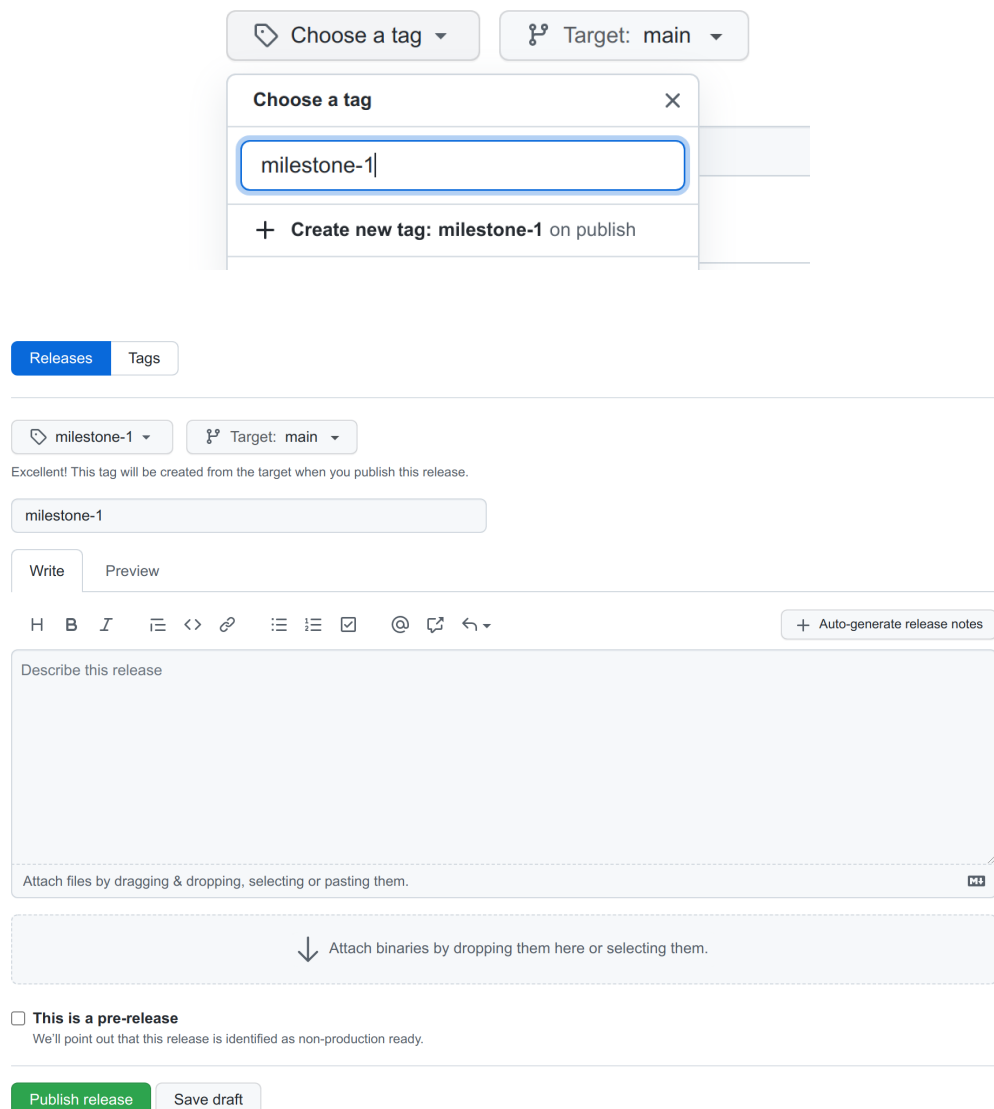


# Projet IFT 6758 : Milestone 3

Publié : Nov 29, 2022

Échéance : Dec 20, 2022

**Important: Avant de commencer à travailler sur le milestone 3, publiez une version pour le milestone 2 en suivant les instructions [ici](#). Vous n'avez pas besoin d'ajouter quoi que ce soit dans la description, ni de télécharger des fichiers binaires. Utilisez `milestone-2` à la fois comme balise et titre de version, par exemple:**



Choose a tag Target: main

Choose a tag X

milestone-1

+ Create new tag: milestone-1 on publish

Releases Tags

milestone-1 Target: main

Excellent! This tag will be created from the target when you publish this release.

milestone-1

Write Preview

H B I

+ Auto-generate release notes

Describe this release

Attach files by dragging & dropping, selecting or pasting them.

Attach binaries by dropping them here or selecting them.

☐ This is a pre-release  
We'll point out that this release is identified as non-production ready.

Publish release Save draft

Jusqu'à présent, vous avez interagi avec une partie importante du flux de travail de la science des données, y compris la gestion des données, l'EDA, la création de visualisations, l'ingénierie des caractéristiques et la modélisation statistique. Donc, nous avons nos modèles, nous avons terminé maintenant, n'est-ce pas ? Eh bien, si vous voulez que personne n'utilise tout le travail que vous avez fait, oui ! Sinon, nous devons avoir une certaine compréhension de ce qui se passe une fois cette phase initiale du projet de science des données est terminée.

Pour cette dernière étape, l'accent sera mis sur la science des données en « production ». Ceci est entre guillemets car il est important d'avoir une **clause** : **il ne s'agit en aucun cas d'un guide pratique sur la manière de créer des systèmes ML robustes pour entreprise**. Au lieu de cela, le but de cette étape est de vous donner un peu de contexte et une idée des types d'outils et d'idées qui pourraient être utilisés pour déployer des modèles comme services utiles pour une entreprise, bien qu'à une échelle beaucoup plus petite. De plus, on ne se focalise que sur la notion de [déploiement de modèle](#); mais n'aborde pas d'autres concepts importants tels que le [monitoring](#).

En utilisant tous les modèles que vous avez déjà créés (et enregistrés dans Comet ML !), vous créerez une application de service de modèle très simple qui sera déployée dans un [conteneur Docker](#) et accessible via un REST API . Vous allez également créer un « client de jeu en direct », qui récupérera les événements de tir des matchs de la LNH en direct (ou historiques), prétraitera les données en caractéristiques compatibles avec votre modèle, puis fera des requêtes à votre service de modèle pour obtenir les prévisions de buts attendus pour chaque événement de tir.

Le client de jeu en direct sera ensuite intégré dans un joli tableau de bord interactif à l'aide de [Streamlit](#).

<b>Une note sur le plagiat</b>	<b>3</b>
<b>Objectifs d'apprentissage</b>	<b>4</b>
<b>Déploiement du modèle (Modèle en tant que service)</b>	<b>4</b>
Services	4
Communication	5
Gestion des dépendances	5
<b>Livrables</b>	<b>6</b>
Détails de la soumission	6
<b>Tâches et questions</b>	<b>7</b>
1. Application Modèle Flask (30 %)	7
2. Clients (10 % )	8
Client de Modèle	9
Client de Jeu	9
3. Docker partie 1 - Modèle (15%)	10
4. Application Streamlit (30%)	12
5. Docker partie 2 - Streamlit (15%)	14
6. BONUS: Streamlit (5%)	15
<b>Et voilà !</b>	<b>16</b>
<b>Évaluations de groupe</b>	<b>17</b>
<b>Références utiles</b>	<b>18</b>

## Une note sur le plagiat

*L'utilisation de code/modèles provenant de ressources en ligne est acceptable et c'est courant dans la science des données, mais soyez clair pour citer exactement d'où vous avez pris le code si nécessaire. Un simple extrait d'une ligne qui couvre une syntaxe simple à partir d'un article ou d'un package Stack Overflow ne justifie probablement pas une citation, mais la copie d'une fonction qui effectue un tas de logique qui crée votre figure le fait. Nous espérons que vous pourrez utiliser votre meilleur jugement dans ces cas, mais si vous avez des doutes, vous pouvez toujours citer quelque chose pour être sûr. Nous effectuerons une détection de plagiat sur votre code et vos livrables, et il vaut mieux prévenir que guérir en citant vos références.*

*L'intégrité est une attente importante de ce projet de cours et tout cas suspect sera poursuivi conformément à la [politique très stricte](#) de l'Université de Montréal. Le texte complet des règlements à l'échelle de l'université peut être trouvé [ici](#). Il est de la responsabilité de l'équipe de s'assurer que cela est suivi rigoureusement et des actions peuvent être prises sur des individus ou sur l'ensemble de l'équipe selon les cas. de s'assurer que cela est suivi rigoureusement et*

que des mesures peuvent être prises individuellement ou sur l'ensemble de l'équipe selon le cas.

## Objectifs d'apprentissage

- **Modèle "Déploiement"**
  - Modèle en tant que service
  - Gestion des dépendances de modèles (Docker)
  - API REST
- **"Production"/Live Workflows**
  - Écriture
- **Visualisation interactive**
  - Rendre vos prédictions facilement accessibles à l'aide de Streamlit

## Déploiement de Modèle (Modèle comme Service)

(Cette introduction est inspirée en grande partie de [Full Stack Deep Learning - Lecture 11](#))

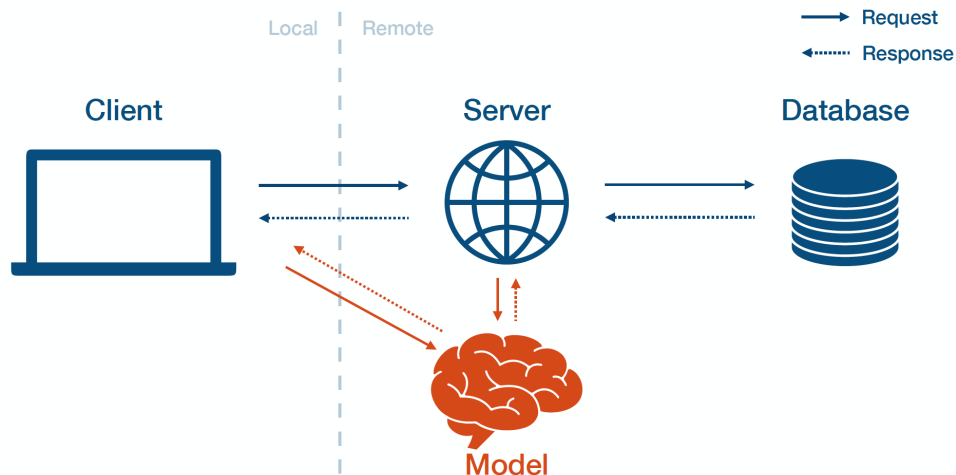


Figure 1 : Modèle comme service ([source](#))

Il existe plusieurs façons d'utiliser des modèles d'apprentissage statistique/machine en production, mais nous nous concentrerons sur l'idée de *modèle comme service*, où un modèle est déployé en tant que son propre service; les utilisateurs et les clients interagissent avec le modèle en faisant des demandes spécifiques au service de modèle et en recevant des réponses de celui-ci.

## Services

Avant de commencer, vous vous demandez peut-être qu'est-ce qu'un service? Cette partie [pourrait être son propre cours](#), mais la seule chose que vous devez vraiment comprendre pour ce projet est qu'il s'agit d'un style architectural de génie logiciel qui construit des composants

logiciels réutilisables et interopérables, par opposition à un seul gros logiciel monolithique. Fondamentalement, cela signifie que vous aurez de nombreux petits éléments autonomes (services) qui sont des « boîtes noires » pour leurs consommateurs (c'est-à-dire que le consommateur n'a pas besoin de savoir comment le service fonctionne). Les services fonctionnent généralement (mais pas toujours) dans leur propre environnement avec du hardware dédié, et ils peuvent également être éphémères (c'est-à-dire mis en place et supprimés dynamiquement au fur et à mesure que l'utilisation fluctue).

## Communication

Ces services peuvent interagir les uns avec les autres de plusieurs manières. Un [API REST](#) est un moyen de communiquer via des requêtes HTTP; cette méthode est très courante et fonctionne bien pour des petites données (comme des données textes ou JSON), mais ne fonctionne pas aussi bien pour les données plus volumineuses (par exemple, des fichiers binaires, des images, etc.). D'autres alternatives existent telles que [gRPC](#), [Kafka](#), [ZeroMQ](#), etc., mais cela relève davantage du domaine des ingénieurs logiciels et ne fait pas partie des objectifs de ce cours. Pour nos besoins, un API REST sera suffisant mais sachez que ce n'est pas le bon outil pour chaque tâche et qu'il **n'existe pas de solution standard pour formater les données qui vont dans un modèle ML**. Nous utiliserons [Flask](#) pour créer notre application Web de prédiction simple.

## Gestion des dépendances



Pour exécuter un modèle, vous avez besoin du code, des dépendances de code et des coefficients du modèle, qui doivent tous être disponibles pour votre service. Les coefficients de modèle sont assez portables, mais le code et les dépendances de code peuvent être beaucoup plus difficiles à gérer. Une solution courante à ce problème est la **conteneurisation**; un **conteneur** est une seule unité de logiciel qui devrait idéalement fournir un environnement cohérent peut importe où<sup>1</sup> vous le déployez. Notez que les conteneurs ne sont pas des

---

<sup>1</sup> Il y a quelques subtilités lorsqu'il s'agit de créer des conteneurs sur différentes architectures de processeur (par exemple x86 (la plupart Intel et AMD) et ARM).

machines virtuelles (VM) - les VM nécessitent que l'hyperviseur virtualise le hardware physique et exécutent généralement plusieurs systèmes d'exploitation simultanément. Les conteneurs, quant à eux, partagent le même host kernel mais sont exécutés dans des environnements d'espace utilisateur isolés. Si ces différences entre les machines virtuelles et les conteneurs n'ont pas beaucoup de sens pour vous, ce n'est pas grave - le principal point à retenir est que les conteneurs sont plus rapides et plus légers que les machines virtuelles.

La plate-forme la plus populaire pour la conteneurisation est **Docker**, que nous utiliserons pour cette étape. Il y a quelques concepts de base avec lesquels vous devriez être familiarisé :

1. **Dockerfile**: Fichier qui définit comment créer une image
2. **Image**: Un environnement construit avec toutes les dépendances
3. **Conteneur**: L'endroit où les images sont exécutées
4. **Dépôt**: Héberge différentes versions d'une image
5. **Registre**: Un ensemble de dépôts

Ceux-ci seront abordés plus en détail dans les définitions des tâches de ce milestone.

## Livrables

Vous devez soumettre:

1. Le code de votre équipe **reproductible**
2. Ajoutez les comptes IFT 6758 pertinents à votre dépôt github et à votre espace de travail comet.ml (si vous rencontrez des problèmes avec cela, nous en informons par courriel comme la dernière fois).

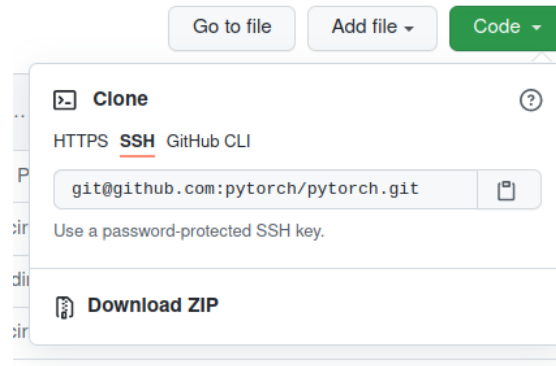
Il n'y aura **pas de composant de blogpost** pour cette étape finale car vous serez principalement évalué sur la soumission de votre code.

## Détails de la soumission

Pour soumettre votre projet, vous devez :

- ☐ Publier votre soumission de milestone finale dans la branche **milestone-3** branche
  - ☐ Créer une version similaire à ce que vous avez fait pour les milestones 1 et 2, mais avec la balise **milestone-3** balise
- ☐ Soumettez un ZIP de votre base de code à [gradescope](#)
- ☐ Ajoutez le [compte Github IFT 6758 TA](#) (@ift-6758) à votre dépôt git en tant que *spectateur* ou *contributeur*, si vous ne l'avez pas déjà fait ou si vous utilisez un nouveau dépôt.
- ☐ Assurez-vous `ift6758` est ajouté à votre espace de travail comet.ml; ceci devrait déjà être fait depuis le milestone 2 (settings > **+Collaborators** )

Pour soumettre un ZIP de votre dépôt, vous pouvez le télécharger via l'interface utilisateur Github :



## Tâches et questions

Les tâches requises pour le milestone 3 sont décrites ici. Cette fois, vous n'aurez pas besoin de soumettre un article de blog ; vous serez évalué sur vos soumissions de code. En termes d'évaluations, vous serez noté dans le style des revues de code pour une partie de votre travail, et nous **exécuterons votre code** dans le système conteneurisé que vous produirez pour la tâche finale.

**Le dépôt de projet GitHub mis à jour peut être trouvé [ici](#).**

### 1. Service de prédiction Flask (30%)

Nous allons d'abord créer une application Flask simple qui fournira les prédictions de nos modèles que nous avons créés dans Milestone 2. Nous ne nous soucierons pas encore d'intégrer cette application dans un conteneur Docker - pour le moment, nous nous concentrons uniquement sur l'exécution de cette application sur notre environnement local. Votre première tâche consiste à créer une application Flask qui implémente les endpoints suivants :

- **/predict** - Prédit la probabilité que le tir soit un but compte tenu des inputs
  - **Input:** Les caractéristiques utilisées par votre modèle, compatibles avec `model.predict()` ou `model.predict_proba()`
    - Indice : [df.to\\_json\(\)](#)
  - **Sortie:** Les prédictions pour toutes les caractéristiques; le output
- **/logs** - Récupère les logs écrits par votre application
- **/download\_registry\_model** - encapsule essentiellement la même fonction dans l' [API comet ml](#) pour télécharger un modèle et mettre à jour le modèle actuellement chargé. Vous devez :
  - Vérifier si le modèle que vous cherchez est déjà téléchargé
    - Si oui, chargez ce modèle et écrivez dans les logs que le modèle change
    - Si non, essayez de télécharger le modèle :
      - Si ça réussit, chargez ce modèle et écrivez dans les logs que le modèle change.
      - En cas d'échec, écrivez dans les logs qu'il y a eu un échec et conservez le modèle actuellement chargé.

Dans le nouveau dépôt de projet, vous trouverez un **app.py** dans le dossier `serving`. Ce fichier contient un template de la structure que vous devez implémenter. Vous **devriez ne pas** mettre de fonctionnalité spécifique à une tâche dans votre application Flask; tout prétraitement des données (comme la création de toutes les caractéristiques nécessaires à votre modèle) doit être effectué avant de faire une demande au service de prédiction. Votre application doit se comporter de la même manière qu'un objet de modèle SK-Learn.



Pour exécuter l'application, si vous êtes dans le même dossier que **app.py**, vous pouvez exécuter l'application à partir de la ligne de commande en utilisant [gunicorn](#) (Unix) ou [waitress](#) (Unix + Windows) :

```
$ gunicorn --bind 0.0.0.0:<PORT> app:app
$ waitress-serve --listen=0.0.0.0:<PORT> app:app
```

gunicorn or waitress peut être installé via :

```
$ pip install gunicorn
$ pip install waitress
```

At à ce stade, vous pouvez tester l'application avec les données d'entraînement/validation existantes que vous avez utilisées dans Milestone 2, et envoyer un ping à l'application directement à l'aide de la bibliothèque de requêtes Python, par exemple:

```
X = get_input_features_df()
r = requests.post(
    "http:// 0.0.0.0:<PORT>/predict",
    json=json.loads(X.to_json())
)
print(r.json())
```

De plus, une fois que vous aurez implémenté **/logs**, vous pourrez voir tous les logs en accédant à <http://0.0.0.0:<PORT>/logs> dans votre navigateur.

### Évaluation :

Le code ne sera pas exécuté à ce stade (il sera testé dans la partie 4). Cependant, des points seront attribués à cette section en fonction de :

- Les trois endpoints (**/predict**, **/logset** **/download\_registry\_model**) sont correctement implémentés
- Le service peut télécharger des modèles stockés dans votre registre de modèles Comet ML et les échanger *sans redémarrer l'application*. Les cas où le nouveau modèle n'est pas téléchargé correctement (par exemple, n'existe pas) sont traités d'une manière qui ne brise pas l'application.
- Vous enregistrez des messages de débogage appropriés et informatifs dans les logs; vous devriez pouvoir déboguer votre application à partir des logs sans aller manuellement dans l'application (et plus tard, dans le conteneur Docker).

**Vous devez uniquement vous concentrer sur les modèles que vous avez produits dans Milestone 2 partie 5, avec l'ensemble de base de caractéristiques avancées. C'est correct s'il manque un tas de vos meilleurs modèles, le but est simplement de créer une sorte de système simple qui démontre la fonctionnalité de base.**

## 2. Clients (10%)

Nous souhaitons maintenant créer une meilleure interface pour l'application Flask que nous venons de créer, ainsi qu'une interface agréable pour récupérer les données des matchs NHL en direct. Pour cela, vous devrez créer 2 clients : un **client fournisseur** et un **client jeu**. Vous recevrez un fichier template modèle pour le client fournisseur (dans `ift6758/ift6758/client/serving_client.py`). Pour le client de jeu, vous avez déjà écrit la majeure partie du code pour extraire les caractéristiques - vous devrez trouver l'implémentation qui a le plus de sens compte tenu des exigences décrites ci-dessous.

### Client fournisseur

Maintenant que notre modèle SKLearn est servi via une application Flask, nous voulons un moyen d'interagir avec lui sans avoir à appeler explicitement `requests.post` à chaque fois que nous voulons soumettre une demande. Implémentez les trois méthodes définies dans `ift6758/ift6758/client/serving_client.py` Ces fonctions seront très simples et directes à implémenter.

### Évaluation

- Les trois méthodes (**`predict(...)`**, **`logs()`** et **`download_registry_models(...)`**) sont correctement implémentées. Les signatures de fonction n'ont pas besoin d'être exactement les mêmes que celles spécifiées (c'est-à-dire que vous pouvez ajouter des arguments supplémentaires si vous le souhaitez). Ce client doit fonctionner avec votre application Flask lorsqu'il est défini avec l'adresse IP et le port corrects.

### Client de jeu

Cette partie est un peu orthogonale au reste des tâches, mais est nécessaire pour interagir avec les matchs NHL «live». Jusqu'à présent, vous avez travaillé avec des vieux jeux des saisons passées. Maintenant, nous voulons mettre nos modèles brillants à l'épreuve sur des jeux en direct! Pour ce faire, nous devons adapter la fonctionnalité que nous avons implémentée pour télécharger les données de jeu afin d'implémenter une manière intelligente d'obtenir des données de jeux en direct. Plus précisément, nous voulons être en mesure d'extraire des données de jeu en direct et de ne traiter que les événements que **nous n'avons pas encore vus**. Il y a plusieurs façons de faire ceci:

1. Obtenez tous les événements pour le même `game_id`, par exemple **2021020329** serait :

<https://statsapi.web.nhl.com/api/v1/game/2021020329/feed/live/>

et obtenez le jeu complet en réponse. Gardez une sorte de tracker interne des événements que vous avez traités jusqu'à présent. Ensuite, traitez les événements que vous n'avez pas vus (par exemple, produisez les caractéristiques requises par le modèle, faites une demande au service de prédiction et stockez les probabilités de but). Mettez à jour le tracker que vous avez implémenté afin que la prochaine fois que vous envoyez un ping au serveur, vous pouvez ignorer les événements que vous avez déjà

traités. Vous devrez probablement utiliser le dernier événement quelque part afin de calculer correctement les caractéristiques de l'événement suivant (car les fonctionnalités avancées impliquent d'inclure les informations sur l'événement précédent).

2. Similaire à ce qui précède, mais utilise le paramètre de requête `diffPatch`. Par exemple :

[https://statsapi.web.nhl.com/api/v1/game/2021020329/feed/live/diffPatch?startTimecode=20211128\\_200600](https://statsapi.web.nhl.com/api/v1/game/2021020329/feed/live/diffPatch?startTimecode=20211128_200600)

Le `startTimecode` est spécifié sous la forme `yyyymmdd_hhmmss`, en heure UTC. Si l'heure spécifiée est à moins de **2 minutes** de l'heure actuelle, vous obtiendrez à la place une liste d'événements à partir de l'heure spécifiée. Sinon, vous obtiendrez simplement la même réponse que si vous aviez interrogé sans le `diffPatch`. Implémentation étrange je sais, mais `\_(ツ)_/`. Vous pouvez (mais n'êtes pas obligé) d'implémenter une logique pour utiliser également les informations `diffPatch` au lieu des événements bruts du jeu en direct. C'est assez amusant et satisfaisant de le faire fonctionner, mais si vous manquez de temps, tenez-vous-en à l'approche 1.

Aucun template n'est fourni pour ce client - vous devez trouver une implémentation qui a le plus de sens pour vous. Vous utiliserez ce client pour la partie 4 lorsque vous intégrerez toutes ces parties ensemble. Notez que vous réutiliserez votre code pour transformer les événements en caractéristiques - tout ce que vous devez vraiment implémenter ici est la logique pour faire une demande pour un jeu en direct et filtrer les événements que vous avez déjà traités auparavant.

### Évaluation

- Votre implémentation fait une demande correctement au jeu spécifique et filtre correctement les événements traités précédemment. Vous obtenez le prochain lot d'inputs à transmettre à votre service de prédiction.

## 3. Docker Partie 1 - Service de Prédiction (15%)

Suivez les [instructions ici](#) pour installer Docker sur votre système. Le **dépôt de projet mis à jour** comprend deux Dockerfiles avec de brèves explications sur les commandes que vous devrez utiliser pour définir votre image Docker. Une chose à noter est qu'il est généralement préférable de s'en tenir à l'utilisation de `pip` au lieu des environnements Conda dans les conteneurs Docker, car cette approche est généralement plus légère que Conda. Un modèle courant consiste à copier le fichier `requirements.txt` dans le conteneur Docker, puis à simplement faire

```
pip install -r requirements.txt
```

à partir du Dockerfile. Vous pouvez également copier un package Python entier (par exemple, le dossier ``ift6758/``) dans le conteneur, et également effectuer un simple

```
pip install -e ift6758/
```

pour installer le package comme vous le feriez dans l'image. Vous trouverez plus de commandes et d'informations utiles dans le fichier README du dépôt de projet mis à jour.

Pour cette section, effectuez les tâches suivantes :

- Mettez l'application Flask que vous avez créée dans la Partie 1 dans un conteneur Docker en remplissant le **Dockerfile.serving**. Cela ne devrait pas être très compliqué; le mien était d'environ 10 lignes de commandes.
  - La clé de l'API Comet doit être transmise au moment de l'exécution via un argument de variable d'environnement - **NE PAS l'intégrer dans votre conteneur**
  - Les conteneurs sont censés être légers - **N'intégrez PAS tous vos modèles dans le conteneur!** De plus, nous pouvons utiliser la fonctionnalité de changement de modèles que nous avons implémentée dans la partie 1.
  - Vous n'avez pas *besoin* de spécifier une commande par défaut pour le conteneur, mais c'est recommandé.
- Remplissez le script de construction (build.sh) avec la commande docker qui construit le conteneur, et remplissez le script d'exécution (run.sh) avec la commande docker qui exécute le conteneur avec les arguments appropriés.
- Reformatez la configuration de docker dans le `docker-compose.yml` fourni, afin que vous puissiez exécuter l'application avec un seul `docker-compose up` au lieu de la commande d'exécution plus longue. Cela peut sembler inutile dans ce contexte, mais docker compose devient très utile lorsque vous commencez à créer des systèmes avec de *nombreux* services docker, qui doivent tous être mis en réseau les uns avec les autres. Cela s'en vient!

## Évaluation

- Cela sera construit et exécuté en exécutant :

```
$ docker-compose up
```

Vous perdrez des points si cela ne fonctionne pas ! Votre service doit également évidemment fonctionner avec des demandes appropriées - ça sera testé dans la dernière section, mais des notes sont également attribuées dans cette section pour l'exactitude de votre service.

- La clé API Comet doit être récupérée à partir de `${COMET_API_KEY}` sur la machine locale et ne doit pas être hardcodée.
- Votre conteneur ne doit pas contenir de poids de modèle supplémentaires ; vous pouvez inclure des poids "par défaut" si vous le souhaitez.
- Vos commandes build et run respectivement dans les `build.sh` et `run.sh` sont correctes (oui, elles n'ont besoin que d'une simple commande).
  - *Le but est de vous exposer à la relation entre le CLI docker et docker-compose.*

#### 4. Application Streamlit (30%)

Nous y sommes presque ! Bien que les clients et les services que vous avez créés soient sympas, ils sont toujours difficiles à interagir avec pour la plupart des gens. La partie amusante est de les transformer en un outil utilisable par toute personne ayant accès à un ordinateur et à Internet. Pour cela, nous utilisons Streamlit.

Streamlit est un framework permettant de créer des applications Web interactives en utilisant uniquement Python! Il fournit une grande quantité de fonctionnalités particulièrement utiles pour afficher des données de différentes formes et fournir des composants interactifs afin que les utilisateurs puissent explorer ces données, le tout sans avoir besoin de connaissances en HTML/CSS/JavaScript. Pour en savoir plus sur les capacités de Streamlit, consultez [leur article de blog](#).

##### Commencer avec Streamlit

Comme vous vous êtes familiarisé avec l'utilisation de Python, l'utilisation de Streamlit devrait vous sembler assez naturelle. Pour commencer, l'exécution de

```
pip install streamlit
```

dans votre environnement virtuel devrait installer tout ce dont vous avez besoin. Ensuite, en suivant les premières étapes mentionnées [ici](#), vous devriez être opérationnel avec votre première application. N'hésitez pas à jouer avec! Bien que nous n'utilisons rien de très compliqué pour ce projet, c'est une bonne idée de jeter un coup d'œil à :

- [Les principaux concepts de Streamlit](#) (en particulier pour comprendre comment Streamlit gère l'interactivité, en réexécutant le script à chaque fois que certains widgets d'entrée changent).
- [La référence de l'API](#) (pour donner une idée de toutes les composantes pré-construits à votre disposition depuis Streamlit!)

## Fonctionnalité à Implémenter

Plus précisément, nous utiliserons votre client de jeu en direct et votre client fournisseur que vous avez implémentés dans la partie 2 pour interagir avec votre service de prédiction conteneurisée que vous avez construit dans la partie 3. Le but est de créer un tableau de bord simple mais en direct pour votre modèle de expected goals (xG).

Vous allez créer une application Web interactive Streamlit qui peut accepter un ID de jeu comme entrée, puis un bouton qui déclenche la méthode "ping game" du client de jeu (ou tout autre méthode que vous avez implémenté). Les nouveaux événements de tir doivent être envoyés à votre service de prédiction, et les probabilités de buts qui sortent de votre modèle doivent être jointes en tant que nouvelle colonne aux nouveaux événements de tir. Vous afficherez ensuite la somme des buts attendus pour chaque équipe, en tant que proxy pour "quelle équipe fait mieux" et la comparerez cette valeur au score réel. Il s'agit d'une estimation simple de l'équipe qui joue le mieux, mais bon, c'est un début !

Pour envoyer un ping à votre service maintenant qu'il réside dans un conteneur Docker, vous devriez pouvoir utiliser l'IP **127.0.0.1** au lieu de **0.0.0.0** comme vous l'avez fait dans la partie 1. Encore une fois, assurez-vous que le port est correct ! Par exemple :

```
X, idx, ... = game_client.ping_game(game_id, idx, ...)
r = requests.post(
    "http://127.0.0.1:<PORT>/predict",
    json=json.loads(X.to_json())
)
print(r.json())
```

Pour votre application interactive, vous devez implémenter en charge les fonctionnalités suivantes :

- **(Saisie de texte)** Espace de travail, Modèle, Version; pour spécifier le modèle à télécharger à partir du registre de modèles CometML. Il peut s'agir de 3 entrées distinctes.
- **(Clic sur le bouton)** Téléchargez le modèle; télécharge le modèle à partir de CometML en utilisant les paramètres spécifiés ci-dessus et l'échange dans le service de prédiction.
- **(Saisie de texte)** Identifiant de jeu; spécifie le jeu auquel votre outil envoie une requête, par exemple **2021020329**
- **(Clic sur le bouton)** Requête jeu
  - Lorsque vous appuyez sur le bouton, votre tableau de bord doit afficher :
    - Les noms des deux équipes
    - La période
    - Le temps restant dans la période
    - Le score actuel

- La somme des buts attendus (xG) pour le jeu entier jusqu'à présent pour les deux équipes (obtenu à partir du client du jeu)
- Différence entre le score actuel et la somme des buts attendus (c'est-à-dire si l'équipe à domicile a marqué 2 buts et que la somme des buts attendus est de 2,4, montrez une différence de  $2,4 - 2 = 0,4$ ). Pour cela, [utilisez les éléments suivants](#).
  - Certains de ces valeurs peuvent être obtenus en faisant des requêtes API directes à l'API NHL, d'autres utiliseront le client de jeu que vous avez créé précédemment
- De plus, vous devez le **dataframe de données de toutes vos caractéristiques**, avec la **probabilité assignée par le modèle à chaque événement**. Assurez-vous que votre logique pour filtrer les jeux est correcte - il ne devrait pas y avoir de copies, et vous ne pouvez pas simplement recalculer les prédictions d'objectifs pour l'ensemble croissant d'événements de tir !

Une fois que tout est terminé, votre application Web devrait ressembler à ceci (avec les noms réels des fonctionnalités/événements) !

×

Workspace

Workspace x

Model

Model y

Version

Version z

Get model

## Hockey Visualization App

Game ID

2021020329

Ping game

### Game 2021020329: Canucks vs Avalanche

Period 1 - 12:35 left

Canucks xG (actual)

3.2 (3)

↓ -0.2

Avalanche xG (actual)

1.4 (2)

↑ 0.4

### Data used for predictions (and predictions)

	feature 0	feature 1	feature 2	feature 3	feature 4	feature 5	feature 6	Model output
Event 0	0.6321	0.2581	0.5314	0.2530	0.2043	0.3173	0.0793	0.4831
Event 1	0.3004	0.6790	0.5043	0.3568	0.0346	0.0457	0.6551	0.7898
Event 2	0.5987	0.0501	0.7842	0.1231	0.2067	0.7573	0.1430	0.8154
Event 3	0.5291	0.7031	0.8345	0.4776	0.6375	0.0044	0.3401	0.8146
Event 4	0.1625	0.6301	0.0157	0.4930	0.8829	0.4434	0.0124	0.0917
Event 5	0.0663	0.3240	0.6063	0.7751	0.3467	0.0621	0.4689	0.2889

## Évaluation

- La fonctionnalité ci-dessus est correctement supportée par votre application interactive, c'est-à-dire pas d'événements de tir en double, l'échange de modèles dans votre service de prédiction se fait bien, etc.
- Le comportement que nous recherchons est que lorsque nous cliquons sur le bouton "ping game", s'il y a des nouveaux événements de tirs, ces nouveaux tirs seront filtrés (afin qu'aucun ancien tir ne soit réévaluée), puis envoyées au service de prédiction. Les probabilités de buts générées seront ensuite jointes au DataFrame qui a été soumis et affichés en tant que output. La somme des buts attendus pour chaque équipe est ensuite mise à jour dans le tableau ci-dessus, en plus des champs PÉRIODE et TEMPS RESTANT en haut.
- Vous ne devriez **pas** ignorer la logique de filtrage et simplement recalculer toutes les probabilités de tir à chaque fois que vous faites une requête de jeu.
- Cette visualisation sera exécutée dans la section suivante (depuis l'environnement Docker).

## 5. Docker part 2 - Streamlit (15%)

C'est la dernière partie, promis! Vous devez créer un autre conteneur docker (dans **Dockerfile.streamlit**). Ce sera essentiellement une copie de **Dockerfile.serving**, mais au lieu de copier l'application de prédiction, vous copierez votre application Streamlit que vous avez créée dans la partie 4. Vous remplacerez ensuite la commande utilisée pour exécuter l'application modèle par la commande suivante pour exécuter Streamlit :

```
streamlit run {STREAMLIT_FILE} --server.port {STREAMLIT_PORT}
--server.address {STREAMLIT_IP}
```

Une fois ce conteneur docker créé, vous devez immédiatement le configurer dans votre fichier docker-compose comme un autre service ; vous pouvez décommenter le reste du modèle docker-compose.yaml fourni pour remplir ce dont vous avez besoin. En effet, la mise en réseau peut devenir assez délicate dans Docker pour les non-initiés, et la composition de docker peut vous faciliter la vie. Plus précisément, docker compose effectue par défaut de belles résolutions de noms. Vous remarquerez que le format de votre fichier docker-compose suit les lignes suivantes :

```
# docker-compose.yaml
services:
  service1:
    ...
  service2:
    ...
```



Pour que cela fonctionne, vous devez modifier l'url auquel vous faites une demande depuis votre application Streamlit pour accéder au service de modèle. Plus précisément, supposons que votre application Streamlit réside dans **service2** et que vous souhaitiez envoyer une requête HTTP à **service1**. En fait, vous n'avez pas besoin de rechercher l'adresse IP du conteneur de **service1** - vous pouvez simplement envoyer une requête HTTP à `http://service1:PORT/endpoint`. La résolution du nom est prise en charge si vous utilisez docker-compose (mais vous devez utiliser le bon port). Cette fois, vous n'avez pas à vous soucier d'écrire les commandes build/run. Lorsque vous exécutez `docker-compose up`, vous devriez voir quelque chose de similaire à ce qui suit :

```
docker-project-template-serving-1 | [2022-11-29 02:29:17 +0000] [7] [INFO] Starting gunicorn 20.1.0
docker-project-template-serving-1 | [2022-11-29 02:29:17 +0000] [7] [INFO] Listening at: http://0.0.0.0:8890 (7)
docker-project-template-serving-1 | [2022-11-29 02:29:17 +0000] [7] [INFO] Using worker: sync
docker-project-template-serving-1 | [2022-11-29 02:29:17 +0000] [10] [INFO] Booting worker with pid: 10
docker-project-template-streamlit-1 | 2022-11-29 02:29:17.962 INFO matplotlib.font_manager: generated new fontManager
docker-project-template-streamlit-1 |
docker-project-template-streamlit-1 | Collecting usage statistics. To deactivate, set browser.gatherUsageStats to False.
docker-project-template-streamlit-1 |
docker-project-template-streamlit-1 | You can now view your Streamlit app in your browser.
docker-project-template-streamlit-1 |
docker-project-template-streamlit-1 | URL: http://0.0.0.0:8892
docker-project-template-streamlit-1 |
```

Si tout est correctement connecté, vous devriez pouvoir accéder au deuxième lien et accéder à votre application Streamlit. Si vous envoyez un ping à un jeu, il devrait pouvoir accéder au service de prédiction (sinon, assurez-vous d'avoir changé l'adresse IP pour le nom du service de prédiction et que vous utilisez le bon port).

## Évaluation

- Nous évaluerons votre visualisation en exécutant

```
$ docker-compose up
```

Cela doit **créer et commencer tous les conteneurs requis** et héberger l'application avec votre visualisation interactive créée dans la partie 4. Docker par conception facilite les environnements reproductibles, il n'y a donc pas raison pour que cela ne fonctionne pas sur ma machine et **vous perdrez des points si cela ne fonctionne pas!**

- Nous devons être en mesure d'accéder à votre application Streamlit et d'interagir avec elle (c'est-à-dire que nous devrions être en mesure de télécharger des modèles et d'envoyer un ping au service de prédiction pour obtenir des informations sur le jeu, la connexion entre les deux services devrait fonctionner immédiatement).

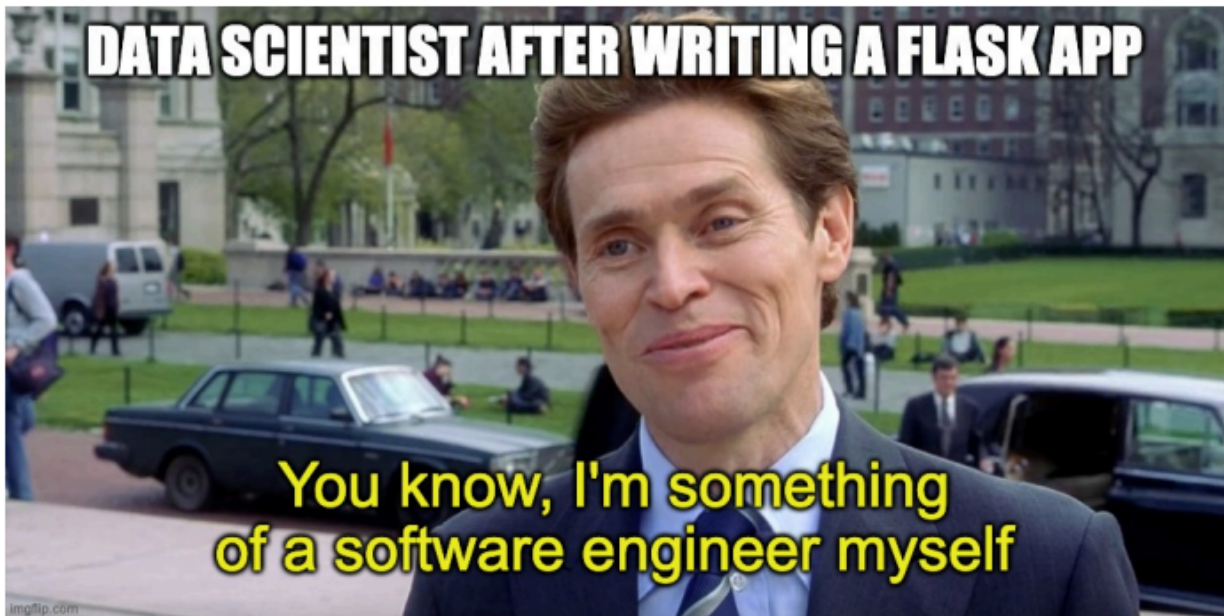
## 6. BONUS : fonctionnalité supplémentaire de Streamlit (facultatif, 5 %)

Si vous jetez un coup d'œil à la référence de l'API de Streamlit, vous verrez qu'il existe une myriade de composants pouvant être inclus dans l'application Web, ainsi que de nombreuses façons d'intégrer de l'interactivité à l'application Web; les exigences que nous avons publiées ne font qu'effleurer la surface.

Inspirez-vous de tout ce que vous avez appris/construit en classe jusqu'à présent et ajoutez des fonctionnalités/fonctionnalités/données/visualisations supplémentaires dans votre application Streamlit qui fourniraient des informations intéressantes aux téléspectateurs de hockey qui utilisent votre application. Il peut s'agir de graphiques que vous avez dû créer pour les milestones précédents, de statistiques globales d'équipe, de graphiques de tir, de comparaisons de résultats de modèles, etc. Soyez créatif !

Si vous visez le bonus, ajoutez un court paragraphe quelque part dans votre application Web en utilisant `st.write` décrivant la fonctionnalité que vous avez ajoutée et comment vous l'avez incluse. Cette section sera jugée en fonction de la créativité, de l'utilité et du défi technique.

Et voila!



Merci d'avoir traversé ce voyage de projet ! Je sais que c'était beaucoup de travail, mais j'espère que vous l'avez trouvé amusant, intéressant et enrichissant. J'espère que vous avez tous appris quelque chose qui vous aidera dans la prochaine étape de votre carrière, qu'il s'agisse de chercher un emploi dans l'industrie ou de faire de la recherche. Bonne chance avec tout ce que votre avenir vous réserve!

## Évaluations de groupe

En plus de la notation décrite ci-dessus, pour chaque étape, il vous sera demandé de noter dans quelle mesure vous pensez que chacun a contribué à cette étape et de fournir des commentaires constructifs à vos coéquipiers, en fonction de leurs forces et de leurs domaines d'amélioration potentiels. Les étudiants qui ne contribuent pas beaucoup pourraient obtenir une mauvaise note et, dans des cas extrêmes, pourraient être retirés du groupe et invités à réaliser les projets individuellement.

Pour une équipe de taille **N**, chaque membre de l'équipe aura **N x 20** points à répartir entre tous les membres de votre groupe (vous compris). Vous attribuerez ensuite à chacun une note comprise entre 10 et 40, où 10 correspond à "demi-effort" et 40 correspond à "double effort". Dans une situation idéale, tout le monde contribuera au projet de manière égale et ainsi chacun attribuera 20 points à chaque coéquipier. Cependant, dans le cas où certaines personnes ont moins contribué que d'autres, vous pouvez leur attribuer moins de points et donner ces points à ceux qui, selon vous, ont plus contribué. Les cas extrêmes entraîneront un suivi par un instructeur auprès de l'équipe pour résoudre toute difficulté potentielle. Cela peut inclure une vérification de l'historique git. Toute tentative de déjouer le système sera gérée manuellement et défavorablement!

En plus du score, vous donnerez également de **brefs commentaires à vos pairs**, à la fois sur les points forts et sur les domaines d'amélioration potentiels. Cette rétroaction sera donnée de manière privée et anonyme à chaque étudiant. Vous pouvez donner votre avis sur plusieurs axes différents, tels que leur **travail d'équipe** (communication, fiabilité) et leur **travail technique** (leur contribution et la qualité de leur travail/code). Si vous donnez une note inférieure à 20, vous devez donner des commentaires constructifs sur les domaines qu'ils peuvent améliorer.

### Comment les scores impactent votre note

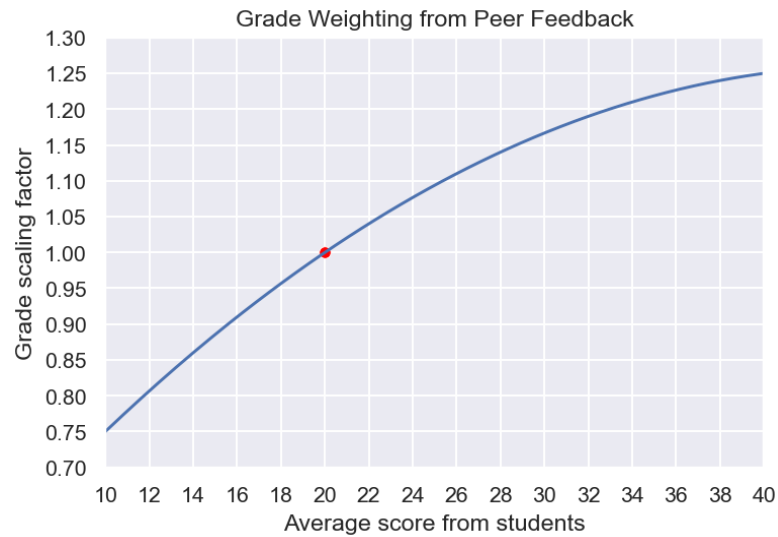
- **x** est la note moyenne d'un élève (hors sa propre note), qui se trouve entre 10 ("demi-effort") et 40 ("double effort").
- Le score d'un étudiant pour le milestone sera multiplié par le facteur de pondération suivant:

$$\text{Scaling}(x) = 0.41667 + 0.0375x - 0.00041667x^2$$

- Ce système a été adopté par [Brian Fraser \(SFU\)](#), qui à son tour est basé sur les travaux de [Bill Gardner](#).

---

<sup>2</sup> The coefficients are obtained by fitting the quadratic  $y(x) = a \cdot x^2 + bx + c$  to the points  $x=10 \rightarrow \text{weight}=0.75$ ;  $x=20 \rightarrow \text{weight}=1.0$ ;  $x=40 \rightarrow \text{weight}=1.25$ .



## Références utiles

- [IFT 6758 Hockey Primer](#)
- [Documentation non officielle de l'API NHL](#)
- [Comet.ml Python SDK](#)
  - `api.download_registry_model(...)`
- [Flask Tutorial](#)
- [Full Stack Deep Learning \(Conférence 11\)](#)
- Docker
  - [Dockerfile bonnes pratiques](#)
- [Streamlit Guide de démarrage](#)
- [Streamlit API Reference](#)