Datenanalyse in Big Data

Christopher Keibel
30341906
South Westphalia University of Applied Sciences

September 21, 2025

Abstract

Diese Arbeit beschreibt den Aufbau einer einfachen Datenanalyse-Pipeline in Apache Spark auf dem Datensatz *2015 Flight Delays and Cancellations*[1]. Die Pipeline liest Rohdaten mit explizitem Schema ein, persistiert sie in Parquet und reichert die Flugdaten um sprechende Airline- und Flughafeninformationen an. Anschließend werden die Daten gefiltert, um ausschließlich Verspätungen (ohne Streichungen) zu erfassen, bevor daraus Auswertungsmerkmale abgeleitet werden. Über Aggregationen werden dann ausgewählte Fragestellungen beantwortet. Ein einfaches Klassifikationsmodell demonstriert abschließend die Anbindung an Spark-ML.

1 Data Ingestion und Persistenz

Die Rohdaten liegen als CSV vor. Um konsistente Datentypen ohne Inferenz-Fehler zu garantieren, werden die Dateien mit einem *expliziten Schema* eingelesen und anschließend in das spaltenorientierte Format *Parquet* konvertiert. Dies verhindert die fehlerhafte Schema-Ableitung durch Spark und ermöglicht performante Folgeoperationen. Parquet reduziert Speicherbedarf und Lesezeit, da nur benötigte Spalten gelesen und Filter effizienter ausgewertet werden können. Die Daten werden nach Jahr und Monat partitioniert, damit zu einem späteren Zeitpunkt nur relevante Daten ausgewählt werden. Da der Datensatz ausschließlich Flugdaten aus dem Jahr 2015 enthält, ist die Jahrespartitionierung hier technisch nicht erforderlich. In einem realen Produktionsszenario mit mehrjährigen Daten würde jedoch die Jahrespartitionierung erhebliche Performance-Vorteile bringen, da zeitliche Abfragen (z.B. "alle Flüge aus 2016") nur die entsprechenden Ordner laden müssten. Um realitätsnahe Implementierungspatterns zu demonstrieren, wurde daher bewusst eine Partitionierung nach Jahr und Monat gewählt.

```
from pyspark.sql import types as T

# Define explicit Flights Schema
schema_flights = T.StructType([
    T.StructField("YEAR", T.IntegerType()),
    T.StructField("MONTH", T.IntegerType()),
    T.StructField("DAY", T.IntegerType()),
    T.StructField("AIRLINE", T.StringType()),
    ...
])

# Read Flights Table with Schema
fl_raw = (spark.read
```

[1] Datensatz - 2015 Flight Delays and Cancellations

```
  .option("header", True)
  .schema(schema_flights)
  .csv("data/raw/flights_small.csv"))

# Save as Parquet
(fl_raw.write.mode("overwrite")
  .partitionBy("YEAR","MONTH")
  .parquet("data/lake/flights_parquet"))
```

2 Enrichment (Joins)

Die Flugdaten enthalten Kürzel für Airline sowie Start- und Zielflughafen. Für verständliche Auswertungen werden diese Kürzel durch Klartextnamen ergänzt. Dazu werden die kleinen Referenztabellen `airlines` und `airports` geladen und per Join angefügt. Vor den Joins werden gezielt Spalten umbenannt und mit Aliasen versehen, um Namenskonflikte zu vermeiden und die Eindeutigkeit der resultierenden Spalten sicherzustellen. Da es sich um kleine Tabellen handelt, werden sie an alle Ausführungsprozesse übertragen (Broadcast). Dadurch lässt sich ein teurer Datenaustausch der größeren `flights` Tabelle vermeiden.

```
from pyspark.sql import functions as F

# Lazy Parquet Reading
flights = spark.read.parquet("data/lake/flights_parquet")
airlines = spark.read.parquet("data/lake/airlines_parquet")
airports = spark.read.parquet("data/lake/airports_parquet")

# Airlines Join
alN = airlines.withColumnRenamed("AIRLINE", "CARRIER_NAME")
flights_airlines_enriched = flights.join(
                                F.broadcast(alN),
                                flights.AIRLINE == alN.IATA_CODE,
                                "left")

# Airport Join
airports_origin = airports.select(F.col("IATA_CODE").alias("ORIGIN_CODE"),
                F.col("AIRPORT").alias("ORIGIN_AIRPORT_NAME"),
                F.col("STATE").alias("ORIGIN_STATE"))

airports_dest = airports.select(F.col("IATA_CODE").alias("DEST_CODE"),
                F.col("AIRPORT").alias("DEST_AIRPORT_NAME"),
                F.col("STATE").alias("DEST_STATE"))

flights_enriched = (flights_airlines_enriched
  .join(F.broadcast(airports_origin),
  flights_airlines_enriched.ORIGIN_AIRPORT == F.col("ORIGIN_CODE"),
  "left")
  .join(F.broadcast(airports_dest),
  flights_airlines_enriched.DESTINATION_AIRPORT == F.col("DEST_CODE"),
  "left"))
```

3 Transformationen und UDFs

Für die Ableitung von Auswertungsmerkmalen werden primär native Spark-Funktionen verwendet, da diese erheblich schneller sind als benutzerdefinierte Funktionen (UDFs). UDFs

kommen nur dort zum Einsatz, wo native Funktionen nicht ausreichen oder spezielle Geschäftslogik erforderlich ist.

Der folgende Code zeigt eine Mischung aus nativen Funktionen für Standard-Transformationen und UDFs nur für komplexe Transformationen wie das Mapping von Stornierungs-Codes[2] auf sprechende Labels.

```
code_mapping = {"A": "Air Carrier", "B": "Extreme Weather",
            "C": "National Aviation System", "D": "Security"}

@F.udf(T.StringType())
def cancel_code_to_label(c):
    if c is None:
        return "Not Canceled"
    return code_mapping.get(str(c).strip().upper(), "Other/Unknown")

flights_transformed = (flights
    .withColumn("IS_DELAYED_15", (F.col("ARRIVAL_DELAY") >= 15).cast("int")) # native
    .withColumn("DOW", F.date_format("FL_DATE", "E")) # native
    .withColumn("CANCEL_REASON_LABEL", cancel_code_to_label("CANCELLATION_REASON")) # UDF
)
```

4 Aggregation und Filterung

Die folgenden Beispiele zeigen systematische Filter- und Aggregationsstrategien zur Beantwortung verschiedener Fragestellungen in der Flugdatenanalyse.

Datenbereinigung. Ausschluss stornierter und umgeleiteter Flüge für eine saubere Analysebasis.

```
df = spark.read.parquet("data/lake/enriched_parquet")
df = df.filter((F.col("CANCELLED")==0) & (F.col("DIVERTED")==0))
```

Für die Verspätungsanalyse werden nur planmäßig durchgeführte Flüge berücksichtigt. Stornierte Flüge (CANCELLED=1) und umgeleitete Flüge (DIVERTED=1) werden ausgeschlossen, da diese nicht als klassische Verspätungen kategorisiert werden können.

Airline-Vergleich. Die Aggregation erstellt ein Ranking der Fluggesellschaften nach ihrer Pünktlichkeit.

```
by_airline = (df.groupBy("AIRLINE","CARRIER_NAME")
  .agg(F.count("*").alias("flights"),
       F.avg("IS_DELAYED_15").alias("p_delay"),
       F.avg("ARRIVAL_DELAY").alias("avg_arr_delay"))
  .filter("flights >= 500")
  .orderBy(F.desc("p_delay")))
```

Die Gruppierung erfolgt nach Airline-Code und Carrier-Name für die vollständige Identifikation der Fluggesellschaften. Berechnet werden die Gesamtanzahl der Flüge, der Anteil verspäteter Flüge und die durchschnittliche Verspätungsdauer. Der Filter auf mindestens 500 Flüge eliminiert kleinere Airlines mit statistisch wenig aussagekräftigen Daten. Die absteigende Sortierung nach Verspätungsanteil ordnet die Airlines nach ihrer Pünktlichkeit, beginnend mit den unpünktlichsten Airlines.

[2] IX. REPORTING THE CAUSES OF CANCELLED AND DELAYED FLIGHTS

Zeitliche Muster. Die stündliche Aggregation zeigt tageszeitabhängige Verspätungsmuster.

```
by_hour = (df.groupBy("DEP_HOUR")
  .agg(F.count("*").alias("n"),
       F.avg("IS_DELAYED_15").alias("p_delay"))
  .orderBy("DEP_HOUR"))
```

Die Gruppierung erfolgt nach der geplanten Abflugstunde (DEP_HOUR), die aus dem ursprünglichen Zeitformat abgeleitet wurde. Berechnet werden die Anzahl der Flüge pro Stunde und der Anteil verspäteter Flüge als Durchschnitt des binären Indikators IS_DELAYED_15. Die aufsteigende Sortierung nach Stunden erstellt eine chronologische Reihenfolge von 0 bis 23 Uhr. Diese Analyse prüft, ob Verspätungen zu bestimmten Tageszeiten häufiger auftreten. Zu Stoßzeiten können sich Verzögerungen verstärken, weil mehr Flugzeuge starten möchten als es die Kapazitäten an Start- und Landebahnen zulassen. Dadurch kann sich eine Verspätung auf die nachfolgenden Flüge übertragen.

Saisonale Kategorisierung. Die Jahreszeit-Aggregation untersucht wetterbedingte Verspätungsmuster.

```
season = (F.when(F.col("MONTH").isin(12,1,2), "winter")
            .when(F.col("MONTH").isin(3,4,5), "spring")
            .when(F.col("MONTH").isin(6,7,8), "summer")
            .otherwise("autumn"))
df = df.withColumn("SEASON", season)

by_season = (df.groupBy("SEASON")
  .agg(F.count("*").alias("n"),
       F.avg("IS_DELAYED_15").alias("p_delay"),
       F.avg("ARRIVAL_DELAY").alias("avg_arr_delay"))
  .orderBy("SEASON"))
```

Die Kategorisierung verwendet eine mehrstufige when/otherwise-Logik zur Zuordnung der Monate zu meteorologischen Jahreszeiten. Winter umfasst die Monate Dezember bis Februar, Frühling März bis Mai, Sommer Juni bis August, der Rest wird als Herbst klassifiziert. Diese neue Spalte SEASON wird dem DataFrame hinzugefügt. Die anschließende Gruppierung erfolgt nach den vier Jahreszeiten mit Berechnung der Fluganzahl, des Verspätungsanteils und der durchschnittlichen Verspätungsdauer. Die alphabetische Sortierung nach Jahreszeit erstellt eine konsistente Reihenfolge. Diese Analyse prüft, ob bestimmte Jahreszeiten anfälliger für Verspätungen sind. Die when/otherwise-Konstruktion zeigt dabei die Vorteile nativer Spark-Funktionen gegenüber benutzerdefinierten Funktionen für solche Kategorisierungsaufgaben.

5 Machine Learning Pipeline

Zur Vorhersage von Flugverspätungen wird eine vollständige Machine-Learning-Pipeline in Apache Spark implementiert. Das Ziel ist die binäre Klassifikation mittels logistischer Regression, um vorherzusagen, ob ein Flug eine Verspätung von mindestens 15 Minuten haben wird.

Datenaufbereitung und Feature Engineering. Der Datensatz wird für das Training aufbereitet, indem kategorische Daten zunächst indexiert und dann One-Hot encodiert werden. Anschließend werden alle Features in Spark ML kompatible Vektoren umgewandelt.

```
# Select Training Features
ml_data = (df
    .withColumn("label", F.col("IS_DELAYED_15").cast("double"))
    .select("label", "DEP_HOUR", "MONTH", "DISTANCE",
            "AIRLINE", "ORIGIN_AIRPORT", "DESTINATION_AIRPORT")
    .na.drop())

# Transform Categorical Features
categorical_columns = ["AIRLINE", "ORIGIN_AIRPORT", "DESTINATION_AIRPORT"]
string_indexers = [StringIndexer(inputCol=col, outputCol=f"{col}_indexed")
                   for col in categorical_columns]

# One-Hot Encoding of Categorical Features
one_hot_encoder = OneHotEncoder(
    inputCols=[f"{col}_indexed" for col in categorical_columns],
    outputCols=[f"{col}_encoded" for col in categorical_columns])

# Create Feature Vectors
feature_assembler = VectorAssembler(
    inputCols=["DEP_HOUR", "MONTH", "DISTANCE"] +
              [f"{col}_encoded" for col in categorical_columns],
    outputCol="features")
```

Training und Evaluation. Die Pipeline automatisiert alle zvor definierten Transformationsschritte und das Training. Die Evaluation erfolgt über die Accuracy zur Bewertung der Vorhersagequalität.

```
# Logistic Regression
logistic_regression = LogisticRegression(featuresCol="features", labelCol="label")

# Create ML-Pipeline inlcuding defined Preprocessing
ml_pipeline = Pipeline(stages=string_indexers +
                       [one_hot_encoder, feature_assembler, logistic_regression])

# Split data into train and test and fit the model
training_data, test_data = ml_data.randomSplit([0.8, 0.2], seed=42)
trained_model = ml_pipeline.fit(training_data)

# Evaluation
accuracy_evaluator = MulticlassClassificationEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="accuracy"
)

predictions = trained_model.transform(test_data)
acc = accuracy_evaluator.evaluate(predictions)
```