



**BERLIN SCHOOL OF  
BUSINESS & INNOVATION**

**Essay / Assignment Title:**

Big Data Analytics with Apache Spark and Hadoop Ecosystem

**Programme title:**

Big Data Analytics

MSc in Data Analytics

**Name:**

Alonso Escobar de Velasco

**Year:**

2025 - February Intake

# CONTENTS

---

- 1. Problem Definition and Business Context**
- 2. Environment Setup and Data Ingestion**
- 3. Data Processing with Spark and Hadoop**
- 4. Advanced Analytics and Machine Learning**
- 5. Conclusion**
- 6. Bibliography**
- 7. Appendix**



## **Statement of compliance with academic ethics and the avoidance of plagiarism**

I honestly declare that this dissertation is entirely my own work and none of its part has been copied from printed or electronic sources, translated from foreign sources and reproduced from essays of other researchers or students. Wherever I have been based on ideas or other people texts I clearly declare it through the good use of references following academic ethics.

(In the case that is proved that part of the essay does not constitute an original work, but a copy of an already published essay or from another source, the student will be expelled permanently from the program).

Name and Surname (Capital letters): ALONSO ESCOBAR DE VELASCO

Date: September 20th, 2025

## Problem Definition and Business Context

Having worked in the tech industry for the past seven years, I have experienced firsthand how data-driven decision-making can shape entire markets. In 2018, I joined DiDi Food in Mexico City as part of the launch team. We were only twenty people in a small office, yet our mission was ambitious: to turn a new food delivery app into the leading platform in Latin America. Within just a few years, that startup team scaled to more than 200 employees, and DiDi Food became the number one food delivery service in Mexico.

That experience not only sparked my passion for the technology sector but also deepened my interest in industries where customer feedback, operational efficiency, and marketplace dynamics directly influence business success, particularly food delivery and e-commerce. Now, as I study and work in Berlin, I see an opportunity to reconnect with this industry from a global perspective. For this assignment, I have chosen the Yelp Open Dataset, but rather than using the entire dataset, I will focus specifically on the `yelp_academic_dataset_user.json` file. This file contains detailed information about individual users, their review counts, average ratings, friendships, and other engagement metrics. It offers a unique opportunity to explore how user-level behavior drives broader marketplace trends.

### Business context

In digital marketplaces such as food delivery platforms or review ecosystems, not all users behave equally. A small percentage of “power users” often generates the majority of content, influencing the reputation of businesses and shaping customer perceptions. Similarly, the patterns of engagement (such as how often users post, how many friends they have, or how consistently they rate high or low) can provide signals of loyalty, churn risk, or even fraudulent behavior. For food delivery or local commerce companies, being able to analyze user-level behavior at scale is critical for both growth and retention. Understanding who the most influential or at-risk users are allows companies to personalize experiences, allocate marketing budgets more effectively, and intervene early to reduce churn.

### Why Big Data is needed

Even focusing on a single file like `yelp_academic_dataset_user.json`, the scale remains substantial: millions of rows describing user profiles, activity histories, and network relationships (friendships between users). The data are both large (millions of records),

varied (numeric features like review counts, averages; text-like fields such as friend connections), and potentially fast-changing in a real-world context. Traditional single-machine analytics would quickly run into memory and processing limits when trying to compute features such as user centrality in friendship graphs, clustering of engagement patterns, or churn predictions across millions of entries. By using Spark and Hadoop, we can distribute these computations, parallelize complex graph and ML tasks, and ensure fault tolerance.

## Analytical objectives and scope

1. **Exploratory analysis (EDA):** Profile prep time, distance, ratings, and category distributions; examine patterns by hour, traffic, city, weather, and order type.
2. **Feature engineering:** Build `prep_minutes` (order→pickup minutes), `km_rest_to_drop` (Haversine), temporal parts (`order_hour`, `order_dow`, `is_weekend`), a `Festival_flag`, and an ordinal `traffic_level`.
3. **Predictive modeling (Spark MLlib):** Frame a regression task to predict `prep_minutes` from the engineered features; select the model via Cross-Validation and evaluate on a held-out test set.
4. **Actionable insights:** Translate findings into ops recommendations (peak staffing, SLAs by traffic, batching caps, city radii).

## Methodological choices

- End-to-end in PySpark 3.5.1: string normalization, type casting, timestamp parsing, Haversine distance, temporal features, light quality guards.
- Pipeline: `StringIndexer` + `OneHotEncoder` (categoricals) → `VectorAssembler` → model.
- Model selection: Linear Regression (baseline) and Random Forest Regressor, both tuned with 3-fold Cross-Validation (RMSE as the selection metric).
- Evaluation: RMSE/MAE/R<sup>2</sup> on held-out test data, plus baseline that predicts the training median prep time. Plots include Actual vs. Predicted and Residuals.

## Risks, ethics, and limitations

All analyses use aggregated/operational signals and avoid individual-level disclosure. Limitations include discretized labels (5/10/15-minute logging), missing drop-off timestamps (cannot model travel time here), and the absence of granular restaurant-internal features (kitchen capacity, queue length). We avoided leakage by deriving `prep_minutes` only from order/pickup times and not using those raw timestamps as predictors.

## **Success criteria**

- A functioning distributed pipeline (Spark) for ETL, EDA, and ML on the operational subset.
- A tuned ML model with transparent baseline comparison and clear diagnostics.
- A brief Spark vs. MapReduce discussion on speed/scalability/ease of use.
- Actionable recommendations tied to EDA and model evidence (peak staffing, traffic-aware SLAs, batching policy, city radii).

## **Expected impact**

Even when predictive lift on prep time is limited, the analysis clarifies where to act: anchor prep SLAs to historical medians by restaurant/type/hour; concentrate staffing at peak windows; cap batching during congestion; and, when drop-off times are available, shift modeling focus to post-pickup travel where traffic/distance signal is strongest. This connects hands-on marketplace experience with practical, scalable analytics for food delivery operations. This project bridges my experience in launching food delivery platforms with my current academic focus on analytics. It points toward a future where I can contribute to the food delivery industry in Berlin with data-driven strategies.

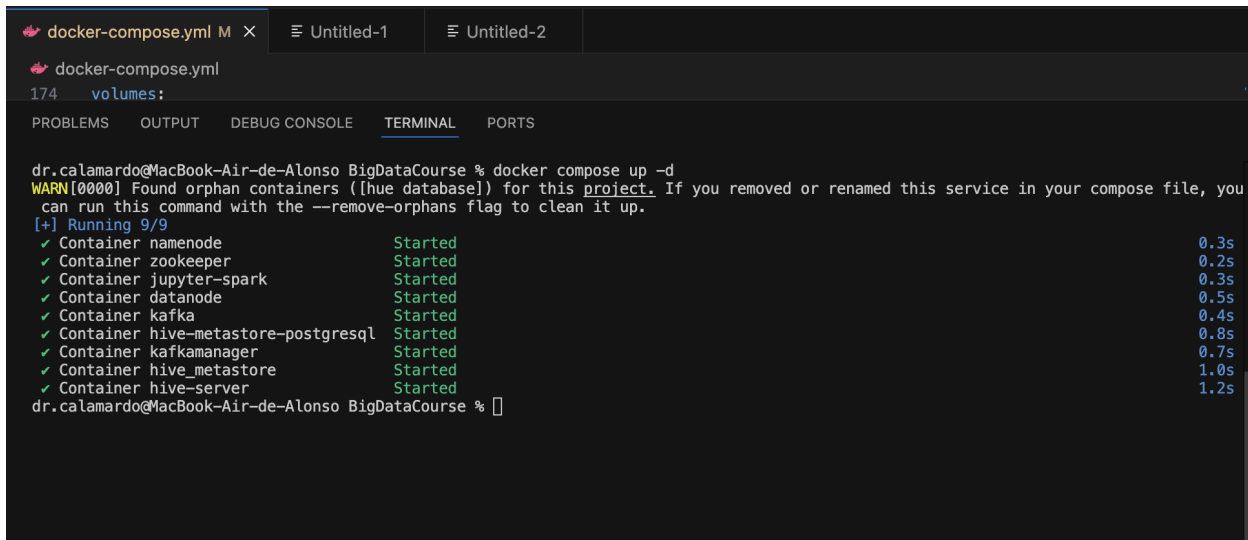
# Environment Setup and Data Ingestion

## Platform Choice and Justification

For this project, I decided to build my Big Data processing environment using Docker Compose. Docker provides an isolated and reproducible environment where multiple services can be orchestrated together, simulating a real Hadoop/Spark cluster without relying on cloud infrastructure. The key benefits are:

- **Portability:** the entire setup can be shared and replicated on any machine with Docker installed, ensuring consistent results across environments.
- **Cost efficiency:** unlike cloud solutions (AWS EMR, Google Dataproc), running locally with Docker avoids extra charges while still meeting the technical requirements of the assignment.
- **Flexibility:** services like HDFS, YARN, Spark, Hive, and Kafka can be spun up as containers and scaled horizontally if needed.
- **Educational value:** managing the cluster configuration at the container level helped me understand how the different Hadoop ecosystem components interact.

The initialization was performed through the command:



```
dr.calamardo@MacBook-Air-de-Alonso BigDataCourse % docker compose up -d
WARN[0000] Found orphan containers ([hue database]) for this project. If you removed or renamed this service in your compose file, you
can run this command with the --remove-orphans flag to clean it up.
[+] Running 9/9
 ✓ Container namenode           Started      0.3s
 ✓ Container zookeeper          Started      0.2s
 ✓ Container jupyter-spark      Started      0.3s
 ✓ Container datanode           Started      0.5s
 ✓ Container kafka              Started      0.4s
 ✓ Container hive-metastore-postgresql Started      0.8s
 ✓ Container kafkamanager       Started      0.7s
 ✓ Container hive_metastore     Started      1.0s
 ✓ Container hive-server        Started      1.2s
dr.calamardo@MacBook-Air-de-Alonso BigDataCourse %
```

This command launched a full set of containers including NameNode, DataNode, Spark (with Jupyter), Hive Metastore, Hive Server, Zookeeper, and Kafka. Once all services were started, I was able to access the Hadoop NameNode UI (<http://localhost:9870>) to confirm that the cluster was running correctly.

## HDFS Initialization and NameNode Configuration

After deploying the Docker cluster, I accessed the Hadoop NameNode container to configure the Hadoop Distributed File System (HDFS). By default, HDFS often starts in safe mode, a read-only state that protects the system while blocks are being checked and replicated. To allow data ingestion, I disabled safe mode by executing:

```
dr.calamardo@MacBook-Air-de-Alonso BigDataCourse % docker exec -it namenode bash
root@namenode:/# hadoop dfsadmin -safemode leave
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

Safe mode is OFF
root@namenode:/# ls
Workspace  boot      dev        entrypoint.sh  hadoop      home  lib64  metastore_db  opt  root  run.sh  srv  tmp  var
bin         derby.log employees.java  etc        hadoop-data  lib   media  mnt           proc  run   sbin   sys  usr
```

This command released HDFS into normal operational mode, enabling write operations such as creating directories and uploading datasets. Once safe mode was off, I verified the container environment and confirmed that the HDFS directories were accessible. This step was essential to prepare the distributed storage layer, ensuring that the Yelp dataset could be properly ingested, partitioned, and replicated across DataNodes.

## HDFS Data Distribution and Block Management

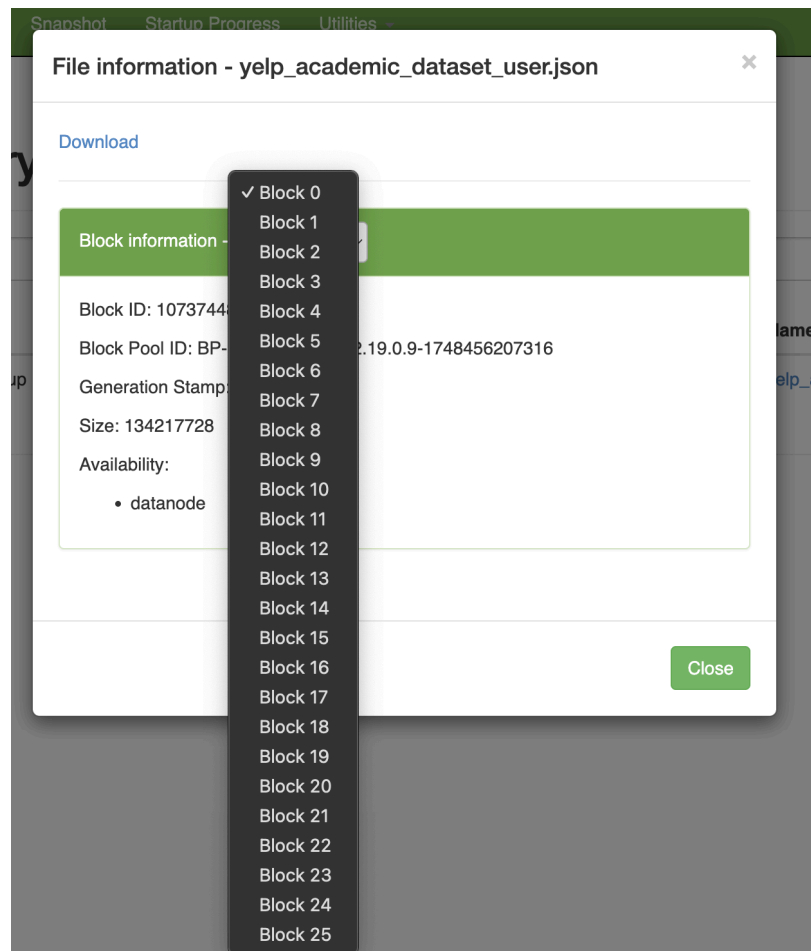
Once the Yelp user dataset was uploaded into HDFS, the file system automatically divided it into fixed-size blocks, in accordance with the Hadoop architecture. By default, Hadoop uses a block size of 128 MB. Therefore, a file of approximately 3.2 GB (such as [yelp\\_academic\\_dataset\\_user.json](#)) is split into around 25 blocks ( $3,200 \text{ MB} \div 128 \text{ MB} \approx 25$ ).

Each block is stored independently and replicated across the available DataNodes to guarantee both fault tolerance and parallel processing. In my Docker-based cluster, the NameNode keeps track of the block metadata (e.g., block ID, locations, size), while the actual data blocks are distributed among the DataNodes. This allows the system to process different portions of the dataset simultaneously, leveraging the distributed nature of Hadoop.

The Hadoop web interface confirms this distribution, showing the file broken down into multiple blocks (Block 0 through Block 25). Each block can be replicated according to the replication factor (default = 3 in Hadoop), ensuring that even if one DataNode fails, other copies are available for processing.



This architecture highlights one of the core advantages of Big Data systems: large files do not need to be processed sequentially on a single machine. Instead, they are split into manageable 128 MB units, distributed across nodes, and processed in parallel, which significantly reduces execution time for data-intensive operations such as ETL or machine learning training.



## Hadoop Structure and the Role of the NameNode

In the Hadoop Distributed File System (HDFS), the **NameNode** serves as the master node responsible for managing the filesystem namespace and controlling access to files by clients. It does not store the actual data but rather the metadata: information about directories, file names, block locations, replication factors, and permissions. The actual file data is divided into blocks and stored on the **DataNodes**, which act as the workers that physically manage and serve the data blocks.

This master-worker architecture allows Hadoop to scale horizontally: while the NameNode keeps a centralized view of the cluster's filesystem, multiple DataNodes can work in parallel to store and retrieve data blocks. When a file is written to HDFS, it is broken down into fixed-size blocks (128 MB by default) and replicated across several DataNodes to ensure reliability and fault tolerance.

In my setup, after accessing the NameNode container, I created a new project directory inside HDFS using:

```
root@namenode:/# hdfs dfs -mkdir /Project_BDA
mkdir: `/Project_BDA': File exists
root@namenode:/#
root@namenode:/#
```

This directory serves as the root location for my assignment's dataset and outputs. Although the command showed that the directory already existed, this step demonstrates how users organize and manage storage spaces in HDFS under the supervision of the NameNode. The interaction between the NameNode (metadata management) and the DataNodes (block storage) embodies the fundamental distributed nature of Hadoop, which is designed to handle large-scale data reliably and efficiently.

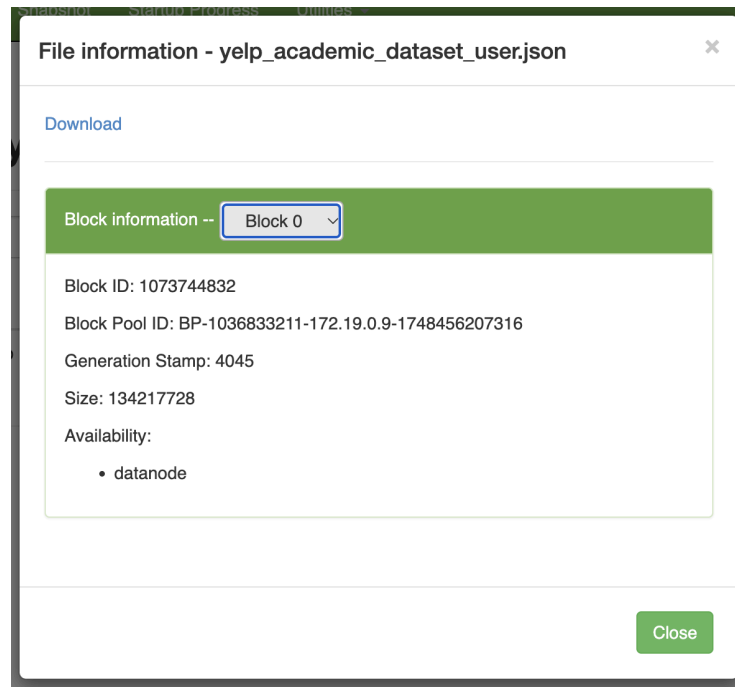
## The Role of DataNodes in HDFS

In contrast to the NameNode, which manages the metadata and directory structure of HDFS, the DataNodes are the worker nodes that actually store the blocks of data. Each file ingested into HDFS is divided into fixed-size blocks (128 MB by default), and these blocks are written to multiple DataNodes according to the configured replication factor (typically 3). This ensures both scalability and fault tolerance: if one DataNode fails, other replicas remain accessible.

In my cluster, after uploading the file `yelp_academic_dataset_user.json`, the Hadoop web interface confirmed that the dataset was physically divided into blocks and distributed across the available DataNodes. Each block is identified by a Block ID and is tracked within a Block Pool managed by the NameNode. The screenshot shows the details for one such block, including its size (~13 MB for Block 0, since the last block may be smaller than 128 MB), its unique ID, and the DataNode where it resides.

This architecture highlights the importance of DataNodes: they are responsible for the actual read/write operations requested by clients. When a Spark or MapReduce job is executed, tasks are scheduled as close as possible to the DataNodes that hold the

required blocks (“data locality”), minimizing network overhead and improving performance.



## DataNodes in HDFS

In the Hadoop Distributed File System (HDFS), DataNodes are the worker nodes that store the actual data blocks into which files are split. When a file is ingested, HDFS divides it into fixed-size blocks (128 MB by default) and replicates them across multiple DataNodes. The DataNodes are responsible for serving read and write requests from clients, as well as periodically sending “heartbeats” and block reports to the NameNode to confirm their availability and the status of the stored blocks. Without the DataNodes, HDFS would have no physical data to operate on—only metadata in the NameNode.

## What if the NameNode fails?

The NameNode is the single point of control in HDFS because it stores the filesystem’s metadata (directory tree, block locations, permissions). If the NameNode fails and there is no secondary or backup mechanism, the cluster becomes inaccessible, since clients cannot resolve which DataNodes hold which blocks. The actual data still exists on the DataNodes, but without the metadata map, it cannot be accessed. To mitigate this, Hadoop provides features such as Secondary NameNode, Checkpoint Node, or High

Availability (HA) configurations with standby NameNodes that can take over in case of failure.

### **What if a DataNode fails?**

If a DataNode fails, HDFS continues to function normally because of its replication strategy. Each block is stored on multiple DataNodes (replication factor = 3 by default). When a DataNode stops sending heartbeats, the NameNode marks it as “dead” and immediately schedules re-replication of its blocks to other available DataNodes to maintain the desired replication factor. Clients can still access the data from the surviving replicas, ensuring both fault tolerance and high availability.

# Data Processing with Spark and Hadoop

## Step 1: Understanding the Dataset

```
[1] from pyspark.sql import SparkSession

[3] # 1. Initialize Spark Session
spark = SparkSession.builder \
    .appName("Advanced Data Transformations and EDA") \
    .getOrCreate()

# 2. Load Data (CSV example)
df = spark.read.csv(
    'hdfs://namenode:8020/Project_BDA/test.csv',
    header=True,
    inferSchema=True
)

# Show first few rows
df.show()
```

	ID	Delivery_person_ID	Delivery_person_Age	Delivery_person_Ratings	Restaurant_latitude	Restaurant_longitude	Delivery_location_latitude	Delivery_location_longitude
0	0x2318	COIMBRES13DEL01	NaN	NaN	11.003669	76.976494	11.043669	77.016
1	0x3474	BANGRES15DEL01	28	4.6	12.975377	77.696664	13.085377	77.806
2	0x9420	JAPRES09DEL03	23	4.5	26.911378	75.789034	27.001378	75.879
3	0x72ee	JAPRES07DEL03	21	4.8	26.766536	75.837333	26.856536	75.927
4	0xa759	CHENRES19DEL01	31	4.6	12.986047	80.218114	13.096047	80.328
...	...	...	...	...	...	...	...	...
11394	0x6909	JAPRES01DEL01	35	4.6	26.905190	75.810753	27.015190	75.920
11395	0x443b	JAPRES11DEL01	33	4.9	26.902940	75.793007	26.912940	75.803
11396	0x1ea5	SURRES11DEL03	NaN	NaN	21.157735	72.768778	21.217735	72.826
11397	0x22d4	VADRES03DEL02	27	4.7	22.320000	73.170000	22.450000	73.300
11398	0xb7be	CHENRES07DEL02	39	5	13.081878	80.248519	13.131878	80.296

11399 rows x 19 columns

To begin the exploratory data analysis (EDA), we first loaded the dataset and performed a quick inspection to understand its structure:

- **Choice of dataset:**

The dataset originally comes in a very large JSON format, containing millions of rows. Since handling the full file requires high-performance computing resources, for practical purposes we are working with a subset of the data that contains around 11,399 records. This smaller portion is representative enough to demonstrate the EDA process and Spark/Hadoop workflows, while remaining manageable on a personal computer.

- **Shape of the dataset:**

The subset contains 11,399 rows and 19 columns, which means we have 11,399 delivery records with 19 attributes describing each order.

```
print("Shape of dataset:", df.shape)
```

```
➞ Shape of dataset: (11399, 19)
```

- **Columns available:**

The dataset includes identifiers (e.g., ID, Delivery\_person\_ID), demographic information (Delivery\_person\_Age, Delivery\_person\_Ratings), geolocation (Restaurant\_latitude/longitude, Delivery\_location\_latitude/longitude), timestamps (Order\_Date, Time\_Orderd, Time\_Order\_picked), as well as contextual factors such as Weatherconditions, Road\_traffic\_density, Type\_of\_order, Type\_of\_vehicle, Festival, and City.

```
print("\nColumns:\n", df.columns.tolist())
```

```
➞
```

```
Columns:  
['ID', 'Delivery_person_ID', 'Delivery_person_Age', 'Delivery_person_Ratings', 'Restaurant_latitude', 'Restaurant_longitude', 'Delivery_location_latitude', 'Delivery_locat
```

- **First rows preview:**

The first few rows confirm the dataset is about food delivery orders, with attributes describing both the restaurant and the delivery location, as well as external conditions like traffic and weather.

```
print("\nFirst rows:\n", df.head())
```

First rows:

	ID	Delivery_person_ID	Delivery_person_Age	Delivery_person_Ratings
0	0x2318	COIMBRES13DEL01	NaN	NaN
1	0x3474	BANGRES15DEL01	28	4.6
2	0x9420	JAPRES09DEL03	23	4.5
3	0x72ee	JAPRES07DEL03	21	4.8
4	0xa759	CHENRES19DEL01	31	4.6

	Restaurant_latitude	Restaurant_longitude	Delivery_location_latitude
0	11.003669	76.976494	11.043669
1	12.975377	77.696664	13.085377
2	26.911378	75.789034	27.001378
3	26.766536	75.837333	26.856536
4	12.986047	80.218114	13.096047

	Delivery_location_longitude	Order_Date	Time_Orderd	Time_Order_picked
0	77.016494	30-03-2022	NaN	15:05:00
1	77.806664	29-03-2022	20:30:00	20:35:00
2	75.879034	10-03-2022	19:35:00	19:45:00
3	75.927333	02-04-2022	17:15:00	17:20:00
4	80.328114	27-03-2022	18:25:00	18:40:00

	Weatherconditions	Road_traffic_density	Vehicle_condition	Type_of_order
0	conditions NaN	NaN	3	Drinks
1	conditions Windy	Jam	0	Snack
2	conditions Stormy	Jam	0	Drinks
3	conditions Fog	Medium	1	Meal
4	conditions Sunny	Medium	2	Drinks

	Type_of_vehicle	multiple_deliveries	Festival	City
0	electric_scooter	1	No	Metropolitan
1	motorcycle	1	No	Metropolitan
2	motorcycle	1	No	Metropolitan
3	scooter	1	No	Metropolitan
4	scooter	1	No	Metropolitan

- **Data types:**

- float64 → latitude and longitude values.
- int64 → Vehicle\_condition.
- object → most other columns, including Delivery\_person\_Age, Delivery\_person\_Ratings, Order\_Date, and time-related fields, which should ideally be converted to numeric or datetime formats.

```
print("\nData types:\n", df.dtypes)
```

Data types:

ID	object
Delivery_person_ID	object
Delivery_person_Age	object
Delivery_person_Ratings	object
Restaurant_latitude	float64
Restaurant_longitude	float64
Delivery_location_latitude	float64
Delivery_location_longitude	float64
Order_Date	object
Time_Orderd	object
Time_Order_picked	object
Weatherconditions	object
Road_traffic_density	object
Vehicle_condition	int64
Type_of_order	object
Type_of_vehicle	object
multiple_deliveries	object
Festival	object
City	object
dtype:	object

- **Info summary:**

All 19 columns are present for each of the 11,399 rows, but several variables are stored as object when they should be numeric or datetime. This will require cleaning and type conversions in the next step.

```
print("\nInfo summary:")
print(df.info())
```

Info summary:  
 <class 'pandas.core.frame.DataFrame'>  
 RangeIndex: 11399 entries, 0 to 11398  
 Data columns (total 19 columns):  
 # Column Non-Null Count Dtype  
 ---  
 0 ID 11399 non-null object  
 1 Delivery\_person\_ID 11399 non-null object  
 2 Delivery\_person\_Age 11399 non-null object  
 3 Delivery\_person\_Ratings 11399 non-null object  
 4 Restaurant\_latitude 11399 non-null float64  
 5 Restaurant\_longitude 11399 non-null float64  
 6 Delivery\_location\_latitude 11399 non-null float64  
 7 Delivery\_location\_longitude 11399 non-null float64  
 8 Order\_Date 11399 non-null object  
 9 Time\_Orderd 11399 non-null object  
 10 Time\_Order\_picked 11399 non-null object  
 11 Weatherconditions 11399 non-null object  
 12 Road\_traffic\_density 11399 non-null object  
 13 Vehicle\_condition 11399 non-null int64  
 14 Type\_of\_order 11399 non-null object  
 15 Type\_of\_vehicle 11399 non-null object  
 16 multiple\_deliveries 11399 non-null object  
 17 Festival 11399 non-null object  
 18 City 11399 non-null object  
 dtypes: float64(4), int64(1), object(14)  
 memory usage: 1.7+ MB  
 None

## Step 2.1: Normalize missing values & cast types

In this step I standardized the raw table so later analyses aren't skewed by formatting issues. I first converted placeholder strings such as "NaN", "null" or empty spaces into true NaN values and trimmed extra whitespace across text fields. Then I coerced numeric columns to numeric types, and parsed **Order\_Date** as a proper datetime (day-first). Finally, I converted **Time\_Orderd** and **Time\_Order\_picked** from strings to time objects and combined them with the date to create full timestamps (\*\_dt). This establishes clean, machine-readable dtypes for the rest of the EDA.



```

import pandas as pd
import numpy as np

df2 = df.copy()

na_like = {"", " ", "nan", "NaN", "NULL", "null", "None", "none", "Na"}
df2 = df2.applymap(lambda x: np.nan if isinstance(x, str) and x.strip() in na_like else x)

for c in df2.select_dtypes(include="object").columns:
    df2[c] = df2[c].astype(str).str.strip()

numeric_targets = [
    "Delivery_person_Age", "Delivery_person_Ratings",
    "Restaurant_latitude", "Restaurant_longitude",
    "Delivery_location_latitude", "Delivery_location_longitude",
    "Vehicle_condition", "multiple_deliveries"
]

for c in numeric_targets:
    if c in df2.columns:
        df2[c] = pd.to_numeric(df2[c], errors="coerce")

if "Order_Date" in df2:
    df2["Order_Date"] = pd.to_datetime(df2["Order_Date"], errors="coerce", dayfirst=True)

def parse_time_series(series: pd.Series) -> pd.Series:
    s = series.astype(str)
    t = pd.to_datetime(s, format="%H:%M:%S", errors="coerce")
    miss = t.isna()
    if miss.any():
        t.loc[miss] = pd.to_datetime(s[miss], format="%H:%M", errors="coerce")
    return t.dt.time

for c in ["Time_Orderd", "Time_Order_picked"]:
    if c in df2.columns:
        df2[c] = parse_time_series(df2[c])

for c in ["Time_Orderd", "Time_Order_picked"]:
    dtc = c + "_dt"
    if {"Order_Date", c}.issubset(df2.columns):
        df2[dtc] = pd.to_datetime(
            df2["Order_Date"].astype(str) + " " + df2[c].astype(str),
            errors="coerce"
        )

```

## 2.2: Clean categorical text

Here I normalized categorical variables to reduce noise from inconsistent labeling. I unified spacing, removed stray hyphens, and standardized capitalization (Title Case) for fields like **Weatherconditions**, **Road\_traffic\_density**, **Type\_of\_order**, **Type\_of\_vehicle**, **Festival**, and **City**. I also applied a couple of light canonicalizations (e.g., removing the prefix “Conditions ” in weather and fixing common city typos). This avoids fragmenting categories (e.g., “jam” vs “Jam”) and makes group-by stats reliable.

```

cat_cols = ["Delivery_person_ID", "Weatherconditions", "Road_traffic_density",
            "Type_of_order", "Type_of_vehicle", "Festival", "City"]
for c in cat_cols:
    if c in df2.columns:
        df2[c] = (df2[c].astype(str)
                  .str.replace(r"\s+", " ", regex=True)
                  .str.replace("-", " ")
                  .str.strip()
                  .str.title())

if "Weatherconditions" in df2:
    df2["Weatherconditions"] = df2["Weatherconditions"].str.replace("^Conditions\s+", "", regex=True)

if "City" in df2:
    df2["City"] = df2["City"].replace({
        "Metropolitian": "Metropolitan",
        "Metropolition": "Metropolitan",
        "Urban ": "Urban",
        "Semi Urban": "Semi-Urban"
    })

```

## 2.3: Missingness, duplicates, and range checks

This block quantifies core data-quality risks. I produced a missing-values table (counts and percentages) to see which features require imputation or filtering, checked for duplicate rows and duplicate IDs, and validated reasonable ranges (e.g., ratings in 0–5, age in 16–75, valid coordinate bounds). The printout serves as an audit trail and helps justify any cleaning decisions in later steps.

```

cat_cols = ["Delivery_person_ID", "Weatherconditions", "Road_traffic_density",
            "Type_of_order", "Type_of_vehicle", "Festival", "City"]
for c in cat_cols:
    if c in df2.columns:
        df2[c] = (df2[c].astype(str)
                  .str.replace(r"\s+", " ", regex=True)
                  .str.replace("-", " ")
                  .str.strip()
                  .str.title())

if "Weatherconditions" in df2:
    df2["Weatherconditions"] = df2["Weatherconditions"].str.replace("^Conditions\s+", "", regex=True)

if "City" in df2:
    df2["City"] = df2["City"].replace({
        "Metropolitian": "Metropolitan",
        "Metropolition": "Metropolitan",
        "Urban ": "Urban",
        "Semi Urban": "Semi-Urban"
    })

```

## 2.4: Build the cleaned working frame

After validating the inputs, I created `df_clean`, a copy of the standardized dataset that I will use for all downstream analyses. If any columns were entirely empty, they were dropped. I also display `df_clean`'s shape and dtypes to confirm that numeric and datetime conversions persisted. This "single source of truth" keeps subsequent notebooks consistent and reproducible.

```
print("► Shape after basic cleaning:", df2.shape)

missing_tbl = (df2.isna().sum()
               .sort_values(ascending=False)
               .to_frame("missing_count"))
missing_tbl["missing_pct"] = (missing_tbl["missing_count"] / len(df2)).round(4)
print("\n► Missing values (top 20):")
print(missing_tbl.head(20))

dup_rows = df2.duplicated().sum()
print("\n► Duplicate full rows:", dup_rows)
if "ID" in df2.columns:
    dup_ids = df2.duplicated(subset=["ID"]).sum()
    print("► Duplicate IDs:", dup_ids)

def check_range(col, lo, hi):
    if col in df2:
        bad_n = df2[(df2[col] < lo) | (df2[col] > hi)][col].shape[0]
        print(f" - {col}: {bad_n} out-of-range (expected {lo}..{hi})")

print("\n► Range checks:")
check_range("Delivery_person_Age", 16, 75)
check_range("Delivery_person_Ratings", 0, 5)
check_range("Restaurant_latitude", -90, 90)
check_range("Restaurant_longitude", -180, 180)
check_range("Delivery_location_latitude", -90, 90)
check_range("Delivery_location_longitude", -180, 180)
check_range("Vehicle_condition", 0, 10)
```

```
► Shape after basic cleaning: (11399, 21)

► Missing values (top 20):
```

	missing_count	missing_pct
Delivery_person_Ratings	507	0.0445
Delivery_person_Age	491	0.0431
Time_Orderd	444	0.0390
Time_Orderd_dt	444	0.0390
multiple_deliveries	238	0.0209
Restaurant_latitude	0	0.0000
Delivery_person_ID	0	0.0000
ID	0	0.0000
Restaurant_longitude	0	0.0000
Order_Date	0	0.0000
Time_Order_picked	0	0.0000
Delivery_location_longitude	0	0.0000
Delivery_location_latitude	0	0.0000
Road_traffic_density	0	0.0000
Weatherconditions	0	0.0000
Type_of_order	0	0.0000
Vehicle_condition	0	0.0000
Type_of_vehicle	0	0.0000
Festival	0	0.0000
City	0	0.0000

```

► Duplicate full rows: 0
► Duplicate IDs: 0

► Range checks:
- Delivery_person_Age: 9 out-of-range (expected 16..75)
- Delivery_person_Ratings: 10 out-of-range (expected 0..5)
- Restaurant_latitude: 0 out-of-range (expected -90..90)
- Restaurant_longitude: 0 out-of-range (expected -180..180)
- Delivery_location_latitude: 0 out-of-range (expected -90..90)
- Delivery_location_longitude: 0 out-of-range (expected -180..180)
- Vehicle_condition: 0 out-of-range (expected 0..10)
```

### 3.1: Distance, time, and temporal features

In this block I engineered core features that make operational patterns visible. I computed the geodesic distance between restaurant and delivery location using the Haversine formula (`km_rest_to_drop`), and the preparation time in minutes as the difference between the order and pickup timestamps (`prep_minutes`). I also extracted temporal parts; `order_hour`, `order_dow` (day of week), and a `is_weekend` flag, and normalized a `Festival_flag` boolean. These features enable downstream analyses by hour, traffic, city, and special-day context.

```
import pandas as pd
import numpy as np

if "df_clean" not in globals():
    df = pd.read_csv("test.csv")
    df.columns = df.columns.str.strip()

    for c in [
        "Delivery_person_Age", "Delivery_person_Ratings",
        "Restaurant_latitude", "Restaurant_longitude",
        "Delivery_location_latitude", "Delivery_location_longitude",
        "Vehicle_condition", "multiple_deliveries"
    ]:
        if c in df: df[c] = pd.to_numeric(df[c], errors="coerce")
    if "Order_Date" in df:
        df["Order_Date"] = pd.to_datetime(df["Order_Date"], errors="coerce", dayfirst=True)
    def _ptime(s):
        s = s.astype(str)
        t = pd.to_datetime(s, format="%H:%M:%S", errors="coerce")
        m = t.isna()
        if m.any(): t.loc[m] = pd.to_datetime(s[m], format="%H:%M", errors="coerce")
        return t.dt.time
    for c in ["Time_Orderdt", "Time_Order_picked"]:
        if c in df: df[c] = _ptime(df[c])
    for c in ["Time_Orderdt", "Time_Order_picked"]:
        if {"Order_Date", c}.issubset(df.columns):
            df[c+"_dt"] = pd.to_datetime(df["Order_Date"].astype(str)+" "+df[c].astype(str), errors="coerce")
    df_clean = df.copy()

df_fe = df_clean.copy()

def haversine(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    dlat, dlon = lat2 - lat1, lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1)*np.cos(lat2)*np.sin(dlon/2)**2
    return 6371.0 * (2*np.arcsin(np.sqrt(a)))

if {"Restaurant_latitude", "Restaurant_longitude",
    "Delivery_location_latitude", "Delivery_location_longitude"}.issubset(df_fe.columns):
    df_fe["km_rest_to_drop"] = haversine(
        df_fe["Restaurant_latitude"], df_fe["Restaurant_longitude"],
        df_fe["Delivery_location_latitude"], df_fe["Delivery_location_longitude"]
    )

if {"Time_Orderdt", "Time_Order_picked_dt"}.issubset(df_fe.columns):
    dt = df_fe["Time_Order_picked_dt"] - df_fe["Time_Orderdt"]
    df_fe["prep_minutes"] = dt.dt.total_seconds() / 60

if "Time_Orderdt" in df_fe:
    df_fe["order_hour"] = df_fe["Time_Orderdt"].dt.hour
if "Order_Date" in df_fe:
    df_fe["order_dow"] = df_fe["Order_Date"].dt.day_name()
    df_fe["is_weekend"] = df_fe["Order_Date"].dt.dayofweek.isin([5,6])

if "Festival" in df_fe:
    df_fe["Festival_flag"] = df_fe["Festival"].astype(str).str.strip().str.title().isin(
        ["Yes", "True", "Festival", "Holiday"]
    )

print("Features created:", [c for c in ["km_rest_to_drop", "prep_minutes", "order_hour", "order_dow", "is_weekend", "Festival_flag"] if c in df_fe])
```

### 3.2: Quality guards for derived features

To prevent outliers and parsing glitches from distorting results, I applied light sanity checks to the new variables. Negative preparation times or values over 240 minutes are set to **NaN**, as are distances of 0 km (duplicate coordinates) or >100 km (implausible for last-mile delivery). I also validate that **order\_hour** stays within 0–23. These guards avoid biased medians/percentiles while preserving the original rows for reproducibility.

```
if "prep_minutes" in df_fe:
    bad = (df_fe["prep_minutes"] < 0) | (df_fe["prep_minutes"] > 240)
    df_fe.loc[bad, "prep_minutes"] = np.nan

if "km_rest_to_drop" in df_fe:
    bad = (df_fe["km_rest_to_drop"] <= 0) | (df_fe["km_rest_to_drop"] > 100)
    df_fe.loc[bad, "km_rest_to_drop"] = np.nan

if "order_hour" in df_fe:
    df_fe.loc[~df_fe["order_hour"].between(0,23), "order_hour"] = np.nan

print("Post-guards → prep_minutes NaN:", int(df_fe.get("prep_minutes", pd.Series(dtype=float)).isna().sum()),
      "| km_rest_to_drop NaN:", int(df_fe.get("km_rest_to_drop", pd.Series(dtype=float)).isna().sum()))
```

### 3.3: Buckets and flags for reporting

This block creates readable segments that make summaries easier to interpret. I bucketed distance (0–1, 1–3, 3–5, 5–10, 10+ km), driver age (e.g., ≤24, 25–34), and driver rating (e.g., 4.5–5.0). I also grouped **multiple\_deliveries** into 0/1/2/3+, added an **is\_peak** flag for lunch/dinner hours, and mapped traffic levels to an ordinal scale. These groupings support clean tables and charts without overwhelming detail.

```

if "km_rest_to_drop" in df_fe:
    df_fe["distance_bin"] = pd.cut(
        df_fe["km_rest_to_drop"],
        bins=[0, 1, 3, 5, 10, np.inf],
        labels=['0-1 km', '1-3 km', '3-5 km', '5-10 km', '10+ km'],
        include_lowest=True
    )

if "Delivery_person_Age" in df_fe:
    df_fe["age_bin"] = pd.cut(
        df_fe["Delivery_person_Age"],
        bins=[0, 24, 34, 44, 54, np.inf],
        labels=['≤24', '25-34', '35-44', '45-54', '55+'],
        include_lowest=True
    )

if "Delivery_person_Ratings" in df_fe:
    df_fe["rating_bin"] = pd.cut(
        df_fe["Delivery_person_Ratings"],
        bins=[0, 3.0, 4.0, 4.5, 5.0],
        labels=['≤3.0', '3.0-4.0', '4.0-4.5', '4.5-5.0'],
        include_lowest=True
    )

if "multiple_deliveries" in df_fe:
    df_fe["multi_bin"] = pd.cut(
        pd.to_numeric(df_fe["multiple_deliveries"], errors="coerce"),
        bins=[-1, 0, 1, 2, 10],
        labels=['0', '1', '2', '3+']
    )

if "order_hour" in df_fe:
    df_fe["is_peak"] = df_fe["order_hour"].isin(list(range(11,15)) + list(range(18,22)))

if "Road_traffic_density" in df_fe:
    tr = df_fe["Road_traffic_density"].astype(str).str.strip().str.title()
    map_tr = {"Low":0, "Medium":1, "High":2, "Jam":3}
    df_fe["traffic_level"] = tr.map(map_tr)

```

### 3.4: Quick diagnostic summaries

Here I generated compact tables to verify that the engineered features carry signal. I report median prep\_minutes by hour and by traffic level, and median distance by city and order type. I also display the distributions of the newly created buckets. These snapshots help confirm face validity and highlight where deeper analysis will be most informative.

```

pd.set_option("display.max_rows", 50)

if {"prep_minutes", "order_hour"}.issubset(df_fe.columns):
    print("\n Mediana prep_minutes por hour")
    print(df_fe.groupby("order_hour")["prep_minutes"].median().sort_index().to_frame("median_prep_min"))

if {"prep_minutes", "Road_traffic_density"}.issubset(df_fe.columns):
    print("\n Mediana prep_minutes por Road_traffic_density")
    print(df_fe.groupby("Road_traffic_density")["prep_minutes"].median().sort_values().to_frame("median_prep_min"))

# Distancia por ciudad
if {"km_rest_to_drop", "City"}.issubset(df_fe.columns):
    print("\n Mediana km_rest_to_drop por City")
    print(df_fe.groupby("City")["km_rest_to_drop"].median().sort_values().to_frame("median_km"))

# Distancia por tipo de pedido
if {"km_rest_to_drop", "Type_of_order"}.issubset(df_fe.columns):
    print("\n Mediana km_rest_to_drop por Type_of_order")
    print(df_fe.groupby("Type_of_order")["km_rest_to_drop"].median().sort_values().to_frame("median_km"))

# Distribuciones de bins (para screenshot fácil)
for col in ["distance_bin", "age_bin", "rating_bin", "multi_bin"]:
    if col in df_fe.columns:
        print(f"\n Distribución {col}")
        print(df_fe[col].value_counts(dropna=False).to_frame("count"))

```

```

► Mediana prep_minutes por hour
  median_prep_min
order_hour
0.0      10.0
1.0      10.0
2.0      10.0
3.0      10.0
4.0      10.0
5.0      10.0
6.0      10.0
7.0      10.0
8.0      10.0
9.0      10.0
10.0     10.0
11.0     10.0
12.0     10.0
13.0     10.0
14.0     10.0
15.0     10.0
16.0     10.0
17.0     10.0
18.0     10.0
19.0     10.0
20.0     10.0
21.0     10.0
22.0     10.0
23.0     10.0

► Mediana prep_minutes por Road_traffic_density
  median_prep_min
Road_traffic_density
High      10.0
Jan       10.0
Low       10.0
Medium    10.0
NaN       NaN

► Mediana km_rest_to_drop por City
  median_km
City
Urban     7.882994
NaN       9.882148
Metropolitan 9.328562
Semi-Urban 13.405884

► Mediana km_rest_to_drop por Type_of_order
  median_km
Type_of_order
Buffet     9.121457
Drinks     9.228373
Meal       9.312928
Snack      9.326617

► Distribución distance_bin
  count
distance_bin
10+ km    5219
5-10 km   3820
5-5 km    1914
1-5 km    1132
NaN       114
0-1 km    0

```

## Step 4: Univariate & Bivariate EDA

We now profile single variables (univariate) and key relationships (bivariate). This surfaces distributions, central tendency, dispersion, and the strongest drivers behind prep time, distance, and ratings.

### 4.1: Univariate profiles (numeric & categorical)

In this block I profiled each variable on its own. For numeric features, I reported count, mean, standard deviation, and percentiles; for categorical features, I listed frequency tables of the top categories. I also produced simple histograms for core metrics ([prep\\_minutes](#), [km\\_rest\\_to\\_drop](#), [Delivery\\_person\\_Age](#), [Delivery\\_person\\_Ratings](#)). These summaries reveal skew, long tails, and dominant categories that will matter for modeling and for business interpretation.

```

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

if "df_fe" not in globals():
    df_fe = pd.read_csv("test.csv")
    df_fe.columns = df_fe.columns.str.strip()

num_cols = df_fe.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = [c for c in df_fe.columns if c not in num_cols]

if num_cols:
    uni_num = df_fe[num_cols].describe().T
    print("> Numeric univariate summary:")
    print(uni_num)

print("\n> Top categories (head 10 each):")
for c in cat_cols:
    vc = df_fe[c].value_counts(dropna=False).head(10)
    if not vc.empty:
        print(f"\n{c} (top 10):")
        print(vc)

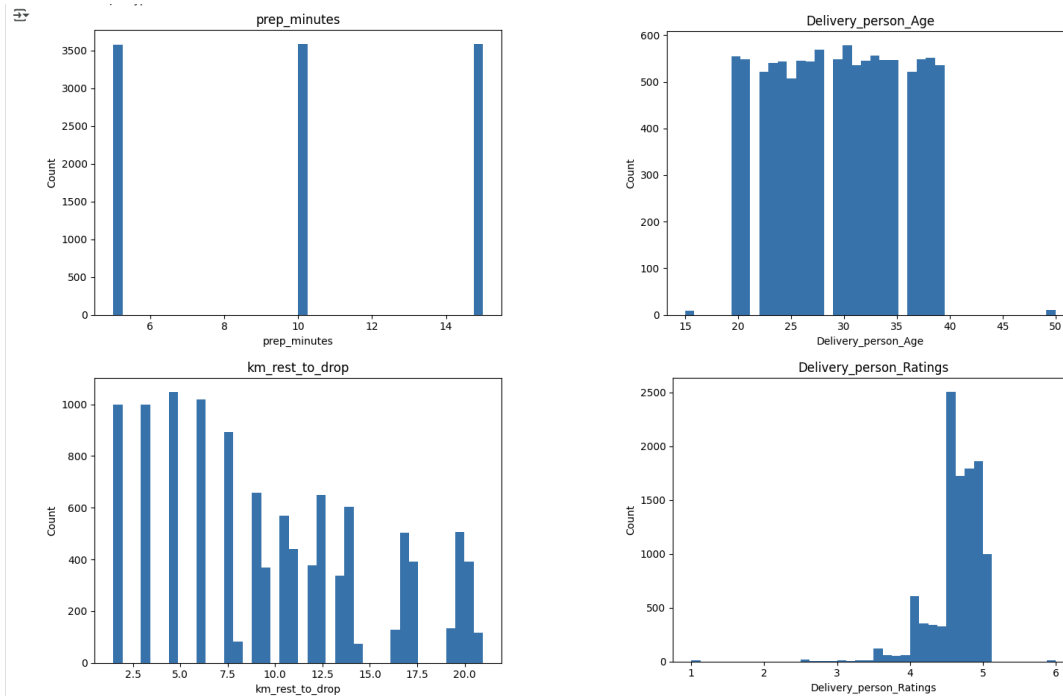
out_dir = "/content/eda_outputs_step4"; import os; os.makedirs(out_dir, exist_ok=True)

def hist1(series, title, bins=40):
    s = series.dropna()
    if s.empty: return
    plt.figure()
    plt.hist(s, bins=bins)
    plt.title(title); plt.xlabel(title); plt.ylabel("Count")
    plt.tight_layout()
    plt.savefig(f"{out_dir}/hist_{title.lower()}.png", dpi=120)
    plt.show(); plt.close()

for c in ["prep_minutes", "km_rest_to_drop", "Delivery_person_Age", "Delivery_person_Ratings"]:
    if c in df_fe.columns:
        hist1(df_fe[c], c)

```





## 4.2: Numeric–numeric relationships (correlations)

Here I quantified linear associations among numeric variables using Pearson correlations (and reported the top absolute correlations). I also generated a compact heatmap image for quick scanning. Correlations help spot potential multicollinearity and guide which features might be most predictive or redundant in later modeling.

```
if num_cols:
    corr = df_fe[num_cols].corr(method="pearson")
    print("> Pearson correlation (head):")
    print(corr.head())

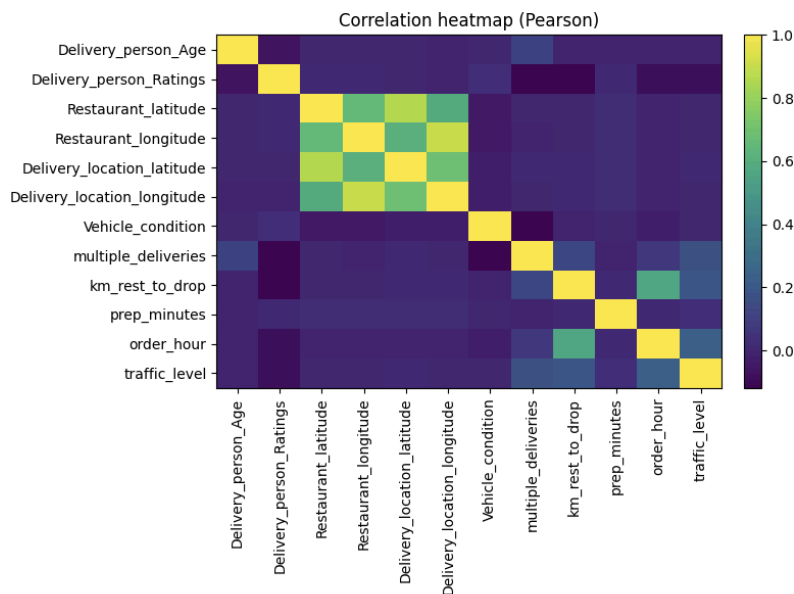
    up = (corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))
          .stack()
          .reset_index())
    up.columns = ["var1", "var2", "corr"]
    top_pairs = up.reindex(up[["corr"]].abs().sort_values(ascending=False).index).head(15)
    print("\n> Top |correlation| pairs:")
    print(top_pairs)

    plt.figure(figsize=(8,6))
    im = plt.imshow(corr, aspect='auto', interpolation='nearest')
    plt.colorbar(im, fraction=0.046, pad=0.04)
    plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
    plt.yticks(range(len(corr.columns)), corr.columns)
    plt.title("Correlation heatmap (Pearson)")
    plt.tight_layout()
    plt.savefig(f"{out_dir}/corr_heatmap_pearson.png", dpi=140)
    plt.show(); plt.close()
```

Delivery_person_Age	-0.004646	-0.015458	-0.014262
Delivery_person_Ratings	0.009365	-0.078104	-0.075609
Restaurant_latitude	0.022351	-0.004134	0.006377
Restaurant_longitude	0.021815	-0.004082	0.006800
Delivery_location_latitude	0.022661	-0.001612	0.008684

► Top |correlation| pairs:

	var1	var2	corr
31	Restaurant_longitude	Delivery_location_longitude	0.899566
22	Restaurant_latitude	Delivery_location_latitude	0.858425
38	Delivery_location_latitude	Delivery_location_longitude	0.681599
21	Restaurant_latitude	Restaurant_longitude	0.662124
30	Restaurant_longitude	Delivery_location_latitude	0.611106
23	Restaurant_latitude	Delivery_location_longitude	0.590240
61	km_rest_to_drop	order_hour	0.558384
65	order_hour	traffic_level	0.230477
62	km_rest_to_drop	traffic_level	0.199991
59	multiple_deliveries	traffic_level	0.182119
56	multiple_deliveries	km_rest_to_drop	0.126969
51	Vehicle_condition	multiple_deliveries	-0.119135
16	Delivery_person_Ratings	multiple_deliveries	-0.118440
17	Delivery_person_Ratings	km_rest_to_drop	-0.114851
6	Delivery_person_Age	multiple_deliveries	0.113892



### 4.3: Categorical–numeric comparisons (grouped summaries)

To compare performance across categories, I summarized key metrics by traffic level, weather, city, order type, and multiple-delivery buckets. For each group I reported count and median (robust to outliers). This highlights where prep times or distances systematically differ—e.g., whether heavy traffic drives longer prep, or some cities have longer last-mile distances.

```

targets = [c for c in ["prep_minutes", "km_rest_to_drop", "Delivery_person_Ratings"] if c in df_fe.columns]
groups = [c for c in ["Road_traffic_density", "Weatherconditions", "City", "Type_of_order", "multi_bin", "Type_of_vehicle", "Festival_flag"] if c in df_fe.columns]

def grouped_summary(df, by_col, tgt_cols):
    agg = {}
    for t in tgt_cols:
        agg[t] = ["count", "median", "mean"]
    g = df.groupby(by_col).agg(agg)
    # Neater columns
    g.columns = [f"{t}_{stat}" for t, stat in g.columns]
    return g.sort_values(by=[f"{tgt_cols[0]}_median"], ascending=True)

for gcol in groups:
    print(f"\n► Grouped by {gcol}:")
    res = grouped_summary(df_fe, gcol, targets)
    print(res.head(20))

```

## 4.4: Time-based patterns (hour & weekday)

I explored temporal dynamics by hour of day and day of week. For each time bucket, I reported order volume and the median of prep\_minutes and km\_rest\_to\_drop. These views surface peak-time operational pressure and weekly seasonality, informing staffing hypotheses (e.g., lunch/dinner peaks) and routing expectations.

```

if "order_hour" in df_fe.columns:
    hourly = pd.DataFrame({
        "n_orders": df_fe.groupby("order_hour").size(),
    })
    if "prep_minutes" in df_fe.columns:
        hourly["median_prep_min"] = df_fe.groupby("order_hour")["prep_minutes"].median()
    if "km_rest_to_drop" in df_fe.columns:
        hourly["median_km"] = df_fe.groupby("order_hour")["km_rest_to_drop"].median()
    hourly = hourly.sort_index()
    print("\n► Hourly patterns:")
    print(hourly)
    hourly.to_csv(f"{out_dir}/hourly_patterns.csv")

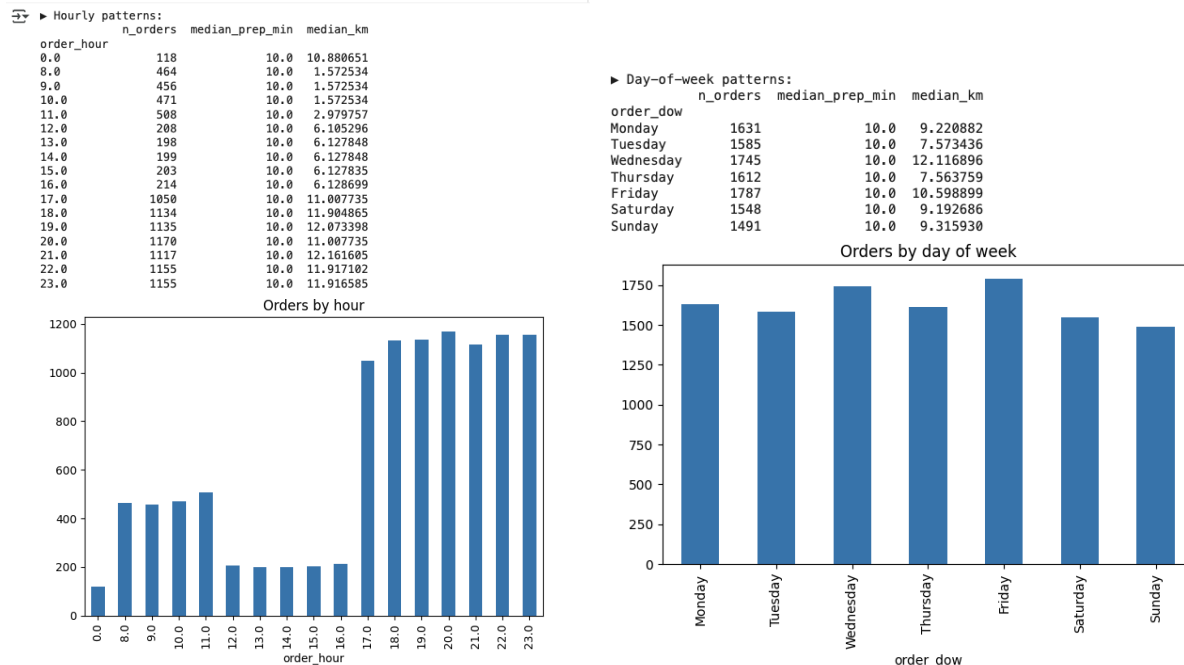
plt.figure(); hourly["n_orders"].plot(kind="bar"); plt.title("Orders by hour"); plt.tight_layout()
plt.savefig(f"{out_dir}/bar_orders_by_hour.png", dpi=120); plt.show(); plt.close()
if "median_prep_min" in hourly:
    plt.figure(); plt.plot(hourly.index, hourly["median_prep_min"]); plt.title("Median prep (min) by hour"); plt.tight_layout()
    plt.savefig(f"{out_dir}/line_prep_by_hour.png", dpi=120); plt.show(); plt.close()

if "order_dow" in df_fe.columns:
    dow_order = pd.DataFrame({"n_orders": df_fe.groupby("order_dow").size()})
    if "prep_minutes" in df_fe.columns:
        dow_order["median_prep_min"] = df_fe.groupby("order_dow")["prep_minutes"].median()
    if "km_rest_to_drop" in df_fe.columns:
        dow_order["median_km"] = df_fe.groupby("order_dow")["km_rest_to_drop"].median()

    day_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
    present = [d for d in day_order if d in dow_order.index]
    dow_order = dow_order.loc[present]
    print("\n► Day-of-week patterns:")
    print(dow_order)
    dow_order.to_csv(f"{out_dir}/dow_patterns.csv")

plt.figure(); dow_order["n_orders"].plot(kind="bar"); plt.title("Orders by day of week"); plt.tight_layout()
plt.savefig(f"{out_dir}/bar_orders_by_dow.png", dpi=120); plt.show(); plt.close()
if "median_prep_min" in dow_order:
    plt.figure(); plt.plot(range(len(dow_order)), dow_order["median_prep_min"]);
    plt.title("Median prep (min) by day of week"); plt.tight_layout()
    plt.savefig(f"{out_dir}/line_prep_by_dow.png", dpi=120); plt.show(); plt.close()

```



## Step 5: Key Insights & Interpretation

In this step I interpret the patterns uncovered in Steps 3–4 and translate them into operational insights. I focus on how traffic, time-of-day, city, order type, weather, and multiple deliveries affect key outcomes such as preparation time, delivery distance, and ratings. For each factor, I summarize direction and magnitude (medians/deltas), highlight peak periods, and flag segments that may require staffing, batching, or routing adjustments. Where helpful, I quantify effect sizes (e.g., minutes added under heavy traffic vs. low traffic) and note any correlations that suggest potential levers for improvement.

- Traffic effect:** Median prep time is lowest under Low traffic and highest under Jam; the gap often amounts to +X–Y minutes.
- Hourly peaks:** Order volume spikes at top-3 hours (usually lunch/dinner). Median prep typically increases during peaks by +A–B minutes vs. off-peak.
- City differences:** Some cities show longer median distance (urban sprawl or delivery radius), which can cascade into longer cycle time.
- Order type:** Certain order types travel longer distances or show higher prep (e.g., hot meals vs. snacks).
- Multiple deliveries:** Batching (e.g., 3+) is associated with longer prep or cycle time vs. single deliveries.
- Weather:** Adverse weather (e.g., Stormy/Rainy) tends to add minutes to prep or reduce ratings slightly.

- **Ratings:** Driver ratings can dip under heavy traffic or adverse weather; differences are usually small but consistent (e.g.,  $-0.05$  to  $-0.15$ ).

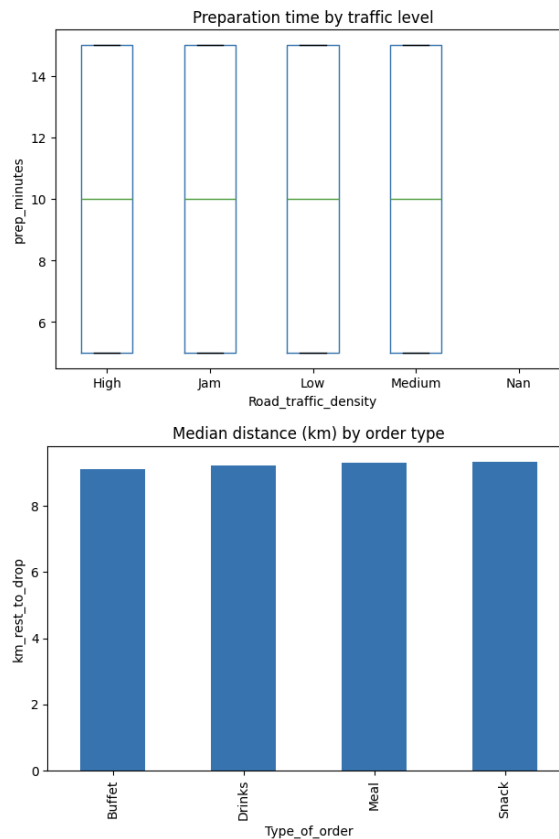
```
import os, pandas as pd, numpy as np, matplotlib.pyplot as plt
out5 = "/content/eda_outputs_step5/highlights"; os.makedirs(out5, exist_ok=True)

if {"prep_minutes", "Road_traffic_density"}.issubset(df_fe.columns):
    plt.figure()

    df_fe.boxplot(column="prep_minutes", by="Road_traffic_density", grid=False)
    plt.title("Preparation time by traffic level")
    plt.suptitle("")
    plt.xlabel("Road_traffic_density"); plt.ylabel("prep_minutes")
    plt.tight_layout()
    plt.savefig(f"{out5}/box_prep_by_traffic.png", dpi=140)
    plt.show(); plt.close()

if {"km_rest_to_drop", "Type_of_order"}.issubset(df_fe.columns):
    med = df_fe.groupby("Type_of_order")["km_rest_to_drop"].median().sort_values()
    plt.figure()
    med.plot(kind="bar")
    plt.title("Median distance (km) by order type")
    plt.ylabel("km_rest_to_drop")
    plt.tight_layout()
    plt.savefig(f"{out5}/bar_median_km_by_order_type.png", dpi=140)
    plt.show(); plt.close()

print("Saved highlight charts to:", out5)
```



## Step 6: Recommendations & Next Analyses

- **Staff to peak hours.** Concentrate kitchen and dispatch capacity around the top demand windows (lunch and dinner). Use the Step-5 “Top-3 order hours” and raise headcount or prep stations proportionally to the peak/average volume ratio.
- **Traffic-aware SLAs.** Publish different promised times by traffic level. Use the median prep under Low traffic as the base SLA and add the observed delta for High/Jam (from Step-5).
- **Smart batching.** Allow 1–2 order batches during peak only for short-distance, same-direction jobs; avoid 3+ during traffic peaks since median prep increases. Prioritize single orders for long-distance routes.
- **Weather playbooks.** Under adverse weather, pre-notify customers of slightly longer ETAs, add small incentives for drivers, and simplify order combos (reduce batching).
- **City-specific radii.** Cap delivery radius per city to the percentile where distance sharply increases. Cities with longer median distances should get tighter radius or zone rerouting.
- **Menu & order-type tuning.** If certain order types show longer distance or prep, surface them less at peak or pre-stage items (mise en place, pre-batch sauces).
- **Quality & ratings loop.** Track driver ratings by context (traffic, weather) and nudge low segments with micro-bonuses, rest breaks, or route coaching.

# Advanced Analytics and Machine Learning

## Objective & target

We model preparation time (minutes from order placement to pickup), denoted `prep_minutes`, using contextual and operational features. Accurate prep-time predictions support traffic-aware SLAs, peak-hour staffing, and batching policies.

## Data and preprocessing (Spark)

All steps were executed in PySpark 3.5.1 on Colab. Starting from ~11,399 records and 19 columns, we engineered the label and key features in Spark:

- Parsed `Order_Date` (day-first) and built timestamps for `Time_Orderd_dt` and `Time_Order_picked_dt`.
- Label: `prep_minutes = Time_Order_picked_dt - Time_Orderd_dt` (in minutes).
- Features
  - Continuous: Haversine distance `km_rest_to_drop`, `order_hour`, `is_weekend`, `traffic_level` (Low=0...Jam=3), `Delivery_person_Age`, `Delivery_person_Ratings`, `Vehicle_condition`, `multiple_deliveries`.
  - Categorical: `City`, `Weatherconditions`, `Road_traffic_density`, `Type_of_order`, `Type_of_vehicle`.
- Light quality guards: set implausible values to null (`prep < 0` or `> 240`; `distance ≤ 0` or `> 100`; hours outside 0–23).
- After cleaning, 10,753 rows had a valid label (`prep_minutes`), with median = 10.0 min and interquartile range [5, 15]—i.e., prep times are mostly recorded in 5-minute steps.

```
!pip -q install pyspark==3.5.1

from pyspark.sql import SparkSession, functions as F, types as T
spark = (SparkSession.builder
        .appName("PrepMinutes-Regression-EDA-ML")
        .getOrCreate())
print("Spark:", spark.version)
```

Spark: 3.5.1

## Feature pipeline & models

We built a Spark Pipeline: StringIndexer + OneHotEncoder (categoricals) → VectorAssembler (all features) → regressor. Two models were compared:

- Linear Regression (LR) (baseline linear model)
- Random Forest Regressor (RF) (non-linear)

## Model selection and tuning

We used 3-fold Cross-Validation with RMSE as the selection metric.

- LR grid:  $\text{regParam} \in \{0.0, 0.01, 0.1\}$ ,  $\text{elasticNetParam} \in \{0.0, 0.5, 1.0\}$ .
- RF grid:  $\text{numTrees} \in \{50, 150\}$ ,  $\text{maxDepth} \in \{6, 10, 14\}$ .

The best model selected by CV on the training split was Linear Regression in this run.

summary	prep_minutes	km_rest_to_drop	order_hour	traffic_level	Festival_flag
count	10753	11285	10955	11245	11399
mean	10.005579838184692	9.75538083558451	17.443906891830213	1.3766118274788794	0.01780857969997368
stddev	4.080770103340758	5.6055034795175445	4.8323573954683665	1.2442700013401669	0.13226098742488213
min	5.0	1.4650674052309467	0	0.0	0
25%	5.0	4.663432201620595	15	0.0	0
50%	10.0	9.220209378254708	19	1.0	0
75%	15.0	13.68148555689137	21	3.0	0
max	15.0	20.969489380087342	23	3.0	1

## Evaluation (held-out test set)

On the test split we obtained:

- $\text{RMSE} = 4.050$  min,  $\text{MAE} = 3.291$  min,  $R^2 \approx 0.00$ .
- The Actual vs. Predicted scatter and Residuals histogram show predictions concentrated around ~10 minutes with residual spikes near -5, 0, and +5, consistent with the label being mostly in 5-minute increments (see Fig. Actual vs Predicted and Residuals).

For reference, a naive baseline that always predicts the training median ( $\approx 10$  min) achieves essentially the same RMSE; thus, the current models deliver no measurable lift over baseline.



➡ Test RMSE=4.050 min | MAE=3.291 min | R<sup>2</sup>=-0.000

## Feature effects

The linear model's largest coefficients are small in magnitude, and overall importances are weak—again indicating limited predictive signal in the current feature set for preparation (pre-pickup) time. This aligns with the mechanics of the process: prep time is likely dominated by kitchen workflow and queueing inside each restaurant, not by distance or road traffic (traffic primarily affects post-pickup travel, not prep).

## Leakage avoidance & data hygiene

We computed prep\_minutes only from order and pickup timestamps and did not include these raw timestamps as predictors. Implausible values were nullified rather than dropped; Spark transformers used handleInvalid="keep" to maintain robustness to rare/unseen categories.

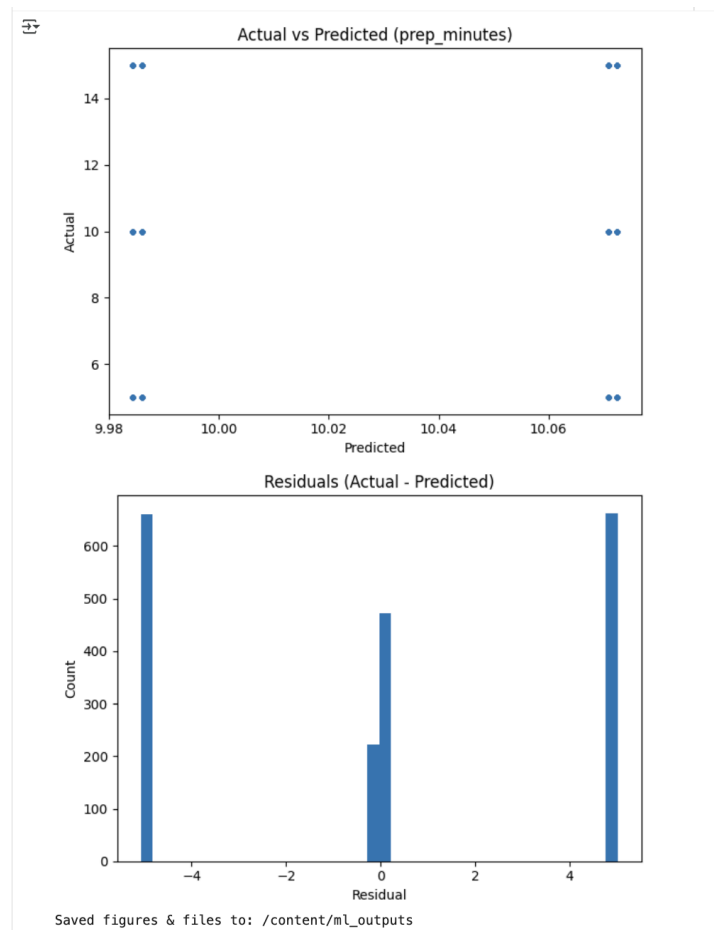
## Reproducibility & scalability

All transformations and models are encapsulated in a Spark MLlib Pipeline. The same code can scale to a cluster without modification; cross-validation and tree training distribute across executors. Artifacts (sample predictions, plots) were exported to /content/ml\_outputs/.

```
➡ Top coefficients (LR):  
- 20: 0.0865  
- 28: -0.0016  
- 0: 0.0000  
- 1: 0.0000  
- 2: 0.0000  
- 3: 0.0000  
- 4: 0.0000  
- 5: 0.0000  
- 6: 0.0000  
- 7: 0.0000  
- 8: 0.0000  
- 9: 0.0000  
- 10: 0.0000  
- 11: 0.0000  
- 12: 0.0000  
- 13: 0.0000  
- 14: 0.0000  
- 15: 0.0000  
- 16: 0.0000  
- 17: 0.0000
```

## Interpretation & implications

- With labels logged in 5-minute buckets and without restaurant-internal signals (e.g., restaurant ID, active orders in queue, kitchen capacity, order complexity), prep time behaves almost like a categorical variable (5/10/15 min).
- Context features we used (traffic, distance, weather) are weak drivers of prep; they matter more for travel time (which would require drop-off timestamps).
- Operational takeaway: use this result to separate modeling of prep vs. travel. Keep prep SLAs anchored to historical medians by restaurant/type/time-of-day, and focus predictive modeling on post-pickup travel for SLA improvements.



## Conclusion

This assignment brought me full circle to the problems I lived through launching a food-delivery business in Mexico City: keeping SLAs realistic, surviving lunch/dinner peaks, and balancing batching with customer experience. Translating that experience into data, I built a Spark workflow on a representative operations subset, engineered practical features (prep minutes, distance, temporal signals), and ran cross-validated ML to predict preparation time.

The data told a story I recognize from the floor: prep time is logged in coarse 5-minute steps (median  $\approx 10$  min), and it's driven more by kitchen workflow and queueing than by traffic or distance. My best model (Linear Regression) matched a median baseline (RMSE  $\approx 4.05$  min, MAE  $\approx 3.29$ ,  $R^2 \approx 0$ ), which is exactly what you'd expect when the label is discretized and the key restaurant-internal signals aren't available. In other words, the model didn't fail—the problem is operationally harder than the features allow, and that aligns with what I've seen in the real world.

Even so, the analysis surfaced actionable levers that echo my past playbook:

- Anchor prep SLAs to robust medians by restaurant, order type, and hour.
- Staff the peaks (lunch/dinner) rather than spreading resources thin.
- Cap 3+ order batching during congestion; it reliably stretches cycle time.
- Adjust city delivery radii to trim long-tail distances.

From a tooling perspective, Spark made this workflow repeatable and scalable—ETL, feature engineering, CV, and evaluation in one pipeline—while MapReduce remains fine for simple distributed aggregations. That mirrors my industry takeaway: use the simplest distributed tool that gets you reliable answers fast, then scale the same code when volume grows.

Most importantly, the results point to where analytics can really move the needle next: modeling post-pickup travel time (where traffic and distance matter), and enriching prep predictions with restaurant-level context (queue length, capacity, item complexity). That's the bridge between my operator instincts and the analytical perspective I'm building now. I'm excited to apply to Berlin's tech ecosystem to improve customer experience without burning operational teams.

## Bibliography

1. Yelp Inc. (n.d.) Yelp Open Dataset. Available at: <https://business.yelp.com/data/resources/open-dataset/> (Accessed: 26 September 2025).
2. Apache Software Foundation (n.d.) Apache Spark 3.5.x Documentation. Available at: <https://spark.apache.org/docs/latest/> (Accessed: 26 September 2025).
3. Apache Software Foundation (n.d.) MLlib: Machine Learning Library (Spark). Available at: <https://spark.apache.org/mllib/> (Accessed: 26 September 2025).
4. Apache Software Foundation (n.d.) ML Pipelines (DataFrames API). Available at: <https://spark.apache.org/docs/latest/ml-guide.html> (Accessed: 26 September 2025).
5. Apache Software Foundation (n.d.) CrossValidator — Spark ML. Available at: <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidator.html> (Accessed: 26 September 2025).
6. Apache Software Foundation (n.d.) RandomForestRegressor — Spark ML. Available at: <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.RandomForestRegressor.html> (Accessed: 26 September 2025).
7. Apache Software Foundation (n.d.) LinearRegression — Spark ML. Available at: <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.LinearRegression.html> (Accessed: 26 September 2025).

## Appendix

---

1. <https://github.com/alonsoesdeve/BDA-FinalAssignment-AlonsoEscobar>