Pedagogical Report: Teaching GIGO Through a Data Quality Pipeline

Author**: Amantha Bhaskarabhatla**

NUID**: 002300618**

1. Teaching Philosophy

1.1 Target Audience

The primary target audience for this teaching module is **graduate-level data science and information systems students**, specifically peers in INFO 7390: Advanced Data Science and Architecture. I assume:

- Students are comfortable with **basic Python and pandas** (loading data, simple transforms, groupby).
- Students have seen **machine learning models** and dashboards before, but often treat data cleaning as a quick, informal pre-processing step.
- Students are familiar with high-level course themes like **GIGO (Garbage In, Garbage Out)** and **Computational Skepticism**, but may not yet have a concrete pattern for systematically operationalizing those ideas.

I assume no prior expertise in professional data-quality frameworks (like Great Expectations or Soda), formal data contracts, or governance tools. The notebook is designed so that someone with intermediate Python skills can both run and modify it in a single sitting.

1.2 Learning Objectives

By the end of this module, students should be able to:

1. **Explain GIGO in concrete, data-science terms**
    - Describe how bad data propagates into misleading metrics, dashboards, and models.
    - Connect GIGO to real-world scenarios (e.g., revenue dashboards, fraud detection, risk scoring).
2. **Identify common data-quality problems in a tabular dataset**
    - Missing values, invalid ranges, invalid categories, outliers, and duplicates.
    - Relate these issues to data quality dimensions like completeness, validity, and reasonableness.
3. **Define a simple data contract / validation configuration**
    - Express “what valid data looks like” using a Python dictionary (validation_rules / QUALITY_CONFIG).
    - Understand the difference between **rules** (declarative) and **pipeline logic** (procedural).

4. **Implement a reusable validation and cleaning pipeline in pandas**
    - Write functions (e.g., data_quality_report) to compute missing and invalid percentages per column.
    - Apply a cleaning pipeline that fixes or removes invalid records in a transparent way.
5. **Quantify the impact of GIGO on a business metric**
    - Compare a metric (average transaction amount per country) before and after cleaning.
    - Interpret how specific data issues distorted the original metric.
6. **Reflect and extend the pattern**
    - Tackle guided exercises that modify rules, add new fields, and extend the pipeline.
    - Practice computational skepticism by asking: "What hidden assumptions are baked into this data?"

1.3 Pedagogical Approach and Rationale

The overall teaching approach follows the **Explain → Show → Try** model specified in the assignment, with an emphasis on **learning by doing** and **computational skepticism**.

1. **Explain**
    - I begin with a short theoretical section that frames GIGO not as a cliché but as a **design principle** for data systems.
    - I connect GIGO to the course's broader themes of **Botspeak**, **GIGO**, and **Computational Skepticism**: students are encouraged to treat datasets like "black boxes" whose assumptions must be interrogated.
    - The explanation is grounded in tangible examples (e.g., dashboards that overestimate revenue due to duplicates, models that under- or overestimate risk because of invalid inputs).
2. **Show**
    - Instead of using a static slide example, the notebook **constructs a synthetic dataset** representing e-commerce-like transactions.
    - I deliberately **inject GIGO** into the dataset (missing values, impossible ages, invalid categories, outliers, duplicates).
    - I then introduce a **data contract** (validation rules) and a **validation function** (data_quality_report) that produces a concise quality report.
    - Finally, I run a **cleaning pipeline** and show how a simple metric (average transaction amount per country) changes before and after cleaning.
3. **Try**

- The notebook includes a structured set of exercises with increasing difficulty:
    - Interpreting the data quality reports and metric changes.
    - Modifying validation rules, thresholds, and cleaning strategies.
    - Adding a new field (e.g., loyalty_score) and extending the full pipeline.
- These exercises are designed to move students from **passive understanding** to **active experimentation**, reinforcing the idea that "data quality is a design choice, not a fixed property."

The rationale for this approach is to make GIGO **actionable**: not just a warning phrase, but a pattern students can reuse in their own projects.

2. Concept Deep Dive

2.1 Technical and Mathematical Foundations

The core concept is **GIGO operationalized through a data-quality pipeline**. Technically, the module covers:

1. **Data Quality Dimensions (informal but practical)**
    - **Completeness**: fraction of non-missing values per column. Mathematically, for column (X) with (n) rows and (m) missing values, completeness = (1 - \frac{m}{n}).
    - **Validity**: proportion of values falling within a defined **range** or **allowed set**:
        - Range check: (\text{valid}(x) = [a \le x \le b])
        - Categorical check: (\text{valid}(x) = [x \in S]) where (S) is an allowed set (e.g., country codes).
    - **Uniqueness**: detection of duplicate rows via df.duplicated().sum().
    - **Reasonableness**: values that are mathematically valid but contextually implausible (e.g., transaction_amount far beyond typical range). For this module, I approximate this via **thresholds** and **distribution-based cutoffs** (like the 99th percentile used in anomaly labelling).
2. **Data Contract / Validation Configuration**

- A **Python dictionary** encodes constraints:
- validation_rules = {
- "age": {"min": 18, "max": 100},
- "transaction_amount": {"min": 0, "max": 1000},
- "country": {"allowed": ["US", "UK", "IN", "DE", "CA"]},
- "product_category": {"allowed": ["Electronics", "Clothing", "Grocery", "Beauty"]},
- }
- This is a miniature version of a **data schema with constraints**: a mapping from column names to sets of rules.
- The validation function interprets these rules and computes per-column metrics, effectively implementing:

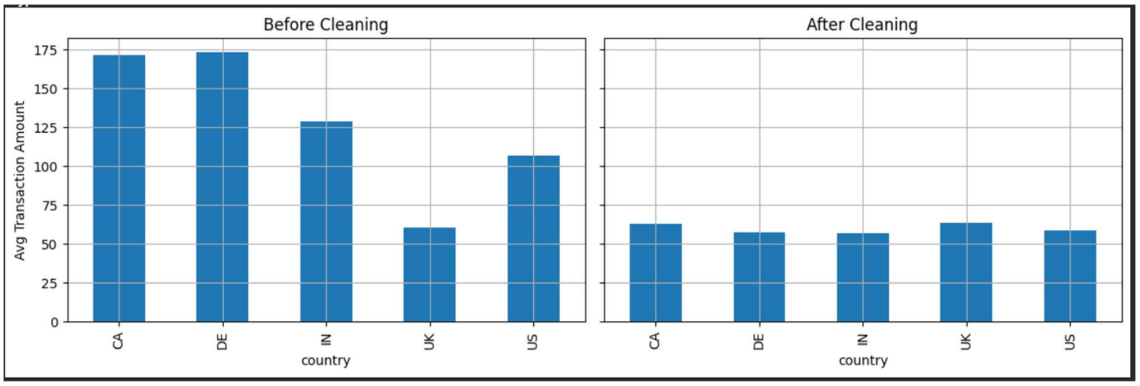
$$
\text{invalid\_pct}(X) = 100 \times \frac{\#\{x \in X : x \text{ violates rule}\}}{n}
$$

3. **Validation vs Cleaning**
   - Validation: a pure function from (df, rules) → report, with no side effects.
   - Cleaning: a transformation df → df_clean, where invalid values are set to NaN, imputed, clamped, or filtered out.
   - Separating these stages mirrors formal **ETL/ELT pipelines** and helps students understand that "detecting" and "fixing" are distinct operations.
4. **Metric Comparison Before vs After Cleaning**
   - For each country (c), we calculate:

$$
\text{avg\_before}(c) = \frac{1}{n_c}\sum_{i \in c} \text{amount}_i
$$

on the dirty data, and similarly ($\text{avg\_after}(c)$) on df_clean.

   - Students can visually compare these and reflect on how invalid/outlier values affected the mean.



2.2 Connection to Course Themes (GIGO, Botspeak, Computational Skepticism)

- **GIGO**: The notebook directly embodies the GIGO principle by constructing a scenario where small data-quality problems have **large impact** on metrics.
- **Botspeak Framework**: The idea of encoding rules in a configuration (validation_rules) aligns with Botspeak's emphasis on **explicit instruction patterns**: instead of "magic" pre-processing, we declare our assumptions in a structured format.
- **Computational Skepticism**:
    - The entire workflow encourages students to question the dataset:
        - "What counts as a valid age here?"
        - "What is a plausible maximum transaction amount?"
        - "What does it mean for a row to be 'trusted'?"
    - The reflection questions and exercises deliberately push students to **challenge their own rules** (What happens if the threshold is too strict? Too lenient?).

2.3 Relationship to Real-World Data Science Workflows

In practice, real data pipelines rarely start with perfect data. The notebook mimics key stages of a production workflow:

1. **Ingestion of Raw Data**
    - df is treated as an "incoming" dataset from some upstream source.
    - Students see that even a simple CSV or API response can contain subtle anomalies.
2. **Automated Data Quality Checks**
    - The data_quality_report function plays a similar role to checks implemented in tools like Great Expectations: quick, automated visibility into missingness and rule violations.
3. **Cleaning and Feature Preparation**

- df_clean reflects the kind of transformations that would happen before data is loaded into a warehouse, used in BI dashboards, or passed into a model-training pipeline.

4. **Metric Monitoring and Drift in Inputs**
    - By comparing average transaction amounts before and after cleaning, students see that changing input quality changes key performance indicators—mirroring real-world **data drift** and **metric drift** issues.
5. **Reusability and Configuration-Driven Design**
    - Centralizing rules in a dictionary mirrors how teams maintain **YAML configs** or **dbt models** to version and share data contracts.

3. Implementation Analysis

3.1 Architecture and Design Decisions

The educational artefact is structured as a **linear but modular notebook pipeline**:

1. **Synthetic Base Dataset (base_df)**
    - Create a clean-ish dataset with five key columns:
        - customer_id
        - age
        - country
        - product_category
        - transaction_amount

```python
np.random.seed(42)

n_rows = 500

customer_ids = np.random.randint(1000, 2000, size=n_rows)
ages = np.random.randint(18, 80, size=n_rows)
countries = np.random.choice(
    ["US", "UK", "IN", "DE", "CA"],
    size=n_rows,
    p=[0.3, 0.2, 0.2, 0.15, 0.15]
)
product_categories = np.random.choice(
    ["Electronics", "Clothing", "Grocery", "Beauty"],
    size=n_rows
)
transaction_amounts = np.round(np.random.exponential(scale=50, size=n_rows) + 10, 2)

base_df = pd.DataFrame({
    "customer_id": customer_ids,
    "age": ages,
    "country": countries,
    "product_category": product_categories,
    "transaction_amount": transaction_amounts
})

base_df.head()

```

2. **GIGO Injection (df)**
    - Copy base_df into df and deliberately corrupt it:
        - Missing ages and countries
        - Impossible ages (-5, 150)
        - Invalid categories (UnknownCategory)
        - Negative and excessively large transaction amounts
        - Duplicate rows

```python
# 1. Missing ages
missing_age_idx = np.random.choice(df.index, size=20, replace=False)
df.loc[missing_age_idx, "age"] = np.nan

# 2. Impossible ages
df.loc[np.random.choice(df.index, size=5, replace=False), "age"] = -5   # negative
df.loc[np.random.choice(df.index, size=5, replace=False), "age"] = 150  # too large

# 3. Missing countries
missing_country_idx = np.random.choice(df.index, size=15, replace=False)
df.loc[missing_country_idx, "country"] = np.nan

# 4. Invalid product categories
df.loc[np.random.choice(df.index, size=10, replace=False), "product_category"] = "UnknownCategory"

# 5. Negative transaction amounts
df.loc[np.random.choice(df.index, size=8, replace=False), "transaction_amount"] *= -1

# 6. Extreme outliers (multiply some by 20)
df.loc[np.random.choice(df.index, size=5, replace=False), "transaction_amount"] = \
    df["transaction_amount"].max() * 20

# 7. Duplicate some rows
duplicates = df.sample(10, random_state=1)
df = pd.concat([df, duplicates], ignore_index=True)

df.head(10)
```

3. **Initial Diagnostics**
    - Use df.info(), df.isna().sum(), describe(), and unique() to illustrate how GIGO appears in standard EDA outputs.
4. **Data Contract Definition (validation_rules)**
    - Define allowed ranges and categorical domains using a configuration dictionary.
5. **Validation Utility (data_quality_report)**
    - Implement a function that:
        - Computes missing_pct and invalid_pct per column.
        - Returns a small, readable report table.

```python
dq_before = data_quality_report(df, validation_rules)
print("Data quality report BEFORE cleaning:")
display(dq_before)

dup_count = df.duplicated().sum()
print(f"\nDuplicate rows: {dup_count} ({dup_count / len(df) * 100:.2f}%)")

```

```
ata quality report BEFORE cleaning:
```

	column	missing_pct	invalid_pct
0	age	3.921569	1.960784
1	country	2.941176	0.000000
2	product_category	0.000000	1.960784
3	transaction_amount	0.000000	2.549020

6. **Cleaning Pipeline (clean_df)**
    - Apply rule-based cleaning:
        - Drop duplicates.
        - Convert out-of-range ages to NaN and impute with median.
        - Convert invalid categories/countries to NaN and impute with modes.
        - Mark and fix invalid transaction amounts.

```python
# 1. Drop duplicates
clean_df = clean_df.drop_duplicates()

# 2. Fix age: out-of-range -> NaN, then fill with median
age_min, age_max = validation_rules["age"]["min"], validation_rules["age"]["max"]
invalid_age = clean_df["age"].notna() & (
    (clean_df["age"] < age_min) | (clean_df["age"] > age_max)
)
clean_df.loc[invalid_age, "age"] = np.nan
clean_df["age"] = clean_df["age"].fillna(clean_df["age"].median())

# 3. Fix country: invalid -> NaN, then fill with mode
valid_countries = validation_rules["country"]["allowed"]
clean_df.loc[~clean_df["country"].isin(valid_countries), "country"] = np.nan
clean_df["country"] = clean_df["country"].fillna(clean_df["country"].mode().iloc[0])

# 4. Fix product_category: invalid -> NaN, then fill with mode
valid_cats = validation_rules["product_category"]["allowed"]
clean_df.loc[~clean_df["product_category"].isin(valid_cats), "product_category"] = np.nan
clean_df["product_category"] = clean_df["product_category"].fillna(
    clean_df["product_category"].mode().iloc[0]
)

# 5. Fix transaction_amount:
#    - non-positive or > max -> NaN, then fill with median
ta_min, ta_max = validation_rules["transaction_amount"]["min"], validation_rules["transaction_amount"]["max"]
invalid_ta = clean_df["transaction_amount"].notna() & (
    (clean_df["transaction_amount"] <= ta_min) |
    (clean_df["transaction_amount"] > ta_max)
)
clean_df.loc[invalid_ta, "transaction_amount"] = np.nan
clean_df["transaction_amount"] = clean_df["transaction_amount"].fillna(
    clean_df["transaction_amount"].median()
)

dq_after = data_quality_report(clean_df, validation_rules)
print("Data quality report AFTER cleaning:")
display(dq_after)
```

7. **Re-validation on Clean Data**
    - Re-run data_quality_report on clean_df to show measurable improvement.
8. **Metric Comparison (avg by country)**
    - Compute and visualize average transaction amount per country before and after cleaning.
9. **Labeling Quality (Optional)**
    - Optionally, introduce a row_valid or quality_label field combining validity and anomaly detection (e.g., “INVALID”, “SUSPICIOUS”, “OK”).
10. **Exercises & Reflection**
    - End with progressive exercises and reflection questions.

This architecture is intentionally straightforward: it’s linear enough for students to follow end-to-end, but modular enough that each piece (simulation, validation, cleaning, metrics) could be reused independently.

3.2 Libraries / Tools Chosen and Why

- **Python + pandas**

  - Ubiquitous in data science.
  - Students are already familiar with it from prior coursework.
  - Provides vectorized operations and convenient handling of missing values.

- **NumPy**

  - Used for random data generation and numeric operations.

- **Matplotlib / pandas plotting**

  - Used to generate simple bar plots for before/after metric comparison.
  - Keeps visualization overhead minimal to focus on data quality concepts.

- **Synthetic Data Instead of Real Data**
  - Allows me to control specific types and frequencies of errors.
  - Makes it easier to align GIGO examples precisely with learning objectives.

I intentionally did not use specialized data-quality frameworks (e.g., Great Expectations) in this notebook, to avoid overwhelming students with another library and to keep the focus on foundational concepts. However, the patterns taught here align with what those tools formalize.

3.3 Performance Considerations

The current implementation operates on a relatively small dataset (~500–1000 rows), so performance is not a bottleneck. Still, the design anticipates scalability in several ways:

- **Vectorized Operations**
  - Range and allowed-value checks are implemented using pandas vectorized operations (between, .isin, boolean masks), which scale better than row-wise Python loops.
- **Config-Driven Rules**
  - Having a single validation_rules dictionary means that scaling to more columns or new datasets is a matter of adding config, not copying and pasting logic.
- **Report Size**

- The data-quality report is intentionally small and focused; in a larger system, this can be logged or pushed into a monitoring dashboard.

In a production environment with millions of rows, this same pattern could be:

- Pushed into a batch job running on a distributed framework (e.g., PySpark with similar concepts).
- Converted into SQL constraints or dbt tests.
- Integrated into a streaming quality check for incremental data ingestion.

3.4 Edge Cases and Limitations

Some key limitations and edge cases are intentionally left as future extensions (and are noted in the reflection / "future improvements" section of the notebook):

1. **Over-simplified Rules**
    - The min/max ranges and allowed sets are somewhat arbitrary and domain-specific. They are sufficient for teaching, but in real life, these thresholds would be negotiated with domain experts.
2. **Imputation Choices**
    - Using median and mode for imputation is simple but naive. Different imputation strategies (e.g., conditional imputation, model-based imputation) might be more appropriate depending on context.
3. **Single-Field Checks Only**
    - The current validation focuses on univariate constraints (per column). Many interesting data-quality issues are multivariate (e.g., age vs. product_category, time-based anomalies).
4. **No Temporal Dimension**
    - The dataset does not include timestamps; time-based validation (e.g., detecting old or out-of-order data) is not covered but is important in real workflows.
5. **Label Noise vs. Feature Noise**
    - The module focuses on feature-quality issues. Label noise (incorrect labels in supervised learning) isn't explicitly addressed, though the same GIGO logic applies.

4. Assessment & Effectiveness

4.1 How Student Understanding is Validated

Student understanding is assessed informally within the notebook and could be assessed formally in the course via:

1. **Reflection Questions**
    - Students are asked to interpret the data-quality reports and before/after metrics:

        - "Which column has the highest invalid percentage and why?"
        - "How did cleaning affect the average transaction amount for each country?"
    - These act as quick comprehension checks.
2. **Guided Exercises**
    - Exercises are designed with **increasing complexity**:
        - Modify the thresholds in validation_rules and observe the impact.
        - Add a new field (loyalty_score) and define new validation rules and cleaning operations.
    - Successfully completing these exercises demonstrates that students can **transfer the pattern** to a slightly different setting.
3. **Code-Based Evidence**
    - Students are encouraged to rerun the notebook cells and see:
        - No errors when rules are applied correctly.
        - Meaningful changes in reports and metrics when they update constraints.

If integrated into a formal course, additional assessment could include:

- Short written quizzes asking students to explain concepts like data contracts, completeness vs validity, and GIGO.
- A small assignment where students must apply the same pattern to a new dataset.

4.2 Common Challenges Students May Face

Based on the design and the kinds of errors already encountered while building the notebook, students are likely to face:

1. **Name Mismatches / KeyErrors**
    - Using validation rules for column names that don't exist (e.g., "quantity" vs "transaction_amount").
    - This reinforces the need to **check df.columns** and keep config and data in sync.
2. **Type Issues and Date Parsing**
    - If students add date fields or more complex types, pd.to_datetime and comparisons may produce unexpected NaT or NaN values.
3. **Overly Strict Rules**
    - Tightening rules too much may cause too many rows to be dropped or marked invalid, changing metrics in ways that don't align with business reality.
    - This becomes a teachable moment about domain context and trade-offs.

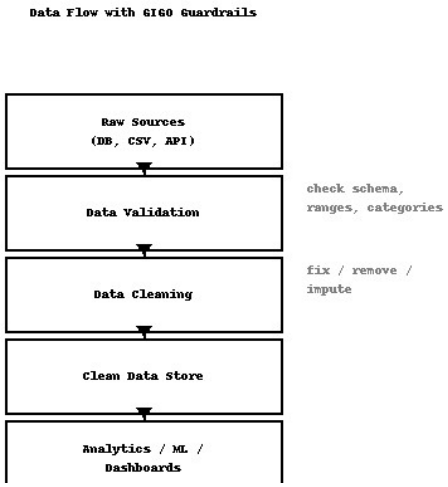
4. **Misinterpreting Metrics**
    - Students may initially misinterpret why averages change after cleaning (e.g., thinking that imputation reduces amounts when actually outliers were lowered).

The notebook anticipates some of these by adding comments and small explanations around the more subtle transformations, but encountering and resolving these issues is also part of the learning process.

4.3 Addressing Different Learning Styles

The teaching artefact deliberately mixes:

- **Textual explanations** (markdown cells with theory, rationale, and reflection prompts).
- **Visual elements**:
    - A GIGO data-flow diagram illustrating the pipeline stages.
    - Before/after bar charts for the business metric.



- **Hands-on code**:
    - Cells that students can run, modify, and extend.
- **Scaffolded exercises**:
    - Clearly labeled exercises with hints, ordered from easy to hard.

This combination supports:

- **Verbal learners** (through narrative and explanations).
- **Visual learners** (through diagrams and plots).
- **Kinesthetic / active learners** (through coding and experimentation).

4.4 Future Improvements and Extensions

There are several obvious directions to improve and extend this teaching module:

1. **Multi-Field and Temporal Checks**

- Add cross-field constraints (e.g., age vs product_category) and time-based patterns (e.g., seasonality in transaction_amount).
- This would deepen the realism and encourage students to think about more complex forms of GIGO.

2. **Integration with Professional Tools**
    - Show how the same concepts map to or complement tools like **Great Expectations** or dbt tests.
    - This would help students see a path from classroom patterns to production practices.
3. **Interactive UI Layer**
    - Wrap the notebook logic into a small Streamlit or web app that lets students:
        - Upload their own CSV,
        - Set validation rules via a UI,
        - And view quality reports and metric comparisons interactively.
4. **More Formal Assessment**
    - Add auto-graded tests (e.g., using pytest or nbgrader) to check that students correctly implemented new rules and transformations in the exercises.
5. **Expanded Metric Analysis**
    - Move beyond a single average to include distribution plots, quantiles, and fairness metrics in cases where GIGO impacts different groups unequally.

Summary

This pedagogical report describes a teaching module that turns **GIGO** from a slogan into a **concrete, reproducible pattern** for data-quality-aware data science. Through a synthetic dataset, deliberate corruption, a config-driven validation function, and a simple cleaning pipeline, students learn to:

- See how **garbage input** becomes **garbage output**.
- Articulate and encode **data contracts**.
- Think like **computational skeptics**, questioning the assumptions baked into datasets.