ODP Dataset Conformance Analysis Script

This script fetches, cleans, and analyses data from the Open Digital Planning (ODP) Datasette to generate a conformance summary for spatial and document datasets. It compares real-time dataset delivery to expected specifications across multiple cohorts and organisations.

Purpose:

- **Evaluate conformance** of datasets submitted by organisations as part of ODP tracks.
- **Summarise performance** by comparing submitted fields with expected specifications.
- **Highlight missing or erroneous data**, and generate structured reports for analysis and presentation.

Key Functionalities:

1. **Provision Check**
   Fetches expected datasets for each organisation based on selected ODP cohorts using the `provision` table.

2. **Field Supply & Mapping Check**
   From `endpoint_dataset_resource_summary`, the script evaluates:

   - Number of fields supplied
   - Number of fields matched against spec
   - Number of fields with errors

3. **Issue Analysis**
   Uses `endpoint_dataset_issue_type_summary` to identify error severity and affected fields per endpoint.

4. **Specification Integration**
   Loads a local JSON-based CSV (`specification.csv`) containing the expected fields per dataset to validate field completeness.

5. **Output Summary**
   Produces:

   - A detailed CSV (`odp-conformance.csv`) with fields supplied, matched, and error-free per endpoint
   - Summary statistics and a breakdown of dataset quality across cohorts

---

Output Columns (per endpoint):

- `organisation`, `dataset`, `licence`, `resource`, `endpoint_no.`
- `field_supplied_pct`, `field_matched_pct`, `field_error_free_pct`
- `field_supplied_count`, `field_matched_count`, `field_error_free_count`

How to Run:

```bash python odp_conformance_summary.py --output-dir ./outputs

In [ ]:
```
"""
Script to fetch, clean, and summarise conformance data from the Open Digital Planni
It checks provisions, endpoint summaries, and issue logs against a specification, p
performance summary for each dataset per organisation.
"""

import json
import logging
import numpy as np
import pandas as pd
import requests
from requests import adapters
from urllib3 import Retry
import argparse
import os

def parse_args():
    """
    Parses command-line arguments for specifying the output directory.

    Returns:
        argparse.Namespace: Parsed arguments containing the output directory path.
    """
    parser = argparse.ArgumentParser(description="Datasette batch exporter")
    parser.add_argument(
        "--output-dir",
        type=str,
        required=True,
        help="Directory to save exported CSVs"
)
    return parser.parse_args()

def get_datasette_http():
    """
    Creates and returns a requests.Session object with retry logic built in.
    Used for robust querying of Datasette endpoints.
    """
    retry_strategy = Retry(total=3, status_forcelist=[400], backoff_factor=0)

    adapter = adapters.HTTPAdapter(max_retries=retry_strategy)

    http = requests.Session()
    http.mount("https://", adapter)
    http.mount("http://", adapter)

    return http

def get_datasette_query(db, sql, filter=None, url="https://datasette.planning.data.
    """
    Executes an SQL query against a Datasette database and returns the result as a

    Args:
        db (str): The database name (e.g. 'digital-land')
        sql (str): SQL query string
        filter (dict, optional): Additional query parameters
        url (str): Base Datasette URL
```
```

```
    Returns:
        pd.DataFrame | None: Query result as a DataFrame or None on failure
    """
    url = f"{url}/{db}.json"
    params = {"sql": sql, "_shape": "array", "_size": "max"}
    if filter:
        params.update(filter)
    try:
        http = get_datasette_http()
        resp = http.get(url, params=params)
        resp.raise_for_status()
        df = pd.DataFrame.from_dict(resp.json())
        return df
    except Exception as e:
        logging.warning(e)
        return None

def get_provisions(selected_cohorts, all_cohorts):
    """
    Queries the Datasette 'provision' table for expected datasets for selected coho

    Args:
        selected_cohorts (list): List of selected cohort IDs (e.g. ['ODP-Track1'])
        all_cohorts (list): List of all valid cohort definitions (each with 'id' ar

    Returns:
        pd.DataFrame: A DataFrame of expected provisions with organisation names ar
    """
    filtered_cohorts = [
        x
        for x in selected_cohorts
        if selected_cohorts[0] in [cohort["id"] for cohort in all_cohorts]
    ]
    cohort_clause = (
        "AND ("
        + " or ".join("c.cohort = '" + str(n) + "'" for n in filtered_cohorts)
        + ")"
        if filtered_cohorts
        else ""
    )
    sql = f"""
    SELECT
        p.cohort,
        p.organisation,
        c.start_date as cohort_start_date,
        org.name as name
    FROM
        provision p
    INNER JOIN
        cohort c on c.cohort = p.cohort
    JOIN organisation org
    WHERE
        p.provision_reason = "expected"
    AND p.project == "open-digital-planning"
    {cohort_clause}
    AND org.organisation == p.organisation
    GROUP BY
        p.organisation
    ORDER BY
        cohort_start_date,
        p.cohort
    """
    provision_df = get_datasette_query("digital-land", sql)
    return provision_df
```

```
SPATIAL_DATASETS = [
    "article-4-direction-area",
    "conservation-area",
    "listed-building-outline",
    "tree-preservation-zone",
    "tree",
]
DOCUMENT_DATASETS = [
    "article-4-direction",
    "conservation-area-document",
    "tree-preservation-order",
]

# Separate variable for all datasets as arbitrary ordering required
ALL_DATASETS = [
    "article-4-direction",
    "article-4-direction-area",
    "conservation-area",
    "conservation-area-document",
    "listed-building-outline",
    "tree-preservation-order",
    "tree-preservation-zone",
    "tree",
]

# Configs that are passed to the front end for the filters
DATASET_TYPES = [
    {"name": "Spatial", "id": "spatial"},
    {"name": "Document", "id": "document"},
]

COHORTS = [
    {"name": "RIPA Beta", "id": "RIPA-Beta"},
    {"name": "RIPA BOPS", "id": "RIPA-BOPS"},
    {"name": "ODP Track 1", "id": "ODP-Track1"},
    {"name": "ODP Track 2", "id": "ODP-Track2"},
    {"name": "ODP Track 3", "id": "ODP-Track3"},
    {"name": "ODP Track 4", "id": "ODP-Track4"},
]


def get_column_field_summary(dataset_clause, offset):
    """
    Retrieves endpoint dataset resource summaries for datasets matching the clause.

    Args:
        dataset_clause (str): SQL filter for datasets (e.g. "edrs.pipeline = 'tree'
        offset (int): Row offset for pagination

    Returns:
        pd.DataFrame: Results from `endpoint_dataset_resource_summary` joined with
    """
    sql = f"""
    SELECT edrs.*, rle.licence
    FROM endpoint_dataset_resource_summary AS edrs
    LEFT JOIN (
        SELECT endpoint, licence, dataset
        FROM reporting_latest_endpoints
    ) AS rle ON edrs.endpoint = rle.endpoint and edrs.dataset = rle.dataset
    LEFT JOIN (
        SELECT endpoint, end_date as endpoint_end_date, dataset
        FROM endpoint_dataset_summary
    ) as eds on edrs.endpoint = eds.endpoint and edrs.dataset = eds.dataset
```

```
    WHERE edrs.resource != ''
    and eds.endpoint_end_date=''
    and ({dataset_clause})
    limit 1000 offset {offset}
    """
    column_field_df = get_datasette_query("performance", sql)

    return column_field_df


def get_issue_summary(dataset_clause, offset):
    """
    Retrieves summarised issue counts per dataset and endpoint.

    Args:
        dataset_clause (str): SQL WHERE clause to filter datasets.
        offset (int): Pagination offset for result set.

    Returns:
        pd.DataFrame: Issue summary from Datasette.
    """
    sql = f"""
    select  * from endpoint_dataset_issue_type_summary edrs
    where ({dataset_clause})
    limit 1000 offset {offset}
    """
    issue_summary_df = get_datasette_query("performance", sql)
    return issue_summary_df


def get_odp_conformance_summary(dataset_types, cohorts):
    """
    Main function that combines provisions, endpoints, and issues to calculate conf

    Args:
        dataset_types (list): One or more of ["spatial", "document"] to filter data
        cohorts (list): List of cohort IDs to include in the summary.

    Returns:
        tuple:
            - dict: Contains headers, rows, stats, and metadata for rendering a rep
            - pd.DataFrame: Detailed CSV output with scores and metadata per datase
    """
    params = {
        "cohorts": COHORTS,
        "dataset_types": DATASET_TYPES,
        "selected_dataset_types": dataset_types,
        "selected_cohorts": cohorts,
    }
    if dataset_types == ["spatial"]:
        datasets = SPATIAL_DATASETS
    elif dataset_types == ["document"]:
        datasets = DOCUMENT_DATASETS
    else:
        datasets = ALL_DATASETS
    dataset_clause = " or ".join(
        ("edrs.pipeline = '" + str(dataset) + "'" for dataset in datasets)
    )

    provision_df = get_provisions(cohorts, COHORTS)

    # Download column field summary table
    # Use pagination in case rows returned > 1000
    pagination_incomplete = True
```

```
    offset = 0
    column_field_df_list = []
    while pagination_incomplete:
        column_field_df = get_column_field_summary(dataset_clause, offset)
        column_field_df_list.append(column_field_df)
        pagination_incomplete = len(column_field_df) == 1000
        offset += 1000
    if len(column_field_df_list) == 0:
        return {"params": params, "rows": [], "headers": []}
    column_field_df = pd.concat(column_field_df_list)

    column_field_df = pd.merge(
        column_field_df, provision_df, on=["organisation", "cohort"], how="left"
    )

    # Optional: Fill missing names or dates (if helpful for display/export)
    column_field_df["organisation_name"] = column_field_df["organisation_name"].fil
    column_field_df["cohort_start_date"] = column_field_df["cohort_start_date"].fil

    # Download issue summary table
    pagination_incomplete = True
    offset = 0
    issue_df_list = []
    while pagination_incomplete:
        issue_df = get_issue_summary(dataset_clause, offset)
        issue_df_list.append(issue_df)
        pagination_incomplete = len(issue_df) == 1000
        offset += 1000
    issue_df = pd.concat(issue_df_list)

    dataset_field_df = get_dataset_field()

    # remove fields that are auto-created in the pipeline from the dataset_field fi
    # ("entity", "organisation", "prefix", "point" for all but tree, and "entity",
    dataset_field_df = dataset_field_df[
        (dataset_field_df["dataset"] != "tree")
        & (
            ~dataset_field_df["field"].isin(
                ["entity", "organisation", "prefix", "point"]
            )
        )
        | (dataset_field_df["dataset"] == "tree")
        & (~dataset_field_df["field"].isin(["entity", "organisation", "prefix"]))
    ]

    # Filter out fields not in spec
    column_field_df["mapping_field"] = column_field_df.replace({'"', ""}).apply(
        lambda row: [
            field
            for field in (
                row["mapping_field"].split(";") if row["mapping_field"] else ""
            )
            if field
            in dataset_field_df[dataset_field_df["dataset"] == row["dataset"]][
                "field"
            ].values
        ],
        axis=1,
    )
    column_field_df["non_mapping_field"] = column_field_df.replace({'"', ""}).apply
        lambda row: [
            field
            for field in (
                row["non_mapping_field"].split(";") if row["non_mapping_field"] els
```

```
            )
            if field
            in dataset_field_df[dataset_field_df["dataset"] == row["dataset"]][
                "field"
            ].values
        ],
        axis=1,
    )

    # Map entity errors to reference field
    issue_df["field"] = issue_df["field"].replace("entity", "reference")
    # Filter out issues for fields not in dataset field (specification)
    issue_df["field"] = issue_df.apply(
        lambda row: (
            row["field"]
            if row["field"]
            in dataset_field_df[dataset_field_df["dataset"] == row["dataset"]][
                "field"
            ].values
            else None
        ),
        axis=1,
    )

    # Create field matched and field supplied scores
    column_field_df["field_matched"] = column_field_df.apply(
        lambda row: len(row["mapping_field"]) if row["mapping_field"] else 0, axis=
    )
    column_field_df["field_supplied"] = column_field_df.apply(
        lambda row: row["field_matched"]
        + (len(row["non_mapping_field"]) if row["non_mapping_field"] else 0),
        axis=1,
    )
    column_field_df["field"] = column_field_df.apply(
        lambda row: len(
            dataset_field_df[dataset_field_df["dataset"] == row["dataset"]]
        ),
        axis=1,
    )

    # Check for fields which have error issues
    results_issues = [
        issue_df[
            (issue_df["resource"] == r["resource"]) & (issue_df["severity"] == "err
        ]
        for index, r in column_field_df.iterrows()
    ]
    results_issues_df = pd.concat(results_issues)

    # Create fields with errors column
    column_field_df["field_errors"] = column_field_df.apply(
        lambda row: len(
            results_issues_df[row["resource"] == results_issues_df["resource"]]
        ),
        axis=1,
    )

    # Create endpoint ID column to track multiple endpoints per organisation-datase
    column_field_df["endpoint_no."] = (
        column_field_df.groupby(["organisation", "dataset"]).cumcount() + 1
    )
    column_field_df["endpoint_no."] = column_field_df["endpoint_no."].astype(str)

    # group by and aggregate for final summaries
```

```
final_count = (
    column_field_df.groupby(
        [
            "organisation",
            "organisation_name",
            "cohort",
            "dataset",
            "licence",
            "endpoint",
            "endpoint_no.",
            "resource",
            "latest_log_entry_date",
            "cohort_start_date",
        ]
    )
    .agg(
        {
            "field": "sum",
            "field_supplied": "sum",
            "field_matched": "sum",
            "field_errors": "sum",
        }
    )
    .reset_index()
)

final_count["field_error_free"] = (
    final_count["field_supplied"] - final_count["field_errors"]
)
final_count["field_error_free"] = final_count["field_error_free"].replace(-1, 0

# add string fields for [n fields]/[total fields] style counts
final_count["field_supplied_count"] = (
    final_count["field_supplied"].astype(int).map(str)
    + "/"
    + final_count["field"].map(str)
)
final_count["field_error_free_count"] = (
    final_count["field_error_free"].astype(int).map(str)
    + "/"
    + final_count["field"].map(str)
)
final_count["field_matched_count"] = (
    final_count["field_matched"].astype(int).map(str)
    + "/"
    + final_count["field"].map(str)
)

# create % columns
final_count["field_supplied_pct"] = (
    final_count["field_supplied"] / final_count["field"]
)
final_count["field_error_free_pct"] = (
    final_count["field_error_free"] / final_count["field"]
)
final_count["field_matched_pct"] = (
    final_count["field_matched"] / final_count["field"]
)

final_count.reset_index(drop=True, inplace=True)
final_count.sort_values(
    ["cohort_start_date", "cohort", "organisation_name", "dataset"], inplace=Tr
)
```

```
provisions_with_100_pct_match = final_count[final_count["field_matched_pct"] ==
percent_100_field_match = (
    round(len(provisions_with_100_pct_match) / len(final_count) * 100, 1)
    if len(final_count)
    else 0
)

out_cols = [
    "cohort",
    "organisation_name",
    "organisation",
    "dataset",
    "licence",
    "endpoint_no.",
    "field_supplied_count",
    "field_supplied_pct",
    "field_matched_count",
    "field_matched_pct",
]

csv_out_cols = [
    "organisation",
    "organisation_name",
    "cohort",
    "dataset",
    "licence",
    "endpoint",
    "endpoint_no.",
    "resource",
    "latest_log_entry_date",
    "field",
    "field_supplied",
    "field_matched",
    "field_errors",
    "field_error_free",
    "field_supplied_pct",
    "field_error_free_pct",
    "field_matched_pct",
]

headers = [
    *map(
        lambda column: {
            "text": make_pretty(column).title(),
            "classes": "reporting-table-header",
        },
        out_cols,
    )
]

rows = [
    [
        {
            "text": make_pretty(cell),
            "classes": "reporting-table-cell " + get_background_class(cell),
        }
        for cell in r
    ]
    for index, r in final_count[out_cols].iterrows()
]

# Calculate overview stats
overview_datasets = [
    "article-4-direction-area",
```

```
        "conservation-area",
        "listed-building-outline",
        "tree",
        "tree-preservation-zone",
    ]
    overview_stats_df = pd.DataFrame()
    overview_stats_df["dataset"] = overview_datasets
    overview_stats_df = overview_stats_df.merge(
        final_count[["dataset", "field_supplied_pct"]][
            final_count["field_supplied_pct"] < 0.5
        ]
        .groupby("dataset")
        .count(),
        on="dataset",
        how="left",
    ).rename(columns={"field_supplied_pct": "< 50%"})
    overview_stats_df = overview_stats_df.merge(
        final_count[["dataset", "field_supplied_pct"]][
            (final_count["field_supplied_pct"] >= 0.5)
            & (final_count["field_supplied_pct"] < 0.8)
        ]
        .groupby("dataset")
        .count(),
        on="dataset",
        how="left",
    ).rename(columns={"field_supplied_pct": "50% - 80%"})
    overview_stats_df = overview_stats_df.merge(
        final_count[["dataset", "field_supplied_pct"]][
            final_count["field_supplied_pct"] >= 0.8
        ]
        .groupby("dataset")
        .count(),
        on="dataset",
        how="left",
    ).rename(columns={"field_supplied_pct": "> 80%"})
    overview_stats_df.replace(np.nan, 0, inplace=True)
    overview_stats_df = overview_stats_df.astype(
        {
            "< 50%": int,
            "50% - 80%": int,
            "> 80%": int,
        }
    )

    stats_headers = [
        *map(
            lambda column: {
                "text": column.title(),
                "classes": "reporting-table-header",
            },
            overview_stats_df.columns.values,
        )
    ]
    stats_rows = [
        [{"text": cell, "classes": "reporting-table-cell"} for cell in r]
        for index, r in overview_stats_df.iterrows()
    ]
    return {
        "headers": headers,
        "rows": rows,
        "stats_headers": stats_headers,
        "stats_rows": stats_rows,
        "params": params,
        "percent_100_field_match": percent_100_field_match,
```

```
    }, final_count[csv_out_cols]

def make_pretty(text):
    """
    Formats text or numerical values for presentation in the UI or report tables.

    Args:
        text (str or float): Raw text or numeric value.

    Returns:
        str: Human-readable formatted string.
    """
    if type(text) is float:
        # text is a float, make a percentage
        return str((round(100 * text))) + "%"
    elif "_" in text:
        # text is a column name
        return text.replace("_", " ").replace("pct", "%").replace("count", "")
    return text

def get_background_class(text):
    """
    Assigns a background class based on the numeric value (for HTML/visual display)

    Args:
        text (float or str): A percentage float value (0.0 to 1.0)

    Returns:
        str: CSS class name string based on value grouping (e.g., 'reporting-90-100
    """
    if type(text) is float:
        group = int((text * 100) / 10)
        if group == 10:
            return "reporting-100-background"
        else:
            return "reporting-" + str(group) + "0-" + str(group + 1) + "0-backgrour
    return ""

def get_dataset_field():
    """
    Loads the official dataset-field specification JSON from a local CSV file.

    Returns:
        pd.DataFrame: Each row represents a dataset/field combination from the JSON
    """
    specification_df = pd.read_csv(
        r"C:\Users\DanielGodden\Documents\MCHLG\collecting_and_managing_data\monitc
    )
    rows = []
    for index, row in specification_df.iterrows():
        specification_dicts = json.loads(row["json"])
        for dict in specification_dicts:
            dataset = dict["dataset"]
            fields = [field["field"] for field in dict["fields"]]
            for field in fields:
                rows.append({"dataset": dataset, "field": field})
    return pd.DataFrame(rows)

if __name__ == "__main__":
    # Parse CLI args
    args = parse_args()
    output_dir = args.output_dir
    output_path = os.path.join(output_dir, "odp-conformance.csv")
```

```
    # Run summary function and filter invalid cohort rows
    _, df = get_odp_conformance_summary(dataset_types=["spatial", "document"], coho
    df = df[df['cohort'].notna() & (df['cohort'].str.strip() != "")]

    # Save final output
    df.to_csv(output_path, index=False)
    print(f"Saved ODP conformance summary to {output_path}")
```