import pandas as pd

import numpy as np

import datacompy

import xlsxwriter

from datetime import datetime

import os

import zipfile

from io import BytesIO

import logging

from typing import Dict, List, Optional, Tuple

try:

    from .profile\_utils import (

        generate\_distribution\_plot,

        generate\_frequency\_plot,

        generate\_comparison\_plot,

        generate\_comparison\_rows,

        calculate\_column\_stats

    )

except ImportError:

    # Fallback to direct import if relative import fails

    from profile\_utils import (

        generate\_distribution\_plot,

        generate\_frequency\_plot,

        generate\_comparison\_plot,

        generate\_comparison\_rows,

        calculate\_column\_stats

    )

logger = logging.getLogger(\_\_name\_\_)

def generate\_mismatch\_html(comparison):

    """Generate HTML for mismatches with proper error handling."""

    try:

        mismatches\_html = []

        # Handle source-only rows

        if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Source</h3>")

            mismatches\_html.append(comparison.df1\_unq\_rows.head(5).to\_html())

        # Handle target-only rows

        if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Target</h3>")

            mismatches\_html.append(comparison.df2\_unq\_rows.head(5).to\_html())

        if mismatches\_html:

            return "\n".join(mismatches\_html)

        return "<p>No mismatches found</p>"

    except Exception as e:

        logger.warning(f"Error generating mismatch HTML: {str(e)}")

        return f"<p>Error generating mismatch details: {str(e)}</p>"

def generate\_datacompy\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            join\_columns: List[str], mapping\_df: pd.DataFrame,

                            join\_mappings: Dict[str, str]) -> Tuple[BytesIO, BytesIO]:

    """Generate a DataCompy comparison report."""

    try:

        # Get excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        # Create mapping dictionary

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Get all columns that need to be processed

        source\_cols = list(valid\_columns.keys())

        target\_cols = [valid\_columns[src] for src in source\_cols]

        # Create unique column names for comparison

        source\_rename = {}

        target\_rename = {}

        final\_join\_columns = []

        # First, handle join columns to ensure they're unique

        for src\_col in join\_columns:

            tgt\_col = join\_mappings[src\_col]

            unique\_join\_name = f"join\_{src\_col}"

            source\_rename[src\_col] = unique\_join\_name

            target\_rename[tgt\_col] = unique\_join\_name

            final\_join\_columns.append(unique\_join\_name)

        # Then handle comparison columns

        for src\_col, tgt\_col in valid\_columns.items():

            if src\_col not in join\_columns:  # Skip if it's already a join column

                unique\_col\_name = f"compare\_{src\_col}"

                source\_rename[src\_col] = unique\_col\_name

                target\_rename[tgt\_col] = unique\_col\_name

        # Create copies of DataFrames with only needed columns

        source\_compare = source\_df[list(set(source\_cols + join\_columns))].copy()

        target\_compare = target\_df[list(set(target\_cols + [join\_mappings[src] for src in join\_columns]))].copy()

        # Rename columns to ensure uniqueness

        source\_compare.rename(columns=source\_rename, inplace=True)

        target\_compare.rename(columns=target\_rename, inplace=True)

        # Update join columns for the comparison

        join\_columns = final\_join\_columns

        logger.info(f"Source columns after renaming: {source\_compare.columns.tolist()}")

        logger.info(f"Target columns after renaming: {target\_compare.columns.tolist()}")

        logger.info(f"Join columns for comparison: {join\_columns}")

        # Create comparison object with proper configuration

        try:

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        except Exception as e:

            logger.warning(f"Initial comparison failed: {str(e)}, trying with string conversion")

            # Convert all columns to string type for comparison

            source\_compare = source\_compare.astype(str)

            target\_compare = target\_compare.astype(str)

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        # Generate Excel report

        excel\_output = BytesIO()

        with pd.ExcelWriter(excel\_output, engine='xlsxwriter') as writer:

            # Write summary

            summary\_data = {

                'Metric': [

                    'Rows in Source',

                    'Rows in Target',

                    'Rows in Common',

                    'Rows Only in Source',

                    'Rows Only in Target',

                    'Columns Match',

                    'All Row Values Match'

                ],

                'Value': [

                    len(source\_compare),

                    len(target\_compare),

                    comparison.intersect\_rows,

                    len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0,

                    len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0,

                    comparison.all\_columns\_match(),

                    comparison.matches()

                ]

            }

            pd.DataFrame(summary\_data).to\_excel(writer, sheet\_name='Summary', index=False)

            # Write column stats

            if hasattr(comparison, 'column\_stats'):

                if isinstance(comparison.column\_stats, pd.DataFrame):

                    comparison.column\_stats.to\_excel(writer, sheet\_name='Column Stats', index=True)

                elif isinstance(comparison.column\_stats, list):

                    pd.DataFrame(comparison.column\_stats).to\_excel(writer, sheet\_name='Column Stats', index=True)

                else:

                    pd.DataFrame({'Message': ['No column statistics available']}).to\_excel(

                        writer, sheet\_name='Column Stats', index=False)

            # Write sample mismatches with proper error handling

            try:

                mismatches\_df = pd.DataFrame()

                # Get source-only rows

                if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

                    source\_mismatches = comparison.df1\_unq\_rows.head(5).copy()

                    source\_mismatches['Match Type'] = 'Source Only'

                    mismatches\_df = pd.concat([mismatches\_df, source\_mismatches])

                # Get target-only rows

                if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

                    target\_mismatches = comparison.df2\_unq\_rows.head(5).copy()

                    target\_mismatches['Match Type'] = 'Target Only'

                    mismatches\_df = pd.concat([mismatches\_df, target\_mismatches])

                if not mismatches\_df.empty:

                    mismatches\_df.to\_excel(writer, sheet\_name='Sample Mismatches', index=True)

                else:

                    pd.DataFrame({'Status': ['No mismatches found']}).to\_excel(

                        writer, sheet\_name='Sample Mismatches', index=False)

            except Exception as e:

                logger.warning(f"Error generating sample mismatches: {str(e)}")

                pd.DataFrame({'Error': [f'Failed to generate sample mismatches: {str(e)}']}).to\_excel(

                    writer, sheet\_name='Sample Mismatches', index=False)

        # Generate HTML report

        html\_output = BytesIO()

        html\_report = f"""

        <!DOCTYPE html>

        <html lang="en">

        <head>

            <meta charset="UTF-8">

            <title>DataCompy Comparison Report</title>

            <style>

                body {{

                    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif;

                    margin: 20px;

                    line-height: 1.6;

                    background-color: #f8f9fa;

                }}

                table {{

                    width: 100%;

                    border-collapse: collapse;

                    margin-bottom: 20px;

                    background-color: #ffffff;

                    box-shadow: 0 1px 3px rgba(0,0,0,0.1);

                }}

                th, td {{

                    border: 1px solid #dee2e6;

                    padding: 12px;

                    text-align: left;

                }}

                th {{

                    background-color: #f8f9fa;

                    font-weight: 600;

                }}

                .pass {{

                    color: #28a745;

                    font-weight: 600;

                }}

                .fail {{

                    color: #dc3545;

                    font-weight: 600;

                }}

                .section {{

                    background-color: #ffffff;

                    padding: 25px;

                    margin-bottom: 30px;

                    border-radius: 8px;

                    box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                }}

                h1, h2 {{

                    color: #333;

                    margin-top: 0;

                }}

                h1 {{

                    font-size: 24px;

                    margin-bottom: 30px;

                    padding: 25px;

                    background-color: #ffffff;

                    border-radius: 8px;

                    box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                }}

                h2 {{

                    font-size: 20px;

                    margin-bottom: 15px;

                }}

                p {{

                    margin: 10px 0;

                }}

            </style>

        </head>

        <body>

            <h1>DataCompy Comparison Report</h1>

            <div class="section">

                <h2>Summary</h2>

                <p>Source rows: {len(source\_compare)}</p>

                <p>Target rows: {len(target\_compare)}</p>

                <p>Rows in common: {comparison.intersect\_rows}</p>

                <p>Rows only in source: {len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0}</p>

                <p>Rows only in target: {len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0}</p>

                <p>Columns match: <span class="{'pass' if comparison.all\_columns\_match() else 'fail'}">{comparison.all\_columns\_match()}</span></p>

                <p>All rows match: <span class="{'pass' if comparison.matches() else 'fail'}">{comparison.matches()}</span></p>

            </div>

            <div class="section">

                <h2>Column Statistics</h2>

                {

                    pd.DataFrame(comparison.column\_stats).to\_html()

                    if hasattr(comparison, 'column\_stats') and

                       (isinstance(comparison.column\_stats, pd.DataFrame) or

                        isinstance(comparison.column\_stats, list))

                    else '<p>No column statistics available</p>'

                }

            </div>

            <div class="section">

                <h2>Sample Mismatches</h2>

                {generate\_mismatch\_html(comparison)}

            </div>

        </body>

        </html>

        """

        html\_output.write(html\_report.encode('utf-8'))

        html\_output.seek(0)

        excel\_output.seek(0)

        return excel\_output, html\_output

    except Exception as e:

        logger.error(f"Error generating DataCompy report: {str(e)}")

        raise Exception(f"Failed to generate DataCompy report: {str(e)}")

def generate\_ydata\_profile(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         mapping\_df: pd.DataFrame) -> Tuple[BytesIO, BytesIO, BytesIO]:

    """Generate column-wise Y-Data Profiling reports."""

    try:

        # Create mapping dictionary

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Initialize output buffers

        source\_output = BytesIO()

        target\_output = BytesIO()

        comparison\_output = BytesIO()

        # Start HTML reports with headers and CSS

        html\_header = """

            <!DOCTYPE html>

            <html lang="en">

            <head>

                <meta charset="UTF-8">

                <title>{title}</title>

                <style>

                    body {{

                        font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;

                        margin: 20px;

                        line-height: 1.6;

                        background-color: #f8f9fa;

                    }}

                    .column-profile {{

                        margin-bottom: 40px;

                        padding: 20px;

                        border: 1px solid #ddd;

                        border-radius: 8px;

                        background-color: #ffffff;

                        box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                    }}

                    .stats-table {{

                        width: 100%;

                        border-collapse: collapse;

                        margin-top: 15px;

                        background-color: #ffffff;

                    }}

                    .stats-table th, .stats-table td {{

                        border: 1px solid #dee2e6;

                        padding: 12px;

                        text-align: left;

                    }}

                    .stats-table th {{

                        background-color: #f8f9fa;

                        font-weight: 600;

                    }}

                    .visualization {{

                        margin-top: 25px;

                        padding: 15px;

                        background-color: #ffffff;

                        border-radius: 6px;

                    }}

                    .chart {{

                        width: 100%;

                        max-width: 800px;

                        margin: 20px auto;

                    }}

                    .header {{

                        background-color: #ffffff;

                        padding: 25px;

                        margin-bottom: 30px;

                        border-radius: 8px;

                        box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                    }}

                    .summary {{

                        margin: 20px 0;

                        display: flex;

                        flex-wrap: wrap;

                        gap: 15px;

                    }}

                    .metric {{

                        display: inline-block;

                        padding: 15px 20px;

                        background: #ffffff;

                        border: 1px solid #dee2e6;

                        border-radius: 6px;

                        box-shadow: 0 1px 3px rgba(0,0,0,0.1);

                    }}

                    .diff {{

                        background-color: #fff3f3;

                        border-color: #ffcdd2;

                    }}

                    .match {{

                        background-color: #f1f8f1;

                        border-color: #c8e6c9;

                    }}

                    h1, h2, h3 {{

                        color: #333;

                        margin-top: 0;

                    }}

                    h1 {{

                        font-size: 24px;

                        margin-bottom: 10px;

                    }}

                    h2 {{

                        font-size: 20px;

                        margin-bottom: 15px;

                    }}

                    h3 {{

                        font-size: 18px;

                        margin-bottom: 10px;

                    }}

                </style>

            </head>

            <body>

            <div class="header">

                <h1>{title}</h1>

                <p>Generated on: {timestamp}</p>

            </div>

        """

        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        source\_html = [html\_header.format(title="Source Data Profile", timestamp=timestamp)]

        target\_html = [html\_header.format(title="Target Data Profile", timestamp=timestamp)]

        comparison\_html = [html\_header.format(title="Column Comparison Profile", timestamp=timestamp)]

        # Process each column pair

        for src\_col, tgt\_col in valid\_columns.items():

            logger.info(f"Processing column pair: {src\_col} -> {tgt\_col}")

            # Get column data

            source\_data = source\_df[src\_col]

            target\_data = target\_df[tgt\_col]

            # Calculate statistics

            source\_stats = calculate\_column\_stats(source\_data)

            target\_stats = calculate\_column\_stats(target\_data)

            # Generate HTML for source column

            source\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {src\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {source\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {source\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in source\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(source\_data) if pd.api.types.is\_numeric\_dtype(source\_data) else generate\_frequency\_plot(source\_data)}

                    </div>

                </div>

            """)

            # Generate HTML for target column

            target\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {target\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {target\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {target\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in target\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(target\_data) if pd.api.types.is\_numeric\_dtype(target\_data) else generate\_frequency\_plot(target\_data)}

                    </div>

                </div>

            """)

            # Generate comparison HTML

            comparison\_html.append(f"""

                <div class="comparison-profile">

                    <h2>Column Comparison: {src\_col} ↔ {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Source Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Target Type:</strong> {target\_data.dtype}

                        </div>

                    </div>

                    <table class="comparison-table">

                        <tr>

                            <th>Metric</th>

                            <th>Source</th>

                            <th>Target</th>

                            <th>Difference</th>

                        </tr>

                        {generate\_comparison\_rows(source\_stats, target\_stats)}

                    </table>

                    <div class="visualization">

                        <h3>Distribution Comparison</h3>

                        {generate\_comparison\_plot(source\_data, target\_data)}

                    </div>

                </div>

            """)

        # Close HTML documents properly

        html\_footer = """

            </body>

        </html>

        """

        # Write to output buffers with proper HTML structure

        source\_output.write(''.join(source\_html + [html\_footer]).encode('utf-8'))

        target\_output.write(''.join(target\_html + [html\_footer]).encode('utf-8'))

        comparison\_output.write(''.join(comparison\_html + [html\_footer]).encode('utf-8'))

        # Reset buffer positions

        source\_output.seek(0)

        target\_output.seek(0)

        comparison\_output.seek(0)

        return source\_output, target\_output, comparison\_output

    except Exception as e:

        logger.error(f"Error generating Y-Data profile: {str(e)}")

        raise Exception(f"Failed to generate Y-Data profile: {str(e)}")

def create\_consolidated\_report(datacompy\_excel: BytesIO,

                             datacompy\_html: BytesIO,

                             ydata\_source\_report: BytesIO,

                             ydata\_target\_report: BytesIO,

                             ydata\_comparison\_report: BytesIO,

                             regression\_report: BytesIO,

                             difference\_report: BytesIO) -> BytesIO:

    """Combine all reports into a single consolidated report."""

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # Add all reports to a single consolidated file

            zf.writestr(f'reports/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_excel.getvalue())

            zf.writestr(f'reports/datacompy\_report\_{timestamp}.html',

                       datacompy\_html.getvalue())

            zf.writestr(f'reports/ydata\_source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'reports/ydata\_target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'reports/ydata\_comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            zf.writestr(f'reports/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            zf.writestr(f'reports/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a summary README

            readme\_content = f"""

            Consolidated Data Comparison Report

            This ZIP file contains the following reports:

            1. datacompy\_report - Detailed comparison of datasets

            2. ydata\_profile - Column-wise data profiling

            3. regression\_report - Aggregation, count, and distinct value checks

            4. difference\_report - Side-by-side differences

            Generated on: {timestamp}

            """

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating consolidated report: {str(e)}")

        raise Exception(f"Failed to create consolidated report: {str(e)}")

def generate\_regression\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            mapping\_df: pd.DataFrame, dtype\_mapping: Dict[str, str]) -> BytesIO:

    """Generate Excel-based regression report with multiple tabs."""

    try:

        output = BytesIO()

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            workbook = writer.book

            # Create formats for PASS/FAIL cells

            pass\_format = workbook.add\_format({'bg\_color': '#90EE90'})  # Light green

            fail\_format = workbook.add\_format({'bg\_color': '#FFB6C6'})  # Light pink

            # Generate AggregationCheck tab

            \_generate\_aggregation\_check(source\_df, target\_df, mapping\_df, writer,

                                     pass\_format, fail\_format)

            # Generate CountCheck tab

            \_generate\_count\_check(source\_df, target\_df, writer, pass\_format, fail\_format)

            # Generate DistinctCheck tab

            \_generate\_distinct\_check(source\_df, target\_df, mapping\_df, writer,

                                  pass\_format, fail\_format)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating regression report: {str(e)}")

        raise Exception(f"Failed to generate regression report: {str(e)}")

def \_generate\_aggregation\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                              mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                              pass\_format: xlsxwriter.format.Format,

                              fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the AggregationCheck tab in the regression report."""

    # Get numeric columns

    numeric\_cols = source\_df.select\_dtypes(include=[np.number]).columns

    results = []

    for col in numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_sum = source\_df[col].sum()

            target\_sum = target\_df[target\_col].sum()

            match = np.isclose(source\_sum, target\_sum, rtol=1e-05)

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Sum': source\_sum,

                'Target Sum': target\_sum,

                'Result': 'PASS' if match else 'FAIL'

            })

    # Create DataFrame and write to Excel

    agg\_df = pd.DataFrame(results)

    if not agg\_df.empty:

        agg\_df.to\_excel(writer, sheet\_name='AggregationCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['AggregationCheck']

        result\_col = agg\_df.columns.get\_loc('Result')

        for row in range(len(agg\_df)):

            if agg\_df.iloc[row]['Result'] == 'PASS':

                worksheet.write(row + 1, result\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, result\_col, 'FAIL', fail\_format)

    else:

        pd.DataFrame({'Message': ['No numeric columns to compare']}).to\_excel(

            writer, sheet\_name='AggregationCheck', index=False)

def \_generate\_count\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         writer: pd.ExcelWriter,

                         pass\_format: xlsxwriter.format.Format,

                         fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the CountCheck tab in the regression report."""

    count\_data = {

        'Source File Name': source\_df.name if hasattr(source\_df, 'name') else 'Source',

        'Target File Name': target\_df.name if hasattr(target\_df, 'name') else 'Target',

        'Source Count': len(source\_df),

        'Target Count': len(target\_df),

        'Result': 'PASS' if len(source\_df) == len(target\_df) else 'FAIL'

    }

    count\_df = pd.DataFrame([count\_data])

    count\_df.to\_excel(writer, sheet\_name='CountCheck', index=False)

    # Apply conditional formatting

    worksheet = writer.sheets['CountCheck']

    result\_col = count\_df.columns.get\_loc('Result')

    if count\_data['Result'] == 'PASS':

        worksheet.write(1, result\_col, 'PASS', pass\_format)

    else:

        worksheet.write(1, result\_col, 'FAIL', fail\_format)

def \_generate\_distinct\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                           mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                           pass\_format: xlsxwriter.format.Format,

                           fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the DistinctCheck tab in the regression report."""

    # Get non-numeric columns

    non\_numeric\_cols = source\_df.select\_dtypes(exclude=[np.number]).columns

    results = []

    for col in non\_numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_distinct = set(source\_df[col].dropna().unique())

            target\_distinct = set(target\_df[target\_col].dropna().unique())

            source\_count = len(source\_distinct)

            target\_count = len(target\_distinct)

            count\_match = source\_count == target\_count

            values\_match = source\_distinct == target\_distinct

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Distinct Count': source\_count,

                'Target Distinct Count': target\_count,

                'Count Match': 'PASS' if count\_match else 'FAIL',

                'Values Match': 'PASS' if values\_match else 'FAIL',

                'Source Distinct Values': ', '.join(map(str, sorted(source\_distinct))),

                'Target Distinct Values': ', '.join(map(str, sorted(target\_distinct)))

            })

    # Create DataFrame and write to Excel

    distinct\_df = pd.DataFrame(results)

    if not distinct\_df.empty:

        distinct\_df.to\_excel(writer, sheet\_name='DistinctCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['DistinctCheck']

        count\_match\_col = distinct\_df.columns.get\_loc('Count Match')

        values\_match\_col = distinct\_df.columns.get\_loc('Values Match')

        for row in range(len(distinct\_df)):

            if distinct\_df.iloc[row]['Count Match'] == 'PASS':

                worksheet.write(row + 1, count\_match\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, count\_match\_col, 'FAIL', fail\_format)

            if distinct\_df.iloc[row]['Values Match'] == 'PASS':

                worksheet.write(row + 1, values\_match\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, values\_match\_col, 'FAIL', fail\_format)

    else:

        pd.DataFrame({'Message': ['No non-numeric columns to compare']}).to\_excel(

            writer, sheet\_name='DistinctCheck', index=False)

def generate\_difference\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                             mapping\_df: pd.DataFrame, join\_columns: List[str],

                             join\_mappings: Dict[str, str]) -> BytesIO:

    """Generate a detailed difference report comparing source and target datasets."""

    try:

        output = BytesIO()

        # Process mappings and join columns

        try:

            # Initialize combined mapping dictionary

            all\_mappings = {}

            # Initialize mappings

            column\_mappings = {}

            # Process mapping\_df

            try:

                # Ensure mapping\_df is a DataFrame

                if not isinstance(mapping\_df, pd.DataFrame):

                    if isinstance(mapping\_df, list):

                        if not mapping\_df:

                            raise ValueError("Empty mapping provided")

                        if isinstance(mapping\_df[0], dict):

                            # Convert list of dicts to DataFrame

                            mapping\_df = pd.DataFrame(mapping\_df)

                        else:

                            # Convert list of lists/tuples to DataFrame

                            mapping\_df = pd.DataFrame(mapping\_df, columns=['Source Column', 'Target Column'])

                    else:

                        raise ValueError("mapping\_df must be a DataFrame or list")

                # Ensure DataFrame has required columns

                required\_cols = {'Source Column', 'Target Column'}

                if not required\_cols.issubset(mapping\_df.columns):

                    # Try to standardize column names

                    col\_map = {}

                    for col in mapping\_df.columns:

                        col\_lower = col.lower().replace('\_', ' ')

                        if col\_lower in ['source', 'source column']:

                            col\_map[col] = 'Source Column'

                        elif col\_lower in ['target', 'target column']:

                            col\_map[col] = 'Target Column'

                        elif col\_lower in ['exclude', 'exclude from comparison']:

                            col\_map[col] = 'Exclude from Comparison'

                    if col\_map:

                        mapping\_df = mapping\_df.rename(columns=col\_map)

                    # Check again after renaming

                    if not required\_cols.issubset(mapping\_df.columns):

                        raise ValueError(f"Mapping must contain columns: {', '.join(required\_cols)}")

                # Create mapping dictionary, handling any NaN values

                column\_mappings = dict(zip(

                    mapping\_df['Source Column'].fillna(''),

                    mapping\_df['Target Column'].fillna('')

                ))

                # Remove empty mappings

                column\_mappings = {k: v for k, v in column\_mappings.items() if k and v}

                # Handle excluded columns

                excluded\_columns = set()

                if 'Exclude from Comparison' in mapping\_df.columns:

                    excluded\_mask = mapping\_df['Exclude from Comparison'].fillna(False)

                    excluded\_columns = set(mapping\_df[excluded\_mask]['Source Column'].dropna())

                # Remove excluded columns and empty mappings

                column\_mappings = {

                    k: v for k, v in column\_mappings.items()

                    if k and v and k not in excluded\_columns

                }

                if not column\_mappings:

                    raise ValueError("No valid mappings found after processing")

                logger.info(f"Processed mapping DataFrame shape: {mapping\_df.shape}")

                logger.info(f"Valid mappings count: {len(column\_mappings)}")

                logger.info(f"Excluded columns: {excluded\_columns}")

            except Exception as e:

                logger.error(f"Error processing mapping\_df: {str(e)}")

                if isinstance(mapping\_df, pd.DataFrame):

                    logger.error(f"Mapping DataFrame shape: {mapping\_df.shape}")

                    logger.error(f"Mapping DataFrame columns: {mapping\_df.columns.tolist()}")

                raise ValueError(f"Invalid mapping format: {str(e)}")

            # Validate and process join columns

            if not isinstance(join\_mappings, dict):

                raise ValueError("join\_mappings must be a dictionary")

            # Ensure all join columns have mappings

            missing\_joins = [col for col in join\_columns if col not in join\_mappings]

            if missing\_joins:

                raise ValueError(f"Missing mappings for join columns: {', '.join(missing\_joins)}")

            # Combine all mappings

            all\_mappings = {\*\*column\_mappings, \*\*join\_mappings}

            # Create valid columns mapping (excluding join columns)

            valid\_columns = {

                src: tgt for src, tgt in column\_mappings.items()

                if src not in join\_columns

            }

            # Validate mappings

            if not valid\_columns:

                raise ValueError("No valid non-join columns found for comparison")

            # Validate source DataFrame columns

            source\_cols = set(join\_columns) | set(valid\_columns.keys())

            missing\_source = source\_cols - set(source\_df.columns)

            if missing\_source:

                raise ValueError(f"Source columns not found: {', '.join(sorted(missing\_source))}")

            # Validate target DataFrame columns

            target\_cols = set(join\_mappings.values()) | set(valid\_columns.values())

            missing\_target = target\_cols - set(target\_df.columns)

            if missing\_target:

                raise ValueError(f"Target columns not found: {', '.join(sorted(missing\_target))}")

        except Exception as e:

            logger.error(f"Error processing column mappings: {str(e)}")

            raise Exception(f"Failed to process column mappings: {str(e)}")

        # Create copies of DataFrames with only needed columns

        source\_cols = list(source\_required)

        target\_cols = list(target\_required)

        source\_compare = source\_df[source\_cols].copy()

        target\_compare = target\_df[target\_cols].copy()

        # Rename target columns to match source for comparison

        reverse\_mapping = {v: k for k, v in valid\_columns.items()}

        reverse\_mapping.update({v: k for k, v in join\_mappings.items()})

        target\_compare.rename(columns=reverse\_mapping, inplace=True)

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            workbook = writer.book

            # Create formats

            header\_format = workbook.add\_format({

                'bold': True,

                'bg\_color': '#D3D3D3',

                'border': 1

            })

            diff\_format = workbook.add\_format({

                'bg\_color': '#FFB6C6',  # Light pink

                'border': 1

            })

            normal\_format = workbook.add\_format({

                'border': 1

            })

            # Merge datasets safely

            try:

                # Ensure join columns exist in both dataframes

                common\_join\_cols = [col for col in join\_columns if col in source\_compare.columns and col in target\_compare.columns]

                if not common\_join\_cols:

                    raise ValueError("No valid join columns found in both datasets")

                merged\_df = source\_compare.merge(

                    target\_compare,

                    on=common\_join\_cols,

                    how='outer',

                    suffixes=('\_source', '\_target'),

                    indicator=True

                )

                # Convert \_merge column to string type

                merged\_df['\_merge'] = merged\_df['\_merge'].astype(str)

                # Calculate differences for matching rows

                matching\_rows = merged\_df[merged\_df['\_merge'] == 'both'].copy()

                source\_cols = [c for c in matching\_rows.columns if c.endswith('\_source') and not c.startswith('join\_')]

                target\_cols = [c.replace('\_source', '\_target') for c in source\_cols]

                # Compare values and count differences

                differences\_mask = False

                for src\_col, tgt\_col in zip(source\_cols, target\_cols):

                    differences\_mask |= matching\_rows[src\_col].ne(matching\_rows[tgt\_col])

                rows\_with\_differences = len(matching\_rows[differences\_mask])

                # Generate summary

                summary\_data = {

                    'Metric': [

                        'Total Rows in Source',

                        'Total Rows in Target',

                        'Rows Only in Source',

                        'Rows Only in Target',

                        'Common Rows',

                        'Rows with Differences'

                    ],

                    'Count': [

                        len(source\_compare),

                        len(target\_compare),

                        len(merged\_df[merged\_df['\_merge'] == 'left\_only']),

                        len(merged\_df[merged\_df['\_merge'] == 'right\_only']),

                        len(merged\_df[merged\_df['\_merge'] == 'both']),

                        rows\_with\_differences

                    ]

                }

            except Exception as e:

                logger.error(f"Error during merge operation: {str(e)}")

                raise Exception(f"Failed to merge datasets: {str(e)}")

            # Write Summary sheet

            summary\_df = pd.DataFrame(summary\_data)

            summary\_df.to\_excel(writer, sheet\_name='Summary', index=False)

            # Format Summary sheet

            summary\_sheet = writer.sheets['Summary']

            for col\_num, value in enumerate(summary\_df.columns.values):

                summary\_sheet.write(0, col\_num, value, header\_format)

            # Write Source Only sheet

            source\_only = merged\_df[merged\_df['\_merge'] == 'left\_only']

            if not source\_only.empty:

                source\_cols = [c for c in source\_only.columns if not c.endswith('\_target')]

                source\_only[source\_cols].drop('\_merge', axis=1).to\_excel(

                    writer, sheet\_name='Source Only', index=False)

            # Write Target Only sheet

            target\_only = merged\_df[merged\_df['\_merge'] == 'right\_only']

            if not target\_only.empty:

                target\_cols = [c for c in target\_only.columns if not c.endswith('\_source')]

                target\_only[target\_cols].drop('\_merge', axis=1).to\_excel(

                    writer, sheet\_name='Target Only', index=False)

            # Write Differences sheet

            differences = merged\_df[merged\_df['\_merge'] == 'both'].copy()

            if not differences.empty:

                # Identify columns with differences

                source\_cols = [c for c in differences.columns if c.endswith('\_source') and

                             not c.startswith('join\_')]

                target\_cols = [c.replace('\_source', '\_target') for c in source\_cols]

                # Create side-by-side comparison

                comparison\_cols = []

                for src, tgt in zip(source\_cols, target\_cols):

                    base\_col = src.replace('\_source', '')

                    differences[f'{base\_col}\_diff'] = differences[src] != differences[tgt]

                    comparison\_cols.extend([src, tgt, f'{base\_col}\_diff'])

                # Add join columns at the start

                final\_cols = join\_columns + comparison\_cols

                differences[final\_cols].to\_excel(writer, sheet\_name='Differences', index=False)

                # Format Differences sheet

                diff\_sheet = writer.sheets['Differences']

                for col\_num, value in enumerate(final\_cols):

                    diff\_sheet.write(0, col\_num, value, header\_format)

                    if value.endswith('\_diff'):

                        # Highlight cells where differences exist

                        for row\_num in range(1, len(differences) + 1):

                            if differences.iloc[row\_num-1][value]:

                                diff\_sheet.write(row\_num, col\_num, 'True', diff\_format)

                            else:

                                diff\_sheet.write(row\_num, col\_num, 'False', normal\_format)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating difference report: {str(e)}")

        raise Exception(f"Failed to generate difference report: {str(e)}")

def create\_individual\_reports\_zip(datacompy\_excel: BytesIO,

                                datacompy\_html: BytesIO,

                                ydata\_source\_report: BytesIO,

                                ydata\_target\_report: BytesIO,

                                ydata\_comparison\_report: BytesIO,

                                regression\_report: BytesIO,

                                difference\_report: BytesIO) -> BytesIO:

    """Create a ZIP file containing individual reports in separate folders."""

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # DataCompy Reports

            zf.writestr(f'datacompy/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_excel.getvalue())

            zf.writestr(f'datacompy/datacompy\_report\_{timestamp}.html',

                       datacompy\_html.getvalue())

            # Y-Data Profiles

            zf.writestr(f'ydata\_profile/source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'ydata\_profile/target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'ydata\_profile/comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            # Regression Report

            zf.writestr(f'regression/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            # Difference Report

            zf.writestr(f'differences/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a README file

            readme\_content = f"""

            Data Comparison Reports

            1. datacompy/ - Contains detailed comparison report

            2. ydata\_profile/ - Contains column-wise data profiling

            3. regression/ - Contains aggregation, count, and distinct value checks

            4. differences/ - Contains side-by-side difference report

            Generated on: {timestamp}

            """

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating individual reports zip: {str(e)}")

        raise Exception(f"Failed to create individual reports zip: {str(e)}")