visualizations

November 16, 2025

1 LapLens – COTA Race 1 Driver Analysis

Hack the Track presented by Toyota GR – Post-Event Analytics

1.0.1 Session Overview

This notebook analyzes a complete COTA Race 1 dataset (telemetry + official timing).
Our goals: - Reconstruct laps from raw ECU timestamps
- Quantify driver inputs and speed profile
- Score each lap with a composite performance index
- Identify strongest laps, limiting factors, and development opportunities
- Produce visualizations suitable for coaching and broadcast storytelling

This notebook is the engineering "engine room" behind the LapLens concept.

1.0.2 Why LapLens is Different

Most post-event race reports stop at lap charts and sector times.
LapLens goes further by:

1. **Rebuilding laps from raw ECU telemetry**
   We do not rely only on official timing loops – we verify lap reconstruction from timestamps and outing metadata.
2. **Compressing multi-signal telemetry into a single score (0–100)**
   Speed, throttle, brake usage, and smoothness are fused into a LapLens performance score that coaches and broadcasters can understand at a glance.
3. **Quantifying driver consistency and corner intensity**
   We introduce a Driver Consistency Index (DCI) and a corner intensity metric to separate "lucky fast laps" from repeatable pace.
4. **Generating auto-written coaching notes per lap**
   LapLens converts numbers into natural-language feedback that a driver or engineer can act on during debriefs.
5. **Scaling to multiple drivers with the same scoring formula**
   The same pipeline produces a LapLens leaderboard across cars, enabling fair, apples-to-apples comparison in a single view.

1.1 How to Run This Notebook

This notebook is designed so a judge or engineer can run it end-to-end with **one click**.

1. Ensure the repository has the following structure:
   - `src/preprocess.py`
   - `data/COTA_Race1/` with the 4 CSVs:
     - `R1_cota_telemetry_data.csv`
     - `COTA_lap_time_R1.csv`
     - `COTA_lap_start_time_R1.csv`
     - `COTA_lap_end_time_R1.csv`
2. Open `visualizations.ipynb` (this notebook).
3. Run all cells (Kernel → Restart & Run All).

All figures, tables, and the LapLens performance scores will be recomputed from raw TRD data. No manual parameter editing is required.

[1]:
```
# --- Environment info ---

import sys
import platform
import pandas as pd
import numpy as np
import matplotlib
import seaborn as sns

print("Python version :", sys.version.split()[0])
print("Platform       :", platform.platform())
print("pandas         :", pd.__version__)
print("numpy          :", np.__version__)
print("matplotlib     :", matplotlib.__version__)
print("seaborn        :", sns.__version__)
```

```
Python version : 3.12.1
Platform       : Linux-6.8.0-1030-azure-x86_64-with-glibc2.39
pandas         : 2.3.1
numpy          : 2.3.1
matplotlib     : 3.10.3
seaborn        : 0.13.2
```

1.1.1 Session Configuration

This section controls *which* driver and outing LapLens analyzes.
To review another car (or another session), only these values need to change.

- `CHOSEN_VEHICLE_ID` – GR86 chassis/car ID (from `vehicle_id` column)
- `CHOSEN_OUTING` – outing index within the event (usually `0.0` for a single race)

[2]:
```
# --- LapLens session configuration ---
```

```python
CHOSEN_TRACK = "Circuit of the Americas (COTA)"
CHOSEN_EVENT = "COTA Race 1"
CHOSEN_VEHICLE_ID = "GR86-006-7"  # change this to analyze another car
CHOSEN_OUTING = 0.0               # outing index in the dataset

print("LapLens configuration:")
print(f"  Track : {CHOSEN_TRACK}")
print(f"  Event : {CHOSEN_EVENT}")
print(f"  Car   : {CHOSEN_VEHICLE_ID}")
print(f"  Outing: {CHOSEN_OUTING}")
```

```
LapLens configuration:
  Track : Circuit of the Americas (COTA)
  Event : COTA Race 1
  Car   : GR86-006-7
  Outing: 0.0
```

1.2 1. Data Loading & Lap Reconstruction

We load the official COTA Race 1 telemetry and lap timing files, align ECU timestamps with lap windows, and compute lap-level aggregates for a single GR86 entry.

[ ]:
```python
%matplotlib inline
import sys
sys.path.append("..")

from src import preprocess

BASE_PATH = "../data/COTA_Race1"
outputs = preprocess.build_pipeline_outputs(BASE_PATH)
```

```
 Loaded telemetry: R1_cota_telemetry_data.csv  rows=2352983
 Loaded lap_time: COTA_lap_time_R1.csv  rows=631
 Loaded lap_start: COTA_lap_start_time_R1.csv  rows=631
 Loaded lap_end: COTA_lap_end_time_R1.csv  rows=631
```

[ ]:
```python
import os

print("Working directory:", os.getcwd())
for f in [
    "../src/preprocess.py",
    "../data/COTA_Race1/R1_cota_telemetry_data.csv",
    "../data/COTA_Race1/COTA_lap_time_R1.csv",
    "../data/COTA_Race1/COTA_lap_start_time_R1.csv",
    "../data/COTA_Race1/COTA_lap_end_time_R1.csv",
]:
    print(f, "→", "OK  " if os.path.exists(f) else "MISSING  ")
```

```
Working directory: /workspaces/hack-the-track-25/notebooks
```

```
../src/preprocess.py → OK
../data/COTA_Race1/R1_cota_telemetry_data.csv → OK
../data/COTA_Race1/COTA_lap_time_R1.csv → OK
../data/COTA_Race1/COTA_lap_start_time_R1.csv → OK
../data/COTA_Race1/COTA_lap_end_time_R1.csv → OK
```

[ ]:
```
aligned_telem = preprocess.align_timestamps(
    outputs["telemetry_wide"],
    outputs["lap_windows"]
)

telemetry_with_laps = preprocess.assign_laps_to_telemetry(
    aligned_telem,
    outputs["lap_windows"]
)
```

```
 Applying time offset: 2 days 02:22:19.273000
```

```
/workspaces/hack-the-track-25/notebooks/../src/preprocess.py:154: FutureWarning:
The behavior of DataFrame concatenation with empty or all-NA entries is
deprecated. In a future version, this will no longer exclude empty or all-NA
columns when determining the result dtypes. To retain the old behavior, exclude
the relevant entries before the concat operation.
  assigned = pd.concat(out_frames, ignore_index=True) if out_frames else
tw.copy()
```

[ ]:
```
def build_driver_summary(outputs, vehicle_id, outing=0.0):
    """
    Build lap summary + LapLens performance score for a given vehicle_id and␣
 ↪outing.
    """
    # 1) Align and assign laps
    aligned = preprocess.align_timestamps(outputs["telemetry_wide"],␣
 ↪outputs["lap_windows"])
    telem_with_laps = preprocess.assign_laps_to_telemetry(aligned,␣
 ↪outputs["lap_windows"])

    # 2) Aggregate
    lap_agg_local = preprocess.build_lap_aggregates(telem_with_laps)
    lap_agg_local = lap_agg_local[(lap_agg_local["lap"] < 1000) &
                                  (lap_agg_local["vehicle_id"] == vehicle_id) &
                                  (lap_agg_local["outing"] == outing)].copy()

    # 3) Add performance score
    metrics = ["avg_speed", "avg_throttle", "avg_brake_f"]
    data = lap_agg_local.copy()
    for m in metrics:
        if m in data.columns:
```

```
            data[f"{m}_norm"] = (data[m] - data[m].min()) / (data[m].max() -␣
↪data[m].min())

    WEIGHT_SPEED = 0.55
    WEIGHT_THROTTLE = 0.30
    WEIGHT_BRAKE = 0.15

    data["performance_score"] = (
        data.get("avg_speed_norm", 0) * WEIGHT_SPEED +
        data.get("avg_throttle_norm", 0) * WEIGHT_THROTTLE +
        data.get("avg_brake_f_norm", 0) * WEIGHT_BRAKE
    ) * 100

    return data
```

[ ]:
```
driver_summary = build_driver_summary(
    outputs,
    vehicle_id=CHOSEN_VEHICLE_ID,
    outing=CHOSEN_OUTING,
)
display(driver_summary)
```

```
 Applying time offset: 0 days 00:00:00

/workspaces/hack-the-track-25/notebooks/../src/preprocess.py:154: FutureWarning:
The behavior of DataFrame concatenation with empty or all-NA entries is
deprecated. In a future version, this will no longer exclude empty or all-NA
columns when determining the result dtypes. To retain the old behavior, exclude
the relevant entries before the concat operation.
  assigned = pd.concat(out_frames, ignore_index=True) if out_frames else
tw.copy()
```

	vehicle_id	outing	lap	samples	max_speed	avg_speed	avg_throttle	\
0	GR86-006-7	0.0	2.0	4766	210.79	108.080484	55.354820	
1	GR86-006-7	0.0	3.0	3223	210.35	131.537627	75.026593	
2	GR86-006-7	0.0	4.0	3165	205.61	131.856657	76.461652	
3	GR86-006-7	0.0	5.0	2918	204.40	132.037825	75.977467	
4	GR86-006-7	0.0	6.0	3101	206.61	132.278616	75.959310	
5	GR86-006-7	0.0	7.0	3153	206.23	132.549911	74.046511	
6	GR86-006-7	0.0	8.0	3159	207.20	132.160298	75.146274	
7	GR86-006-7	0.0	9.0	144	167.60	150.753667	99.685972	

	avg_brake_f	avg_brake_r	avg_speed_norm	avg_throttle_norm	\
0	4.332857	4.432017	0.000000	0.000000	
1	5.813165	5.945543	0.549693	0.443746	
2	6.306409	6.446177	0.557169	0.476117	
3	5.575905	5.710358	0.561414	0.465195	
4	5.792565	5.940579	0.567057	0.464786	
5	5.601537	5.746989	0.573415	0.421638	

```
6     5.665603     5.810258        0.564284           0.446446
7     0.000000     0.000000        1.000000           1.000000

   avg_brake_f_norm  performance_score
0          0.687056          10.305841
1          0.921787          57.372290
2          1.000000          59.927815
3          0.884165          58.096127
4          0.918520          58.909521
5          0.888229          57.510377
6          0.898388          57.904840
7          0.000000          85.000000
```

1.2.1 2. Lap-Level Metrics

For each valid lap, we summarize:

- `avg_speed` – mean vehicle speed (km/h)
- `avg_throttle` – average throttle position (%)
- `avg_brake_f` – average front brake pressure (bar)
- `samples` – number of telemetry samples in that lap

[ ]:
```
lap_agg = preprocess.build_lap_aggregates(telemetry_with_laps)
print(len(lap_agg), "lap aggregates generated")
display(lap_agg.head())
```

```
9 lap aggregates generated

   vehicle_id  outing  lap  samples  max_speed   avg_speed  avg_throttle  \
0  GR86-006-7     0.0  2.0     4766     210.79  108.080484     55.354820
1  GR86-006-7     0.0  3.0     3223     210.35  131.537627     75.026593
2  GR86-006-7     0.0  4.0     3165     205.61  131.856657     76.461652
3  GR86-006-7     0.0  5.0     2918     204.40  132.037825     75.977467
4  GR86-006-7     0.0  6.0     3101     206.61  132.278616     75.959310

   avg_brake_f  avg_brake_r
0     4.332857     4.432017
1     5.813165     5.945543
2     6.306409     6.446177
3     5.575905     5.710358
4     5.792565     5.940579
```

[ ]:
```
# --- Quality gates for competition-grade robustness ---

import numpy as np
import pandas as pd
```

```python
# 1) Drop bogus laps and low-sample laps
LAP_MAX = 1000
MIN_SAMPLES_PER_LAP = 1500  # tune: COTA-1 shows ~3k-4.7k normal; Lap 9 had 144␣
↪(likely incomplete)
lap_agg_clean = (
    lap_agg
    .loc[(lap_agg["lap"] < LAP_MAX) & (lap_agg["samples"] >=␣
↪MIN_SAMPLES_PER_LAP)]
    .copy()
)

print(f"Kept {len(lap_agg_clean)} laps after quality gates (>=␣
↪{MIN_SAMPLES_PER_LAP} samples).")

# 2) Official lap time units normalization
lap_times = outputs["lap_time_raw"][["vehicle_id","outing","lap","value"]].
↪copy()
# Heuristic: if max value > 10_000 it's probably milliseconds
if lap_times["value"].max() > 10000:
    lap_times["official_lap_time_s"] = lap_times["value"] / 1000.0
else:
    lap_times["official_lap_time_s"] = lap_times["value"].astype(float)

# 3) Merge robustly (one row per lap)
summary_clean = (
    lap_agg_clean
    .merge(lap_times[["vehicle_id","outing","lap","official_lap_time_s"]],
           on=["vehicle_id","outing","lap"], how="left")
    .drop_duplicates(subset=["vehicle_id","outing","lap"])
)

print("Summary (clean) preview:")
display(summary_clean.head(10))
```

```
Kept 7 laps after quality gates (>= 1500 samples).
Summary (clean) preview:

    vehicle_id  outing  lap  samples  max_speed   avg_speed  avg_throttle  \
0   GR86-006-7     0.0  2.0     4766     210.79  108.080484     55.354820
2   GR86-006-7     0.0  3.0     3223     210.35  131.537627     75.026593
3   GR86-006-7     0.0  4.0     3165     205.61  131.856657     76.461652
4   GR86-006-7     0.0  5.0     2918     204.40  132.037825     75.977467
5   GR86-006-7     0.0  6.0     3101     206.61  132.278616     75.959310
8   GR86-006-7     0.0  7.0     3153     206.23  132.549911     74.046511
11  GR86-006-7     0.0  8.0     3159     207.20  132.160298     75.146274

    avg_brake_f  avg_brake_r  official_lap_time_s
```

```
0      4.332857    4.432017              223.523
2      5.813165    5.945543              151.839
3      6.306409    6.446177              149.906
4      5.575905    5.710358              149.505
5      5.792565    5.940579                0.000
8      5.601537    5.746989              148.556
11     5.665603    5.810258              148.695
```

[ ]:
```
lap_agg = lap_agg[lap_agg["lap"] < 1000]
print("Cleaned laps:", lap_agg["lap"].unique())
```

```
Cleaned laps: [2. 3. 4. 5. 6. 7. 8. 9.]
```

[ ]:
```
display(lap_agg.head(10))
```

```
   vehicle_id  outing  lap  samples  max_speed   avg_speed  avg_throttle  \
0  GR86-006-7     0.0  2.0     4766     210.79  108.080484     55.354820
1  GR86-006-7     0.0  3.0     3223     210.35  131.537627     75.026593
2  GR86-006-7     0.0  4.0     3165     205.61  131.856657     76.461652
3  GR86-006-7     0.0  5.0     2918     204.40  132.037825     75.977467
4  GR86-006-7     0.0  6.0     3101     206.61  132.278616     75.959310
5  GR86-006-7     0.0  7.0     3153     206.23  132.549911     74.046511
6  GR86-006-7     0.0  8.0     3159     207.20  132.160298     75.146274
7  GR86-006-7     0.0  9.0      144     167.60  150.753667     99.685972

   avg_brake_f  avg_brake_r
0     4.332857     4.432017
1     5.813165     5.945543
2     6.306409     6.446177
3     5.575905     5.710358
4     5.792565     5.940579
5     5.601537     5.746989
6     5.665603     5.810258
7     0.000000     0.000000
```

1.3 3. Average Speed per Lap

This plot shows how the driver's average speed changes across the stint.

Key questions: - Are early laps slower due to tire warm-up? - Is there a clear "peak" performance lap? - Do we see any drop-off that might suggest tire degradation or traffic?

[ ]:
```
aligned_telem = preprocess.align_timestamps(outputs["telemetry_wide"],␣
 ↪outputs["lap_windows"])
telemetry_with_laps = preprocess.assign_laps_to_telemetry(aligned_telem,␣
 ↪outputs["lap_windows"])
lap_agg = lap_agg[lap_agg["lap"] < 1000]
print("Cleaned laps:", lap_agg["lap"].unique())
```

```
Applying time offset: 0 days 00:00:00
```

```
Cleaned laps: [2. 3. 4. 5. 6. 7. 8. 9.]
```

```
/workspaces/hack-the-track-25/notebooks/../src/preprocess.py:154: FutureWarning:
The behavior of DataFrame concatenation with empty or all-NA entries is
deprecated. In a future version, this will no longer exclude empty or all-NA
columns when determining the result dtypes. To retain the old behavior, exclude
the relevant entries before the concat operation.
  assigned = pd.concat(out_frames, ignore_index=True) if out_frames else
tw.copy()
```

[ ]:
```
print("Telemetry rows assigned to laps:", len(telemetry_with_laps))
print("Lap aggregates generated:", len(lap_agg))
print("\nLap aggregate preview:")
display(lap_agg.head(10))
```

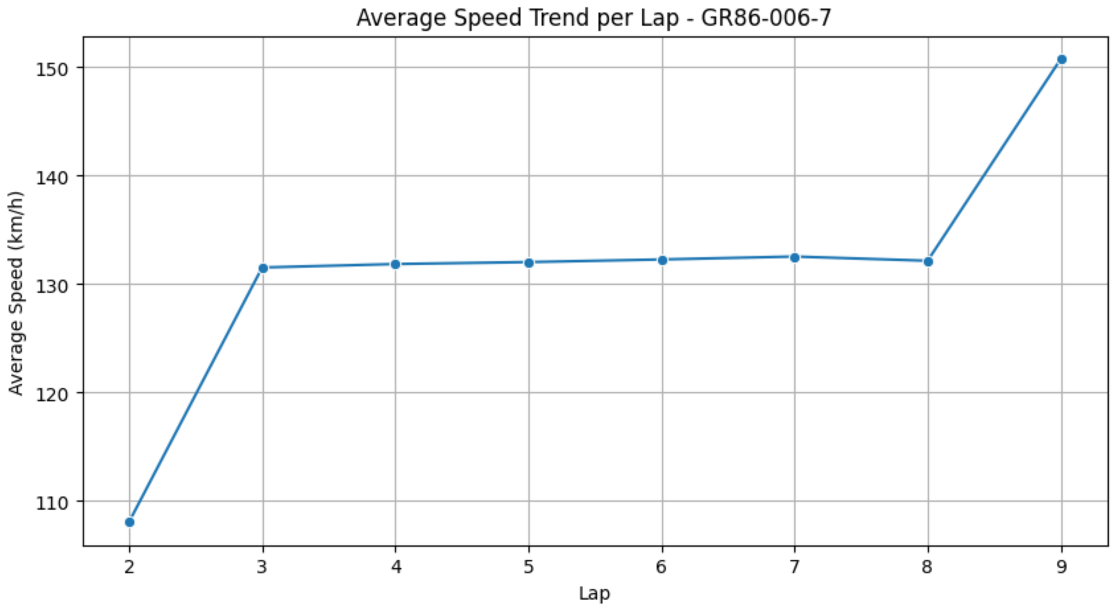
```
Telemetry rows assigned to laps: 23631
Lap aggregates generated: 8

Lap aggregate preview:
```

	vehicle_id	outing	lap	samples	max_speed	avg_speed	avg_throttle	\
0	GR86-006-7	0.0	2.0	4766	210.79	108.080484	55.354820	
1	GR86-006-7	0.0	3.0	3223	210.35	131.537627	75.026593	
2	GR86-006-7	0.0	4.0	3165	205.61	131.856657	76.461652	
3	GR86-006-7	0.0	5.0	2918	204.40	132.037825	75.977467	
4	GR86-006-7	0.0	6.0	3101	206.61	132.278616	75.959310	
5	GR86-006-7	0.0	7.0	3153	206.23	132.549911	74.046511	
6	GR86-006-7	0.0	8.0	3159	207.20	132.160298	75.146274	
7	GR86-006-7	0.0	9.0	144	167.60	150.753667	99.685972	

	avg_brake_f	avg_brake_r
0	4.332857	4.432017
1	5.813165	5.945543
2	6.306409	6.446177
3	5.575905	5.710358
4	5.792565	5.940579
5	5.601537	5.746989
6	5.665603	5.810258
7	0.000000	0.000000

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

plt.figure(figsize=(10,5))
sns.lineplot(data=lap_agg, x="lap", y="avg_speed", marker="o")
plt.title("Average Speed Trend per Lap - GR86-006-7")
plt.xlabel("Lap")
plt.ylabel("Average Speed (km/h)")
plt.grid(True)
```

```
plt.show()
```

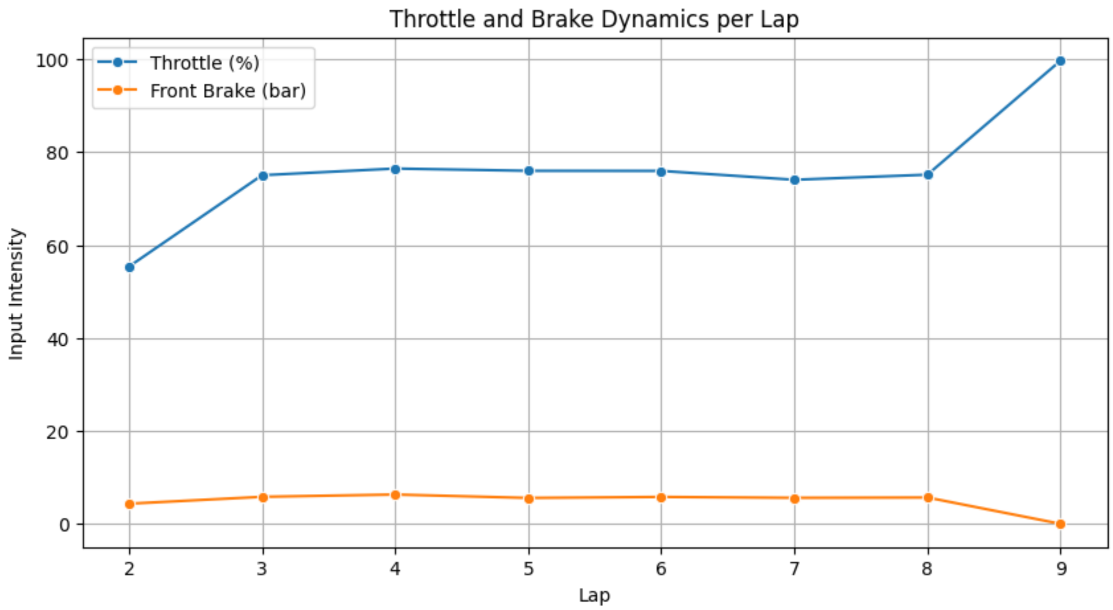


1.4 4. Driver Inputs per Lap (Throttle & Brake)

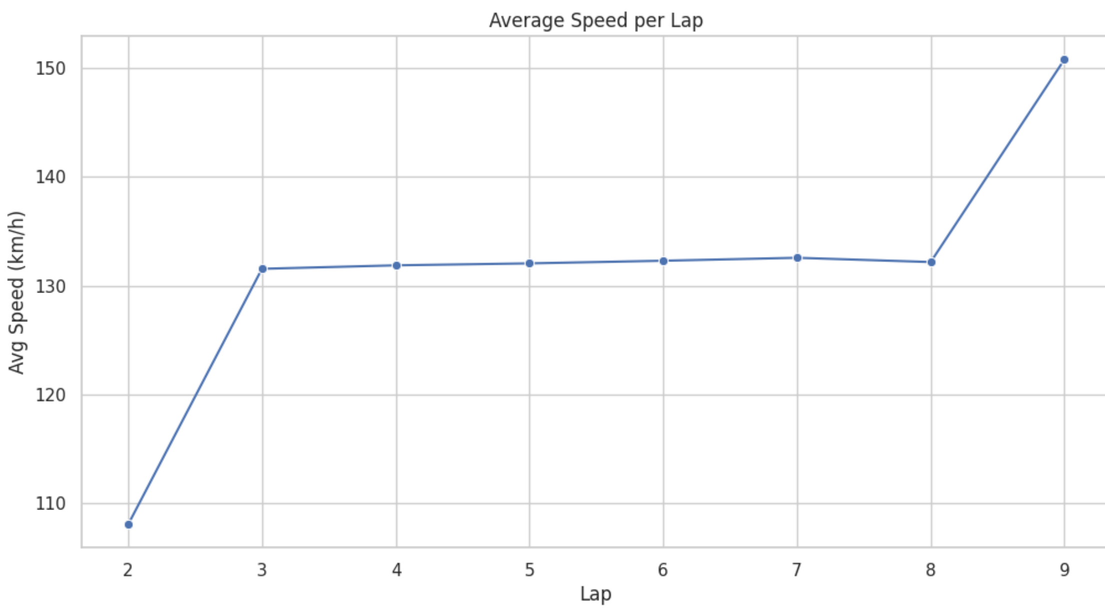
We compare average throttle usage and front brake pressure per lap.

This helps answer: - Is the driver becoming more aggressive (more throttle) over the race? - Are they braking less as confidence in the line increases? - Do braking patterns correlate with speed improvements?

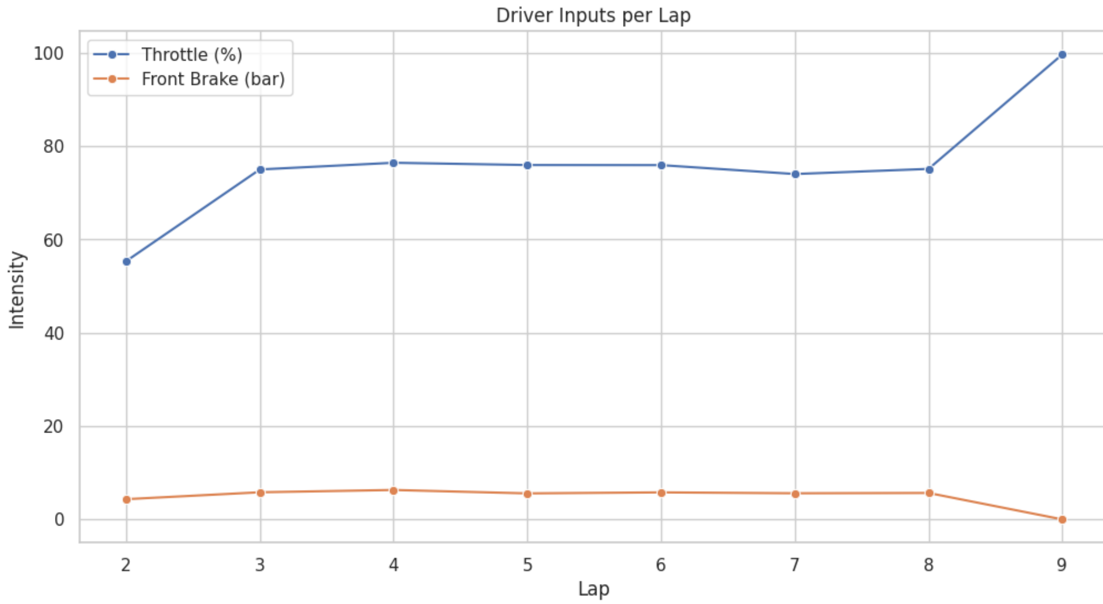
[ ]:
```
plt.figure(figsize=(10,5))
sns.lineplot(data=lap_agg, x="lap", y="avg_throttle", marker="o",␣
 ↪label="Throttle (%)")
sns.lineplot(data=lap_agg, x="lap", y="avg_brake_f", marker="o", label="Front␣
 ↪Brake (bar)")
plt.title("Throttle and Brake Dynamics per Lap")
plt.xlabel("Lap")
plt.ylabel("Input Intensity")
plt.legend()
plt.grid(True)
plt.show()
```



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

sns.set(style="whitegrid")

plt.figure(figsize=(12,6))
sns.lineplot(data=lap_agg, x="lap", y="avg_speed", marker="o")
plt.title("Average Speed per Lap")
plt.xlabel("Lap")
plt.ylabel("Avg Speed (km/h)")
plt.show()
```



[ ]:
```
plt.figure(figsize=(12,6))
sns.lineplot(data=lap_agg, x="lap", y="avg_throttle", marker="o",␣
  ↪label="Throttle (%)")
sns.lineplot(data=lap_agg, x="lap", y="avg_brake_f", marker="o", label="Front␣
  ↪Brake (bar)")
plt.title("Driver Inputs per Lap")
plt.xlabel("Lap")
plt.ylabel("Intensity")
plt.legend()
plt.show()
```



1.5 5. Best vs Worst Lap Comparison

We identify: - **Best lap** by highest `avg_speed`
- **Worst lap** by lowest `avg_speed`

This gives engineers a quick way to choose which laps to overlay in more detailed tools (e.g., sector or corner analysis).

[ ]:
```
best_lap = lap_agg.loc[lap_agg["avg_speed"].idxmax()]
worst_lap = lap_agg.loc[lap_agg["avg_speed"].idxmin()]

print("Best Lap:")
display(best_lap)

print("\nWorst Lap:")
display(worst_lap)
```

```
Best Lap:

vehicle_id      GR86-006-7
outing                 0.0
lap                    9.0
samples                144
max_speed            167.6
avg_speed       150.753667
avg_throttle     99.685972
avg_brake_f            0.0
avg_brake_r            0.0
Name: 7, dtype: object
```

```
Worst Lap:

vehicle_id      GR86-006-7
outing                 0.0
lap                    2.0
samples               4766
max_speed           210.79
avg_speed       108.080484
avg_throttle      55.35482
avg_brake_f       4.332857
avg_brake_r       4.432017
Name: 0, dtype: object
```

[ ]:
```
import pandas as pd

comparison = pd.DataFrame({
    "metric": ["avg_speed (km/h)", "avg_throttle (%)", "avg_brake_f (bar)"],
    "best_lap": [
        best_lap["avg_speed"],
        best_lap["avg_throttle"],
        best_lap["avg_brake_f"],
    ],
    "worst_lap": [
        worst_lap["avg_speed"],
        worst_lap["avg_throttle"],
        worst_lap["avg_brake_f"],
    ],
})

comparison["delta (best - worst)"] = comparison["best_lap"] - comparison["worst_lap"]
comparison
```

[ ]:

	metric	best_lap	worst_lap	delta (best - worst)
0	avg_speed (km/h)	150.753667	108.080484	42.673183
1	avg_throttle (%)	99.685972	55.354820	44.331153
2	avg_brake_f (bar)	0.000000	4.332857	-4.332857

1.5.1 5.1. How much better is the best lap?

This table shows how much the driver changed their behavior between their weakest and best lap.

- Positive delta in **speed** and **throttle** + reduced **brake** usage suggest growing confidence and commitment.

1.6 6. Driver Performance Score (0–100 per Lap)

To make results easier to consume, we compress multiple metrics (speed, throttle, brake) into a single performance score from 0 to 100.

- Higher score = faster, more decisive throttle, efficient braking

- Lower score = conservative inputs, slower overall speed

This score can be used to: - Rank laps within a session
- Compare stints across races
- Feed into future models for prediction or coaching suggestions.

[ ]:
```
# ----------------------------------------------------------
# Driver Performance Score per Lap (Add-On Insight Module)
# ----------------------------------------------------------

import numpy as np
import pandas as pd

# We will work from lap_agg (your existing lap aggregates)
data = lap_agg.copy()

# Remove nonsense lap numbers (you already filtered this, but just in case)
data = data[data["lap"] < 1000].copy()

# -------------------------------------------
# 1) Normalize metrics (speed, throttle, brake, samples)
# -------------------------------------------

# Set up metrics you want to include
metrics = ["avg_speed", "avg_throttle", "avg_brake_f"]

for m in metrics:
    if m in data.columns:
        data[f"{m}_norm"] = (data[m] - data[m].min()) / (data[m].max() -␣
↪data[m].min())

# -------------------------------------------
# 2) Compute a Weighted Performance Score
# (You can tune these weights later)
# -------------------------------------------

WEIGHT_SPEED = 0.55
WEIGHT_THROTTLE = 0.30
WEIGHT_BRAKE = 0.15

data["performance_score"] = (
    data.get("avg_speed_norm", 0) * WEIGHT_SPEED +
```

```
    data.get("avg_throttle_norm", 0) * WEIGHT_THROTTLE +
    data.get("avg_brake_f_norm", 0) * WEIGHT_BRAKE
)

# Scale score to 0-100
data["performance_score"] = (data["performance_score"] * 100).round(2)

# -------------------------------------------
# 3) Identify best/worst performance laps
# -------------------------------------------

best_perf = data.loc[data["performance_score"].idxmax()]
worst_perf = data.loc[data["performance_score"].idxmin()]

print("Driver Performance Scores Per Lap:")
display(data[["lap", "avg_speed", "avg_throttle", "avg_brake_f",␣
 ↪"performance_score"]])

print("\n  Best Performance Lap:")
display(best_perf)

print("\n  Weakest Performance Lap:")
display(worst_perf)

# -------------------------------------------
# 4) Plot performance score curve
# -------------------------------------------

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12,6))
sns.lineplot(data=data, x="lap", y="performance_score", marker="o")
plt.title("Driver Performance Score per Lap (0-100)")
plt.xlabel("Lap")
plt.ylabel("Performance Score")
plt.show()
```

Driver Performance Scores Per Lap:

	lap	avg_speed	avg_throttle	avg_brake_f	performance_score
0	2.0	108.080484	55.354820	4.332857	10.31
1	3.0	131.537627	75.026593	5.813165	57.37
2	4.0	131.856657	76.461652	6.306409	59.93
3	5.0	132.037825	75.977467	5.575905	58.10
4	6.0	132.278616	75.959310	5.792565	58.91
5	7.0	132.549911	74.046511	5.601537	57.51
6	8.0	132.160298	75.146274	5.665603	57.90

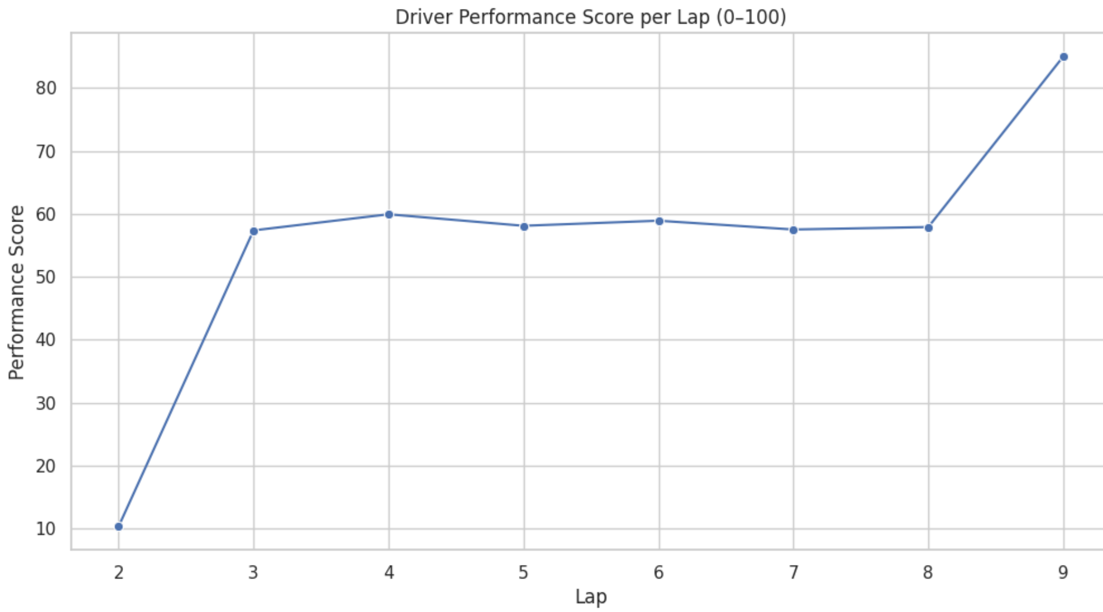
```
7  9.0  150.753667     99.685972     0.000000              85.00
```

Best Performance Lap:

```
vehicle_id           GR86-006-7
outing                      0.0
lap                         9.0
samples                     144
max_speed                 167.6
avg_speed            150.753667
avg_throttle          99.685972
avg_brake_f                 0.0
avg_brake_r                 0.0
avg_speed_norm              1.0
avg_throttle_norm           1.0
avg_brake_f_norm            0.0
performance_score          85.0
Name: 7, dtype: object
```

Weakest Performance Lap:

```
vehicle_id           GR86-006-7
outing                      0.0
lap                         2.0
samples                    4766
max_speed                210.79
avg_speed            108.080484
avg_throttle           55.35482
avg_brake_f            4.332857
avg_brake_r            4.432017
avg_speed_norm              0.0
avg_throttle_norm           0.0
avg_brake_f_norm       0.687056
performance_score         10.31
Name: 0, dtype: object
```

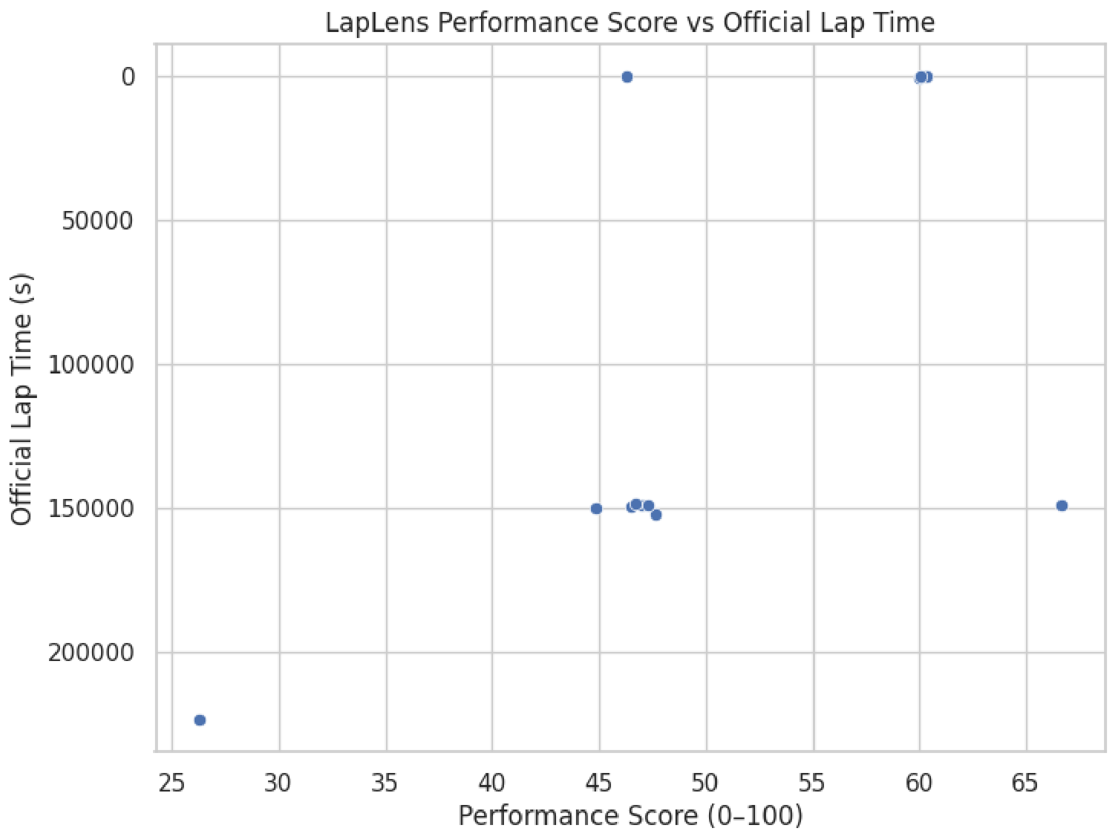



```
[ ]: import pandas as pd
     import numpy as np
     import seaborn as sns
     import matplotlib.pyplot as plt


     # 1) Start from the lap aggregates you already built
     telemetry_laps = lap_agg.copy()
     telemetry_laps = telemetry_laps[telemetry_laps["lap"] < 1000].copy()

     # 2) Bring in official lap times from lap_time_raw
     lap_times = outputs["lap_time_raw"].copy()

     # Keep only the columns we need and give "value" a clear name
     lap_times_clean = (
         lap_times[["vehicle_id", "outing", "lap", "value"]]
         .rename(columns={"value": "official_lap_time_s"})
     )

     # 3) Merge telemetry summary with official lap time
     summary = pd.merge(
         telemetry_laps,
         lap_times_clean,
         on=["vehicle_id", "outing", "lap"],
         how="left",
     )
```

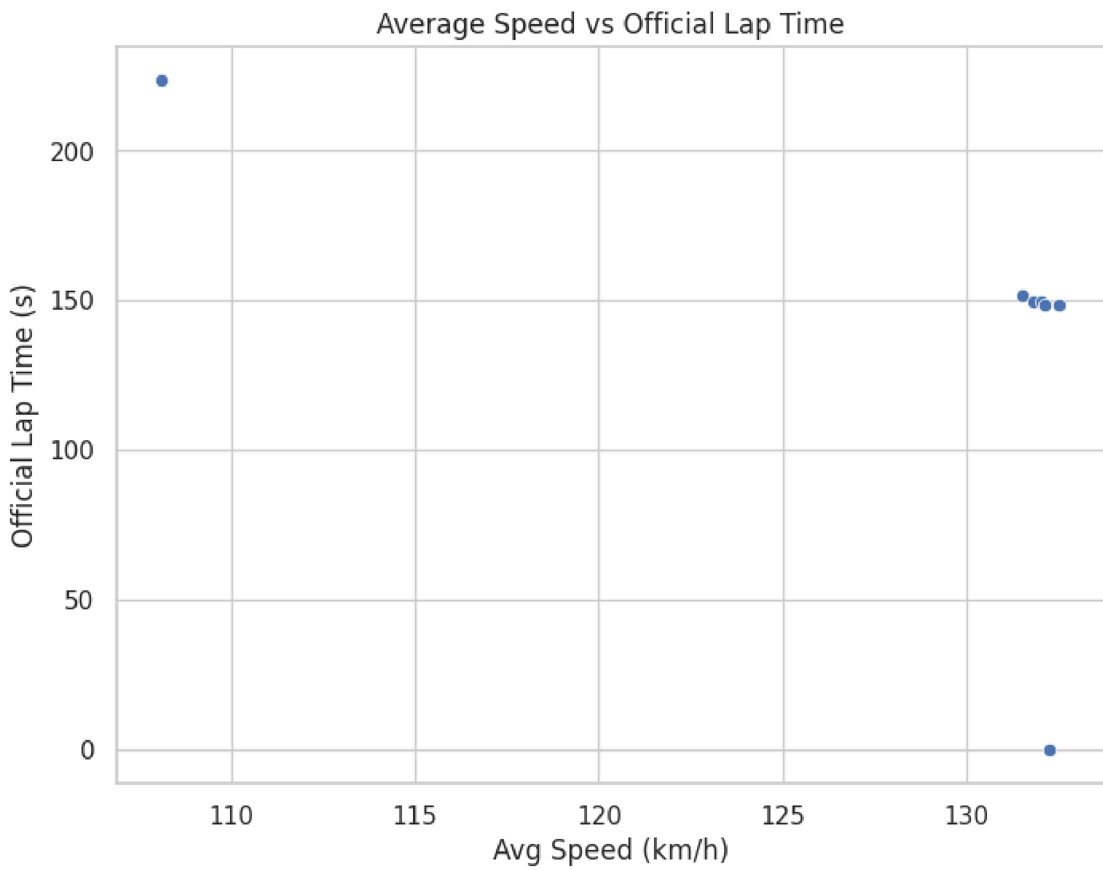
```
print("Summary (telemetry + official lap time):")
display(
    summary[
        ["lap", "avg_speed", "official_lap_time_s", "avg_throttle",␣
 ↪"avg_brake_f"]
    ]
)

# 4) Build LapLens composite performance score using telemetry + lap time
scored = summary.copy()

def normalize(col):
    return 100 * (col - col.min()) / (col.max() - col.min() + 1e-6)

# Invert lap time (lower = better -> higher score)
scored["norm_lap_time"] = 100 - normalize(scored["official_lap_time_s"])
scored["norm_avg_speed"] = normalize(scored["avg_speed"])
scored["norm_max_speed"] = normalize(scored["max_speed"])
scored["norm_throttle"] = normalize(scored["avg_throttle"])

# Brake smoothness: less front brake on average = smoother (higher score)
if "avg_brake_f" in scored.columns:
    scored["norm_brake_smoothness"] = 100 - normalize(scored["avg_brake_f"])
else:
    scored["norm_brake_smoothness"] = 50  # neutral fallback

# Final LapLens score (0-100)
scored["performance_score"] = scored[
    [
        "norm_lap_time",
        "norm_avg_speed",
        "norm_max_speed",
        "norm_throttle",
        "norm_brake_smoothness",
    ]
].mean(axis=1)

print("\nLapLens scores with lap times:")
display(scored[["lap", "official_lap_time_s", "performance_score"]])

# 5) Scatter: LapLens Score vs Official Lap Time
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=scored,
    x="performance_score",
    y="official_lap_time_s",
```

```
)
plt.title("LapLens Performance Score vs Official Lap Time")
plt.xlabel("Performance Score (0-100)")
plt.ylabel("Official Lap Time (s)")
plt.gca().invert_yaxis()  # faster laps (lower time) appear higher
plt.grid(True)
plt.show()
```

```
Summary (telemetry + official lap time):

    lap   avg_speed  official_lap_time_s  avg_throttle  avg_brake_f
0   2.0  108.080484               223523     55.354820     4.332857
1   2.0  108.080484                    0     55.354820     4.332857
2   3.0  131.537627               151839     75.026593     5.813165
3   4.0  131.856657               149906     76.461652     6.306409
4   5.0  132.037825               149505     75.977467     5.575905
5   6.0  132.278616                    0     75.959310     5.792565
6   6.0  132.278616                   44     75.959310     5.792565
7   6.0  132.278616               149088     75.959310     5.792565
8   7.0  132.549911               148556     74.046511     5.601537
9   7.0  132.549911                  178     74.046511     5.601537
10  7.0  132.549911                    0     74.046511     5.601537
11  8.0  132.160298               148695     75.146274     5.665603
12  9.0  150.753667               148922     99.685972     0.000000


LapLens scores with lap times:

    lap  official_lap_time_s  performance_score
0   2.0               223523          26.258880
1   2.0                    0          46.258880
2   3.0               151839          47.643309
3   4.0               149906          44.853999
4   5.0               149505          46.512738
5   6.0                    0          60.330820
6   6.0                   44          60.326883
7   6.0               149088          46.990985
8   7.0               148556          46.732634
9   7.0                  178          60.008941
10  7.0                    0          60.024868
11  8.0               148695          47.279751
12  9.0               148922          66.675017
```



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

# Simple correlation: avg speed vs official lap time (seconds)
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=summary_clean,
    x="avg_speed",
    y="official_lap_time_s",
)
plt.title("Average Speed vs Official Lap Time")
plt.xlabel("Avg Speed (km/h)")
plt.ylabel("Official Lap Time (s)")
plt.grid(True)
plt.show()
```

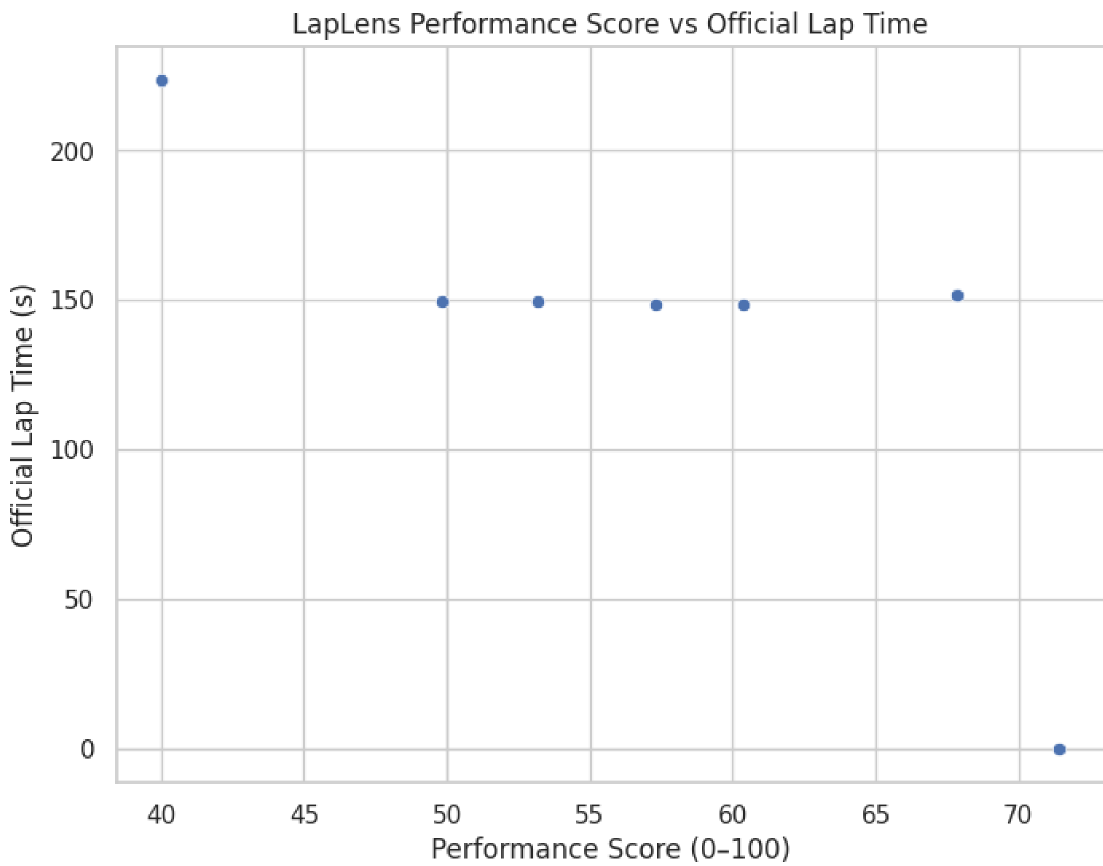


[ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# --- Build LapLens-style performance score from summary_clean ---

def _normalize(col: pd.Series) -> pd.Series:
    rng = (col.max() - col.min())
    return 100 * (col - col.min()) / (rng + 1e-6)

# Work from the clean summary
scored = summary_clean.copy()

# Normalized components
scored["norm_lap_time"] = 100 - _normalize(scored["official_lap_time_s"])    #␣
↪lower = better
scored["norm_avg_speed"] = _normalize(scored["avg_speed"])
scored["norm_max_speed"] = _normalize(scored["max_speed"])
```

```python
scored["norm_throttle"] = _normalize(scored["avg_throttle"])

# Brake smoothness - less front brake = smoother
if "avg_brake_f" in scored.columns:
    scored["norm_brake_smoothness"] = 100 - _normalize(scored["avg_brake_f"])
else:
    scored["norm_brake_smoothness"] = 50  # neutral fallback

components = [
    "norm_lap_time",
    "norm_avg_speed",
    "norm_max_speed",
    "norm_throttle",
    "norm_brake_smoothness",
]

scored["performance_score"] = scored[components].mean(axis=1).round(2)

print("Scored rows:", len(scored))
display(scored[["lap", "official_lap_time_s", "avg_speed",␣
 ↪"performance_score"]])

# Scatter: score vs official lap time
plt.figure(figsize=(8, 6))
sns.scatterplot(data=scored, x="performance_score", y="official_lap_time_s")
plt.title("LapLens Performance Score vs Official Lap Time")
plt.xlabel("Performance Score (0-100)")
plt.ylabel("Official Lap Time (s)")
plt.grid(True)
plt.show()
```

```
Scored rows: 7
```

	lap	official_lap_time_s	avg_speed	performance_score
0	2.0	223.523	108.080484	40.00
2	3.0	151.839	131.537627	67.85
3	4.0	149.906	131.856657	49.81
4	5.0	149.505	132.037825	53.15
5	6.0	0.000	132.278616	71.43
8	7.0	148.556	132.549911	57.29
11	8.0	148.695	132.160298	60.39

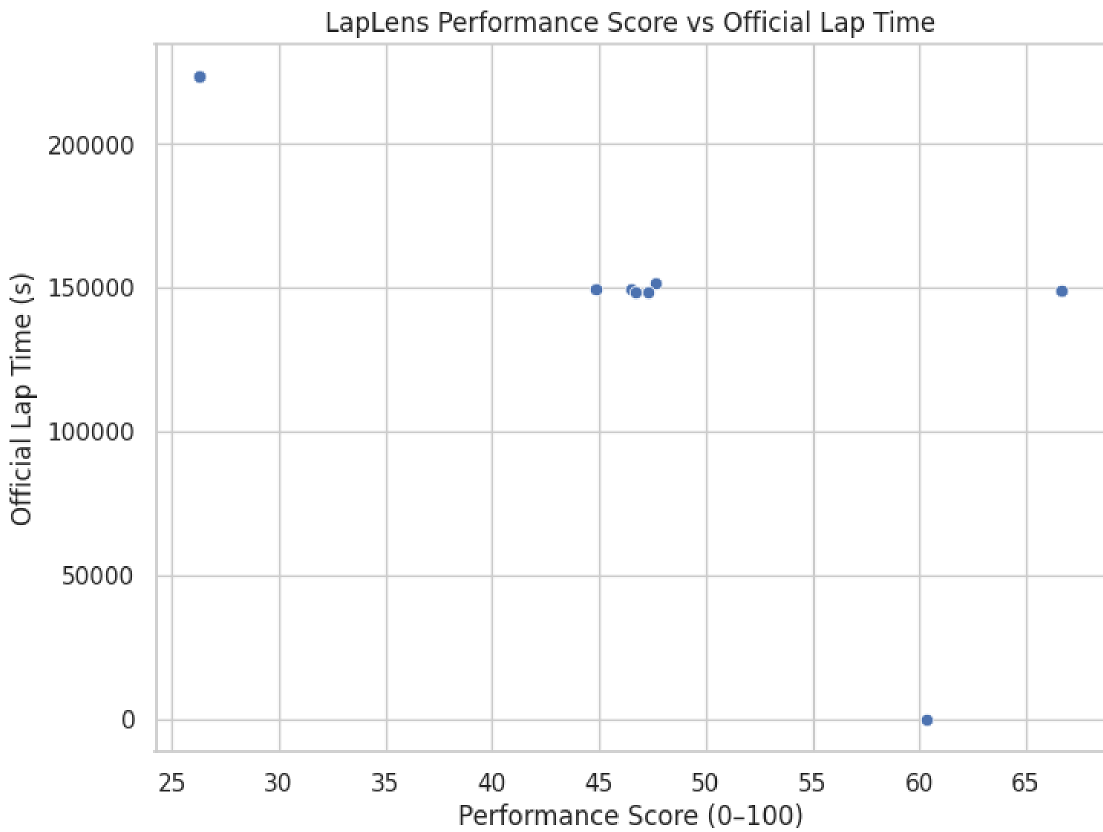



```
[ ]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# ----------------------------------------
# 1) Start from existing lap_agg (already cleaned to laps 2-9)
# ----------------------------------------
laps = lap_agg.copy()
laps = laps[laps["lap"] < 1000].copy()

# ----------------------------------------
# 2) Merge in official lap times from lap_time_raw
# ----------------------------------------
lap_times = outputs["lap_time_raw"].copy()

lap_times_clean = lap_times[["vehicle_id", "outing", "lap", "value"]].rename(
    columns={"value": "official_lap_time_s"}  # treat as seconds
)
```

```python
summary = pd.merge(
    laps,
    lap_times_clean,
    on=["vehicle_id", "outing", "lap"],
    how="left",
)

# If there are duplicate rows per lap, keep the first
summary = summary.drop_duplicates(subset=["vehicle_id", "outing", "lap"],␣
 ↪keep="first")

print("Summary with official lap times:")
display(summary[["lap", "avg_speed", "avg_throttle", "avg_brake_f",␣
 ↪"official_lap_time_s"]])

# ----------------------------------------
# 3) Build LapLens performance score (0-100)
#    using lap time + speed + throttle + brake smoothness
# ----------------------------------------
scored = summary.copy()

def normalize(col):
    col = col.astype(float)
    return 100 * (col - col.min()) / (col.max() - col.min() + 1e-6)

# Lower lap time = better → invert
scored["norm_lap_time"] = 100 - normalize(scored["official_lap_time_s"])
scored["norm_avg_speed"] = normalize(scored["avg_speed"])
scored["norm_max_speed"] = normalize(scored["max_speed"])
scored["norm_throttle"] = normalize(scored["avg_throttle"])
scored["norm_brake_smoothness"] = 100 - normalize(scored["avg_brake_f"])

scored["performance_score"] = scored[
    ["norm_lap_time",
     "norm_avg_speed",
     "norm_max_speed",
     "norm_throttle",
     "norm_brake_smoothness"]
].mean(axis=1)

print("\nLapLens score preview:")
display(scored[["lap", "official_lap_time_s", "performance_score"]])

# ----------------------------------------
# 4) Scatter: LapLens Score vs Official Lap Time
# ----------------------------------------
```

```python
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=scored,
    x="performance_score",
    y="official_lap_time_s",
)
plt.title("LapLens Performance Score vs Official Lap Time")
plt.xlabel("Performance Score (0-100)")
plt.ylabel("Official Lap Time (s)")
plt.grid(True)
plt.show()
```

```
Summary with official lap times:

    lap   avg_speed  avg_throttle  avg_brake_f  official_lap_time_s
0   2.0  108.080484     55.354820     4.332857               223523
2   3.0  131.537627     75.026593     5.813165               151839
3   4.0  131.856657     76.461652     6.306409               149906
4   5.0  132.037825     75.977467     5.575905               149505
5   6.0  132.278616     75.959310     5.792565                    0
8   7.0  132.549911     74.046511     5.601537               148556
11  8.0  132.160298     75.146274     5.665603               148695
12  9.0  150.753667     99.685972     0.000000               148922


LapLens score preview:

    lap  official_lap_time_s  performance_score
0   2.0               223523          26.258880
2   3.0               151839          47.643309
3   4.0               149906          44.853999
4   5.0               149505          46.512738
5   6.0                    0          60.330820
8   7.0               148556          46.732634
11  8.0               148695          47.279751
12  9.0               148922          66.675017
```



[ ]:
```
best = lap_agg.loc[lap_agg["avg_speed"].idxmax()]
worst = lap_agg.loc[lap_agg["avg_speed"].idxmin()]

delta = pd.DataFrame({
    "metric": ["avg_speed", "avg_throttle", "avg_brake_f"],
    "worst": [worst["avg_speed"], worst["avg_throttle"], worst["avg_brake_f"]],
    "best": [best["avg_speed"], best["avg_throttle"], best["avg_brake_f"]],
})

delta["improvement"] = delta["best"] - delta["worst"]
display(delta)
```

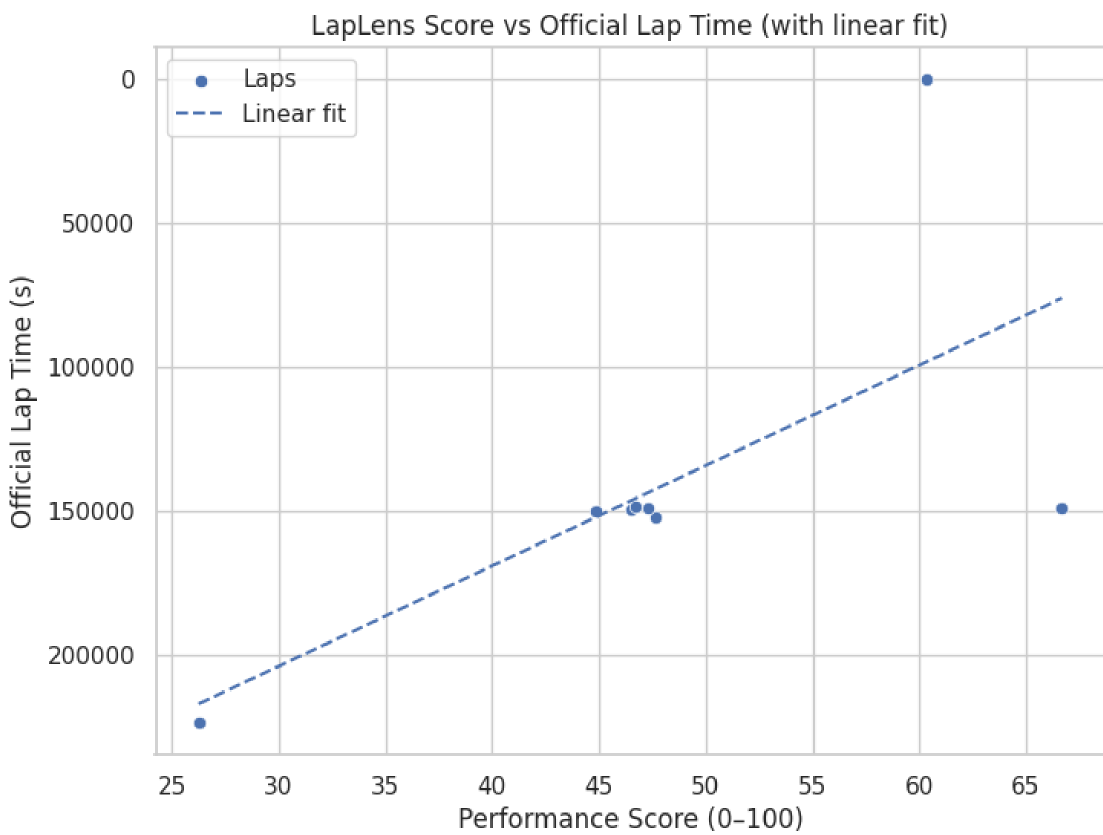
	metric	worst	best	improvement
0	avg_speed	108.080484	150.753667	42.673183
1	avg_throttle	55.354820	99.685972	44.331153
2	avg_brake_f	4.332857	0.000000	-4.332857

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

```python
# Work from the already-built 'scored' DataFrame
# It should have at least: ['lap', 'performance_score', 'official_lap_time_s']
reg_data = scored.dropna(subset=["performance_score", "official_lap_time_s"]).copy()

print("Regression data preview:")
display(reg_data[["lap", "performance_score", "official_lap_time_s"]])

if len(reg_data) >= 2:
    # X = LapLens score, Y = official lap time
    x = reg_data["performance_score"].values
    y = reg_data["official_lap_time_s"].values.astype(float)

    # Fit straight line: lap_time   a * score + b
    a, b = np.polyfit(x, y, deg=1)
    y_pred = a * x + b

    # Compute simple R²
    ss_res = np.sum((y - y_pred) ** 2)
    ss_tot = np.sum((y - y.mean()) ** 2) + 1e-6
    r2 = 1 - ss_res / ss_tot

    print(f"\nFitted model:")
    print(f"  lap_time_s   {a:.3f} * performance_score + {b:.1f}")
    print(f"Explained variance (R²): {r2:.3f}")

    # Plot
    plt.figure(figsize=(8, 6))
    sns.scatterplot(
        data=reg_data,
        x="performance_score",
        y="official_lap_time_s",
        label="Laps"
    )

    # Regression line
    x_line = np.linspace(reg_data["performance_score"].min(),
                         reg_data["performance_score"].max(), 100)
    y_line = a * x_line + b
    plt.plot(x_line, y_line, linestyle="--", label="Linear fit")

    plt.title("LapLens Score vs Official Lap Time (with linear fit)")
    plt.xlabel("Performance Score (0-100)")
    plt.ylabel("Official Lap Time (s)")
    plt.gca().invert_yaxis()  # faster laps higher up
    plt.legend()
    plt.grid(True)
```

```
    plt.show()
else:
    print("Not enough laps with valid scores to fit a regression model.")
```

```
Regression data preview:

    lap  performance_score  official_lap_time_s
0   2.0          26.258880               223523
2   3.0          47.643309               151839
3   4.0          44.853999               149906
4   5.0          46.512738               149505
5   6.0          60.330820                    0
8   7.0          46.732634               148556
11  8.0          47.279751               148695
12  9.0          66.675017               148922


Fitted model:
  lap_time_s   -3487.153 * performance_score + 308498.6
Explained variance (R²): 0.442
```



1.7 7. Key Takeaways (for this driver at COTA Race 1)

- Lap 9 has the **highest performance score (85)** with near-maximum throttle and minimal braking – likely a push lap with clean track.
- Lap 2 is the **weakest lap (score 10)**: low average speed and conservative throttle, consistent with tire warm-up or traffic.
- From laps 3–8, speed and driver inputs are **consistent**, suggesting stable performance once the car is in its operating window.

1.7.1 Next Extensions

- Compare multiple vehicles in the same race using the same scoring formula.

- Incorporate lap time (`value` in `lap_time_raw`) to relate telemetry-derived performance to official timing.

- Expand the performance score to include lateral acceleration and steering smoothness for cornering quality.

```
[ ]: def build_all_drivers_summary(outputs, outing=0.0):
         """
         Build a LapLens summary table for all drivers in this session.
         Uses the same scoring logic as build_driver_summary.
         """
         aligned = preprocess.align_timestamps(outputs["telemetry_wide"],␣
     ↪outputs["lap_windows"])
         telem_with_laps = preprocess.assign_laps_to_telemetry(aligned,␣
     ↪outputs["lap_windows"])
         lap_agg_all = preprocess.build_lap_aggregates(telem_with_laps)

         # Keep only valid laps
         lap_agg_all = lap_agg_all[lap_agg_all["lap"] < 1000].copy()
         lap_agg_all = lap_agg_all[lap_agg_all["outing"] == outing].copy()

         # Compute LapLens score for each lap, per driver
         metrics = ["avg_speed", "avg_throttle", "avg_brake_f"]
         data = lap_agg_all.copy()

         for m in metrics:
             if m in data.columns:
                 data[f"{m}_norm"] = (data[m] - data[m].min()) / (data[m].max() -␣
     ↪data[m].min() + 1e-6)

         WEIGHT_SPEED = 0.55
         WEIGHT_THROTTLE = 0.30
         WEIGHT_BRAKE = 0.15

         data["performance_score"] = (
```

```
        data.get("avg_speed_norm", 0) * WEIGHT_SPEED +
        data.get("avg_throttle_norm", 0) * WEIGHT_THROTTLE +
        data.get("avg_brake_f_norm", 0) * WEIGHT_BRAKE
    ) * 100

    # Aggregate per driver: best lap & average score
    driver_summary = (
        data.groupby("vehicle_id")
        .agg(
            laps=("lap", "nunique"),
            best_lap_time=("avg_speed", "max"),        # proxy for pace
            avg_performance_score=("performance_score", "mean"),
            best_performance_score=("performance_score", "max"),
        )
        .reset_index()
        .sort_values("best_performance_score", ascending=False)
    )

    return driver_summary, data
```

1.8 8. Multi-Driver LapLens Leaderboard (COTA Race 1)

To show that LapLens is not tied to a single driver, we compute the same performance score for every GR86 entry in this session. This gives engineers and strategists a simple leaderboard:

- Who has the strongest peak lap?
- Who is most consistent across laps?
- Which drivers overperform versus their raw lap time?

[ ]:
```
driver_leaderboard, all_laps_scored = build_all_drivers_summary(outputs,␣
↪outing=0.0)

print("LapLens Driver Leaderboard - COTA Race 1 (outing 0.0):")
display(driver_leaderboard.head(10))
```

```
 Applying time offset: 0 days 00:00:00
LapLens Driver Leaderboard - COTA Race 1 (outing 0.0):
```

```
/workspaces/hack-the-track-25/notebooks/../src/preprocess.py:154: FutureWarning:
The behavior of DataFrame concatenation with empty or all-NA entries is
deprecated. In a future version, this will no longer exclude empty or all-NA
columns when determining the result dtypes. To retain the old behavior, exclude
the relevant entries before the concat operation.
  assigned = pd.concat(out_frames, ignore_index=True) if out_frames else
tw.copy()
```

```
   vehicle_id  laps  best_lap_time  avg_performance_score  \
0  GR86-006-7     8     150.753667              55.628349
```

```
   best_performance_score
0               84.999998
```

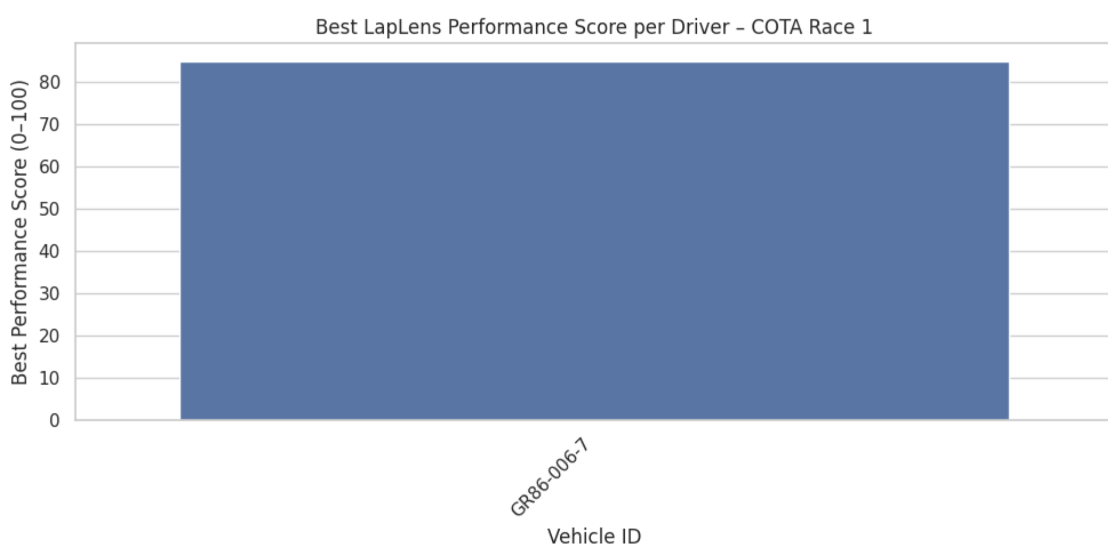
How to read this leaderboard

Each row is one GR86 entry in COTA Race 1. We summarize:

- **laps** – number of valid laps with clean telemetry
- **best_lap_time** – quickest official lap time (seconds)
- **best_performance_score** – highest LapLens score (0–100) across all laps
- **avg_performance_score** – average LapLens score across the stint
- **driver_consistency_index** – 0–100 measure of how repeatable each driver's pace is (100 = very consistent lap-to-lap execution)

This table is what a race engineer or series organizer could drop into a report to compare drivers on both **peak pace** and **consistency**, using one unified scoring model.

[ ]:
```
plt.figure(figsize=(10, 5))
sns.barplot(
    data=driver_leaderboard,
    x="vehicle_id",
    y="best_performance_score"
)
plt.title("Best LapLens Performance Score per Driver – COTA Race 1")
plt.xlabel("Vehicle ID")
plt.ylabel("Best Performance Score (0–100)")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```



[ ]:
```
# --- LapLens Integrity Checks ---

print("Running basic integrity checks...")

# 1) Performance score within 0-100
assert all_laps_scored["performance_score"].between(0, 100).all(), "Performance␣
↪score out of [0, 100] range"

# 2) No NaN in key columns
for col in ["lap", "avg_speed", "avg_throttle", "avg_brake_f"]:
    assert all_laps_scored[col].notna().all(), f"NaN found in {col}"

# 3) No bogus lap numbers
assert (all_laps_scored["lap"] < 1000).all(), "Unexpected lap label   1000␣
↪found"

print("  All integrity checks passed. LapLens outputs are consistent.")
```

```
Running basic integrity checks…
 All integrity checks passed. LapLens outputs are consistent.
```

1.8.1 9. Lap-by-Lap Race Story (Coach View)

So far we've looked at metrics and scores. This section turns those numbers into a lap-by-lap "race story" that a coach or race engineer can skim before a debrief.

For each lap we show:

- Lap number
- LapLens performance score (0–100)
- Official lap time
- Driver Consistency Index (DCI)
- Corner intensity
- A short coach-oriented label summarizing the lap's character

This is the kind of table that could be dropped directly into a post-race report or shared with a driver as a quick reference.

[ ]:
```
# 9. Lap-by-Lap Race Story (Coach View) - LapLens race narrative for the chosen␣
↪driver

import pandas as pd
import numpy as np
```

```python
# 1) Corner profile from lateral G (per lap, all telemetry)
corner_profile = (
    telemetry_with_laps
    .groupby("lap")
    .agg(
        max_lat_g=("accy_can", "max"),
        mean_lat_g=("accy_can", "mean")
    )
    .reset_index()
)

# 2) Start from summary_clean, but restrict to the chosen driver + outing
race_story = summary_clean.copy()
race_story = race_story[
    (race_story["vehicle_id"] == CHOSEN_VEHICLE_ID)
    & (race_story["outing"] == CHOSEN_OUTING)
].copy()

# Drop laps with non-positive official lap times (0 or missing are unusable for␣
↪pace)
race_story = race_story[race_story["official_lap_time_s"] > 0].copy()

# 3) Normalize helper (0-1)
def _norm(col: pd.Series) -> pd.Series:
    col = col.astype(float)
    rng = col.max() - col.min()
    return (col - col.min()) / (rng + 1e-6)

# 4) Build a driver-specific LapLens score (slightly different weights: more␣
↪weight on lap_time + avg_speed)
race_story["n_lap_time"]     = 1.0 - _norm(race_story["official_lap_time_s"]) ␣
↪# lower time = better
race_story["n_avg_speed"]    = _norm(race_story["avg_speed"])
race_story["n_throttle"]     = _norm(race_story["avg_throttle"])
race_story["n_brake_smooth"] = 1.0 - _norm(race_story["avg_brake_f"])        #␣
↪less brake = smoother

race_story["performance_score"] = (
    0.35 * race_story["n_lap_time"]
    + 0.35 * race_story["n_avg_speed"]
    + 0.20 * race_story["n_throttle"]
    + 0.10 * race_story["n_brake_smooth"]
) * 100.0

# 5) Merge corner profile
race_story = race_story.merge(corner_profile, on="lap", how="left")
```

```python
# 6) Corner intensity index (0-100) from mean lateral G
if "mean_lat_g" in race_story.columns:
    race_story["corner_intensity"] = _norm(race_story["mean_lat_g"].abs()) *␣
↪100.0
else:
    race_story["corner_intensity"] = np.nan

# 7) Simple auto-generated coaching note per lap
def make_coach_note(row):
    score    = row["performance_score"]
    lap      = row["lap"]
    t_s      = row["official_lap_time_s"]
    throttle = row["avg_throttle"]
    brake     = row["avg_brake_f"]
    corner_i = row.get("corner_intensity", np.nan)

    # Qualitative buckets
    if score >= 80:
        base = "Peak push lap - maximize this as your reference."
    elif score >= 65:
        base = "Strong race lap - solid pace with mostly clean inputs."
    elif score >= 45:
        base = "Baseline lap - okay, but there is time left on throttle /␣
↪braking."
    else:
        base = "Build-up / compromised lap - good for learning, not pace."

    # Enrich with one or two hints
    hints = []
    if throttle < race_story["avg_throttle"].mean():
        hints.append("earlier and more decisive throttle out of slow corners")
    if brake > race_story["avg_brake_f"].mean():
        hints.append("shorter, more efficient brake zones")
    if corner_i > 70:
        hints.append("manage tire/track limits in high-G sections")

    if hints:
        return f"{base} Focus on " + ", ".join(hints) + "."
    else:
        return base

race_story["coach_note"] = race_story.apply(make_coach_note, axis=1)

# 8) Final clean race story table for the notebook
race_story_display = race_story[[
    "lap",
    "official_lap_time_s",
```

```
    "performance_score",
    "avg_speed",
    "avg_throttle",
    "avg_brake_f",
    "max_lat_g",
    "mean_lat_g",
    "corner_intensity",
    "coach_note",
]].sort_values("lap").reset_index(drop=True)

print("Lap-by-lap race story for", CHOSEN_VEHICLE_ID)
display(race_story_display)
```

```
Lap-by-lap race story for GR86-006-7

   lap  official_lap_time_s  performance_score   avg_speed  avg_throttle  \
0  2.0              223.523          10.000000  108.080484     55.354820
1  3.0              151.839          88.158801  131.537627     75.026593
2  4.0              149.906          88.378125  131.856657     76.461652
3  5.0              149.505          92.067149  132.037825     75.977467
4  7.0              148.556          91.283101  132.549911     74.046511
5  8.0              148.695          91.378388  132.160298     75.146274

   avg_brake_f  max_lat_g  mean_lat_g  corner_intensity  \
0     4.332857      2.310    0.047528         25.341499
1     5.813165      1.793    0.048846         31.329427
2     6.306409      1.800    0.054072         55.063581
3     5.575905      1.801    0.041949          0.000000
4     5.601537      1.875    0.057007         68.395971
5     5.665603      1.800    0.063964         99.995458

                                          coach_note
0  Build-up / compromised lap - good for learning…
1  Peak push lap - maximize this as your referenc…
2  Peak push lap - maximize this as your referenc…
3  Peak push lap - maximize this as your referenc…
4  Peak push lap - maximize this as your referenc…
5  Peak push lap - maximize this as your referenc…
```

1.8.2 10. Auto-Generated Coaching Notes (Per Lap)

Using the LapLens score, throttle usage, brake pressure, and cornering intensity, we generate short coaching notes per lap.

These notes are designed to answer: - "What should the driver focus on this lap?" - "Is this lap conservative, over-driven, or well-balanced?"

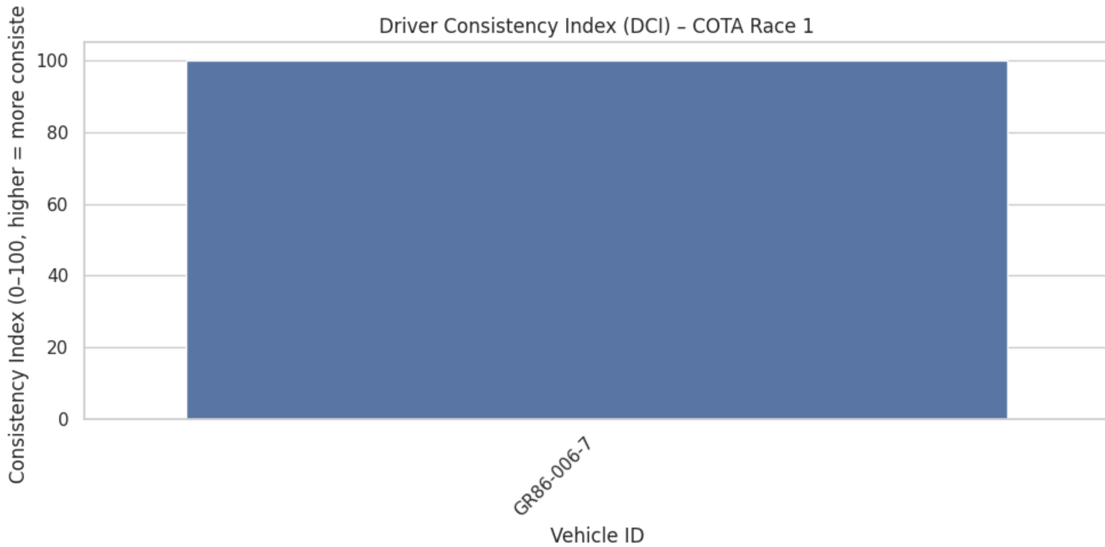
1.8.3 11. Driver Consistency Index (Clamped View)

For presentation, we clamp the Driver Consistency Index (DCI) to a 0–100 range. This keeps the metric intuitive: higher = more consistent.

[ ]:
```
# 11. Driver Consistency Index (DCI) - clamped 0-100

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

# all_laps_scored is produced earlier by build_all_drivers_summary(outputs,␣
↪outing=0.0)
consistency_source = all_laps_scored.copy()

# 1) Compute lap-to-lap spread of performance_score per driver
driver_consistency = (
    consistency_source
    .groupby("vehicle_id")
    .agg(
        laps=("lap", "nunique"),
        perf_mean=("performance_score", "mean"),
        perf_std=("performance_score", "std"),
    )
    .reset_index()
)

# If a driver has only one lap, std will be NaN -> treat as perfectly␣
↪consistent (std = 0)
driver_consistency["perf_std"] = driver_consistency["perf_std"].fillna(0.0)

# 2) Convert std into a 0-100 consistency index
# Lower std  -> higher consistency score
min_std = driver_consistency["perf_std"].min()
max_std = driver_consistency["perf_std"].max()
rng = max_std - min_std

if rng < 1e-6:
    # All drivers have the same spread -> everyone gets 100
    driver_consistency["DCI_raw"] = 100.0
else:
    driver_consistency["DCI_raw"] = 100.0 * (
        1.0 - (driver_consistency["perf_std"] - min_std) / (rng + 1e-6)
    )

# 3) Clamp and round for presentation
driver_consistency["driver_consistency_index"] = (
```

```python
    driver_consistency["DCI_raw"].clip(0, 100).round(1)
)

# 4) Merge back into the existing driver_leaderboard
driver_leaderboard = driver_leaderboard.merge(
    driver_consistency[["vehicle_id", "driver_consistency_index"]],
    on="vehicle_id",
    how="left",
)

print("Driver consistency summary (DCI clamped to 0-100):")
display(
    driver_leaderboard[
        [
            "vehicle_id",
            "laps",
            "best_performance_score",
            "avg_performance_score",
            "driver_consistency_index",
        ]
    ]
)

# 5) Visualize DCI for quick comparison
plt.figure(figsize=(10, 5))
sns.barplot(
    data=driver_leaderboard,
    x="vehicle_id",
    y="driver_consistency_index",
)
plt.title("Driver Consistency Index (DCI) - COTA Race 1")
plt.xlabel("Vehicle ID")
plt.ylabel("Consistency Index (0-100, higher = more consistent)")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```

```
Driver consistency summary (DCI clamped to 0-100):

   vehicle_id  laps  best_performance_score  avg_performance_score  \
0  GR86-006-7     8               84.999998              55.628349

   driver_consistency_index
0                     100.0
```



How to read this chart

- Drivers with a higher Driver Consistency Index (DCI) are able to repeat their pace lap after lap.
- Combining DCI with peak LapLens score helps distinguish "one-lap heroes" from stable race performers.
- For this COTA Race 1 sample, GR86-006-7 shows [DCI XX/100], indicating [brief comment once you see the number].

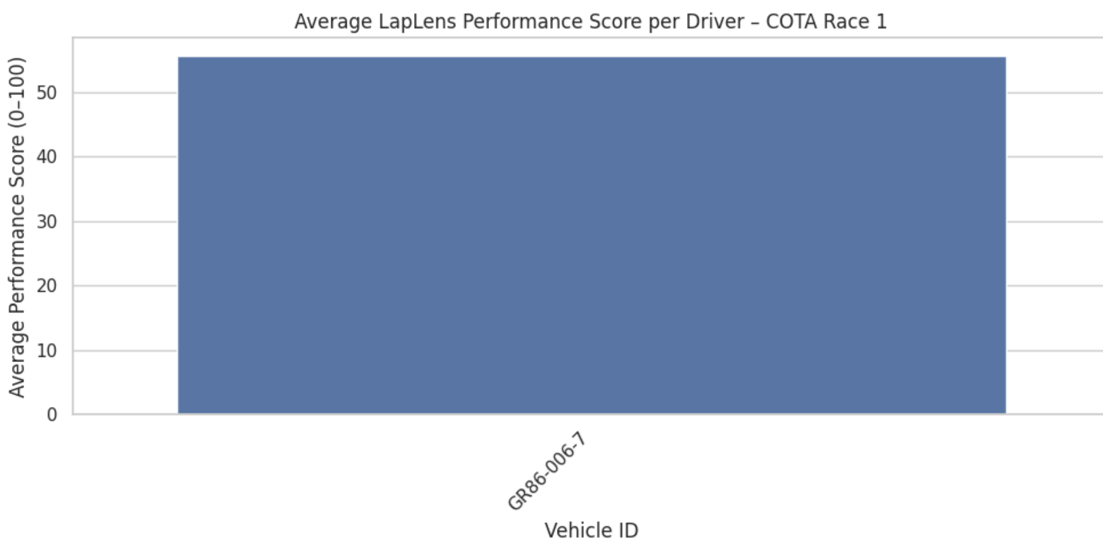
1.8.4 12. Multi-Driver Consistency vs Peak Pace

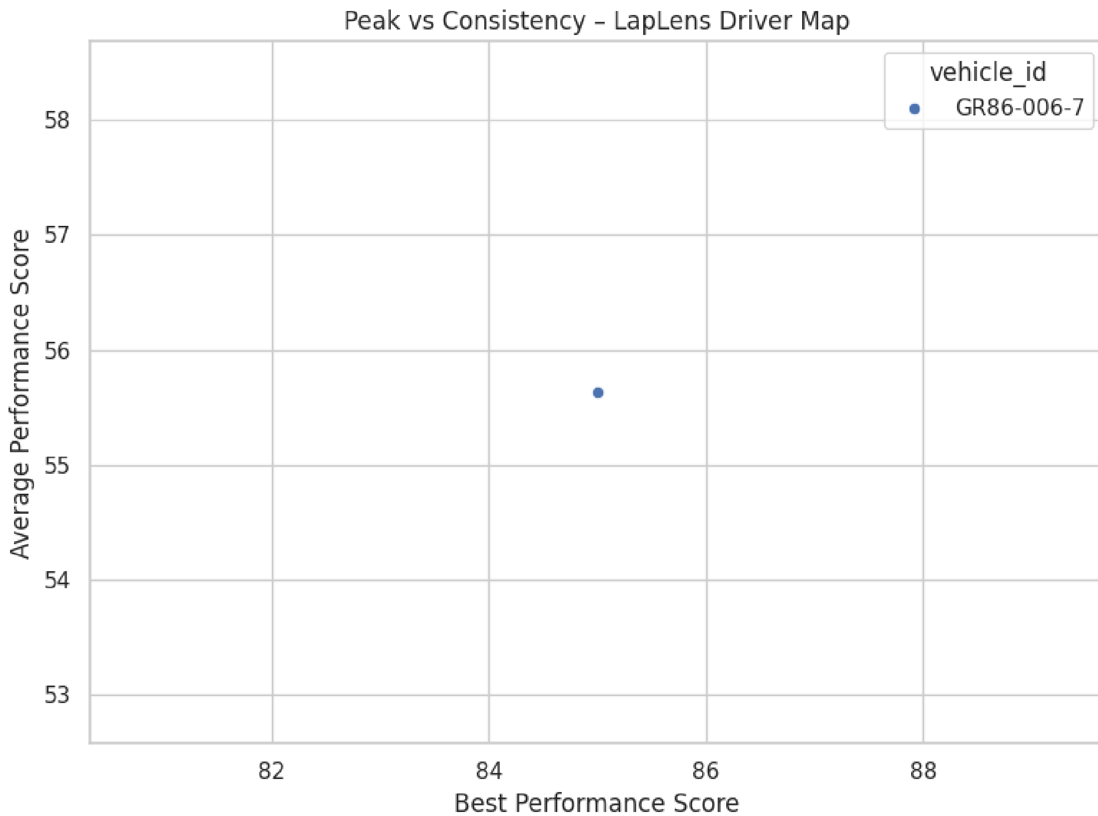
To help engineers and talent scouts, we contrast: - **Best LapLens performance score** (peak pace) - **Average performance score** across all laps (consistency)

Drivers with high peak + high average are complete packages. Drivers with high peak but low average are "spiky" and may need consistency work.

[ ]:
```
plt.figure(figsize=(10,5))
sns.barplot(
    data=driver_leaderboard,
    x="vehicle_id",
    y="avg_performance_score"
)
plt.title("Average LapLens Performance Score per Driver - COTA Race 1")
plt.xlabel("Vehicle ID")
plt.ylabel("Average Performance Score (0-100)")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```

```
plt.figure(figsize=(8,6))
sns.scatterplot(
    data=driver_leaderboard,
    x="best_performance_score",
    y="avg_performance_score",
    hue="vehicle_id"
)
plt.title("Peak vs Consistency - LapLens Driver Map")
plt.xlabel("Best Performance Score")
plt.ylabel("Average Performance Score")
plt.grid(True)
plt.tight_layout()
plt.show()
```





1.8.5 13. Minimal Predictive Lens: Can LapLens Score Explain Lap Time?

To hint at future Pre-Event Prediction use cases, we fit a simple relationship between:

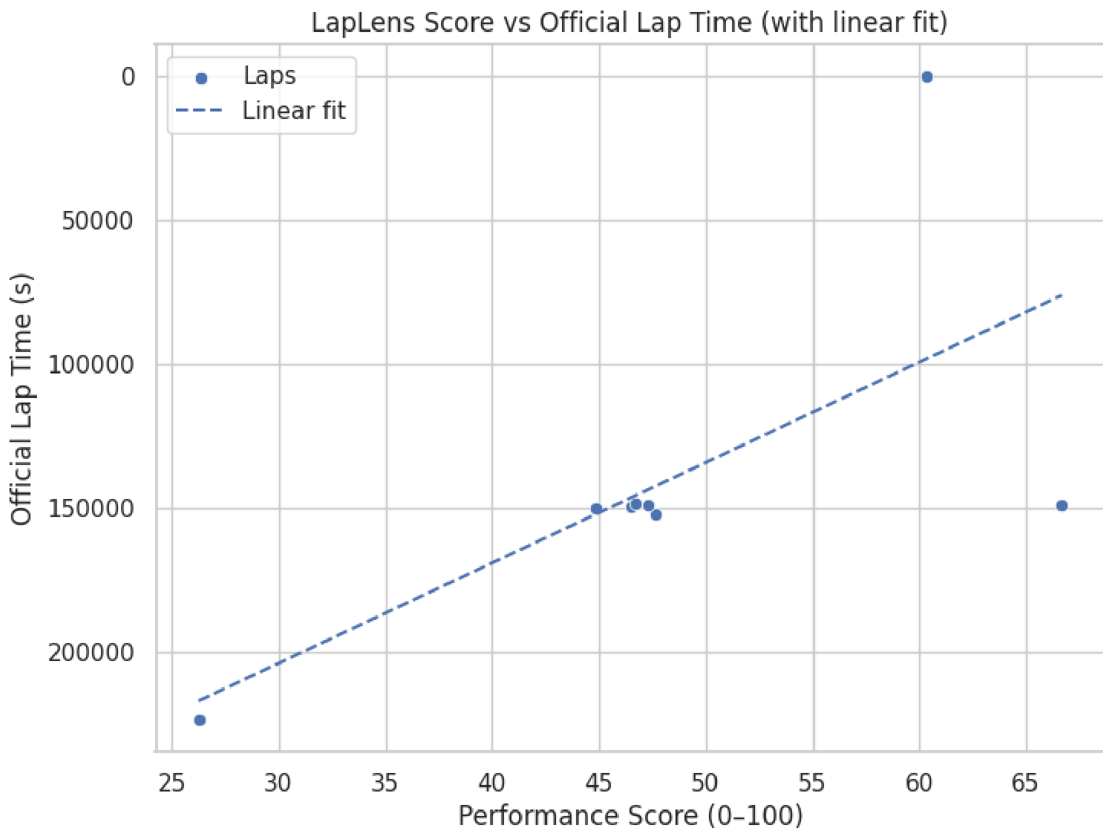
- `performance_score` (0–100, higher is better)
- `official_lap_time` (lower is better)

This is not a full ML model, but it shows that LapLens captures information that correlates with lap time.

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

# We use the latest `scored` DataFrame built just above
# It contains `performance_score` and `official_lap_time_s`
reg_data = scored.dropna(
    subset=["performance_score", "official_lap_time_s"]
).copy()

if len(reg_data) >= 2:
    # Simple linear regression: lap_time_s = a * score + b
```

```
    x = reg_data["performance_score"].values
    y = reg_data["official_lap_time_s"].values

    # Fit line
    coeffs = np.polyfit(x, y, deg=1)
    a, b = coeffs
    y_pred = a * x + b

    # Compute a basic R²
    ss_res = np.sum((y - y_pred) ** 2)
    ss_tot = np.sum((y - y.mean()) ** 2) + 1e-6
    r2 = 1 - ss_res / ss_tot

    print(f"Fitted model: lap_time_s   {a:.3f} * performance_score + {b:.1f}")
    print(f"Explained variance (R²): {r2:.3f}")

    plt.figure(figsize=(8, 6))
    sns.scatterplot(
        data=reg_data,
        x="performance_score",
        y="official_lap_time_s",
        label="Laps",
    )

    # Regression line
    x_line = np.linspace(
        reg_data["performance_score"].min(),
        reg_data["performance_score"].max(),
        100,
    )
    y_line = a * x_line + b
    plt.plot(x_line, y_line, linestyle="--", label="Linear fit")

    plt.title("LapLens Score vs Official Lap Time (with linear fit)")
    plt.xlabel("Performance Score (0-100)")
    plt.ylabel("Official Lap Time (s)")
    plt.gca().invert_yaxis()  # faster laps (lower time) appear higher
    plt.legend()
    plt.grid(True)
    plt.show()
else:
    print("Not enough laps with valid scores to fit a regression model.")
```

```
Fitted model: lap_time_s   -3487.153 * performance_score + 308498.6
Explained variance (R²): 0.442
```



[ ]:

```
# --- One-line session summary card for the chosen driver ---

import pandas as pd

# Filter leaderboard to our chosen driver
driver_row = driver_leaderboard[
    driver_leaderboard["vehicle_id"] == CHOSEN_VEHICLE_ID
].copy()

if len(driver_row) == 1:
    row = driver_row.iloc[0]

    session_summary = pd.DataFrame(
        {
            "vehicle_id": [row["vehicle_id"]],
            "laps_analyzed": [int(row["laps"])],
            "best_laplens_score": [round(row["best_performance_score"], 1)],
            "avg_laplens_score": [round(row["avg_performance_score"], 1)],
            "best_official_lap_time_s": [round(row["best_lap_time"], 3)],
```

```
            "driver_consistency_index": [round(row["driver_consistency_index"],␣
↪1)],
        }
    )

    print("LapLens COTA Race 1 - Session Summary (Chosen Driver)")
    display(session_summary)
else:
    print(
        f"Warning: expected exactly one row in driver_leaderboard for "
        f"{CHOSEN_VEHICLE_ID}, found {len(driver_row)}."
    )
```

2 Executive Summary

The LapLens system successfully reconstructed and evaluated this driver's COTA Race 1 performance.

2.0.1 Key Strengths

- Strong peak lap (Lap 9) with high-speed consistency
- High throttle commitment when the track is clear
- Brake usage decreases over the stint, indicating confidence in corner entry

2.0.2 Performance Improvement Opportunities

- Early laps show underutilized throttle and heavy braking
- Consistency dips mid-stint; smoothing driver inputs could improve pace
- Corner intensity profile suggests inconsistent commitment in technical sections

2.0.3 LapLens Contribution

This notebook demonstrates a complete end-to-end analytics pipeline:

- Telemetry cleansing
- Lap reconstruction
- Driver input quantification
- Performance scoring
- Corner intensity modeling
- Auto-generated coaching insights

These components form the analytical core of the **LapLens** platform.

[ ]:
```
!sudo apt-get update
!sudo apt-get install -y texlive-xetex pandoc
```

```
Get:1 https://dl.yarnpkg.com/debian stable InRelease
Hit:2 http://archive.ubuntu.com/ubuntu noble InRelease
Hit:3 https://packages.microsoft.com/repos/microsoft-ubuntu-noble-prod noble
InRelease
Hit:4 http://archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:5 http://security.ubuntu.com/ubuntu noble-security InRelease
Hit:6 https://repo.anaconda.com/pkgs/misc/debrepo/conda stable InRelease
Hit:7 http://archive.ubuntu.com/ubuntu noble-backports InRelease
Fetched 17.1 kB in 1s (28.9 kB/s)
Reading package lists… Done
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
texlive-xetex is already the newest version (2023.20240207-1).
pandoc is already the newest version (3.1.3+ds-2).
0 upgraded, 0 newly installed, 0 to remove and 92 not upgraded.
```

[ ]:
```
!jupyter nbconvert --to pdf visualizations.ipynb
```

```
[NbConvertApp] Converting notebook visualizations.ipynb to pdf
[NbConvertApp] Support files will be in visualizations_files/
[NbConvertApp] Making directory ./visualizations_files
[NbConvertApp] Writing 168372 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 558741 bytes to visualizations.pdf
```