Data Wrangling: From Raw to Ready

Dr. Sucheta Ghosh

March 27, 2025

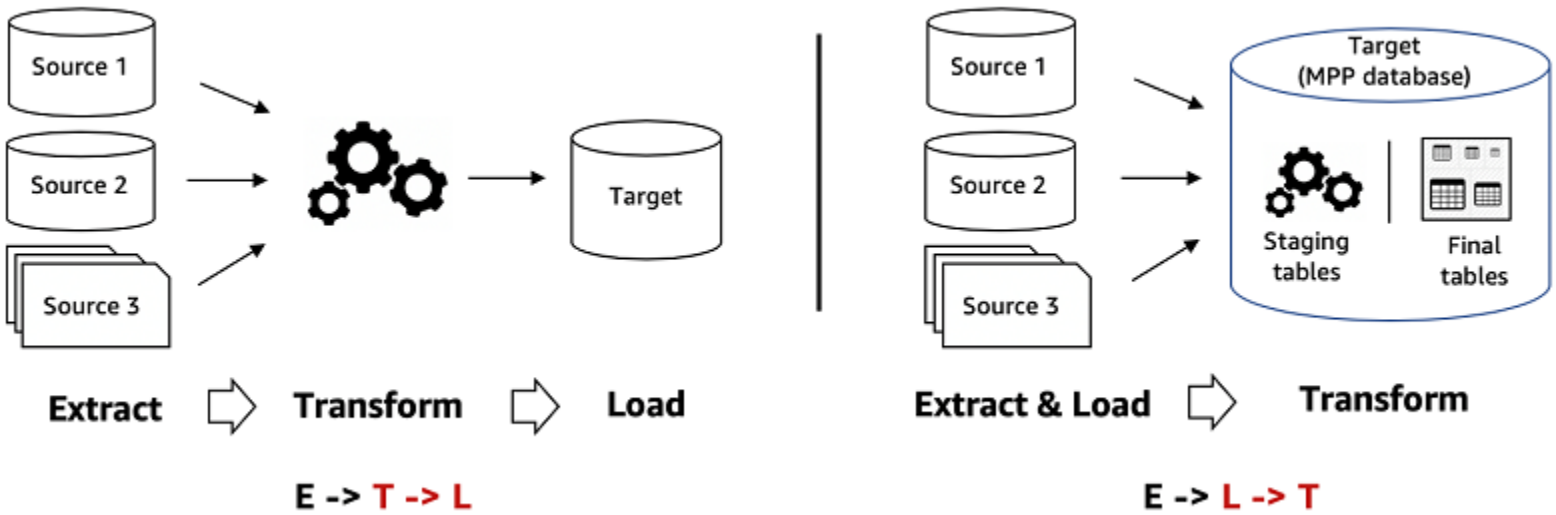
Agenda

- What is Data Wrangling?
- Profiling Raw Data
- Cleaning Data
- Transforming and Reshaping
- Flattening Nested Data
- Data Wrangling for Big Data
- Discussions + Exercises
- Scaling Wrangling (with Dask)

HOCH SCHULE OFFEN BURG

Revision+

- Data can be numeric(continuous or discrete), categorical (grouped into categories), or ordinal (ordered categories).
- Data can be either structured (organized in a defined format like tables) or unstructured (like free text, images, or videos), and semi-structured.
- ETL vs. ELT



Data Wrangling Matters...

- Real-world data is messy, inconsistent, and incomplete.
- Clean data is crucial for reliable analysis and machine learning.
- Wrangling is a major step in ETL pipelines.

HOCH SCHULE OFFEN BURG

Data Profiling

- Check structure: `df.info()`, `df.head()`
- Summarize data: `df.describe()`, `value_counts()`
- Identify missing values, outliers, and wrong types

Formula: Mean

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Formula: Variance

$$\mathrm{Var}(x) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

HOCH SCHULE OFFEN BURG

Handling Missing Values

Types of Missing Data:

- MCAR — Missing Completely At Random
- MAR — Missing At Random
- MNAR — Missing Not At Random

Formula: Mean Imputation

$$x_i = \begin{cases} x_i, & \text{if not missing} \\ \bar{x} = \frac{1}{n_{\text{obs}}} \sum_{j=1}^{n_{\text{obs}}} x_j, & \text{if missing} \end{cases}$$

where n_{obs} is the number of observed (non-missing) values.

Advanced Missing Value Imputation

Beyond Mean Imputation:

- `Forward Fill / Backward Fill`
- `Interpolation (linear, polynomial)`
- Model-based Imputation (e.g., KNN, regression)

Example: Linear Interpolation

$$x_{\text{missing}} = x_i + \frac{x_{i+1} - x_i}{t_{i+1} - t_i} \cdot (t_{\text{missing}} - t_i)$$

Tip: Evaluate missing patterns before selecting strategy.

HOCH SCHULE OFFEN BURG

Interpolation: Polynomial Method

Purpose: Estimate missing values using a fitted polynomial curve.
Idea: Fit a polynomial of degree d through known data points and use it to estimate the unknown.
Polynomial Interpolation Formula:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$$

In Pandas: `df["col"].interpolate(method="polynomial", order=2)`
Caution:

- Higher degrees may overfit or oscillate
- Works best with time series or ordered data

HOCH SCHULE OFFEN BURG

Model-Based Imputation: KNN and Regression

K-Nearest Neighbors (KNN) Imputation:

- Estimate missing values based on similarity to k nearest samples.
- Distance metric (e.g., Euclidean) used to find neighbors.

Formula (for mean-based imputation):

$$x_{\text{missing}} = \frac{1}{k} \sum_{j=1}^{k} x_{\text{known}}^{(j)}$$

Regression Imputation:

- Predict missing values using regression on other available features.
- Fit model: $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$
- Use to estimate missing y

Tip: Can be used iteratively (e.g., MICE — Multiple Imputation by Chained Equations)

HOCH SCHULE OFFEN BURG

Detecting Outliers

Common Methods:

- Z-score method
- IQR method (Interquartile Range)

Formula: IQR Rule

$$\text{Lower Bound} = Q_1 - 1.5 \times IQR, \quad \text{Upper Bound} = Q_3 + 1.5 \times IQR$$

where $IQR = Q_3 - Q_1$

Outlier Detection with Z-Score

Z-Score Method: Measures how many standard deviations a data point is from the mean.

Formula:

$$z_i = \frac{x_i - \bar{x}}{\sigma}$$

Flag as outlier if $|z_i| > k$, typically $k = 2$ or 3

Pros:

- Simple to implement
- Effective for normally distributed data

Limitation: Not suitable for skewed or non-Gaussian distributions.

Handling Wrong Data Types

Examples:

- Dates stored as strings
- Numeric values stored as categorical (e.g., ZIP codes)

Fix Example: Convert string to date

$$\text{ParsedDate} = \texttt{datetime.strptime}(\text{DateStr}, \text{"\%Y-\%m-\%d"})$$

Fix Example: Convert string to number

$$x = \texttt{float}(\text{str\_val})$$

HOCH SCHULE OFFEN BURG

Detecting & Fixing Wrong Data Types

Detection Methods:

- Schema validation (data type mismatches)
- Inconsistent parsing (e.g., string numbers: "123", "one-two-three")
- Regex rules for parsing formats

Correction Example:

- Convert strings to datetime:

```
pd.to_datetime(df["date_str"], format="%Y-%m-%d")
```

- Convert numeric strings:

```
df["col"] = pd.to_numeric(df["col"], errors="coerce")
```

HOCH SCHULE OFFEN BURG

Sample Profiling Code

```
df.info()
df.describe()
df['price'].value_counts(dropna=False)
df.isna().sum()
```

Sample Profiling Code

```
import pandas as pd
import numpy as np

# Sample data with missing values
df = pd.DataFrame({
    'x': np.arange(10),
    'y': [1, 4, np.nan, 16, 25, np.nan, 49, 64, 81,
        100]
})

# Interpolate using polynomial of degree 2
df['y_interp'] = df['y'].interpolate(method='
    polynomial', order=2)
print(df)
```

Sample Profiling Code

```
import pandas as pd
import numpy as np
from sklearn.impute import KNNImputer

# Sample data
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [9, 10, 11, 12]
})

imputer = KNNImputer(n_neighbors=2)
imputed = imputer.fit_transform(df)
df_imputed = pd.DataFrame(imputed, columns=df.columns)
print(df_imputed)
```

HOCH SCHULE OFFEN BURG

Sample Profiling Code

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample data
df = pd.DataFrame({
    'feature': [1, 2, 3, 4, 5],
    'target': [2, 4, np.nan, 8, 10]
})
# Separate rows with and without missing values
train = df[df['target'].notnull()]
missing = df[df['target'].isnull()]
# Train regression model
model = LinearRegression()
model.fit(train[['feature']], train['target'])
# Predict and fill missing values
df.loc[df['target'].isnull(), 'target'] = model.
    predict(missing[['feature']])
print(df)
```

HOCH SCHULE OFFEN BURG

Data Cleaning Techniques

- Handle missing values: `fillna()`, `dropna()`
- Convert types: `astype()`, `pd.to_datetime()`
- Remove duplicates: `drop_duplicates()`
- Standardize text: lowercase, strip, replace

Missing Value Ratio:

$$\text{Missing\%} = \left(\frac{\text{Missing Entries}}{\text{Total Entries}} \right) \times 100$$

Sample Cleaning Code

```
df['price'] = df['price'].fillna(df['price'].median())
df['date'] = pd.to_datetime(df['date'], errors='coerce
    ')
df = df.drop_duplicates()
df['product'] = df['product'].str.lower().str.strip()
```

Data Transformation

- Aggregation: `groupby()`, `agg()`
- Reshaping: `pivot()`, `melt()`
- Feature engineering

Group Mean Formula:

$$\mu_{group} = \frac{1}{|G_i|} \sum_{j \in G_i} x_j$$

HOCH SCHULE OFFEN BURG

Aggregation

Definition: Aggregation is the process of computing a summary statistic for groups of data.

Common Aggregation Functions:

- Mean, Median, Sum, Count
- Min/Max, Standard Deviation

Formula: Group Mean

$$\bar{x}_g = \frac{1}{n_g} \sum_{i=1}^{n_g} x_{i,g}$$

where $x_{i,g}$ is the i-th value in group g, and n_g is the number of items in group g.

Multi-Level Aggregation

Hierarchical Grouping: Useful when aggregating over multiple dimensions (e.g., region and product).

Formula:

$$\bar{x}_{g,h} = \frac{1}{n_{g,h}} \sum_{i=1}^{n_{g,h}} x_{i,g,h}$$

where g and h are grouping variables (e.g., region and category)

Example (Pandas): `df.groupby(["region", "category"]).mean()`

Reshaping Data

Definition: Reshaping refers to changing the layout or structure of data without changing the actual content.

Common Operations:

- `pivot` / `pivot_table`
- `melt` — unpivoting columns into rows
- `stack` / `unstack` for multi-index

Example: Wide to Long Format

```
melted_df = pd.melt(df, id_vars=["id"], var_name="variable", val
```

Pivot vs Melt

Pivot Table: Converts long to wide format **Melt:** Converts wide to long format

Pivot Formula:

$$y_{i,j} = f(x_{i,j}) \quad \text{(reshape using function } f \text{ e.g. mean)}$$

Example:

```
df.pivot(index="id", columns="year", values="sales")
df.melt(id_vars=["id"], var_name="year", value_name="sales")
```

Reshaping Example

```
sales = df.groupby('category')['revenue'].sum().
    reset_index()
long = df.melt(id_vars=['date'], value_vars=['sales_a'
    , 'sales_b'])
```

Feature Engineering

Definition: The process of creating new input features from existing data to improve model performance.

Common Techniques:

- Binning/Discretization
- Polynomial features
- Encoding categorical variables (e.g., One-Hot)
- Date/time feature extraction

Polynomial Feature Example:

$$x_{\text{poly}} = [x, x^2, x^3, \ldots, x^d]$$

One-Hot Encoding:

$$\text{color} = \text{"red"} \rightarrow [1, 0, 0] \text{ (red, green, blue)}$$

HOCH SCHULE OFFEN BURG

Datetime Feature Engineering

Useful Extracted Features:

- Year, Month, Day
- Day of Week
- Is Weekend, Is Holiday

Example (Pandas):

- `df["year"] = df["date"].dt.year`
- `df["weekday"] = df["date"].dt.weekday`

Cyclical Encoding (e.g., hour of day):

$$\text{hour\_sin} = \sin\left(\frac{2\pi \cdot \text{hour}}{24}\right), \quad \text{hour\_cos} = \cos\left(\frac{2\pi \cdot \text{hour}}{24}\right)$$

Flattening Nested Data

- JSON fields need normalization
- Use `pd.json_normalize()` or recursion

Flatten JSON Example

```
import json
with open("orders.json") as f:
    data = json.load(f)

df = pd.json_normalize(data, record_path='items', meta
    =['order_id', 'date'])
```

Scaling Data Wrangling

- Use Dask for larger-than-memory data
- Compatible with Pandas syntax
- Ideal for batch processing or cluster computing

Dask Example

```
import dask.dataframe as dd
df = dd.read_csv("bigdata/*.csv")
df.groupby("region")["sales"].sum().compute()
```

Discussion Prompt

What problems could arise if:

- You drop all rows with any missing values?
- You fill missing values with zero?

Merging DataFrames in Pandas

pandas.merge() allows combining two DataFrames on common columns or indices.

- Similar to SQL joins (inner, left, right, outer)
- Syntax: `pd.merge(df1, df2, on='key', how='inner')`

Join Types:

- `inner`: only matching rows
- `left`: all rows from left, matched from right
- `right`: all rows from right, matched from left
- `outer`: all rows from both, NaN for missing

Example:

- import pandas as pd
- df1 = pd.DataFrame('id': [1, 2], 'name': ['A', 'B'])
- df2 = pd.DataFrame('id': [1, 3], 'score': [90, 80])
- merged = pd.merge(df1, df2, on='id', how='inner')

Joining in Pandas

DataFrame.join() is used for combining columns of two DataFrames based on their index or key.

- Often simpler than `merge()` when joining on index
- Syntax: `df1.join(df2, how='left')`

Example:

- df1 = pd.DataFrame('name': ['A', 'B'], index=[1, 2])
- df2 = pd.DataFrame('score': [90, 80], index=[1, 3])
- joined = df1.join(df2, how='left')

Use `join()` for index-based joins, and `merge()` for column-based.

Few Practical Tips

- Always inspect before cleaning
- Save intermediary versions
- Use assertions and sanity checks
- Document all transformations

Data Wrangling for Big Data

Key Strategies for Big Data Wrangling:

- Use distributed computing tools like `Apache Spark` or `Dask`
- Process data incrementally in batches or streams
- Use efficient file formats like `Parquet` or `ORC`
- Apply out-of-core processing to handle larger-than-memory data
- Use parallel and distributed data wrangling operations

Common Tools:

- `Apache Spark`, `Dask`, `Hadoop`
- `Google BigQuery`, `Amazon Redshift`
- `Vaex` for out-of-core processing

Tip: Use distributed systems and optimized formats for scalability and performance.

HOCH SCHULE OFFEN BURG

Efficient File Formats for Big Data:

- **Parquet:** A columnar storage format for Hadoop, optimized for read-heavy analytics.
  - Benefits:
    - Supports complex nested data structures (e.g., arrays, maps).
    - Highly efficient for queries involving specific columns.
    - Supports predicate pushdown, reducing the amount of data read.
  - Commonly used in `Apache Spark` and `Hive`.
- **ORC (Optimized Row Columnar):** Another columnar format primarily used within `Apache Hive`.
  - Benefits:
    - Higher compression than Parquet (more efficient storage).
    - Better suited for high-performance queries (read-heavy workloads).
    - Supports predicate pushdown and advanced compression techniques.
  - Commonly used in `Apache Hive` and `Apache HBase`.

Data Wrangling for Big Data

Efficient File Formats for Big Data:

- **Avro:** A row-based format ideal for writing to data lakes and log files.
  - Benefits:
    - Schema is stored alongside the data for easier integration.
    - Optimized for write-heavy operations and serializing data between services.
  - Commonly used in streaming data pipelines (e.g., `Apache Kafka`).
- **Delta Lake:** An open-source storage layer that provides ACID transactions on top of `Apache Spark` and `Parquet`.
  - Benefits:
    - Provides transactional consistency and support for time travel (historical data access).
    - Enables batch and streaming workloads to coexist.
  - Commonly used for data lakes in real-time analytics.

HOCH SCHULE OFFEN BURG

Summary

When to Use Each Format:

- `Parquet`: Best for analytics and read-heavy workloads on large datasets.
- `ORC`: Best for high-performance querying in `Apache Hive` or columnar storage systems.
- `Avro`: Ideal for row-based storage, especially in log files and data pipelines.
- `Delta Lake`: Best for transactional data lakes and combining batch + streaming workloads.

Tip: Use columnar formats like `Parquet` and `ORC` for efficient storage and retrieval in big data wrangling tasks.

Summary

- Data wrangling is essential for analysis and ML
- Learn to profile, clean, and reshape data
- Practice on messy real-world datasets

Note on ACID Properties

- Atomicity: Consider a bank transfer where $100 is moved from Account A to Account B. If the debit from Account A succeeds but the credit to Account B fails, atomicity ensures that the debit is undone, leaving both accounts unchanged.
- Consistency: In the bank transfer example, if the database has a constraint that an account balance cannot be negative, consistency ensures that no transaction will violate this rule.
- Isolation: If one transaction is transferring $100 from Account A to Account B and another transaction is reading the balance of Account A, isolation ensures that the second transaction will only see the updated balance after the transfer is fully complete.
- Durability: Once the bank transfer is successfully completed, the changes to the account balances are saved permanently, even if the system crashes immediately after.

HOCH SCHULE OFFEN BURG

Thank You!

Questions? sucheta.ghosh@lehrb.hs-offenburg.de