Faculty of Engineering and Applied Science

SOFE 3U Software Quality and Project Management

Assignment 5

Data Quality and Validation

Victor Oladeinde	100874455

Date: 25/03/25

GitHub Link: https://github.com/vicjustine/SQLAB5

Task 1

Screenshots:

Great Expectations Task

1. Install Great Expectations Library

```
[ ] # If not already installed, uncomment and run:
    !pip install great_expectations==0.15.50
```

```
atisfied: tqdm>=4.59.0 in /usr/local/lib/python3.11/dist-packages (from great_expectations==0.15.50) (4.67.1)
atisfied: typing-extensions>=3.10.0.0 in /usr/local/lib/python3.11/dist-packages (from great_expectations==0.15.50) (4.12.2)
atisfied: tzlocal>=1.2 in /usr/local/lib/python3.11/dist-packages (from great_expectations==0.15.50) (5.3.1)
27,>=1.25.4 (from great_expectations==0.15.50)
-1.26.20-py2.py3-none-any.whl.metadata (50 kB)
━━━━━━━━━━━━━━━━━━━━━━━━ 50.1/50.1 kB 2.7 MB/s eta 0:00:00
atisfied: numpy>=1.23.0 in /usr/local/lib/python3.11/dist-packages (from great_expectations==0.15.50) (1.26.4)
atisfied: pandas>=1.3.0 in /usr/local/lib/python3.11/dist-packages (from great_expectations==0.15.50) (2.1.4)
atisfied: entrypoints in /usr/local/lib/python3.11/dist-packages (from altair<4.2.1,>=4.0.0->great_expectations==0.15.50) (0.4)
atisfied: toolz in /usr/local/lib/python3.11/dist-packages (from altair<4.2.1,>=4.0.0->great_expectations==0.15.50) (0.12.1)
atisfied: cffi>=1.12 in /usr/local/lib/python3.11/dist-packages (from cryptography>=3.2->great_expectations==0.15.50) (1.17.1)
atisfied: zipp>=3.20 in /usr/local/lib/python3.11/dist-packages (from importlib-metadata>=1.7.0->great_expectations==0.15.50) (3.21.0)
atisfied: setuptools>=18.5 in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (75.1.0)
 (from Ipython>=7.16.3->great_expectations==0.15.50)
19.2-py2.py3-none-any.whl.metadata (22 kB)
atisfied: decorator in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (4.4.2)
atisfied: pickleshare in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (0.7.5)
atisfied: traitlets>=4.2 in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (5.7.1)
atisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (3.(
atisfied: pygments in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (2.18.0)
atisfied: backcall in /usr/local/lib/python3.11/dist-packages (from Ipython>=7.16.3->great_expectations==0.15.50) (0.2.0)
```

✓ 0s completed at 7:23 PM

Commands + Code + Text RAM Disk

2. Import Necessary Libraries

```
[1]
import great_expectations as gx
import pandas as pd
import numpy as np
```

3. Load Labels.csv

```
# If the file is in your Google Drive, make sure you mount Drive first:
# from google.colab import drive
# drive.mount('/content/drive')

# Adjust the path to your CSV file accordingly
df = pd.read_csv("Labels.csv")
```

Commands + Code + Text RAM Disk

4. Preview the Dataset

```
# Look at the first 5 rows
df.head()
```

	Timestamp	Car1_Location_X	Car1_Location_Y	Car1_Location_Z	Car2_Location_X	Car2_Location_Y	Car2_Location_Z	Occluded_Image_view	Occluding_Car_view
0	1736796157	-51.402977	143	0.596902	-59.320270	140	0.596902	A_001.png	B_001.png
1	1736796167	-53.819637	143	0.596902	-59.196568	140	0.596902	A_002.png	B_002.png
2	1736796178	-50.239144	143	0.596902	-56.744479	140	0.596902	A_003.png	B_003.png
3	1736796188	-53.707220	143	0.596902	-57.309380	140	0.596902	A_004.png	B_004.png
4	1736796198	-52.053721	143	0.596902	-59.545897	140	0.596902	A_005.png	B_005.png

Next steps: Generate code with df | View recommended plots | New interactive sheet

5. Set Up Great Expectations Context and Data Source

```
[5] import great_expectations as gx

    # Initialize a DataContext using the new API
    context = gx.get_context()
```

```
INFO:great_expectations.data_context.types.base:Created temporary directory '/tmp/tmp8k8w2jf4' for ephemeral docs site
```

6. Define and Create a Data Batch

```python
from great_expectations.core.batch import RuntimeBatchRequest

# Create a RuntimeBatchRequest to wrap your existing DataFrame
batch_request = RuntimeBatchRequest(
    datasource_name="my_pandas_datasource",           # Your previously added datasource name
    data_connector_name="default_runtime_data_connector_name",  # Your connector name
    data_asset_name="my_data_asset",                    # An arbitrary asset name
    runtime_parameters={"batch_data": df},             # Your loaded DataFrame
    batch_identifiers={"default_identifier": "default_identifier"}
)

# Print the batch_request to confirm it's created
print(batch_request)
```


```
{
  "datasource_name": "my_pandas_datasource",
  "data_connector_name": "default_runtime_data_connector_name",
  "data_asset_name": "my_data_asset",
  "runtime_parameters": {
    "batch_data": "<class 'pandas.core.frame.DataFrame'>"
  },
  "batch_identifiers": {
    "default_identifier": "default_identifier"
  }
}
```


Expectation 1

+ Code + Text

```python
[17] # Write code here
# Expectation 1: Ensure "Car1_Location_X" values are between 0 and 1000
validator.expect_column_values_to_be_between(
    column="Car1_Location_X",
    min_value=0,
    max_value=1000
)
```

```
WARNING:py.warnings:/usr/local/lib/python3.11/dist-packages/great_expectations/expectations/expectation.py:1477: UserWarning: `result_format` configured
  warnings.warn(
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 107.74it/s]


```
{
  "success": false,
  "expectation_config": {
    "expectation_type": "expect_column_values_to_be_between",
    "kwargs": {
      "column": "Car1_Location_X",
      "min_value": 0,
      "max_value": 1000,
      "batch_id": "c74d5a16eef4c7b627dbb5a322b6018b"
    },
    "meta": {}
  },
  "result": {
    "element_count": 121,
    "unexpected_count": 121,
    "unexpected_percent": 100.0,
    "partial_unexpected_list": [
      -51.40297655,
```


✓ 0s completed at 7:23 PM

Validate Data Against Expectation 1

```python
[18] # Write code here
# Validate Expectation 1
print("Validation Result for Expectation 1:")
result1 = validator.validate()
print(result1)
```


```
      },
      "result": {
        "element_count": 121,
        "unexpected_count": 121,
        "unexpected_percent": 100.0,
        "partial_unexpected_list": [
          -51.40297655,
          -53.81963722,
          -50.23914439,
          -53.70722021,
          -52.05372109,
          -53.93975603,
          -50.30258412,
          -53.17447194,
          -52.72667437,
          -50.18179353,
          -52.40699613,
          -52.38122971,
          -53.01906414,
          -50.85034015,
          -51.93070037,
          -50.75051989,
          -50.63015195,
          -50.69818291,
          -51.95966168,
          -50.88663347
```


✓ 0s completed at 7:23 PM

Expectation 2

[19]
```
# Write code here
# Expectation 2: "Timestamp" values should be unique
validator.expect_column_values_to_be_unique(
    column="Timestamp"
)
```

```
WARNING:py.warnings:/usr/local/lib/python3.11/dist-packages/great_expectations/expectations/expectation.py:1477: UserWarning: `result_format` configured
  warnings.warn(
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 123.49it/s]


```
{
  "success": true,
  "expectation_config": {
    "expectation_type": "expect_column_values_to_be_unique",
    "kwargs": {
      "column": "Timestamp",
      "batch_id": "c74d5a16eef4c7b627dbb5a322b6018b"
    },
    "meta": {}
  },
  "result": {
    "element_count": 121,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {}
```


✓ 0s completed at 7:23 PM

Validate Data Against Expectation 2

[20]
```
# Write code here
# Validate Expectation 2
print("Validation Result for Expectation 2:")
result2 = validator.validate()
print(result2)
```

```
Validation Result for Expectation 2:
```

Calculating Metrics: 100% 13/13 [00:00<00:00, 63.19it/s]


```
{
  "success": false,
  "results": [
    {
      "success": false,
      "expectation_config": {
        "expectation_type": "expect_column_values_to_be_between",
        "kwargs": {
          "column": "Car1_Location_X",
          "min_value": 0,
          "max_value": 1000,
          "batch_id": "c74d5a16eef4c7b627dbb5a322b6018b"
        },
        "meta": {}
      },
      "result": {
        "element_count": 121,
        "unexpected_count": 121,
        "unexpected_percent": 100.0,
        "partial_unexpected_list": [
          -51.40297655,
```


✓ 0s completed at 7:23 PM

Expectation 3

[22]
```
# Clean column names to remove spaces
df.columns = df.columns.str.strip()
```

[25]
```
# New Expectation 3: "Car2_Location_X" should be between 0 and 1000
validator.expect_column_values_to_be_between(
    column="Car2_Location_X",
    min_value=0,
    max_value=1000
)
```

```
WARNING:py.warnings:/usr/local/lib/python3.11/dist-packages/great_expectations/expectations/expectation.py:1477: UserWarning: `result_format` configured
  warnings.warn(
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 107.09it/s]


```
{
  "success": false,
  "expectation_config": {
    "expectation_type": "expect_column_values_to_be_between",
    "kwargs": {
      "column": "Car2_Location_X",
      "min_value": 0,
      "max_value": 1000,
      "batch_id": "c74d5a16eef4c7b627dbb5a322b6018b"
    },
    "meta": {}
  },
  "result": {
    "element_count": 121,
    "unexpected_count": 121,
```


✓ 0s completed at 7:23 PM

Expectation 1

```
[17] # Write code here
     # Expectation 1: Ensure "Car1_Location_X" values are between 0 and 1000
     validator.expect_column_values_to_be_between(
         column="Car1_Location_X",
         min_value=0,
         max_value=1000
     )
```

Explanation:

This expectation ensures that all values in the Car1_Location_X column fall within a realistic range — between 0 and 1000. This column likely represents the X-axis coordinate of a car in a simulation or image frame.

This is important because it ensures data consistency for positional tracking, helps catch sensor glitches or out-of-bound values that could break visualization or modeling tasks, and validates that coordinate data is constrained to expected environment dimensions.

Expectation 2

```
# Write code here
# Expectation 2: "Timestamp" values should be unique
validator.expect_column_values_to_be_unique(
    column="Timestamp"
)
```

Explanation:

This expectation checks that each Timestamp value appears only once — meaning every data entry represents a unique moment in time.

This is important because it prevents duplicate entries that could distort time-series analysis or tracking accuracy, ensures chronological consistency in simulations or video frame-by-frame processing, and supports the assumption that time progresses forward without overlaps.

Expectation 3

```
# Clean column names to remove spaces
df.columns = df.columns.str.strip()
```

```
[25] # New Expectation 3: "Car2_Location_X" should be between 0 and 1000
     validator.expect_column_values_to_be_between(
         column="Car2_Location_X",
         min_value=0,
         max_value=1000
     )
```

Explanation:

This expectation validates that the Car2_Location_X column contains coordinate values within the expected 0–1000 range for the second car.

This is important because it confirms that Car 2’s position is also within the defined scene or simulation space, detects outliers or mislabels (e.g., negative values or values way beyond screen limits), and helps maintain geometric and spatial validity in datasets with multiple moving entities.

Task 2

Why Might This Data Point Be Mislabeled?

1. Inconsistent Feature Distribution

CleanLab typically identifies data points whose features (e.g., numeric measurements, categorical indicators) do not align with the usual distribution for the assigned class/label. If a point is labeled \u201cClass A\u201d but its features look statistically more similar to \u201cClass B,\u201d CleanLab scores that point as suspicious.

2. Model Confusion

When a trained classifier consistently predicts a different label than the one assigned, and this mismatch cannot be easily explained by randomness, it suggests the label might be incorrect.

3. Outlier Within Its Labeled Class

The flagged point could be an extreme outlier if you look at the feature space for its stated label. In other words, it sits far away (in distance or probability) from other samples sharing the same label.

Which Feature Values Could Have Caused the Misclassification?

1. Feature(s) That Deviate from the Typical Range:

One or more features in this data point may be outside the normal range observed for its assigned label. For example, in a flower dataset (like Iris), a \u201cSetosa\u201d labeled point might have a petal length or width typically seen only in \u201cVirginica.\u201d In a tabular classification dataset (e.g., Adult income), a point labeled \u201c>50K\u201d might have age, education, or occupation values that usually align with \u201c\u226450K,\u201d or vice versa.

2. Feature(s) That Strongly Overlap with Another Class:

Even if the feature values are within a plausible range, they might better match the distribution of a different class. CleanLab compares how likely each point\u2019s features are under each possible label.

3. Out-of-Pattern Categorical Combination:

If you have categorical variables, sometimes a combination of categories is nearly impossible (or extremely rare) for a valid label. For instance, if \u201cmarital status = single\u201d and \u201crelationship = husband\u201d appeared together, that inconsistent pairing might raise suspicion.

Example Explanation (adapt this logic to your specific dataset): \u201cCleanLab flagged data point #123 because its features (e.g., PetalLength=4.7, PetalWidth=1.5) are uncharacteristically high for the label Iris-setosa and align more closely with typical Iris-versicolor measurements. This discrepancy makes the model more confident the correct label is \u2018versicolor,\u2019 thus suggesting the sample was mislabeled as \u2018setosa.\u2019\u201d

Task 3

1. Do these suspected anomalous data points match what you expect for their species? Why or why not?

Typically, no. The flagged data points often have feature values (e.g., petal length, sepal width) that deviate from the usual range for their assigned species.

- In the Iris dataset, each species\u2014Iris setosa, Iris versicolor, and Iris virginica\u2014tends to cluster in feature space (e.g., petal length, petal width).
- CleanLab identifies anomalies by comparing these data points\u2019 features against typical patterns for their labeled species.
- If a point labeled Iris setosa shows petal lengths or widths more common in Iris versicolor or Iris virginica, it raises a flag.
- Hence, the suspected anomalies usually do not match what we\u2019d expect from that species, which is why they\u2019re labeled as suspicious.

2. Which feature (sepal length, petal length, etc.) seems most unusual in these points?

Often, petal length or petal width are the most telling features in the Iris dataset, but it depends on your specific results.

Detailed Reasoning: In many versions of the Iris data, petal length is a strong differentiator:

Setosa tends to have the smallest petals,

Virginica the largest,

Versicolor in between.

If a flagged \u201csetosa\u201d point has a petal length closer to the virginica or versicolor range, that\u2019s unusual.

Similarly, sepal width can also show anomalies if the sample is drastically larger or smaller than typical for its labeled species.

Refer to the notebook\u2019s summary or feature importance: whichever feature has the biggest discrepancy is typically the culprit.

3. How can you check if these values are truly anomalies using the original dataset?

Compare the flagged points' raw measurements against the entire dataset and use domain knowledge or additional validation to confirm.

1. Direct Inspection

- Look up the raw rows in the original dataset or CSV file
- Identify potential data entry errors or typos (e.g., missing decimal place)

2. Statistical Comparisons

Descriptive Statistics

- Examine mean, median, and standard deviation for each species
- Determine if flagged points fall far outside the normal range

Visualization

- Create histograms or boxplots for each feature by species
- Identify points:
  - Beyond whiskers
  - In regions unoccupied by other points of the same label

3. Cross-Validation Techniques

Classification Validation

- Train a simple classifier on the main dataset
- Check if the flagged points are consistently misclassified

Model Performance Test

- Remove flagged points from the dataset
- Re-run training
- Compare model performance to detect potential mislabeled data

4. Domain Knowledge Verification

- Consult domain experts (e.g., botanists for Iris dataset)
- Confirm if specific measurements are:
  - Impossible
  - Extremely rare for that species