IFT6758 Blog

Milestone 1

24 Oct 2022

1. Data Acquisition

Question 1.1

Hockey data is perfect for a data science project - rich, consistent, and always expanding. Play-by-play data is especially rich, and when combined with player information has a load of variables to sift through for patterns.

If you're looking to be able to download play-by-play data for a project, you're in luck! We've developed a function to download as much data as you want.

Overview

In this blog post, we're going to show you how to use a REST API to:

1. Download NHL hockey play-by-play data by the season.
2. Download play-by-play data for a single game, if you know the game id.

We'll try to explain what particular parts of the code are doing as we go.

What is a game id?

The game id is the unique identifier you'll use to access (and download) a particular game. The unofficial API doc (https://gitlab.com/dword4/nhlapi/-/blob/master/stats-api.md#game-ids) gives a good description of what a game id is. Basically, a game id is a 10-digit number encoding information about when the game occurred and what type of game it was. From the API doc: “The first 4 digits identify the season of the game (ie. 2017 for the 2017-2018 season). The next 2 digits give the type of game, where 01 = preseason, 02 = regular season, 03 = playoffs, 04 = all-star. The final 4 digits identify the specific game number”.

What is a REST API?

A REST API is an *application programming interface* that conforms to a set of design requirements known as REST, altogether providing a consistent and straightforward way to access data. Basically, it makes it much easier for us to get play-by-play data from the NHL.

Code

Batch downloading by season

Here’s the full code for a function that downloads a batch of data by season:

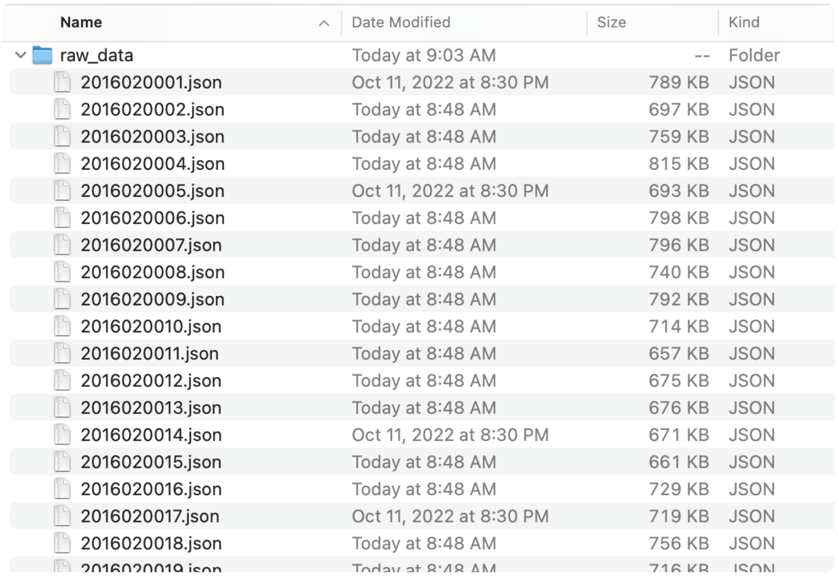
```
//Imports

import requests
import json
import os

//Function to download files
def download_json(allseasons, path):
    //Traverse through all seasons
    for season in allseasons:
        season = "https://statsapi.web.
nhl.com/api/v1/schedule?season="+str(se
ason)
        season_response = requests.get(
season)
        season_json = json.loads(season
_response.content)

        for item in season_json["dates"
]:
                games_id = map(lambda g
ame: game["gamePk"], item["games"])

                for game_id in games_id
:
                    //If the file has a
lready been downloaded:
```

```
                    if f"{game_id}.json
" in os.listdir(path):
                        //Print a messa
ge that the file is already there.
                        print(f"File {g
ame_id}.json is already in directory {p
ath}.")
                    else:
                        //Filter to be
 able to only select regular season (0
2) or playoff (03) games
                        game_type = str
(game_id)[4:6]
                        //Go through an
d download all games from regular seaso
n or playoffs
                        if game_type ==
'02' or game_type == '03':
                            //Traverse
 through all games
                            url="http
s://statsapi.web.nhl.com/api/v1/game/"+
str(game_id)+"/feed/live/"
                            response =
requests.get(url)
                            game = json
.loads(response.content)
                            with open(p
ath+str(game_id)+'.json', 'w', encoding
='utf-8') as f:
                                json.du
mp(game, f, ensure_ascii=False, indent=
4)
                                f.close
()
```

If we run the function given above using the following code:

```
path = "raw_data/"
allseasons = ["20162017","20172018","20
182019","20192020","20202021"]

download_json(allseasons, path)
```

we download each file and save it as a .json in the target directory, resulting in a huge set of raw files (Figure 1)!



Name	Date Modified	Size	Kind
raw_data	Today at 9:03 AM	--	Folder
2016020001.json	Oct 11, 2022 at 8:30 PM	789 KB	JSON
2016020002.json	Today at 8:48 AM	697 KB	JSON
2016020003.json	Today at 8:48 AM	759 KB	JSON
2016020004.json	Today at 8:48 AM	815 KB	JSON
2016020005.json	Oct 11, 2022 at 8:30 PM	693 KB	JSON
2016020006.json	Today at 8:48 AM	798 KB	JSON
2016020007.json	Today at 8:48 AM	796 KB	JSON
2016020008.json	Today at 8:48 AM	740 KB	JSON
2016020009.json	Today at 8:48 AM	792 KB	JSON
2016020010.json	Today at 8:48 AM	714 KB	JSON
2016020011.json	Today at 8:48 AM	657 KB	JSON
2016020012.json	Today at 8:48 AM	675 KB	JSON
2016020013.json	Today at 8:48 AM	676 KB	JSON
2016020014.json	Oct 11, 2022 at 8:30 PM	671 KB	JSON
2016020015.json	Today at 8:48 AM	661 KB	JSON
2016020016.json	Today at 8:48 AM	729 KB	JSON
2016020017.json	Oct 11, 2022 at 8:30 PM	719 KB	JSON
2016020018.json	Today at 8:48 AM	756 KB	JSON
2016020019.json	Today at 8:48 AM	716 KB	JSON

Figure 1. Contents of the raw_data folder after batch downloading.

The **requests** library does the heavy lifting of contacting the NHL and providing us the info we're looking for, while the **json** library helps with parsing the data that's received and saving it as a .json file.

Notice that because of the specification *if f"{game_id}.json" in os.listdir(path)*: the function only downloads the specified file if it isn't already in the target folder.

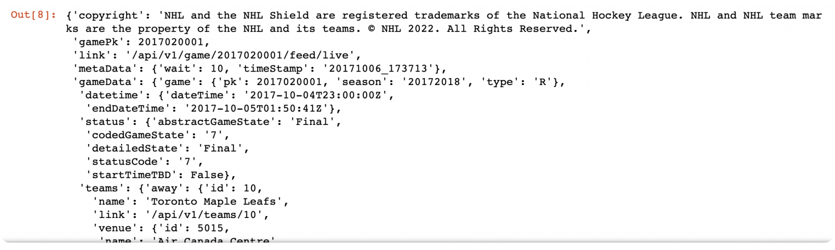
The specification *if game_type == '02' or game_type == '03'*: makes sure that we're only downloading regular season (02) or playoff (03) games - feel free to change this part if you want to use this code to download all games in a season.

Downloading a single game using the game id

If we want to download only one specific file, there's code for that too:

```
gameid = "2017020001"
url=f"https://statsapi.web.nhl.com/api/
v1/game/{gameid}/feed/live/"
response = requests.get(url)
game = json.loads(response.content)
path = "raw_data/"
if f"{gameid}.json" in os.listdir(path
):
        print(f"File {gameid}.json already
 present in dir {path}")
else:
        with open(path+gameid+'.json', 'w',
encoding='utf-8') as f:
                json.dump(game, f, ensure_ascii
=False, indent=4)
                f.close()
game
```

Just change the gameid and specify the download path you want. If run in a Jupyter Notebook, the last line (game) will show you the contents of the file you just downloaded, like this (Figure 2):



```
Out[8]: {'copyright': 'NHL and the NHL Shield are registered trademarks of the National Hockey League. NHL and NHL team mar
        ks are the property of the NHL and its teams. © NHL 2022. All Rights Reserved.',
         'gamePk': 2017020001,
         'link': '/api/v1/game/2017020001/feed/live',
         'metaData': {'wait': 10, 'timeStamp': '20171006_173713'},
         'gameData': {'game': {'pk': 2017020001, 'season': '20172018', 'type': 'R'},
          'datetime': {'dateTime': '2017-10-04T23:00:00Z',
           'endDateTime': '2017-10-05T01:50:41Z'},
          'status': {'abstractGameState': 'Final',
           'codedGameState': '7',
           'detailedState': 'Final',
           'statusCode': '7',
           'startTimeTBD': False},
          'teams': {'away': {'id': 10,
            'name': 'Toronto Maple Leafs',
            'link': '/api/v1/teams/10',
            'venue': {'id': 5015,
             'name': 'Air Canada Centre'
```

Figure 2. Output of downloading a single game using the standalone (non-batch) code.

Happy downloading!

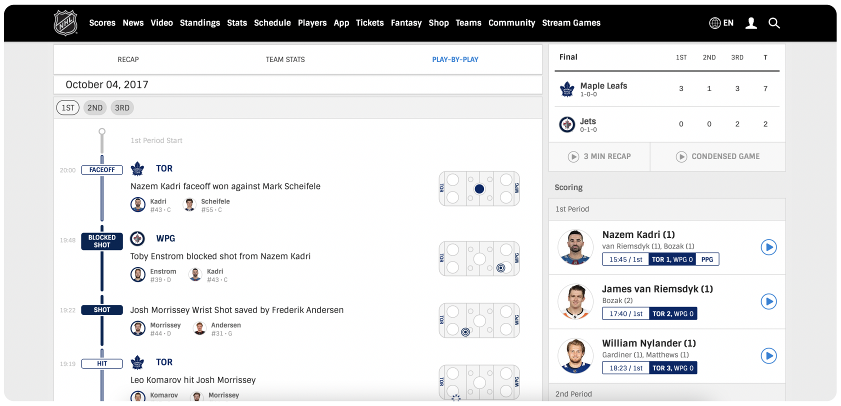
2. Interactive Debugging Tool

Question 2.1

The "Interactive Debugging Tool" allows the user (whoever runs the notebook cells) to flip through a set of hockey data by selecting a season, filter (regular season games or playoff games), game (given by game_id) and event in the game. The tool then prints out a set of descriptive information including the teams playing, the score, and a description of the event in question. If there are coordinates of the event, the tool draws these coordinates on an image of an ice rink.

As you can see below, the (graphically very pretty) results for a particular action (in this case, Toby Enstrom blocked shot from Nazeem Kadri) of a particular game from the

NHL website (Figure 3) is recapitulated in the debugging tool (Figure 4)


Scores News Video Standings Stats Schedule Players App Tickets Fantasy Shop Teams Community Stream Games
EN
RECAP
TEAM STATS
PLAY-BY-PLAY
1ST 2ND 3RD
3 MIN RECAP
CONDENSED GAME

October 04, 2017

1st Period Start

20:00 FACEOFF TOR
Nazem Kadri faceoff won against Mark Scheifele
Kadri #43 · C Scheifele #55 · C

19:48 BLOCKED SHOT WPG
Toby Enstrom blocked shot from Nazem Kadri
Enstrom #39 · D Kadri #43 · C

19:22 SHOT
Josh Morrissey Wrist Shot saved by Frederik Andersen
Morrissey #44 · D Andersen #31 · G

19:19 HIT TOR
Leo Komarov hit Josh Morrissey
Komarov Morrissey

Final	1ST	2ND	3RD	T
Maple Leafs 1-0-0	3	1	3	7
Jets 0-1-0	0	0	2	2

Scoring

1st Period

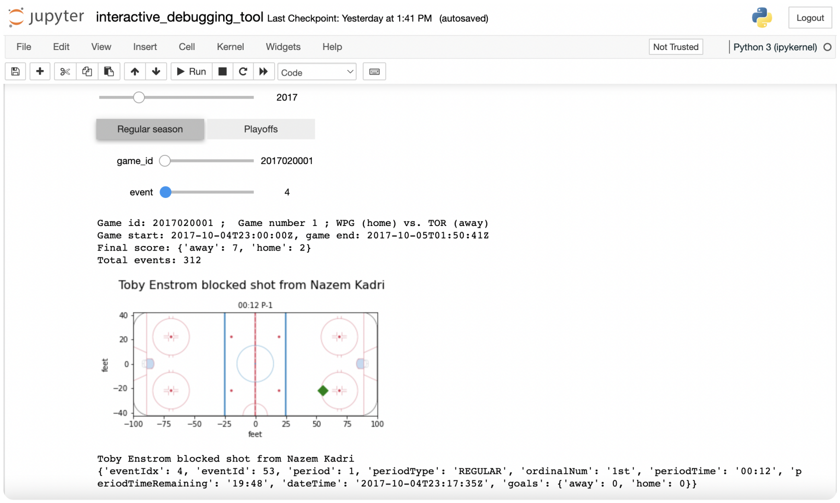
Nazem Kadri (1)
van Riemsdyk (1), Bozak (1)
15:45 / 1st TOR 1, WPG 0 PPG

James van Riemsdyk (1)
Bozak (2)
17:40 / 1st TOR 2, WPG 0

William Nylander (1)
Gardiner (1), Matthews (1)
18:23 / 1st TOR 3, WPG 0

2nd Period

*Figure 3. Play-by-play data from game 1 first period of 2017-10-04 from nhl.com (https://www.nhl.com/gamecenter/tor-vs-wpg/2017/10/04/2017020001/recap/plays#game=2017020001,game_state=final,lock_*


jupyter interactive_debugging_tool Last Checkpoint: Yesterday at 1:41 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Code
2017
Regular season
Playoffs
game_id 2017020001
event 4
Toby Enstrom blocked shot from Nazem Kadri
00:12 P-1
feet
40
20
0
−20
−40
−100 −75 −50 −25 0 25 50 75 100

```
Game id: 2017020001 ;  Game number 1 ; WPG (home) vs. TOR (away)
Game start: 2017-10-04T23:00:00Z, game end: 2017-10-05T01:50:41Z
Final score: {'away': 7, 'home': 2}
Total events: 312
```

```
Toby Enstrom blocked shot from Nazem Kadri
{'eventIdx': 4, 'eventId': 53, 'period': 1, 'periodType': 'REGULAR', 'ordinalNum': '1st', 'periodTime': '00:12', 'p
eriodTimeRemaining': '19:48', 'dateTime': '2017-10-04T23:17:35Z', 'goals': {'away': 0, 'home': 0}}
```

Figure 4: Output of the Interactive Debugging Tool for the same event as above.

In the "display.info()" function of the code, one could change the output to show whatever other stats or event data of interest.

Code

The code for the tool is as follows:

```
//Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import json

//Imports for JupyterLite
try:
    import piplite
    await piplite.install(['ipywidgets'
])
except ImportError:
    pass
import ipywidgets as widgets
from IPython.display import display


//Importing Data and needed files

def combine_jsons(folder):
    """
    Takes a directory (folder) and read
s all of the .json files in the folder
 into a dictionary.
    Returns a dictionary containing all
filenames as keys, the file contents ar
e the values.
    """
    //Make a dictionary to store all th
e jsons
    all_data = {}
    files_read = []
    //Go through each file in directory
    for filename in os.listdir(folder):
        //Parse the filename
        name = filename.split('.json')[
0]
        is_json = filename[-5:] == '.js
on'
        //If it's a json:
        if is_json:
            //Open and load file
            file = open(f"{folder}{file
name}")
            file_dict = json.load(file)
            files_read.append(name)
            //Add file to dictionary wi
th key as filename (minus .json)
            all_data[name] = file_dict
//print(f"Files read into dictionary
 \"data\": {files_read}")
    return all_data
//Read in the image of the rink
rink = plt.imread("nhl_rink.png")
```

```
//Load the data
data = combine_jsons('raw_data/')


//Widget-containing class


class Debugger:
    def __init__(self,data):

        //INITIALIZATION VARIABLE

        self.data = data

        //CORE VARIABLES

        self.current_data = data //dic
t, contains the current dictionary (may
be a pared-down version of data)
        self.current_season = str('2016
') //str, gives the current season
        self.current_game_type = 'Regul
ar season' //str, gives the current gam
e type ("Regular season" or "Playoffs")
        self.current_game_id = '2016020
532' //str, is the game currently being
focused on ; initializes at first posit
ion
        self.current_event = 1 //int, i
s the event currently being focused on

        //DESCRIBER VARIABLES (used in
 description of event)

        self.away_team = self.current_d
ata[self.current_game_id]['gameData']['
teams']['away']['abbreviation']
        self.home_team = self.current_d
ata[self.current_game_id]['gameData']['
teams']['home']['abbreviation']
        self.start_time = self.current_
data[self.current_game_id]['gameData'][
'datetime']['dateTime']
        self.end_time = self.current_da
ta[self.current_game_id]['gameData']['d
atetime']['endDateTime']
        self.final_score = self.current
_data[self.current_game_id]['liveData']
['plays']['allPlays'][-1]['about']['goa
ls']
        self.coordinates = self.current
_data[self.current_game_id]['liveData']
['plays']['allPlays'][self.current_even
t]['coordinates']
```

```
        self.event_description = self.c
urrent_data[self.current_game_id]['live
Data']['plays']['allPlays'][self.curren
t_event]['result']['description']
        self.event_time = f"{self.curre
nt_data[self.current_game_id]['liveDat
a']['plays']['allPlays'][self.current_e
vent]['about']['periodTime']} P-{self.c
urrent_data[self.current_game_id]['live
Data']['plays']['allPlays'][self.curren
t_event]['about']['period']}"
        self.about_event = self.current
_data[self.current_game_id]['liveData']
['plays']['allPlays'][self.current_even
t]['about']

        //WIDGETS

        //Controls the output of the ce
ll; makes sure messages don't build up
 in the cell.
        self.output = widgets.Output()
        //Slides to select a particular
season
        self.season = widgets.IntSlider
(
            value = self.season_range()
[0],
            min = self.season_range()[0
],
            max = self.season_range()[1
],
        )
        self.season.observe(self.update
_season,'value')
        //Toggles between playoff games
and regular season games
        self.game_type = widgets.Toggle
Buttons(options=['Regular season','Play
offs'])
        self.game_type.observe(self.upd
ate_game_type,'value')

        //Slides to select a particular
game
        self.game_id = widgets.Selectio
nSlider(
            options = sorted(list(self.
current_data.keys())),
            value = self.current_game_i
d,
            description='game_id',
        )
        self.game_id.observe(self.updat
e_game_id,'value')

        //Slides to select a particular
event in a game
        self.event = widgets.IntSlider(
            value = 1,
            min = 1,
            max = len(self.current_data
[self.current_game_id]['liveData']['pla
ys']['allPlays']) ,
            description = 'event'
        )
        self.event.observe(self.update_
event,'value')

    //HELPER FUNCTION TO CREATE WIDGETS

    def season_range(self):
        """
        Lists the files (labeled by gam
e_id) in the "data" dictionary and retu
rns a tuple indicating
        the minimum season and maximum
 season (both formatted as strings).
        Needed for "season" widget
        """
        min_season = 3000
        max_season = 0
        for game_id in self.data.keys
():
            season = int(game_id[:4])
            if season < min_season:
                min_season = season
            if season > max_season:
                max_season = season
        return (str(min_season), str(ma
x_season))

    //FUNCTIONS TO DISPLAY INFORMATION

    def plot_coordinates(self):
        """
        Takes self.coordinates (if not
 empty), which is a dictionary of coord
inates{'x':x,'y':y} and
        plots a point at these coordina
tes on a hockey rink image with a descr
iption + event_time as titles.
        """
        if self.coordinates:
            plt.scatter(x = self.coordi
nates['x'],y = self.coordinates['y'], z
order = 1, c = 'green', marker = 'D', s
= 100)
```

```
            plt.imshow(rink,zorder=0,ex
tent = [-100,100,-42.5,42.5])
            plt.title(self.event_time,f
ontsize = 10)
            plt.suptitle(self.event_des
cription, fontsize = 16, y=.90)
            plt.xlabel('feet')
            plt.ylabel('feet')
            plt.show()

    def display_info(self):
        """
        Displays select info about the
 currently selected event.
        """
        print(f"Game id: {self.current_
game_id} ;  Game number {self.current_g
ame_id[-4:].lstrip('0')} ; {self.home_t
eam} (home) vs. {self.away_team} (away)
")
        print(f"Game start: {self.start
_time}, game end: {self.end_time}")
        print(f"Final score: {self.fina
l_score}")
        print(f"Total events: {len(sel
f.current_data[self.current_game_id]['l
iveData']['plays']['allPlays'])}")
        self.plot_coordinates()
        print(self.event_description)
        print(self.about_event)

    //FUNCTIONS TO UPDATE VARIABLES/WID
GETS

    def filter_season(self):
        """
        Updates "self.current_data" wit
h only the entries of "data" that fit t
hat season.
        Needed for updating "current_da
ta" to only show particular season inf
o.
        """
        data_in_season = {}
        for game_id in self.data.keys
():
            if str(self.current_season)
== game_id[:4]:
                data_in_season[game_id]
= self.data[game_id]
        self.current_data = data_in_sea
son

    def filter_playoffs(self):
        """
        Updates the "self.current_data" dict to filter by playoffs  (depending on the state of self.game_type).
        Needed for updating "current_data" to only show info for regular season/playoffs.
        """
        data_in_playoffs = {}
        for game_id in self.current_data.keys():
            if self.current_game_type == "Playoffs":
                if game_id[4:6] == '03':
                    data_in_playoffs[game_id] = data[game_id]
            else:
                if game_id[4:6] == '02':
                    data_in_playoffs[game_id] = data[game_id]
        self.current_data = data_in_playoffs

    def update_vars(self):
        """
        Catch-all function, updates the descriptive variables used in display_info().
        """
        self.away_team = self.current_data[self.current_game_id]['gameData']['teams']['away']['abbreviation']
        self.home_team = self.current_data[self.current_game_id]['gameData']['teams']['home']['abbreviation']
        self.start_time = self.current_data[self.current_game_id]['gameData']['datetime']['dateTime']
        self.end_time = self.current_data[self.current_game_id]['gameData']['datetime']['endDateTime']
        self.final_score = self.current_data[self.current_game_id]['liveData']['plays']['allPlays'][-1]['about']['goals']
        self.coordinates = self.current_data[self.current_game_id]['liveData']['plays']['allPlays'][self.current_event]['coordinates']
        self.event_description = self.current_data[self.current_game_id]['liveData']['plays']['allPlays'][self.curren
```

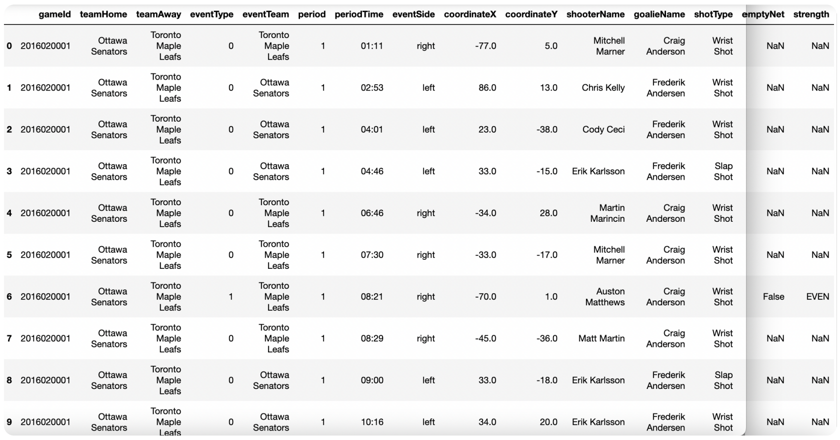
```
t_event]['result']['description']
        self.event_time = f"{self.curre
nt_data[self.current_game_id]['liveDat
a']['plays']['allPlays'][self.current_e
vent]['about']['periodTime']} P-{self.c
urrent_data[self.current_game_id]['live
Data']['plays']['allPlays'][self.curren
t_event]['about']['period']}"
        self.about_event = self.current
_data[self.current_game_id]['liveData']
['plays']['allPlays'][self.current_even
t]['about']

    def update_season(self,x):
        """
        Updates self.current_season wit
h the new input from the season widget.
        """
        self.output.clear_output()
        with self.output:
            self.current_season = x.new
            self.filter_season()
            self.filter_playoffs()
            self.game_id.options = sort
ed(list(self.current_data.keys()))
            self.game_id.value = self.c
urrent_game_id
            self.update_vars()
            self.display_info()

    def update_game_type(self,x):
        """
        Updates self.current_game_type
 with the new input from the game_type
 widget.
        """
        self.output.clear_output()
        with self.output:
            self.current_game_type = x.
new
            self.filter_season()
            self.filter_playoffs()
            self.game_id.options = sort
ed(list(self.current_data.keys()))
            self.game_id.value = self.c
urrent_game_id
            self.update_vars()
            self.display_info()

    def update_game_id(self,x):
        """
        Updates self.current_game_id wi
th the new input from the game_id widge
t.
```

```
        """
        self.output.clear_output()
        with self.output:
            self.current_game_id = x.ne
w
            self.update_vars()
            self.display_info()

    def update_event(self,x):
        """
        Updates self.current_event with
the new input from the event widget.
        """
        self.output.clear_output()
        with self.output:
            self.current_event = x.new
            self.update_vars()
            self.display_info()

    def display_output(self):
        display(self.output)



// Running the widget

d = Debugger(data)
display(d.season)
display(d.game_type)
display(d.game_id)
display(d.event)
d.display_output()
```

4. Tidy Data

Question 4.1

Raw data from the downloaded .json files was tidied and entered into a final pandas DataFrame (Figure 5).



	gameId	teamHome	teamAway	eventType	eventTeam	period	periodTime	eventSide	coordinateX	coordinateY	shooterName	goalieName	shotType	emptyNet	strength
0	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Toronto Maple Leafs	1	01:11	right	-77.0	5.0	Mitchell Marner	Craig Anderson	Wrist Shot	NaN	NaN
1	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Ottawa Senators	1	02:53	left	86.0	13.0	Chris Kelly	Frederik Andersen	Wrist Shot	NaN	NaN
2	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Ottawa Senators	1	04:01	left	23.0	-38.0	Cody Ceci	Frederik Andersen	Wrist Shot	NaN	NaN
3	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Ottawa Senators	1	04:46	left	33.0	-15.0	Erik Karlsson	Frederik Andersen	Slap Shot	NaN	NaN
4	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Toronto Maple Leafs	1	06:46	right	-34.0	28.0	Martin Marincin	Craig Anderson	Wrist Shot	NaN	NaN
5	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Toronto Maple Leafs	1	07:30	right	-33.0	-17.0	Mitchell Marner	Craig Anderson	Wrist Shot	NaN	NaN
6	2016020001	Ottawa Senators	Toronto Maple Leafs	1	Toronto Maple Leafs	1	08:21	right	-70.0	1.0	Auston Matthews	Craig Anderson	Wrist Shot	False	EVEN
7	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Toronto Maple Leafs	1	08:29	right	-45.0	-36.0	Matt Martin	Craig Anderson	Wrist Shot	NaN	NaN
8	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Ottawa Senators	1	09:00	left	33.0	-18.0	Erik Karlsson	Frederik Andersen	Slap Shot	NaN	NaN
9	2016020001	Ottawa Senators	Toronto Maple Leafs	0	Ottawa Senators	1	10:16	left	34.0	20.0	Erik Karlsson	Frederik Andersen	Wrist Shot	NaN	NaN

Figure 5: head(10) of the final dataframe generated after data tidying of downloaded files.

Note that the dataframe is so wide that we had to stitch two screenshots together to capture the two right-hand columns.

Question 4.2

Adding information to the strength field

You could add the actual strength information to both shots and goals by keeping track of the number of players allowed on the ice at the time. In fact, it's theoretically possible to keep track of how many players there were for any event.

The dataset contains a list of plays (events) that resulted in penalties. If you go to one of these events, you see that the event results in a penalty, and also some very useful information:

- what time the penalty occurred (period number and period time)
- who the penalty was on (player and team)
- the kind of penalty (e.g. "Minor") and the number of minutes of penalty that were awarded.

You could use this information to decrement the number of players allowed on the ice for the team the penalty was on (e.g. 5v4, 5v3,4v4…), and maintain that decrease for any events that occur during the subsequent penalty time.

Question 4.3

Additional features that could be created

Additional features one could consider creating from the data available in this dataset include:

1. Faceoffs won: given in [‘allPlays’] is an eventTypeId and a playerType: ‘Winner’ or ‘Loser’. If one wins the faceoff, it seems possible they could be more likely to convert it into a goal using the initiative they’ve gained.

2. Player information: There are really a lot of features available through ‘gameData’‘players’ that we haven’t exploited here. In this dataset we have every player in the game, their age, position (center, Forward), height, weight and other factors (rookie) that could contribute to their effectiveness.

3. Rebounds: We can see whether a shot occurred within <5 seconds (picked somewhat arbitrarily) of a teammate taking a shot, and occurred close to the goal. In the confusion after a blocked (but unstopped) shot, there might be a higher chance of scoring a goal.

In general, there is a potentially rich feature space that could be mined by looking at temporal information and location/coordinate information (e.g. whether events happen within a short time of each other - for instance, a shot off the rush could occur if one team takes a shot, and then the opposing team takes a shot on the opposite side of the rink shortly afterwards).

5. Simple Data Visualization

Question 5.1

Shots over all teams

We can look at the number of shot types aggregated over all teams in a given season (Figure 6) and quickly see what the most common type of shot is overall: the wrist shot.

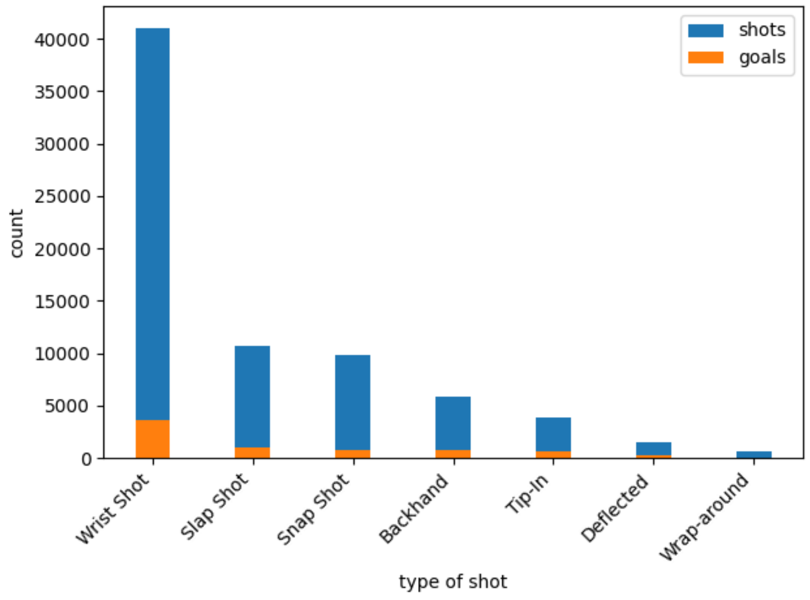


Figure 6. Total numbers of shots and goals in the 2019-20 season, broken down by type of shot. Goals are shown here overlaid on shots.

From the above figure it is difficult to confidently say what the most "dangerous" type of shot is. Let's define the most "dangerous" type of shot as the one that has the highest chance of succeeding ie. a bigger proportion of shots of this type result in goals than other types do. If we squint at the above figure, it looks like the most dangerous type is a "deflected" shot, followed closely by "tip-in" shots, but since there are so many more shots than goals and since the "wrist shot" category is so bigger than any others, it is very difficult to tell.

A slightly easier way to view the same data is by plotting the y axis on a logarithmic scale, like Figure 7. People aren't as used to seeing a log scale and without carefully looking at the axis, they might not recognize how much bigger the "wrist shot" category is than the others, but it has the advantage of making the other categories more visible. Still, it's hard to determine exactly which is the most "dangerous" type of shot based on raw counts - a clearer analysis would look at success rate, plotting goals/shots (percent of shots that succeeded) for each category.

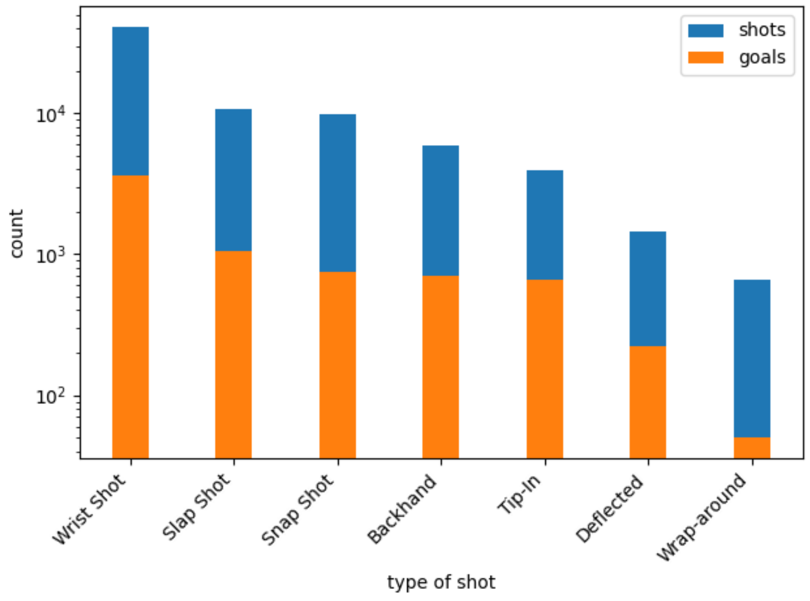


Figure 7: Total numbers of shots and goals in the 2019-20 season plotted on a logarithmic axis, broken down by type of shot. Goals are shown here overlaid on shots.

We chose a barplot because it shows the result clearly and intuitively: the higher the bar, the more shots of that type.

Question 5.2

Relationship between shot distance and success

Generally, the closer the shooter is to the net the higher their chance of success. Far more shots are taken from within 75 ft of the goal. Below, we've plotted the relationship between the distance a shot was taken and the chance it was a goal for each season between 2018-19 and 2020-21, inclusive (Figures 8-10).

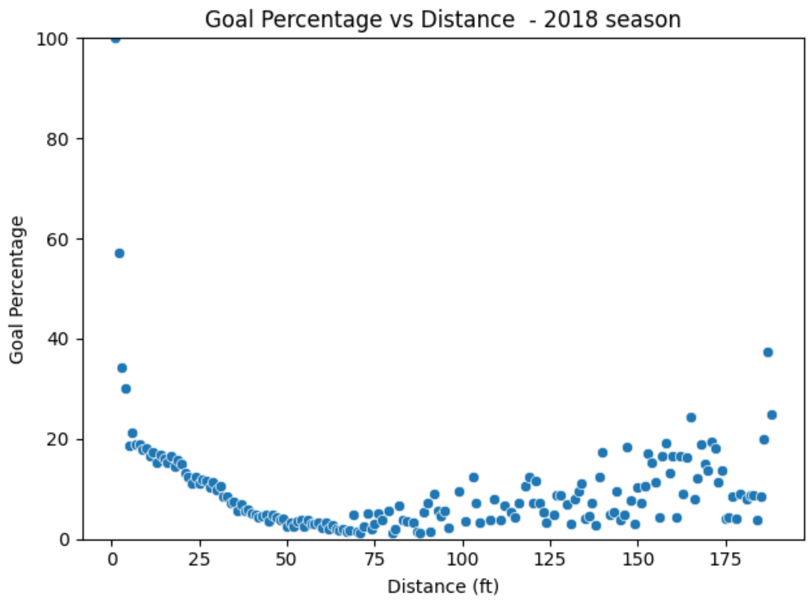


Figure 8. Relationship of shot distance and goal chance over the 2018-19 season.

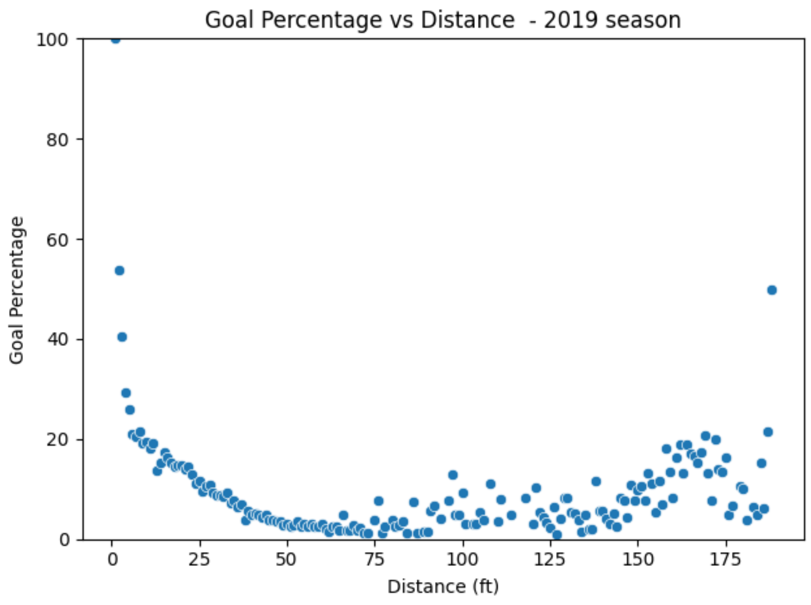


Figure 9. Relationship of shot distance and goal chance over the 2019-20 season.

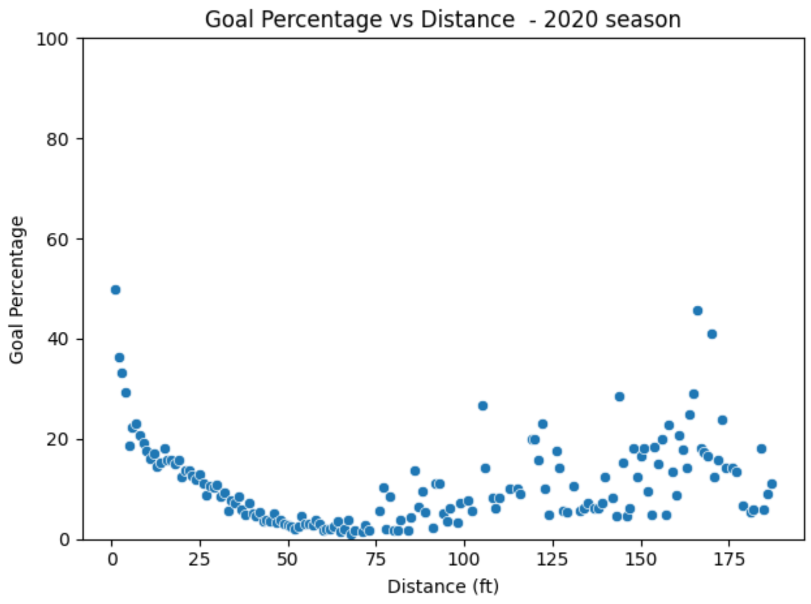


Figure 10. Relationship of shot distance and goal chance over the 2020-21 season.

Even though we've plotted a number of shots that seem to come from across the rink (>100 ft), we might want to treat that side of the graph with skepticism. It seems plausible that a hail-mary shot from farther down the ice could result in a goal sometimes, but the number of points we see suggests that we may want to scrutinize the data more closely. There may be something in the underlying data that our transformation (flipping all events that occurred on the left of the rink over to the right) did not take into account, such as mistakes in data entry.

If we compare the distance:success relationships between the three seasons we've plotted, the region between 0 and 75 ft looks rather similar between them. The region between 100 and 150 feet *may* be a bit broader (the region contains more successful shots) for the 2020-21 season (Figure 10) but we should not confidently make that conclusion from these figures.

We chose to plot these data as scatterplots because they accurately represent the data; a lineplot would not necessarily give the right impression of how scattered the shots far away from the goal (> 100 ft) are.

Question 5.3

Combine the information from the previous sections to produce a figure that shows the goal percentage (# goals / # shots) as a function of both distance from the net, and the category of shot types (you can pick a single season of your choice). Briefly discuss your findings; e.g. what might be the most dangerous types of shots?

Combining the information from the previous sections, we can look at the data in a couple of different ways. The first is as a density plot (Figure 11) which essentially shows the probability mass of successful shots as a function of distance.

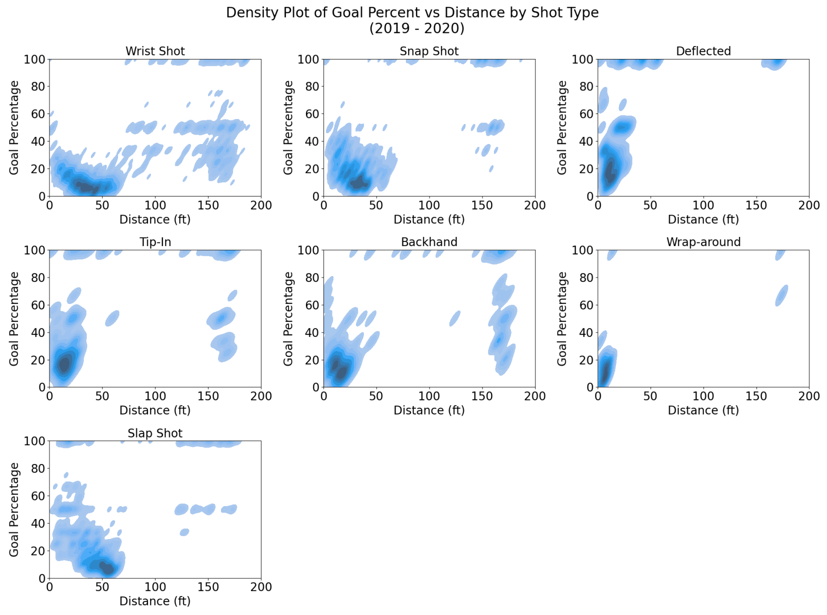


Figure 11. Relationships between success of shots and distance from the net for the 2019-20 season.

Looking at where the blobs are darker, we can see that “deflected” shots and “tip-in” shots seem to be both quite dangerous at close range and generally more successful, with a mass that stretches farther up the y-axis of success and even a reasonably-sized mass hovering up around the 100% region. “Wrist shots” are clearly less successful overall, being fairly compressed and generally lower than 20% successful.

Interestingly, “slap shots” have a different look to them than other shots, being centred farther out than other kinds (other than wrist shots). This probably reflects the nature of the shot - it's hard to get a good slap-shot from close in, in the heat of the moment.

We can also represent the above data as a violin plot (Figure 12), where the shape of the violin along the y dimension represents a probability distribution.

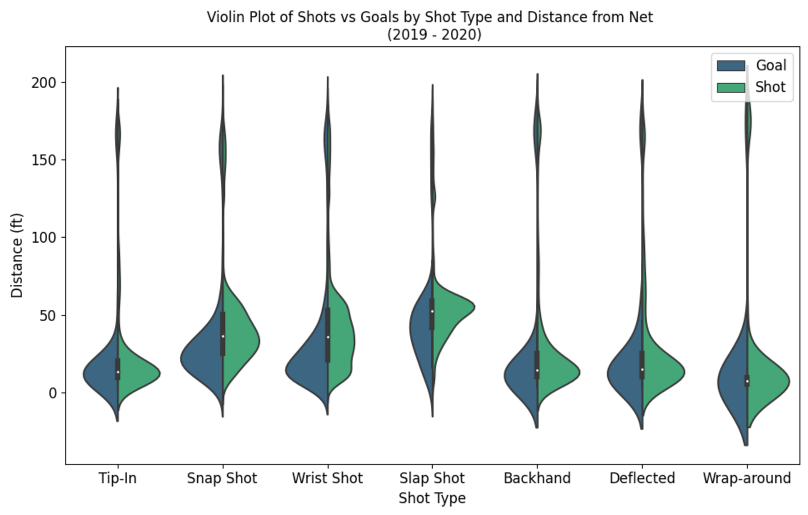


Figure 12. Relationship of shot distance and goal chance over the 2019-20 season.

Using this type of plot, you can see for example that "tip-ins", "backhands" and "deflected" shots have a fairly symmetrical distribution, suggesting that most shots taken at a particular distance are goals. For "snap shots", "wrist shots" and "slap shots" the unevenness of the distributions for shots vs goals indicates that these kind of shots aren't very successful on average.

6. Advanced Visualizations: Shot Maps

Question 6.1

Question 6.2

These plots allow you to see where a team's shot position behaviour differs from that of other teams. In other words, where this team takes more of their shots from than all other teams do. Red areas indicate that they take more shots from a certain region, and blue indicates that they take fewer, while white is neutral (same as the league average).

This gives an insight into either psyche or strategy; either the team in question is composed of more players who feel more or less comfortable shooting from particular regions than other players would, or the coaches have told their players to shoot from particular regions if they can.

Question 6.3

During the 2016-17 season, the Colorado Avalanche was less likely than the league average to take shots close to the goal or to the right of the goal. They seem to have generally preferred to take their shots from farther out and left compared to other teams.

During the 2020-21 season, in contrast, even though the Avalanche still liked to shoot from the out-and-left position they shifted their behaviour to *more* aggressively shoot from the middle of the rink and closer in to the net than other teams.

This shift in behaviour, combined with the knowledge that shots are generally more successful closer to the net (see discussions in Simple Visualization), makes it easier to understand why the Colorado Avalanche went from the bottom of their division (2016-17 season) with a 22:56 win:loss to the top of the division (2020-21 season) with a 56:19 win:loss, according to a quick Googling.

Question 6.4

Generally, the Buffalo Sabres have struggled to shoot from the centre of the rink and towards the net, while Tampa Bay have been better at exploiting the middle of the ice (not usually too close to the goal, but directly in front of the goalie). Even though Tampa Bay tends to shoot from closer to the net than Buffalo does, which is a (very general) predictor of success, it is hard to use only these data to fully explain why Tampa Bay has been so much more successful than Buffalo. It is rare that any data analysis like this will paint a particularly complete picture, since we need to section the data in different ways to be able to draw more nuanced conclusions.

Older Newer