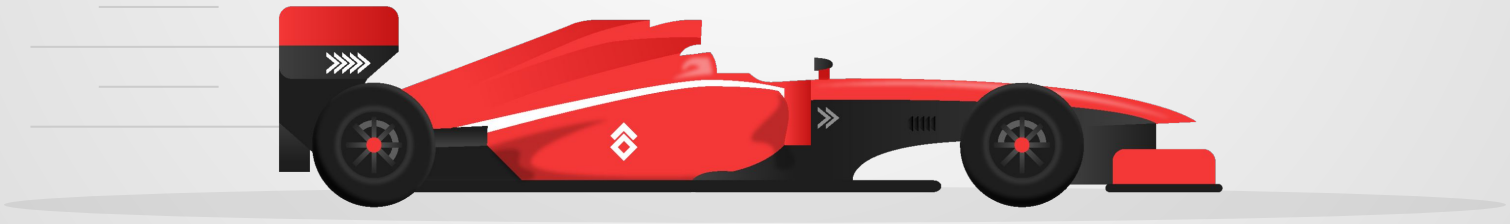
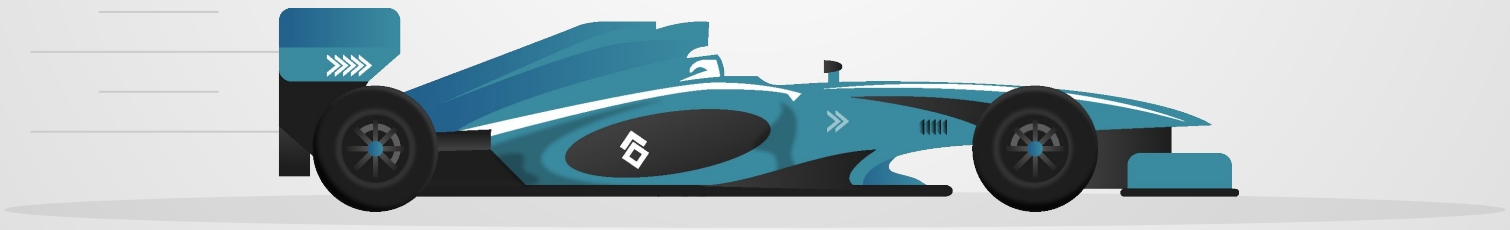
# Formula 1 Machine Learning Project

Stephan Karas

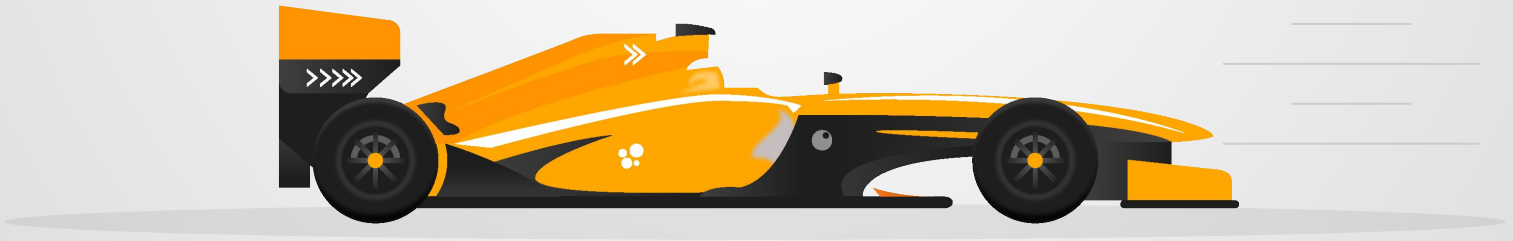


# TABLE OF CONTENTS

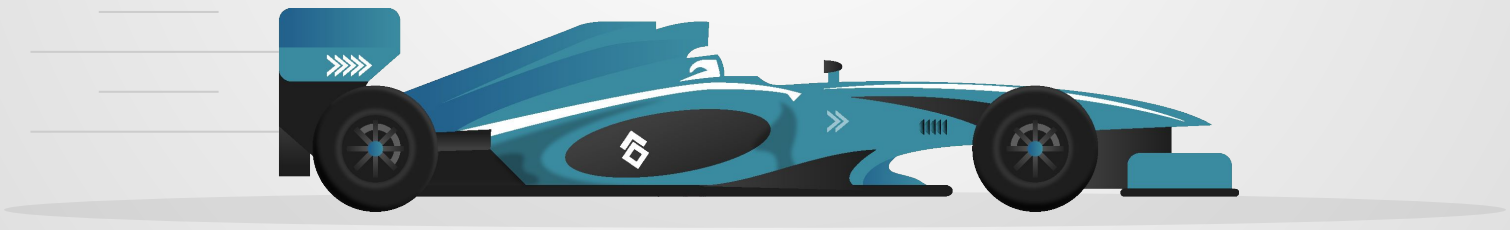
# 01 Problem Statement



The primary objective is to predict the finishing position of a Formula One driver in a race. We propose developing a predictive model incorporating time-sensitive qualifying data and weather data for short-term forecasts



# 02 Data Overview



# Data Sources

## Ergast Database

developed by Chris Newell. It is a free service that provides a historical record of Formula One (F1) racing results and statistics.

## Visual Crossing

The Visual Crossing Weather API provides historical, current, and forecast weather data for any location worldwide, enabling detailed weather insights for various applications.

# Dataset

To achieve this objective, we will use a comprehensive dataset that includes:

- **Driver Attributes:** Information such as age, nationality, career length, and average points per race.

- **Constructor Attributes:** Team references, nationality, and historical performance.

- **Race and Circuit Information:** Details like race year, round, and circuit specifics.

- **Weather Conditions:** Expected weather conditions on race day, sourced from the Visual Crossing API.

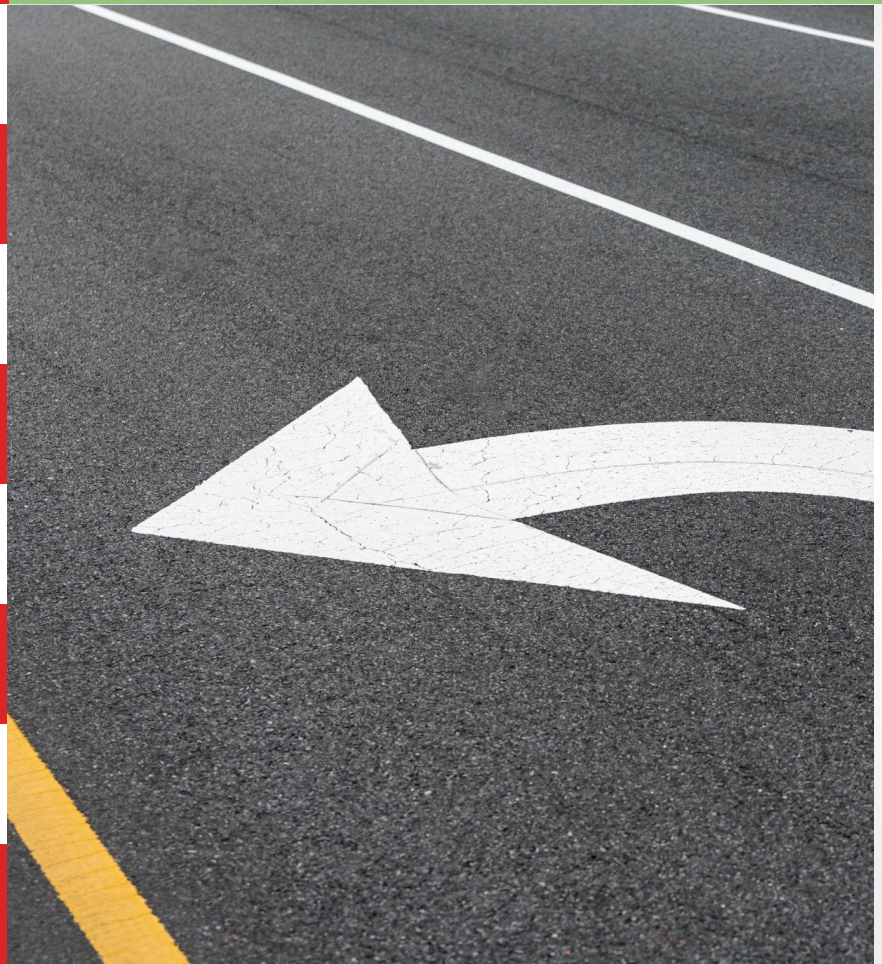
- **Circuit-Specific Historical Performance:** Past performance data of drivers on specific circuits.

# Dataset

Each row represents a driver's participation in a specific race.

| | raceId | race_year | race_round | race_circuitId | circuitId | circuit_alt | tempmax | tempmin | humidity | precip | ... | circuit_country_Turkey | circuit_countr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2009 | 1 | 1 | 1 | 10 | 76.7 | 47.1 | 50.7 | 0.000 | ... | 0 | |
| 1 | 2 | 2009 | 2 | 2 | 2 | 18 | 89.7 | 75.5 | 84.8 | 1.075 | ... | 0 | |
| 2 | 3 | 2009 | 3 | 17 | 17 | 5 | 71.4 | 60.9 | 87.3 | 0.370 | ... | 0 | |
| 3 | 5 | 2009 | 5 | 4 | 4 | 109 | 68.2 | 59.4 | 78.7 | 0.009 | ... | 0 | |
| 4 | 6 | 2009 | 6 | 6 | 6 | 7 | 75.8 | 62.3 | 77.6 | 0.000 | ... | 0 | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| 9043 | 939 | 2015 | 13 | 15 | 15 | 18 | 93.5 | 57.5 | 40.9 | 0.000 | ... | 0 | |
| 9044 | 940 | 2015 | 14 | 22 | 22 | 45 | 86.7 | 66.0 | 73.7 | 0.315 | ... | 0 | |
| 9045 | 944 | 2015 | 18 | 18 | 18 | 785 | 83.1 | 65.3 | 75.1 | 0.197 | ... | 0 | |
| 9046 | 943 | 2015 | 17 | 32 | 32 | 2227 | 75.0 | 48.3 | 73.3 | 0.008 | ... | 0 | |
| 9047 | 942 | 2015 | 16 | 69 | 69 | 161 | 63.2 | 57.8 | 89.1 | 0.688 | ... | 0 | |

9048 rows × 90 columns



# Target Variable

The target variable for our model is the driver's finishing position, categorized as 'driver_race_result_category', with the following classifications:

- 0: Finished outside the top 10.
- 1: Finished in the top 10 but outside the top 5.
- 2: Finished in the top 5 but did not win.
- 3: Won the race.

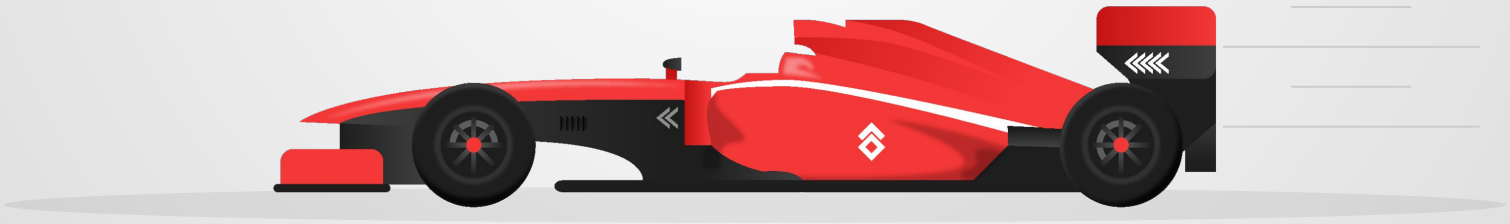
# Challenges

- Integrating multiple data sources, including historical race data from the Ergast API and weather data from Visual Crossing Weather API.

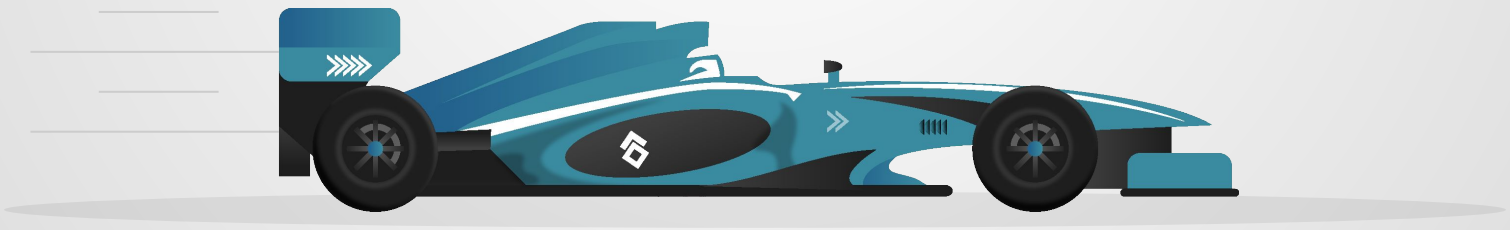
- Ensuring data consistency and synchronization between different sources.

- Handling potential class imbalance in race outcome categories.

- Effectively incorporating real-time data (qualifying results) into the model.

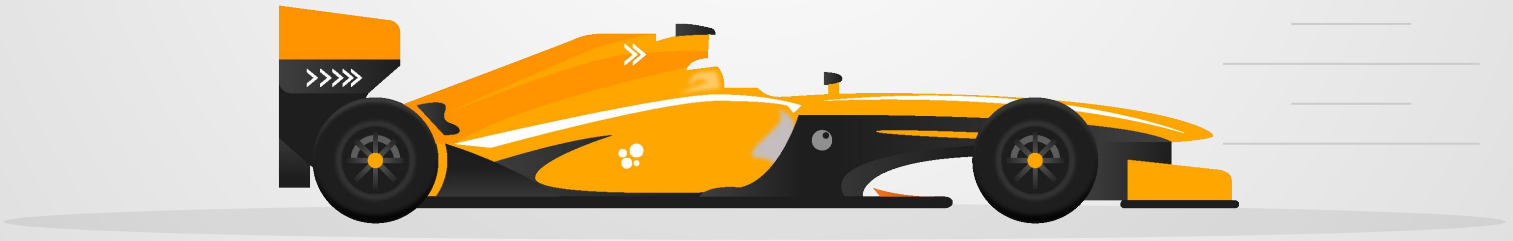


# 03 Data Preprocessing



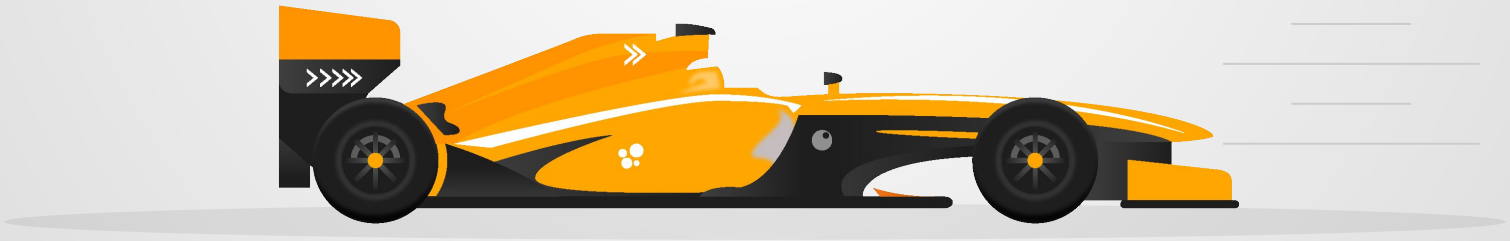
# DATA PRELOADING

```python
import pandas as pd

drivers = pd.read_csv('f1db_csv\drivers.csv')
constructors = pd.read_csv('f1db_csv\constructors.csv')
races = pd.read_csv(r'f1db_csv\races.csv')
results = pd.read_csv(r'f1db_csv\results.csv')
circuits = pd.read_csv('f1db_csv\circuits.csv')
qualifying = pd.read_csv('f1db_csv\qualifying.csv')
sprint = pd.read_csv('f1db_csv\sprint_results.csv')
```



# Feature Engineering

```
# Calculate historical points per race
results['points'] = results['points'].astype(float) # convert points to floaat

# Calculate the average points each driver has earned per race, grouped by their unique driver ID
avg_points = results.groupby('driverId')['points'].mean().reset_index()  # Group by driver and calculate mean points

# Merge the average points data back into the drivers DataFrame
# This adds the avg_points column to the drivers DataFrame where the driverId matches
drivers = drivers.merge(avg_points, on='driverId', how='left')
```

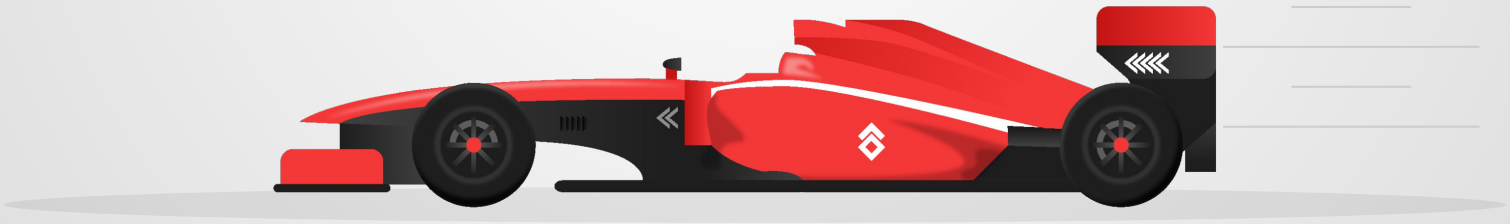


# Merging DB Tables

Merging the relevant database tables, bringing together our dataset

1. Renaming the columns of each dataset to include its name
2. Merging Races & Circuits table (to initially incorporate date and location data)

```
# Merge races and circuits

races_circuits = pd.merge(races, circuits, left_on='race_circuitId', right_on='circuitId')
```



# Incorporating Weather Data

- We used the Visual Crossing Weather API to include weather forecasts in our predictions.
- Relevant weather features included: tempmax, tempmin, humidity, precip, windspeed, and conditions
- Example code to fetch and integrate weather data:

```
weather_features = ['tempmax', 'tempmin', 'humidity', 'precip',
'windspeed', 'conditions']
for feature in weather_features:
    races_circuits[feature] = None

races_circuits['race_date'] =
pd.to_datetime(races_circuits['race_date']).dt.strftime('%Y-%m-%d'
)

for index, row in races_circuits.iterrows():
    weather_data =
get_relevant_weather_features(str(row['circuit_location']),
str(row['race_date']), "YOUR_API_KEY")
    if weather_data:
        for feature in weather_features:
            races_circuits.at[index, feature] = weather_data[feature]
            print("Collected data for", str(row['circuit_location']), ",",
str(row['race_date']))
```

```
import requests

def get_relevant_weather_features(location, date, api_key):
    api_url = f"https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timeline/{location}/{date}"
    params = {
        'unitGroup': 'us',
        'key': api_key,
        'include': 'obs'
    }
    response = requests.get(api_url, params=params)
    if response.status_code == 200:
        data = response.json()
        if 'days' in data and len(data['days']) > 0:
            weather_day = data['days'][0]
            relevant_weather_features = {
                'tempmax': weather_day.get('tempmax'),
                'tempmin': weather_day.get('tempmin'),
                'humidity': weather_day.get('humidity'),
                'precip': weather_day.get('precip'),
                'windspeed': weather_day.get('windspeed'),
                'conditions': weather_day.get('conditions')
            }
            return relevant_weather_features
    else:
        print(f"Failed to retrieve data for {location} on {date}: {response.status_code}")
        return None
```

# Merging DB Tables

Merging the relevant database tables to create a comprehensive dataset

1. Merging Results with Circuits and Races.

```
races_circuits_results = pd.merge(races_circuits, results, on='raceId')
```

2. Creating the target variable, categorizing the results value

```
def classify_position(pos):
    if pos == 1:
        return 3  # Winner
    elif pos <= 5:
        return 2  # Top 5
    elif pos <= 10:
        return 1  # Top 10
    else:
        return 0  # Outside Top 10

# Assuming 'result_positionOrder' is the column indicating the final race position
races_circuits_results['driver_race_result_category'] = races_circuits_results['result_positionOrder'].apply(classify_position)
```

# Merging DB Tables

Merging the relevant database tables to create a comprehensive dataset

3. Merging the Drivers and Constructors, and Qualification tables

```
df_quali = pd.merge(df_cleaned, qualifying, on=['raceId', 'driverId'])

# Drop columns that arent relevant
df_cleaned = df_quali.drop(columns=[
    'race_url', 'circuit_circuitRef', 'circuit_url', 'result_positionText',
    'driver_forename', 'driver_surname', 'driver_number', 'driver_driverRef',
    'driver_code', 'driver_age', 'driver_url', 'constructor_constructorRef',
    'constructor_url', 'race_fp1_date', 'race_fp1_time', 'race_fp2_date',
    'race_fp2_time', 'race_fp3_date', 'race_fp3_time', 'race_quali_date',
    'race_quali_time', 'race_sprint_date', 'race_sprint_time', 'Unnamed: 0', 'quali_qualifyId',
    'constructorId_y', 'quali_number', 'quali_position', 'constructorId_x'
])
```

# Cleaning weather data

- Removed rows with NA weather data (older races) to ensure data quality
  - Combined driver's forename and surname for a unique identifier.
    - Example code:

```
df_filtered_weather = races_circuits.dropna(subset=['tempmax', 'tempmin',
'humidity', 'precip', 'windspeed', 'conditions'])
df_filtered_weather['fullname'] = df_filtered_weather['driver_forename'] + '_' +
df_filtered_weather['driver_surname']
df_filtered_weather.to_csv('data_preview/filtered_weather_data.csv')
```

# Removing Irrelevant Features

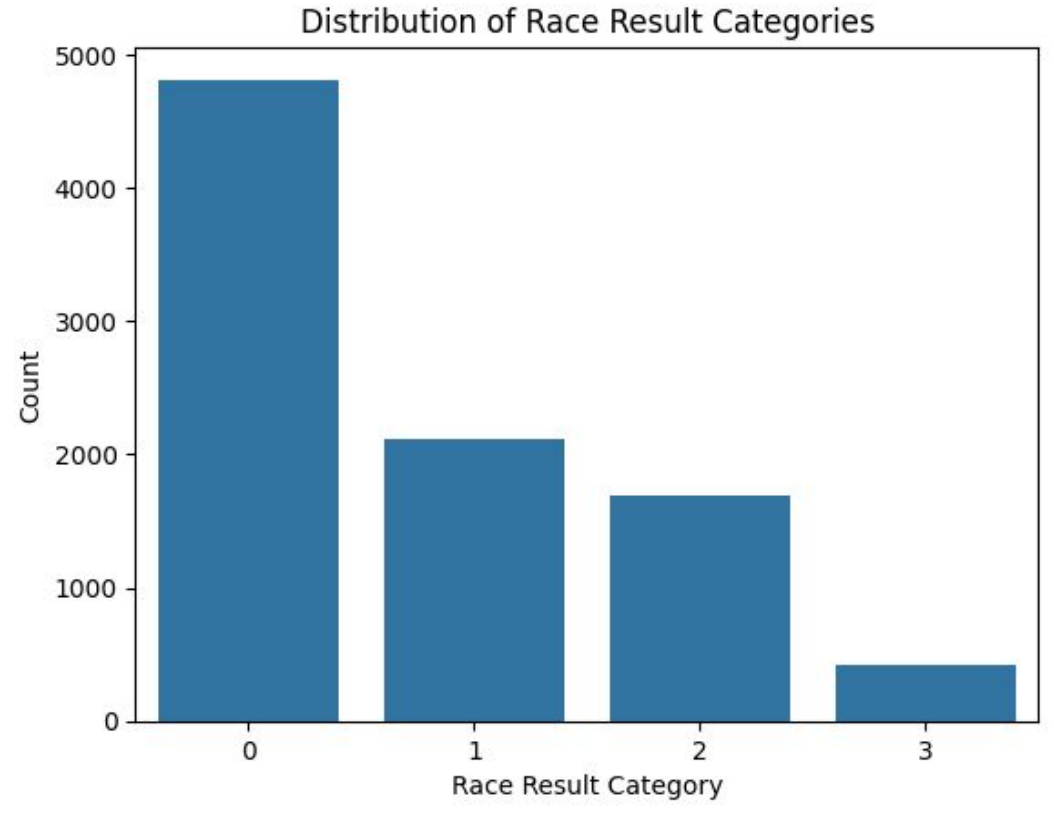
- Dropped columns that are not relevant to the prediction task to reduce dimensionality and improve model performance.
  - Example code:

```
df_cleaned = pd.read_csv('data_preview/filtered_weather_data.csv')
df_quali = pd.merge(df_cleaned, qualifying, on=['raceId', 'driverId'])

df_cleaned = df_quali.drop(columns=[
    'race_url', 'circuit_circuitRef', 'circuit_url', 'result_positionText',
    'driver_forename', 'driver_surname', 'driver_number', 'driver_driverRef',
    'driver_code', 'driver_age', 'driver_url', 'constructor_constructorRef',
    'constructor_url', 'race_fp1_date', 'race_fp1_time', 'race_fp2_date',
    'race_fp2_time', 'race_fp3_date', 'race_fp3_time', 'race_quali_date',
    'race_quali_time', 'race_sprint_date', 'race_sprint_time', 'Unnamed: 0', 'quali_qualifyId',
    'constructorId_y', 'quali_number', 'quali_position', 'constructorId_x'
])
df_cleaned.to_csv('data_quali_preview/cleaned_data_quali.csv')
```

# Converting Qualifying Timings

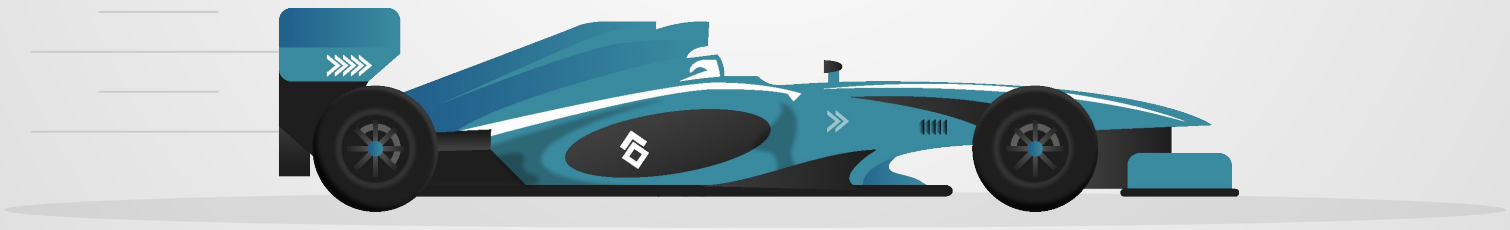
- Converted qualifying timings from string format to numerical seconds to facilitate model training
- Filled NaN values in qualifying columns with a large time value to handle missing data.

```
def time_to_seconds(time_str):
    """Convert time string in format 'm:ss.mmm' to seconds."""
    if pd.isnull(time_str):
        return np.nan
    mins, secs = time_str.split(':')
    return int(mins) * 60 + float(secs)

# Convert time strings to numerical format
all_quali_columns = ['quali_q1', 'quali_q2', 'quali_q3']  # Example column names
for col in all_quali_columns:
    df_na_cleaned[col] = df_na_clean
```




- majority of drivers finished outside the top 10 (category 0)
- a smaller number finishing in the top 10 but not the top 5 (category 1)
- fewer still in the top 5 but not winning (category 2)
- the least number of drivers won the race (category 3).

# 04 Model Choice



# RandomForest + Parameter Grid Tuning

- We chose the Random Forest algorithm due to its ability to handle large datasets and capture complex interactions between features.
- Initial parameter selection based on common starting values and domain knowledge.
- Created a parameter grid for hyperparameter tuning.

```
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None],
    'bootstrap': [True, False]
}
```

# How RandomForest Works for us

**Data**: We have a comprehensive dataset that includes our relevant features.

**Training**: Creates multiple decision trees, each trained on a different random subset of our dataset.

- allows each tree to learn from different parts of the data, such as specific race days, varying weather conditions, or different driver performance statistics.

**Prediction**: Each decision tree independently analyzes its subset of data and makes a prediction about the driver's finishing position.

- I.e. one tree might consider how the driver performed in similar weather conditions, while another looks at qualifying results.

**Final Decision**: The Random Forest aggregates the predictions from all the trees. By using a majority vote system, it determines the most likely finishing position for the driver.

# Parameters choice

- Choosing the right parameters is crucial for the performance of our Random Forest model.

n_estimators: specifies the number of trees in the forest.

max_depth: limits the number of nodes in the tree.

min_samples_split: sets minimum number of samples required in a node to split into two new nodes.

min_samples_leaf: min no. of samples required to be at a leaf node.

max_features: no. of features to consider when looking for the best split.

bootstrap: boolean to use bootstrap samples when building trees. (Drawing random samples with replacement from the original dataset.)

# Ensuring Fair Train-Test Data Split

- We used the StratifiedShuffleSplit technique to split our dataset. This method ensures that both training and test sets maintain the same class distribution as the original dataset.
    - stratifying the data on driver_race_result_category, we preserve the proportion of each race result category in both sets
    - provides a balanced and representative sample for training and evaluation.

- Applied SMOTE  (Synthetic Minority Oversampling Technique) to the training set.
    - generating synthetic samples for the minority classes, enhancing their representation
    - improving the model's ability to learn from the underrepresented categories.
- set class_weight='balanced' to ensure the model pays more attention to minority classes.

# RandomizedSearchCV Hyperparameter tuning

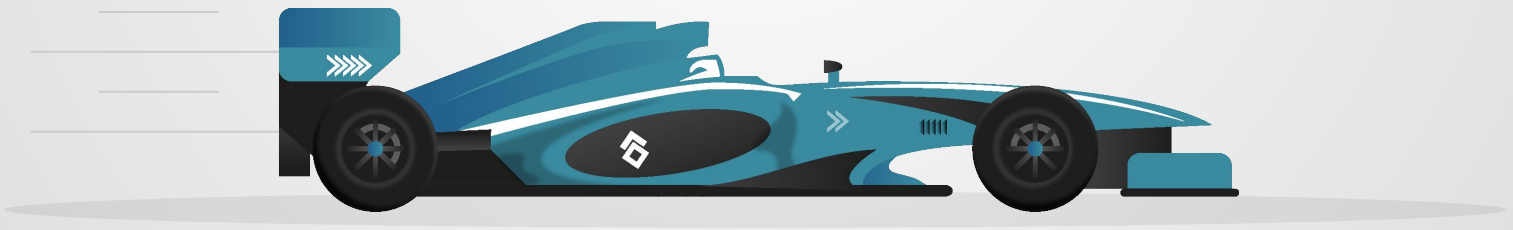
- RandomizedSearchCV works by randomly sampling a fixed number of hyperparameter combinations from a specified range and evaluating each combination using cross-validation, then selecting the best combination based on performance.

- It uses K-fold cross-validation to evaluate the performance of each parameter combination.

# Baseline Classifier: KNN

- KNN classifies a data point by looking at the k closest data points (neighbors) in the dataset and assigning it to the most common class among those neighbors.
- Same Data preparation as RandomForest:
  - Grouped Train-Test Split (20% test size)
  - Standardized features with StandardScaler
- KNN (K-Nearest Neightbors):
  - Optimized k using RandomizedSearchCV
  - Tested odd values of k (1-29) to avoid ties (even k can result in equal votes from neighbors)
  - In KNN, a tie occurs when an equal number of neighbors vote for different classes.
- Results:
  - Best k: 27

# 05 Model Evaluation



# Classification Report

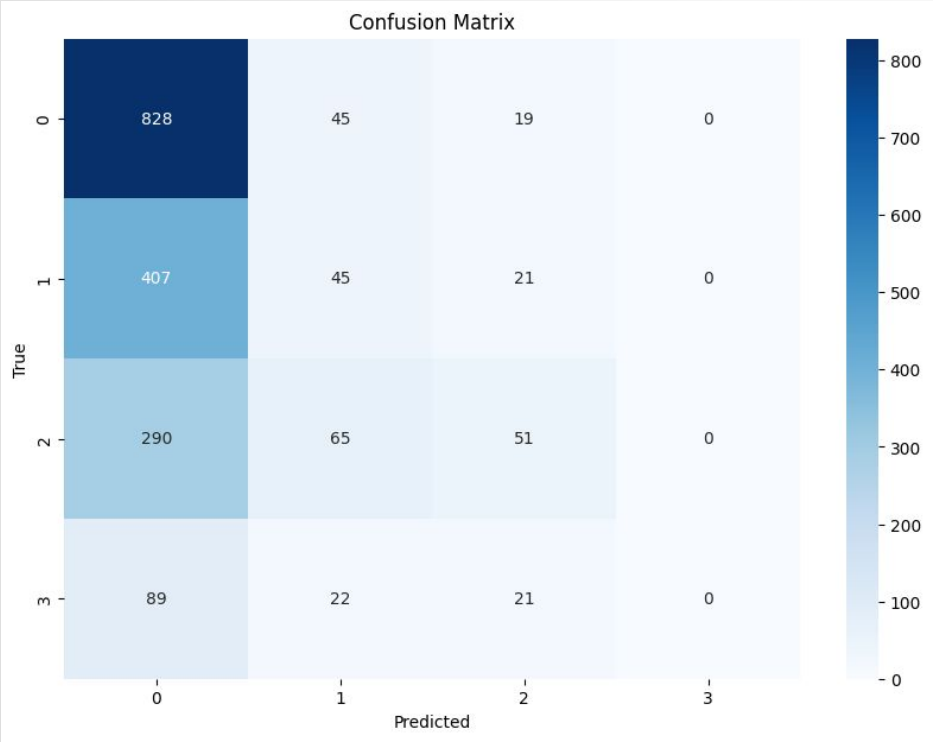
```
Best parameters: {'n_estimators': 400, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': 100, 'bootstrap': False}
```

Accuracy on test set: 0.5994475138121547

Classification Report on test set:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.72 | 0.76 | 0.74 | 963 |
| 1 | 0.40 | 0.30 | 0.35 | 423 |
| 2 | 0.48 | 0.55 | 0.52 | 339 |
| 3 | 0.44 | 0.44 | 0.44 | 85 |
| accuracy | | | 0.60 | 1810 |
| macro avg | 0.51 | 0.51 | 0.51 | 1810 |
| weighted avg | 0.59 | 0.60 | 0.59 | 1810 |

# Confusion Matrix



# Baseline Classifier: KNN

Best parameters: {'n_neighbors': 27}

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.51 | 0.93 | 0.66 | 892 |
| 1 | 0.25 | 0.10 | 0.14 | 473 |
| 2 | 0.46 | 0.13 | 0.20 | 406 |
| 3 | 0.00 | 0.00 | 0.00 | 132 |
| accuracy | | | 0.49 | 1903 |
| macro avg | 0.31 | 0.29 | 0.25 | 1903 |
| weighted avg | 0.40 | 0.49 | 0.39 | 1903 |

# Thank You!