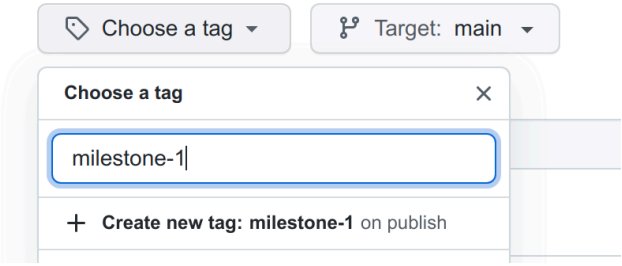
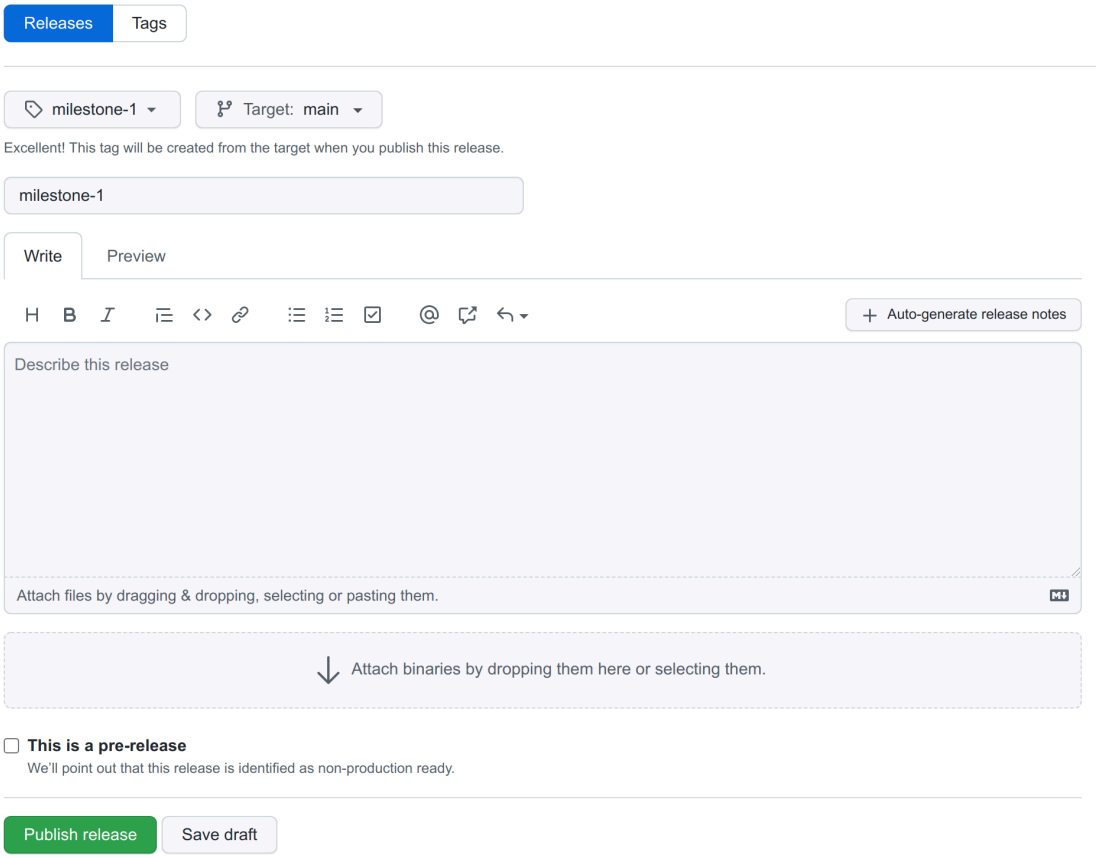
IFT 6758 Project: Milestone 3

Released: Nov 14, 2025
Due date: Dec 8, 2025 12:00 p.m.

Important: Before beginning work on milestone 3, publish a release for milestone 2 by following the instructions here. You do not need to add anything in the description, or upload any binaries. Use `milestone-3` as both the tag and release title, eg:





So far, you've played with a significant portion of the data science workflow, including data wrangling, EDA, creating visualizations, feature engineering, and statistical modelling. So, we've got our models, we're done now right? Well, if you want nobody to use all of the work that you've done, yes! Otherwise, we need to have some understanding of what happens once this initial phase of the data science project is complete.

The focus for this last milestone will be data science in "production". This is in quotes because it is important to have a **disclaimer**: **this is by no means a practical guide for how to actually build robust ML systems for a business**. Instead, the purpose of this milestone is to give you some context and flavour of what types of tools and ideas might be used to deploy models into useful services for a business, albeit on a much smaller scale. Furthermore, this only focuses on the notion of **model deployment**; but does not touch on other important concepts such as **monitoring**.

Using all of the models that you have already created (and saved into WandB!), you will build a very simple model service app that will be deployed into a Docker container and accessible through a REST API. You will also build a "Live game client", which will retrieve shot events from live (or historical) NHL games, preprocess the data into features compatible with your model, and then query your model service for expected goal predictions for each shot event.

The live game client will then be packaged into a nice interactive dashboard using Streamlit.

A note on Plagiarism

Using code/templates from online resources is acceptable and it is common in data science, but be clear to cite exactly where you took code from when necessary. A simple one line snippet which covers some simple syntax from a StackOverflow post or package documentation probably doesn't warrant citation, but copying a function which does a bunch of logic that creates your figure does. We trust that you can use your best judgement in these cases, but if you have any doubts you can always just cite something to be safe. We will run some plagiarism detection on your code and deliverables, and it is better to be safe than sorry by citing your references.

Integrity is an important expectation of this course project and any suspected cases will be pursued in accordance with Université de Montréal's very strict policy. The full text of the university-wide regulations can be found here. It is the responsibility of the team to ensure that

this is followed rigorously and action can be taken on individuals or the entire team depending on the case.

Learning Objectives

- **Model "Deployment"**
  - Model as a service
  - Model dependency management (Docker)
  - REST APIs
- **"Production"/Live Workflows**
  - Writing
- **Interactive visualization**
  - Making your predictions easily accessible using Streamlit

Model Deployment (Model as a Service)

(This primer borrows heavily from Full Stack Deep Learning - Lecture 11)

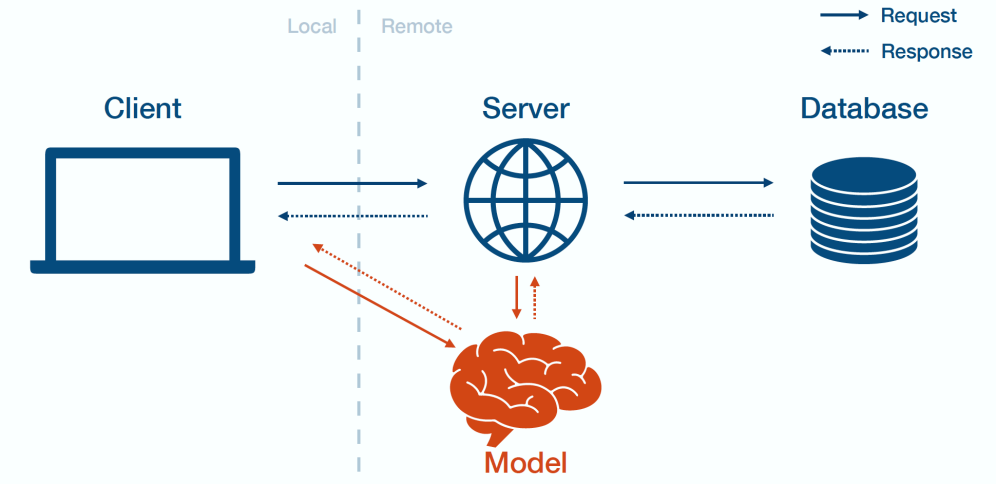


Figure 1: Model as a Service (source)

There are multiple ways to use statistical/machine learning models in production, but we will be focusing on the idea of *Model as a Service*, where a model is deployed as its own service; users and clients interact with the model by making requests specifically to the model service, and receiving the responses.

Services

Before we move on, what is a service you may ask? This can go pretty deep, but the only thing you really need to understand for this project is that it's an architectural style in software engineering which builds reusable and interoperable software components, as opposed to a single large monolithic piece of software. Basically this means that you will have many small,

self-contained pieces (services) that are “black boxes” for its consumers (i.e. the consumer doesn’t need to know how the service works). Services will typically (but not always) operate in their own environment with dedicated hardware, and they may also be ephemeral (i.e. dynamically stood up and torn down as usage fluctuates).

Communication

There are many ways these services can interact with each other. A REST API is a way to communicate through HTTP requests; it is very common and handles small data (such as text or JSON) well, but it doesn’t work as well for larger data (e.g. binaries, images, etc). Other alternatives exist such as gRPC, Kafka, ZeroMQ, etc., but this is more in the domain of software engineers and is not the focus of this course. For our purposes, a REST API will be sufficient but know that this is not the right tool for every task and there is **no standard solution for formatting data that goes into an ML model**. We will be using Flask to build our simple prediction web application.

Dependency Management

To run a model, you require the code, code dependencies, and model weights, all of which need to be available to your service. Model weights are fairly portable, but code and code dependencies can be quite a lot trickier to manage. A common solution to this problem is **containerization**; a **container** is a single fully packaged unit of software which ideally should provide a consistent environment anywhere[1] you may deploy it. Note that containers are not the same as virtual machines (VMs) - VMs require the hypervisor to virtualize physical hardware and typically runs multiple operating systems simultaneously. Containers on the other hand share the same host kernel, but run in isolated user-space environments. If that stuff about VMs didn’t make much sense to you, that’s okay - the main takeaway is that this makes them faster and more lightweight than VMs.

[1] There’s some subtleties when it comes to building containers on different CPU architectures (e.g. x86 (most Intel and AMD), and ARM (originally more common in embedded devices, but now Apple has adopted ARM architecture for their entire PC line). There are ways to provide cross-compatibility between the two architectures, but naively they are incompatible.

The most popular platform for containerization is **Docker**, which is what we will be using for this milestone. There are a few basic concepts that you should be familiar with:

1. **Dockerfile** defines how to build an image (you will write one)
2. **Image** is a built packaged environment (you will create one)
3. **Container** is where images are run inside (you will run one)
4. **Repository** hosts different versions of an image
5. **Registry** is a set of repositories

These will be touched on more in the actual milestone task definitions.

Deliverables

You must submit:

1. Your team's codebase **that is reproducible**
2. Add the relevant IFT 6758 accounts to your github repo and WandB workspace (if you are having trouble with this, we will deal with it on a case-by-case basis).

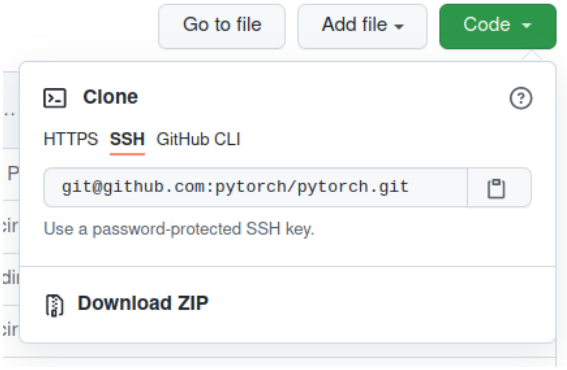
There will be **no blogpost component** for this final milestone as you will be primarily evaluated on your code submission.

Submission Details

To submit your project, you must:

- ☐ Publish your final milestone submission to the `milestone-3` branch
  - ☐ Create a release similar to what you did for milestone 1 and 2, but with the `milestone-3` tag.
- ☐ Submit a ZIP of your codebase to gradescope
- ☐ Add the IFT 6758 TA Github account (`@ift6758-25`) to your git repo as a *viewer* or *contributor*, if you have not done so already or are using a new repo.
- ☐ Ensure `ift6758.2025@gmail.com` is added to your WandB workspace; this should be completed from Milestone 2

To submit a ZIP of your repository, you can download it via the Github UI:



Tasks and Questions

The tasks required for milestone 3 are outlined here. This time, you will not need to submit a blog post; you will be evaluated on your code submissions. In terms of evaluations, you will be marked in the style of code reviews for a portion of your work, and we **will be running your code** in the containerized system you will produce by the final task.

The updated GitHub project repository can be found here.

1. Update API client (10 %)

Unfortunately, the NHL has made a change to their API endpoints. The new play-by-play API can be found below:

Old API:

https://statsapi.web.nhl.com/api/v1/game/{GAME_ID}/feed/live/

New API:

https://api-web.nhle.com/v1/gamecenter/{GAME_ID}/play-by-play

E.g. for the game 2022030411 can be found here:

https://api-web.nhle.com/v1/gamecenter/2022030411/play-by-play

The API has all of the old information, but with some new fields as well! New event types are also available, such as `"missed shots"`. Specifically, the `"plays"` field will have all the event-related information you'll need to complete the milestone!

While the change is unfortunate, issues like this are common-place and it is a good exercise for dealing with the common day-to-day challenges of working in data science. You will need to rewrite the parts of your code that query the API to accommodate this change (this only needs to be done for the **Game Client in Section 2)**. To limit the complexity, we will only ask you to recover **basic features** from the data (i.e. no advanced features/models are needed).

Specifically you must recover:

- **Distance from net**
- **Angle from net**
- **Is goal** (0 or 1)
- **Empty Net** (0 or 1; you can assume NaNs are 0)
    - For this, you can use the `situationCode` which works as follows:
        - First digit: 1 for goalie on ice for away team
        - Second digit: Number of on ice skaters for away team
        - Third digit: Number of on ice skaters for home team
        - Fourth digit: 1 for goalie on ice for home team
    - Combine this with `eventOwnerTeamId` to see if it was an empty net goal.

2. Model Serving Flask app (25%)

We'll first create a simple Flask app to serve our expected goal models that we created in Milestone 2. We won't yet worry about embedding this application into a Docker container - right now we're just focusing on running this app on our local environment. Your first task is to create a Flask app that implements the following endpoints:

- **/predict** - predict the shot's probability of being a goal given the inputs
    - **Input**: the input features to your model, compatible with model.predict() or model.predict_proba()
        - *Hint: df.to_json()*
    - **Output**: the predictions for all of the features; the output

- **/logs** - retrieve logs written from your app

- **/download_registry_model** - basically wraps a similar function in the WandB API to download a model and update the currently loaded model. You should:
    - Check to see if the model you are querying for is already downloaded
        - if yes, load that model and write to the log about the model change.
        - if no, try downloading the model:
            - if it succeeds, load that model and write to the log about the model change.
            - If it fails, write to the log about the failure and keep the currently loaded model

Note: because of the simplification we made to the data client (basic features only), we will only be testing the logistic regression models on distance and distance+angle models! Don't worry about the rest of the models you may have trained.

In the new project repo, you will find an **app.py** template in the `serving` directory. This has the skeleton and boilerplate for what you need to implement. You **should not** put any task specific functionality into your Flask app; any data preprocessing (such creating all of the features

needed for your model) should be done prior to making the request to the serving app. Your app should behave similarly to a SK-Learn model object.

To run the app, if you are in the same directory as **app.py**, you can run the app from the command line using gunicorn (Unix) or 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 can be installed via:

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

At this point, you can test the app with existing training/validation data that you used in Milestone 2, and ping the app directly using the Python requests library, eg:

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

Furthermore, once you implement the **/logs** endpoint, you will be able to view all of the logs by navigating to http://0.0.0.0:<PORT>/logs in your browser.

Evaluation:
Code will not be run at this stage (it will be tested in Part 4). However points will be assigned to this section based on:

- All three endpoints (**/predict**, **/logs**, and **/download_registry_model**) are implemented correctly
- The service can download models stored in your WandB model registry, and swap them out on the fly *without restarting the app*. Cases where the new model does not download successfully (e.g. doesn't exist) are handled gracefully.
- You log appropriate and informative debugging messages to the logger; you should be able to debug your app from the logs without manually going and poking in the application (and later on, in the Docker container)

You only need to focus on the models you produced in Milestone 2 part 2, with the core set of basic features. It's fine if this is missing a bunch of your best models, the point is just build up some sort of simple system that demonstrates the core functionality.

3. Clients (10%)

We now want to have a cleaner interface with the Flask app that we just created, as well as a nice interface to retrieve live NHL game data. For this, you will need to create 2 clients: a **serving client** and a **game client**. You will be provided with a template file for the serving app client (in `ift6758/ift6758/client/serving_client.py`). For the game client, you have already written the bulk of the code to extract features - you will need to come up with the implementation that makes the most sense given the specified requirements described below.

Serving Client

Now that we have our SKLearn model served through a Flask app, we want a way to interact with it without having to explicitly call `requests.post` every time we want to submit a request. Implement the three methods defined in `ift6758/ift6758/client/serving_client.py` These functions will be very simple and straightforward to implement.

Evaluation

- The three methods (**predict(...)**, **logs()**, and **download_registry_models(...)**) are correctly implemented. The function signatures do not need to be the exact same as what is specified (i.e. you can add additional arguments if you'd like). This client must work with your Flask app when set with the correct IP and port.

Game Client

This part is a little orthogonal to the rest of the tasks, but is necessary to interface with "live" NHL games. So far, you have queried old games from bygone seasons of yore. Now, we want to put our shiny models to the test on the latest games that are happening right NOW! To do this, we need to adapt the functionality that we implemented to download game data to support a more intelligent way of pinging live games. Specifically, we want to be able to pull live game data and only process events that we haven't yet seen.

1. Get the game data for a given game_id. The goal is to only process events that you have not yet seen; if you can do this via the API you're welcome to. Otherwise, you could keep some sort of internal tracker of what events you have processed so far and then process the events that you haven't seen (i.e. produce the features required by the model, query the prediction service, and store the goal probabilities). Update the tracker you implemented so that the next time you ping the server, you will be able to ignore the events that you've already processed.

No template is provided for this client - you must come up with what you think makes the most sense. You will be using this client for Part 4 when you integrate all of these parts together. Note that you will be reusing your code for processing the events into features - all you really need to implement here is the logic to query a live game, and filter out events that you've already processed before.

Evaluation

- Your implementation correctly queries the specific game, and filters the previously processed events correctly. You get the next batch of inputs to pass to your prediction service.

4. Docker part 1 - Serving (15%)

Follow the instructions here to install Docker on your system. Included in **the updated project repo** are two Dockerfiles with brief explanations of the commands you will need to use to define your Docker image. One thing to note is you should **stick to using pip instead of Conda** environments in Docker containers, as this approach is generally more lightweight than Conda. A common pattern is to copy the requirements.txt file into the docker container and then simply do

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

from the Dockerfile. You can also copy an entire Python package (e.g. the `ift6758/` folder) into the container, and also do a simple

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

to install the package as you would in the image. You'll find more useful commands and information in the README of the updated project repo.

For this section, complete the following tasks:

- Embed the Flask app you built in Part 1 into a Docker container by filling in the provided **Dockerfile.serving**. This should not be very complicated; mine was ~10 lines of actual commands.
    - The Comet API key should be passed at RUN time via an environment variable argument - **do NOT embed into your container**
    - Containers are meant to be lightweight - **do NOT embed all of your models into the container!** Plus we can leverage the "hot swapping" functionality we implemented in Part 1.
    - You don't *need* to specify a default command for the container, but it's definitely nice to have.

- Fill in the build script (build.sh) with the docker command which builds the container, and fill in the run script (run.sh) with the docker command which runs the container with appropriate arguments.

- Reformat the docker configuration in the `docker-compose.yaml` file provided, so you can run the app with a single `docker-compose up` instead of the longer run command. This may seem pointless in this context, but docker compose becomes very

useful when you start to create systems with *many* docker services, all of which need to network with each other. This is coming up!

Evaluation

- This will be built and run by running:

  ```
  $ docker-compose up
  ```

  You will lose marks if this does not work! Your service should also obviously work with appropriate requests - this is tested in the final section, but marks are also assigned in this section for correctness.

- The WandB API key must be retrieved from the `${WANDB_API_KEY}` environment variable on the local machine, **not hardcoded anywhere**.

- Your container must not contain extra model weights; you can include one "default" if you'd like.

- Your build and run commands in the `build.sh` and `run.sh` files respectively are correct (yes, they only need one simple command).
  - *The purpose of this is to expose you to the relationship between the docker CLI and docker-compose.*

5. Streamlit App (25%)

We're almost there! While the clients and services you've created are cool, they are still hard to interact with for most people. The fun part is to turn them into a tool that can be used by any person with access to a computer and the internet. For that we use Streamlit.

Streamlit is a framework for building interactive web apps using just Python! It provides a large amount of functionality that is particularly useful for displaying data of different forms and providing interactive components so that users can explore this data, all without needing any HTML/CSS/JavaScript knowledge. For more on Streamlit's capabilities, see their blog post.

Getting Started with Streamlit

As you've become comfortable with using Python, using Streamlit should feel quite natural. To get started, running

```
pip install streamlit
```

inside your virtual environment should install everything you need. Then, once that's done, following the first steps mentioned here should have you up and running with your first app. Feel

free to play around with it! While we won't be using anything too fancy for this project, it's a good idea to take a look at:

- The main Streamlit concepts (especially understanding how Streamlit deals with interactivity, by rerunning the script whenever some input widget changes).
- The API reference (to give an idea of all the pre-built components available to you from Streamlit!).

Functionality to Implement

Specifically, we will use your Live Game Client and Serving Client that you implemented in Part 2 to interface with your containerized serving app that you implemented in Part 3 to create a simple, but live dashboard for your expected goals model. You will be creating a Streamlit interactive web app that can accept a game ID as an input and then a button which triggers the Game Client's "ping game" method (or whatever you implemented). New shot events should be sent to your prediction service, and the goal probabilities that your model outputs should be joined as a new column to the new shot events. You will then display the sum of expected goals for each team, as a proxy for "which team is doing better" and compare it to the actual score. This is a very crude estimate of which team is doing better, but hey it's a start!

To ping your service now that it lives in a docker container, you should be able to use the IP **127.0.0.1** instead of **0.0.0.0** as you did in part 1. Once again, make sure that the port is correct! For example:

```
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())
```

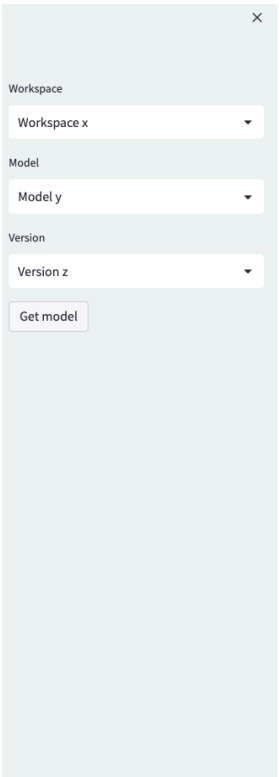
For your interactive app, you must support the following functionality:

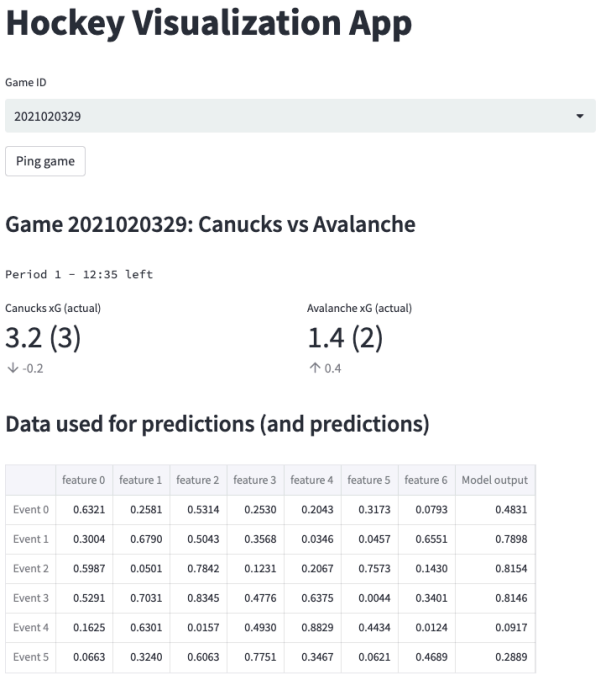
- **(Text input)** Workspace, Model, Version; for specifying what model to download from the CometML model registry. This can be 3 separate inputs.
- **(On button click)** Download model; downloads the model from CometML using the parameters specified above and swaps it in the serving app.

- **(Text input)** Game ID; specifies what game your tool will ping, eg **2021020329**
- (On button click) Ping game
    - When the button is pressed, your dashboard should show the:
        - Home and away team names
        - Period
        - Time left in the period

        - Current score
        - Sum of expected goals (xG) for the entire game so far for both teams (obtained from the game client)
        - Difference between current score and sum of expected goals (i.e. if the home team has scored 2 goals and the sum of expected goals is 2.4, show a difference of 2.4-2=0.4). For this, use the following.
    - Some of these can be obtained by doing direct API requests to the NHL API, others will make use of the game client you created earlier
- In addition, you must display the **dataframe of all of your features**, with the **model output probability for each event.** Make sure that your logic to filter out games is correct - there should be no duplicates, and you can't simply recalculate the goal predictions for the entire growing set of shot events!

Once everything is done, your web app should look something like this (with the actual feature/event names)!





	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

Evaluation

- The above functionality is correctly supported in your interactive app, i.e. no duplicate shot events, models swap in your prediction service, etc.

- The behaviour we're looking for is that when we click on the "ping game" button, if there are new shot events, these new shots will be filtered (so that no old shots are re-evaluated), and then sent to the model prediction service. The outputted goal probabilities will then be joined to the feature dataframe that was submitted, and displayed as the output. The sum of expected goals over each team is then updated in the above table, in addition to the PERIOD and TIME LEFT fields at the top.

- You should **not** be ignoring the filtering logic and simply recomputing all of the shot probabilities every time you ping the game.

- This visualization will be run in the next section (from the Docker environment).

6. Docker part 2 - Streamlit (15%)

This is the final part, I promise! You need to create one more docker container (in **Dockerfile.streamlit**). This will basically be a copy of **Dockerfile.serving**, but instead of copying in the serving app, you will copy in your Streamlit app that you created in part 4. You will then replace the command used to run the model app with the following command to run Streamlit:

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

Once this docker container is built, you should immediately set it up in your docker compose file as another service; you can uncomment the rest of the docker-compose.yaml template provided to fill in what you need. This is because networking can become sort of tricky in Docker for the uninitiated, and docker compose can make your life quite a lot easier. Specifically, docker compose does some nice name resolution stuff for you by default. You'll notice the format of your docker compose file is along the lines of:

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

For this to work, you'll need to change the url you're querying from your Streamlit App to access the model service. Specifically, say your Streamlit app lives in **service2**, and you want to make an HTTP request to **service1**. You actually don't need to look for the container IP of **service1** - you can simply make an HTTP request to `http://service1:PORT/endpoint`. The name resolution is taken care of if you're using docker compose (but you do need to keep track of the port). This time, you don't need to worry about writing down the build/run commands. When you run `docker-compose up`, you should see something similar to the following:

```
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  |
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  |
```

If everything is hooked up correctly, you should be able to go to the second link and access your Streamlit app. If you ping a game, it should be able to access the serving service (if not, make sure you changed the IP to the name of the serving service and that you are using the correct port.

Evaluation

- We will be evaluating your visualization by running

  ```
  $ docker-compose up
  ```

  This must **build and spin up all of the required containers**, and host the app with your interactive visualization created in Part 4. Docker by design facilitates reproducible environments, so there is no reason for this to not run on my machine and **you will lose marks if this does not work!**

- We must be able to access and interact with your Streamlit app (i.e. we should be able to download models and ping the serving service to get game info, the connection between the two services should work out of the box).

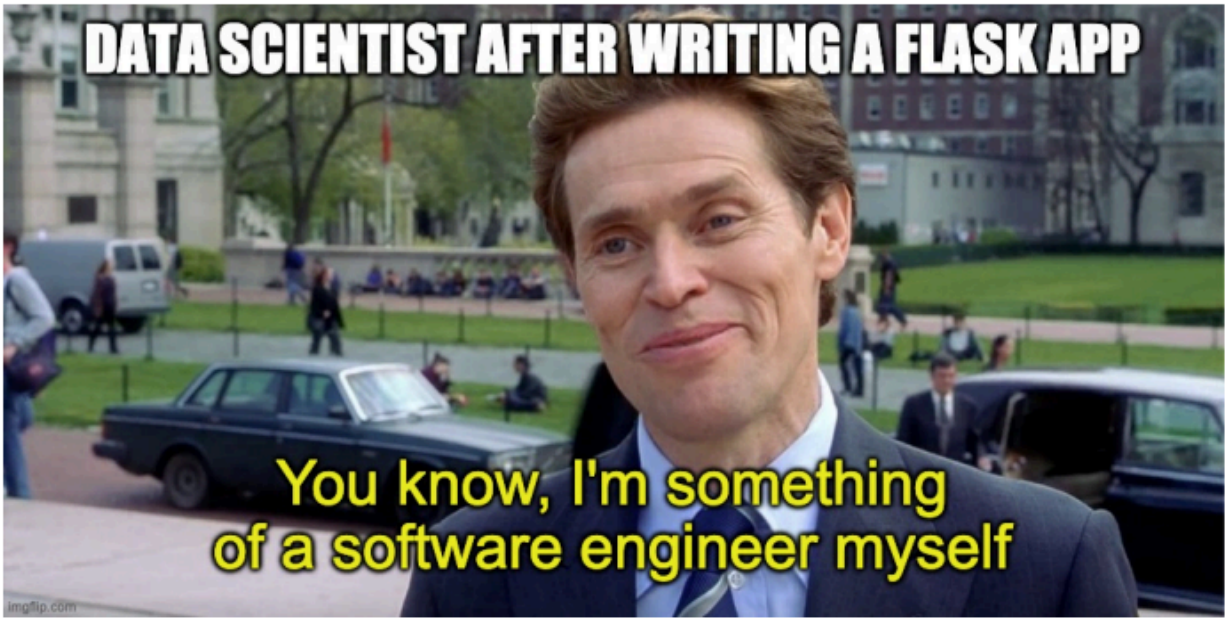
7. BONUS: Streamlit Additional Functionality (optional, 5%)

If you take a look at Streamlit's API reference, you'll see there are a myriad of components that can be included in the web app as well as many ways to build in interactivity to the web app; the requirements we posted are just scratching the surface.

Drawing inspiration from everything you've learned/built in class thus far, add additional functionality/features/data/visualizations into your Streamlit app that would provide interesting information to hockey viewers that happen upon your app. This could be graphs you've had to make for previous milestones, aggregate team statistics, shot graphs, model output comparisons, etc. Be creative!

If aiming for the bonus, add a short paragraph somewhere in your web app using `st.write` describing the functionality you've added and how you included it. This section will be judged based on creativity, usefulness and technical challenge.

And there you have it!



Thanks for making it through this project journey! I know it was a lot of work, but I hope you've found it fun, interesting, and rewarding. I hope you all learned something that will help you in the next step of your career, be it searching for jobs in the industry, or carrying on in research. Good luck with whatever your future holds!

Group Evaluations

In addition to the grading described above, for each milestone you will be asked to score with how much you think everyone contributed to this milestone, and provide constructive feedback to your teammates, based on their strengths and potential areas of improvement. Students who do not contribute much will likely earn a poor grade, and in extreme cases might be removed from the group and asked to complete the projects individually.

For a team of size **N**, everyone in the team will have **N x 20** points to allocate between everyone in your group (including yourself). You will then assign everyone a score between **10 – 40**, where **10** corresponds to “half effort”, and **40** corresponds to “double effort” (**you cannot assign points outside of this range!**). In an ideal situation, everyone will contribute to the project equally and thus everyone will assign 20 points to every teammate. However, in the case where some people contributed less than others, you could assign them less points and give those points to who you think contributed more. Extreme cases will result in an instructor following up with the team to resolve any potential difficulties. This could include auditing git history in your repositories. *Any attempts to game the system will be handled manually and unfavourably!*

In addition to the score, you will also **give short feedback to your peers**, on both areas of strengths and potential areas of improvement. This feedback will be given privately and anonymously to each student. You can give feedback along several different axes, such as their **teamwork** (communication, reliability) and **technical work** (how much they contributed and the quality of their work/code). If you give a score below 20, you must give constructive feedback on areas that they can improve.

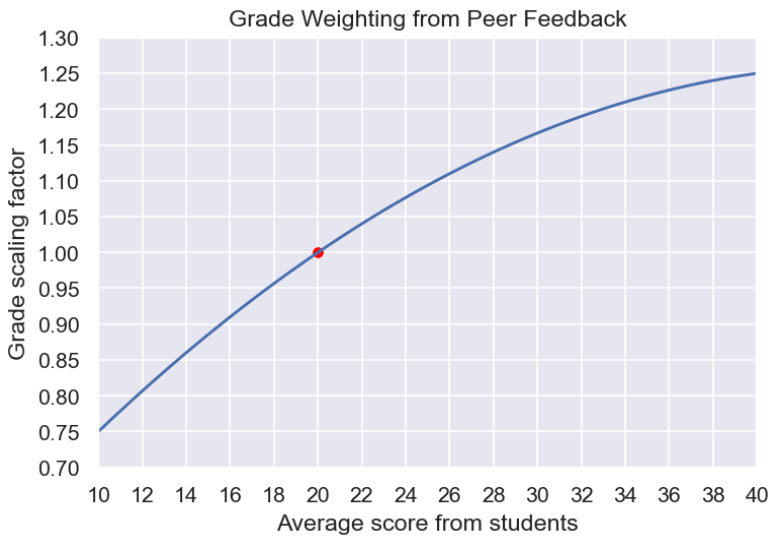
How scores impact your grade

- **x** is the average score for a student (excluding their own score), clamped between 10 ("half effort") and 40 ("double effort").
- A student's score on the milestone will be multiplied by the following weighting factor[2]:

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

- This system was adopted from Brian Fraser (SFU), which in turn is based on the work of Bill Gardner.

[2] *The coefficients are obtained by fitting the quadratic $y(x) = a*\mathbf{x}^2 + b\mathbf{x} + c$ to the points y(10) = 0.75, y(20) = 1.0, and y(40) = 1.25.*



Useful References

- IFT 6758 Hockey Primer
- Unofficial NHL API Documentation
- Wandb Python SDK
- https://docs.wandb.ai/guides/artifacts/download-and-use-an-artifact/
  - `artifact.download()`
- Flask Tutorial
- Full Stack Deep Learning (Lecture 11)
- Docker
  - Dockerfile best practices
- Streamlit Getting Started Guide
- Streamlit API Reference