même si beaucoup de choses sont transparentes, des bugs peuvent survenir en raison même du caractère distribué du système. C'est pourquoi c'est une forme de développement dont il faut considérer les difficultés. Ce serait dommage d'en avoir peur et de s'en priver. Mais ce serait aussi dommage de ne pas avoir conscience des problèmes qu'on risque de rencontrer.

Choisir une stratégie de développement

Il y a plusieurs manières de passer à l'échelle. Une stratégie est de commencer à développer de manière non distribuée un prototype. Quand celui-ci est validé et qu'on veut passer celui-ci en production, on pense alors à l'étape suivante qui est de distribuer le code. Cela revient souvent à réécrire ce dernier. C'est beaucoup d'effort et de perte de temps pour le résultat final. Développer sans arme de manière distribuée n'est pas évidente. C'est pourquoi il ne faut pas négliger les formations sur le sujet. Cela rendra les choses plus simples.

Dans certains cas, on peut quand même vouloir faire un prototype le plus rapidement possible pour le valider le plus rapidement possible. Développer de manière distribuée ne permet pas d'être rapide. Cela devient plus simple si on y est formé et habitué, d'où l'idée des formations. D'une manière générale, je recommanderais d'avoir des prototypes le plus proche possible de ce qui est censé partir en production ou alors de les faire durer le moins possible. Le smell « prototype » est un phénomène courant en data science. Emmener du code en production qui ressemble davantage à du code de prototype n'est pas une manière sereine de vivre un projet.

Pour conclure, le machine learning n'a peut-être pas autant besoin qu'autrefois de big data pour exister. Cependant, il y est souvent confronté. Ce buzzword aujourd'hui moins attrayant a encore de beaux jours devant lui. Savoir le traiter est souvent une compétence à avoir dans son bagage de data scientiste. Il existe différentes formes de scalabilité : horizontale et verticale. La scalabilité verticale renvoie au fait d'augmenter les ressources de calcul et de stockage en augmentant les moyens de la machine. La scalabilité horizontale est un système qui travaille en cluster. Cela signifie qu'au lieu de travailler sur une seule machine, on travaille avec un groupe de machines qui agissent ensemble. De cette manière-là, quand on a besoin d'ajouter de la puissance de calcul, il suffit d'ajouter des machines.
La scalabilité horizontale est à préférer. On n'a pas besoin de changer la RAM d'une machine par exemple. Il suffit d'en ajouter une. De plus, en fonction du calcul, on peut utiliser une ou plusieurs parties des machines à notre disposition.
J'englobe dans le terme de scalabilité deux concepts en réalité : celui de scalabilité des données et de calcul. Le big data renvoie au fait de devoir stocker de gros volumes de données, mais aussi de devoir réaliser des calculs sur celles-ci. Ce sont deux aspects différents, mais fortement liés.
Pour faire de la scalabilité horizontale, on va découper les fichiers et les distribuer sur différentes machines. Pour l'utilisateur, cela est transparent, mais cela nous permet de stocker la donnée. On parle de distribution des données.
La scalabilité du calcul renvoie au fait que plus on ajoute de machines, plus le calcul se fait sur cet ensemble de machines et gagne en puissance. On parle de distribution du calcul.
Spark est un des grands frameworks qui permet cela. Aujourd'hui, on a aussi des logiciels comme Ray qui permettent de distribuer toute sorte de calcul et non pas seulement ce qui est contenu dans les APIs Spark.
Il est plus dur de développer de manière distribuée. Il vaut mieux s'y préparer et ne pas se faire surprendre. Même si les frameworks cachent pour nous une grosse partie des spécificités internes, il reste que beaucoup d'erreurs sont liées à la distribution.
Il y a plusieurs stratégies pour gérer cela dans un projet de data science. On peut commencer par développer sur un échantillon sans big data, puis passer à l'échelle une fois en production. Je recommanderais cependant de s'y confronter le plus tôt possible pour éviter les surprises.

Mes conseils, en résumé, seraient les suivants :
- Savoir développer avec un ou plusieurs frameworks dédiés aux systèmes distribués
- Comprendre à la fois le fait de distribuer ses données et de distribuer son calcul
- S'engager dans des formations. Développer de manière distribuée est complexe.
- Éviter le smell « prototype » en développant dès le début du projet de manière distribuée quand on sait qu'on va en avoir besoin.

- D’une manière générale, faire durer les prototypes le moins longtemps possible.

Conduire le succès d'un système prédictif

Donner confiance dans le modèle

Donner confiance dans le modèle en fonction du contexte est un élément optionnel d'une mise en production. Pourtant, c'est un élément qui peut être important. On a critiqué les modèles de machine learning et notamment de deep learning. On leur a reproché d'être biaisés, discriminants, dangereux, etc. Au-delà de ces reproches, ce qui perturbe c'est l'absence de transparence.
Que font ces boîtes noires ? C'est un sujet qui fait de plus en plus parler de lui.

Pourquoi donner confiance aux utilisateurs dans le modèle ?

La tentation d'utiliser le machine learning avec parcimonie

Les raisons pour comprendre un modèle sont multiples. Il y a d'abord le besoin éthique. C'est ce qui est le plus souvent mis en avant. Par exemple, les personnes qui se voient refuser un prêt veulent savoir pourquoi.
Une manière de se protéger de ces méfaits est peut-être d'utiliser ces modèles avec parcimonie. Le journal le New York Times utilise un système d'apprentissage automatique pour modérer les commentaires des articles. Ils ont assez rapidement découvert des biais. Le modèle discriminait les minorités. Afin de pallier cela, ils ont décidé de l'utiliser avec modération. Une équipe de modérateurs humains est toujours en place et prend les décisions finales. C'est ce que l'équipe du New York Times raconte dans son article intitulé *To Apply Machine Learning Responsibly, We Use It in Moderation (Pour utiliser du machine learning de manière responsable, nous l'utilisons avec parcimonie)*. C'était sans doute inévitable pour le New York Times. Dans d'autres contextes, il est possible d'éviter la tentation d'utiliser le machine learning avec parcimonie.

Des boîtes plus ou moins noires

Une autre solution peut être de sortir de la boîte noire. Celle-ci n'est pas toujours totale en réalité. On sait qu'il est relativement facile d'expliquer les modèles linéaires tels que les régressions linéaires ou les régressions logistiques. Les formules mathématiques derrière sont assez simples. De plus, on peut en déduire des graphiques qui rendent le tout très visuel.
Les arbres de décisions aussi sont plutôt simples à comprendre. Il s'agit finalement d'un ensemble de « si », « alors ». Les personnes l'apprécient souvent. Il est à même de rendre facilement compte d'une pensée humaine. De la même manière, on peut envisager des visuels assez parlants.

Cela se complique un peu avec les modèles ensemblistes tels que RandomForest, AdaBoost ou GradientBoosted Tree par exemple. Il devient difficile d'expliquer rapidement un modèle ou de le représenter graphiquement.
Les choses deviennent encore plus difficiles avec les réseaux de neurones. Un simple réseau de neurones avec une couche et peu d'entrées est facilement explicable. Mais la plupart du temps, si on utilise des réseaux de neurones, c'est pour aller plus loin. On cherche généralement à faire de l'apprentissage profond. On multiplie ainsi les couches.
On peut considérer que les modèles linéaires et les arbres de décisions sont des boîtes blanches. Les réseaux de neurones profonds sont des boîtes noires. Quant aux algorithmes ensemblistes, ils sont dans un intermédiaire.
Dans tous les cas, la transparence est de plus en plus demandée.

Des raisons diverses : éthique, sécurité, commercial

Tout le monde ne réclame pas de comprendre la boîte. Certains utilisateurs n'en ont que faire. D'autres au contraire y sont très attachés. Parfois, c'est le business qui veut comprendre. Afin d'être sûr de ce qui part en production, il apprécie de plus amples explications. D'autres fois encore, ce sont les data scientistes qui s'intéressent à cette partie.
Il y a plusieurs raisons à cela. Elles peuvent se compléter. Comme je l'ai dit, celle dont on entend le plus souvent parler est l'éthique. Plus on comprend la boîte, plus on a des chances de la maîtriser et d'éviter ses défauts.
Une autre raison est celle de la sécurité. Supposons un algorithme permettant de prédire des cancers. Afin d'être sûr de l'utiliser à bon escient, il vaut mieux le comprendre et ne pas l'utiliser comme un outil magique. Les risques sont importants dans ce genre de cas.
Avec ces exemples, on pourrait se dire que sortir de la boîte n'est une nécessité que dans certains cas bien définis. Pourtant, il y a d'autres raisons pour vouloir comprendre un modèle Ces raisons sont proches du quotidien de plus de personnes qu'on ne pourrait le croire.
Donner une explication du modèle à l'utilisateur peut être un argument commercial. Un système de recommandation peut éclairer l'utilisateur en lui expliquant les raisons de sa recommandation. L'utilisateur est alors plus enclin à croire le système. Il lui fait davantage confiance.

La confiance, une raison clef

La confiance est une raison importante qui peut pousser plus d'un à vouloir sortir de la boîte. Je souhaiterais raconter l'histoire d'un système de machine learning assez banal pour lequel comprendre le modèle était important.
Supposons un système qui prédit des pannes d'ascenseur. Le système n'est pas vraiment une boite noire. Il ne s'agit pas d'un réseau de neurones. Cependant, il ne s'agit pas non plus d'une régression linéaire. Nous avons affaire à un RandomForest. Nous avons deux événements : le temps que l'ascenseur passe entre deux étages et le temps passé entre une ouverture et une fermeture de portes. Toutes ces données sont les entrées de l'algorithme. Quand le système prédit une panne, pour inciter les techniciens à agir, il s'agit de leur donner confiance en la prédiction. Il est plus facile d'intervenir quand la panne est là que quand on suppose qu'elle aura lieu.

Imaginez-vous par exemple en train d'essayer de résoudre un bug supposé arrivé. C'est challengeant. L'idée est d'expliquer cette boîte grise. Par exemple, vous découvrez que la panne est prédite parce que le temps moyen de l'ascenseur entre deux étages excède deux secondes. Vous pouvez donner cette explication aux techniciens. Cela ne permet pas de tout comprendre. Mais c'est déjà un début. Nous reviendrons sur les limites de cette histoire. Ainsi, rendre compte de ce qui se passe dans la boîte sert la confiance des utilisateurs. Dans certains cas, elle sert aussi le business responsable de délivrer le produit. Lui aussi a besoin d'avoir confiance.

Sortir de la boîte noire pour débuguer et améliorer son modèle

Les data scientistes eux-mêmes dans leur travail quotidien peuvent rechercher cette compréhension. En effet, afin d'améliorer un algorithme, il vaut mieux le connaître un minimum. Cela peut donc servir au debug et à rechercher un modèle plus intéressant.
Pour résumer, les raisons pour vouloir expliquer un modèle sont multiples : commercial, éthique, sécurité, confiance, technique, etc. Il n'est pas rare de voir des modèles en production sans cela. Cependant, beaucoup de projets peuvent être amenés à en avoir besoin. Il ne s'agit pas d'un outil destiné à l'armée, aux médecins ou à Google.

Solutions pour donner confiance aux utilisateurs dans le modèle

Des façons simples de sortir de la boîte

Les critiques envers le machine learning grandissantes, plusieurs outils ont vu le jour afin d'y remédier. Ils permettent de donner de plus amples explications. Nous y reviendrons plus tard. Dans cette partie, je voudrais davantage mettre l'accent sur le fait que bien avant ceux-ci, les data scientistes cherchaient déjà à expliquer leurs modèles. Il y a plusieurs moyens d'y parvenir.

Privilégier les modèles simples

Une première manière de réaliser cela est de s'interdire les modèles trop compliqués. Souvent, la mesure pour choisir le meilleur algorithme est la justesse, la précision, l'AUC, etc. Ces métriques servent en fait à évaluer la capacité d'un modèle à faire une prédiction. Dans certains cas, la meilleure métrique à prendre en compte est parfois la compréhension de l'algorithme.
C'est pourquoi certains projets peuvent s'interdire d'utiliser des réseaux de neurones ou d'autres types de modèles. Elles se privent de meilleurs produits plus à même de réaliser des prédictions correctes. Elles mettent davantage l'action sur la compréhension.
D'autres projets ne s'interdisent pas de modèle, mais utilisent des modèles linéaires pour expliquer et rendre visuels les structures présentes au sein des données. Elles ont ainsi plusieurs modèles qui cohabitent : l'un sert à être précis dans les prédictions, le second a davantage pour objectif de rendre ces dernières compréhensibles.

Utiliser des outils visuels

Enfin, il existe des outils visuels qui tentent d'expliquer jusqu'aux réseaux de neurones. Le Grand Tour peut rendre compte de la manière dont le modèle apprend. Le Grand Tour est une technique de data visualisation. Elle a été développée par Daniel Asimov en 1985. Il s'agit d'une animation qui ressemble à un film. Cette méthode peut être utilisée pour comprendre les itérations d'apprentissage dans un système de réseaux de neurones par exemple. Supposons qu'on lui donne le dataset de MNIST. Ce dernier est une base de données qui comprend des images des chiffres de 0 à 10. Tous ont été écrits à la main. L'idée est de prédire à quel chiffre correspond quelle image. Le Grand Tour permet de voir dans ce cas que les chiffres 1 et 7 sont difficiles à départager et que le modèle passe plus de temps à apprendre sur ces éléments-là que sur d'autres.
Cela n'explique pas tout du modèle, mais donne un point de départ intéressant.
Ainsi, on voit qu'il existe déjà des méthodes utilisées pour comprendre davantage les modèles que nous utilisons. Peut-être même avant que les critiques ne se fassent de plus en plus entendre, avions-nous déjà commencé à prendre cela en compte.

Interprabilité et explainabilité

Le domaine a évolué jusqu'à donner les concepts d'interprétabilité (interpretability en anglais) et explainabilité (explainability en anglais).
Comme l'explique Christoph Molnar dans son livre *Interpretable Machine Learning*, il n'y a pas de définition mathématique à l'interprétabilité. Une définition non mathématique a été donnée par Miller en 2017 : « l'interprétabilité est le degré auquel un humain peut comprendre la cause d'une décision (interpretability is the degree to which a human can understand the cause of a decision) ».
Une autre définition donnée par Kim, Been, Rajiv Khanna, et Oluwasanmi O. Koyejo en 2016 est : « L'interprétabilité est le degré auquel un humain peut systématiquement prédire le résultat d'un modèle (Interpretability is the degree to which a human can consistently predict the model's result) ».
Dans cette course à l'interprétabilité, plusieurs algorithmes ont vu le jour. On peut citer SHAP ou Lime. Dans le papier *Why should I trust you ? (Pourquoi devrais-je te faire confiance ?)*, les auteurs expliquent comment ils ont mis au point Lime pour permettre d'expliquer n'importe quel classifieur.
Un cas intéressant est par exemple celui des images. Ils ont donné à un classifieur des photos de chien et de loup. Le modèle permettait en effet de classifier les animaux correctement. L'algorithme Lime permet ensuite de dire quelles parties de l'image permettent de conclure à un certain type d'espèce. Il se trouve que ce ne sont pas les caractéristiques de l'animal même qui aboutissaient à la conclusion. Les loups apparaissaient beaucoup plus souvent dans la neige. La présence de cet élément climatique était ce qui influençait le résultat du modèle. Nous voyons là un biais dans ce dernier. Il nous a été possible de nous en rendre compte avec l'outil Lime.
Lime marche en modifiant les entrées d'un modèle. Cela permet de savoir quelles sont les entrées les plus importantes qui permettent de faire la prédiction. Dans le cas des images de chiens et de loups, on peut supprimer la neige. En faisant cela, on se rend compte que la prédiction n'est plus aussi bonne. C'est donc bien la présence de la poudre blanche sur la photo qui permet de conclure à tel ou tel type d'animaux.

Lime marche aussi pour le traitement naturel automatique du langage. Il permet de souligner les mots qui expliquent davantage le résultat du modèle. Il est agnostique et convient à n'importe quel classifieur. Il sait aussi mettre en avant les features les plus importantes d'un modèle. Il établit cela en brouillant les entrées et en analysant les résultats.
Ces algorithmes dont fait partie Lime sont intégrés dans des librairies en Python par exemple. Ils permettent de donner de l'interprétabilité à partir de n'importe quelle entrée. Ils sont divers et s'adaptent aux différents types de régressions et de classifications. L'avantage est que la librairie choisit l'algorithme le plus adapté en fonction du contexte. Pour ne donner qu'un exemple de librairie qui donne de l'interprétabilité, il existe Eli5. Le terme vient d'une expression américaine qui signifie : explique-moi cela comme si j'avais 5 ans. Cet outil fait appel en interne à Lime ou SHAP par exemple pour renvoyer de l'interprétabilité locale et globale.
Il y a ce qu'on appelle l'interprétabilité locale et l'interprétabilité globale. L'interprétabilité globale donne des informations générales sur le modèle. L'interprétabilité locale donne des informations sur une prédiction.
L'information globale sert par exemple davantage aux data scientistes et aux personnes du business. L'information locale sert à tout le monde, mais peut-être utilisée directement par l'utilisateur final. Cette interprétabilité permet d'apporter beaucoup de données et aide à mieux comprendre la boîte noire. Elle peut servir au debug.

Aller au-delà de l'interprétabilité : l'explainabilité

Cependant, ce n'est pas toujours assez. Revenons à l'exemple de prédictions de pannes d'ascenseur. Avec l'interprétabilité globale et un outil tel qu'ELI5, nous savons que quand le temps entre deux étages excède deux secondes, nous avons généralement une panne qui va survenir. Cela nous permet de débuguer et améliorer le modèle. Cela donne aussi quelques informations aux personnes du business. Nous comprenons tous mieux le modèle.
Avec l'interprétabilité locale, nous sommes aussi en mesure de comprendre pourquoi l'algorithme prédit spécifiquement une panne. C'est utile à tout le monde et peut être donné à l'utilisateur. Le technicien a ainsi beaucoup plus de contexte face à une panne. Il lui est a priori plus aisé de réaliser son intervention. Il sait où il va mettre les pieds.
Pourtant, la panne n'existe toujours pas quand il se rend sur les lieux de son travail. L'investigation risque d'être longue. Il peut interroger des usagers qui passent : « Avez-vous remarqué des problèmes avec l'ascenseur ces derniers temps ». On lui répondra « Non, rien d'anormal ». Ou alors « Il est de plus en plus lent ». Peut-être aura-t-il de la chance et quelqu'un ajoutera : « J'ai remarqué un son étrange derrière le miroir. Comme un cliquetis ». En bref, une enquête se met en place. C'est l'histoire d'une panne qui n'a pas encore eu lieu. Notons d'ailleurs que le machine learning n'est pas infaillible. En effet, il se peut que le système se trompe. Le technicien devrait donc réaliser son enquête sur une panne qui n'aura peut-être jamais lieu. Pour sûr, la tâche est ardue. Il faut s'armer de patience et de volonté. L'information donnée par l'interprétabilité n'est pas suffisante. Ce n'est pas magique.
Supposons que le technicien ait le choix entre deux interventions. L'une est une panne prédite par le système. Elle risque d'embêter tous les habitants d'un immeuble. On sait qu'il y a dans celui-ci des personnes handicapées qui ne peuvent pas sortir de chez eux sans un ascenseur. L'intervention est donc vivement conseillée. Cependant, il y a une autre panne. Un petit immeuble. Les habitants sont

peu nombreux et ne présentent aucune difficulté physique. Ils seront beaucoup moins impactés par l'incident. La différence, c'est que celui-ci est bien réel. La panne est avérée et a lieu en ce moment. Quel choix pourrait faire le technicien ? Entre une panne à venir avec d'importantes conséquences et un incident mineur, mais bien réel. Il pourrait vouloir choisir ce dernier. Au moins, il sait ce sur quoi il intervient. Son enquête est plus simple. Le crime a bien eu lieu.
Nous voyons donc que l'interprétabilité dans ce cas n'est pas suffisante. Que faudrait-il alors faire ? Notre but est premièrement de donner confiance au technicien. Le deuxième est de lui donner le plus d'éléments possible pour réaliser une enquête que nous savons compliquée.
La data science n'est pas qu'une histoire de machine learning. Nous disposons généralement de beaucoup de données pour réaliser nos systèmes. Celles-ci peuvent tout à fait servir à décrire davantage l'état de l'ascenseur. Avec de belles data visualisations, nous avons alors un produit qui prend forme.
Nous pouvons aller plus loin en nous servant cette fois-ci de l'apprentissage automatique. Nous pouvons par exemple utiliser ce dernier afin de définir les raisons les plus probables d'une panne. C'est la cerise sur le gâteau. Le risque que nous nous trompions n'est pas exclu. Cependant, nous sommes plus avancés dans l'explication de notre modèle.
Il y a en fait une différence entre l'interprétabilité et l'explainabilité. L'interprétabilité porte bien son nom. Son but est d'interpréter le modèle. La boîte perd son caractère mystérieux. Cependant, bien souvent, l'interprétabilité n'établit que des corrélations. Elle ne dresse pas nécessairement un lien de causalité. De plus, si les événements à l'origine de la prédiction sont difficilement interprétables en eux-mêmes, il n'est pas sûr qu'il soit d'une grande utilité.
Le concept d'explainabilité est différent. Son but est bien la cause pour le coup. Il s'agit de vraiment comprendre ce qui se passe. Dans le cas des pannes, on cherche la cause principale. On veut savoir les raisons de la panne. La question est : qu'est-ce qui dans la vie réelle est à l'origine de l'incident ? Nous notons en fait qu'il n'y a pas d'outil magique pour réaliser cela. Contrairement à l'interprétabilité, il n'y a pas de librairies orientées explainabilité.

Nouveaux problèmes

Ai-je besoin de donner confiance en mon modèle ?

L'interprétabilité et l'explainabilité sont des éléments qui peuvent pérenniser une mise en production et conduire à son succès. Ces concepts sont de plus en plus mis en avant. Cependant, on n'a pas toujours un utilisateur final ou quelqu'un du business qui le réclame. Le modèle peut aussi lui-même être assez simple et ne pas réclamer beaucoup de debug ou d'optimisation.
Or, même si le coût est moindre en ce qui concerne la mise en place de l'interprétabilité, il existe toujours. Quant à celui de l'explainabilité, il peut être assez onéreux.
L'idée est d'utiliser ce dont le produit a besoin. Si l'interprétabilité et l'explainabilité ne sont pas des prérequis, il n'est pas besoin de les ajouter. C'est toujours quelque chose à construire et maintenir.

Comment trouver la cause source de la prédiction ?

L'explainabilité n'est pas forcément évidente. Elle réclame souvent de l'investigation et du développement. Si on ne parvient pas à trouver la cause source de la prédiction, on peut toujours fournir

davantage de données. Cela permet d'avoir plus d'informations. Il faut cependant veiller à ne pas noyer la personne réceptrice de celles-ci.

Est-il pertinent d'afficher tel quel le résultat de l'interprétabilité à l'utilisateur final

Il n'est pas sûr qu'on veuille afficher l'interprétabilité et l'explainabilité de la même manière au business, à l'utilisateur final ou aux data scientistes. Pour ces derniers, on peut afficher telle quelle la sortie de l'outil. Pour le business, on peut réaliser un graphique assez simple. Enfin pour l'utilisateur final, cela peut être un visuel plus travaillé ou une explication textuelle mieux formulée.
L'outil permet d'arriver au résultat. Mais on n'est point obligé de l'utiliser tel quel.

Pour conclure, les raisons pour vouloir interpréter un modèle sont multiples : éthique, confiance, marketing, debug, etc. Contrairement aux idées reçues, on ne fait pas cela uniquement pour des raisons éthiques. On peut le faire pour des raisons très commerciales comme inciter davantage les utilisateurs à croire en la prédiction et acheter ce superbe nouveau produit.
Plusieurs personnes peuvent être intéressées par l'interprétation d'un modèle : utilisateurs finaux, business ou data scientistes. On l'oublie, mais c'est aussi un outil de debug pour mieux comprendre son modèle.
Il y a des modèles naturellement plus ou moins durs à interpréter. Ceux linéaires sont plus évidents que les réseaux neuronaux par exemple.
Afin donc de sortir de la boîte noire, on peut transformer des modèles complexes en modèles linéaires ou utiliser de la visualisation.
Il existe aujourd'hui des outils dédiés au fait d'interpréter les modèles complexes. Ils donnent des interprétations globales du modèle et locale des prédictions. On parle d' « interprétabilité ».
L'interprétabilité n'est pas toujours suffisante. Parfois, il faut avoir recours à de l'explainabilité. Celle-ci cherche la cause même de la prédiction. Elle est souvent à construire manuellement.

Je finirais par les conseils suivants :
• Se demander si on a besoin d'interprétabilité. Pour qui ? Pour quoi ?
• Se demander si un outil automatique peut y répondre ou s'il faut davantage se tourner vers de l'explainabilité

L'interprétabilité et l'explainabilité peuvent être des concepts clefs pour le succès d'un système prédictif. Un autre élément de réussite est sans contexte la manière d'organiser son équipe.

Organiser son équipe

Dans cette partie, j'aimerais parler des différentes stratégies qu'on peut mettre en place pour organiser son équipe quand on fait de la data science. Cela peut paraître sans corrélation avec le fait de mettre du machine learning en production. J'ai hésité au début à écrire ce chapitre. Cependant, aujourd'hui je suis convaincue que c'est intrinsèque d'une bonne mise en production. Je voudrais vous parler à la fois d'organisation d'équipe avec des concepts comme l'Agile par exemple ou des stratégies de déploiement comme le feature flipping ou encore l'AB testing.

S'organiser autour d'une méthode : Agile, CRISP ou TDSM ?

C'est un non-sens de parler d'Agile comme d'une méthodologie. Cependant, pour les besoins de ce chapitre, nous le rangerons dans cette catégorie. L'idée est donc de se demander face aux différentes méthodes d'organisation qui s'offrent à nous laquelle est la meilleure que nous puissions utiliser.

Agile

Retour aux origines

Je viens de classer l'Agile dans les méthodologies, mais d'entrée de jeu, je ne peux m'empêcher d'avoir envie de donner une meilleure définition. Il s'agit plutôt d'une manière de voir le développement. Il s'agit aussi d'un manifeste rédigé en 2001 par plusieurs développeurs ayant eux-mêmes créés plusieurs méthodes (Scrum et Extreme Programming par exemple) qu'on associe traditionnellement au monde Agile. Le manifeste dit cela :

« Nous découvrons comment mieux développer des logiciels
par la pratique et en aidant les autres à le faire.
Ces expériences nous ont amenés à valoriser :

• Les individus et leurs interactions plus que les processus et les outils
• Des logiciels opérationnels plus qu'une documentation exhaustive
• La collaboration avec les clients plus que la négociation contractuelle
• L'adaptation au changement plus que le suivi d'un plan

Nous reconnaissons la valeur des seconds éléments, mais privilégions les premiers. »

Il est suivi de 12 principes sous-jacents qui permettent à mon sens de l'éclairer :
« Nous suivons ces principes :
• Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.
• Accueillez positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client.
• Livrez fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts.
• Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.
• Réalisez les projets avec des personnes motivées. Fournissez-leur l'environnement et le soutien dont ils ont besoin et faites-leur confiance pour atteindre les objectifs fixés.
• La méthode la plus simple et la plus efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face.
• Un logiciel opérationnel est la principale mesure d'avancement.
• Les processus Agiles encouragent un rythme de développement soutenable. Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant.

• Une attention continue à l'excellence technique et à une bonne conception renforce l'Agilité.
• La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle.
• Les meilleures architectures, spécifications et conceptions émergent d'équipes autoorganisées.
• À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence. »

Une lecture qui est intéressante pour comprendre l'Agile, c'est celle de Robert C. Martin. Il s'agit d'un des signataires du manifeste qui a publié un livre intitulé *Clean Agile, Back to basics*. Je vais ici résumer les parties intéressantes pour l'ingénierie du machine learning. Robert C. Martin commence par revenir sur l'histoire du développement. Au début, en 1950 et 1960, de petites équipes faisaient de petites choses. Dans les années 1970, la population des programmeurs commençait à exploser. Est venue alors l'idée de faire des grosses applications avec de grosses équipes. Pour Robert C. Martin, le secret était de faire une multitude de petites choses avec une multitude de petites équipes. C'est le concept de l'Agile : les grosses applications sont réalisées par une multitude de petites équipes qui collaborent et font de petites choses. Aujourd'hui, Agile est un terme qui a été étendu à Lean ou Safe. Ces méthodes ne sont pas forcément mauvaises. Mais ce n'est pas l'origine du message agile.
En 2001, le cycle en V était partout. 17 experts se sont alors réunis. Ils voulaient créer un manifeste. Le but était d'introduire une approche plus légère et plus effective. Ces 17 hommes venaient des équipes Extreme programming, Scrum et d'autres processus légers. D'autres n'étaient pas affiliés.
Ils voulaient partager leurs croyances communes à propos du développement de logiciels en général. Ils ont alors formé les valeurs et principes Agile. Pour Robert C.Martin, les cycles en V fonctionnent bien pour certains projets. Ces derniers sont coûteux, ont des problèmes bien définis et des buts extrêmement spécifiques. Agile fonctionne davantage pour un autre style de projets. Ceux-ci ont des problèmes partiellement définis et sont plus efficaces en produisant des petits incréments peu coûteux.
Quelque chose de vraiment appréciable en tant qu'ingénieur machine learning est le fait que Robert C. Martin décrit l'Agile comme une approche orientée data-driven. L'Agile est une manière d'avoir du feedback en récoltant de la donnée. On comprend mieux ce qui se passe dans le projet. On ne veut pas gérer un projet avec de l'espoir. L'Agile tue l'espoir et donne des données à la place. Il ne s'agit pas d'aller vite, mais d'avoir une meilleure vue de ce qui se passe. Il s'agit de produire de la donnée qui nous aide à prendre des décisions.

L'Agile ne concerne pas uniquement le management. C'est souvent l'erreur qu'on fait. La technique aussi est importante. Scrum peut être un bon début, mais sans les principes et pratiques de l'Extreme programming, l'Agile n'est pas agile. Par exemple, les tests, le code qui doit appartenir à tous ou le code propre font partie de l'Agile.

Robert C. Martin explique que l'Agile a été conçue pour les équipes de développements. Il est embarrassé quand on lui demande comment appliquer l'Agile dans d'autres domaines. C'est là un élément qui m'apparaît important et qui nous servira à expliquer pourquoi l'Agile peut ne pas toujours être très efficace en data science. Ce concept a été pensé pour les équipes de développement logiciel, non pas des équipes de machine learning. À la fin, l'auteur laisse un coach parler et donner sa vision de l'Agile. Ce dernier explique que pour lui l'Agile est un commencement. C'est un début pour trouver ses propres principes et pratiques dans son travail actuel. La conclusion à en tirer, c'est que l'Agile peut être une source d'inspiration malgré le fait que cela ne fonctionnera pas tout à fait

avec les besoins de la data science.

Agile et data science : pires ennemis ou meilleurs amis ?

Est-ce que l'Agile convient à la data science ? Dans quelle mesure ? Nous pouvons donc reprendre la définition de Robert C. Martin. J'aimerais cependant aussi reprendre celle de Rachel Davies et Liz Sedley, deux coachs qui ont écrit *Agile coaching*. Pour elles, l'Agile fait référence à un mélange d'Extreme Programming, de Scrum et de Lean.

Les systèmes de machine learning sont différents des systèmes de programmation traditionnelle. On commence peut-être à trop le dire. Pourtant, nombreuses restent les tentatives de calquer simplement les modèles du développement traditionnel sur la data science. Dans la programmation classique, on a de la donnée et on construit le programme en vue d'un résultat. En machine learning, c'est différent. Comme nous l'avons vu, une grande part du programme vient des données et du modèle. On essaye toujours de répondre aux besoins d'un utilisateur final, mais c'est quand même bien différent. On ne peut pas prédire ce que le programme fera avant d'explorer les données et de construire le modèle. De plus, une part du programme est stochastique : le modèle lui-même. Un « product owner » peut aider les data scientistes à penser le produit, mais pas comme il le ferait dans le développement logiciel.
Les estimations sont dures. Elles semblent impossibles pour de la data exploration. Scrum met en avant l'estimation. Par définition, la data exploration est de l'exploration. On ne sait pas ce qu'on va trouver. Alors, faire un « planning poker » par-dessus est presque impossible. On peut définir ce qu'on appelle des « spikes », des cartes auxquelles on assigne un temps délimité. C'est une idée. Cependant, il ne faut pas être surpris si à la fin du « spike », rien n'a abouti.
Définir les « user stories » aussi est un processus différent. Entre autres, le machine learning traite de data exploration, analyse, modélisation, etc. Ces étapes sont dures à estimer. Elles sont encore plus dures à planifier. On ne sait pas ce qu'on va trouver et quel chemin on va finir par suivre. Alors, l'Agile est-elle inutile ? Je ne pense pas.
L'Agile et notamment Scrum, s'ils ne peuvent être appliqués tels quels à la data science restent une source d'inspiration. Ils sont notamment utiles pour les étapes de mise en production d'un modèle. Ce moment est moins incertain et peut davantage bénéficier de ces méthodes.
L'Agile est un processus orienté data. Pour un data scientiste, c'est une idée délicieuse. Agile aide à collecter des données telles que les suivantes :
- Ce que nous avons fait pendant cette itération
- Ce que nous prévoyons de faire
- Quelles actions pourraient améliorer notre manière de travailler

Les données aident à prendre de meilleures décisions.

L'Agile aide à mettre en place une boucle d'amélioration continue. Avec les itérations, Agile met en place une boucle. Dans celle-ci, il y a des opportunités pour s'améliorer. Pour moi, la plus importante, c'est la rétrospective. C'est une manière de discuter des manières d'améliorer l'équipe, notre vie au quotidien en tant que data scientiste, etc.
L'Agile peut aider à communiquer. Quelquefois, j'entends des personnes se plaindre qu'il y a trop de réunions en Agile. Ils font référence à Scrum et l'idée d'un « daily meeting », réunion de 15 minutes,

tous les jours. Pour commencer, toutes les équipes n'ont pas besoin de ce genre de réunion. Cela peut passer par une mise à jour informelle dans un chat ou des échanges continuels comme dans du pair programming. J'ai lu une fois qu'il fallait être agile avec l'Agile. J'aime bien cette idée. Cependant, Agile donne des idées pour communiquer : des dashboards partagés et visibles, les fameux « daily meetings », etc. C'est ensuite à vous de sélectionner ce qui vous apparaît le plus utile là-dedans. Essayer, remettre en question, innover, c'est aussi cela l'esprit Agile.
L'Agile est une manière de se rappeler que les méthodes techniques comptent. Un des principes de l'Agile est l'excellence technique. L'Extreme programming est aussi une méthode Agile qui met l'accent sur les méthodes techniques. Pour avoir de solides bases et mettre en production sereinement de la data science, l'excellence technique est une manière d'y parvenir certaine. « Test Driven Development », « Refactoring », « Pair programming » sont des techniques apportées par l'Agile. Choisissez celles qui vous conviennent au moment importun. Je donnerais le même conseil : essayer, remettre en question, innover, ne pas hésiter à réessayer, recritiquer, etc.

En conclusion, là où l'Agile reste utile, c'est qu'il s'agit là d'une réserve d'idées. Le mot de la fin sur ce sujet serait : rester Agile avec l'Agile. Mais ne le rejetez pas d'emblée sans vous être familiarisé avec. C'est un concept riche d'enseignements.

Focus sur la rétrospective

Je pense que s'il y a une pratique à garder de l'Agile, c'est la rétrospective. Que faisons-nous lors de celle-ci ? Elle peut se dérouler de différentes manières, chaque facilitateur et équipe y incluant des spécificités. L'intérêt est en tout cas d'avoir trois éléments : qu'avons-nous apprécié récemment dans notre travail ? qu'avons-nous peu aimé ? que souhaitons-nous améliorer ?
Pour commencer, on peut l'imaginer avec plusieurs étapes. Il s'agit de réunir tout le monde autour d'une table et d'un même écran. Chacun doit pouvoir prendre du temps pour réfléchir à ce qu'il a pensé de la dernière itération et ce qu'il souhaiterait voir améliorer dans la prochaine. Je pense qu'il vaut mieux que cela prenne une forme graphique. En présentiel, chacun peut à son tour coller des post-its sur un mur. Chacun représente une idée que la personne peut développer oralement. En distanciel, il y a des outils qui permettent de faire du partage de post-its virtuels. Une fois que chacun a exprimé son opinion, il s'agit alors d'évaluer les idées que nous voulons aborder en priorité. S'il y en a peu, il n'est pas besoin de passer par cette phase. Sinon, on peut faire un système de votes. On prend alors plusieurs minutes pour discuter des sujets. C'est le moment où on essaye de faire émerger des axes d'améliorations qui soient réalisables. Dans la dernière phase, on s'accorde sur les idées d'actions qui ont vu le jour. On désigne un responsable pour chacune. À la prochaine rétrospective, on suivra alors l'avancée dans celle-ci. Le processus peut ainsi être vu ainsi :
- Récapitulatif des actions décidées lors de la dernière rétrospective et évaluation de leur statut
- Écriture des post-its
- Partage des post-its
- Vote
- Discussions sur les sujets priorisés
- Revue des idées d'actions qui ont émergé
- Désignation d'un responsable pour chaque action

On peut voir des rétrospectives animer différemment. Certaines sont plus ludiques que d'autres. Le plus important est de pouvoir partager son ressenti et essayer de trouver des solutions ensemble sur les problèmes que l'on rencontre. C'est ainsi qu'on rentre dans une boucle d'amélioration continue. Il m'apparaît important qu'il y ait un facilitateur. Celui-ci peut être un membre de l'équipe. Il ne faut pas que ce soit toujours le même. Il s'agit dans ce cas de veiller à ne pas mélanger son rôle de facilitateur et de co-équipier. Son rôle est de veiller à donner la parole à tout le monde et de veiller au bon déroulement de la réunion. Il s'agit aussi de prendre du recul pour discerner les potentielles actions que peut-être les membres parce qu'ils sont immergés dans leurs problèmes ne voient pas. C'est presque un travail psychologique si on est assez bon pour évaluer les gens. Dans tous les cas, le facilitateur permet de faire en sorte que la réunion aboutisse sur des améliorations. L'un des pièges est que cela devienne un espace de défoulement sans plus-value.

Crisp-DM et TDSM

Quand on fait de la data science, on entend davantage parler de méthodes comme Crisp-DM ou TDSM. Je vous avoue d'entrée de jeu les avoir moins expérimentées. Je ne connais TDSM qu'en théorie par exemple, mais cela m'apparaissait important d'en discuter aussi. Ces méthodes ont été créées dans le contexte de la data science. Elles apportent donc un autre regard. Commençons par CRISP-DM.

CRISP-DM signifie « Cross Industry Standard Process for Data Mining ». Cette méthode a été créée en 1996 et est assez répandue. Elle se décompose en six étapes :
• Connaissance du Métier
• Connaissance des Données
• Préparation des Données
• Modélisation des données
• Évaluation
• Déploiement

Dans la « connaissance du métier », on s'intéresse aux problèmes et on discute avec le business et les utilisateurs pour cerner un point en particulier qu'on pourrait résoudre. Dans la « connaissance des données », il s'agit de voir si des données peuvent nous aider à répondre aux problèmes. C'est aussi là une phase où on voit la faisabilité du projet. Si les données sont manquantes, biaisées ou impropres d'une manière générale, on ne pourra pas aller plus loin. Dans la préparation des données, on va plus loin dans l'analyse des données. On leur donne forme pour préparer la modélisation. C'est aussi le moment où on crée les « features » qui serviront au modèle à apprendre. La « modélisation des données » est l'étape dont tout le monde parlait il y a quelque temps, celle où on titille les algorithmes pour leur faire apprendre des choses nouvelles. L' « évaluation » est une étape qui permet de valider le/les modèles ainsi que leur configuration avec notamment les hyperparamètres. Enfin, on déploie. Ce qu'il y a d'appréciable dans cette méthode, c'est qu'elle présente les étapes que tout projet de data science comporte. Ce qu'il y a de plus préjudiciable, c'est qu'elle peut paraître ressembler à une forme de tunnel où on n'a un résultat final qu'à la fin. En cours de route, on jugera peut-être que le projet ne vaut pas le coup et qu'il vaut mieux s'en séparer. Ce n'est pas grave. Mais plus vite, on arrive à une évaluation du produit dans la vie réelle, plus vite on peut ajuster et le rendre plus

performant. Il est aussi regrettable que le « déploiement » soit envisagé comme une phase finale. C'est un concept qu'on peut cependant implicitement inclure dès les premières phases du projet. En effet, quand on parle de « faisabilité » qu'on évalue dès les premières étapes, on s'interroge sur la capacité du produit à réellement partir en production. Il est difficile malgré tout d'y répondre sans être allé jusqu'au bout. Ce me manque finalement dans le CRISP-DM, c'est le caractère agile avec cette boucle de feedback rapide. C'est pourquoi les équipes de Microsoft ont décidé de créer TDSP pour Team Data Science Process. Ce processus se veut plus Agile. TDSP a les composants clefs suivants :

- Une définition d'un cycle de data science
- Une structure de projet standardisé
- Une infrastructure et des ressources recommandées pour les projets de data science
- Des outils et utilitaires recommandés pour l'exécution d'un projet

En termes de processus, il y a quatre grandes étapes :

- Présentation de l'entreprise
- Acquisition des données et compréhension
- Modélisation
- Déploiement

Ce qu'il y a d'appréciable avec ce processus, c'est que les étapes ne sont pas forcément linéaires. On peut commencer à déployer tout de suite après l'acquisition des données et leur compréhension. Il peut aussi y avoir un va-et-vient entre cette étape et la modélisation. Le retour au business et aux utilisateurs est aussi constant. Sur le papier, ce processus est alléchant, mais je ne l'ai jamais testé.

Que choisir alors entre tous ces différents processus ?

Nous avons vu que l'Agile n'était pas pensé pour la data science et qu'en ce sens, elle n'est pas complètement adaptée. Elle reste cependant une source d'inspiration et un moyen d'entrer dans une boucle d'amélioration continue en étant data driven. CRISP-DM a été pensé pour la data science. C'est un processus linéaire qui a l'avantage d'être pensé pour développer du machine learning. Quant à TDSP, il est attrayant parce qu'il semble mélanger un peu d'Agile et les étapes du CRISP-DM. Que faire alors ? Que choisir ?

Un jour, j'ai vu dans une conférence une personne qui expliquait que la plupart des équipes de data science n'ont pas de processus. Elle disait alors que partir sur n'importe lequel était déjà un début. J'aime beaucoup cette idée. On pourrait se battre à l'infini sur les meilleures manières d'organiser au mieux les processus de son équipe. Une première étape est de s'organiser, puis de voir ce qu'on peut améliorer. On itère alors dans une boucle d'amélioration continue. Peu importe qu'on choisisse Scrum, Kanban, TDSP, l'important est de faire un choix, de s'adapter et d'itérer. Il ne faut pas être dogmatique. Ce n'est pas grave de ne pas respecter le guide Scrum à la lettre. Ce qui est grave, c'est de le respecter sans savoir pourquoi et dans un contexte où ce n'est pas adapté. De la même manière, être dogmatique c'est aussi rejeter une idée pour le simple fait qu'on l'a trop entendue ou qu'elle ne nous a pas convenu dans une précédente expérience. Ainsi, ce qui est bénéfique, c'est de choisir une stratégie d'organisation et de l'améliorer au fur et à mesure en n'ayant pas peur de croiser les sources et inspirations.

S'organiser dans une entreprise

Les data scientistes sont-ils des développeurs ?

Cette question peut paraître anodine. Pourtant, plus le temps passe, plus je crois qu'elle est cruciale. Je la crois importante non pas parce qu'il y aurait une réponse universelle et indivisible, mais parce que le data scientiste qui y répond donne aussi à voir la manière dont il veut travailler et évoluer.
Pour la petite histoire, j'ai posé cette question sur Twitter. J'ai eu 39 réponses. Il y avait 44% de non, 36% de oui et 21% de personnes qui ne se prononçaient pas. J'aurais pour ma part répondu « non ». Je rejoins la définition de Wikipédia qui explique que le développement logiciel n'est qu'une composante parmi d'autres de la data science. Mon sondage bien qu'amusant ne révèle pas grand-chose en soi. Je pense que les personnes qui me suivent sont pour la majeure partie d'entre elles des développeurs web, français, du même âge que moi et inscrits sur Twitter. Cela constitue beaucoup de biais. Mais, cette question m'a permis d'avoir des discussions plus intéressantes et de savoir la manière dont mes collègues se positionnaient. Certains se voient comme des développeurs et ne veulent pas interagir directement avec le métier. Ils souhaitent se concentrer sur la modélisation. Bien que les développeurs ne soient pas forcément éloignés des problématiques business et des personnes qui le composent, la position du data scientiste dans ce cas nous en apprend davantage sur la manière dont il entend mener son travail. Une autre personne m'a répondu que cela dépendait de quel type de data scientiste il s'agissait. Un data ingénieur est certainement plus proche d'un développeur qu'un spécialiste en data visualisation par exemple. Elle avait dans l'idée qu'aujourd'hui, les data scientistes étaient multitâches, mais que le métier se spécialiserait au fur et à mesure des années. Ainsi, la question n'aurait plus d'intérêt.
Pour ma part, je ne pense pas que la data science soit du développement classique. C'est une idée que j'ai de mon point de vue d'ingénieur machine learning. J'ai la conviction que tous les codes ne se mettent pas en production de la même manière et qu'un système de machine learning a des spécificités qu'il faut prendre en compte pour avoir une mise en production possible, facile, sereine et qui a du succès. Mon point de vue est aussi biaisé. Il l'est tellement que j'ai décidé d'écrire un guide sur le sujet afin de vous aider dans cette entreprise. Celle-ci ressemble à la production de programmes traditionnels, mais s'en éloigne aussi grandement.

La place des data scientistes dans l'entreprise

Vaut-il mieux avoir des data scientistes au sein des équipes business ou regroupés dans une équipe dédiée ? C'est une question qui revient souvent. Elle revient à se demander s'il faut disperser les data scientistes au sein de l'entreprise. Il y a plusieurs avantages à avoir des data scientistes répartis à différents endroits de l'entreprise. Le premier est que ceux-ci sont au plus près du business, en comprennent davantage les enjeux et peuvent proposer des solutions optimales. Le deuxième grand avantage, c'est que les concepts de data science, machine learning et même intelligence artificielle ont alors une chance de se démocratiser. Or, comprendre ces termes est une clef de réussite pour l'entreprise. Le problème c'est que les data scientistes se retrouvent ainsi souvent isolés. La standardisation dans la manière de développer et de mettre en production risque de plus d'être manquante. On peut pallier cela, mais c'est plus compliqué de gérer des personnes dispersées au

sein d'équipes différentes ayant chacune leurs propres contraintes et manières de travailler. Je n'ai jamais personnellement expérimenté cette organisation. Ce sont les avantages et inconvénients que je discerne après discussions avec plusieurs personnes l'ayant expérimentée.
La deuxième solution est d'avoir une équipe de data scientistes dans une équipe dédiée. L'avantage c'est la force de travail, la stimulation et les échanges possibles. Standardiser les manières de travailler est plus aisé. L'inconvénient, c'est qu'on est moins proche du business. C'est lui cette fois-ci qu'on voit dispersé dans d'autres équipes. C'est ce que j'ai surtout expérimenté et apprécié malgré les défauts que j'y vois.
J'aimerais ajouter que j'ai parfois travaillé dans des équipes formées uniquement de data ingénieurs en étant parmi les personnes qui établissaient le lien avec les data scientistes. J'ai aussi travaillé avec une équipe composée uniquement de data scientistes en étant parmi ceux qui établissaient le lien avec les data ingénieurs. Je n'ai jamais rencontré de situations parfaitement heureuses dans ces contextes. Je n'ai jamais vu de guerre ouvertement déclarée, mais très souvent des incompréhensions sur les fondements du travail d'autrui. Je trouve cela regrettable. Je pense que ces a priori sont dus au fait que les équipes sont différentes. Une seule et même équipe de personnes spécialisées dans la data me paraît la meilleure des organisations. J'avoue ne l'avoir jamais expérimentée, mais je ne peux pas m'empêcher de penser que c'est ce qui évite toute tension, permet une collaboration apaisée et rapide pour une mise en production sereine où chaque acteur comprend les intérêts de son collègue. Le data ingénieur comprend l'importance de tester la donnée et versionner son modèle. Le data scientiste comprend l'intérêt d'avoir un code propre qui est aisé à maintenir et débuguer.

Le métier d'ingénieur machine learning

Qu'est-ce que le métier d'ingénieur machine learning ? Il y a plusieurs définitions. Elles se recoupent, mais se contredisent aussi parfois. Pourtant, j'ai fait appel plusieurs fois à ce rôle pour parler de mise en production. Je suis ingénieur machine learning et propose donc de vous montrer la manière dont je vois ce rôle. J'espère que cela vous aidera à y voir plus clair.
Pour moi, l'ingénieur machine learning est un data scientiste qui est spécialisé dans tous les éléments qui concernent la mise en production. C'est un champ vaste. Cela va de la robustesse du code au monitoring, en passant par les tests et les meilleures manières de servir un modèle. Cela demande plusieurs compétences. On peut d'ailleurs être spécialisé dans certaines d'entre elles. Je listerais celles-là ainsi :
- Machine learning : pour moi c'est la première compétence à avoir. Il s'agit de savoir ce qu'est le machine learning, ses spécificités pour savoir le gérer intelligemment en production.
- Monitoring : monitoring classique et monitoring spécifique du machine learning (performance du modèle dans le training, détection d'anomalies dans les données, data drift)
- Tests : du code, du modèle et des données
- Clean code pour avoir quelque chose de maintenable, transférable et facile à debugger
- Déploiement : savoir gérer des pipelines de build, tests et déploiement
- Datavisualisation, API web
- Développement distribué : savoir travailler avec des données volumineuses
- Intérprétabilité/explainabilité : comprendre les enjeux et savoir y répondre
- Produit : discuter avec le business pour savoir comment s'interfacer avec le produit final

Cela représente beaucoup de choses. Certains diront que c'est peut-être trop. Dans tous les cas, avoir une personne dans son équipe qui aide les autres à mettre en production apporte des avantages certains. Je souhaite cependant que mon métier disparaisse dans les années à venir. Cela peut paraître étrange, mais je vois aussi ce rôle comme celui de quelqu'un qui doit transmettre ses compétences à l'ensemble des autres data scientistes autour de lui pour que mettre en production devienne une chose naturelle. Je voudrais que plus personne n'ait peur de mettre en production et soit serein face à cette idée non pas parce qu'il est inconscient, mais parce qu'il a les armes pour le faire.
Je voulais vous parler quelque peu du métier d'ingénieur machine learning. Mon objectif était de vous montrer ce qui se cache derrière ce terme afin que vous en ayez une meilleure idée. Vous saurez davantage comment une telle personne pourrait intégrer vos équipes et les mener en avant vers des productions sereines.

S'organiser au sein de son équipe

Les standards

Dans une équipe de data science, on peut être amenés à travailler sur beaucoup de projets. Ce n'est pas comme en programmation classique où il y a un produit. Il peut y avoir un produit central qu'on fait évoluer et qu'on améliore. Mais parfois, on a une myriade de petits projets qu'on doit mettre en production et maintenir.

Définir des standards

Pour ce faire, il vaut mieux instaurer des standards. Je ne parlerai pas de « bonnes pratiques » parce que je ne crois pas qu'il y ait en soi de « bonnes pratiques » qui vaudraient pour tout contexte et toute équipe. Il y a sans doute des pratiques qui émergent plus que d'autres et qui semblent avoir fait leurs preuves. Mais l'important à mon sens est d'adapter cela à son équipe. Par exemple, je ne crois pas que les notebooks en production soient une bonne idée pour des raisons déjà évoquées. Cependant, si mon équipe a des difficultés à coder de manière « classique » et performe sur des notebooks, c'est une solution envisageable.
Le principal est d'être en accord sur certaines pratiques et de les appliquer à l'ensemble de nos projets. Le mieux est que cela émerge d'un consensus et que cela se fasse peu à peu en laissant plusieurs options. L'équipe ne doit pas avoir l'impression qu'on lui impose quelque chose. Si c'était le cas, elle aurait du mal à y adhérer et la pratique mettrait plus de temps à s'installer. Les standards s'intègrent dans une équipe. Cela n'est pas à voir comme une grossière erreur. Cela permet de prendre à chacun le temps de les absorber. Enfin, un peu de flexibilité est salutaire. Ce n'est pas parce qu'une grande partie de mon équipe préfère travailler avec des notebooks que certaines personnes n'ont pas le droit de travailler directement avec des fichiers Python.
Une fois que les standards ont émergé, il s'agit de leur donner une forme. Si on ne les matérialise pas, cela ne sera pas forcément évident de s'y conformer. Il y a plusieurs manières pour réaliser cela. La plus évidente à mon sens est d'avoir un projet type qui serve d'exemple. Son processus de réalisation, son code, tout ce que l'on veut standardiser, doit y être présent. Cela est plus simple de se référer à un projet existant quand on veut savoir quels sont les standards d'une équipe. On le fait souvent

sans même en prendre clairement conscience. Ensuite, cela peut s'appuyer sur une documentation qui explique alors ce qui ne se voit pas. Principalement, cela explique les choix qui ont été faits. C'est aussi une forme de guide pour démarrer.
Que veut-on standardiser ? La question qui suit est : que veut-on laisser flexible ? Cela dépend. Il ne s'agit pas de tout standardiser car c'est risquer de faire perdre trop d'autonomie aux data scientistes et donc de la motivation. Cependant, on peut envisager de standardiser dans une certaine mesure certains principes de code, le pipeline de déploiement, la manière de versionner le code, le modèle et les données. Certaines choses sont plus simples à appréhender quand elles respectent un modèle. On peut réaliser cela avec tous les outils qui servent à tester, monitorer et formater son code. En revanche, il n'est pas obligatoire de devoir imposer un IDE ou un notebook en particulier. Cela me semble trop contraignant pour les personnes qui y sont confrontées.

Les standards déjà définis : la notion de clean code

Pour sûr, les bonnes pratiques n'existent pas en soi. Cependant, il serait fort dommage de ne pas lever la tête de son clavier pour regarder les solutions proposées dans le monde actuel et les tester. Il n'est pas besoin de réinventer la roue à chaque fois. On peut bien sûr l'adapter, mais pas forcément la reconstruire depuis ses fondations. Ainsi, il y a des standards qui sont déjà définis. Il y a des outils qui peuvent aider. Par exemple, pour formater du code Python, on peut utiliser Black. Ensuite, en termes de code, il y a des choses sur lesquelles on s'accorde plus que d'autres. En réalité, le débat fait parfois rage sur certains éléments. Pourtant, je ne peux m'empêcher de penser que comprendre la notion de « clean code » peut aider. Elle est utile, à mon avis, pour deux raisons. Premièrement, cela sert à comprendre pourquoi écrire du code sans se soucier de la qualité est dommageable pour la maintenance du projet. Deuxièmement, cela sert à s'interroger sur les meilleures manières de produire un code de qualité. Pour sûr la question n'est pas tranchée, mais s'interroger sérieusement dessus, c'est déjà un gage de qualité. Il est vrai que le code en data science est moins imposant et moins important qu'en programmation traditionnelle. Cependant, il n'est pas non plus négligeable.
La notion de « clean code » vient de Robert C. Martin qui a écrit les livres *Clean code* et *Clean coder*. Pourquoi est-il important d'avoir un code propre ? Quand on part en maintenance, on traite avec un code qu'on va devoir débuguer et faire évoluer. Ce n'est plus un code tout frais qu'on vient de créer et qu'on connaît bien. Y replonger peut être périlleux. Avoir un code propre signifie avoir un code lisible dans lequel on a envie de pénétrer et avec lequel on a envie de travailler. Robert C. Martin définit donc des standards comme des limites dans le nombre de paramètres d'une fonction, des noms de variable expressifs, etc. Certains paraissent peut-être évidents. D'autres sont parfois sujets à débat. Le plus important est de se soucier de la qualité de son code comme on se soucie de la qualité de ses données. Au-delà du fait d'écrire des documentations (qu'on ne lit pas toujours) pour suivre des principes, une technique pour avoir des standards communs est d'établir une forte collaboration au sein d'une équipe avec un système de relecture par exemple. C'est l'occasion de dédier un moment au code propre. On se pose des questions telles que : aurais-je envie de relire ce code dans six mois ? Est-ce que je le comprends du premier coup ? Y a-t-il plus lisible ? C'est aussi l'occasion en réalité de se poser des questions sur la qualité du modèle même ou la performance de l'exécution.
Pour résumer cette section, il existe des manuels de code propre. Bien sûr, il n'y a aucune obligation

à adhérer à l'entièreté de ces guides. Ils représentent cependant une manière de démarrer non négligeable.

La collaboration

Comment ces standards peuvent-ils finalement émerger ? Il y a beaucoup de solutions qui nous viennent du monde des développeurs. Vous pouvez piocher celles qui vous intéressent le plus. C'est pourquoi je vous propose de faire un petit tour.

Pull requests et reviews

L'idée est de faire valider son travail par ses pairs. Le terme de « pull request » vient du monde Git. On crée une branche à partir de la branche principale du code. On réalise ce qu'on veut réaliser sur sa branche. Puis on demande le droit d'intégrer ses ajouts, modifications et suppressions dans la branche principale. L'avantage, c'est qu'une ou plusieurs personnes peuvent alors venir lire le contenu de cette « pull request », faire des commentaires, approuver ou même désapprouver. Ce principe de relecture est appelé une « review » en franglais. C'est l'occasion de discuter du code. En data science, on veut aussi discuter du modèle et des données. L'intégration avec les outils Git des résultats d'une expérience n'est pas encore très aisée. Mais cela peut être compensé avec des descriptions qu'on génère automatiquement pour une « pull request ». Pour commencer, j'ai l'habitude d'utiliser ce modèle pour expliquer le contenu d'une « pull request ».

```
## Quoi

## Pourquoi

## Comment
```

C'est déjà l'occasion d'expliquer un peu plus le but de la « pull request ». On peut se servir de ces entrants pour donner plus de contexte. Par exemple, si on a testé un nouvel hyperparamètre et qu'on veut l'intégrer dans la base de code commune, c'est l'endroit pour indiquer les performances du modèle. Certains outils proposent aujourd'hui de faire cela automatiquement.

Une « pull request » a pour but d'entraîner une revue. C'est l'occasion de discuter les tenants et aboutissants de ce qu'on veut fusionner. Il y a quelques années un développeur a écrit un article à ce sujet qui continue à m'inspirer. Gilles Roustan est un blogueur, artisan développeur web à l'origine de l'article *La revue de code bienveillant* écrit en 2017. Aujourd'hui encore, j'applique les principes qui y sont décrits. Par exemple, je n'utilise pas le « tu ». Le code appartient à tout le monde. Si je suis mandatée pour le relire c'est qu'il m'appartient aussi. Je ne cherche donc pas à incriminer l'autre sur ce qu'il aurait pu faire. Donc si j'ai une remarque à faire, je préfère utiliser le « on ». « On devrait » est à mon sens plus appréciable qu'un « tu devrais ». Je m'inclus dedans. Je montre par là que je suis davantage concernée et impactée. J'apprécie aussi de faire des suggestions. Les remarques du style « je n'aime pas », « ce n'est pas très propre » ne font pas progresser la personne qui soumet son code

en revue. Il faut aller plus loin que cela. Si j'ai une remarque, je l'accompagne d'une suggestion. Je préfère dire « pour plus de lisibilité, on pourrait extraire une fonction X de ces lignes » plutôt que « ce n'est pas très propre ». Pour résumer, utiliser un « on » qui m'inclut et montre que je suis concernée ainsi que faire des suggestions plutôt que juste des remarques me paraît être deux éléments pour une revue qui porte ses fruits.
La revue de code est une des premières manières de collaborer et ce que j'apprécie c'est qu'elle permet de se poser et de s'interroger sur ce qu'on veut intégrer dans la base de code commune. En data science, elle a cela de complexe qu'il s'agit aussi de s'interroger sur le modèle et les données en lui-même. Il faut aller plus loin que des questions comme « est-ce que ce code fonctionne », « est-ce qu'il est propre ». Il s'agit aussi de se demander si les résultats du nouveau modèle mis en place sont pertinents. Le nouveau modèle est-il plus performant ? En termes de justesse, précision, mais aussi transparence, temps d'exécution, etc. Ces éléments ne sont pas lisibles avec le code uniquement. Il faut aller plus loin. Est-ce que cet hyperparamètre est plus adéquat ? Est-ce que cet ajout de feature est pertinent ? Les performances dans son ensemble sont-elles accrues ? Voilà des questions qu'il s'agit de se poser en plus quand on est face à un système prédictif. Les erreurs et améliorations en data science sont globalement moins visibles. Cela est difficilement décelable juste en lisant le code. Il faut souvent aller chercher du côté de l'expérience qu'on aura réalisée. C'est pourquoi il est important de versionner aussi son modèle et ses expériences. C'est là qu'on voit les changements qui s'opèrent dans la performance du modèle même.
Les pull requests avec la revue qui y est associée est un moyen de poser la question des standards qu'on veut mettre en place dans l'équipe. Cela permet de les faire naturellement émerger et aussi de leur donner une vie concrète.

Pair programming

Si j'apprécie les revues de code pour partager et discuter autour des évolutions du système, je suis encore plus fan du pair programming. Voilà encore une technique qui nous vient du monde des développeurs et plus précisément de l'Extreme Programming, méthode Agile. Dans le pair programming, on décide de travailler à deux. Il y a plusieurs manières de réaliser cela. Beaucoup de personnes apprécient la technique du ping-pong pair programming très orienté autour des tests. Le processus dans le ping-pong pair programming est le suivant :
- A rédige un test
- B rédige le code correspondant et rédige un nouveau test
- A rédige le nouveau code correspondant au nouveau test et rédige encore un nouveau texte
- Etc.

Les avantages d'un tel processus c'est qu'on n'oublie pas les tests et qu'il y a deux cerveaux qui contribuent au même problème. Certaines personnes font du pair programming de manière plus souple sans respecter forcément cette règle. L'idée est de trouver des solutions pour que chacun contribue à la construction du système et allie ses forces vers un même but.
En data science, je ne pense pas que ce processus soit utile pour toutes les phases. Par exemple, on a parfois besoin de se plonger dans un sujet et de s'y plonger seul. Quand on a l'idée qu'une nouvelle feature peut être plus performante ou qu'on veut faire de l'analyse de données, je comprends qu'on n'ait pas forcément envie de réaliser cela à deux. On a parfois trop vanté le travail solitaire. Il ne

s'agit pas de tomber dans l'effet inverse. Parfois, se concentrer seul est plus utile. De plus travailler à deux est plus demandant. Il n'y a aucune place pour la détente. L'esprit est en ébullition constante. Une fois, j'ai rencontré un développeur qui m'expliquait que lorsqu'il décidait d'avoir une journée en pair programming, il l'écourtait systématiquement. Je comprends complètement ce parti-pris. Une fois cela dit, le pair programming ou pair thinking d'une manière générale, s'il m'est permis ce néologisme, a plus d'un intérêt. N'y a-t-il pas une meilleure façon de faire sien un code ou un modèle qu'en travaillant dessus ? L'idée que le code ou le modèle appartienne à tout le monde, du moins à toute l'équipe concernée, se trouve exacerbée quand on travaille véritablement dessus. De plus, travailler à deux sur un même problème permet d'échanger directement. On ne doit pas attendre la revue pour enfin avoir le droit d'échanger. On peut réaliser cela d'entrée de jeu. L'un des soucis avec la revue de la pull request, c'est qu'elle a lieu après l'expérimentation, après le code. L'un des avantages du pair programming est que la revue prend place en même temps que le modèle se construit ou évolue. On peut réagir directement. Cela permet de ne pas oublier ce qu'on voulait faire ou ce qu'on voulait dire. C'est instantané. Au-delà de cet aspect pratique concernant la revue, je considère le pair programming comme intéressant pour d'autres raisons. Il est utile pour la standardisation. On réfléchit ensemble aux standards au moment où on les crée. Cela permet aussi d'apprendre d'autres manières de travailler. C'est à mon sens la meilleure façon d'aligner tout le monde. C'est encore plus intéressant quand on mélange des profils différents tels qu'un data ingénieur et un data scientiste. Chacun a sa spécialité. Le data ingénieur a l'habitude du code. Il sait comment le manier. En contrepartie, le data scientiste en sait davantage sur les statistiques et les spécificités du machine learning. En travaillant ensemble pour certaines tâches, ils allient leurs forces pour être à même de créer la meilleure plus-value qu'il soit du système. Il s'agit cependant de réaliser cela avec un esprit ouvert. Il ne faut pas rester braquer sur ses positions, mais essayer d'allier le meilleur des deux mondes : des statistiques pertinentes et du code maintenable. L'idée est de s'appuyer sur les forces des uns et des autres dans le but de construire un produit de qualité qu'on a hâte et non pas peur de mettre en production.

Mob programming

Il y a une variante au pair programming qui s'appelle le mob programming. Cette fois-ci, toute l'équipe se met ensemble pour travailler sur la même chose. C'est finalement simplement une extension du pair programming. D'ailleurs, il y a une extension du ping-pong pair programming dans le mob programming. Il y a un navigateur et un pilote. Le navigateur est celui qui code et le pilote est celui qui dicte quoi faire. Le reste de l'assemblée donne des idées en s'adressant uniquement au pilote. Toutes les X minutes, on tourne. Les itérations doivent être courtes et on peut mettre un sablier pour éviter de déborder. L'avantage du mob programming par rapport au pair programming c'est qu'au-delà d'intégrer des data ingénieurs et des data scientistes, on peut aussi intégrer les gens du business. Ce qu'on évite, ce sont les aller-retours avec celui-ci pour poser certaines questions. Les choses se construisent ensemble. Je suis assez fan de cette technique. Elle permet aussi une standardisation plus générale en incluant des personnes du business. Cependant, comme pour le pair progamming, je crois qu'en data science, la pratique du mob programming n'est pas toujours idéale. Quand on passe beaucoup de temps à tuner le modèle ou mettre en place du monitoring, je ne crois pas que le mob programming soit forcément nécessaire. Mais cela permet de partager les

connaissances, diminuer les erreurs et standardiser le code ainsi que les manières de travailler. Il faut penser à s'outiller pour faire du mob programming : partage d'écran ou logiciel pour se timeboxer.

Ces manières de collaborer, à savoir les pull requests, revues de données, pair programming et mob programming, sont différentes manières de parvenir à faire émerger et donner vie et sens à des standards au sein d'une équipe. Ils viennent du monde de la programmation classique, mais peuvent être adaptés au monde de la data science. La standardisation est un élément qui permet plus facilement de maintenir un ensemble de projets. C'est utile pour une mise en production sereine. Cela ne doit pas forcément être réalisé avec un grand effort et une énorme documentation qu'on met à jour une fois et qu'on ne relit plus jamais. Il s'agit davantage de faire vivre concrètement ces standards pour ne pas les perdre de vue.

Arrêter d'avoir peur de la production

En tant qu'ingénieur machine learning, je ne peux que redouter de travailler avec des personnes qui ont peur de mettre en production. Je ne dis pas qu'il faille mettre en production n'importe quoi à tout prix. Mais il est important de rappeler que ce que nous faisons au quotidien n'a que peu de valeur s'il n'y a pas d'utilisateur en face pour s'en servir. Mettre en production, c'est donner à un ou des utilisateurs un outil. C'est avoir un impact dans la vraie vie. Il est cependant compréhensible qu'on veuille s'assurer de mettre quelque chose de correct en production. Cela a été tout l'objet de notre guide. Nous avons vu comment versionner, tester et monitorer nos différents éléments afin d'avoir confiance en ce que nous faisons. Même dans les stratégies de mise en production, il y a des manières de gagner en confiance. Ce sont les choses que j'aimerais évoquer dans cette partie.

Mettre en production le plus vite possible le moins de choses possible

C'est le premier conseil que je donnerais. Trop souvent, je vois la tentation de passer des mois et des mois à avoir quelque chose de parfait avant d'appuyer sur le bouton « mise en production ». Cette idée va aussi souvent avec celle qui veut qu'une fois que quelque chose est en production, on n'y touche plus à part peut-être pour débuguer. Bien entendu, c'est mal vu de retoucher à quelque chose qui est en production. La maintenance a presque quelque chose de sale dans ce genre de considérations. Pourtant, à bien y regarder, tous les projets open source que nous utilisons sont en maintenance. Il n'y a donc rien de mal là-dedans. Passer du temps à améliorer son produit et à maintenir sa prédiction apporte de la valeur. C'est peut-être moins perceptible parce qu'il n'y a pas d'effet « whaou ». Pourtant, on continue à créer de la valeur en maintenance. Souvent, on me demande alors comment mettre le moins de choses possible en production. Rentrons plus avant dans ces considérations. Il y a plusieurs cas de figure. Pour commencer, on croit très souvent que tout ce que nous faisons est critique. Ce n'est pas toujours le cas. Cela ne veut pas dire qu'on n'apporte pas de valeur, mais simplement que tout n'est pas critique. Un algorithme de recommandation sur une plateforme de musique par exemple quand il n'y a rien de mis en place n'est pas forcément critique. Ne vous y trompez pas. Je ne dis pas qu'il faut mettre n'importe quoi en production. Ce que je dis, c'est que dans certains cas, c'est moins grave de ne pas avoir l'algorithme parfait. Il ne s'agit pas d'introduire des bugs en production et d'envoyer des choses non testées et non monitorées.

Mais quelquefois, on peut se permettre de mettre en production un algorithme imparfait, mais qui constitue une première version sur laquelle on peut itérer pour améliorer. En data science, une prédiction n'est jamais parfaite. On remplace un processus cognitif en essayant de faire le moins d'erreurs possible. Dans certains cas, on me répond à juste titre que si l'algorithme est moyen, l'utilisateur risque de s'en détourner. En industrie, quand on remplace des processus, c'est en effet à mon avis un grand enjeu. La confiance de l'utilisateur en matière d'intelligence artificielle est à gagner. Il y a aussi des cas où l'algorithme est critique. Le premier exemple qui me vient en tête est celui des voitures autonomes. Je ne voudrais pas être pessimiste, mais dans ce dernier cas, nous ne sommes pas sûrs qu'elles existeront un jour dans notre quotidien. Dans ces cas-là, il est difficile d'envisager une mise en production rapide du modèle. On peut alors mettre en production l'architecture et l'infrastructure. On ne met pas à disposition la prédiction à l'utilisateur, mais on commence à effectuer un premier travail de déploiement et de tests d'infrastructure qui peut parfois être conséquent.

Feature flipping

Il y a aussi des manières de mettre tout en production en cachant complètement cela à l'utilisateur. Le feature flipping est une technique qui consiste à activer au cas par cas des fonctionnalités. On a ainsi développé la fonctionnalité et on peut décider de la rendre active ou pas. Cela peut être réalisé de différentes manières. La façon la plus simple que je connaisse est de passer par un fichier de configuration qui pourrait ressembler à cela :

```
prediction = inactif
feedback = inactif
image = actif
```

La fonctionnalité « image » est activée. Les autres fonctionnalités « prédiction » et « feedback » sont désactivées. On peut bien entendu envisager des manières beaucoup plus ergonomiques mais aussi complexes de faire cela. J'ai connu un cas où nous avions une interface web pour gérer cela. C'était très pratique car très visuel. Cela marchait plutôt bien. Cependant, développer cette interface avait été plus complexe. Il fallait aussi se plugger à cette interface pour récupérer les informations.

Dans le code, une fois qu'on a défini quelles fonctionnalités sont activées ou pas, on peut utiliser des propositions conditionnelles pour rendre accessible tel ou tel morceau de code. Pour exemple :

```
 if prediction == actif :
        Prediction = compute_prediction()
```

La prédiction n'est calculée que si la fonctionnalité « prediction » est activée.
Mais quel avantage à faire du feature flipping si la fonctionnalité n'est pas donnée à l'utilisateur ? C'est une question légitime. La réponse rejoint l'idée de mettre en production l'architecture et l'infrastructure. Si le système prédictif n'est pas vraiment en place, on s'assure au moins que tout le reste l'est. On fait de l'intégration au fur et à mesure. Ce sera toujours cela de moins à tester ou dont

nous aurons peur le jour où on activera la fonctionnalité. De plus, la mise à disposition du système prédictif aux utilisateurs est beaucoup moins effrayante. Il s'agit de changer une option et le tour est joué.
J'ai parfois comme objection que mettre en place un système de feature flipping est complexe. Quand il s'agit uniquement d'un système de fichiers, c'est gérable. Il est vrai qu'il faut nettoyer son code pour ne pas laisser traîner de « if » et « else » de partout. Mais c'est là une action classique du refactoring que de supprimer le code mort. Le feature flipping, non seulement peut être implémenté de manière très peu complexe, mais il apporte une sûreté dans la mise en production. Il retire une partie de la complexité de celle-ci ou du moins la masque en permettant aux éléments d'être intégrés les uns après les autres.

Évaluation utilisateurs et data analyse

Comment évaluer la pertinence de ce que l'on met en production ? Il y a plusieurs manières d'y parvenir. Cela dépend du nombre d'utilisateurs que nous allons avoir, du contexte dans lequel la prédiction est faite et du type d'entreprise dans lequel nous travaillons. Ici, nous allons commencer par envisager deux manières de recueillir du feedback : l'évaluation utilisateurs et l'analyse des données utilisateurs.
On peut commencer par faire des interviews avec les utilisateurs. Non seulement on peut les réaliser avec les utilisateurs finaux, mais aussi avec la personne commanditaire du système prédictif. Il s'agit en fait d'évaluer l'impact que l'on cherche à avoir, le contexte que l'on cherche à améliorer et/ou le processus qu'on veut tirer vers le haut. Mais au-delà de ces interviews préliminaires, on peut aussi décider d'en avoir tout du long. D'abord, les choses peuvent changer. Les interlocuteurs peuvent changer d'opinion ou voir leurs conditions évoluer et donc leurs besoins changer. Puis, c'est aussi une manière de savoir si on est dans la cible une fois le système rendu.
Il est à noter que les besoins du commanditaire ne sont pas forcément les mêmes que ceux des utilisateurs. Par exemple, un commanditaire peut vouloir automatiser une partie du travail de techniciens. Ces derniers sont les utilisateurs du produit mais ne sont pas en phase avec cette volonté d'automatisation. C'est important d'avoir ce genre d'entrées pour mesurer la méfiance des utilisateurs par rapport à l'outil qu'on leur propose.

Les évaluations utilisateurs peuvent donc être intéressantes et nous en apprendre davantage sur la pertinence de l'outil qu'on propose. Cependant, elles ne sont pas toujours suffisantes. Les utilisateurs ne disent pas toujours ce qu'ils pensent vraiment. Il peut être difficile de discerner leurs besoins. Par exemple, on sait que les émissions de télé-réalité cartonnent. Les chiffres ne mentent pas. Pourtant, les utilisateurs sont assez peu nombreux à dire très clairement que ce sont là leurs émissions préférées. C'est pourquoi la parole des utilisateurs est aussi à prendre avec précaution. Une autre manière d'évaluer l'impact du produit qu'on propose, c'est de collecter les données autour et de les analyser. L'outil est-il utilisé ? L'intérêt des utilisateurs a-t-il augmenté ? Voilà des questions auxquelles on peut répondre en collectant des données. Pour des personnes passionnées par la data, être data-driven en récupérant et analysant ces données est une stratégie alléchante. Elle porte ses fruits. Cela ne nous empêche pas de combiner les deux approches : évaluation utilisateurs et analyse des données.

Pilote et AB testing

On n'est point obligé d'évaluer le produit sur l'ensemble des utilisateurs. On peut commencer par le faire tester à quelques-uns pour commencer et vérifier que tout va bien. Il y a plusieurs manières de réaliser cela. Nous allons parler de « pilote » et d'« AB testing ». On va choisir l'une ou l'autre des stratégies en fonction du contexte, de la taille de la cible et de l'accès aux utilisateurs. En réalité, on peut aussi utiliser les deux méthodes pour évaluer son système prédictif soit en parallèle soit en séquentiel.
La différence que je fais entre ce que j'appelle pilote et AB testing, c'est le côté manuel et l'interaction directe avec l'utilisateur. Dans le cas d'un pilote, on va interagir directement avec lui et recueillir son feedback. Cela peut se faire par l'intermédiaire d'une interface web ou d'un commentaire ajouté dans un dashboard. L'idée est aussi de tester le système prédictif sur un sous-ensemble de la population cible. Cela peut être un pays ou une région ou tout simplement un lot d'utilisateurs désignés. Cela me paraît toujours une bonne idée de tester sur un petit échantillon avant de lancer le produit en production. L'avantage de faire cela avec un « pilote », c'est qu'on embarque les utilisateurs dans ce processus. Cela signifie qu'ils sont au courant qu'on leur demande leur participation et avis. On peut aller les voir et ajuster le système prédictif par rapport à leurs attentes. On peut choisir ces utilisateurs et désigner des personnes qu'on pense davantage à même de nous donner du feedback intéressant. L'inconvénient de ce processus, c'est justement qu'on peut choisir et donc se tromper et donc produire une application biaisée. Notre échantillon ainsi choisi a assez peu de chances d'être représentatif de la population à laquelle on dédie le système. Même dans le cas où on l'ouvre à toute une région ou pays, cela comprend des dangers. Les comportements ne sont pas forcément les mêmes d'un endroit à un autre de la planète. Les prédictions elles-mêmes pourraient se trouver très adéquates pour une région et complètement inappropriées pour le reste du monde. C'est pourquoi il faut choisir son échantillon avec soin. Je nuancerais cependant en ajoutant que réaliser un pilote est peut-être plus judicieux que de ne pas tester du tout la performance et l'adhésion au nouveau système prédictif. Comment peut-on avoir un échantillon représentatif ? Il y a plusieurs solutions. L'aléatoire est la clef de tout. Notons aussi que dans un pilote, on n'est pas obligés de ne s'appuyer que sur les retours officiels des utilisateurs. On peut aussi collecter et analyser leurs données d'usage.

La construction d'un échantillon non biaisé et la validation par les données sont deux composants de l'AB testing. L'AB testing n'est cependant pas pour tout le monde. Le problème n'est pas qu'il est trop complexe à mettre en place. Ce n'est pas là le problème. Mais il faut avoir un certain nombre d'utilisateurs pour que cela devienne intéressant. Comment marche l'AB testing ? On prend souvent des exemples de design pour cela et je trouve que c'est intéressant. Supposons un site web qui veut tester deux fonds de couleur : orange et violet. Il voudrait savoir si l'une des deux couleurs a une incidence sur les utilisateurs. Restent-ils plus longtemps sur la page en fonction de l'une d'entre elles ? Est-ce que le comportement reste le même dans les deux cas ? Pour faire cela, les développeurs décident de séparer en deux l'expérience. Ils prennent un échantillon de leurs utilisateurs, disons 10% et soumettent à 50% d'entre eux la nouvelle couleur violette. Le reste de l'échantillon continue à voir la couleur orange. Il s'agit alors de savoir quelle couleur performe davantage. Pour cela, on va établir une ou des métriques à suivre : le temps resté sur la page, le nombre de fois où la page a été visitée, etc. Cela peut dépendre de ce que l'on cherche à faire. Parfois, on veut fidéliser. Parfois, on veut attirer de nouvelles personnes. À la fin de l'expérience, on établit un vainqueur. Supposons que

l'orange l'emporte. Cette couleur est alors diffusée à l'ensemble de la population. On peut imaginer des cas où aucune couleur ne l'emporte. Dans ce cas, on peut choisir celle qu'on préfère ou celle qui est la moins coûteuse à développer. Réaliser de l'AB testing n'est pas forcément complexe à mettre en place. On peut user les moyens du bord pour se faire. La meilleure manière d'y parvenir est d'utiliser ce qu'on a à disposition. J'ai connu un cas où le système de « feature flipping » avait été poussé à son maximum pour activer des fonctionnalités en fonction des utilisateurs au hasard. La difficulté en AB testing est davantage dans le fait de valider s'il y a un vainqueur et lequel est-il. Les débats sont importants sur la toile : statistiques fréquentistes versus statistiques bayésiennes ? Dans tous les cas, il est vrai qu'il peut y avoir des pièges comme la saisonnalité ou le fait qu'à l'apparition d'un nouveau système, les utilisateurs puissent avoir un comportement extrême, négatif ou positif, puis revenir à un comportement neutre plus tard. Il faut aussi veiller à ce que les AB tests ne s'influencent pas dans le cas d'une application. Dans le cas d'un système prédictif, on pourrait par exemple imaginer faire cela pour tester un nouveau modèle. Un cas intéressant, c'est celui des systèmes de recommandations difficiles à évaluer. On peut en mettre en production sous forme d'AB tests quelques-uns et voir lesquels performent le mieux. J'ai dit qu'il valait mieux avoir quelques utilisateurs pour faire de l'AB testing. Cependant, ce n'est pas une technique uniquement réservée aux grands de l'IT. On peut très bien le faire dans le cas d'une application de maintenance modeste par exemple. C'est pratique pour tester un nouveau modèle d'une manière générale et vérifier que tout va bien avant de le diffuser à tout un ensemble.

Canary testing

Dans les profondeurs de la terre, les mineurs étaient soumis à toutes sortes de dangers. L'un d'entre eux était le monoxyde de carbone. Il s'agit d'un gaz qui n'a pas d'odeur, mais qui est mortel. Une technique pour éviter cet accident était de placer des canaris dans une cage. Ces derniers sont plus sensibles au monoxyde de carbone. Quand les canaris arrêtent de chanter, c'est qu'il y a danger éminent et qu'il faut tout arrêter et fuir très vite avant d'y laisser sa peau. On retrouve la même idée avec le canary testing. On ne déploie pas tout à tout le monde. On va déployer au fur et à mesure en vérifiant que les canaris continuent à chanter. Concrètement, on déploie à 2% des utilisateurs. Si tout se passe bien, s'il n'y a pas d'alerte, on augmente la portion des utilisateurs d'1%. De nouveau, on vérifie que tout va bien. On continue à ajouter des pourcentages jusqu'à déployer à l'ensemble des utilisateurs. Le nombre de pourcentages varie selon les contextes et besoins. Pour vérifier si tout se passe bien, on doit bien entendu mettre en place du monitoring et des systèmes d'alerte. Comme avec l'AB testing, on teste d'abord que tout va bien. Il s'agit davantage d'un test technique que d'un test de satisfaction. De plus, un AB test peut durer des mois. Ce n'est pas le but du canary testing. C'est une technique qui s'avère utile pour mettre en production pas à pas sans peur. Cela s'applique plus souvent à des applications ayant quelques utilisateurs. Mais encore une fois, ce n'est pas que pour les grands de l'IT. C'est aussi un système qui peut paraître complexe à mettre en place. Pourtant, comme pour l'AB testing, on peut faire cela avec les moyens du bord. On peut réutiliser son système d'AB testing pour faire du canary testing. En termes de déploiement, c'est finalement la même chose. Il y aussi des outils de déploiement qui permettent de faire du canary testing.

Multi-Armed Brandit

Je n'ai jamais testé cette technique de mise en production. Pourtant, je ne me voyais pas l'évoquer. Elle semble avoir de plus en plus le vent en poupe pour des raisons que je conçois. Il s'agit à mon sens d'une nouvelle étape dans l'AB testing. Supposons que nous ayons dix versions à tester. Cela peut faire un peu beaucoup, mais nous choisissons ce nombre pour l'exemple. Ces dix versions sont dix modèles. On cherche à évaluer le plus performant. Pour cela, on sélectionne un échantillon de notre population auquel on soumet les dix modèles. Au départ, on distribue les modèles équitablement. Chaque modèle est évalué sur 10% de l'échantillon. Au fur et à mesure des tests, certains modèles performent davantage. Supposons qu'il s'agisse des modèles B, C et D. On va alors exposer plus d'utilisateurs à ces derniers. 20% de l'échantillon sera exposé au modèle B, 20% au C, 20% au D tandis que le reste se partage les modèles qui restent. Au fur et à mesure, on équilibre afin d'arriver à n'exposer les utilisateurs plus qu'au meilleur modèle.

Shadow model

La technique du shadow model met en parallèle plusieurs modèles, généralement deux. L'un est en production et donne des prédictions aux utilisateurs. Un autre est dans l'ombre. C'est le « shadow model ». Il donne des prédictions qui ne sont pas données à voir aux utilisateurs, mais sont mises de côté pour évaluer leur pertinence. De cette manière-là, on commence à tester son modèle. Ce n'est pas suffisant puisqu'on n'a pas d'interaction réelle avec l'utilisateur. On est dans une validation hors ligne mais qui se rapproche quand même un peu d'une validation en ligne. C'est donc un bon début.

Ode à la maintenance

On peut avoir tendance à voir la maintenance de manière négative. Dans certaines entreprises, il y a des équipes dédiées aux applications en maintenance et d'autres dédiées à la création de projets. Cette deuxième catégorie est vue comme celle dans laquelle il faut être. Ceux qui assurent la maintenance sont assez peu enviés. Il y a quelque temps, j'ai demandé à une personne quels modèles elle avait mis en production dans son ancienne entreprise. Elle m'a répondu qu'elle avait surtout passé du temps à améliorer l'existant, mais qu'en fait cela avait apporté beaucoup de plus-value, peut-être plus que d'avoir un nouveau projet sur le tapis. Autre fait en faveur de la maintenance : les projets open source que nous apprécions sont tous en maintenance. Ce n'est donc pas quelque chose à diaboliser.

Mais qu'est-ce que la maintenance finalement et comment la considérer ? Une fois qu'un système prédictif a eu sa première mise en production, on rentre finalement en phase de maintenance. On en établit généralement deux formes : la corrective et l'évolutive. Quand on fait de la correction, on vérifie que le système marche bien et on corrige quand on voit des bugs. On a toujours ce mode de maintenance. Il y a toujours des erreurs qui arrivent en production. C'est là où il faut déployer du monitoring pour savoir quand il y a des problèmes et y réagir. Ce monitoring en machine learning est double. On a à la fois du monitoring classique d'infrastructure et d'application. A cela, s'ajoute le monitoring de data drift. Dans une application plus classique, certains projets comptent sur les utilisateurs pour remonter les bugs. Cette manière de faire dans la programmation classique est déjà

périlleuse. Si on peut repérer un bug avant l'utilisateur et le corriger, c'est mieux. Cela peut être facilité avec un monitoring robuste. Pour les systèmes prédictifs, cela devient délicat de compter sur les utilisateurs. Certains n'apprécieront peut-être pas la prédiction qui leur est faite et le remonteront, mais il y a de fortes chances pour que certains continuent à l'utiliser sans ne rien remarquer. En machine learning, les erreurs sont la plupart du temps silencieuses. C'est assez dangereux ainsi de ne compter que sur l'avis des utilisateurs. La maintenance d'un système de machine learning a donc ses spécificités. Mais cela ne veut pas dire qu'il faut la craindre. C'est même quelque chose à souhaiter. Contrairement à ce que l'on pourrait penser, améliorer la solution d'un problème a de la plus-value. Il ne s'agit pas uniquement d'apporter de nouveaux projets sur la table pour apporter de l'intérêt business. La maintenance n'a peut-être pas d'effet « wahou », mais elle a plus de plus-value qu'on ne le croit. C'est une phase du projet à évaluer. Cela ne va pas forcément ralentir tout le monde. Cela veut plutôt dire qu'on va travailler différemment et apporter de la valeur différemment. Il y a fort à parier qu'une fois en production, le modèle aura des corrections et des évolutions. Ce n'est pas quelque chose qu'il faut redouter.

Nouveaux problèmes

L'organisation d'une équipe pour la mise en production est-elle vraiment importante ?

Ce n'est pas parce qu'on n'a pas la meilleure organisation du monde qu'on ne sera pas capable de mettre en production de beaux projets. Cependant, travailler son organisation aide. Peu importe les standards ou manières de faire qu'on décide d'avoir, s'accorder sur certains facilite le travail. Chaque projet est différent et certains nécessiteront sans doute de forts ajustements. Pourtant, s'accorder sur une manière d'arriver à notre but pour un projet est une étape qui permet de faire découler tout le reste. Il s'agit bien entendu de rester agile et de remettre en question ces processus.

Comment peut-on démarrer pour mettre en place Scrum ou une méthodologie qu'on ne connait pas ?

C'est une question intéressante. Il y a beaucoup de documentations et de témoignages en ligne et dans les conférences. Certaines personnes parviennent peut-être à se frayer un chemin sur cette base-là. J'avoue que ce n'est pas mon cas. Autant, je peux apprendre relativement toute seule à déployer du Kubernetes. Autant, je n'aurais jamais été en mesure de faciliter une rétrospective si je n'en avais pas eu l'expérience en tant que participante. La première fois que j'en ai facilité une, je n'en avais expérimenté qu'une auparavant. Ce fut une catastrophe. Il y a sans doute d'autres choses qui ont contribué à cet échec. Cependant, je crois aussi que je n'avais pas eu suffisamment le temps de comprendre la manière de dérouler le processus parce que je n'avais pas eu assez le temps de regarder d'autres faire. Aujourd'hui, j'anime ces séances plus sereinement. Je ne dis pas que c'est parfait, mais je crois humblement pouvoir dire que je suis loin de mon premier échec. Cela est dû, entre autres choses, au fait que j'ai eu le temps d'expérimenter en tant que participante les rétrospectives. Du coup, comment faire pour acquérir cette expérience si personne ne l'a dans l'équipe ? Je conseillerais plusieurs choses. Soit, on peut s'aider d'un coach Agile. Soit, on peut faire

une formation ou un workshop. Soit, si aucune de ces solutions n'est possible, on peut se lancer et apprendre de ses échecs. Je peux donner l'impression qu'il est impossible de se lancer soi-même. Je pense plutôt que c'est plus difficile, mais qu'il vaut mieux se lancer qu'attendre. Le principe même des rétrospectives est de mettre en place une boucle d'amélioration continue. L'idée est qu'on fera mieux la prochaine fois.

En conclusion, si au départ, j'ai hésité à réaliser ce chapitre, je ne le regrette pas du tout. Aujourd'hui, je vois au contraire qu'il avait toute sa place dans ce guide.

En résumé, il y a plusieurs méthodes d'organisation. Je sais qu'Agile n'est pas une méthode, mais voyons-le ainsi dans un but pragmatique. Il y a donc des méthodes qui viennent du monde des développeurs et des méthodes créées spécifiquement pour la data science. Lesquelles sont les plus appropriées ? Le plus important est de démarrer avec l'une d'entre elles, d'itérer et de ne pas hésiter à construire son propre chemin.

Les équipes autour de ces méthodes peuvent être organisées de différentes manières. Il y a des avantages dans les deux écoles : laboratoire spécialisé ou data scientistes citoyens au sein des équipes métier. Dans un laboratoire, les compétences sont centralisées. Les standards peuvent émerger et l'excellence est sur la data science et la mise en production. Quand les data scientistes sont disséminées dans différentes équipes, les compétences ne sont pas mises en commun. Dans le même temps, la data science est au plus proche des intérêts business. Elle est donc plus à même de comprendre les enjeux et d'y répondre. J'ai profité de ce chapitre pour parler du métier d'ingénieur machine learning dont le rôle principal pour moi est d'aider à la mise en production. Celle-ci doit être facile et apaisée. Cela va finalement du déploiement à la maintenance en passant par beaucoup d'éléments comme le clean code, l'infrastructure ou encore les stratégies d'évaluation du modèle dans la vraie vie. Les compétences sont vastes. Un ingénieur machine learning ne sera sans doute pas un expert dans tous les domaines. Le plus important est qu'il comprenne les spécificités du machine learning par rapport à la mise en production et la gestion d'un projet en général. Je ne crois pas que n'importe quelle personne orientée DevOps qui ne voudrait pas s'intéresser aux spécificités du machine learning puisse remplir ce rôle. Je ne dis pas qu'on ne peut pas faire de transition, mais qu'il faut être ouvert d'esprit et commencer par reconnaître que les systèmes prédictifs ne se mettent pas en production comme n'importe quel système informatique.
L'un des rôles primordiaux à mon sens de l'ingénieur machine learning, c'est-à-dire la tâche suprême qu'il devrait viser s'il n'y en avait qu'une, ce serait d'aider son équipe à ne plus avoir peur de mettre en production. Je rencontre souvent des gens effrayés par cela. Je trouve cela regrettable. Nous ne créons de la valeur que si nous allons en production. Il est vrai qu'il ne s'agit pas de mettre n'importe quoi en production parce que c'est aussi comme cela qu'on crée des catastrophes. C'est pourquoi il existe tout un champ de réflexion autour du sujet. Beaucoup de solutions et stratégies ont été proposées. Il s'agit de mettre en production le plus vite possible le moins de choses possible. Le déploiement continu est une arme certaine pour y parvenir. Ensuite, il existe plusieurs stratégies automatiques et manuelles pour se débarrasser de ses peurs et intégrer du code au fur et à mesure : feature flipping, évaluation utilisateurs, pilote, AB testing, canary testing et multi-armed brandit. Il ne faut pas hésiter à venir y piocher. Ces méthodes peuvent paraître effrayantes et complexes. Mais les moyens d'y parvenir sont flexibles. Il existe des outils qui font le travail à notre place. On peut aussi commencer graduellement avec juste un fichier de configuration.

Conclusion

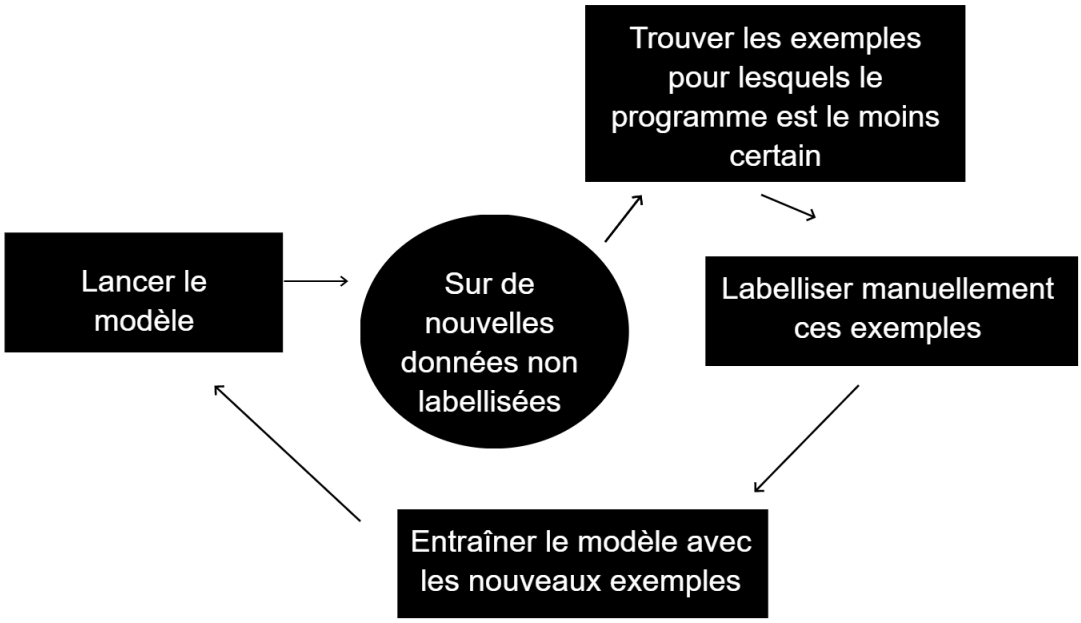
Aperçu d'autres formes de machine learning

J'avais prévu de concentrer ce guide sur l'apprentissage automatique supervisé en mode batch. C'est ce que j'ai fait. Cependant, pour finir j'aimerais au moins évoquer rapidement d'autres formes de machine learning qui peuvent avoir un impact en production.

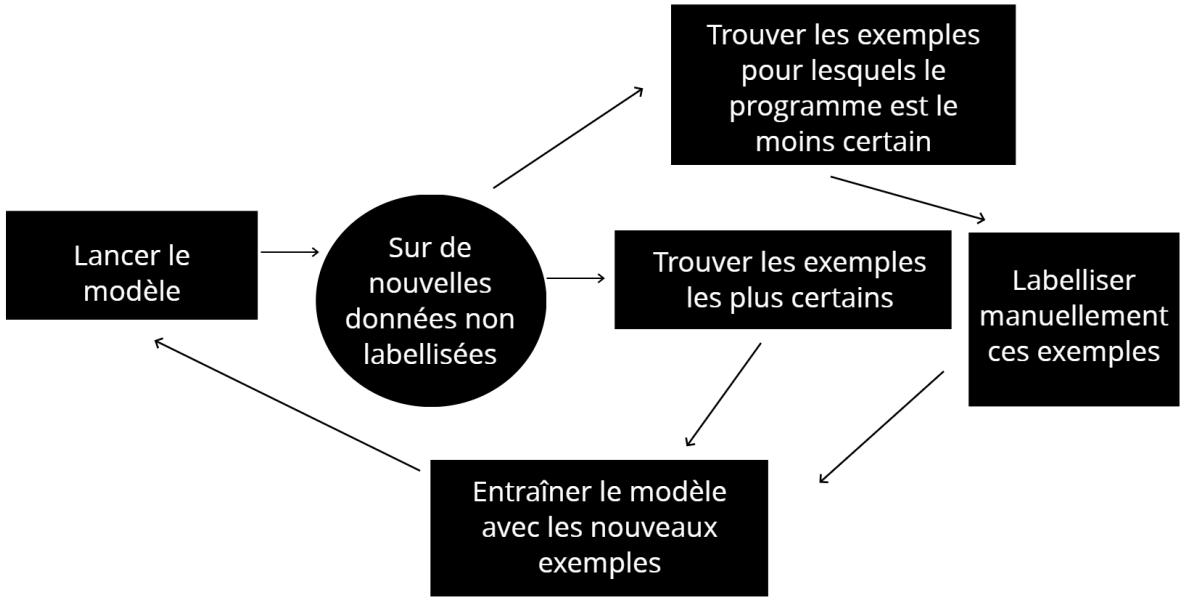
Active learning

La première c'est l'active learning. Labelliser les data en apprentissage supervisé peut être long et fastidieux. Il est parfois possible de trouver les labels. D'autres fois, c'est plus difficile et cela doit être réalisé à la main. C'est un travail qui apporte peu de plus-value et est assez ennuyeux. De plus, c'est onéreux de devoir faire les choses ainsi. Heureusement, il y a plusieurs manières de s'aider dans cette tâche.
On peut faire de l'augmentation de data. Par exemple, avec une image, on peut en extraire plusieurs autres. On peut prendre une image de chat, la renverser, la rogner, la retourner, etc. À partir d'une image qu'on a labellisée à la main, on obtient plusieurs autres images automatiquement labellisées. Il existe des librairies comme Codebox software pour réaliser cela.
Le machine learning non supervisé est aussi une manière de pallier le problème de la labellisation. Par exemple, supposons que nous voulions extraire des anomalies d'un ensemble. Avec l'apprentissage supervisé, nous pouvons labelliser plusieurs exemples et ensuite entraîner le modèle. À la place, on pourrait faire de l'apprentissage non supervisé où on regrouperait dans des clusters les éléments qui se ressemblent. On pourrait faire cela avec KMeans ou KMedoids. Les groupes vraiment différents des autres pourraient être des anomalies. C'est un exemple simple, mais vous pouvez à partir de là en imaginer des plus complexes.
Avec le reinforcement learning, on apprend à partir d'expériences. On a une cible. Un agent essaye alors de performer des actions. Ses succès et échecs vis-à-vis de la cible permettent d'entraîner le modèle dans la bonne direction. Un jour, quelqu'un m'a dit : « Le reinforcement learning, c'est juste du machine learning supervisé avec des data. Seulement, tu obtiens ces données avec le temps ». Cela permet d'éviter le goulot d'étranglement qu'est la labellisation.
Enfin et c'est le nom de cette partie, il y a l'active learning qui permet d'optimiser l'acte de poser des labels sur les choses. Le système extrait les données qui ont le plus besoin d'être labellisées. Il fait une demande manuelle de labellisation sur certains cas bien identifiés. Ce fait-là dépend de la stratégie utilisée. Je vais traiter de deux d'entre elles.

La première consiste à lancer le modèle qu'on a préentraîné sur des données non labellisées. On repère les éléments que le modèle a le plus de mal à prédire. On demande une labellisation manuelle pour ces éléments. On entraîne le modèle en ajoutant ces nouvelles données.



On peut aussi faire du machine learning semi-supervisé. En plus de ce qu'on faisait déjà, le système va regarder les prédictions pour lesquelles il est le plus sûr et les enregistrer comme des données labellisées.



C'est donc ce qu'on appelle l'active learning. Ce concept, si on l'utilise, fait alors partie d'une mise en production intégrante. C'est pourquoi je voulais l'évoquer. C'est une manière différente d'envisager la labellisation et le réentraînement. Ce dernier n'est pas commandé parce que le modèle décroît. Ce n'est pas non plus lié à une fréquence. Du moins, ce n'est pas la seule manière de lancer un

réentraînement. Le système d'active learning lance le réentraînement. Il doit être testé, automatisé et monitoré.

Apprentissage par renforcement ou reinforcement learning

J'avais prévu de ne pas en parler dans l'ensemble du guide. C'est ce que j'ai fait. Ici, je voudrais simplement l'évoquer pour en voir quelques spécificités dans une mise en production. Pour moi, les deux grands éléments qui changent avec le reinforcement learning, c'est :
- La façon d'acquérir les données
- Le temps réel

Les données sont acquises au fur et à mesure. Cependant, on peut commencer avec des données initiales obtenues grâce à de l'apprentissage supervisé classique. Mais ensuite, le principe est de faire agir un agent. Il acquière alors ce dont il a besoin au fur et à mesure.
Le reinforcement learning, c'est aussi du temps réel. Cette manière de développer est plus dure que le mode batch. On dit souvent que le machine learning est complexe. On oublie cependant d'en célébrer aussi les facilités. L'une d'entre elles à mon sens est le mode batch assez répandu. En reinforcement learning, on perd cette facilité. On doit alors s'ajuster et savoir entraîner au fur et à mesure. Les technologies et techniques vont donc varier. Il n'y a pas de fréquence de réentraînement par exemple. Des outils plus orientés temps réel ou presque temps réel peuvent alors être utilisés. Je pense à Kafka par exemple. Ce dernier est un logiciel un peu complexe à prendre en main, mais qui peut faciliter la vie quand on sort du mode batch.
Le reinforcement learning a donc des spécificités qui peuvent changer une mise en production. On ne monitore pas du temps réel comme on monitore du batch. On ne gère pas les erreurs de la même manière non plus. En mode batch, on a le temps. En mode temps réel, c'est plus compliqué.

Apprentissage non-supervisé

Je n'ai pas non plus parlé d'apprentissage non-supervisé. Je ne pense pas que l'apprentissage non-supervisé soit aussi éloigné de l'apprentissage supervisé que le reinforcement learning ne l'est, du moins en regard de la mise en production. L'apprentissage non supervisé est différent, mais ne réclame pas autant de challenge que l'apprentissage supervisé. Dans l'apprentissage non supervisé, on ne labellise pas les données. Il s'agit davantage de découvrir des structures au sein même des données. Les phases d'entraînement et de prédiction au cœur de l'apprentissage non supervisé disparaissent alors. Cependant, on a toujours un processus qu'on peut versionner avec ses données, automatiser, donner à voir dans un dashboard, une API web, etc. De mon expérience, la difficulté de l'apprentissage non supervisé, c'est le monitoring des performances. Par exemple, quand on réalise des clusters, il y a des manières automatiques de vérifier que ces derniers font sens les uns avec les autres. C'est plutôt utile quand on veut faire de la découvrabilité. Mais si nous voulons utiliser ces clusters pour faire de la détection d'anomalies par exemple, c'est plus compliqué. Comment savoir automatiquement que les anomalies en sont bien ? Je n'ai pas de réponse magique. On peut s'appuyer sur les retours utilisateurs par exemple ou ce genre de choses. En tous les cas, si on utilise l'apprentissage non supervisé pour le pousser dans ses limites, c'est à mon sens le grand changement quand on pense à la mise en production.

Cela peut paraître étrange de consacrer une section même si courte à des éléments que nous n'avons pas du tout évoqués dans le reste du guide. Je ne pouvais cependant pas vous laisser en faisant comme s'ils n'existaient pas. Ce sont pour moi trois éléments du machine learning qui changent tout ce dont nous avons dit auparavant. Il y en a sans doute d'autres. Ce sont ceux qui me sont venus à l'esprit comme étant les plus percutants. Ils viennent parfois contredire ou au moins donner une autre vision de ce qu'on pourrait considérer comme un standard du machine learning : l'apprentissage automatique supervisé en mode batch.
L'active learning est une méthode qui a pour ambition de tordre le coup au coût de l'étiquetage en incluant cela de manière semi-automatique.
Le reinforcement learning est aussi différent. On apprend au fur et à mesure. Il n'y a pas une grande phase d'entraînement suivie d'une phase de prédictions.
Quant à l'apprentissage automatique non supervisé, il n'y a même pas à proprement dit de prédictions. Du moins, on n'apprend pas du tout de la même manière. Le monitoring aussi est très différent.

Le mot de la fin

J'ai parlé de beaucoup de sujets dans ce guide qui je l'espère vous inspireront. J'ai livré ici mes expériences et mon point de vue. Je vais profiter de cette conclusion pour résumer les éléments qui sont les plus importants. Comme nous l'avons vu, mettre en production des modèles signifie beaucoup de choses. Il s'agit de s'assurer d'avoir des données pérennes mises à jour régulièrement sans encombre, versionnées et monitorées pour être sûrs de ce que l'on reçoit. Il s'agit aussi d'automatiser l'entraînement et l'inférence, les deux grandes parties de notre modèle. J'inclus dans l'entraînement le feature engineering. En fonction des options choisies, il y a plusieurs manières d'y parvenir. Il s'agit en tous les cas de versionner son modèle, son code et ses données. Il s'agit aussi de tester ce même ensemble. Enfin, il faut déployer sur un serveur ces éléments-là. Cela peut demander tout un champ de compétences différentes. Parfois, les prédictions sont données dans une API web, parfois dans un dashboard. Dans tous les cas, il s'agit d'éviter de faire des choses manuellement, de penser à tester l'ensemble des solutions choisies et de s'aider des outils qui ont été conçus pour rendre les choses plus simples pour les data scientistes. En web, par exemple, la combinaison FastAPI et Streamlit est une idée pour faire du web quand on n'en a pas forcément l'habitude. Le monitoring est une chose importante pour tout système informatique. La data science n'y coupe pas. Plus que cela, elle nécessite même davantage de monitoring avec le concept de drift qui a une importance considérable. Il ne faut pas s'attendre à ce qu'un modèle entraîné et tuné à un instant T fonctionne toujours aussi bien des années après. Il y a peu de chance pour que cela arrive ne serait-ce que parce que le monde évolue. Il vaut mieux s'y préparer que tomber des nues. Même si le drift est encore un sujet ouvert de la recherche, de nombreuses solutions prêtes à l'emploi ou tunables existent. Il y a des librairies et des plateformes qui font ce travail. Je recommanderais de ne pas se méprendre sur le travail que cela demande. Cela peut demander une personne dédiée à temps plein à ce travail. C'est la raison pour laquelle, je conseillerais si votre équipe ne peut pas se permettre cela de passer par une personne dédiée pour a minima mettre les premiers éléments en place.
Ensuite, mettre en production peut renvoyer à des éléments comme la mise à l'échelle ou l'interpréta-

bilité. Cela peut être optionnel et dépend de votre contexte. La mise à l'échelle, c'est finalement être à même d'avoir un modèle qui performe avec de grosses données. Il y a de nombreux frameworks qui permettent de faire cela. HDFS est le plus connu pour la distribution des données. Spark est le plus connu pour la distribution du travail. Ces frameworks s'ils permettent une mise à l'échelle, sont aussi demandant en termes de compétences. Je conseillerais de conseiller par une formation. Le développement distribué comporte quelques surprises. En avoir conscience économise des heures de sueur. Quant à l'interprétabilité/explanabilité, on rapproche encore trop souvent à mon goût cela d'un besoin éthique. Je ne dis pas que cela n'est pas utile pour éviter les biais et au moins les comprendre. Cependant, je ne crois pas que l'interprétabilité soit réservée uniquement aux grands de l'IT ou à des systèmes prédictifs du domaine public. L'utilité peut être marketing. Une recommandation expliquée a davantage de poids. Cela donne plus de confiance à l'utilisateur d'une manière générale. Cela peut aussi servir au debug. Aujourd'hui, il existe de nombreuses librairies qui permettent en quelques lignes de code d'avoir une idée de pourquoi on reçoit telle ou telle prédiction. Ce n'est donc pas forcément un travail éreintant.
Même si je ne souhaitais pas m'épancher sur le sujet, j'ai tout de même tenu à évoquer quelques spécificités de l'active learning, de l'apprentissage non supervisé et du renforcement learning. Je ne sais pas quelle place ces éléments prendront demain dans les entreprises qui font du machine learning. Cependant, je ne me voyais pas vous laisser sans les avoir évoqués. Par-là, je voulais surtout signifier que tout le reste du guide vaut surtout pour l'apprentissage supervisé encore très répandu aujourd'hui.
Enfin, tout cela ne peut se faire sans une organisation d'équipe et des stratégies qui aident à ne plus avoir peur de la mise en production. Entre toutes les méthodes qui existent (Agile, CRISP-DM, etc.), je conseillerais surtout d'en choisir une et d'itérer en n'hésitant pas à piocher de partout dans ce qui peut être utile. La meilleure manière d'organiser ensuite son ou ses équipes comporte des avantages et des inconvénients dans tous les cas. Le but de tout cela est bien de ne plus avoir peur de la production. On peut utiliser différentes stratégies pour cela. Au-delà du fait de tester et de monitorer, on peut utiliser du feature flipping, de l'AB testing, du canary testing ou encore du multi-armed-brandit. Ce dernier a vu le jour récemment. Cela signifie qu'il faut rester alerte et ne pas rester camper sur ses positions. Il faut reconnaître ses difficultés et regarder autour de soi les solutions qui existent pour y faire face.

Pour conclure, les conseils finaux que je donnerais seraient les suivants : tester et monitorer le code, les modèles et les data, mettre en place une boucle de feedback continue, embrasser les erreurs qui viendront et en faire des forces. Il s'agit de penser itératif et amélioration continue. De plus, il ne faut pas avoir peur de la mise en production et des différentes étapes que cela renferme. Les choses viennent peu à peu. Je conseillerais si on peut de ne pas hésiter à se faire aider avec des formations pour mettre le pied à l'étrier. Certains sujets comme le calcul distribué peuvent paraître challengeant. Quelques heures de workshop permettent de dédramatiser les choses.
Enfin, notre monde évolue. La data science évolue. Il s'agit donc de ne pas s'arrêter là. Il s'agit de rester alerte. De plus, la bonne attitude est de ne pas hésiter à se dire qu'on s'est trompé ou qu'on peut faire mieux maintenant qu'on a plus de connaissance ou que quelque chose de mieux est arrivé. Nous avons parlé d'outils et techniques qui un jour pour sûr seront dépassés. Il n'y a pas non plus de « silver bullets ». On a toujours besoin d'adapter les choses à son contexte. Si certains se ressemblent,

il n’y en pas deux pareils.

Je vous remercie pour votre lecture. J’espère que ce guide vous aura aidé dans votre entreprise.

Les resources qui m'ont aidé à écrire ce guide

Livres :
• Machine Learning Engineering – Andriy Burkov
• Data Science in Production : Building Scalable Model Pipelines with Python – B.G Weber
• Big data, que sais-je - Pierre Delort
• Interpretable Machine Learning - Christoph Molnar
• Clean Code – Robert C.Martin
• Clean Coder – Robert C.Martin
• Clean Agile – Back to Basics – Robert C. Martin
• Agile Coaching - Rachel Davies and Liz Sedley
• Refactoring : Improving the Design of Existing Code – Martin Fowler
• Extreme Programming Explained – Kent Beck
• Test Driven Development by Example – Kent Beck
• Test Driven Development : A practical Guide – David Astels
• La Formule du Savoir : Une philosophie unifiée du savoir fondée sur le théorème de Bayes – Lê Nguyên Hoang

Papiers :
• Hidden Technical Debt in Machine Learning Systems - D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, ToddPhilli, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo et Dan Dennison
• Characterizing Concept Drift - Geoffrey I Webb, Roy Hyde, Hong Cao, Hai-Long Nguyen et François Petitjean.
• Survey of distance measures for quantifying concept drift and shift in numeric data - Igor Goldenberg et Geoffrey I Webb
• Monitoring and explainability of models in production - Janis Klaise, Arnaud Van Looveren, Clive Cox, Giovanni Vacanti et Alexandru Coca
• "Why Should I Trust You ?" : Explaining the Predictions of Any Classifier - Marco Tulio Ribeiro, Sameer Singh, Carlos Guestrin
• Failing Loudly : An Empirical Study of Methods for Detecting Dataset Shift - Stephan Rabanser, Stephan Günnemann, Zachary C. Lipton

Articles :
• Continuous Delivery for machine learning - Danilo Sato, Arif Wider et Christophe Windheuser,
• Computing machinery and intelligence - 1950 par Alan Turing
• Faire des tests unitaires est impossible - Xavier Detant
• Productionizing Machine Learning : From Deployment to Drift Detection - Joel Thomas et Clemens

Medwald
• How to detect drifting models - Martin Schmidtz
• To Apply Machine Learning Responsibly, We Use It in Moderation – New York Times
• La revue de code bienveillant - Gilles Roustan

Conférences :
• Simplifying Model Management with Mlflow - Matei Zaharia - Corey Zumar