Workflow for Detecting and Visualising Duplicate Geometries in Open Digital Planning Data

Daniel Godden

This set of scripts comprises a complete pipeline for identifying, analysing, and visualising suspected duplicate geometries across UK spatial planning datasets published via the Digital Land platform. It supports diagnostics and quality assurance of data submitted by local authorities and planning bodies.

Script 1: Extract and Enrich Duplicate Geometry Records (`duplicate_entity_expectation.csv`)

This script loads the full `expectation` table from the Digital Land Datasette, filtering for rows where the `operation` is `"duplicate_geometry_check"`. These rows contain nested JSON-style fields under the `details` column that report detected overlaps between spatial features.

- The script parses these details using `ast.literal_eval` and extracts two types of matches:
  - `complete_matches`: two known `entity` records flagged as duplicates.
  - `single_matches`: only one entity detected as potentially problematic.
- It then joins these extracted entity IDs (`entity_a`, `entity_b`) with canonical entity tables from the relevant spatial datasets (`conservation-area`, `article-4-direction-area`, etc.).
- Each entity's geometry, name, entry/end dates, and organisation are appended.
- The final output, `duplicate_entity_expectation.csv`, contains a fully enriched tabular view of all matches, suitable for inspection, analysis, or geospatial plotting.

Script 2: Summarise and Visualise Duplicate Match Counts (`duplicate_geometry_summary.csv`, `duplicate_geometry_plot.png`)

This script provides a statistical and visual summary of the parsed duplicates:

- A helper function extracts key stats from each expectation row: actual vs expected count, and the number of complete/single matches.

- It outputs a summary CSV file (`duplicate_geometry_summary.csv`) that aggregates this information per dataset.
- A stacked bar chart (`duplicate_geometry_plot.png`) visualises the volume of duplicate matches (both types) across datasets.

This enables a high-level overview of where geometry duplication issues are most common, helping prioritise follow-up actions.

Script 3: Interactive Preview of Duplicate Geometries in a Notebook

The third script defines a reusable function `preview_duplicate_geometries(df, n=5)` that allows analysts to visually inspect geometries using an interactive Leaflet map via `geopandas.explore()` .

- It takes `n` rows from the full match table, parses the WKT geometries for both A and B entities, and color-codes them (red for A, blue for B).
- The resulting `folium.Map` object shows entity ID, organisation, and match label as tooltips.

This step enables spot-checking of potential duplicates, making it easier to validate whether overlaps are legitimate or due to data quality issues.

Overall Goal

Together, these scripts support:

- Automatic extraction and transformation of duplication reports from the Open Digital Planning platform.
- Metadata enrichment to connect raw entity IDs to meaningful spatial and organisational information.
- Clear tabular, visual, and geospatial outputs that help users understand and act on geometry duplication patterns in planning data.

In [ ]:
```
# Imports

import pandas as pd
import ast
import matplotlib.pyplot as plt
import geopandas as gpd
from shapely import wkt
from IPython.display import display
```

Duplicate Geometry Check

This script identifies, extracts, and enriches duplicate spatial geometries from UK planning datasets using the Digital Land platform. It performs the following steps:

---

1. **Load the Expectation Table**

The script downloads the full expectations table from Datasette and filters it to only include rows where `operation == "duplicate_geometry_check"`. These rows contain structured details on detected duplicate geometry issues.

2. **Parse the 'details' Field**

The `details` column contains Python-like dictionaries stored as strings. These are parsed into usable dictionaries using `ast.literal_eval`. From each parsed row, two types of matches are extracted:

- `complete_matches`: where two entities (`entity_a`, `entity_b`) share identical or highly similar geometries.
- `single_matches`: where only a single entity is flagged (e.g., it may overlap multiple others without clear pairing).

These matches are flattened into a structured list of dictionaries and converted into a DataFrame `df_matches`, with columns including:

- `dataset`, `operation`, `message`, `entity_a`, `organisation_entity_a`, `entity_b`, `organisation_entity_b`.

3. **Load and Prepare Entity Tables**

A set of relevant entity tables is loaded via streaming CSVs for:

- `conservation-area`, `article-4-direction-area`, `listed-building-outline`, `tree-preservation-zone`, and `tree`.

Each table is trimmed to key columns: `entity`, `dataset`, `end_date`, `entry_date`, `geometry`, `name`, `organisation_entity`. These are stored in a combined `df_entities` DataFrame for later merging.

4. **Merge Metadata for Each Entity**

Metadata for `entity_a` and `entity_b` are added to `df_matches` by joining on `dataset` and `entity`. The merged columns are renamed to:

- For `entity_a`: `entity_a_name`, `entity_a_organisation`, `entity_a_entry_date`, `entity_a_end_date`, `entity_a_geometry`.
- For `entity_b`: similar names with a `b_` prefix.

This results in each match having full metadata for both entities involved.

5. **Reorder and Export**

The final DataFrame `df_matches` is reordered for clarity and exported to `duplicate_entity_expectation.csv`. This CSV provides a comprehensive view of suspected duplicate geometries, enriched with entity metadata and geometry WKT for downstream analysis or visualisation.

In [2]:
```
# Load expectations table
url = "https://datasette.planning.data.gov.uk/digital-land/expectation.csv?_stre
df = pd.read_csv(url)
df = df[df["operation"] == "duplicate_geometry_check"]

# Parse 'details' column
def parse_details(val):
    try:
        return ast.literal_eval(val)
    except Exception:
        return {}

df["details_parsed"] = df["details"].apply(parse_details)

# Extract match records
records = []
for _, row in df.iterrows():
    dataset = row["dataset"]
    operation = row["operation"]
    details = row["details_parsed"]

    # Complete matches
    for match in details.get("complete_matches", []):
        records.append({
            "dataset": dataset,
            "operation": operation,
            "message": "complete_match",
            "entity_a": match.get("entity_a"),
            "organisation_entity_a": match.get("organisation_entity_a"),
            "entity_b": match.get("entity_b"),
            "organisation_entity_b": match.get("organisation_entity_b"),
        })

    # Single matches
    for match in details.get("single_matches", []):
        records.append({
            "dataset": dataset,
            "operation": operation,
            "message": "single_match",
            "entity_a": match.get("entity_a"),
            "organisation_entity_a": match.get("organisation_entity_a"),
            "entity_b": match.get("entity_b"),
            "organisation_entity_b": match.get("organisation_entity_b"),
        })

df_matches = pd.DataFrame(records)

# Load entity tables
url_map = {
    "conservation-area": "https://datasette.planning.data.gov.uk/conservation-ar
    "article-4-direction-area": "https://datasette.planning.data.gov.uk/article-
    "listed-building-outline": "https://datasette.planning.data.gov.uk/listed-bu
    "tree-preservation-zone": "https://datasette.planning.data.gov.uk/tree-prese
```

```
    "tree": "https://datasette.planning.data.gov.uk/tree/entity.csv?_stream=on",
}

columns_to_keep = ["entity", "dataset", "end_date", "entry_date", "geometry", "n

entity_tables = {}
for dataset_name, entity_url in url_map.items():
    df_entity = pd.read_csv(entity_url)
    df_entity["dataset"] = dataset_name
    entity_tables[dataset_name] = df_entity[columns_to_keep].copy()

# Combine all entity tables
df_entities = pd.concat(entity_tables.values(), ignore_index=True)

# Merge entity_a metadata
df_matches = df_matches.merge(
    df_entities,
    how="left",
    left_on=["dataset", "entity_a"],
    right_on=["dataset", "entity"]
).rename(columns={
    "end_date": "entity_a_end_date",
    "entry_date": "entity_a_entry_date",
    "geometry": "entity_a_geometry",
    "name": "entity_a_name",
    "organisation_entity": "entity_a_organisation"
}).drop(columns=["entity"])

# Merge entity_b metadata
df_matches = df_matches.merge(
    df_entities,
    how="left",
    left_on=["dataset", "entity_b"],
    right_on=["dataset", "entity"]
).rename(columns={
    "end_date": "entity_b_end_date",
    "entry_date": "entity_b_entry_date",
    "geometry": "entity_b_geometry",
    "name": "entity_b_name",
    "organisation_entity": "entity_b_organisation"
}).drop(columns=["entity"])

# Reorder columns
ordered_cols = [
    "dataset", "operation", "message",
    "entity_a", "entity_a_name", "entity_a_organisation", "entity_a_entry_date",
    "entity_b", "entity_b_name", "entity_b_organisation", "entity_b_entry_date",
]
df_matches = df_matches[ordered_cols]

# Export
df_matches.to_csv("duplicate_entity_expectation.csv", index=False)
```

Duplicate Geometry Summary and Visualisation

This script analyzes the parsed `duplicate_geometry_check` expectations data and generates a summary CSV and bar chart to help understand the scale of duplication issues in each dataset.

---

1. **Extract Key Stats from Parsed Details**

A helper function `extract_stats()` is defined to pull structured metrics from each row's `details_parsed` dictionary:

- `"actual"` and `"expected"`: general stats from the check.
- `"complete_match_count"`: number of detected complete geometry duplicates.
- `"single_match_count"`: number of single matches.
- `"complete_matches"` and `"single_matches"`: the raw match lists (kept for inspection).

These stats are then combined with the original `dataset`, `message`, and `severity` columns into a new DataFrame called `stats_df`.

---

2. **Generate and Export Summary Table**

The `complete_matches` and `single_matches` raw lists are removed, and the resulting summary DataFrame is exported to `duplicate_geometry_summary.csv`. This table includes:

- Dataset name
- Severity level
- Message
- Actual vs expected counts
- Total counts of complete and single matches

The data is sorted in descending order of `complete_match_count`.

---

3. **Visualise Duplicate Match Counts**

A stacked bar chart is created using matplotlib to compare the volume of `complete_match_count` and `single_match_count` per dataset. The key steps include:

- Setting the dataset as the index.
- Plotting match counts as a stacked bar chart using a categorical colormap.
- Applying titles, axis labels, and rotation for clarity.

The resulting plot is saved as `duplicate_geometry_plot.png`.

This visualisation helps prioritise which datasets are most affected by duplicate geometries, aiding data stewards in targeting cleanup efforts.

In [3]:
```
# Extract key stats
def extract_stats(d):
    return pd.Series({
        "actual": d.get("actual"),
        "expected": d.get("expected"),
        "complete_match_count": len(d.get("complete_matches", [])) if isinstance
```

```
        "single_match_count": len(d.get("single_matches", [])) if isinstance(d.g
        "complete_matches": d.get("complete_matches", []),
        "single_matches": d.get("single_matches", [])
    })

stats_df = pd.concat([df[["dataset", "message", "severity"]], df["details_parsed
stats_df = stats_df.sort_values(by="complete_match_count", ascending=False).rese

# Export summary CSV
summary_path = "duplicate_geometry_summary.csv"
stats_df.drop(columns=["complete_matches", "single_matches"]).to_csv(summary_pat

# Bar chart of match counts
plt.figure(figsize=(10, 6))
stats_df.set_index("dataset")[["complete_match_count", "single_match_count"]].pl
    kind="bar", stacked=True, colormap="tab20", figsize=(12, 6)
)
plt.title("Duplicate Geometry Match Counts by Dataset")
plt.xlabel("Dataset")
plt.ylabel("Number of Duplicates")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()

plt.savefig("duplicate_geometry_plot.png")
plt.close()
```

```
<Figure size 1000x600 with 0 Axes>
```

Preview Duplicate Geometries on Interactive Map

This function, `preview_duplicate_geometries`, visualises pairs of geometries (entity A vs entity B) flagged as potential duplicates using Folium's interactive map capabilities via GeoPandas' `.explore()` method. It is particularly useful for visually validating geometry duplication issues found in spatial planning datasets.

Function Parameters

- `df` — A pandas DataFrame containing the cleaned duplicate match data, including WKT geometries for `entity_a_geometry` and `entity_b_geometry`.
- `n` — Number of rows (i.e., duplicate match records) to preview. Default is 5.

Workflow

1. **Subset the Data**: The function selects the first `n` rows of the input DataFrame.
2. **Extract and Convert Geometry Records**: For each selected row:
    - If `entity_a_geometry` exists, a record is created with:
        - Entity ID
        - Organisation name (if available)
        - Label `"A"`
        - Parsed geometry using `shapely.wkt.loads()`
    - If `entity_b_geometry` exists, a similar record is created with label `"B"`.

3. **Create GeoDataFrame**: All geometry records are compiled into a GeoDataFrame (`gdf`) with CRS set to WGS84 (`EPSG:4326`).

4. **Render Interactive Map**: The `.explore()` method is used to generate an interactive Leaflet map:

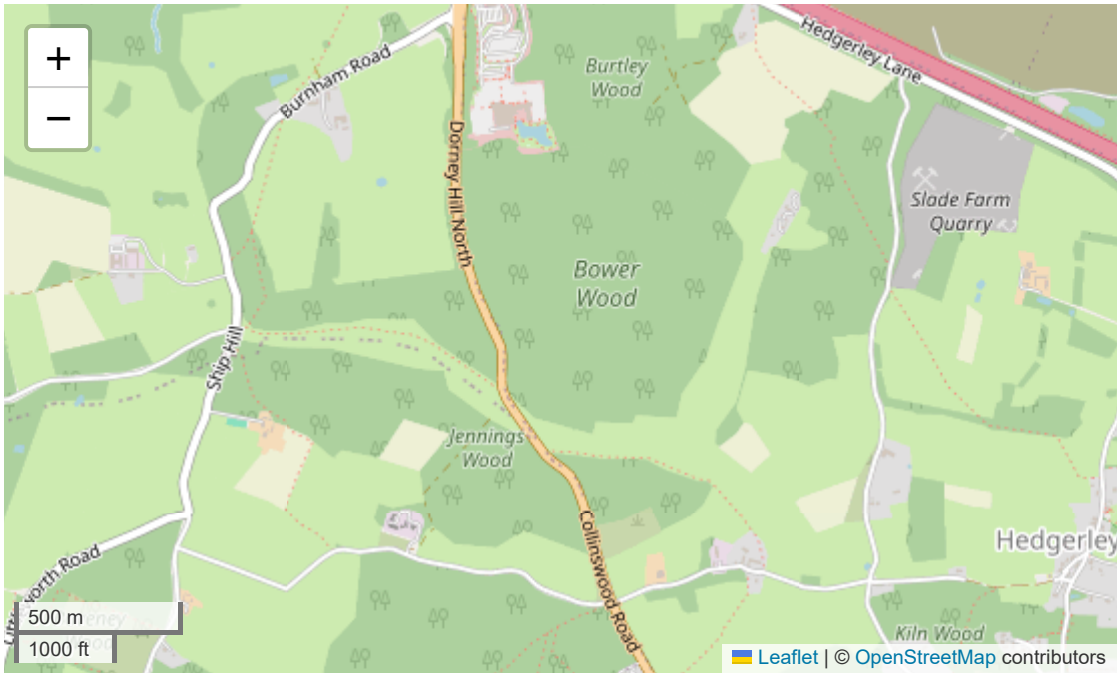
    - Entity A geometries are colored **red**, Entity B geometries **blue**.
    - Tooltips show entity ID, organisation, and label.
    - The map is returned and displayed inline in a Jupyter notebook environment.

Purpose

This visual tool helps users quickly and intuitively inspect spatial overlaps and verify if flagged geometry matches (either "complete" or "single") appear visually plausible as duplicates.

In [4]:
```
def preview_duplicate_geometries(df, n=5):
    """
    Display map of entity_a and entity_b geometries for the first n rows in a Ju

    Parameters:
    - df: DataFrame with geometry columns 'entity_a_geometry' and 'entity_b_geom
    - n: Number of rows to preview

    Returns:
    - folium.Map object rendered in notebook via .explore()
    """
    rows = df.head(n)
    records = []

    for _, row in rows.iterrows():
        if pd.notna(row["entity_a_geometry"]):
            records.append({
                "entity": row["entity_a"],
                "organisation": row.get("entity_a_organisation") or row.get("org
                "label": "A",
                "geometry": wkt.loads(row["entity_a_geometry"])
            })
        if pd.notna(row["entity_b_geometry"]):
            records.append({
                "entity": row["entity_b"],
                "organisation": row.get("entity_b_organisation") or row.get("org
                "label": "B",
                "geometry": wkt.loads(row["entity_b_geometry"])
            })

    gdf = gpd.GeoDataFrame(records, crs="EPSG:4326")

    # Pass literal colors based on label
    return gdf.explore(
        color=gdf["label"].map({"A": "red", "B": "blue"}),
        tooltip=["entity", "organisation", "label"],
        legend=False
    )

preview_duplicate_geometries(df_matches, n=3)
```

Out[4]:


Burnham Road
Burtley Wood
Hedgerley Lane
Dorney Hill North
Slade Farm Quarry
Bower Wood
Ship Hill
Jennings Wood
Collinswood Road
Hedgerley
Kiln Wood
500 m
1000 ft
Leaflet | © OpenStreetMap contributors