🚦 Traffic Prediction & Optimization

Ziel

Dieses Notebook zeigt eine **vollständige Analyse- und Modellierungs-Pipeline** zur **Vorhersage von Verkehrsaufkommen** – von der Datenbasis bis zum Modellvergleich.

🔄 Analyse-Pipeline

- 📊 **Datenbasis**
  - Synthetische Verkehrsdaten generieren **oder**
  - CSV-Datei laden (mit `ds` und `y`)
- 🔍 **Explorative Datenanalyse (EDA)**
  - Zeitliche Muster
  - Verteilungen
  - Saisonalitäten
- 🛠️ **Feature Engineering**
  - Lag-Features
  - Rolling Statistics
  - Kalender-Features (Wochentag, Feiertage etc.)
  - Wetter-Features (optional)
- 🤖 **Modelle**
  - Lineare Regression
  - Random Forest
  - *(Optional)* Prophet
  - *(Optional)* LSTM
- 📈 **Evaluation & Vergleich**
  - Modellvergleich anhand geeigneter Metriken
  - Visualisierung der Prognosen
- 🧪 **Optional**
  - MLflow Tracking für Experimente

🎯 Problemdefinition

- **Zielvariable:** `y` – Verkehrsaufkommen
- **Zeitspalte:** `ds` – Datetime (z. B. stündlich oder täglich)
- **Laufzeit:** ⏱️ *< 5 Minuten auf CPU*

📘 So nutzt du dieses Notebook

- ✅ Laufzeit < 5 Minuten auf CPU
- 🔁 Zufallsseeds sind gesetzt für Reproduzierbarkeit
- 🧱 Du kannst:
  - synthetische Verkehrsdaten erzeugen **oder**
  - eine eigene CSV-Datei laden (`ds` , `y`)
- 📊 Führe EDA durch, trainiere mehrere Modelle und vergleiche sie
- 🧠 Optional:
  - MLflow für Experiment-Tracking
  - Prophet oder LSTM für fortgeschrittene Zeitreihenmodelle

1️⃣ Imports & Setup

In [ ]:
```
# ====================================
# Imports & Globales Setup
# ====================================

import warnings
warnings.filterwarnings("ignore")

import pathlib
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

# ====================================
# Machine Learning
# ====================================

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
    r2_score
)
from sklearn.model_selection import GridSearchCV, TimeSeriesSplit
from sklearn.preprocessing import MinMaxScaler

# ====================================
# Plot Styling
# ====================================

plt.style.use("seaborn-v0_8-darkgrid")
sns.set_palette("husl")

# ====================================
# Reproduzierbarkeit
# ====================================

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
```

```
# ====================================
# Optionale Abhängigkeiten
# ====================================

# --- MLflow ---
USE_MLFLOW = True
try:
    import mlflow
    import mlflow.sklearn
except Exception as e:
    USE_MLFLOW = False
    print("MLflow nicht verfügbar:", e)

# --- Prophet ---
USE_PROPHET = True
try:
    from prophet import Prophet
except Exception as e:
    USE_PROPHET = False
    print("Prophet nicht verfügbar:", e)

# --- TensorFlow / LSTM ---
USE_TF = True
try:
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Dropout
except Exception as e:
    USE_TF = False
    print("TensorFlow nicht verfügbar:", e)

print(f"MLflow: {USE_MLFLOW} | Prophet: {USE_PROPHET} | TensorFlow: {USE_TF}")
```

2️⃣ Daten: Laden oder synthetisch generieren

In diesem Abschnitt werden die **Verkehrsdaten** entweder aus einer eigenen CSV-Datei geladen oder **synthetisch erzeugt**, um die komplette Pipeline reproduzierbar zu demonstrieren.

📥 Eigene Daten laden

- CSV-Datei mit:
  - `ds` → Datum/Zeit (Datetime)
  - `y` → Verkehrsaufkommen (Zielvariable)

🧪 Synthetische Daten (optional)

- Realistische Zeitmuster:
  - Tages- und Wochenzyklen
  - Rauschen / Zufallseinflüsse
  - Trend-Komponente

➕ Optionale Zusatzfeatures

- **Wetter** (z. B. Temperatur, Regen)
- **Feiertag** (binär oder kategorial)

Diese Features können später im **Feature Engineering** genutzt werden, um die Prognosequalität zu verbessern.

In [ ]:

```
# ====================================
# 2. Daten: Laden oder synthetisch generieren
# ====================================

USE_CSV = False
CSV_PATH = "data/traffic.csv"


def generate_realistic_traffic_data(days=60, start="2024-01-01"):
    """
    Generiert realistische stündliche Verkehrsdaten
    mit Tages-, Wochenmustern, Trend und Rauschen.
    """
    # Zeitachse (stündlich)
    dates = pd.date_range(start=start, periods=days * 24, freq="h")
    t = np.arange(len(dates))

    # Muster & Effekte
    daily_pattern = 30 * np.sin(2 * np.pi * t / 24)
    weekly_pattern = 10 * np.sin(2 * np.pi * t / (24 * 7))
    trend = np.linspace(0, 5, len(t))
    noise = np.random.normal(0, 4, len(t))

    # Traffic berechnen
    traffic = 50 + daily_pattern + weekly_pattern + trend + noise
    traffic = np.clip(traffic, 5, 100)

    # Optionale Zusatzfeatures
    weather = np.random.choice(
        ["Sonnig", "Bewölkt", "Regen"],
        size=len(t),
        p=[0.5, 0.3, 0.2]
    )

    is_holiday = np.zeros(len(t))
    is_holiday[::168] = 1  # ca. einmal pro Woche

    # DataFrame
    df = pd.DataFrame({
        "ds": dates,
        "y": traffic,
        "Geschwindigkeit": 120 - traffic * 0.5 + np.random.normal(0, 5, len(t)),
        "Wetter": weather,
        "Feiertag": is_holiday.astype(int),
    })

    # Zeitbasierte Features
    df["Stunde"] = df["ds"].dt.hour
    df["Wochentag"] = df["ds"].dt.dayofweek
```

```
    return df


# ====================================
# Daten laden oder generieren
# ====================================

if USE_CSV:
    df = pd.read_csv(CSV_PATH)
    df["ds"] = pd.to_datetime(df["ds"])
else:
    df = generate_realistic_traffic_data(days=60)

print(f"Datensatz: {df.shape}")
print(f"Zeitraum: {df['ds'].min()} bis {df['ds'].max()}")

df.head()
```

3️⃣ EDA – Explorative Datenanalyse

Ziel der explorativen Datenanalyse ist es, **Strukturen, Muster und Auffälligkeiten** in den Verkehrsdaten zu erkennen, bevor Features erzeugt oder Modelle trainiert werden.

🔍 Fragestellungen

- Wie entwickelt sich das Verkehrsaufkommen über die Zeit?
- Gibt es **Tages- oder Wochenmuster**?
- Wie ist die Verteilung der Zielvariable `y`?
- Gibt es Zusammenhänge mit Wetter oder Feiertagen?

📈 Analyseschritte

- Zeitreihenplot des Verkehrsaufkommens
- Tagesprofil (durchschnittlicher Traffic je Stunde)
- Wochenprofil (durchschnittlicher Traffic je Wochentag)
- Verteilungen & Boxplots
- Korrelationen zwischen numerischen Features

🧠 Erkenntnisse (Platzhalter)

> Die wichtigsten Beobachtungen aus der EDA werden hier kurz zusammengefasst (z. B. starke Rush-Hour-Effekte, geringerer Traffic an Wochenenden etc.).

Diese Erkenntnisse fließen direkt in das **Feature Engineering** und die **Modellauswahl** ein.

In [ ]:
```
# ====================================
# 3. EDA – Explorative Datenanalyse
```

```python
# ====================================

print("Deskriptive Statistik der Zielvariable y:")
display(df["y"].describe())

# ------------------------------------
# Visualisierungen
# ------------------------------------

fig, axes = plt.subplots(2, 2, figsize=(14, 8))

# 1️⃣ Traffic über Zeit
axes[0, 0].plot(df["ds"], df["y"], linewidth=1)
axes[0, 0].set_title("Traffic über Zeit")
axes[0, 0].set_ylabel("y")

# 2️⃣ Durchschnittlicher Traffic pro Stunde
hourly_mean = df.groupby("Stunde")["y"].mean()
axes[0, 1].bar(hourly_mean.index, hourly_mean.values)
axes[0, 1].set_title("Ø Traffic pro Stunde")
axes[0, 1].set_xlabel("Stunde")

# 3️⃣ Durchschnittlicher Traffic pro Wochentag
dow_mean = df.groupby("Wochentag")["y"].mean()
axes[1, 0].plot(dow_mean.index, dow_mean.values, marker="o", linewidth=2)
axes[1, 0].set_title("Ø Traffic pro Wochentag")
axes[1, 0].set_xticks(range(7))
axes[1, 0].set_xticklabels(["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"])

# 4️⃣ Verteilung der Zielvariable
axes[1, 1].hist(df["y"], bins=30, edgecolor="black")
axes[1, 1].set_title("Verteilung von y")
axes[1, 1].set_xlabel("y")

plt.tight_layout()
plt.show()
```

4️⃣ Train/Test Split & Feature Engineering

In diesem Abschnitt werden aus den Rohdaten **modellrelevante Features** erzeugt und die Zeitreihe **korrekt in Trainings- und Testdaten** aufgeteilt.

🛠️ Feature Engineering

Zeitbasierte Features

- Stunde (`Stunde`)
- Wochentag (`Wochentag`)
- Optional: Monat, Wochenende, Feiertag

Lag-Features

- Verzögerte Zielwerte:

- `y_lag_1`
- `y_lag_24`
- `y_lag_168` (eine Woche)

Rolling Features

- Gleitende Mittelwerte:
  - `y_roll_24`
  - `y_roll_168`

Diese Features helfen den Modellen, **kurz- und mittelfristige Muster** zu lernen.

⏱️ Zeitreihen-Train/Test-Split

- Keine zufällige Durchmischung ❌
- Split erfolgt **chronologisch**
- Typischer Split:
  - **Train:** erste 80 %
  - **Test:** letzte 20 %

🎯 Ziel

- Vermeidung von **Data Leakage**
- Realistische Simulation einer echten Vorhersagesituation
- Einheitliche Feature-Basis für alle Modelle

In [ ]:
```python
# ====================================
# 4. Train / Test Split (Zeitreihe)
# ====================================

SPLIT_RATIO = 0.8
split_idx = int(len(df) * SPLIT_RATIO)

train_df = df.iloc[:split_idx].copy()
test_df = df.iloc[split_idx:].copy()

print(f"Train: {train_df.shape}")
print(f"Test:  {test_df.shape}")


# ====================================
# Feature Engineering
# ====================================

def create_features(data: pd.DataFrame) -> pd.DataFrame:
    """
    Erstellt Feature-Matrix mit:
    - Kalenderfeatures
    - Lag-Features
    - Rolling Mean Features
```

```
    - Wetter-Indikatoren (optional)
    """
    X = pd.DataFrame(index=data.index)

    # Kalenderfeatures
    X["hour"] = data["Stunde"]
    X["day_of_week"] = data["Wochentag"]
    X["is_holiday"] = data["Feiertag"]

    # Lag-Features
    for lag in [1, 6, 24]:
        X[f"lag_{lag}"] = data["y"].shift(lag)

    # Rolling Means
    for window in [6, 24]:
        X[f"rolling_mean_{window}"] = data["y"].rolling(window).mean()

    # Wetter (falls vorhanden)
    if "Wetter" in data.columns:
        X["is_rain"] = (data["Wetter"] == "Regen").astype(int)
        X["is_cloudy"] = (data["Wetter"] == "Bewölkt").astype(int)
    else:
        X["is_rain"] = 0
        X["is_cloudy"] = 0

    # Fehlende Werte durch Backfilling behandeln
    return X.bfill()


# ====================================
# Feature- & Zieldefinition
# ====================================

X_train = create_features(train_df)
X_test = create_features(test_df)

y_train = train_df["y"].values
y_test = test_df["y"].values

print(f"X_train: {X_train.shape}")
display(X_train.head())
```

5 Modelle: Linear Regression & Random Forest

In diesem Abschnitt werden zwei klassische Regressionsmodelle für die Verkehrsprognose trainiert und verglichen:

- **Lineare Regression** als einfache, gut interpretierbare Baseline
- **Random Forest Regressor** zur Modellierung nichtlinearer Zusammenhänge

---

📐 Modelle im Überblick

1 Lineare Regression

- Schnelles Baseline-Modell

- Gut interpretierbar
- Erwartung: begrenzte Leistung bei komplexen Mustern

2️⃣ Random Forest Regressor

- Ensemble aus Entscheidungsbäumen
- Kann Nichtlinearitäten und Interaktionen erfassen
- Robuster gegenüber Ausreißern
- Höherer Rechenaufwand (aber < 5 Min CPU)

🧪 Trainingsstrategie

- Training auf **Train-Daten**
- Evaluation auf **Test-Daten**
- Zeitreihen-konformer Split (kein Shuffle)
- Metriken:
  - MAE
  - RMSE
  - R^2

🎯 Ziel

- Quantitativer Vergleich einfacher vs. komplexerer Modelle
- Grundlage für spätere Erweiterungen (Prophet, LSTM)
- Entscheidung, ob Feature Engineering ausreichend ist

In [ ]:

```python
# ====================================
# 5. Modelle: Linear Regression & Random Forest
# ====================================

def eval_metrics(y_true, y_pred):
    """
    Berechnet gängige Regressionsmetriken.
    """
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    r2 = r2_score(y_true, y_pred)
    return mae, rmse, r2


results = {}

# ====================================
# Optional: MLflow Setup
# ====================================

if USE_MLFLOW:
    mlflow.set_experiment("Traffic Prediction & Optimization")
```

```python
# ====================================
# 1️⃣ Linear Regression (Baseline)
# ====================================

lr = LinearRegression()
lr.fit(X_train, y_train)

pred_lr = lr.predict(X_test)
mae, rmse, r2 = eval_metrics(y_test, pred_lr)

results["Linear Regression"] = {
    "MAE": mae,
    "RMSE": rmse,
    "R2": r2,
    "pred": pred_lr,
}

print(f"Linear Regression → MAE={mae:.3f} | RMSE={rmse:.3f} | R²={r2:.3f}")

if USE_MLFLOW:
    with mlflow.start_run(run_name="Linear Regression"):
        mlflow.log_metrics({
            "mae": mae,
            "rmse": rmse,
            "r2": r2
        })
        mlflow.sklearn.log_model(lr, "model")


# ====================================
# 2️⃣ Random Forest Regressor
# ====================================

rf = RandomForestRegressor(
    n_estimators=200,
    max_depth=15,
    random_state=RANDOM_SEED,
    n_jobs=-1
)

rf.fit(X_train, y_train)

pred_rf = rf.predict(X_test)
mae, rmse, r2 = eval_metrics(y_test, pred_rf)

results["Random Forest"] = {
    "MAE": mae,
    "RMSE": rmse,
    "R2": r2,
    "pred": pred_rf,
}

print(f"Random Forest → MAE={mae:.3f} | RMSE={rmse:.3f} | R²={r2:.3f}")

if USE_MLFLOW:
    with mlflow.start_run(run_name="Random Forest"):
        mlflow.log_params({
            "n_estimators": 200,
            "max_depth": 15
        })
```

```
        mlflow.log_metrics({
            "mae": mae,
            "rmse": rmse,
            "r2": r2
        })
        mlflow.sklearn.log_model(rf, "model")


# ====================================
# Feature Importances (Random Forest)
# ====================================

fi = (
    pd.DataFrame({
        "feature": X_train.columns,
        "importance": rf.feature_importances_
    })
    .sort_values("importance", ascending=False)
)

display(fi.head(10))
```

6️⃣ Optional: Prophet (Zeitreihenprognose)

Prophet ist ein spezielles Zeitreihenmodell von Facebook, das besonders gut geeignet ist für:

- **Tägliche, wöchentliche und jährliche Saisonalitäten**
- **Trendänderungen**
- **Feiertage und besondere Events**

🔷 Vorgehensweise mit Prophet

1. **Daten vorbereiten**
    - Prophet erwartet zwei Spalten:
        - `ds` → Datum/Zeit
        - `y` → Zielvariable (Verkehrsaufkommen)
2. **Modell definieren**
    - Saisonalitäten (daily, weekly) automatisch von Prophet erkannt
    - Optional: Feiertage als zusätzliche Features einbinden
3. **Train/Test-Split**
    - Chronologisch, wie zuvor, um Data Leakage zu vermeiden
4. **Training & Vorhersage**
    - Fit auf Trainingsdaten
    - Forecast auf Testdaten
5. **Evaluation**
    - MAE, RMSE, R^2

- Vergleich mit Linear Regression und Random Forest

🎯 Ziel

- Prophet als **zeitreihenorientierte Alternative** testen
- Prüfen, ob Saisonalitäten und Feiertage die Prognose verbessern
- Optional: Grundlage für **komplexere Modelle wie LSTM**

In [ ]:

```python
# ====================================
# 6. Optional: Prophet (Zeitreihenprognose)
# ====================================

if not USE_PROPHET:
    print("Prophet übersprungen.")
else:
    # -------------------------------------
    # Trainingsdaten für Prophet vorbereiten
    # Prophet erwartet nur 'ds' (Datetime) und 'y' (Zielvariable)
    # -------------------------------------
    prophet_train = train_df[["ds", "y"]].copy()

    # -------------------------------------
    # Prophet Modell definieren
    # -------------------------------------
    m = Prophet(
        yearly_seasonality=False,   # keine jährliche Saisonalität
        weekly_seasonality=True,    # wöchentliche Saisonalität
        daily_seasonality=True,     # tägliche Saisonalität
        interval_width=0.95
    )

    # Modell trainieren
    m.fit(prophet_train)

    # -------------------------------------
    # Vorhersage auf Testdaten
    # -------------------------------------
    future = m.make_future_dataframe(periods=len(test_df), freq="h")
    fc = m.predict(future)

    # Prognose für Testzeitraum extrahieren
    pred_prophet = fc["yhat"].tail(len(test_df)).to_numpy()

    # -------------------------------------
    # Evaluation
    # -------------------------------------
    mae, rmse, r2 = eval_metrics(y_test, pred_prophet)
    results["Prophet"] = {
        "MAE": mae,
        "RMSE": rmse,
        "R2": r2,
        "pred": pred_prophet
    }

    print(f"Prophet → MAE={mae:.3f} | RMSE={rmse:.3f} | R²={r2:.3f}")

    # -------------------------------------
```

```
    # Optional: MLflow Logging
    # ------------------------------------
    if USE_MLFLOW:
        with mlflow.start_run(run_name="Prophet"):
            mlflow.log_metrics({"mae": mae, "rmse": rmse, "r2": r2})
```

7️⃣ Optional: LSTM (Deep Learning)

Long Short-Term Memory (LSTM) ist ein **rekurrentes neuronales Netzwerk**, das speziell für Zeitreihen entwickelt wurde. Es kann **langfristige Abhängigkeiten** und **nichtlineare Muster** erfassen, die klassische Modelle wie Linear Regression oder Random Forest oft nicht erkennen.

🔷 Vorgehensweise mit LSTM

1. **Daten skalieren**
   - LSTM ist empfindlich auf Skalen, daher Normalisierung z.B. mit `MinMaxScaler`.
2. **Sequenzielle Features erzeugen**
   - Sliding Window Ansatz: für jede Zeitschritt-Vorhersage `y_t`, werden vorherige `n_lags` Werte als Input verwendet.
3. **Train/Test Split**
   - Wie zuvor, chronologisch, ohne Shuffle.
4. **Modellarchitektur**
   - LSTM Layer
   - Optional Dropout zur Regularisierung
   - Dense Output Layer für Regression
5. **Training & Vorhersage**
   - Loss: Mean Squared Error
   - Optimizer: Adam
   - Epochs und Batchgröße definieren
6. **Evaluation**
   - MAE, RMSE, R^2
   - Vergleich mit Linear Regression, Random Forest und Prophet

---

🎯 Ziel

- Erfassen komplexer zeitlicher Muster im Verkehr
- Vergleich zwischen klassischen Modellen, Prophet und Deep Learning
- Optional: Grundlage für weiterführende Zeitreihenprognosen

In [ ]:
```
# ====================================
# 7. Optional: LSTM (Deep Learning)
```

```python
# ====================================

if not USE_TF:
    print("LSTM übersprungen.")
else:
    # -------------------------------------
    # Helper: Sequenzen erstellen (Sliding Window)
    # -------------------------------------
    def create_sequences(arr, seq_len=24):
        """
        Erstellt Eingabesequenzen für LSTM.
        """
        X, y = [], []
        for i in range(len(arr) - seq_len):
            X.append(arr[i:i + seq_len])
            y.append(arr[i + seq_len])
        return np.array(X), np.array(y)

    # -------------------------------------
    # Zielvariable skalieren
    # -------------------------------------
    scaler = MinMaxScaler()
    y_scaled = scaler.fit_transform(df["y"].to_numpy().reshape(-1, 1))

    # Sequenzen erstellen
    X_seq, y_seq = create_sequences(y_scaled, seq_len=24)

    # Chronologischer Train/Test-Split
    split = int(len(X_seq) * 0.8)
    X_tr, y_tr = X_seq[:split], y_seq[:split]
    X_te, y_te = X_seq[split:], y_seq[split:]

    # Input für LSTM: (samples, timesteps, features)
    X_tr = X_tr.reshape((X_tr.shape[0], X_tr.shape[1], 1))
    X_te = X_te.reshape((X_te.shape[0], X_te.shape[1], 1))

    # -------------------------------------
    # LSTM Modell definieren
    # -------------------------------------
    model = Sequential([
        LSTM(64, activation="relu", input_shape=(24, 1), return_sequences=True),
        Dropout(0.2),
        LSTM(32, activation="relu"),
        Dropout(0.2),
        Dense(16, activation="relu"),
        Dense(1)
    ])

    model.compile(optimizer="adam", loss="mse", metrics=["mae"])

    # -------------------------------------
    # Training
    # -------------------------------------
    history = model.fit(
        X_tr, y_tr,
        epochs=30,
        batch_size=32,
        validation_split=0.2,
        verbose=0
    )
```

```python
# ------------------------------------
# Vorhersage & Inverse Scaling
# ------------------------------------
pred_scaled = model.predict(X_te, verbose=0)
pred = scaler.inverse_transform(pred_scaled).ravel()
y_true = scaler.inverse_transform(y_te.reshape(-1, 1)).ravel()

# Metriken berechnen
mae, rmse, r2 = eval_metrics(y_true, pred)
pred_aligned = pred[:len(y_test)]  # optional an Testdatenlänge anpassen
results["LSTM"] = {"MAE": mae, "RMSE": rmse, "R2": r2, "pred": pred_aligned}

print(f"LSTM → MAE={mae:.3f} | RMSE={rmse:.3f} | R²={r2:.3f}")

# ------------------------------------
# Optional: MLflow Logging
# ------------------------------------
if USE_MLFLOW:
    with mlflow.start_run(run_name="LSTM"):
        mlflow.log_params({"epochs": 30, "batch_size": 32})
        mlflow.log_metrics({"mae": mae, "rmse": rmse, "r2": r2})

# ------------------------------------
# Trainingsverlauf plotten
# ------------------------------------
plt.figure(figsize=(12, 4))
plt.plot(history.history["loss"], label="train loss")
plt.plot(history.history["val_loss"], label="val loss")
plt.title("LSTM Training History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.show()
```

8️⃣ Modellvergleich

In diesem Abschnitt werden die bisherigen Modelle **Linear Regression, Random Forest, Prophet** und optional **LSTM** miteinander verglichen, um die Prognosequalität zu bewerten.

🔷 Vergleichskriterien

- **MAE** → Mittlerer absoluter Fehler
- **RMSE** → Wurzel des mittleren quadratischen Fehlers
- **R^2** → Gütemaß für die erklärte Varianz
- Visuelle Gegenüberstellung: **Ist vs. Vorhersage**

📊 Vorgehensweise

1. Ergebnisse aus allen Modellen in einer **Tabelle zusammenfassen**
2. **Balkendiagramme** für MAE, RMSE, R^2 erstellen

3. Zeitreihenplot: **Ist vs. Prognose** für alle Modelle
4. Optional: **Feature Importances** für Random Forest darstellen

---

🎯 Ziel

- Schneller Überblick, welches Modell die beste Prognose liefert
- Identifikation von Stärken / Schwächen klassischer vs. zeitreihen- bzw. Deep Learning-Modelle
- Grundlage für weitere Optimierungen oder produktive Nutzung

In [ ]:

```
# ====================================
# 8. Modellvergleich
# ====================================

# ------------------------------------
# Ergebnisse aller Modelle in DataFrame zusammenfassen
# ------------------------------------
rows = [
    {"model": name, "MAE": d["MAE"], "RMSE": d["RMSE"], "R2": d["R2"]}
    for name, d in results.items()
]

results_df = (
    pd.DataFrame(rows)
    .set_index("model")
    .sort_values("MAE")  # nach MAE aufsteigend sortieren
)

display(results_df)

# ------------------------------------
# Balkendiagramme für MAE, RMSE, R²
# ------------------------------------
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# MAE
axes[0].bar(results_df.index, results_df["MAE"])
axes[0].set_title("MAE")
axes[0].tick_params(axis="x", rotation=45)

# RMSE
axes[1].bar(results_df.index, results_df["RMSE"])
axes[1].set_title("RMSE")
axes[1].tick_params(axis="x", rotation=45)

# R²
axes[2].bar(results_df.index, results_df["R2"])
axes[2].set_title("R²")
axes[2].tick_params(axis="x", rotation=45)

plt.tight_layout()
plt.show()

# ------------------------------------
# Bestes Modell identifizieren
# ------------------------------------
```

```
best = results_df.index[0]
print(f"Bestes Modell: {best} (MAE={results_df.loc[best, 'MAE']:.3f})")
```

9️⃣ Vorhersagen visualisieren

In diesem Abschnitt werden die Vorhersagen der Modelle **gegenüber den echten Verkehrsdaten** dargestellt.

🔷 Ziele der Visualisierung

1. **Vergleich von Ist vs. Prognose**
   - Erkennen von Mustern, z. B. Rush-Hour, Wochenend-Effekte
2. **Modellvergleich auf Zeitachse**
   - Welches Modell folgt dem realen Verlauf am besten?
3. **Einschätzung der Vorhersagequalität**
   - Ergänzt die Metriken (MAE, RMSE, R^2) durch visuelle Analyse

---

📊 Vorgehensweise

- Linienplot über den Testzeitraum
- Jede Modellvorhersage als eigene Linie
- Originalwerte (`y_test`) als Referenz
- Optional: farbliche Hervorhebung des besten Modells

In [ ]:
```
# ====================================
# 9. Vorhersagen visualisieren
# ====================================

# Fenstergröße für Plot (max. 1 Woche)
test_window = min(len(y_test), 7 * 24)

# Anzahl Modelle und Subplot-Layout
n_models = len(results)
ncols = 2
nrows = int(np.ceil(n_models / ncols))

fig, axes = plt.subplots(nrows, ncols, figsize=(16, 4 * nrows), squeeze=False)

# -------------------------------------
# Plot Ist vs. Prognose für jedes Modell
# -------------------------------------
for (name, d), ax in zip(results.items(), axes.ravel()):
    # Echte Werte
    ax.plot(range(test_window), y_test[:test_window], label="Ist", linewidth=2)
    # Modellvorhersage
    ax.plot(range(test_window), d["pred"][:test_window], label="Prognose",
            linewidth=2, linestyle="--")

    ax.set_title(f"{name} (MAE={d['MAE']:.2f})")
    ax.set_xlabel("Stunde im Test")
    ax.set_ylabel("y")
```

```
    ax.legend()
    ax.grid(True, alpha=0.3)

# Leere Subplots ausblenden
for ax in axes.ravel()[n_models:]:
    ax.axis("off")

plt.tight_layout()
plt.show()
```

🔟 Optional: Random Forest Tuning

In diesem Abschnitt wird der **Random Forest Regressor** optimiert, um die Prognosequalität zu verbessern.

🔷 Ziele des Tunings

1. **Hyperparameter-Optimierung**:
   - Anzahl der Bäume (`n_estimators`)
   - Maximale Baumtiefe (`max_depth`)
   - Minimale Samples pro Blatt (`min_samples_leaf`)
2. **Leistungssteigerung**:
   - Bessere Modellanpassung ohne Overfitting
3. **Evaluation mit TimeSeriesSplit**:
   - Zeitreihen-konformer Cross-Validation
   - Vermeidung von Data Leakage

📊 Vorgehensweise

1. **Parameter-Raster definieren** für GridSearchCV
2. **TimeSeriesSplit** für trainierende Teilmengen
3. **GridSearchCV** zur Auswahl der besten Parameter
4. **Bestes Modell trainieren** und auf Testdaten evaluieren
5. Optional: **MLflow Logging** der Hyperparameter und Metriken

---

🎯 Ziel

- Den Random Forest optimal an die Verkehrszeitreihe anpassen
- Vergleich mit vorherigen Standardmodellen (Linear Regression, RF baseline, Prophet, LSTM)
- Grundlage für produktive Vorhersagen

In [ ]:
```
# ====================================
# 10. Optional: Random Forest Tuning
# ====================================

DO_TUNING = True
```

```python
if DO_TUNING:
    # ------------------------------------
    # TimeSeriesSplit für Cross-Validation
    # ------------------------------------
    tscv = TimeSeriesSplit(n_splits=5)

    # ------------------------------------
    # Parameter-Raster für GridSearchCV
    # ------------------------------------
    param_grid = {
        "n_estimators": [100, 200],
        "max_depth": [10, 15, 20],
        "min_samples_split": [2, 10]
    }

    # Basis-Modell
    base = RandomForestRegressor(random_state=RANDOM_SEED, n_jobs=-1)

    # GridSearchCV mit Zeitreihen-CV
    gs = GridSearchCV(
        base,
        param_grid=param_grid,
        cv=tscv,
        scoring="neg_mean_absolute_error",
        n_jobs=-1
    )

    # Training / Hyperparameter-Suche
    gs.fit(X_train, y_train)

    # Beste Parameter
    print(f"Beste Parameter: {gs.best_params_}")

    # Vorhersage auf Testdaten
    pred = gs.predict(X_test)
    mae, rmse, r2 = eval_metrics(y_test, pred)
    print(f"Tuned RF → MAE={mae:.3f} | RMSE={rmse:.3f} | R²={r2:.3f}")

    # ------------------------------------
    # Optional: MLflow Logging
    # ------------------------------------
    if USE_MLFLOW:
        with mlflow.start_run(run_name="RF GridSearch"):
            mlflow.log_params(gs.best_params_)
            mlflow.log_metrics({"mae": mae, "rmse": rmse, "r2": r2})
```

📝 Zusammenfassung

Dieses Notebook zeigt eine **komplette Pipeline für Verkehrsprognosen**:

- Daten synthetisch generieren oder CSV laden
- Explorative Datenanalyse (EDA)
- Feature Engineering (Lag, Rolling, Kalender, Wetter)
- Modelle: Linear Regression, Random Forest, Prophet, optional LSTM
- Modellvergleich & Visualisierung
- Optional: Random Forest Hyperparameter-Tuning

🔷 Nächste Schritte / Empfehlungen

- **Echte Verkehrsdaten anbinden** (CSV, API oder Datenbank)
- **Externe Features erweitern** (Wetter, Events, Ferienkalender)
- **Robustes Backtesting** mit Rolling Origin Cross-Validation
- **Deployment**: z. B. Streamlit-App oder REST-Service mit gespeicherten Modellartefakten