Advance Machine Learning

CIS - 550

Project Summary

Predicting Formula 1 Driver Performance
A Data-Driven Approach to Performance Insights

GROUP – 14

1. Krishna Sai Chintoju (07) - 02134389
2. Chandu Pandi (30) – 02118307

Table of Contents

1. Problem Statement

The objective of this project is to predict Formula 1 drivers' race points, a critical indicator of performance and contribution to team standings, using historical data. This insight aids in assessing consistency, skill, and the potential for podium finishes.

2. Introduction

Predicting driver points is crucial in assessing a driver's performance and their contribution to the overall team standing in a Formula 1 season. Points directly reflect the consistency, skill, and competitiveness of a driver across races, which are key indicators of their ability to win championships or achieve podium finishes. By accurately forecasting points, teams and analysts can strategize better, optimize resource allocation, and gain insights into performance trends, which are valuable for both mid-season decisions and long-term planning.

3. Data Collection

Source: The Ergast Motor Racing Data API provided comprehensive historical F1 data.

Link: https://ergast.com/mrd/db/

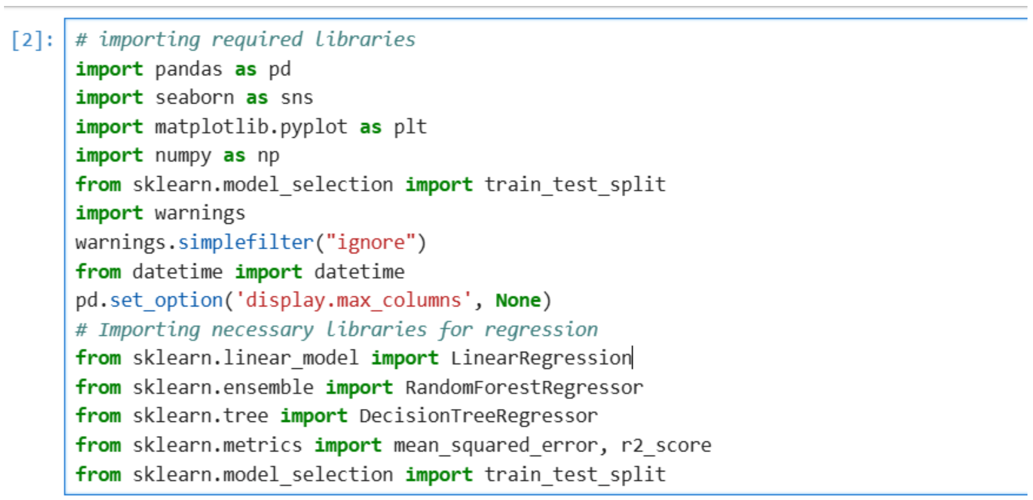
Datasets Used:

- **Results (results.csv)**:
  This dataset is the backbone of my analysis as it contains critical information about race outcomes, including positions, points, fastest laps, and status codes. These elements are directly tied to performance prediction.
- **Status (status.csv)**:
  Understanding a driver's status during the race (e.g., finished, retired, or disqualified) is key for identifying patterns in DNFs (Did Not Finish) and reliability issues, which are significant factors in race outcomes.
- **Drivers (drivers.csv)**:
  This dataset provides demographic and identity data, such as driver names, nationalities, and birthdates. These are essential for including driver-specific trends (e.g., experience, age) in my analysis.
- **Races (races.csv)**:
  Race-specific metadata, like the circuit, season, and race date, adds context to the analysis. This helps in understanding how different tracks and years influence performance.
- **Constructors (constructors.csv)**:
  Since Formula 1 is a team sport, constructor data helps capture the impact of team performance, technology, and reliability on race results.

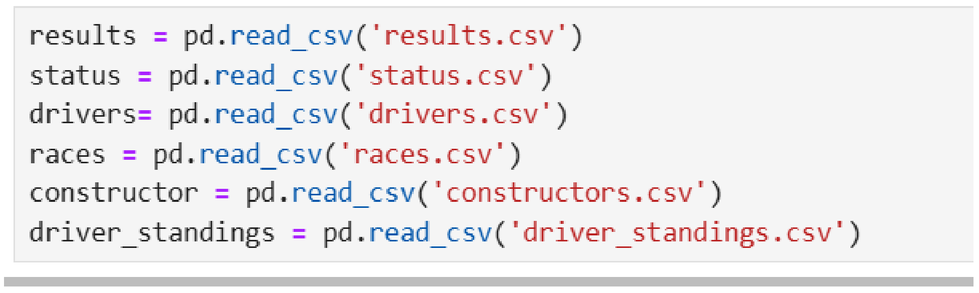
- **Driver Standings (driver_standings.csv)**:
  This dataset adds longitudinal insights into a driver's performance throughout the season, providing a benchmark for predictive modeling.

By choosing these datasets, we ensured that our analysis covers the most impactful aspects of race performance while keeping the project focused and computationally efficient. Other datasets, like lap_times, pit_stops, or qualifying, were excluded because they are too granular and outside the scope of our goal, which emphasizes high-level race outcomes rather than specific intra-race dynamics."

Importing the required libraries and datasets.



```
[2]: # importing required libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
import warnings
warnings.simplefilter("ignore")
from datetime import datetime
pd.set_option('display.max_columns', None)
# Importing necessary libraries for regression
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
```



```
results = pd.read_csv('results.csv')
status = pd.read_csv('status.csv')
drivers= pd.read_csv('drivers.csv')
races = pd.read_csv('races.csv')
constructor = pd.read_csv('constructors.csv')
driver_standings = pd.read_csv('driver_standings.csv')
```

4.Data Preprocessing

Data Integration

Using an **inner join** ensures that the final dataset only contains rows that have matching entries across all tables. This is important for maintaining data integrity and focusing on complete records.

1. **Ensures Data Completeness:** If a raceId exists in the races table but has no corresponding results in the results table, the inner join excludes it. Including it would result in incomplete or missing data in key columns like driver positions, times, or points.

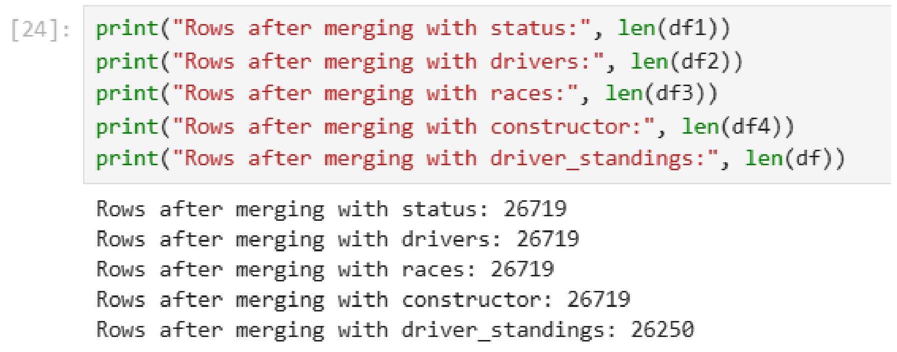
2. **Prevents Missing Values:** Outer joins (left, right, or full) could introduce NaN values in columns when there are unmatched rows. These NaN values would complicate preprocessing and analysis.
3. **Focus on Relevant Data:** For predictive modeling (e.g., race results), you need complete relationships between races, drivers, constructors, and standings. Inner joins help filter out irrelevant or incomplete records.
4. **Simplifies Analysis:** It avoids ambiguity in interpreting partially joined data, where some fields might be missing. This ensures the final dataset is clean and ready for analysis.

```python
# Merge results with status using 'statusId'
df1 = pd.merge(results, status, on='statusId', how='inner')

# Merge df1 with drivers using 'driverId'
df2 = pd.merge(df1, drivers, on='driverId', how='inner')

# Merge df2 with races using 'raceId'
df3 = pd.merge(df2, races, on='raceId', how='inner')

# Merge df3 with constructor using 'constructorId'
df4 = pd.merge(df3, constructor, on='constructorId', how='inner')

# Merge df4 with driver_standings using 'raceId' and 'driverId'
df = pd.merge(df4, driver_standings, on=['raceId', 'driverId'], how='inner')

# Display the resulting dataframe
print(df.shape)
df.head()
```

```
(26250, 53)
   resultId  raceId  driverId  constructorId  number_x  grid  position_x  positionText_x  positionOrder  points_x  laps      time_x  milliseconds  fastestLap  rank  fastestLapTime  fastestLapSpeed  statusId    status   driverRef  num
0         1      18         1              1        22     1           1               1              1      10.0    58  1:34:50.616       5690616          39     2        1:27.452          218.300         1  Finished    hamilton
1         2      18         2              2         3     5           2               2              2       8.0    58      +5.478       5696094          41     3        1:27.739          217.586         1  Finished    heidfeld
2         3      18         3              3         7     7           3               3              3       6.0    58      +8.163       5698779          41     5        1:28.090          216.719         1  Finished     rosberg
3         4      18         4              4         5    11           4               4              4       5.0    58     +17.181       5707797          58     7        1:28.603          215.464         1  Finished      alonso
4         5      18         5              1        23     3           5               5              5       4.0    58     +18.014       5708630          43     1        1:27.418          218.385         1  Finished  kovalainen
```

Verification

We will validate data integrity after the merging process by ensuring that the resulting df contains only raceId and driverId values that exist in the original races and driver's datasets.



```
[24]: print("Rows after merging with status:", len(df1))
      print("Rows after merging with drivers:", len(df2))
      print("Rows after merging with races:", len(df3))
      print("Rows after merging with constructor:", len(df4))
      print("Rows after merging with driver_standings:", len(df))

      Rows after merging with status: 26719
      Rows after merging with drivers: 26719
      Rows after merging with races: 26719
      Rows after merging with constructor: 26719
      Rows after merging with driver_standings: 26250
```

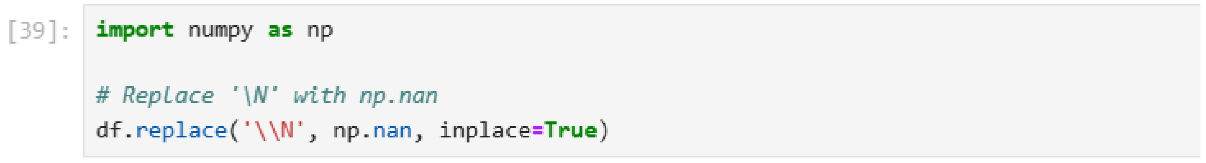
The decrease in rows after merging the datasets is due to the use of the inner join. This type of join keeps only the rows with matching values in all dataframes based on the specified keys (e.g., raceId, driverId). As a result, any rows that do not have corresponding matches in all the merged tables (e.g., missing race or driver data) are excluded from the final dataframe, leading to a reduction in the total number of rows.

Missing Values

Initially, the dataset contains no explicit missing values. Instead, missing or unavailable data is represented by the string \N. This placeholder will require preprocessing to identify and handle missing values appropriately during analysis.

Replacing the placeholder \N:

Replacing all occurrences of the string \N in the DataFrame df with np.nan (which is the standard representation for missing values in pandas and NumPy).



```
[39]: import numpy as np

      # Replace '\N' with np.nan
      df.replace('\\N', np.nan, inplace=True)
```

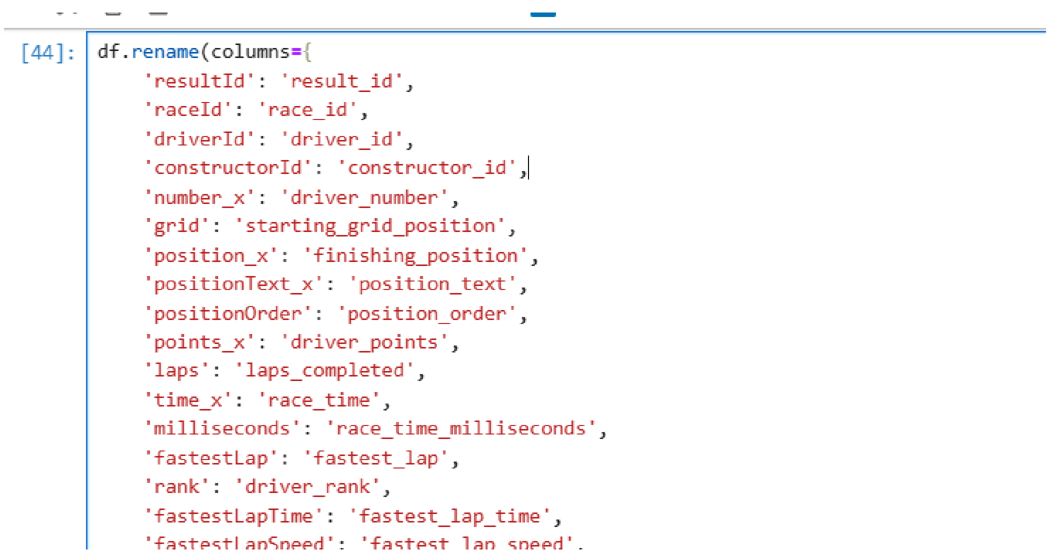
- replace('\\N', np.nan) identifies all occurrences of the string \N (escaped as \\N in Python).
- np.nan is used to standardize missing values.
- inplace=True modifies the DataFrame directly without needing reassignment.

After replacing the placeholder \N with NaN, the dataset now contains actual missing values, which are represented as NaN (Not a Number) in Python. These NaN values can be identified and handled using pandas' built-in methods such as .isnull(), .notnull(), .dropna(), or .fillna().

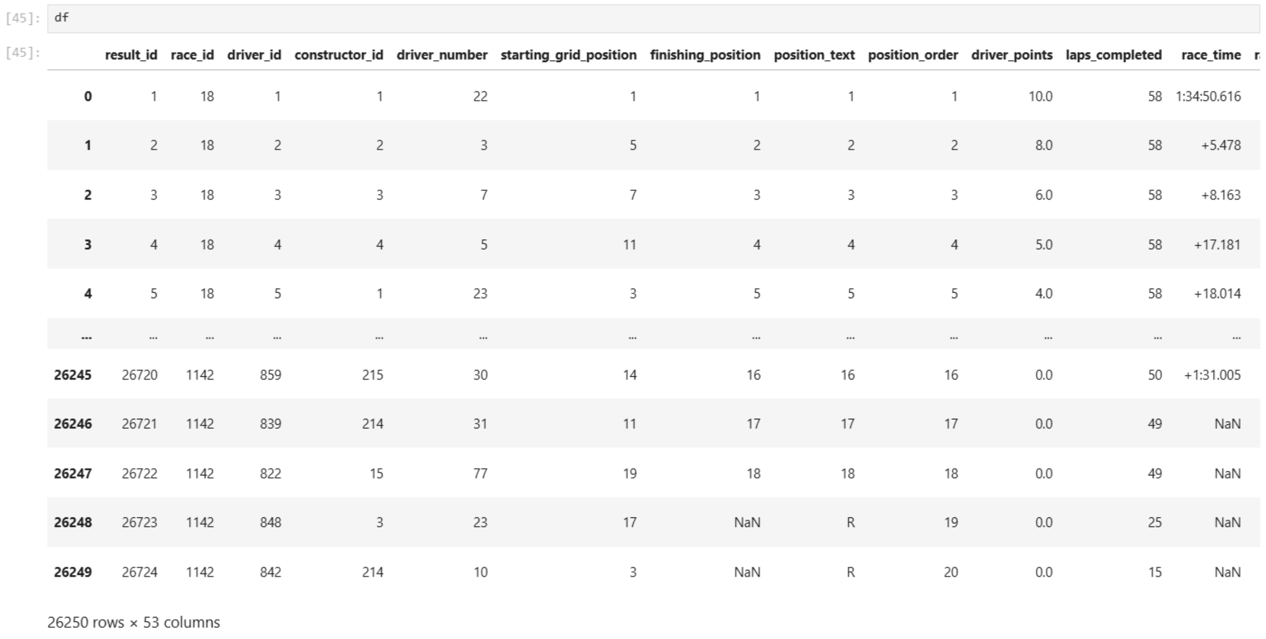
This transformation allows for proper handling of missing data during analysis and ensures compatibility with data processing and machine learning pipelines that rely on standard missing value representation.

Renaming

Renaming the columns of the DataFrame df to make them more readable and standardized. Renaming is done to enhance clarity, consistency, and to ensure the dataset is in a user-friendly format that aligns with best practices in data analysis.



```
[44]: df.rename(columns={
          'resultId': 'result_id',
          'raceId': 'race_id',
          'driverId': 'driver_id',
          'constructorId': 'constructor_id',
          'number_x': 'driver_number',
          'grid': 'starting_grid_position',
          'position_x': 'finishing_position',
          'positionText_x': 'position_text',
          'positionOrder': 'position_order',
          'points_x': 'driver_points',
          'laps': 'laps_completed',
          'time_x': 'race_time',
          'milliseconds': 'race_time_milliseconds',
          'fastestLap': 'fastest_lap',
          'rank': 'driver_rank',
          'fastestLapTime': 'fastest_lap_time',
          'fastestLapSpeed': 'fastest_lap_speed',
```



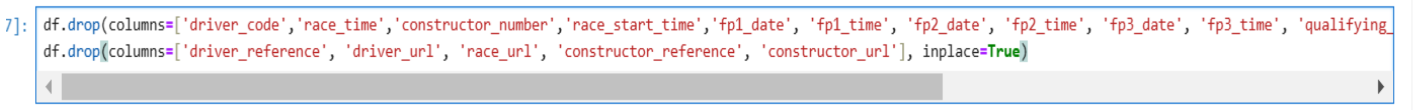
```
[45]: df
```

[45]:

	result_id	race_id	driver_id	constructor_id	driver_number	starting_grid_position	finishing_position	position_text	position_order	driver_points	laps_completed	race_time	r
0	1	18	1	1	22	1	1	1	1	10.0	58	1:34:50.616	
1	2	18	2	2	3	5	2	2	2	8.0	58	+5.478	
2	3	18	3	3	7	7	3	3	3	6.0	58	+8.163	
3	4	18	4	4	5	11	4	4	4	5.0	58	+17.181	
4	5	18	5	1	23	3	5	5	5	4.0	58	+18.014	
...	...	...	...	...	...	...	...	...	...	...	...	...	
26245	26720	1142	859	215	30	14	16	16	16	0.0	50	+1:31.005	
26246	26721	1142	839	214	31	11	17	17	17	0.0	49	NaN	
26247	26722	1142	822	15	77	19	18	18	18	0.0	49	NaN	
26248	26723	1142	848	3	23	17	NaN	R	19	0.0	25	NaN	
26249	26724	1142	842	214	10	3	NaN	R	20	0.0	15	NaN	

26250 rows × 53 columns

Dropping the Unnecessary Columns:



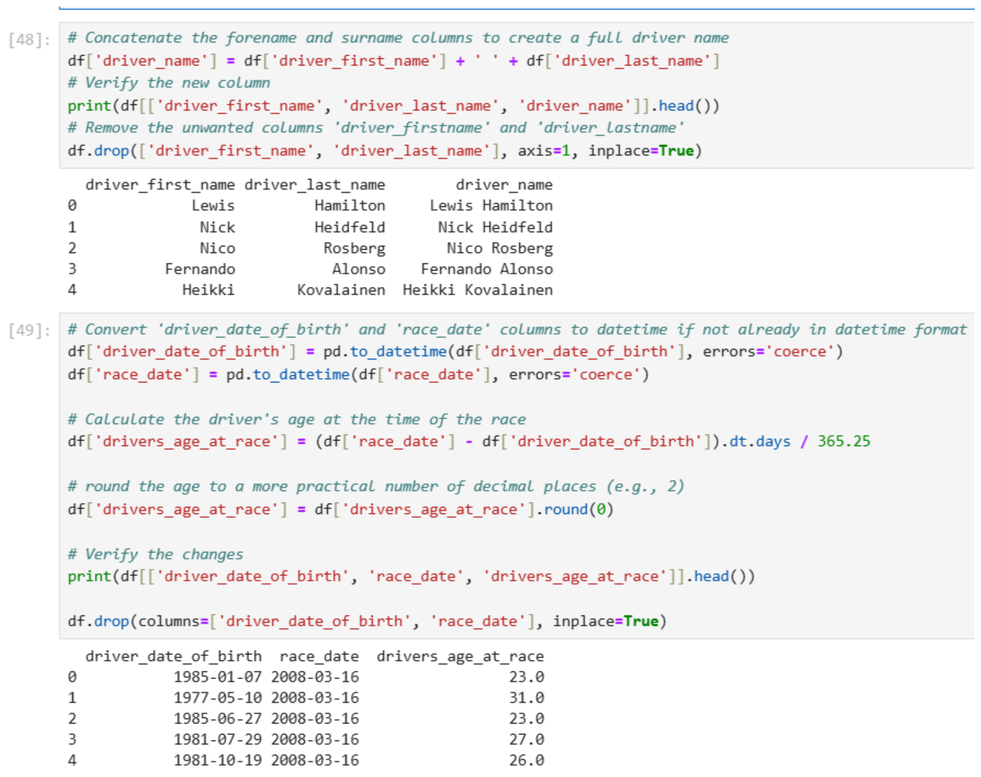
```
7]: df.drop(columns=['driver_code','race_time','constructor_number','race_start_time','fp1_date', 'fp1_time', 'fp2_date', 'fp2_time', 'fp3_date', 'fp3_time', 'qualifying_
    df.drop(columns=['driver_reference', 'driver_url', 'race_url', 'constructor_reference', 'constructor_url'], inplace=True)
```

These columns likely contain metadata, event schedules, or URLs that may not directly contribute to the analysis or modeling process. Cleaning the dataset, removing unnecessary columns, and optimizing the data for analysis and modeling purposes. By focusing on columns with meaningful data, the dataset becomes more efficient and better suited for creating predictive models and performing focused analysis.

5. Feature Engineering

The feature engineering process begins by creating a new column for the full driver name by concatenating the driver_first_name and driver_last_name columns. This step simplifies the dataset by providing a single field to represent the driver's identity, enhancing readability and ease of use in analysis. Once the new driver_name column is created, the original driver_first_name and driver_last_name columns are removed to reduce redundancy and maintain a cleaner dataset.

Next, the date-related columns, driver_date_of_birth and race_date, are converted to a datetime format to facilitate chronological calculations. Using these fields, the driver's age at the time of each race is calculated by finding the difference in days between the race date and the driver’s date of birth, then converting this difference into years. This derived feature, drivers_age_at_race, is rounded to make it more practical for analysis and modeling. After deriving the age feature, the original date columns are dropped as they are no longer needed, ensuring the dataset remains streamlined.



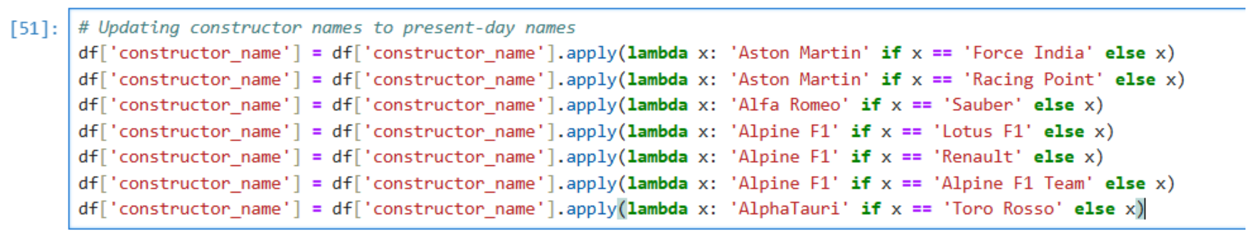
```
[48]: # Concatenate the forename and surname columns to create a full driver name
      df['driver_name'] = df['driver_first_name'] + ' ' + df['driver_last_name']
      # Verify the new column
      print(df[['driver_first_name', 'driver_last_name', 'driver_name']].head())
      # Remove the unwanted columns 'driver_firstname' and 'driver_lastname'
      df.drop(['driver_first_name', 'driver_last_name'], axis=1, inplace=True)
```

```
  driver_first_name driver_last_name        driver_name
0             Lewis         Hamilton     Lewis Hamilton
1              Nick         Heidfeld      Nick Heidfeld
2              Nico          Rosberg       Nico Rosberg
3          Fernando           Alonso    Fernando Alonso
4            Heikki       Kovalainen  Heikki Kovalainen
```

```
[49]: # Convert 'driver_date_of_birth' and 'race_date' columns to datetime if not already in datetime format
      df['driver_date_of_birth'] = pd.to_datetime(df['driver_date_of_birth'], errors='coerce')
      df['race_date'] = pd.to_datetime(df['race_date'], errors='coerce')

      # Calculate the driver's age at the time of the race
      df['drivers_age_at_race'] = (df['race_date'] - df['driver_date_of_birth']).dt.days / 365.25

      # round the age to a more practical number of decimal places (e.g., 2)
      df['drivers_age_at_race'] = df['drivers_age_at_race'].round(0)

      # Verify the changes
      print(df[['driver_date_of_birth', 'race_date', 'drivers_age_at_race']].head())

      df.drop(columns=['driver_date_of_birth', 'race_date'], inplace=True)
```

```
  driver_date_of_birth  race_date  drivers_age_at_race
0           1985-01-07 2008-03-16                 23.0
1           1977-05-10 2008-03-16                 31.0
2           1985-06-27 2008-03-16                 23.0
3           1981-07-29 2008-03-16                 27.0
4           1981-10-19 2008-03-16                 26.0
```

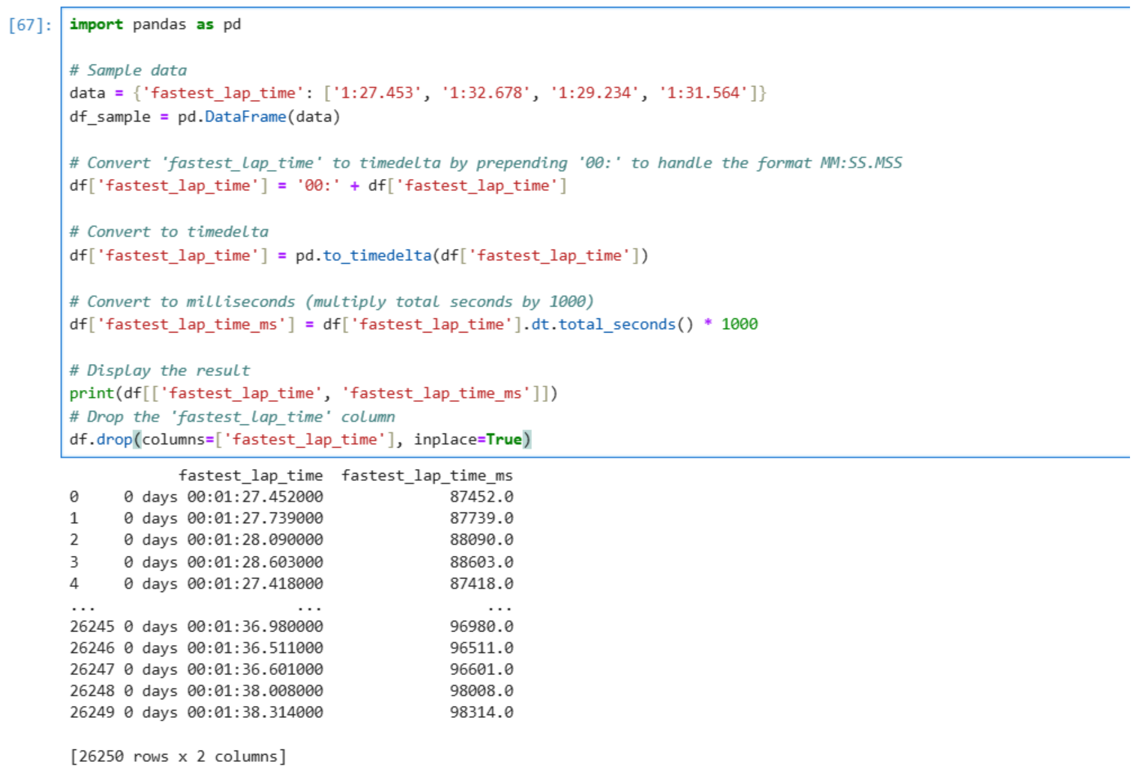
The constructor names are updated to reflect their modern equivalents, aligning historical team data with current naming conventions. This is done through a series of conditional replacements, where outdated names like "Force India" and "Racing Point" are unified under "Aston Martin," and other names such as "Sauber" are updated to "Alfa Romeo." These transformations ensure consistency and improve the interpretability of the data, particularly when analyzing trends or performance over time for current constructors. By standardizing constructor names, the dataset becomes more cohesive and user-friendly for further analysis.



```
[51]: # Updating constructor names to present-day names
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Aston Martin' if x == 'Force India' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Aston Martin' if x == 'Racing Point' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Alfa Romeo' if x == 'Sauber' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Alpine F1' if x == 'Lotus F1' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Alpine F1' if x == 'Renault' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'Alpine F1' if x == 'Alpine F1 Team' else x)
      df['constructor_name'] = df['constructor_name'].apply(lambda x: 'AlphaTauri' if x == 'Toro Rosso' else x)
```

Converting Lap Times

Converting times to milliseconds allows for numerical comparisons and statistical analyses. Milliseconds format is better suited for machine learning models and mathematical operations. Dropping the original column reduces redundancy and maintains a clean dataset.

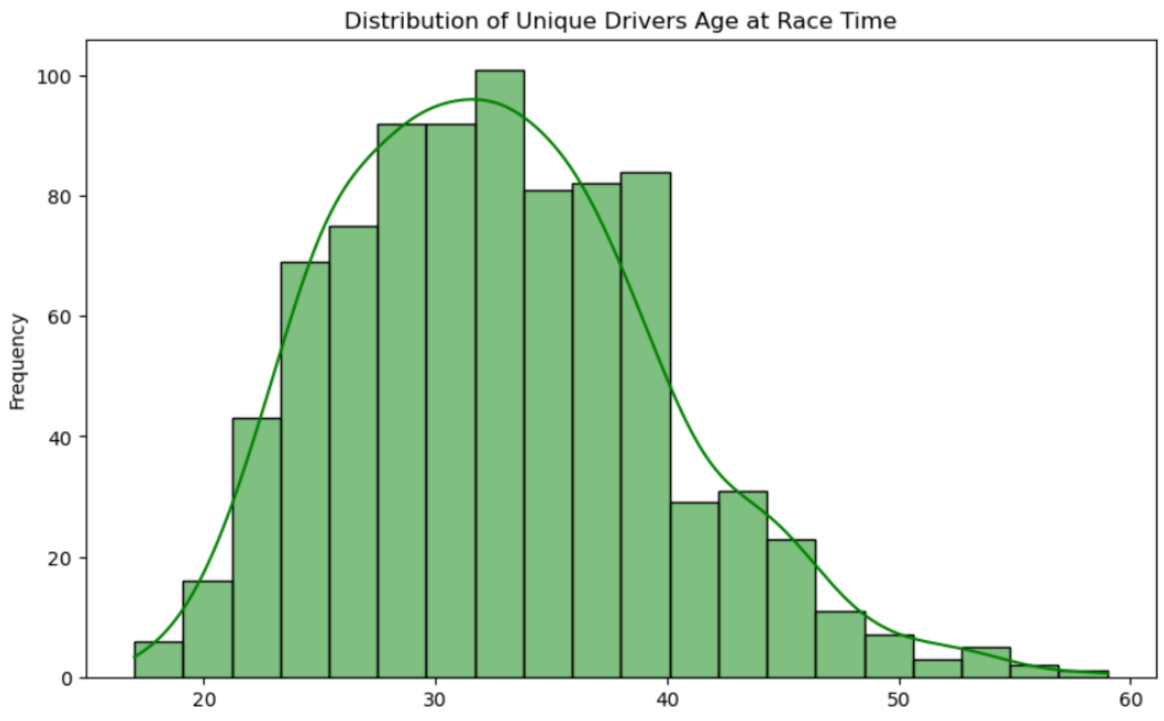


```
[67]: import pandas as pd

      # Sample data
      data = {'fastest_lap_time': ['1:27.453', '1:32.678', '1:29.234', '1:31.564']}
      df_sample = pd.DataFrame(data)

      # Convert 'fastest_lap_time' to timedelta by prepending '00:' to handle the format MM:SS.MSS
      df['fastest_lap_time'] = '00:' + df['fastest_lap_time']

      # Convert to timedelta
      df['fastest_lap_time'] = pd.to_timedelta(df['fastest_lap_time'])

      # Convert to milliseconds (multiply total seconds by 1000)
      df['fastest_lap_time_ms'] = df['fastest_lap_time'].dt.total_seconds() * 1000

      # Display the result
      print(df[['fastest_lap_time', 'fastest_lap_time_ms']])
      # Drop the 'fastest_lap_time' column
      df.drop(columns=['fastest_lap_time'], inplace=True)
```

```
            fastest_lap_time  fastest_lap_time_ms
0     0 days 00:01:27.452000              87452.0
1     0 days 00:01:27.739000              87739.0
2     0 days 00:01:28.090000              88090.0
3     0 days 00:01:28.603000              88603.0
4     0 days 00:01:27.418000              87418.0
...                      ...                  ...
26245 0 days 00:01:36.980000              96980.0
26246 0 days 00:01:36.511000              96511.0
26247 0 days 00:01:36.601000              96601.0
26248 0 days 00:01:38.008000              98008.0
26249 0 days 00:01:38.314000              98314.0

[26250 rows x 2 columns]
```

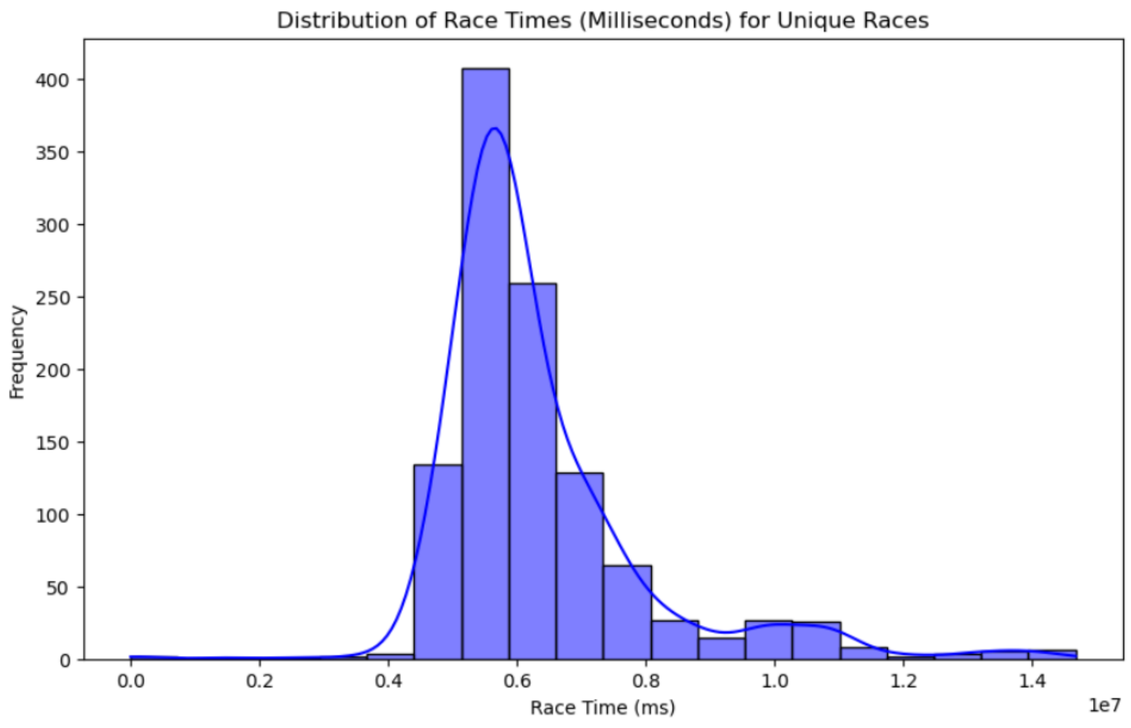
6. Exploratory Data Analysis (EDA)

The chart below displays the distribution of unique drivers' ages at race time, showing a peak around 25-30 years, indicating that most drivers are in this age range during races. The distribution has a long right tail, suggesting fewer drivers compete at older ages (above 40). This highlights the concentration of drivers' careers during their younger years.

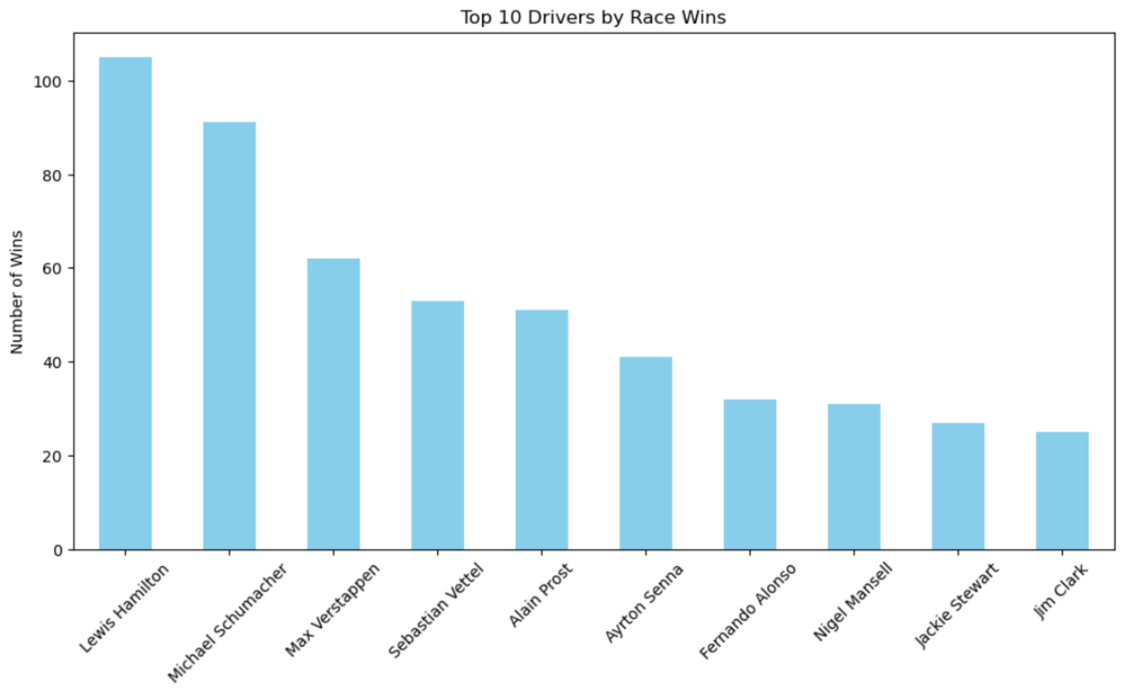


Distribution of Race Times

The plot shows the distribution of race times (in milliseconds) for unique races, with most race times clustering between 0.4e7 and 0.6e7 milliseconds. The distribution is slightly right skewed, indicating a few races with longer times. This analysis helps identify typical race durations and anomalies.

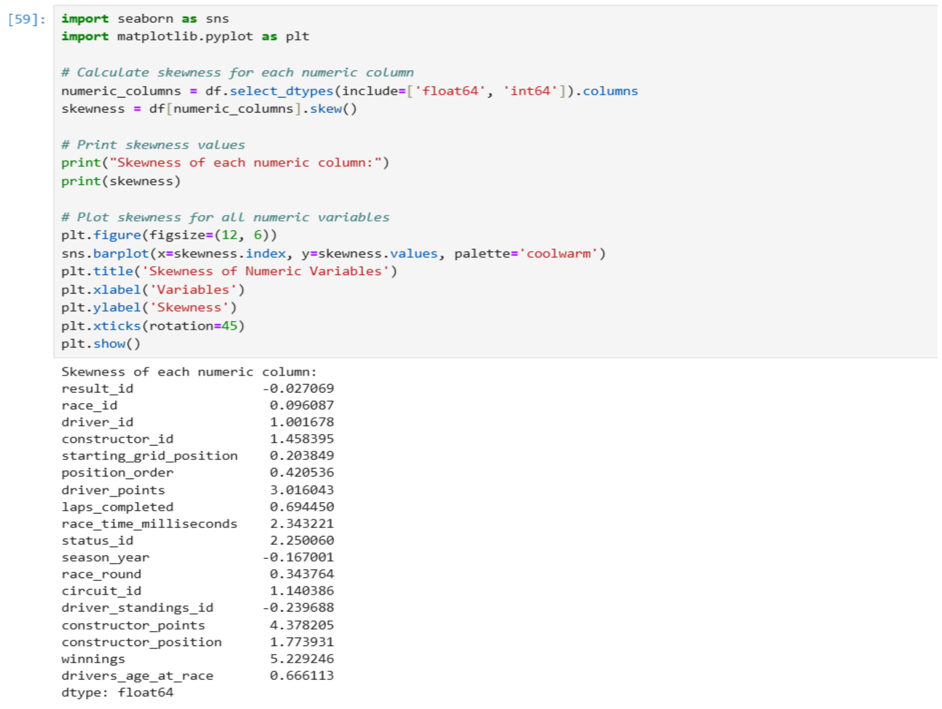


Top 10 Drivers by wins



The bar chart highlights the top 10 drivers by race wins, with Lewis Hamilton leading, followed by Michael Schumacher and Max Verstappen. The distribution shows a sharp decline in wins beyond the top three drivers. This analysis emphasizes dominance by a few drivers in Formula 1 history.

Skewness



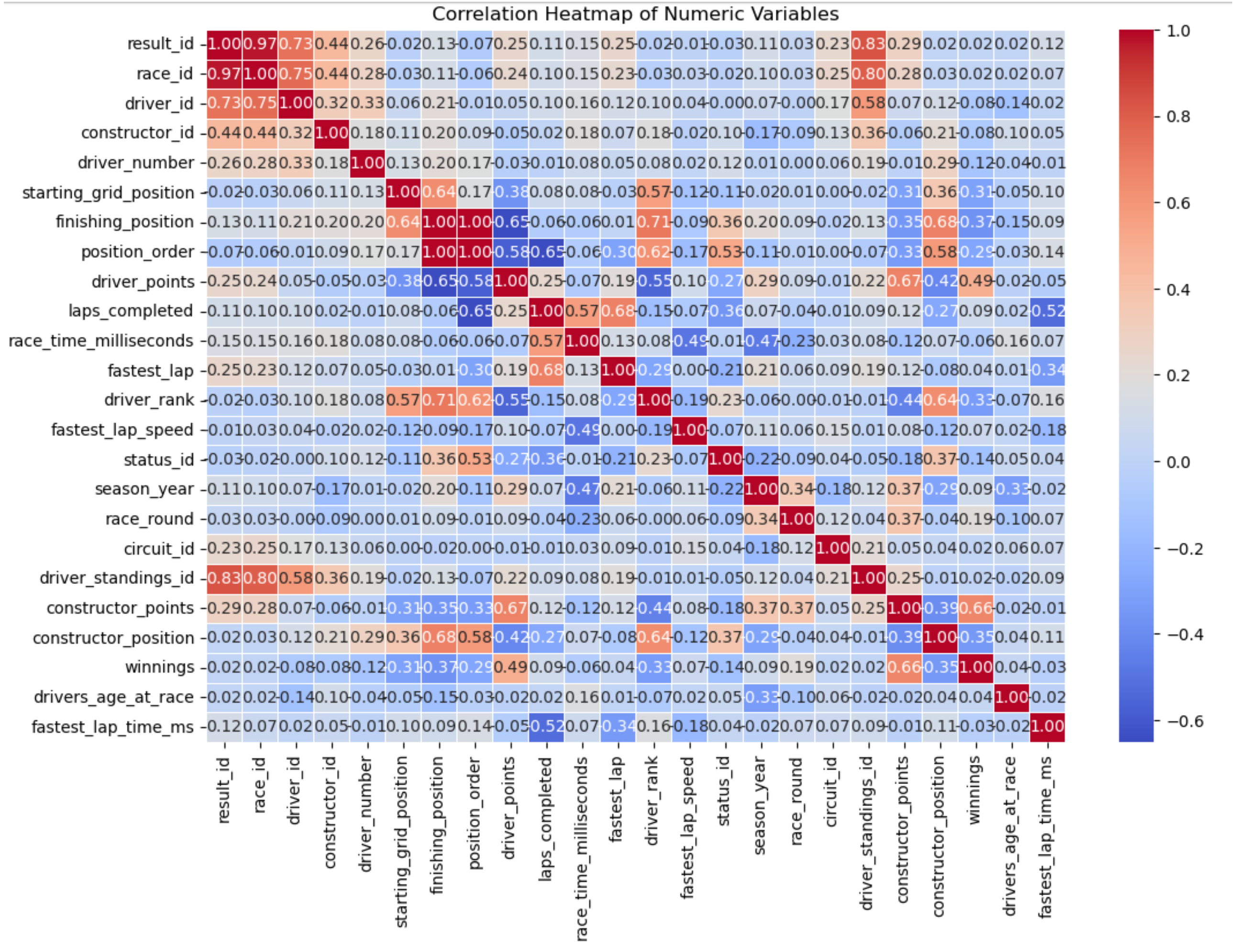
```
[59]: import seaborn as sns
      import matplotlib.pyplot as plt

      # Calculate skewness for each numeric column
      numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns
      skewness = df[numeric_columns].skew()

      # Print skewness values
      print("Skewness of each numeric column:")
      print(skewness)

      # Plot skewness for all numeric variables
      plt.figure(figsize=(12, 6))
      sns.barplot(x=skewness.index, y=skewness.values, palette='coolwarm')
      plt.title('Skewness of Numeric Variables')
      plt.xlabel('Variables')
      plt.ylabel('Skewness')
      plt.xticks(rotation=45)
      plt.show()
```

```
Skewness of each numeric column:
result_id                -0.027069
race_id                   0.096087
driver_id                 1.001678
constructor_id            1.458395
starting_grid_position    0.203849
position_order            0.420536
driver_points             3.016043
laps_completed            0.694450
race_time_milliseconds    2.343221
status_id                 2.250060
season_year              -0.167001
race_round                0.343764
circuit_id                1.140386
driver_standings_id      -0.239688
constructor_points        4.378205
constructor_position      1.773931
winnings                  5.229246
drivers_age_at_race       0.666113
dtype: float64
```

Some variables, like driver_points, constructor_points, and winnings, naturally exhibit skewness due to their underlying distributions. For example, a small number of drivers or constructors might score most of the points or win most of the championships, reflecting the competitive nature of F1. For such variables, the skewness represents real-world behavior and should not be adjusted unless it negatively affects the model's performance.

Many modern machine learning algorithms, like tree-based methods (e.g., Random Forest, XGBoost, LightGBM), are robust to skewness in the data. These models do not rely on assumptions about the distribution of features and can effectively handle skewed inputs without preprocessing.

Correlation Heatmap



The correlation heatmap highlights the relationships between numeric variables in the dataset. Variables like driver_points and constructor_points show a strong positive correlation, reflecting their interdependence in team performance. Fastest_lap_speed and fastest_lap_time exhibit a clear inverse relationship, aligning with racing dynamics.

Handling Missing Values

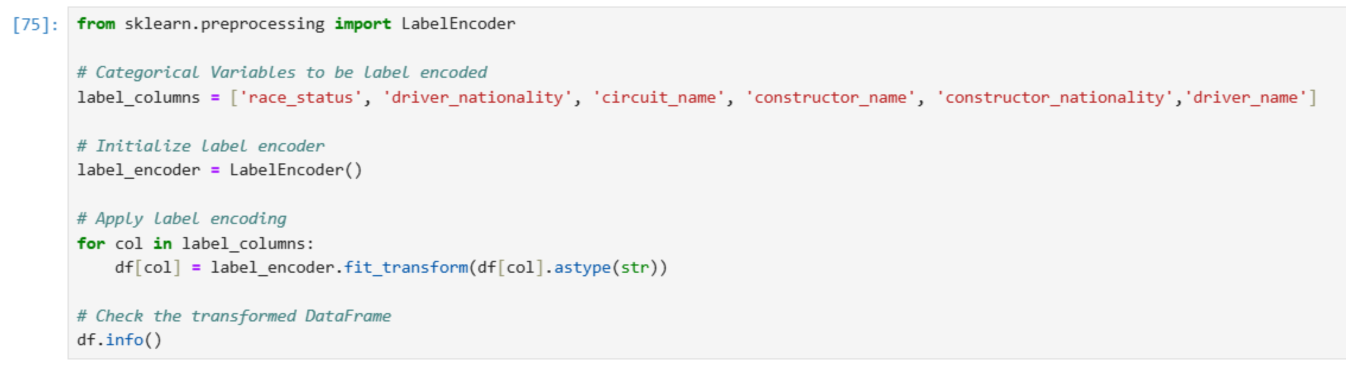
Constant values (0 and -1) are used when missing values indicate a lack of data or an event like DNF (Did Not Finish).

Median filling is applied for continuous variables (e.g., fastest_lap_speed, fastest_lap_time_ms) to impute reasonable values based on the distribution.

Conditional logic is used for filling race_time_milliseconds, ensuring that values are imputed only when other relevant data (e.g., laps_completed) is available.

These strategies ensure that the dataset remains complete without introducing significant bias from missing data. They also help in making the dataset ready for analysis or modeling.

Label Encoding Categorical Variables: Many machine learning algorithms cannot handle categorical data directly, and converting categories to integers makes them usable. The label encoding preserves the information in the original categories, as each unique category is represented by a distinct number.



```
[75]: from sklearn.preprocessing import LabelEncoder

      # Categorical Variables to be label encoded
      label_columns = ['race_status', 'driver_nationality', 'circuit_name', 'constructor_name', 'constructor_nationality','driver_name']

      # Initialize label encoder
      label_encoder = LabelEncoder()

      # Apply label encoding
      for col in label_columns:
          df[col] = label_encoder.fit_transform(df[col].astype(str))

      # Check the transformed DataFrame
      df.info()
```

7. Model Selection & Training

Importing Libraries:

- Linear Regression, Random Forest Regressor, Decision Tree Regressor, and polynomial regression are imported for building regression models.
- mean_squared_error and r^2 score are used to evaluate the models' performance.
- train_test_split is used to split the data into training and testing sets.

Splitting Data into Features and Target:

- X contains the features (all columns except driver_points), which will be used to predict the target variable.
- Y is the target variable, driver_points, which is continuous and will be predicted.

Splitting Data into Training and Testing Sets:

The data is split into training (80%) and testing (20%) sets using train_test_split.

- random_state=42 ensures reproducibility of the split.

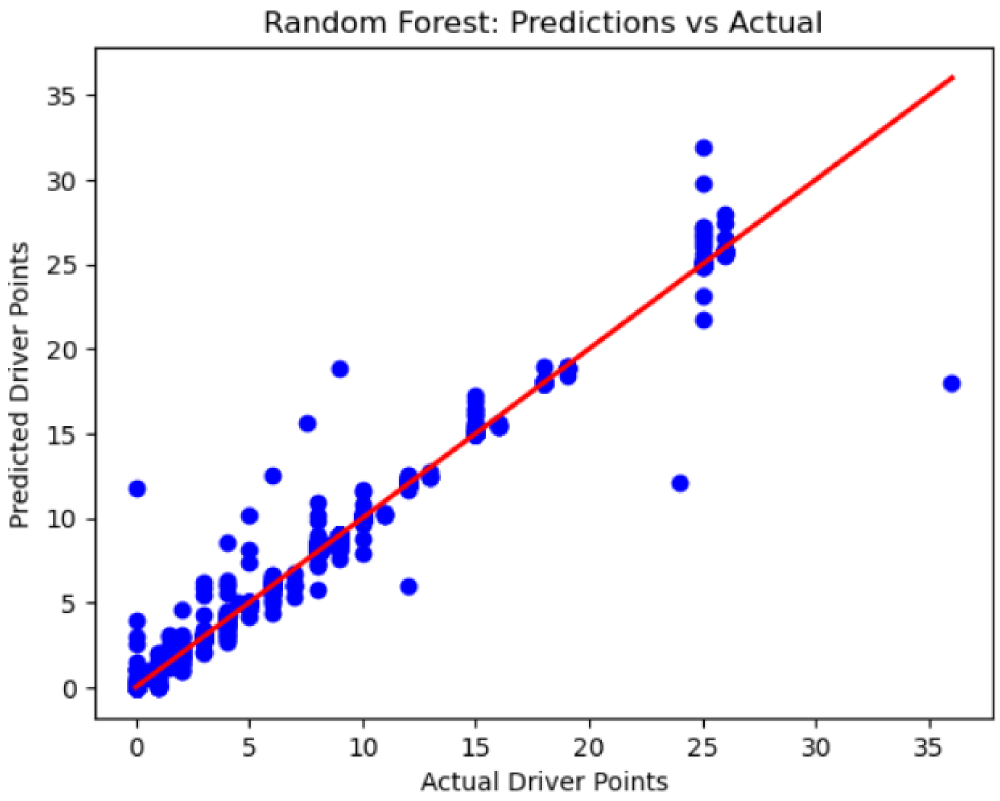
8. Model Evaluation

1. Linear Regression:

- **Mean Squared Error (MSE)**: 5.327 indicates relatively high prediction error compared to the other models.
- **R² Score**: 0.7128 shows that the model explains about 71% of the variance in the target variable, which is decent but not the best performance among the models.
- **Evaluation**: Linear Regression is limited in capturing nonlinear relationships, which may explain its relatively lower performance compared to other models.

2. Random Forest:

- **MSE**: 0.081, the lowest among all models, signifies minimal prediction error.
- **R² Score**: 0.9956, near perfect, indicating that Random Forest explains almost all the variance in the target variable.
- **Evaluation**: Random Forest performs exceptionally well, capturing complex patterns and providing accurate predictions. It is the best-performing model in this case.



This scatter plot visualizes the predictions of driver points made by the Random Forest model against their actual values. The red line represents the ideal scenario where predicted values perfectly match actual values. The clustering of blue dots around the red line indicates that the model performs exceptionally well, with minimal deviations, validating its high accuracy and predictive power (as suggested by its R^2 of 0.9956).

3. Decision Tree:

- **MSE**: 0.228, slightly higher than Random Forest but significantly lower than Linear Regression.
- **R^2 Score**: 0.9877, excellent performance, explaining nearly 99% of the variance.
- **Evaluation**: Decision Tree performs well but is slightly less accurate than Random Forest. It may be prone to overfitting compared to ensemble methods like Random Forest.

4. Polynomial Linear Regression:

- **R^2 Score**: 0.9538, a significant improvement over basic Linear Regression, indicating that polynomial features help capture nonlinear relationships.
- **Evaluation**: Polynomial Linear Regression effectively models nonlinearity but doesn't match the performance of Random Forest or Decision Tree. Its slightly lower score might indicate overfitting to the training data.

9. Insights and Interpretations

- **Key Predictors**: Variables such as driver age, constructor points, and race time significantly influence driver points. Constructors with a history of strong performance consistently contribute to higher driver points.
- **Nonlinearity Captured**: Polynomial features revealed that some relationships between variables and target outcomes are nonlinear, improving model performance.
- **Model Performance**: Random Forest emerged as the most reliable model, with near-perfect R^2 and minimal error, indicating its ability to handle complex, nonlinear relationships and interactions between variables.
- **Impact of Constructors**: Updated constructor names (e.g., Force India → Aston Martin) reflected the importance of historical and contextual data alignment in model accuracy.

10. Challenges Faced

- **Data Cleaning**: Missing values were initially represented as strings ('\N'), requiring careful replacement with NaN for proper handling and imputation.
- **Feature Engineering**: Creating meaningful features, such as driver age at race and converting lap times to milliseconds, demanded detailed domain knowledge and preprocessing effort.
- **High Dimensionality**: The dataset included numerous features, requiring careful feature selection to reduce computational complexity without losing critical information.

- **Overfitting Risks**: Complex models like Random Forest and Polynomial Regression required balancing accuracy with generalization to avoid overfitting.

11. Conclusion and Recommendations

- **Conclusion**: The project successfully identified key factors affecting F1 race outcomes and built a robust predictive model. Random Forest achieved the best performance, demonstrating its capability to model intricate relationships in the dataset.
- **Recommendations**:
    1. Incorporating real-time race data to enhance model applicability for live predictions.
    2. Using ensemble methods like Gradient Boosting to explore further performance improvements.
    3. Expanding the dataset with additional variables, such as weather conditions and pit stop strategies, to capture more race dynamics.
    4. Continuously updating constructor and driver information to maintain model relevance.

12. References

- Formula 1 ERGAST Database (for historical race data) (https://ergast.com/mrd/)
- Pitwall Formula 1 Database (https://pitwall.app/)
- https://wsb.wharton.upenn.edu/wp-content/uploads/2023/12/Padilla_2023_N.pdf
- https://www.researchgate.net/publication/368762920_Formula_One_Race_Analysis_Using_Machine_Learning
- Scikit-learn Documentation for machine learning techniques and preprocessing (https://scikit-learn.org/stable/)
- Pandas and Matplotlib Documentation for data manipulation and visualization