**Data Quality Assessment Framework (DQAF)**

**Documentation**

**Prepared By: BDR Team**

**Date: May 28, 2025**

**Version: 2.0**

[1. Introduction 2](#_Toc1424875359)

[1.1 Purpose of the document 2](#_Toc112748661)

[1.2 Key features of DQAF 2](#_Toc1057197878)

[2. Overview of DQAF assessment functions 3](#_Toc1785094927)

[2.1. Coordinate Precision 3](#_Toc177552655)

[2.2. Coordinate Location in Australia 3](#_Toc2070566516)

[2.3. Coordinate Completeness 4](#_Toc2119327273)

[2.4. Coordinate Unusual 4](#_Toc283641800)

[2.5. Assess Geospatial Accuracy Precision 4](#_Toc695370952)

[2.6. Assess Duplicate Value Combination 4](#_Toc1470420763)

[2.7. Datum Validation 5](#_Toc1976688127)

[2.8. Date Recency 5](#_Toc366165859)

[2.9. Date Format Validation 5](#_Toc640556956)

[2.10. Date Completeness 6](#_Toc1170104300)

[2.11. Scientific Name Completeness 6](#_Toc1234645523)

[2.12. Datum Completeness 6](#_Toc219705836)

[2.13. Datum Validation 6](#_Toc264879883)

[2.14. Coordinate Outlier Detection IQR Method 6](#_Toc2108745333)

[2.15. Coordinate Outlier Detection Isolation Forest Method 7](#_Toc2071437022)

[2.16. Coordinate Outlier Detection Robust Covariance Method 7](#_Toc1390524846)

[2.17. Coordinate Outlier Detection Z-score Method 7](#_Toc158204042)

[2.18. Date Outlier Detection K-means Method 7](#_Toc2083334965)

[2.19. Date Outlier Detection IQR Method 8](#_Toc1130872646)

[2.20. Scientific Name Validation (Not functional at this time) 8](#_Toc770795401)

[3. DQAF scoring mechanism 9](#_Toc748104399)

[3.1 Understanding the Weighting System 9](#_Toc77930477)

[3.2 How overall scores are calculated 9](#_Toc481549550)

[3.3 Calculating the overall score 10](#_Toc421264321)

[4. Use case labelling in DQAF 10](#_Toc1083892833)

[4.1 Understanding the use case structure 11](#_Toc1594703688)

[4.2 How to define and apply use cases 11](#_Toc1248189051)

[4.3 Example use case definition and labelling 12](#_Toc442773721)

[4.4 Example labelling process 12](#_Toc2084034228)

# 1. Introduction

## 1.1 Purpose of the document

The Data Quality Assessment Framework (DQAF) is a tool designed to assess the quality of RDF datasets in the BDR based on a series of structured functions, allowing users to evaluate and then to apply weighted numerical scores or categorical labels indicating the suitability of the observation records for specific purposes.

This document is aimed at end-users and describes three main aspects of the DQAF functionality: performing assessments using predefined functions, scoring observation records and datasets based on customizable criteria, and labelling each occurrence within a dataset for use-case suitability.

This guide is designed to be straightforward and user-friendly, making it easy for end-users to understand and apply the DQAF.

## 1.2 Key features of DQAF

1.  **Data Quality Assessments:** 20 assessment functions are applied to each occurrence in a dataset, to verify different quality aspects such as completeness, consistency, and accuracy. Based on future requirements, we can edit existing functions or add more.

2.  **Scoring System for each occurrence**: Based on a weighted scoring system, observation records are evaluated and given a cumulative score between 0 and 1, where 0 is lowest quality and 1 is highest quality.

3. **Use Case Labels**: Each occurrence is labelled according to its applicability to specific use cases, to help identify which data are most relevant for a particular use case. Labelling for use cases allows users to categorize data based on its relevance and suitability for specific applications. This ensures that high-quality data is used where it is most impactful, and lower-quality data is appropriately flagged for improvement or limited use. Examples of labels used are “Baseline-SDMFFP1” (highest quality for Baseline SDM use case), “Baseline-SDMFFP2” (medium quality) and “Baseline-SDMFFP3” (lowest quality).

This document ensures users can confidently apply the DQAF to meet their needs and tailor it to their specific data quality requirements.

# 2. Overview of DQAF assessment functions

This section provides an overview of the functions used in BDR’s DQAF to evaluate data quality, focusing on attributes such as coordinate precision, completeness, unusual coordinate patterns, location, outlier detection, date validation, scientific name, and datum information.

## 2.1. Coordinate Precision

**Definition**

This function assesses the level of precision in the latitude and longitude values by evaluating the number of decimal places provided. A higher number of decimal places typically indicates greater precision in geospatial data.

**SPARQL Implementation:**

The function is executed using a SPARQL query (assess\_coordinate\_precision.sparql) which counts the number of decimal digits in both geo:lat and geo:long fields and classifies the precision level accordingly.

**Python Data Generation**

Synthetic RDF data is generated using data\_generation\_coordinate\_precision.py to simulate varying levels of coordinate precision (e.g., 1.23 vs. 151.209333). This helps test the accuracy of precision assessment.

**Pytest-Based Unit Testing**

The unit test function test\_coordinate\_precision() in test\_dqaf\_sparql\_dimensions.py validates that the SPARQL query correctly assigns the expected precision level to each test occurrence.

**Assessment Levels**

* DQAF.coordinate\_precision:Low: Either latitude or longitude has fewer than 2 decimal digits, or one of the coordinates is missing.
* DQAF.coordinate\_precision:Medium: Both latitude and longitude have 2 to 4 decimal digits.
* DQAF.coordinate\_precision:High: Both latitude and longitude have more than 4 decimal digits.

## 2.2. Coordinate Location in Australia

**Purpose**

This rule checks whether the provided coordinate is located within the geographic boundaries of mainland Australia. It ensures that spatial data corresponds to valid regions of interest and excludes data points from outside the expected national scope.

**Methodology**

The Python implementation defines a bounding box that covers mainland Australia, roughly from longitude **113.0** to **154.0**, and latitude **-44.0** to **-10.0**. Each coordinate is checked to determine if it falls within this bounding rectangle.

* Coordinates that fall **within** the bounding box are marked as "inside\_australia".
* Coordinates that fall **outside** the box are marked as "outside\_australia".

**SPARQL Integration**

The result is asserted using SPARQL INSERT operations with:

* sosa:observedProperty linked to dqaf:coordinate\_inside\_australia\_check
* schema:value containing "inside\_australia" or "outside\_australia" as a literal

**Test Coverage**

The function is tested using various points across Australia, as well as outside the bounding area, using pytest. Test cases verify correct labeling for edge cases and ensure the logic behaves correctly for all valid inputs.

## 2.3. Coordinate Completeness

**Purpose**

To check if both latitude and longitude are present and not missing from the dataset. Missing coordinates can severely impact geospatial analysis.

**SPARQL Assessment Logic**

The query in assess\_coordinate\_completeness.sparql looks for geometry instances that are missing geo:asWKT values and flags the associated observations.

**Data Generation**

The file data\_generation\_coordinate\_completeness.py creates RDF samples, some with complete geometry information and others intentionally missing either the geometry triple or the WKT value.

**Test Approach**

Using test\_dqaf\_sparql\_dimensions.py, tests validate that incomplete coordinate cases (i.e., missing lat/lon data) are flagged correctly as incomplete\_coordinate.

**Output**

Each observation is annotated as either incomplete\_coordinate or complete\_coordinate, using the dqaf:hasResult predicate.

## 2.4. Coordinate Unusual

**Purpose**

To detect geographic coordinates that are statistically unusual compared to the general distribution of coordinates in the dataset. This is often indicative of input errors or rare but potentially correct edge cases.

**SPARQL Assessment Logic**

The SPARQL query in assess\_coordinate\_unusual.sparql calculates the mean and standard deviation of the latitude and longitude values and flags coordinates that are more than 3 standard deviations away from the mean.

**Data Generation**

The Python file data\_generation\_coordinate\_unusual.py generates normal coordinates centered around Australia and inserts a small number of outliers significantly outside the norm to simulate unusual data.

**Test Approach**

The test\_dqaf\_sparql\_dimensions.py file includes unit tests to verify the correct identification of unusual coordinates using Pytest. The test validates that coordinates lying far outside the expected ranges are flagged correctly.

**Output:**  
Each RDF observation receives a classification: unusual\_coordinate or normal\_coordinate, added using the dqaf:hasResult property, referencing dqaf:fullResults.

## 2.5. Assess Geospatial Accuracy Precision

**Definition**

This metric checks the **number of decimal places** in coordinate values to assess how precise the spatial data is. Higher decimal places indicate more accurate locations.

**Importance**

* 1 decimal place ≈ 11 km precision
* 4 decimal places ≈ 10 m precision (suitable for most spatial analyses)
* Low precision can affect mapping, modeling, and data integration.

**Threshold**

* The default threshold is **4 decimal places** for both latitude and longitude.
* Coordinates with fewer decimals are flagged as **low precision**.

**SPARQL Query**

assess\_geo\_spatial\_accuracy\_precision.sparql checks each WKT coordinate string for decimal precision.

**Test Data**

The script data\_generation\_geospatial\_accuracy\_precision.py creates examples with various precision levels.  
Tests are in test\_dqaf\_sparql\_dimensions.py.

## 2.6. Assess Duplicate Value Combination

**Definition**

This rule identifies duplicate entries in the dataset where the same combination of latitude, longitude, and timestamp appears more than once. Such duplication may indicate redundant records or data entry errors, which can affect analysis and lead to misinterpretation of data trends.

**SPARQL Logic**

The assess\_duplicate\_entries.sparql query uses a GROUP BY clause over longitude, latitude, and timestamp, followed by a HAVING(COUNT(\*) > 1) clause to detect identical combinations that occur more than once in the RDF store.

**Test Data**

The test data generator data\_generation\_duplicate\_entries.py creates RDF triples with intentional duplication of coordinates and timestamps across multiple observations to simulate real-world redundancy.

**Assessment Process**

* Observations are parsed.
* Latitude, longitude, and timestamps are extracted and grouped.
* Duplicate groups are marked as having issues.
* Unique observations are marked as clean.

**Result Representation**

Each duplicate observation is linked to a result node via dqaf:hasResult, where the schema:value is "duplicate" or "unique".

## 2.7. Datum Validation

**Objective**

Validate that the datum used is from a list of recognized types (e.g., GDA94, WGS84).

**SPARQL Rule**

The query validates whether the geo:datum value is within an accepted controlled vocabulary. Invalid or unknown values are flagged.

**Test Data Generation**

The test dataset includes valid datums like "GDA94" and "WGS84", as well as incorrect or unrecognized entries to test the rule's accuracy.

**Expected Output**

Observations are labeled based on their datum type:

* "valid\_datum" if it matches accepted standards,
* "invalid\_datum" if it does not.

## 2.8. Date Recency

**Definition**

This rule evaluates whether the temporal values in the dataset are recent relative to a defined reference year. A common threshold for recency is set at **2005**. Observations with dates earlier than this threshold may represent outdated records, which could skew temporal analyses or lead to incorrect assumptions about the timeliness of the data.

**SPARQL Logic**

The assess\_date\_recency.sparql query extracts dates from RDF observations, converts them into a comparable format, and classifies them based on whether they are older or newer than the defined threshold (2005). Observations before 2005 are tagged as "old\_date", while the rest are marked as "recent\_date".

**Test Data**

The data\_generation\_date\_recency.py script creates RDF test data with a mix of historical and recent dates to evaluate the rule's performance. It includes observations spread across several decades to simulate a realistic dataset with both old and current entries.

**Assessment Process**

* Observations are parsed from RDF.
* Dates are extracted and converted to numerical format.
* Comparisons are made against the threshold year.
* Each date is labeled as "old\_date" or "recent\_date" accordingly.

**Result Representation**

Each observation is associated with a result via dqaf:hasResult. The result node contains schema:value of "old\_date" or "recent\_date" to indicate its classification.

## 2.9. Date Format Validation

**Definition**

This rule ensures that date values are provided in a standardized and parsable format, typically ISO 8601 (e.g., YYYY-MM-DD). Incorrect formatting can cause parsing errors, hinder integration with other datasets, and lead to erroneous computations.

**SPARQL Logic**

The assess\_date\_format\_validation.sparql query applies regex or type validation on date literals to confirm conformance to accepted formats.

**Test Data**

The data\_generation\_date\_format\_validation.py script produces RDF data containing both valid and invalid date formats, such as "2023-01-15" versus "15/01/2023" or "January 15th, 2023".

**Assessment Process**

* Dates are extracted from RDF literals.
* Format validation is applied using regular expressions.
* Correctly formatted entries are marked "valid\_format\_date".
* Entries with inconsistent or erroneous formatting are tagged "invalid\_format\_date".

**Result Representation**

Each observation links to a result node with a schema:value of "valid\_format\_date" or "invalid\_format\_date" indicating compliance with date formatting standards.

## 2.10. Date Completeness

**Definition**

This rule checks whether each observation contains a valid and non-missing date field. Missing or null date values can lead to incomplete analyses, especially in time-sensitive studies.

**SPARQL Logic**

The assess\_datum\_completeness.sparql query checks for the presence of a sosa:resultTime or equivalent date-related property. If the date is missing, the observation is flagged as incomplete.

**Test Data**

The data\_generation\_date\_completeness.py script generates RDF observations with a mixture of complete and incomplete date fields. This mimics common real-world issues where timestamps may be lost or omitted.

**Assessment Process**

* Observations are scanned for date fields.
* Presence or absence is evaluated.
* Observations with valid dates are marked "complete\_date".
* Missing dates are labeled "missing\_date".

**Result Representation**

Each result is linked to the observation and tagged with schema:value as "complete\_date" or "missing\_date"

## 2.11. Scientific Name Completeness

**Definition**

This rule evaluates whether the scientific name field exists for each observation. Missing scientific names can reduce the semantic richness of the dataset and hinder integration with biodiversity knowledge systems.

**SPARQL Logic**

The assess\_scientific\_name\_completeness.sparql query identifies observations where the scientific name is either missing entirely or has a null-like value (e.g., empty string). It checks for the presence of the dwc:scientificName property.

**Test Data**

The script data\_generation\_scientific\_name\_completeness.py generates RDF data where some observations are deliberately left without a scientific name, simulating incomplete metadata.

**Assessment Process**

* Each observation is checked for a dwc:scientificName value.
* If missing or empty, the observation is flagged as "incomplete".
* If present, it's marked as "complete".

**Result Representation**

Results are expressed using dqaf:hasResult linking the observation to a result node. The schema:value indicates either "complete" or "incomplete".

## 2.12. Datum Completeness

**Objective**

Ensure that each geometry record in the dataset includes a defined datum property. A missing datum may lead to inaccurate spatial interpretations or transformations.

**SPARQL Rule**

The rule checks for the presence of a geo:datum property within each geometry object. Observations with missing datum values are flagged.

**Test Data Generation**

In the test dataset, several geometries are intentionally left without a geo:datum to simulate missing entries and trigger the rule.

**Expected Output**

Each observation is evaluated, and if its associated geometry lacks a datum, it will be marked as "missing\_datum"; otherwise, it is "datum\_present".

## 2.13. Datum Validation

**Objective**

Validate that the datum used is from a list of recognized types (e.g., GDA94, WGS84).

**SPARQL Rule**

The query validates whether the geo:datum value is within an accepted controlled vocabulary. Invalid or unknown values are flagged.

**Test Data Generation**

The test dataset includes valid datums like "GDA94" and "WGS84", as well as incorrect or unrecognized entries to test the rule's accuracy.

**Expected Output**

Observations are labeled based on their datum type:

* "valid\_datum" if it matches accepted standards,
* "invalid\_datum" if it does not.

## 2.14. Coordinate Outlier Detection IQR Method

**Definition**

This rule detects spatial outliers using the Interquartile Range (IQR) method. Points that fall below Q1 - 1.5×IQR or above Q3 + 1.5×IQR in either latitude or longitude are flagged as outliers.

**Detection Logic**

The coordinate\_outlier\_iqr.py function calculates Q1 and Q3 for both latitude and longitude, derives the IQR, and flags any coordinate beyond the whisker boundaries as an outlier.

**Test Data**

data\_generation\_coordinate\_outlier\_iqr.py generates synthetic RDF triples with points both within and outside the normal spatial range to validate the detection.

**Result Representation**

Each observation is assessed and tagged with dqaf:hasResult linking to a value of either outlier\_coordinate or normal\_coordinate.

## 2.15. Coordinate Outlier Detection Isolation Forest Method

**Definition**

This rule applies a machine learning-based approach (Isolation Forest) to detect anomalous or outlier geographic coordinates in the dataset. Isolation Forest is effective in identifying rare or unusual observations by isolating them from the rest of the data points based on recursive partitioning.

**Implementation Summary**

The model is trained on coordinate data (longitude, latitude) using the script train\_isolation\_forest\_model.py, producing isolation\_forest\_model.pkl. During assessment, the script coordinate\_outlier\_isolation\_forest.py loads this trained model and applies it to new RDF data to identify outliers.

**Test and Training Data**

* Training data is generated by data\_generation\_coordinate\_outlier\_isolation\_forest\_train.py, which simulates realistic and edge-case spatial distributions.
* Test data is prepared by data\_generation\_coordinate\_outlier\_isolation\_forest\_test.py to validate the model on unseen examples.

**Assessment Process**

* The Isolation Forest model predicts whether a point is an outlier (-1) or normal (1).
* The model also provides anomaly scores to indicate the strength of outlierness.
* Results are written to RDF using dqaf:hasResult and schema:value as "outlier\_coordinate" or "normal\_coordinate".

**Threshold**

The decision is based on the Isolation Forest's internal scoring mechanism. The boundary is implicitly defined by the model's contamination parameter during training, which determines the expected proportion of outliers.

**Result Representation**

* Each coordinate observation is linked to a result node.
* The value is set to either "outlier\_coordinate" or "normal\_coordinate".
* A map visualization is generated (coordinate\_outliers\_isolation\_forest\_map.html) to highlight normal (green) and outlier (red) points based on location and score.

## 2.16. Coordinate Outlier Detection Robust Covariance Method

**Definition**

This rule applies a multivariate outlier detection technique based on Robust Covariance estimation, useful when spatial data is influenced by correlation between latitude and longitude.

**Detection Logic**

coordinate\_outlier\_robust\_covariance.py fits a robust covariance model to the dataset and computes distances to detect spatial outliers.

**Test Data**

data\_generation\_coordinate\_outlier\_robust\_covariance.py includes multivariate normal points and correlated anomalies to test the model's robustness.

**Result Representation**

Outliers are flagged and mapped in test\_dq\_coordinate\_outlier\_robust\_covariance\_map.html, with RDF results tagged similarly to other outlier functions.

## 2.17. Coordinate Outlier Detection Z-score Method

**Definition**

This method identifies spatial outliers by computing the Z-score of each coordinate. Observations with a Z-score above a specified threshold (commonly ±3.0) are considered outliers.

**Detection Logic**

coordinate\_outlier\_zscore.py standardizes latitude and longitude values, computes Z-scores, and flags points that fall outside the threshold.

**Threshold**

The threshold is usually set to ±3.0. This value can be adjusted based on sensitivity needs.

**Test Data**

data\_generation\_coordinate\_outlier\_zscore.py includes normal and extreme values to ensure both typical and anomalous cases are tested.

**Result Representation**

Results are visualized in test\_dq\_coordinate\_outlier\_zscore\_map.html and saved as RDF triples with schema:value set to outlier\_coordinate or normal\_coordinate.

## 2.18. Date Outlier Detection K-means Method

**Definition**

This rule identifies temporal outliers in observation records based on K-Means clustering applied to timestamp data. The method detects dates that deviate significantly from temporal clusters, indicating possibly erroneous or rare entries.

**Implementation Details**

* The algorithm clusters observation dates into a predefined number of groups (e.g., 3 clusters).
* Dates that are significantly far from the cluster centers (based on distance threshold) are labeled as outliers.

**Test Data**

The data\_generation\_date\_outlier\_iqr\_kmeans.py script generates sample RDF observations distributed across clusters of typical dates and some rare or erroneous dates to simulate outliers.

**Assessment Process**

* Extract timestamps from RDF observations.
* Transform dates to numerical values (e.g., ordinal).
* Cluster dates using K-Means.
* Measure distance from cluster centers.
* Label distant dates as outliers.

**Result Representation**

Each observation is linked to a result node via dqaf:hasResult, with schema:value set as "outlier\_date" or "normal\_date" based on its distance from the cluster center.

## 2.19. Date Outlier Detection IQR Method

**Definition**

This rule detects temporal anomalies using the Interquartile Range (IQR) method. Dates that fall below Q1 − 1.5×IQR or above Q3 + 1.5×IQR are considered temporal outliers.

**Implementation Details**

* Convert dates into ordinal form for numeric analysis.
* Compute Q1 (25th percentile), Q3 (75th percentile), and IQR.
* Detect values outside the [Q1 − 1.5×IQR, Q3 + 1.5×IQR] range.

**Test Data**

The same generator (data\_generation\_date\_outlier\_iqr\_kmeans.py) also creates data for this function, with most dates tightly clustered and a few significantly earlier or later to serve as outliers.

**Assessment Process**

* Parse observation timestamps.
* Convert to numeric representation.
* Calculate IQR thresholds.
* Classify outliers outside of these thresholds.

**Result Representation**

Each observation is assigned a result node via dqaf:hasResult, with a schema:value of "outlier\_date" for outliers and "normal\_date" for typical entries.

## 2.20. Scientific Name Validation (Not functional at this time)

**SPARQL Logic**

The assess\_scientific\_name\_validation.sparql query uses pattern matching to check if the dwc:scientificName follows a valid format (e.g., two words with initial capital for genus, lowercase for species).

**Test Data**

The script data\_generation\_scientific\_name\_validation.py generates test RDF data with a mix of valid and invalid scientific name formats.

**Assessment Process**

* The value of dwc:scientificName is retrieved.
* A regex or rule-based validation is performed on the format.
* Valid names are labeled "valid", and others as "invalid".

**Result Representation**

Each observation links to a result node via dqaf:hasResult, and the schema:value indicates "valid" or "invalid" accordingly.

# 3. DQAF scoring mechanism

The scoring process in DQAF evaluates the quality of each occurrence by applying weighted scores to the results of the assessment functions described above. Each function’s result has an associated weight, allowing for overall scoring to be customised as needed to prioritize different aspects of data quality. The weights used here, found under the BDR\_General\_Weight column in the assertions\_score\_weighting\_definition.xlsx file, serve as an example. Applications or other users of the DQAF can define their unique scoring criteria, adjusting weights to fit specific data quality needs. Based on these custom-defined weights, each occurrence will receive a unique quality score.

## 3.1 Understanding the Weighting System

In the assertions\_score\_weighting\_definition.xlsx file:

- Each specific assessment outcome (e.g., coordinate\_precision:Low, coordinate\_precision:High) has a default assigned weight indicating its impact on the overall score. These default values can be changed in the spreadsheet.

- a higher weight indicates higher importance assigned to a metric.

- users may modify the BDR\_General\_Weight values or define new columns for alternative scoring strategies tailored to specific applications.

For example, if **coordinate\_precision:Low** has a default weight of 0.2.   and **coordinate\_precision:High** has a default weight of 0.5, observation records with high coordinate precision will contribute more to the final score than those with low precision.

## 3.2 How overall scores are calculated

1. **Run assessment functions**: Each data occurrence is evaluated using all assessment functions, resulting in individual results.
2. **Apply weights**: For each result, multiply the result by the corresponding weight.
3. **Aggregate scores**: Sum the weighted values to produce a total quality score for each occurrence.

**Example of scoring calculation**

Consider an occurrence with the following assessment results and weights:

|  |  |  |
| --- | --- | --- |
| **Assessment Function** | **Outcome** | **Weight** |
| coordinate\_precision:High | coordinate\_precision:High | 0.5 |
| coordinate\_completeness:non\_empty | coordinate\_completeness:non\_empty | 0.0 |
| location\_outlier:None | location\_outlier:None | 0.3 |
| date\_validity:Valid | date\_validity:Valid | 0.4 |

## 3.3 Calculating the overall score

Each function’s contribution is determined by multiplying the function’s result (as a numerical value) by its weight.

1. **coordinate\_precision:High** (Weight: 0.5)

* The occurrence meets this outcome, receiving the full weight of 0.5.

1. **coordinate\_completeness:non\_empty** (Weight: 0.0)

* This outcome has a weight of 0.0, so it does not add to the score.

1. **location\_outlier:None** (Weight: 0.3)

* The occurrence meets this outcome, contributing 0.3.

1. **date\_validity:Valid** (Weight: 0.4)

* The occurrence meets this outcome, contributing 0.4.

The total score for this occurrence is calculated as follows: [(0.5) + (0.0) + (0.3) + (0.4) = 1.2]

**Interpreting the scores**

Once calculated, the total score can be interpreted based on quality thresholds, such as:

- **High Quality**: Score ≥ 0.8

- **Medium Quality**: 0.5 ≤ Score < 0.8

- **Low Quality**: Score < 0.5

These thresholds help categorize data observation records according to overall quality, enabling users to filter or prioritize observation records based on specific use case requirements.

# 4. Use case labelling in DQAF

The DQAF framework includes a flexible system for labelling data observation records according to specific use cases. Users or teams can define these use cases, such as Baseline-SDMFFP1, Baseline-SDMFFP2, or custom labels, in the usecase\_definition.xlsx file. This allows applications to filter or categorize biodiversity data based on its suitability for particular requirements, such as research, conservation, or species tracking.

## 4.1 Understanding the use case structure

In the usecase\_definition.xlsx file:

- Each **use case** (e.g., Baseline-SDMFFP1, Baseline-SDMFFP2) is represented by a column.

Each use case category in the usecase\_definition.xlsx file represents a suitability level for specific applications based on data quality requirements. For example:

* Baseline-SDMFFP1: High suitability for baseline species distribution modeling, requiring high-quality data with precise coordinates and valid scientific names.
* Baseline-SDMFFP2: Medium suitability for baseline species distribution modeling, suitable for use cases with moderate data quality.
* Baseline-SDMFFP3: Low suitability for baseline species distribution modeling, acceptable for exploratory analyses or cases with limited data accuracy.

- Each row represents a **data quality assertion** (e.g., coordinate\_precision:Low, coordinate\_completeness:non\_empty).

- Cells contain either a 1 (indicating that the data quality assertion is required for the use case) or are left blank (indicating irrelevance).

For example:

- coordinate\_precision:High has a value of 1 for both Baseline-SDMFFP1 and Baseline-SDMFFP2, indicating that high coordinate precision is required for these use cases.

- Users can add new use cases by creating additional columns and marking relevant assertions with 1s to represent essential criteria.

## 4.2 How to define and apply use cases

1. **Define new use cases**: Add a new column for each use case in usecase\_definition.xlsx and populate it with 1s to represent essential data quality assertions.

* Example: To create a use case category called Conservation\_FFP1, add a column titled Conservation\_FFP1 and mark 1 on the combination of assertions required for this use case, such as coordinate\_precision:High and scientific\_name\_validity:Valid.

1. **Evaluate each occurrence**: During the assessment, each data occurrence is evaluated based on matches with the assertions marked in the selected use case.
2. **Label observation records**: For each occurrence, check if it meets the criteria for the chosen use case (i.e., if it fulfills all marked assertions with 1s in that column). Based on the evaluation, assign a label such as suitable (all marked assertions matched) or not suitable (not all marked assertions matched).

## 4.3 Example use case definition and labelling

Consider the following partial structure in usecase\_definition.xlsx for two use cases in biodiversity studies:

|  |  |  |
| --- | --- | --- |
| **Data quality assertion** | **Baseline-SDMFFP1** | **Baseline-SDMFFP2** |
| coordinate\_precision:High | 1 | 1 |
| coordinate\_completeness:non\_empty | 1 | 1 |
| location\_outlier:None | 1 |  |
| scientific\_name\_validity:Valid | 1 | 1 |
| date\_validity:Valid |  | 1 |

* **Baseline-SDMFFP1** is a general use case requiring high coordinate precision, complete coordinate data, no location outliers, and valid scientific names.
* **Baseline-SDMFFP2** is stricter, additionally requiring valid dates for observation records but does not require the location\_outlier check.

## 4.4 Example labelling process

Assume an occurrence data with the following quality outcomes:

* **coordinate\_precision:High** (meets requirement)
* **coordinate\_completeness:non\_empty** (meets requirement)
* **location\_outlier:None** (meets requirement)
* **scientific\_name\_validity:Valid** (meets requirement)
* **date\_validity:Missing** (does not meet requirement)

For this occurrence:

- It would be labelled as **suitable** for Baseline-SDMFFP1 since all required assertions for this use case are met.

- It would be labelled as **not suitable** for Baseline-SDMFFP2 because it fails the date\_validity:Valid requirement.

This labelling indicates that the occurrence is suitable for conducting basic species distribution modelling, but not for specific applications that need accurate date information, such as tracking seasonal migration patterns.

**Customizing use case requirements**

The flexible structure of DQAF’s use case labelling system allows users to adapt to diverse requirements in biodiversity data analysis. For example, users can:

* **Modify existing use cases**: Update any use case by adjusting the 1s in the corresponding column to include or exclude certain assertions.
* **Add complex use cases**: Define multiple use cases for layered analysis. For example, you might want to separately label observation records for “Flora Conservation” and “Fauna Tracking” to apply different criteria based on plant or animal data.