Formula 1 Predictions

Caleb Koresh, Jacob Nakamura, Alex Jones

Outline

Context

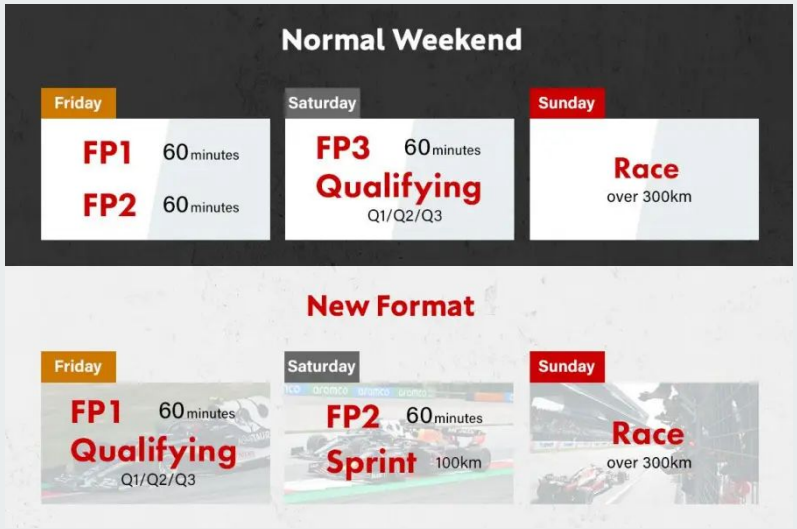
Data Collection and Cleaning

Data Visualization

Neural Networks

Improvements

Context



Formula One is the highest level of motorsport, where drivers participate in Grand Prix Weekends

Grand Prix Weekends Include:

- Friday: Practice 1, Practice 2
- Saturday: Practice 3, Qualifying Session
- Sunday: Race!

Our goal is to use Practice Data to predict Qualifying Times.

Note: Some Grand Prix Weekends are defined as “Sprint Weekends”, where only one practice is done before qualifying.

Data Collection and Cleaning

Main packages used: fastf1.api, dfply

Collecting the data we need

To do this, we will use fastf1.api to collect data and store it as a dataframe.

```python
def practiceScrape(grandPrix, raceDate, pracNum, pracDate):
    """
    Input: Grand Prix, Grand Prix Date, Which Practice, Which Date
    Output: Dataframe with each driver's timed lap matched with tire compound and weather conditions
    """
    #Make Path to scrape practice data
    FP = fastf1.api.make_path(grandPrix, raceDate, f'Practice {pracNum}', pracDate)

    #Scraping timing data, timing app data (for info on tire compund), and weather data
    FPTiming = fastf1.api.timing_data(FP)[0]
    FPTimingApp = fastf1.api.timing_app_data(FP)
    FPWeatherDict = fastf1.api.weather_data(FP)
    FPWeatherData = pd.DataFrame.from_dict(FPWeatherDict)
```

General Data Cleaning

```
#Remove entries with no laptimes, and round session time to nearest minute (for merging purposes)
FPTiming = FPTiming[FPTiming.LapTime.notnull()]
FPTimingApp >>= mutate(Time = X.Time.round('60s')) >> arrange(X.Driver, X.Time)
FPTimingApp = FPTimingApp.reset_index()

#In timing app dataframe, tire compound is only noted when new set is put on
#Since dataframe is organized by driver and then time, we can replace empty entries with the previous tire compound
#Also, dataframe sometime doubles on driver-time combo, so removing the unnecessary ones
for index in range(len(FPTimingApp)):
    if not FPTimingApp.loc[index,"Compound"]:
        FPTimingApp.loc[index, "Compound"] = FPTimingApp.loc[index-1, "Compound"]
    if not FPTimingApp.loc[index,"New"]:
        FPTimingApp.loc[index,"New"] = False
    if index == 0:
        continue
    if FPTimingApp.loc[index,"Time"] == FPTimingApp.loc[index-1,"Time"]:
        if not FPTimingApp.loc[index,"LapTime"]:
            FPTimingApp.loc[index,"Driver"] = np.nan
        else:
            FPTimingApp.loc[index-1,"Driver"] = np.nan
FPTimingApp = FPTimingApp[FPTimingApp.Driver.notnull()]
```

	LapNumber	Driver	LapTime	Stint	TotalLaps	Compound	New	TyresNotChanged	Time	LapFlags	LapCountTime	StartLaps	Outlap
167	NaN	1	NaT	0	0.0	SOFT	True	0	0 days 00:32:15.340000	0.0	None	0.0	None
197	NaN	1	NaT	0	1.0	None	None	None	0 days 00:34:00.332000	NaN	None	NaN	None
216	NaN	1	NaT	0	2.0	None	None	None	0 days 00:35:25.340000	NaN	None	NaN	None
217	3.0	1	0 days 00:01:28.541000	0	NaN	None	None	None	0 days 00:35:25.380000	3.0	None	NaN	None
241	NaN	1	NaT	0	3.0	None	None	None	0 days 00:38:00.530000	NaN	None	NaN	None
...	...	...	...	...	...	...	...	...	...	...	...	...	...
596	NaN	77	NaT	3	7.0	SOFT	False	1	0 days 01:14:30.695000	0.0	None	7.0	None
616	NaN	77	NaT	3	8.0	None	None	None	0 days 01:16:05.704000	NaN	None	NaN	None
643	NaN	77	NaT	3	9.0	None	None	None	0 days 01:18:00.761000	NaN	None	NaN	None
644	20.0	77	0 days 00:01:58.649000	3	NaN	None	None	None	0 days 01:18:01.021000	NaN	None	NaN	None
668	NaN	77	NaT	3	10.0	None	None	None	0 days 01:20:05.792000	NaN	None	NaN	None

676 rows × 13 columns

	Stint	Driver	TotalLaps	Compound	New	Time
0	0	1	0.0	SOFT	True	0 days 00:32:00
1	0	1	1.0	SOFT	False	0 days 00:34:00
3	0	1	NaN	SOFT	False	0 days 00:35:00
4	0	1	3.0	SOFT	False	0 days 00:38:00
5	0	1	4.0	SOFT	False	0 days 00:40:00
...	...	...	...	...	...	...
670	2	77	7.0	SOFT	False	0 days 01:12:00
671	3	77	7.0	SOFT	False	0 days 01:15:00
672	3	77	8.0	SOFT	False	0 days 01:16:00
674	3	77	NaN	SOFT	False	0 days 01:18:00
675	3	77	10.0	SOFT	False	0 days 01:20:00

507 rows × 6 columns

Merging Dataframes

The merged dataframe gives us detailed information about each individual lap time, such as tire compound used and weather conditions

```
#Merge Dataframes, keyed by time and driver, then dropping irrelevant columns
FPCompiledTiming = pd.merge(FPTiming,FPTimingApp, on=['Time', 'Driver'])
FPWeatherData >>= mutate(Time = X.Time.round('60s'))
FPCompiledTiming = pd.merge(FPCompiledTiming, FPWeatherData, on=['Time'])
FPCompiledTiming >>= arrange(X.Driver, X.Time) >> mutate(Practice = pracNum) >> drop(X.WindDirection, X.Stint, X.TotalLaps)
FPCompiledTiming = FPCompiledTiming.reset_index()
return FPCompiledTiming
```

Function: Weekend Compiler

Creates dataframes for each practice session, concatenates all of them, and then merges them with the qualifying dataframe

```
def weekendCompiler(grandPrix, raceDate, sprint = False):
    """
    Input: Grand Prix with appropriate race date, and whether grand prix was a sprint weekend
    Output: Dataframe with all timed practice laps, merged with info about best qualifying lap
    """
    #Creating relevant dates for session calling purposes
    raceDate = datetime.strptime(raceDate, "%Y-%m-%d")
    friDate = str(raceDate - timedelta(days = 2))[:10]
    satDate = str(raceDate - timedelta(days = 1))[:10]
    raceDate = str(raceDate)[:10]

    #Collecting Practice Data
    #Note: On sprint weekends only one practice is done before qualifying, so FP1 is the only relevant data
    FP1CompiledTiming = practiceScrape(grandPrix, raceDate, 1, friDate)
    if not sprint:
        FP2CompiledTiming = practiceScrape(grandPrix, raceDate, 2, friDate)
        FP3CompiledTiming = practiceScrape(grandPrix, raceDate, 3, satDate)

    #Collecting Quali Data:
    #Note: On sprint weekends, qualifying is done on Friday, as opposed to usual Saturdays
    if not sprint:
        QualiDF = QualiScrape(grandPrix, raceDate, satDate)
    else:
        QualiDF = QualiScrape(grandPrix, raceDate, friDate)

    #Concatenating Practice Dataframes
    if not sprint:
        frames = [FP1CompiledTiming, FP2CompiledTiming, FP3CompiledTiming]
        practiceCompiled = pd.concat(frames)
    else:
        practiceCompiled = FP1CompiledTiming

    #Merging Quali Data with compiled Practice data, such that each laptime has the appropriate quali data
    practiceCompiled = practiceCompiled.reset_index()
    practiceCompiled >>= drop(X.level_0)
    practiceCompiled = pd.merge(practiceCompiled, QualiDF, on=['Driver']) >> mutate(Weekend = grandPrix)

    return practiceCompiled
```

Creating the master list

```
raceCalendar = {1:["Bahrain Grand Prix", "2022-03-20"], 2:["Saudi Arabian Grand Prix", "2022-03-27"], 3:["Australian Grand Prix",
                4:["Emilia Romagna Grand Prix", "2022-04-24"], 5:["Miami Grand Prix", "2022-05-08"], 6:["Spanish Grand Prix", "26
                7:["Monaco Grand Prix", "2022-05-29"], 8:["Azerbaijan Grand Prix", "2022-06-12"], 9:["Canadian Grand Prix",  "202
                10:["British Grand Prix", "2022-07-03"], 11:["Austrian Grand Prix", "2022-07-10"], 12:["French Grand Prix", "2022
                13:["Hungarian Grand Prix", "2022-07-31"], 14:["Belgian Grand Prix", "2022-08-28"], 15:["Dutch Grand Prix", "2022
                16:["Italian Grand Prix", "2022-09-11"], 17:["Singapore Grand Prix", "2022-10-02"], 18:["Japanese Grand Prix", "2
                19:["United States Grand Prix", "2022-10-23"], 20:["Mexico City Grand Prix", "2022-10-30"], 21:["São Paulo Grand
dfList = []
for key in raceCalendar:

    print(f"Collecting Data for {raceCalendar[key][0]}")
    #Imola, Austria, and Brazil are sprint weekends, so they have a special case where sprint = true
    if key not in [4,11,21]:
        dfList.append(weekendCompiler(raceCalendar[key][0], raceCalendar[key][1]))
    else:
        dfList.append(weekendCompiler(raceCalendar[key][0], raceCalendar[key][1], sprint = True))

masterList = pd.concat(dfList)
masterList >>= drop(X.Time)
```

	Driver	LapTime	NumberOfLaps	NumberOfPitStops	Sector1Time	Sector2Time	Sector3Time	SpeedI1	SpeedI2	SpeedFL	...	Practice	QualifyingLapTime
0	1	97.766	2.0	0.0	30.695	41.580	25.491	228.0	263.0	176.0	...	1	90.681
1	1	94.783	5.0	1.0	30.119	40.789	23.875	232.0	263.0	278.0	...	1	90.681
2	1	128.760	6.0	1.0	42.765	60.164	25.831	146.0	181.0	278.0	...	1	90.681
3	1	94.742	7.0	1.0	30.224	40.815	23.703	231.0	263.0	279.0	...	1	90.681
4	1	113.218	8.0	2.0	34.951	49.152	29.115	179.0	200.0	NaN	...	1	90.681
...	...	...	...	...	...	...	...	...	...	...	...	...	...
464	77	121.193	27.0	4.0	53.614	49.680	17.899	263.0	230.0	308.0	...	1	75.486
465	77	73.561	28.0	4.0	18.662	37.741	17.158	316.0	257.0	317.0	...	1	75.486
466	77	118.153	29.0	4.0	22.166	66.874	29.113	143.0	31.0	299.0	...	1	75.486
467	77	121.426	30.0	4.0	26.704	52.349	42.373	239.0	136.0	45.0	...	1	75.486
468	77	106.958	31.0	4.0	43.255	42.261	21.442	300.0	227.0	NaN	...	1	75.486

17577 rows × 30 columns

Data Visualization

Visualizing Our Data

To check for patterns and the presence of outliers in our data, visualization was a good option*. Data visualization involved:

1. Making a class to encapsulate data for a single plot
2. Making a class to generate subplots
3. Reorganizing dataframes into dictionaries to make data more easily manageable and accessible by said classes
4. Plotting data for each field for each race for each driver

*Unfortunately, we ended up doing this part towards the end

The PlotData Class

Manages data for individual plots

```
%matplotlib inline
class PlotData():
    """
    A module for storing plot data in a more portable, reusable form
    Stores x and y data and metadata (title, labels, etc.)
    Parameters: 2 arrays of x and y datasets (to allow multiple relations per plot),
                plot type as a string, title, axis labels, and grid boolean
    """
    def __init__(self, xdata, ydata, ptype='plot', title='Figure', xlabel='x axis', ylabel='y axis', grid=True):

        # Only check these if we have ydata
        if ydata != None:
            # Throws error if xdata and ydata length do not match
            if not len(xdata) == len(ydata):
                raise Exception('Data mismatch error: must have the same number of corresponding x and y fields')

            # Throws error if x and y subdata length do not match
                for i in range(len(xdata)):
                    if not len(xdata[i]) == len(ydata[i]):
                        raise Exception(f'Data mismatch error: {i}th set of x and y data have unequal length')


        self.xdata = xdata
        self.ydata = ydata
        self.ptype = ptype
        self.title = title
        self.xlabel = xlabel
        self.ylabel = ylabel
        self.grid = grid


    def plot(self, style='b--'):
        """
        May not be showing it but it do be plotting the data
        Input: Optional style
        """
        # Plots based on type
        if self.ptype == 'plot':
            for i in range(len(self.xdata)):
                plt.plot(self.xdata[i], self.ydata[i], style)

        elif self.ptype == 'scatter':
            for i in range(len(self.xdata)):
                plt.scatter(self.xdata[i], self.ydata[i], style)

        elif self.ptype == 'bar':
            for i in range(len(self.xdata)):
                plt.bar(self.xdata[i], self.ydata[i])

        elif self.ptype == 'boxplot':
            plt.boxplot(self.xdata)

        else:
            print(f'{self.title} has invalid plot type')

        plt.title(self.title)
        plt.xlabel(self.xlabel)
        plt.ylabel(self.ylabel)
        if self.grid:
            plt.grid()


    def addData(x, y):
        """
        Clears the figure and adds a new x-y relation to the data
        Input: x and y data to add
        Output: It adds the data wow
        """
        plt.clf()
        self.xdata.append(x)
        self.ydata.append(y)
```

The Plotter Class

Organizes PlotData instances into a figure of subplots

```
class Plotter():
    """
    Takes plot data and auto-generates subplots
    Parameters: a list of PlotData objects
    """

    def __init__(self, plots, subdim=None):
        self.plots = plots
        self.subdim = subdim
        self.arrange(self.subdim)
        self.figsize = (5*self.subdim[1], 5*self.subdim[0])


    def addPlot(plot, loc=-1, subdim=None):
        """
        Adds a new subplot and rearranges the figure
        Input: the PlotData to add, the location to display it, and a custom aspect ratio if desired
        Output:  It does it
        """
        self.plots.insert(loc, plot)
        if not subdim == None:
            self.subdim = subdim
        else:
            arrange()


    def addData(x, y, loc):
        """
        Adds data to a subplot
        Input: x and y data to add, and which subplot to add to
        Output: It also does it
        """
        self.plots[loc].addData(x, y)


    def arrange(self, subdim=None):
        """
        Helper function to redistrubte subplots to maximize squareness or fulfill user-set dimensions
        Input: Optional custom dimensions
        Output: :)
        """
        if not subdim == None:
            # Throw error if there are not enough subplot spaces
            if subdim[0] * subdim[1] < len(self.plots):
                raise Exception('Specified plot dimensions cannot fit specidied subplot data. Try doing math')
            self.subdim = subdim
        else:
            # Find the arrangement closest to square that fills all subplots
            for i in range(int(np.sqrt(len(self.plots))), 0, -1):
                if len(self.plots) % i == 0:
                    self.subdim = (i, len(self.plots) // i)
                    break


    def plot(self):
        """
        Generates the figure
        """
        plt.figure(figsize=self.figsize)
        for i in range(len(self.plots)):
            plt.subplot(self.subdim[0], self.subdim[1], i + 1)
            self.plots[i].plot()
        plt.tight_layout()
```

Reorganizing the Data pt. 1

Turning dataframes into nested dictionaries

```
def splitDriverData(weekendDF):
    """
    Generates a dictionary of driver IDs to driver stats
    Input: Dataframe
    Output: Dictionary of driver IDs to driver stat dataframe,
    which is a subset of the input
    """
    idList = list(set(weekendDF['Driver'].values))
    return {ID : weekendDF.loc[weekendDF['Driver'] == ID] for ID in idList}
```

```
plotDataFrame = masterList.copy()

# Split data into a dictionary by race
raceNames = [race[0] for race in raceCalendar.values()]
raceDataFrames = {race : plotDataFrame.loc[plotDataFrame["Weekend"] == race] for race in raceNames}

# Split dictionary of races further into to dictionaries of driver IDs to driver data
# Get a specific stat with driverDataFrames['raceName']['driverID']['field'].values
driverDataFrames = {name : splitDriverData(raceDataFrames[name]) for name in raceNames}

# Make sorted list of driver IDs
driverIDs = list(set([int(ID) for ID in plotDataFrame['Driver'].values]))
driverIDs.sort()
driverIDs = [str(ID) for ID in driverIDs]

# Select races to plot
selectedRaces = raceNames
```

Reorganizing the Data pt. 2

Turning dataframes into nested dictionaries

```
def getDataField(field):
    """
    Takes a field name and produces a dictionary of races to
    dictionaries of drivers to numpy arrays of driver data
    Input: field to extract
    """
    dataField = {}
    buffer = {}

    for race in selectedRaces:
        for ID in driverIDs:
            try:
                arr = driverDataFrames[race][ID][field].dropna().values
                if arr.dtype == 'timedelta64[ns]':
                    buffer[ID] = arr.astype('float64') / 1e9
                else:
                    buffer[ID] = arr
            except KeyError:
                buffer[ID] = np.array([0])

        dataField[race] = buffer.copy()
        buffer.clear()

    return dataField
```

Plotting the Data

Making the automated plotting process iterative

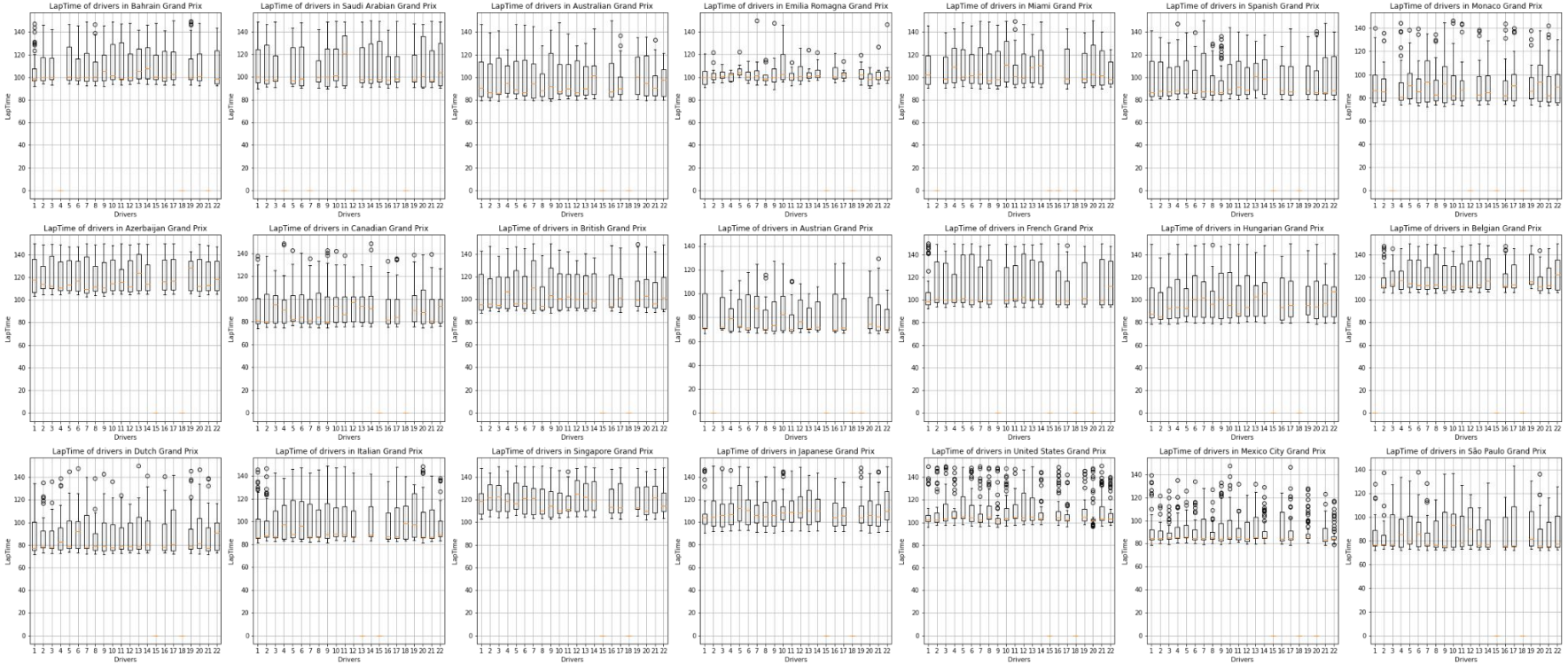
```python
def plotField(field, saveName=None):
    """
    Given a field name, generates boxplots for that field for
    each driver across each race, optionally saves an image
    Input: field to plot, file name to save plot as
    """
    # Gets nested dictionaries of field
    dataField = getDataField(field)
    fieldPlotData = []

    # Generate list of PlotData
    for race, driverData in dataField.items():
        raceData = [driverField for driverField in driverData.values()]
        title = f'{field} of drivers in {race}'
        fieldPlotData.append(PlotData(raceData, None, 'boxplot', title, 'Drivers', field))

    # Create the figure
    plotter = Plotter(fieldPlotData)
    plotter.plot()

    # Save image if necessary
    if saveName != None:
        plt.savefig('plots/' + saveName)
```

```python
# Lists of field names
timeFields = ['LapTime', 'Sector1Time', 'Sector2Time','Sector3Time']
speedFields = ['SpeedI1', 'SpeedI2', 'SpeedFL', 'SpeedST']
environmentFields = ['AirTemp', 'Humidity', 'Pressure', 'TrackTemp', 'Windspeed']
```

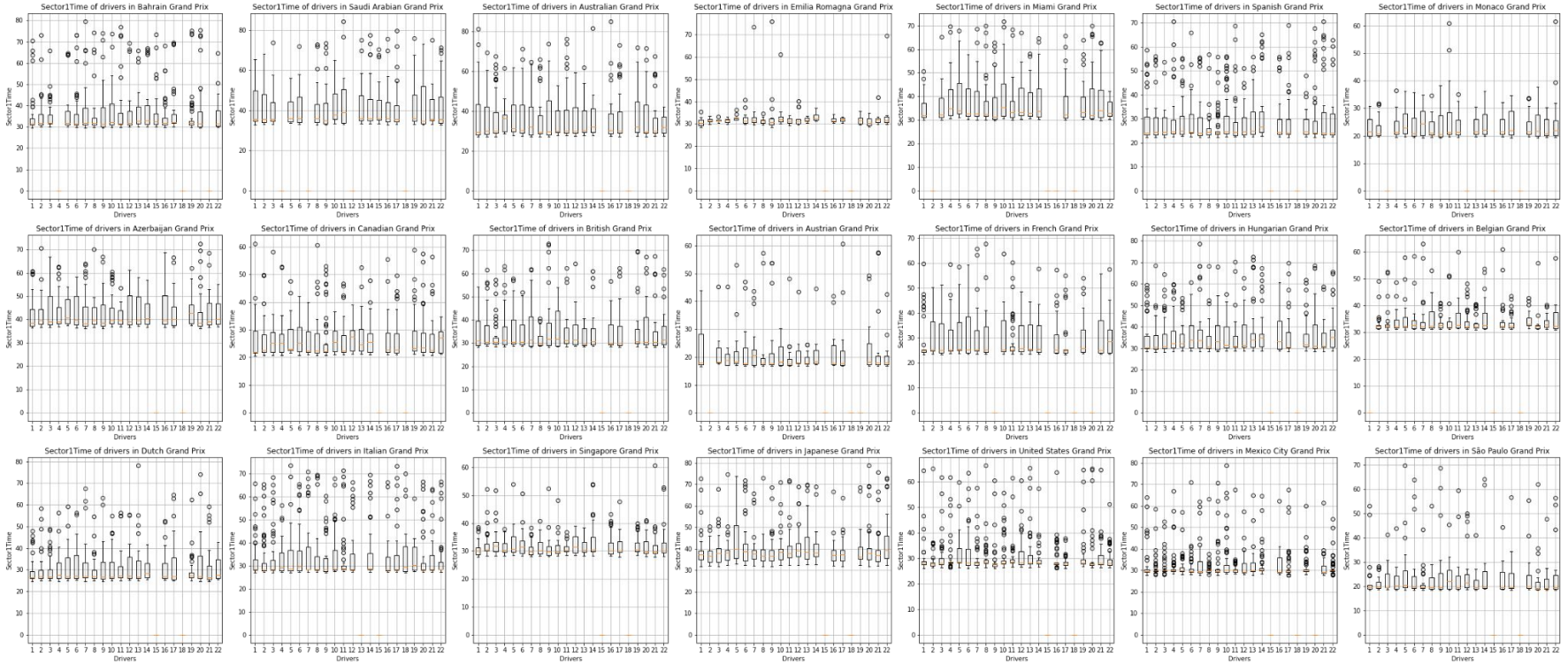
```python
# Plot practice data
allFields = timeFields + speedFields + environmentFields
for field in allFields:
    plotField(field, field + 'training/BoxPlots.png')
```

```python
# Plot qualifying data
allFields = ['Qualifying' + field for field in allFields]
for field in allFields:
    plotField(field, field + 'qualifying/BoxPlots.png')
```

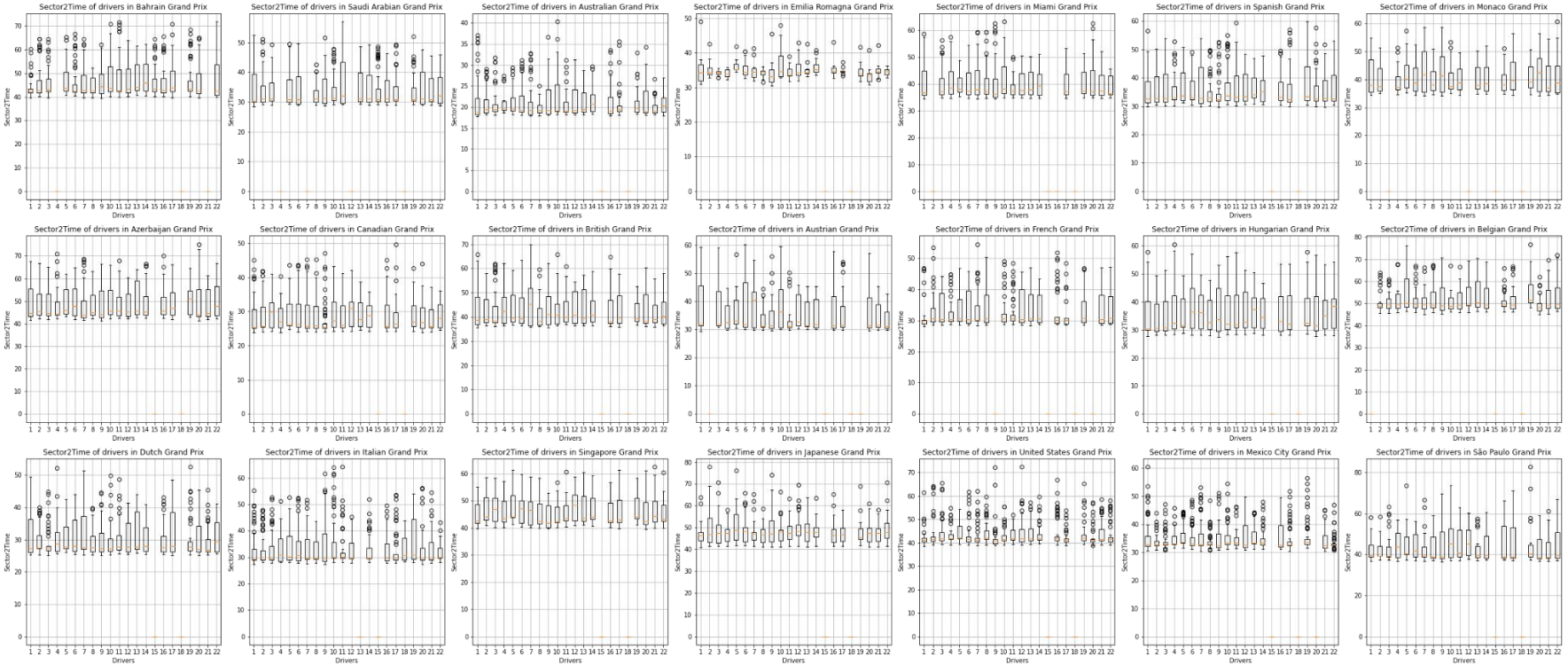
Box Plots - Lap Time



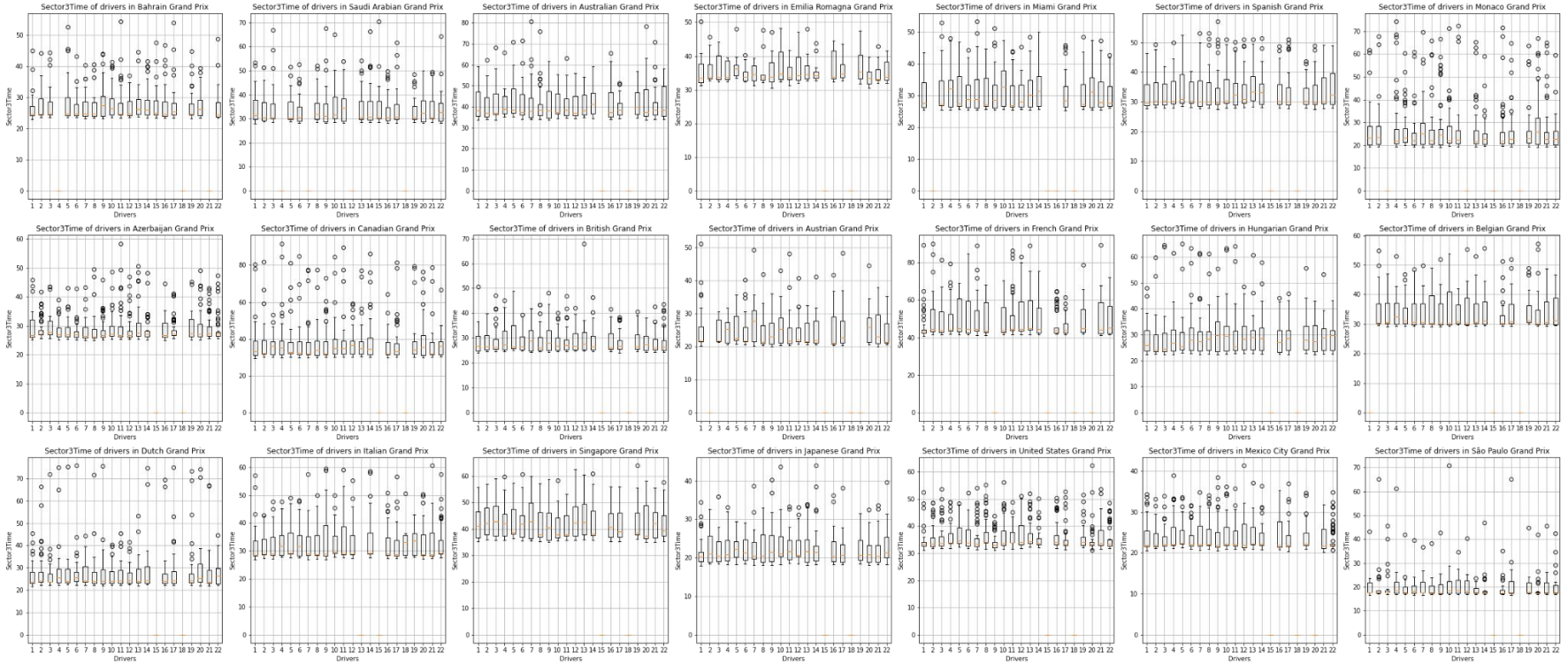
Box Plots - Sector 1 Time



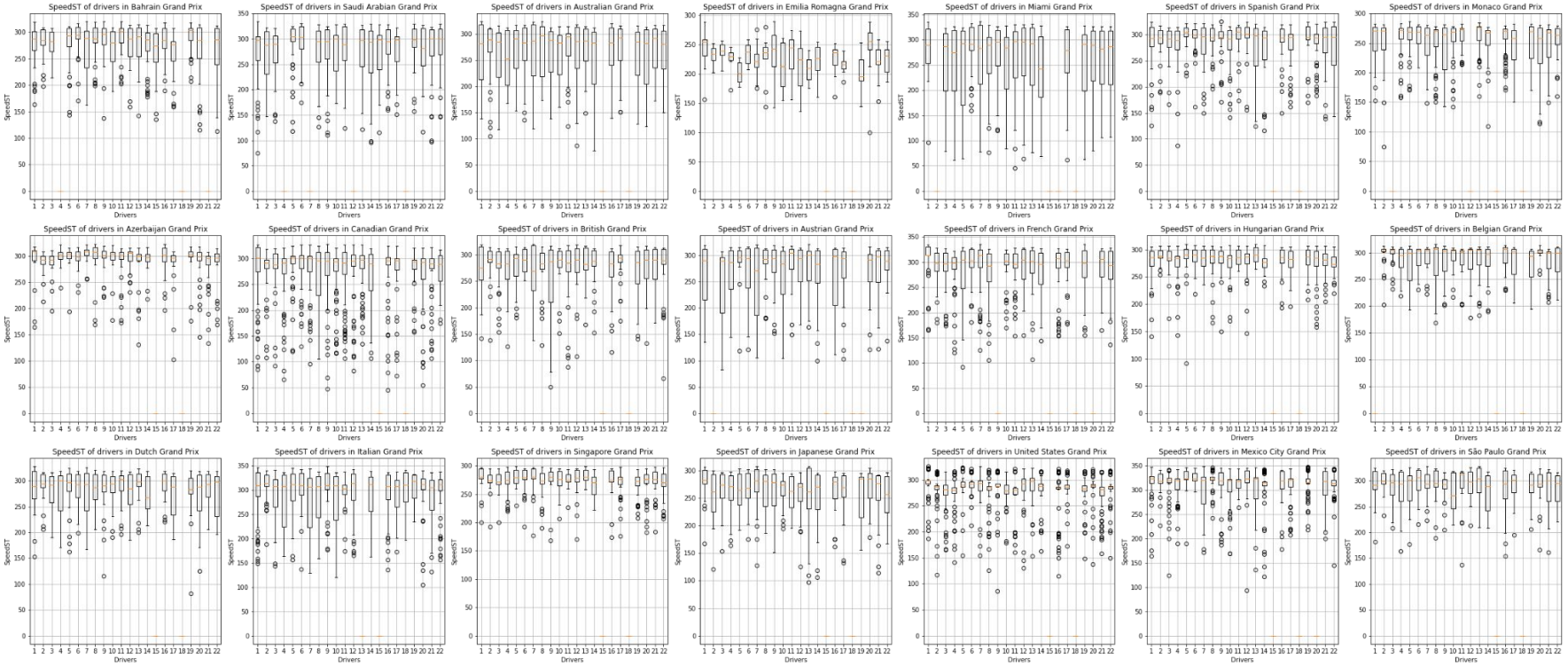
Box Plots - Sector 2 Time



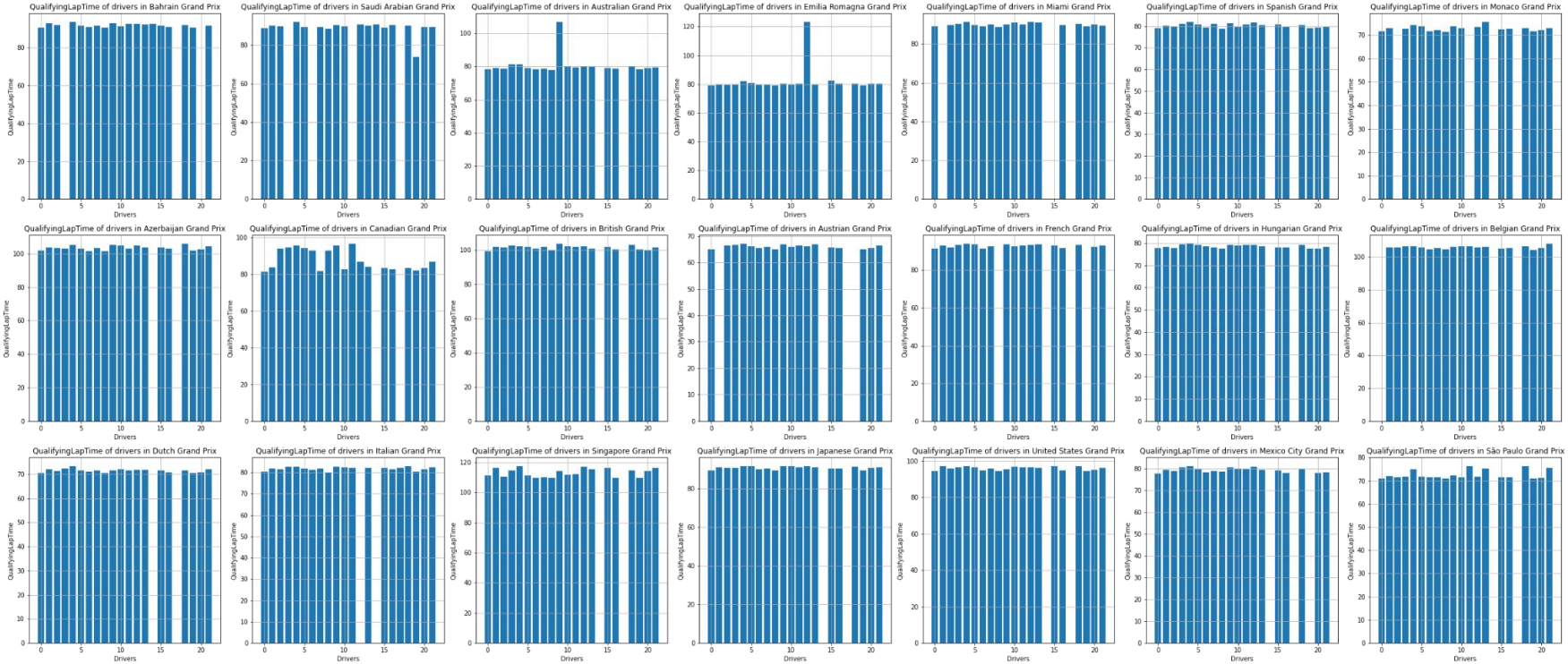
Box Plots - Sector 3 Time



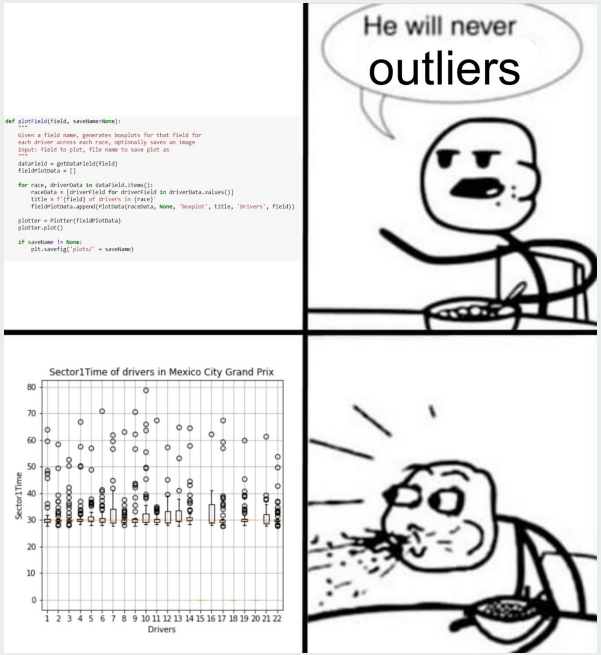
Box Plots - Speed Trap



Box Plots - Qualifying Lap Time



Observations



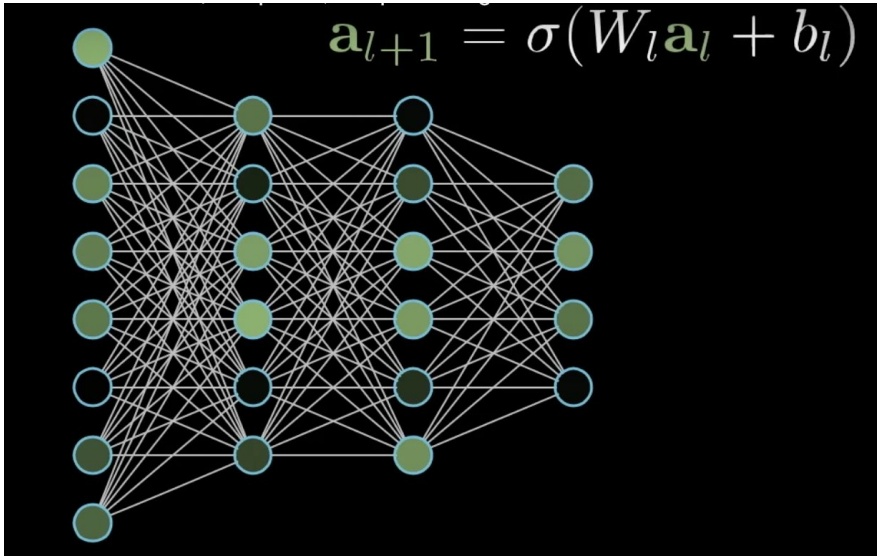
We made the following observations:

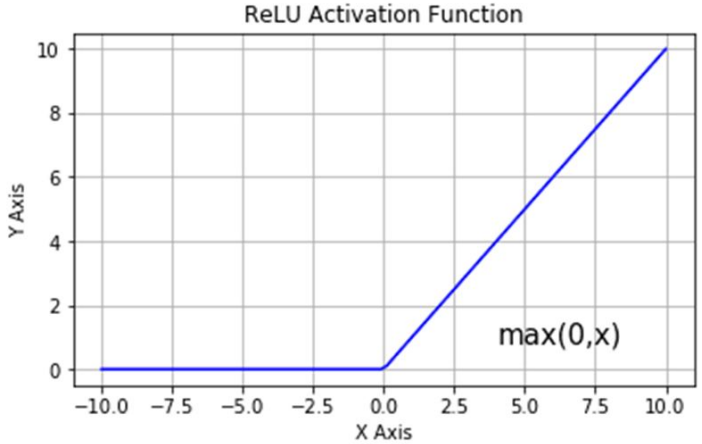
1. There were many more outliers than originally expected.
2. Outliers seemed to be prevalent on a per-race basis. Either all drivers had consistent times, or all drivers were scattered.
3. Sector times were more scattered than the lap times.

Neural Networks

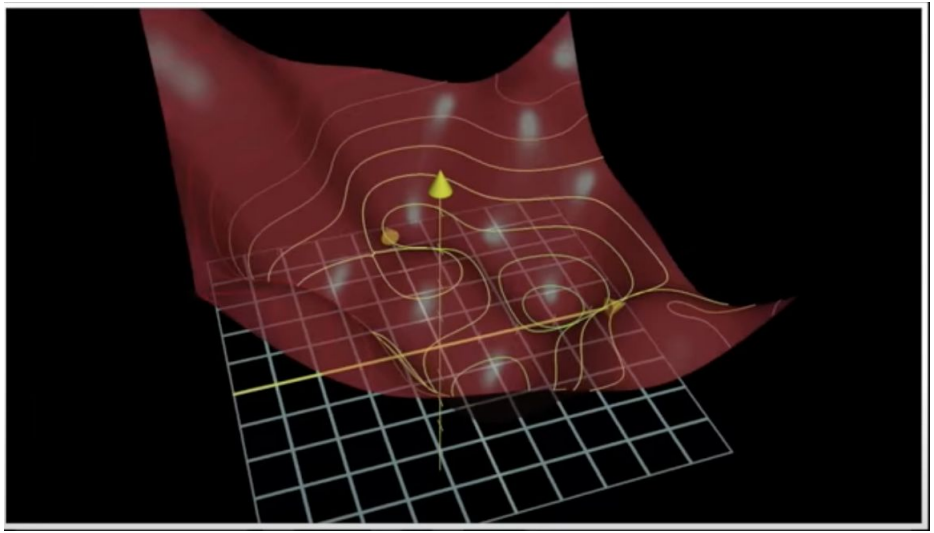
Neural Networks 101

Source: https://www.3blue1brown.com/topics/neural-networks

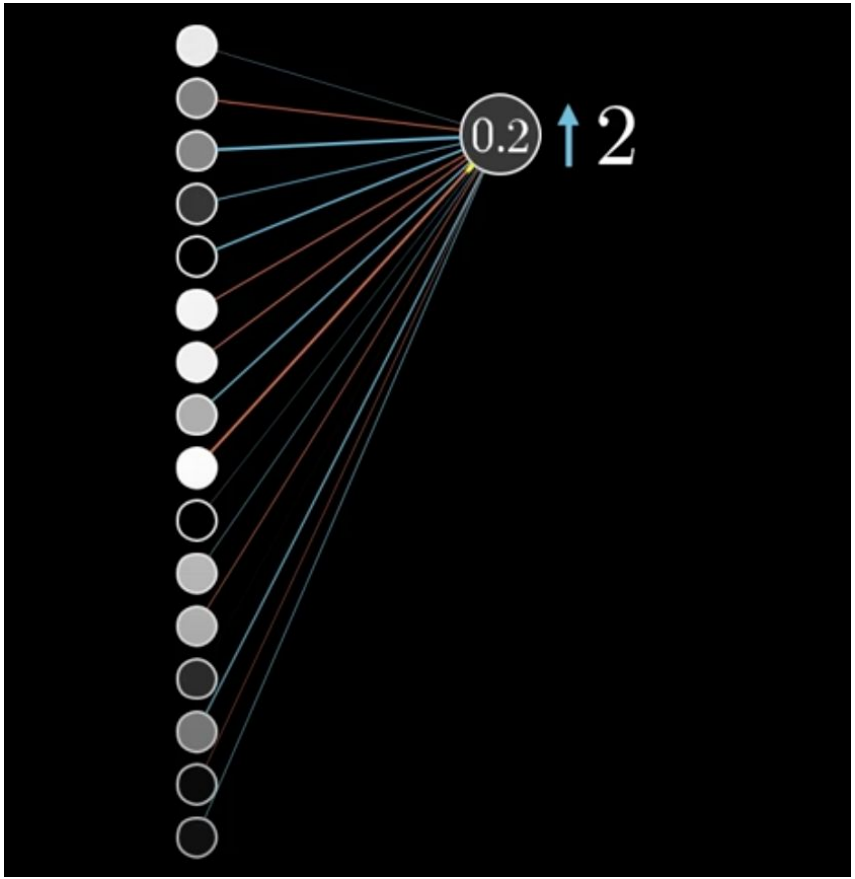




Gradient Descent



Backpropagation



Preparing Data for Neural Network

```
nn_copy = masterList.copy()
nn_copy = pd.get_dummies(nn_copy, columns = ['Weekend', 'Driver', 'Compound', 'QualifyingCompound'])

#Adjust Boolean Data
nn_copy['New'] = pd.to_numeric(nn_copy['New'], errors = 'coerce')

nn_copy = nn_copy.dropna()
```

Making Training, Validation, and Test Sets

```
#Set training variables(X) and prediction variables (Y)
x = nn_copy >> drop(X.QualifyingLapTime)
y = nn_copy['QualifyingLapTime']

#Normalize Training Data
x_scaler = preprocessing.StandardScaler().fit(x)
x_norm = x_scaler.transform(x)

#Split Data Into Test and Training Sets
x_train, x_test, y_train, y_test = train_test_split(x_norm, y, test_size = 0.2)

x_train = np.asarray(x_train).astype(float)
y_train = np.asarray(y_train).astype(float)
x_test = np.asarray(x_test).astype(float)
y_test = np.asarray(y_test).astype(float)

#Split Test Data Into Validation and Test Sets
x_validation, x_test, y_validation, y_test = train_test_split(x_test, y_test, test_size = 0.5)
```

Sources:
https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5
https://www.tensorflow.org/tutorials/quickstart/beginner

First Neural Network

```
#One layer is generally enough for simple problems like this (As opposed to something like digit recognition)
#Hidden layer size is generally between the input size (number of x variables) and the output size (number of y variables)
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, activation = 'relu'),
])

#Adam is the recommended default optimizer and does not have much competition
#Mean squared error loss function as seen in class
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
#Start Batch Size Small (Powers of 2 for efficient GPU usage)
#Epochs rule of thumb is to start with triple the amount of columns
#Verbose determines how it prints the training progress
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 2712.7739 - mean_squared_error: 2712.7739 - mean_absolute_error: 33.5383 - 115ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, activation = 'relu'),
    tf.keras.layers.Dense(16, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 0.3118 - mean_squared_error: 0.3118 - mean_absolute_error: 0.3657 - 120ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, activation = 'relu'),
    tf.keras.layers.Dense(16, activation = 'relu'),
    tf.keras.layers.Dense(16, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 478.9241 - mean_squared_error: 478.9241 - mean_absolute_error: 5.7160 - 124ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 2674.6145 - mean_squared_error: 2674.6145 - mean_absolute_error: 32.6951 - 116ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 0.0819 - mean_squared_error: 0.0819 - mean_absolute_error: 0.1483 - 142ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

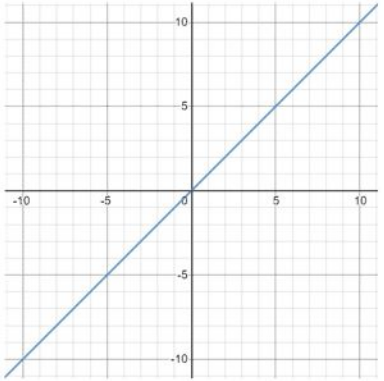
```
44/44 - 0s - loss: 479.0946 - mean_squared_error: 479.0946 - mean_absolute_error: 5.5963 - 122ms/epoch - 3ms/step
```

What If We Adjust the Architecture?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(16, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 0.1189 - mean_squared_error: 0.1189 - mean_absolute_error: 0.1349 - 127ms/epoch - 3ms/step
```

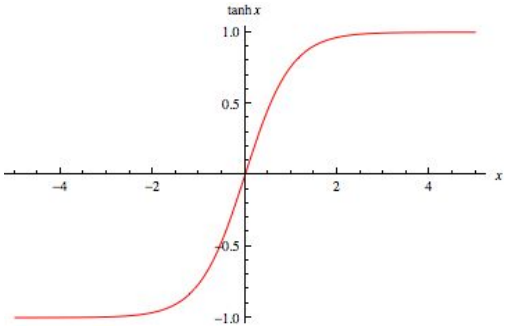
What About Different Activation Functions?




```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'linear'),
    tf.keras.layers.Dense(32, activation = 'linear'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 2.1090 - mean_squared_error: 2.1090 - mean_absolute_error: 0.8378 - 146ms/epoch - 3ms/step
```

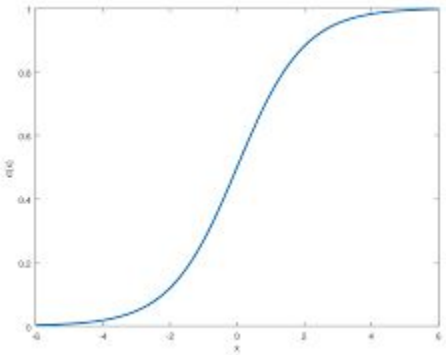
What About Different Activation Functions?




```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'tanh'),
    tf.keras.layers.Dense(32, activation = 'tanh'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 7488.8525 - mean_squared_error: 7488.8525 - mean_absolute_error: 85.7438 - 135ms/epoch - 3ms/step

[7488.8525390625, 7488.8525390625, 85.74382019042969]
```

What About Different Activation Functions?



```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'sigmoid'),
    tf.keras.layers.Dense(32, activation = 'sigmoid'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 7486.9248 - mean_squared_error: 7486.9248 - mean_absolute_error: 85.7323 - 122ms/epoch - 3ms/step
```

What If We Change the Batch Size?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 16, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 0.9071 - mean_squared_error: 0.9071 - mean_absolute_error: 0.1575 - 130ms/epoch - 3ms/step
```

What If We Add More Training Epochs?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 1000, verbose = 2)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
347/347 - 0s - loss: 0.0052 - mean_squared_error: 0.0052 - mean_absolute_error: 0.0520 - 250ms/epoch - 719us/ste
Epoch 996/1000
347/347 - 0s - loss: 0.0087 - mean_squared_error: 0.0087 - mean_absolute_error: 0.0670 - 250ms/epoch - 719us/ste
Epoch 997/1000
347/347 - 0s - loss: 0.0092 - mean_squared_error: 0.0092 - mean_absolute_error: 0.0699 - 241ms/epoch - 695us/ste
Epoch 998/1000
347/347 - 0s - loss: 0.0168 - mean_squared_error: 0.0168 - mean_absolute_error: 0.0881 - 248ms/epoch - 716us/ste
Epoch 999/1000
347/347 - 0s - loss: 0.0092 - mean_squared_error: 0.0092 - mean_absolute_error: 0.0672 - 249ms/epoch - 719us/ste
Epoch 1000/1000
347/347 - 0s - loss: 0.0033 - mean_squared_error: 0.0033 - mean_absolute_error: 0.0413 - 246ms/epoch - 710us/ste
44/44 - 0s - loss: 0.2263 - mean_squared_error: 0.2263 - mean_absolute_error: 0.0550 - 159ms/epoch - 4ms/step
```

How Does Our Model Generalize to Other Data?

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                         tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_test, y_test, verbose = 2)
```

```
44/44 - 0s - loss: 0.0525 - mean_squared_error: 0.0525 - mean_absolute_error: 0.1576 - 117ms/epoch - 3ms/step
```

Any Improvements?

- Find some way to include car data; fastf1 includes ways to scrape car data for each millisecond, i.e. throttle input, engine RPM, DRS effects, etc.
- 2022 is a unique year: because car specs were overhauled from 2021, data from earlier years are not applicable.
    - That being said, the next big overhaul is scheduled for 2026, so data from this year onward will be applicable for the near future
- We should remove outliers (esp. Practice Lap Times), but the neural network worked pretty good with them in there.

References

FastF1 Package: https://theoehrly.github.io/Fast-F1/index.html

dfply Package: https://github.com/kieferk/dfply

Using Tensorflow:
https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5
https://www.tensorflow.org/tutorials/quickstart/beginner

Github with the code: https://github.com/alexsanderjones/F1Project