



Foto von Kolby Milton auf Unsplash

Analyse und Clustering von Rundenzeiten beim 24-Stunden-Rennen am Nürburgring 2023

Angewandte Programmierung - SoSe 2023 - FOM - Köln

Sonstige Beteiligung, 2. Juni 2023

Patrick Haas

Inhalt

- Business Understanding
 - Forschungsfrage
 - Datenherkunft
- Data Understanding/Data Preparation
 - Data Wrangling
 - Datenexploration
- Modeling
- Evaluation
 - Kritische Würdigung
 - Ausblick
- Referenzen

```
In [1]:  
import math  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
import seaborn as sns  
import suncalc  
  
from sklearn import cluster, mixture  
  
_DARK_MODE = False
```

Business Understanding

Am langen Wochenende vom 18.5.-21.5.2023 fand das *51. 24-Stunden-Rennen auf dem Nürburgring* statt [1]. Neben Kirmesatmosphäre rund um die Strecke konnte erstmals ein Ferrari das Rennen für sich entscheiden (und damit auch das erste Mal seit 2002 kein deutsches Fabrikat).

Rund um das Rennen kam in Berichterstattung und Interviews immer wieder die Behauptung auf, dass die anderen zum Gesamtsieg fähigen Fahrzeuge gegenüber dem Ferrari durch die *Balance of Performance (BoP)* benachteiligt seien. Die BoP ist ein Verfahren, das für die Fahrzeuge unterschiedliche Mindestgewichte, Luftmengenbegrenzungen, Ladedruckkurven und Tankvolumina vorgibt, um alle Kandidaten trotz unterschiedlicher Fahrzeugkonzepte (Saug-/Turbomotoren, Motorplatzierung) möglichst chancengleich starten zu lassen. Sie wird vor jedem Rennen tagesaktuell durch die Rennleitung festgelegt, um die Chancengleichheit zu wahren [2].

Forschungsfrage

Lässt sich diese Behauptung durch die erreichten Runden- und Sektorzeiten der einzelnen Fahrzeuge fundieren? Wurden die Ferrari-Fahrzeuge tatsächlich durch die BoP bevorzugt?

Datenherkunft

Die Datenbasis ist die offizielle Zeitnahme des Rennens, die vom Veranstalter auf dem Teilnehmerportal des 24-Stunden-Rennens zur Verfügung gestellt wird [3].

Data Understanding/Data Preparation



Darstellung via Midjourney AI

Data Wrangling

Auf den ersten Blick wirken die Zeitnahmedaten, als ob sie aus einer über lange Zeit gewachsenen Excel-Applikation exportiert wurden. Um den Datensatz erkundbar zu machen, war an dieser Stelle jede Menge Iteration zwischen Data Understanding und Data Preparation erforderlich.

Hier die Highlights des zeitraubenden Iterationsprozesses:

1. Die CSV-Dateien werden im ISO 8859-15-Encoding zur Verfügung gestellt. Um sowohl mit dem Editor als auch mit Python beschwerdefrei arbeiten zu können, konvertieren wir sie mit `iconv` nach UTF-8:

```
for I in *.csv ; do iconv -f ISO-8859-15 -t UTF-8 -o $I.temp $I ; mv -f $I.temp $I ; done
```

1. Alle Rundenzeiten sind in einem Kurzformat angegeben, das grob `HH:MM:ss.SSS` entspricht. Die Behandlung von führenden Nullen ist inkonsistent und Nullwerte werden meist komplett weggelassen, was die Verwendung von Python- oder pandas-Bordmitteln zum Parsing erschwert. Wir übergeben `pd.read_csv()` daher eine Konverterfunktion, die solche Zeiten in Millisekunden umrechnet:

```
In [2]: def timing_to_milliseconds(time: str) -> np.int64 | float:  
    """Converts a timing string to milliseconds."""  
    seconds = minutes = hours = microseconds = np.int64(0)  
  
    match time.split("."):  
        case [r, ms]:  
            rest, milliseconds = r, np.int64(ms)  
        case _:  
            return np.nan  
  
    match rest.split(":"):  
        case [h, m, s]:  
            hours = np.int64(h)  
            minutes = np.int64(m)  
            seconds = np.int64(s)  
        case [m, s]:  
            minutes = np.int64(m)  
            seconds = np.int64(s)  
        case [s]:  
            seconds = np.int64(s)  
        case _:  
            return np.nan  
  
    return hours * 3600000 + minutes * 60000 + seconds * 1000 + milliseconds
```

1. Einige Strings müssen von Leerzeichen befreit werden. Wir verwenden `str.strip` als Konverterfunktion

1. Die Sektordaten enthalten die Information, ob eine gegebene Sektorzeit zum entsprechenden Zeitpunkt eine Bestzeit war (`SEKTOR{n}_BESTE_ZEIT`). Diese booleschen Werte sind als 'J' und 'N' kodiert. `pd.read_csv()` bietet zwar die Möglichkeit, über `true_values` und `false_values` Zeichenketten für Wahrheitswerte zu definieren, aber das führt dazu, dass fehlende Werte in anderen Spalten falsch interpretiert werden. Wir verwenden auch hierfür eine eigene Konverterfunktion.

```
In [3]: def read_sector_times_csv(_source: str | list[str]) -> pd.DataFrame:  
    source = []  
    if type(_source) == str:  
        source = [_source]  
    elif type(_source) == list:  
        source = _source
```

```

else:
    raise ValueError("Invalid source type. Must be str or list[str]")

timing_converters = {
    feature: timing_to_milliseconds
    for feature in [
        "RUNDENZEIT",
        "THEORETISCHE_BESTZEIT",
        "DURCHGANGSZEIT",
        "SEKTOR1_ZEIT",
        "SEKTOR2_ZEIT",
        "SEKTOR3_ZEIT",
        "SEKTOR4_ZEIT",
        "SEKTOR5_ZEIT",
        "SEKTOR6_ZEIT",
        "SEKTOR7_ZEIT",
        "SEKTOR8_ZEIT",
        "SEKTOR9_ZEIT",
        "THEORETISCHE_BESTZEIT",
    ]
}

string_converters = {
    feature: str.strip
    for feature in ["KLASSE", "KLASSESORT", "UNTERKLASSE", "KLASSEKURZ"]
}

bool_converters = {
    feature: lambda x: True if x == "J" else False
    for feature in [
        "SEKTOR1_BESTE_ZEIT",
        "SEKTOR2_BESTE_ZEIT",
        "SEKTOR3_BESTE_ZEIT",
        "SEKTOR4_BESTE_ZEIT",
        "SEKTOR5_BESTE_ZEIT",
        "SEKTOR6_BESTE_ZEIT",
        "SEKTOR7_BESTE_ZEIT",
        "SEKTOR8_BESTE_ZEIT",
        "SEKTOR9_BESTE_ZEIT",
    ]
}

return pd.concat(
    [
        pd.read_csv(
            f,
            delimiter=";",
            # can't use true_values and false_values or
            # some missing sector data gets parsed as boolean
            # true_values=["J"],
            # false_values=["N"],
            converters={
                **timing_converters,
                **string_converters,
                **bool_converters,
            },
            dtype={"FAHRER3_ORT": str},
        )
        for f in source
    ]
)

```

1. Geschmackssache, aber da wir gerade dabei sind: Die in den CSV-Dateien enthaltenen Spaltentitel sind großgeschrieben und auf Deutsch. Wir ersetzen sie durch kleingeschriebene englische Titel:

```
In [4]: def rename_sector_times_columns(df: pd.DataFrame) -> pd.DataFrame:
    header_mapping = {
        "STNR": "car_number",
        "KUERZEL": "abbreviation",
        "FAHRER_NR": "driver_number",
        # generate mappings for FAHRER{N}
        "NATION": "team_nationality",
        "BEWERBER": "team",
        "TEAM": "team_category",
        "FAHRZEUG": "car",
        "RUNDE_NR": "lap_number",
        "DURCHGANGSZEIT": "elapsed_time",
        "TAGESZEIT": "time_of_day",
        "RUNDENZEIT": "lap_time",
        "RUNDENZEIT_IN_SEKUNDEN": "lap_time_seconds",
        "DIESCHNELLSTE": "is_fastest",
        "KLASSE": "class",
        "KLASSESORT": "class_sort",
        "UNTERKLASSE": "subclass",
        "KLASSEKURZ": "class_short",
        # generate mapping for SEKTOR{N}
        "TOPSPEED_KMH": "top_speed_kmh",
        "TOPSPEED_BESTE_SPEED": "top_speed_best_speed",
        "TOPSPEED_KMH2": "top_speed_kmh2",
        "TOPSPEED_BESTE_SPEED2": "top_speed_best_speed2",
        "TOPSPEED_KMH3": "top_speed_kmh3",
        "TOPSPEED_BESTE_SPEED3": "top_speed_best_speed3",
        "SEKTOR_BESTE_SUMME": "sector_best_sum",
        "RANG": "rank",
        "INPIT": "in_pit",
        "CANCELLED": "cancelled",
        "PITSTOPDURATION": "pitstop_duration",
        "BEWERBERLIZENZ": "applicant_license",
        "PITIN_TIME": "pitin_time",
        "RUNDE_NR_TEXT": "lap_number_text",
        "WET": "wet",
        "PRO": "pro",
        "PROAM": "pro_am",
        "PRIO": "priority",
        "STINT": "stint",
        "LAPINSTINT": "lap_in_stint",
        "THEORETISCHE_BESTZEIT": "theoretical_best_time",
    }

    # generate mapping for FAHRER{N}
    for i in range(9):
        a = f"FAHRER{i+1}"
        b = f"driver{i+1}"
        driver = {
            f"{a}_NAME": f"{b}_lastname",
            f"{a}_VORNAME": f"{b}_firstname",
            f"{a}_ORT": f"{b}_city",
            f"{a}_SPONSORLIZENZ": f"{b}_sponsor_license",
            f"{a}_SPONSOR": f"{b}_sponsor",
            f"{a}_LIZENZ": f"{b}_license",
            f"{a}_NATION": f"{b}_nationality",
            f"{a}_PUNKTEWERTUNG": f"{b}_points",
        }
        header_mapping = {**header_mapping, **driver}

    # generate mapping for SEKTOR{N}
    for i in range(10):
        a = f"SEKTOR{i+1}"
        b = f"sector"
        sector = {
            f"{a}_NAME": f"{b}_name",
            f"{a}_VORNAME": f"{b}_firstname",
            f"{a}_ORT": f"{b}_city",
            f"{a}_SPONSORLIZENZ": f"{b}_sponsor_license",
            f"{a}_SPONSOR": f"{b}_sponsor",
            f"{a}_LIZENZ": f"{b}_license",
            f"{a}_NATION": f"{b}_nationality",
            f"{a}_PUNKTEWERTUNG": f"{b}_points",
        }
        header_mapping = {**header_mapping, **sector}
```

```

sector = {
    f"{{a}}_ZEIT": f"{{b}}time_{i+1}",
    f"{{a}}_BESTE_ZEIT": f"{{b}}is_best_time_{i+1}",
    f"{{a}}_KMH": f"{{b}}speed_kmh_{i+1}",
    f"{{a}}_BESTE_SPEED": f"{{b}}is_best_speed_{i+1}",
}
header_mapping = {**header_mapping, **sector}

return df.rename(columns=header_mapping)

```

1. In der explorativen Datenanalyse könnten die jeweils aktiven Fahrer:innen pro Startnummer und Runde interessant sein. Alle möglichen Fahrer:innen einer Startnummer sind als FAHRER{n}_NAME , FAHRER{n}_VORNAME , usw. in jeder Zeile der Zeitnahmedaten aufgeführt. Die Spalte FAHRER_NR besagt, welche:r Fahrer:in gerade aktiv ist. Wir verwenden eine Funktion, um die redundanten Informationen zu entfernen und nur aktive Fahrer:innen beizubehalten:

```
In [5]: def select_current_driver(df: pd.DataFrame) -> pd.DataFrame:
    newdf = pd.DataFrame(df)
    newdf["driver_name"] = newdf.apply(
        lambda x: f"{x.iloc[3 + (x['driver_number'] - 1) * 8]}, {x.iloc[4 + (x['driver_n
    axis=1,
)

newdf["driver_license"] = newdf.apply(
    lambda x: x.iloc[8 + (x["driver_number"] - 1) * 8], axis=1
)
return newdf
```

1. Wir entfernen alle Features, die für die weitere Analyse nicht von Belang sind:

```
In [6]: def drop_unused_columns(df: pd.DataFrame) -> pd.DataFrame:
    return df.drop(
        columns=[
            "abbreviation",
            "applicant_license",
            "cancelled",
            "class_sort",
            "class",
            "driver_license",
            "driver_number",
            "driver1_city",
            "driver1_firstname",
            "driver1_lastname",
            "driver1_license",
            "driver1_nationality",
            "driver1_points",
            "driver1_sponsor_license",
            "driver1_sponsor",
            "driver2_city",
            "driver2_firstname",
            "driver2_lastname",
            "driver2_license",
            "driver2_nationality",
            "driver2_points",
            "driver2_sponsor_license",
            "driver2_sponsor",
            "driver3_city",
            "driver3_firstname",
            "driver3_lastname",
            "driver3_license",
            "driver3_nationality",
            "FAHRER_NR",
            "FAHRER_NAME",
            "FAHRER_VORNAME",
            "FahrerName"
        ]
    )
```

"driver3_points",
"driver3_sponsor_license",
"driver3_sponsor",
"driver4_city",
"driver4_firstname",
"driver4_lastname",
"driver4_license",
"driver4_nationality",
"driver4_points",
"driver4_sponsor_license",
"driver4_sponsor",
"driver5_city",
"driver5_firstname",
"driver5_lastname",
"driver5_license",
"driver5_nationality",
"driver5_points",
"driver5_sponsor_license",
"driver5_sponsor",
"driver6_city",
"driver6_firstname",
"driver6_lastname",
"driver6_license",
"driver6_nationality",
"driver6_points",
"driver6_sponsor_license",
"driver6_sponsor",
"driver7_city",
"driver7_firstname",
"driver7_lastname",
"driver7_license",
"driver7_nationality",
"driver7_points",
"driver7_sponsor_license",
"driver7_sponsor",
"driver8_city",
"driver8_firstname",
"driver8_lastname",
"driver8_license",
"driver8_nationality",
"driver8_points",
"driver8_sponsor_license",
"driver8_sponsor",
"is_fastest",
"lap_number_text",
"lap_time_seconds",
"pitin_time",
"pitstop_duration",
"priority",
"pro",
"pro_am",
"rank",
"sector_best_sum",
"sectoris_best_time_1",
"sectoris_best_speed_1",
"sectorspeed_kmh_1",
"sectoris_best_time_2",
"sectoris_best_speed_2",
"sectorspeed_kmh_2",
"sectoris_best_time_3",
"sectoris_best_speed_3",
"sectorspeed_kmh_3",
"sectoris_best_time_4",
"sectoris_best_speed_4",
"sectorspeed_kmh_4",
"sectoris_best_time_5",

```

        "sectoris_best_speed_5",
        "sectorspeed_kmh_5",
        "sectoris_best_time_6",
        "sectoris_best_speed_6",
        "sectorspeed_kmh_6",
        "sectoris_best_time_7",
        "sectoris_best_speed_7",
        "sectorspeed_kmh_7",
        "sectoris_best_time_8",
        "sectoris_best_speed_8",
        "sectorspeed_kmh_8",
        "sectoris_best_time_9",
        "sectoris_best_speed_9",
        "sectorspeed_kmh_9",
        "subclass",
        "team_category",
        "team_nationality",
        "top_speed_best_speed",
        "top_speed_best_speed2",
        "top_speed_best_speed3",
        "top_speed_kmh",
        "top_speed_kmh2",
        "top_speed_kmh3",
        "wet",
    ],
)

```

1. Dann definieren wir noch einen Index, um jede Zeile durch Startnummer und Rundennummer eindeutig zu identifizieren:

```
In [7]: def make_lap_index(df: pd.DataFrame) -> pd.DataFrame:
    return df.set_index(["car_number", "lap_number"])
```

1. Abschließend setzen wir die Puzzleteile zusammen und laden die Daten in ein DataFrame:

```
In [8]: def read_sector_times(_source: str | list[str]) -> pd.DataFrame:
    df = read_sector_times_csv(_source)
    df = rename_sector_times_columns(df)
    df = select_current_driver(df)
    df = drop_unused_columns(df)
    df = make_lap_index(df)
    return df

df_race_sectors = read_sector_times(
    [
        "data/2023/24h/race_rennen_sektorzeiten_part_1.csv",
        "data/2023/24h/race_rennen_sektorzeiten_part_2.csv",
    ]
)
df_race_sectors.sample(5)
```

		team	car	elapsed_time	time_of_day	lap_time	class_short	sectorTime_1	...
car_number	lap_number								
89	77	Haupt Racing Team	Mercedes-AMG GT4	104330802	04:58:50.802	789273	SP 10	220822.0	
182	84	SRS Team Sorg	Porsche 718	107667690	05:54:27.690	564945	Cup 3	44720.0	

Rennsport	Cayman GT4 Clubspo							
90	72	Adrenalin Motorsport Team Motec	BMW M4 GT4 (Evo2021)	102533079	04:28:53.079	674110	SP 10	56409.0
110	122	Max Kruse Racing	VW GOLF GTI TCR DSG	132961719	12:56:01.719	616370	SP 3T	45344.0
40	9	Audi Sport Team Scherer PHX	Audi R8 LMS GT3 evo II	62319478	17:18:39.478	520277	SP 9	40845.0

Die Daten sind jetzt in einem Zustand, in dem wir sie für die explorative Datenanalyse verwenden können.

Datenexploration

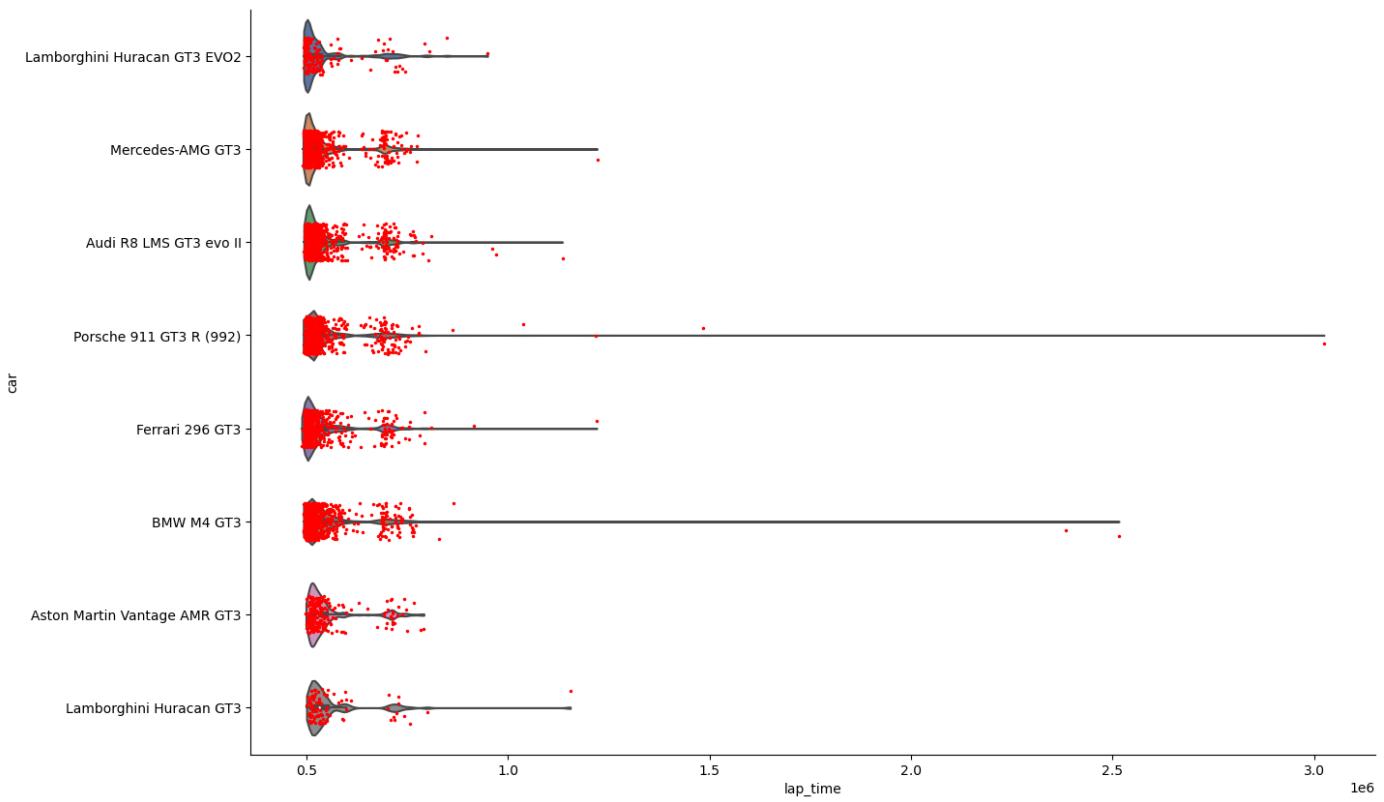
Wir sehen uns zuerst die Verteilung der Rundenzeiten in der gesamtsiegfähigen SP 9-Klasse an. Dazu filtern wir die Daten nach der Klasse, lassen Boxenrunden weg, gruppieren nach Fahrzeug und plotten die Rundenzeiten als Violinplot mit überlagerten Einzelwerten:

```
In [9]: def plot_time_distribution(
    df: pd.DataFrame, value: str = "lap_time", hue: str | None = None
) -> None:
    df = df.copy()
    df["__sort_q2"] = df.groupby("car") [value].transform(np.quantile, q=0.5)
    f, ax = plt.subplots(figsize=(15, 10))
    df.sort_values("__sort_q2", inplace=True)

    sns.violinplot(
        y=df["car"],
        x=df[value],
        bw=0.1,
        palette="deep",
        cut=0,
    )
    sns.stripplot(
        y=df["car"],
        x=df[value],
        jitter=0.2,
        size=2.5,
        color="red" if not hue else None,
        palette="bright" if hue else None,
        hue=df[hue] if hue else None,
    )
    sns.despine()
```

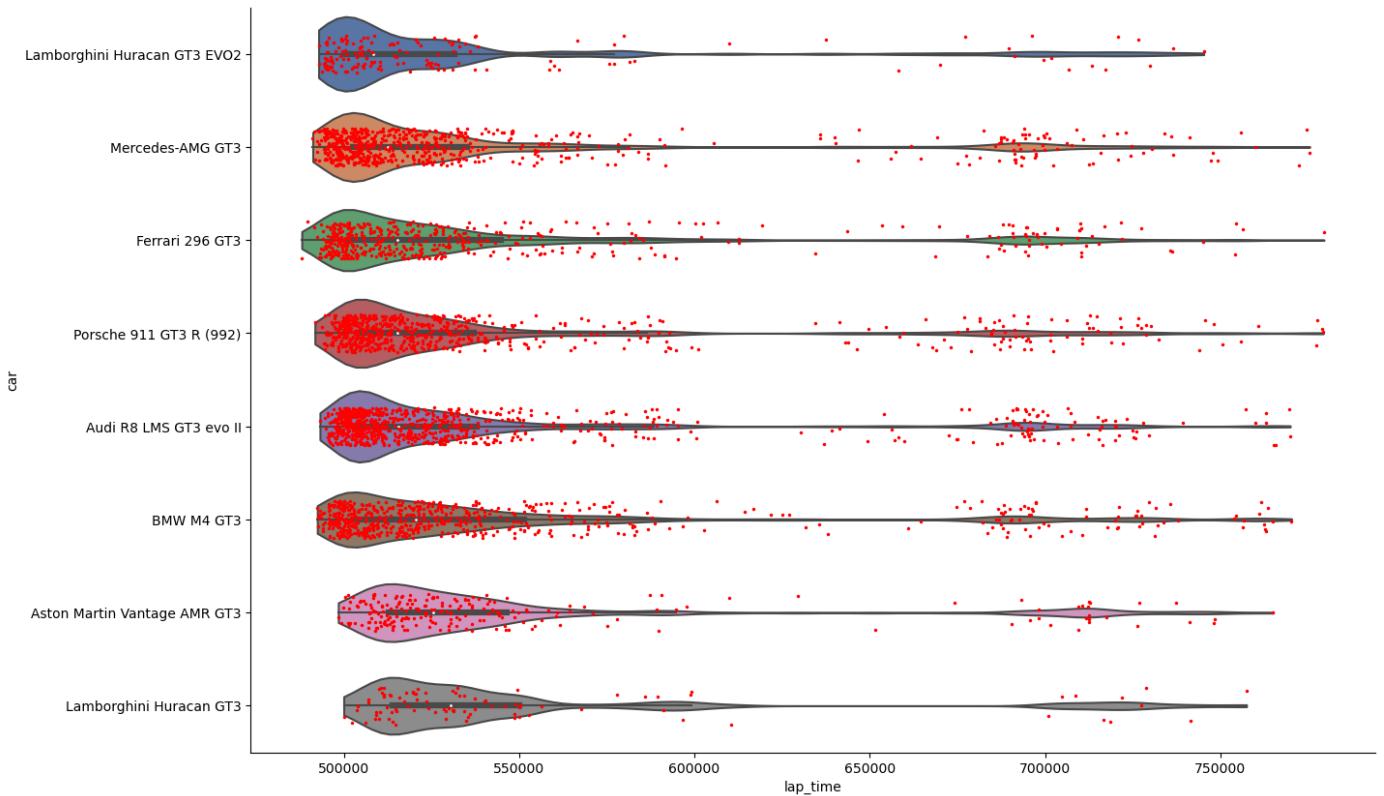
```
In [10]: df_sp9_sectors = pd.DataFrame(df_race_sectors.query("class_short == 'SP 9'"))
```

```
In [11]: plot_time_distribution(df_sp9_sectors)
```



Erster Eindruck: Einige extreme Ausreißer sorgen dafür, dass der interessante Bereich der Verteilung schwer zu erkennen ist. Wir filtern die Daten nach Rundenzeiten unter 13 Minuten:

```
In [12]: plot_time_distribution(df_sp9_sectors.query("lap_time < 780000"))
```



Auffällig ist die Q-Tip-Form. Die Rundenzeiten streuen insgesamt zwar stark, sammeln sich aber dennoch um einige Schwerpunkte.

Die Intuition aus dem Business Understanding dazu ist, dass Profifahrer:innen sehr konstante Rundenzeiten fahren können. Das bedeutet, dass die Verteilung ihrer Zeiten sich mit geringer Varianz um einen Mittelwert

bewegen und nach dem zentralen Grenzwertsatz durch eine Gauß-Verteilung annäherbar sein sollte. Allerdings gilt das nur für ungestörte Runden.

Werfen wir einen kurzen Blick auf mögliche Störgrößen:

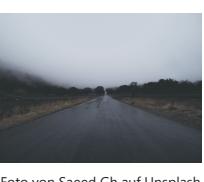
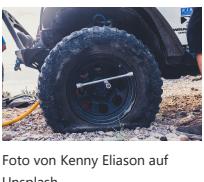


Foto von Musa Haef auf
Unsplash

Foto von david latorre romero
auf Unsplash

Foto von Kenny Eliason auf
Unsplash

Foto von Saeed Gh auf Unsplash

- **Verkehr:** Die Fahrer:innen müssen langsamere Fahrzeuge überholen
- **Gelbe Flagge:** Überholverbot bis zur Aufhebung. Die Fahrer:innen hängen im gegebenenfalls deutlich langsameren Verkehr fest
- **Doppelte gelbe Flagge:** Wie Gelb, zusätzlich mit Geschwindigkeitsbegrenzung auf 120 km/h
- **Code 60:** Wie Gelb, zusätzlich mit Geschwindigkeitsbegrenzung auf 60 km/h
- **Boxenstops:** Die Fahrer:innen steuern geplant zum Tanken, Reifenwechsel und Fahrerwechsel oder ungeplant zur Reparatur die Box an. Für Boxenstopps wird eine Mindeststandzeit vorgeschrieben
- **Defekt:** Die Fahrer:innen müssen in langsamer Fahrt zurück an die Box oder fallen endgültig aus
- **Streckenbedingungen:** Bei nasser Fahrbahn müssen die Fahrer:innen ihre Fahrweise deutlich anpassen
- **Kombinationen** aus den genannten Störgrößen

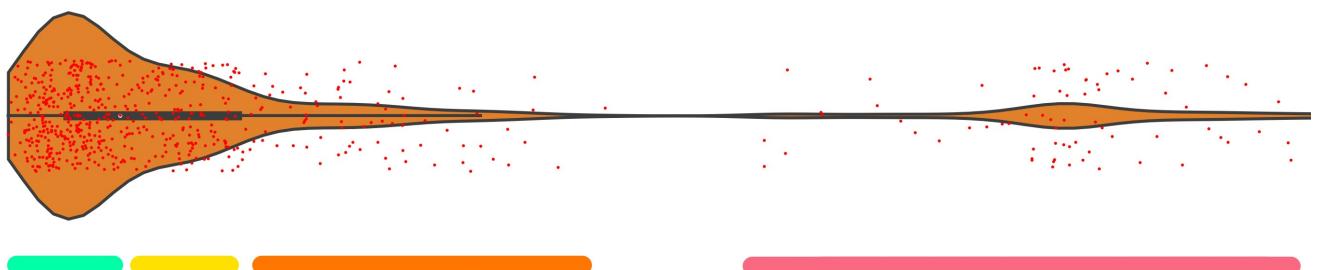
Doppeltgelbe, Code-60-Bereiche und geplante Boxenstops haben intuitiv einen konstanten, vergleichsweise großen Einfluss auf die Rundenzeiten aller betroffenen Fahrer. Sie verschieben Teile der Gaußverteilung mehr oder weniger stark nach rechts verschieben und sorgen für die Schwerpunkte im Stab der Q-Tips.

Verkehr und **gelbe Flaggen** sind dagegen eher zufällige Faktoren mit unsystematischem Einfluss, die sich in der Varianz der Verteilungen widerspiegeln.

Die **Streckenbedingungen** waren während des Rennens 2023 außergewöhnlich konstant. Es wurde keine Rennrunde unter nassen Bedingungen gefahren. Tag-, Nacht- und Blendbedingungen sowie die Außentemperatur sind ebenfalls wichtige Faktoren, die allerdings alle Teilnehmer gleichermaßen betreffen.

Defekte sind ein Sonderfall, da sie die Rundenzeit eines einzelnen Fahrers in der Regel dramatisch beeinflussen, aber eher seltene Ereignisse sind.

Um die Forschungsfrage zu beantworten, wollen wir Rundenzeiten, welche Störgrößen enthalten, möglichst herausfiltern. Wir gehen dazu von der Annahme aus, dass die Rundenzeitverteilung einem Gaußschen Mischmodell [4] folgt, d.h. dass sie sich in mehrere überlagerte Gaußverteilungen zerlegen lassen. Wir sortieren die gefundenen Komponenten nach ihrem Mittelwert und nutzen das Modell anschließend zur Klassifikation der Rundenzeiten - je kleiner der Clusterindex, desto ungestörter die Rundenzeit.



Modeling/Feature Learning

Um die Rundenzeiten der Fahrzeuge zu clustern und das Cluster den Daten hinzuzufügen, verwenden wir das `GaussianMixture`-Modell aus `scikit-learn` [5]. Das Modell ist ein Unsupervised Learning-Verfahren, das mit der Anzahl der erwarteten Cluster initialisiert wird und dann die Parameter der Gaußverteilungen, die die Daten am besten beschreiben, lernt.

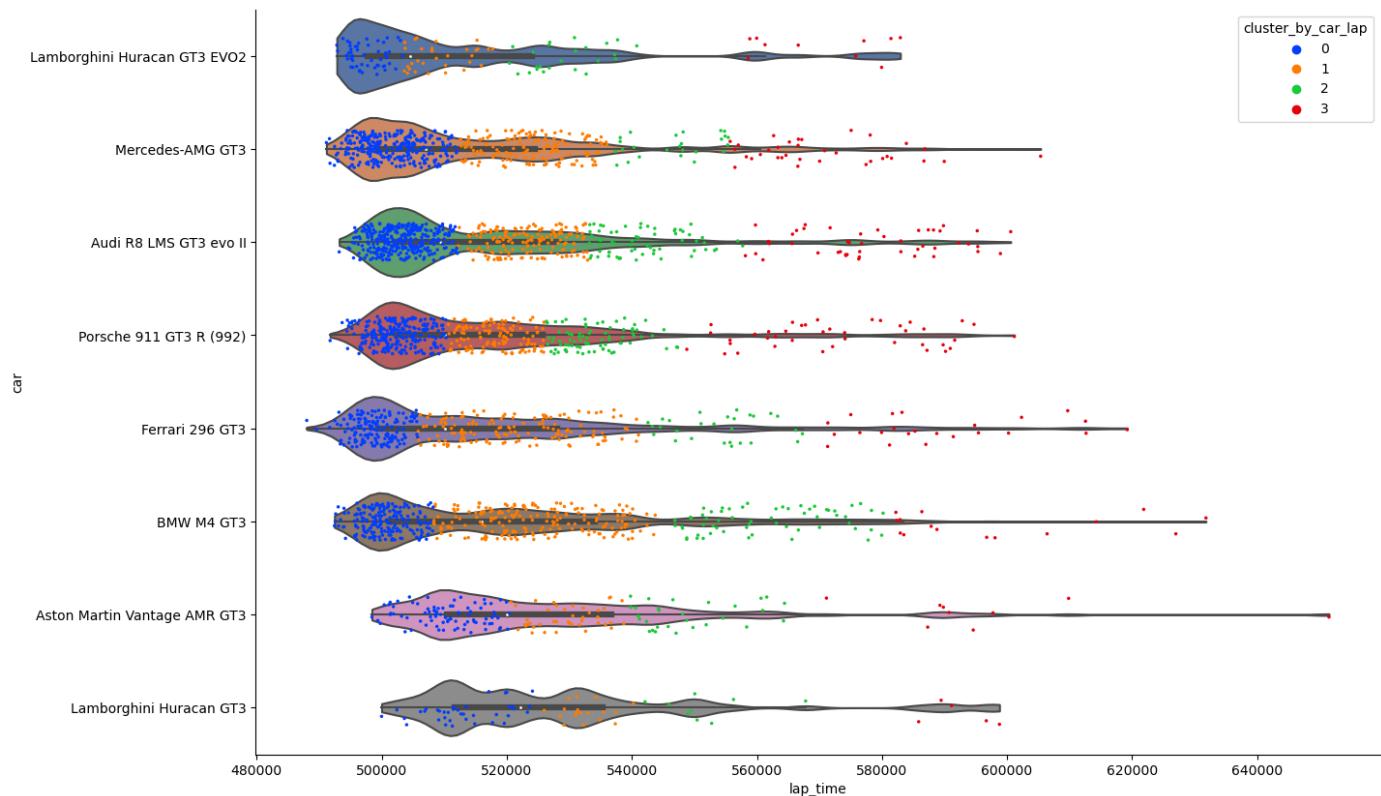
Runden mit Boxenstopp-Störung sind in den Daten entsprechend markiert, wir filtern sie hier heraus. Ebenso entfernen wir Runden, die mehr als 11 Minuten dauern. Diese sind in der Regel durch Defekte oder Unfälle verursacht und würden die Clusterbildung stören.

```
In [13]: def gmc(df: pd.DataFrame, column: str, group_name: str) -> pd.DataFrame:
    gmc = mixture.GaussianMixture(n_components=4)
    res = pd.DataFrame(df)
    component_labels = gmc.fit_predict(res[[column]])
    component_mean_ranks = gmc.means_.flatten().argsort().argsort()
    component_ranks = np.vectorize(component_mean_ranks.take)(component_labels)
    res[f"cluster_by_{group_name}"] = component_ranks
    return res

df_sp9_sectors = df_sp9_sectors.query("in_pit == 'N' and lap_time < 660000")
df_sp9_sectors = df_sp9_sectors.groupby("car", group_keys=False).apply(
    gmc, "lap_time", "car_lap"
)
```

Nach dem Fitting des Modells können wir die Clusterzugehörigkeit der Runden abfragen und die Plausibilität des Clusterings anhand der Visualisierung bewerten:

```
In [14]: plot_time_distribution(df_sp9_sectors, hue="cluster_by_car_lap")
```

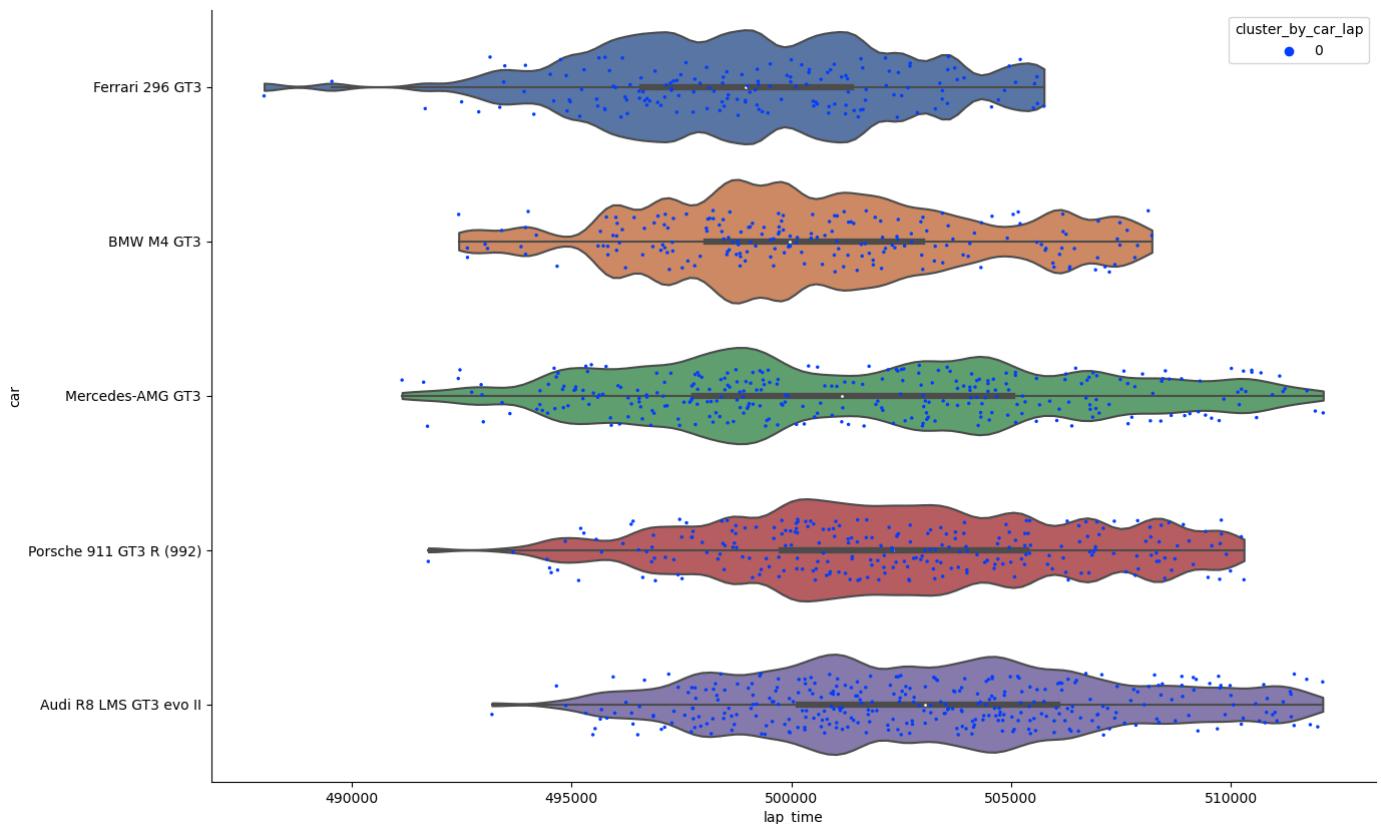


Das Modell ist insbesondere bei den Fahrzeugen, die vergleichsweise wenige Runden gefahren sind, nicht perfekt, fängt aber die Struktur der Verteilung gut ein.

Evaluation

Nun können wir die Clusterinformation nutzen, um ungestörte Rundenzeiten zu selektieren und deren Verteilung für die einzelnen Fahrzeuge zu visualisieren (wir lassen die abgeschlagenen Fahrzeuge ab hier weg):

```
In [15]: df_sp9_sectors = df_sp9_sectors.query(
    "car != ['Lamborghini Huracan GT3', 'Lamborghini Huracan GT3 EVO2', 'Aston Martin Va
)
df_clean_laps = pd.DataFrame(df_sp9_sectors.query("cluster_by_car_lap == 0"))
plot_time_distribution(df_clean_laps, hue="cluster_by_car_lap")
```



Für den 296 GT3 zeigt sich eine gute, wenn auch nicht die klar beste Rundenzeitverteilung. Wir vergleichen abschließend die Sektorzeiten in schnellen Runden zwischen Ferraris und anderen Fabrikaten (in Leserichtung, Sektor 1 links oben):

```
In [16]: from itertools import chain

def get_sector_times(df: pd.DataFrame) -> pd.DataFrame:
    columns = [f"sectortime_{i}"] for i in range(1, 10)
    columns = [*columns, ["lap_time"]]
    columns = list(chain.from_iterable(columns))
    dfst = df.loc[:, columns]
    dfst.columns = pd.MultiIndex.from_tuples(
        [tuple(c.split("_")) for c in dfst.columns]
    )
    return dfst

def plot_sectors(df: pd.DataFrame) -> None:
    df = df.copy()

    fig = plt.figure(figsize=(15, 10))
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
```

```

for i in range(1, 10):
    df_sub = df.query(f"variable_1 == '{i}'")
    ax = fig.add_subplot(3, 3, i)
    sns.violinplot(
        data=df_sub,
        y="name",
        x="value",
        palette="deep",
        cut=0,
        showfliers=False,
        ax=ax,
    )
    # see https://stackoverflow.com/questions/42404154/increase-tick-label-font-size
    # prevents: UserWarning: FixedFormatter should only be used together with FixedL
    ax.set_xticks(ax.get_xticks()[1:])
    # set the y-labels with
    _ = ax.set_xticklabels(ax.get_xticks(), size=6)

    if i % 9 != 1:
        ax.set(yticklabels=[])    # remove axis info
        ax.set(ylabel=None)
        ax.tick_params(left=False)
plt.show()

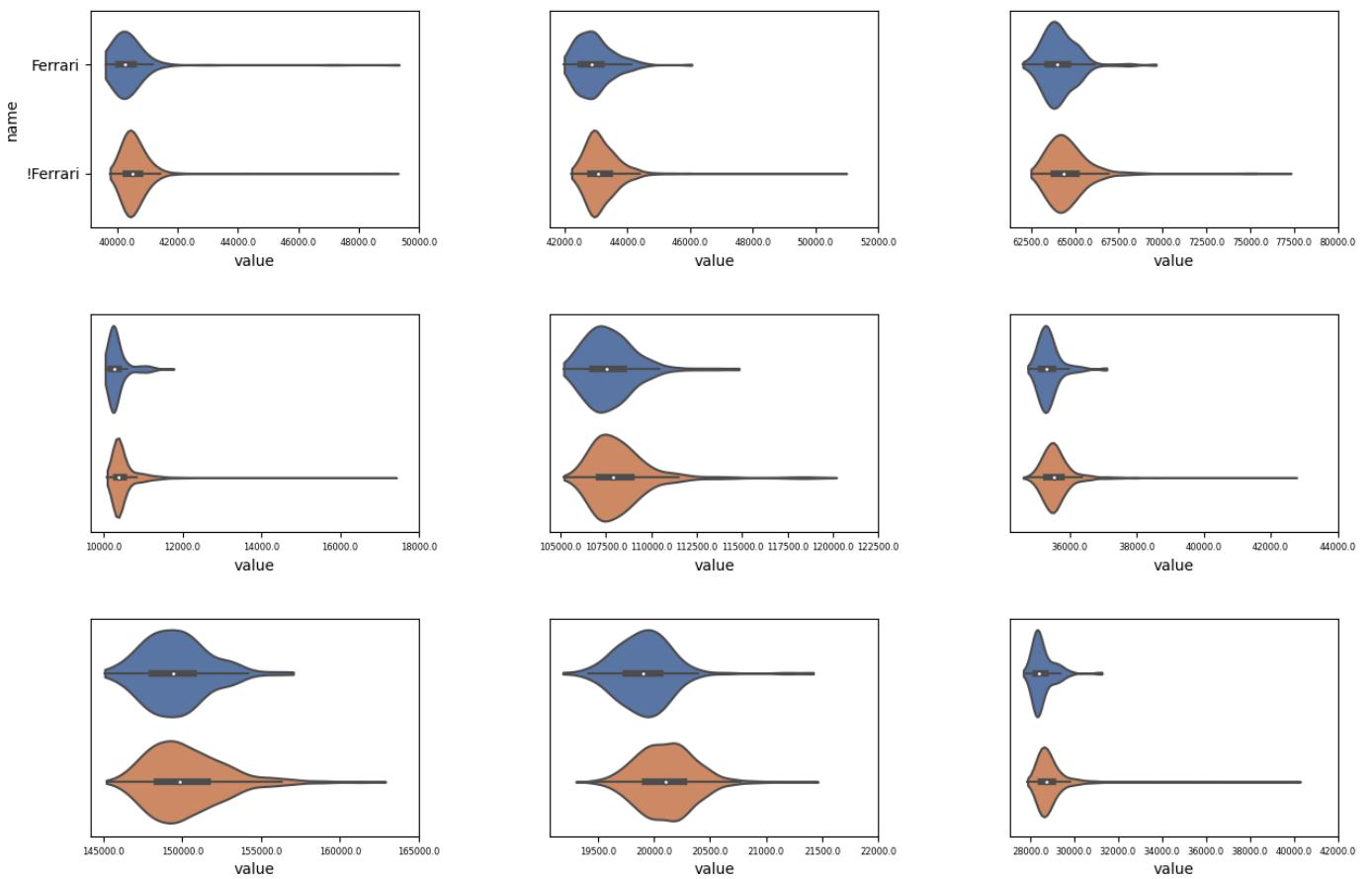
```

```

df_ferrari_sectors = (
    get_sector_times(df_clean_laps.query("car == 'Ferrari 296 GT3'"))
    .melt()
    .query("variable_0 == 'sectortime'")
)
df_ferrari_sectors["is_ferrari"] = True
df_ferrari_sectors["name"] = "Ferrari"
df_unferrari_sectors = (
    get_sector_times(df_clean_laps.query("car != 'Ferrari 296 GT3'"))
    .melt()
    .query("variable_0 == 'sectortime'")
)
df_unferrari_sectors["is_ferrari"] = False
df_unferrari_sectors["name"] = "!Ferrari"

df_all_sectors = pd.concat([df_ferrari_sectors, df_unferrari_sectors])
plot_sectors(df_all_sectors)

```



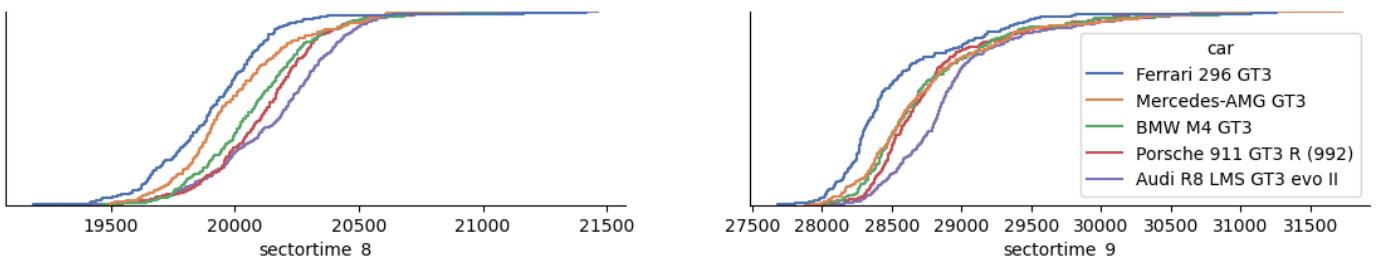
Hier zeigt sich ein leichter Vorteil für den 296 GT3. Insbesondere in den Sektoren 8 und 9 macht sich der Unterschied bemerkbar:

```
In [17]: def plot_time_ecdf(
    df: pd.DataFrame, values: list[str] = ["lap_time"], hue: str | None = None
) -> None:
    df = df.copy()
    f, ax = plt.subplots(ncols=len(values), figsize=(14, 2))

    for i, value in enumerate(values):
        df["__sort_q2"] = df.groupby("car")[value].transform(np.quantile, q=0.5)
        df.sort_values("__sort_q2", inplace=True)

        p = sns.ecdfplot(
            hue=df["car"],
            x=df[value],
            palette="deep",
            ax=ax[i],
            legend=False if i == 0 else True,
        )
        p.set(yticklabels=[]) # remove the tick labels
        p.set(ylabel=None) # remove the axis label
        p.tick_params(left=False)
    sns.despine()

plot_time_ecdf(
    df_clean_laps.query("cluster_by_car_lap == 0 and sectortime_9 < 32000"),
    values=["sectortime_8", "sectortime_9"],
)
```



Ein Blick auf die Streckenkarte zeigt, dass die Sektoren 8 und 9 zur langen Geraden vor Start und Ziel gehören:

In [18]:

```
from dataclasses import dataclass

@dataclass
class Coords:
    lat: float
    lon: float

map_center = Coords(
    50.628615,
    6.883192,
)
track_center = Coords(50.35849, 6.95916)

tiles = "CartoDB Dark_Matter" if _DARK_MODE else "CartoDB positron"

nbr_map = folium.Map(
    location=[track_center.lat, track_center.lon],
    zoom_start=13,
    tiles=tiles,
    attr='&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> con scrollWheelZoom=False,
)

folium.Marker(
    [track_center.lat, track_center.lon], popup="Nürburgring track center"
).add_to(nbr_map)

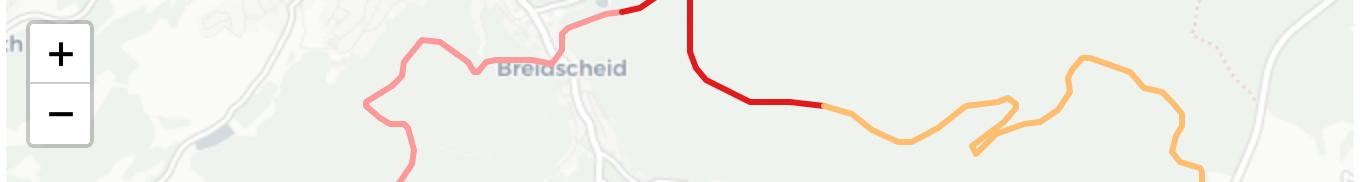
# create a color palette with 9 distinct colors
colors = sns.color_palette("Paired", n_colors=9).as_hex()

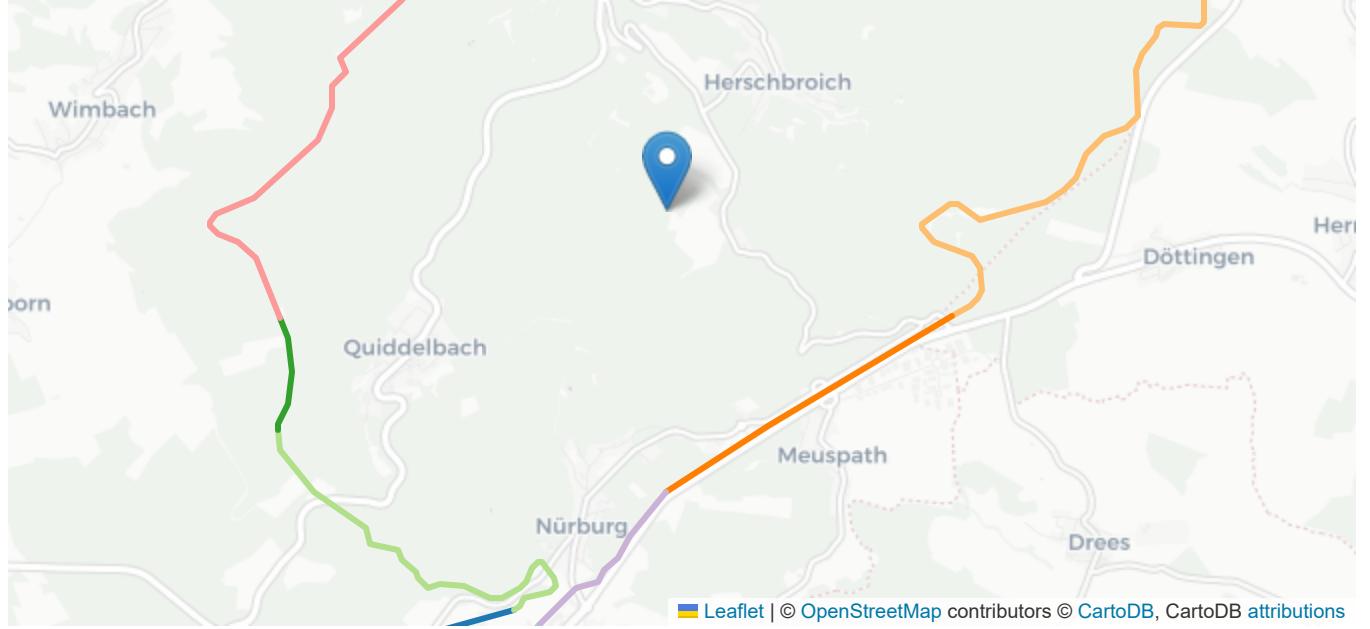
def style_function(x: dict) -> dict[str, str]:
    return {"color": colors[int(x["properties"]]["name"][-1]) - 1]}

for i in range(9):
    folium.GeoJson(
        f"data/gis/NBR 24h S{i+1}.geojson",
        style_function=style_function,
        tooltip=f"Sektor {i+1}",
    ).add_to(nbr_map)

nbr_map
```

Out[18]:





Ein Rennkommissar könnte angesichts dieser Ergebnisse zu dem Schluss kommen, dass es hinreichende Indizien für einen leichten Vorteil des 296 GT3 gegenüber den anderen Fahrzeugen gibt, um die Balance of Performance einer erneuten Prüfung zu unterziehen. Ein Renningenieur eines der Ferrari-Teams könnte hingegen auf den Gedanken kommen, seinen Fahrern für das nächste Vorbereitungsrennen oder das nächste Qualifikationstraining etwas weniger Drehzahl auf der langen Geraden zu empfehlen.

Kritische Würdigung

Ein positiver Aspekt dieses Projekts ist, dass es zeigt, wie nützlich das *CRISP-DM*-Modell für die Arbeit mit großen Datensätzen sein kann. Es ist leicht, sich in den Details und Nebensächlichkeiten solcher Datensätze zu verlaufen.

Auf der anderen Seite war der Aufwand für das Data Wrangling ausgesprochen hoch - ein häufiger Stolperstein in Datenanalyse-Projekten. Die Umwandlung realer Daten in eine für die Analyse geeignete Form erfordert viel Boilerplate-Code und hat hier einen Großteil der Gesamtzeit in Anspruch genommen.

Das verwendete Clustering-Verfahren, GaussianMixture, hat sich als nicht besonders robust erwiesen und benötigte relativ viel manuelles Tuning. Insbesondere war die "richtige" Anzahl von Clustern nicht im Voraus bestimmbar, sondern musste iterativ ermittelt werden. Ein Grid-Search-Ansatz hätte in dieser Situation nützlich sein können, dazu wäre jedoch ein Score nötig gewesen, der die Qualität der Cluster misst, wie zum Beispiel der Silhouette Score [6].

Schließlich ist zu beachten, dass die Wetterbedingungen während des Rennens 2023 außergewöhnlich konstant waren und es insbesondere immer trocken war. Das wechselhafte Wetter wird in zukünftigen Analysen vorraussichtlich wieder berücksichtigt werden müssen.

Ausblick

In zukünftigen Analysen könnten weitere oder andere Verfahren zur Anomalieerkennung zum Einsatz kommen, um die Ergebnisqualität zu verbessern und weitere Einsichten in die Fahrzeugleistung zu gewähren.

Darüber hinaus könnten mehr Daten in die Analyse einbezogen werden. Dies könnte Daten aus dem Training und dem Qualifying, Daten aus weiteren Rennen und Daten von weiteren Strecken umfassen.

Auch könnten wir weitere Features in die Analyse einbeziehen, beispielsweise die Streckentemperatur und eine Bewertung der Fahrerleistung (vergleichbar dem ELO-Score aus dem Schach). Diese könnten dazu beitragen, ein umfassenderes Bild der Faktoren zu zeichnen, die die Rundenzeiten beeinflussen.

Vielen Dank! Fragen?

Referenzen

- [1] In Wikipedia, „24-Stunden-Rennen auf dem Nürburgring 2023“, Wikipedia, 29. Mai 2023. https://de.wikipedia.org/w/index.php?title=24-Stunden-Rennen_auf_dem_N%C3%BCrburgring_2023&oldid=234125692 (zugegriffen 1. Juni 2023).
- [2] ADAC Nordrhein e.V., „24h-Teilnehmerportal - Bulletins & BoP“, 2023. <https://24h-information.de/index.php?id=102&jahr=2023&uid=5565> (zugegriffen 1. Juni 2023).
- [3] ADAC Nordrhein e.V., „24h-Teilnehmerportal - Zeitnahme / Ergebnisse“, 2023. <https://24h-information.de/index.php?id=102&jahr=2023&uid=5561> (zugegriffen 1. Juni 2023).
- [4] The scikit-learn developers, „Gaussian mixture models“, scikit-learn, 2023. <https://scikit-learn/stable/modules/mixture.html> (zugegriffen 1. Juni 2023).
- [5] The scikit-learn developers, „sklearn.mixture.GaussianMixture“, scikit-learn, 2023. <https://scikit-learn/stable/modules/generated/sklearn.mixture.GaussianMixture.html> (zugegriffen 1. Juni 2023).
- [6] K. R. Shahapure und C. Nicholas, „Cluster Quality Analysis Using Silhouette Score“, in 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA), Sydney, Australia: IEEE, Okt. 2020, S. 747–748. doi: 10.1109/DSAA49011.2020.00096.
- [7] K. Barron, „suncalc-py“. 5. Mai 2023. Zugegriffen: 1. Juni 2023. [Online]. Verfügbar unter: <https://github.com/kylebarron/suncalc-py>
- [8] European Commission, „Regulation 923/2012“. 26. September 2012. [Online]. Verfügbar unter: <https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32012R0923>
- [9] Deutscher Wetterdienst, „Climate Data Center“, 2023. <https://cdc.dwd.de/portal/shortlink/b0076c6a-0617-4c11-94bd-ff059cafd543> (zugegriffen 1. Juni 2023).
- [10] The pandas development team, „pandas.merge_asof — pandas 2.0.2 documentation“, 2023. https://pandas.pydata.org/docs/reference/api/pandas.merge_asof.html (zugegriffen 1. Juni 2023).

Backup

```
In [19]: df_race_sectors.groupby("car_number").min().sort_values("theoretical_best_time").loc[:, ["team", "car", "time_of_day", "theoretical_best_time"]].head(10)
```

Out[19]:

car_number	team	car	time_of_day	theoretical_best_time
------------	------	-----	-------------	-----------------------

30	Frikadelli Racing Team		Ferrari 296 GT3	00:08:13.926	486623
4	Mercedes-AMG Team Bilstein		Mercedes-AMG GT3	00:02:24.700	487109
98	ROWE RACING		BMW M4 GT3	00:01:12.252	487212
2	Mercedes-AMG Team GetSpeed		Mercedes-AMG GT3	00:10:28.985	487900
27	ABT Sportsline	Lamborghini Huracan GT3 EVO2		00:05:33.990	488351
911	Manthey EMA	Porsche 911 GT3 R (992)		00:07:58.487	489081
99	ROWE RACING		BMW M4 GT3	00:11:00.028	489356
72	BMW Junior Team		BMW M4 GT3	00:10:38.355	490034
5	Scherer Sport PHX	Audi R8 LMS GT3 evo II		00:06:38.395	490190

Sonnenstand

Um Einflüsse durch den Sonnenstand, insbesondere durch Blendung bewerten zu können, bestimmen den Azimut und die Höhe der Sonne für jede Runde mit dem suncalc-Paket von Kyle Barron [7]-Paket von Kyle Barron. Wir berechnen die Position der Sonne relativ zu einem willkürlich gewählten Punkt nahe dem Zentrum der Strecke ($50^{\circ}21'30.6^{"}$ N $6^{\circ}57'33.0^{"}$ E) zum Zeitpunkt jeder abgeschlossenen Runde. Wir verwenden dazu nicht den Wert `time_of_day`, da dieser kein Datum enthält und wir den Übergang vom Samstag zum Sonntag gesondert behandeln müssten. Besser eignet sich der `time_elapsed`-Wert - die seit dem 20.05.2023 um 00:00:00+02:00 zum Abschluss jeder Runde verstrichene Zeit in Millisekunden:

In [20]:

```
import math

elapsed_zero = pd.Timestamp("2023-05-20 00:00:00", tz="Europe/Berlin").tz_convert("utc")
df_sp9_sectors["lap_timestamp"] = elapsed_zero + df_sp9_sectors["elapsed_time"].astype(
    "timedelta64[ms]"
)

solar_positions = pd.DataFrame(
    suncalc.get_position(
        df_sp9_sectors["lap_timestamp"], track_center.lon, track_center.lat
    )
)

df_sp9_sectors = pd.concat([df_sp9_sectors, solar_positions], axis=1)
```

Mit der Ermittlung der Sonnenhöhe für jede Runde sind wir nun in der Lage, die Runden zu unterscheiden, die während der Nachtstunden absolviert wurden. Gemäß der zivilen Dämmerungsgrenze nach der Definition der [European Union Aviation Safety Agency \(EASA\)](#) betrachten wir eine Runde als "unter dunklen Bedingungen" durchgeführt, wenn die Sonnenhöhe unter -6° liegt [8]. Suncalc gibt die Sonnenhöhe in Radianten zurück, daher müssen wir den Schwellenwert umrechnen: $-\frac{6}{180}\pi \approx -0,10472$

In [21]:

```
night_threshold = -6 / 180 * math.pi
df_sp9_sectors["night"] = df_sp9_sectors["altitude"] < (night_threshold)

glare_lower_threshold = -1 / 180 * math.pi
glare_upper_threshold = 15 / 180 * math.pi
df_sp9_sectors["glare"] = df_sp9_sectors["altitude"].apply(
    lambda x: glare_lower_threshold < x < glare_upper_threshold
)

df_sp9_sectors.sample(5).loc[:, ["lap_timestamp", "night", "glare"]]
```

Out[21]:

lap_timestamp night glare

car_number	lap_number				
25	20	2023-05-20 16:55:16.283000+00:00	False	False	
4	93	2023-05-21 04:02:42.447000+00:00	False	True	
20	13	2023-05-20 15:51:44.795000+00:00	False	False	
99	66	2023-05-20 23:48:42.021000+00:00	True	False	
100	138	2023-05-21 11:51:18.382000+00:00	False	False	

Wetter

Wir fügen Wetterdaten hinzu, um die Rundenzeiten in Abhängigkeit von den Wetterbedingungen analysieren zu können. Wir beziehen die Wetterdaten vom [Deutschen Wetterdienst \(DWD\)](#), der aktuelle und historische Stationsdaten im CSV-Format zur Verfügung stellt. Wir nutzen die Lufttemperatur in 2 Metern Höhe (OBS_DEU_PT10M_T2M) und den Luftdruck auf Stationsniveau (OBS_DEU_PT10M_PP), die an der Station Nürburg-Barweiler in 10-Minuten-Intervallen gemessen wurden (etwa 6,34 km vom Referenzpunkt entfernt) [9].

Spaßiges Problem: Die CSV des DWD haben ein zusätzliches abschließendes Trennzeichen pro Zeile, was dazu führt, dass der Pandas-CSV-Parser die erste Spalte als Zeilenindex des DataFrames behandelt und die restlichen Daten sich ein Feld nach links schieben. Um dies zu beheben, übergeben wir `index_col=False` an `pd.read_csv()`.

Wir stellen sicher, dass die Zeitstempel auf die gleiche Zeitzone wie die Rundenzeiten gesetzt sind und indizieren die Wetterdaten nach Zeitstempel, um das Zusammenführen zu erleichtern.

In [22]:

```
def read_weather_data(filename: str) -> pd.DataFrame:
    weather_data = pd.read_csv(
        filename,
        sep=",",
        index_col=False,
        usecols=["Zeitstempel", "Wert"],
        # parse_dates=["Zeitstempel"],
    )
    weather_data["Zeitstempel"] = pd.to_datetime(weather_data["Zeitstempel"], utc=True)
    weather_data.set_index("Zeitstempel", inplace=True)
    return weather_data

air_temp_df = read_weather_data("data/weather/data_OBS_DEU_PT10M_T2M_3660.csv")
air_temp_df.rename(columns={"Wert": "air_temperature"}, inplace=True)
air_pressure_df = read_weather_data("data/weather/data_OBS_DEU_PT10M_PP_3660.csv")
air_pressure_df.rename(columns={"Wert": "air_pressure"}, inplace=True)

atm_df = pd.merge(
    air_temp_df,
    air_pressure_df,
    left_index=True,
    right_index=True,
)
atm_df.sample(5)
```

Out[22]:

air_temperature air_pressure

Zeitstempel

2023-05-22 04:10:00+00:00	12.7	957.3
2023-05-21 22:50:00+00:00	15.4	958.0
2023-05-22 05:30:00+00:00	13.4	957.7
2023-05-21 16:50:00+00:00	20.0	957.7
2023-05-21 02:00:00+00:00	9.9	959.1

Wir verknüpfen nun die gesammelten Wetterdaten mit unseren Rundenzeiten. Da die Zeitstempel der Wettermessungen nicht mit den Durchgangszeiten synchronisiert sind, führen wir einen **Merge nach Schlüsseldistanz** [10] durch und fügen jeder Runde den zeitlich nächsten Messwert hinzu:

```
In [23]: # Both dataframes must be sorted by timestamp
df_sp9_sectors.sort_values(by="lap_timestamp", inplace=True)
atm_df.sort_index(inplace=True)

df_sp9_sectors = pd.merge_asof(
    df_sp9_sectors,
    atm_df,
    left_on="lap_timestamp",
    right_index=True,
    direction="nearest",
)

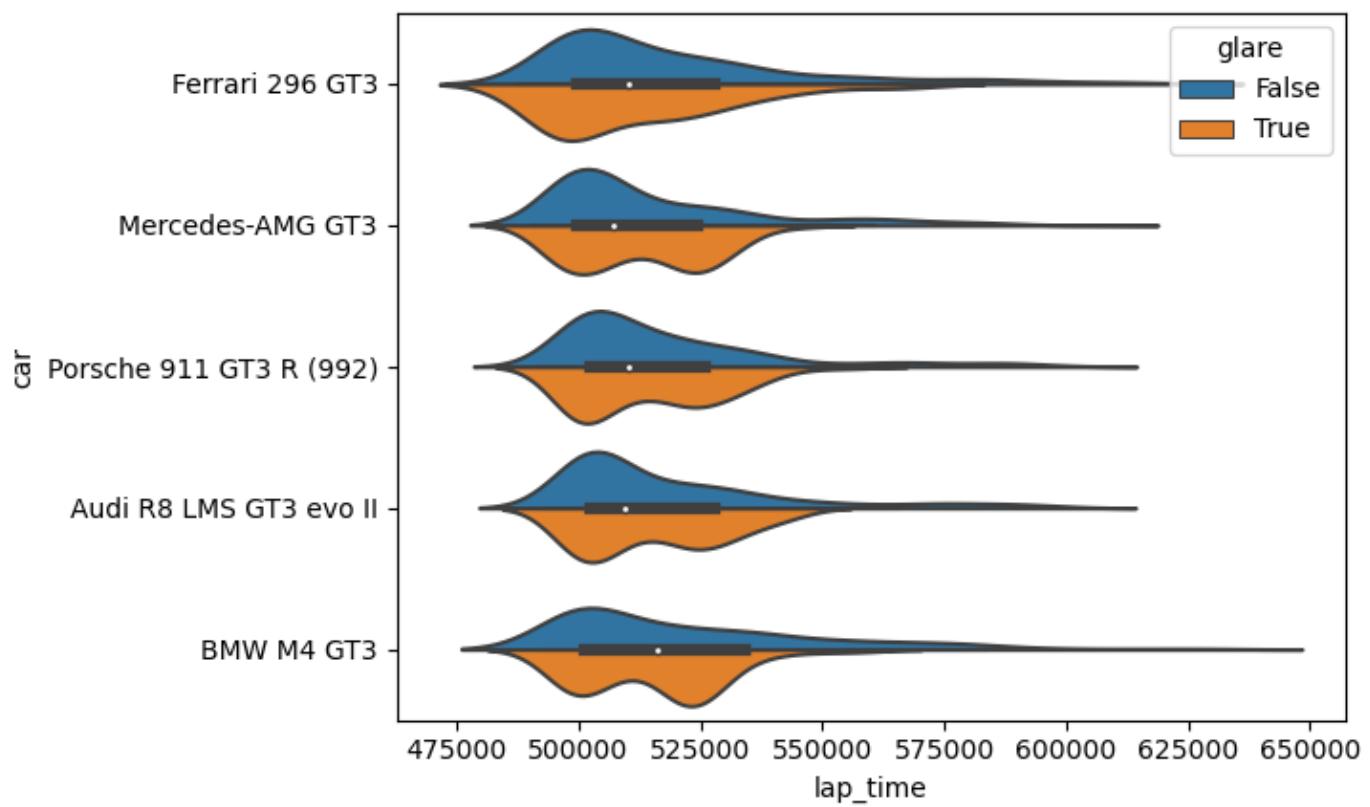
df_sp9_sectors.sample(5).loc[:, ["lap_timestamp", "air_temperature", "air_pressure"]]
```

car_number	lap_number	lap_timestamp	air_temperature	air_pressure
1	68	2023-05-21 00:12:11.048000+00:00	10.2	959.7
96	157	2023-05-21 13:32:01.260000+00:00	19.8	957.9
40	49	2023-05-20 21:21:18.125000+00:00	11.3	960.9
20	136	2023-05-21 10:26:44.323000+00:00	18.4	958.2
72	19	2023-05-20 16:46:14.820000+00:00	14.8	960.9

Wir vergleichen die Rundenzeiten unter Blendbedingungen und ohne Blendbedingungen:

```
In [24]: sns.violinplot(
    data=df_sp9_sectors,
    y="car",
    x="lap_time",
    hue="glare",
    split=True,
    showfliers=False,
)
```

Out[24]: <Axes: xlabel='lap_time', ylabel='car'>



Und wir stellen den gleichen Vergleich für die Rundenzeiten unter Tages- und Nachtbedingungen an:

```
In [25]: sns.violinplot(
    data=df_sp9_sectors,
    y="car",
    x="lap_time",
    hue="night",
    split=True,
    showfliers=False,
)
```

```
Out[25]: <Axes: xlabel='lap_time', ylabel='car'>
```

