import pandas as pd

import numpy as np

import datacompy

from ydata\_profiling import ProfileReport

import xlsxwriter

from datetime import datetime

import os

import zipfile

from io import BytesIO

import logging

from typing import Dict, List, Optional, Tuple

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

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]) -> BytesIO:

    """

    Generate a DataCompy comparison report.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        join\_columns: List of source columns to join on

        mapping\_df: DataFrame containing column mapping information

        join\_mappings: Dictionary mapping source join columns to target join columns

    Returns:

        BytesIO object containing the report

    """

    try:

        # Get excluded columns

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

        # Create mapping dictionary from mapping\_df

        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

        }

        # Prepare DataFrames for comparison

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

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

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

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

        # Rename target columns to match source columns for comparison

        target\_compare.columns = source\_cols

        # Get the target join column names

        target\_join\_columns = [join\_mappings[src] for src in join\_columns]

        # Rename join columns in target DataFrame to match source

        join\_column\_mapping = dict(zip(target\_join\_columns, join\_columns))

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

        # Ensure datetime columns are properly converted

        for col in source\_compare.columns:

            if source\_compare[col].dtype == 'datetime64[ns]':

                target\_compare[col] = pd.to\_datetime(target\_compare[col], errors='coerce')

            elif target\_compare[col].dtype == 'datetime64[ns]':

                source\_compare[col] = pd.to\_datetime(source\_compare[col], errors='coerce')

        # Ensure all columns are strings to prevent type comparison issues

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

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

        # Create comparison object

        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.error(f"Error creating comparison object: {str(e)}")

            # Try alternative comparison with converted data types

            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 report

        output = BytesIO()

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

            # Write summary

            pd.DataFrame({

                '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,

                         comparison.df1\_unq\_rows, comparison.df2\_unq\_rows,

                         comparison.all\_columns\_match, comparison.matches()]

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

            # Write column stats

            if hasattr(comparison, 'column\_stats') and comparison.column\_stats is not None:

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

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

                else:

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

            # Write sample mismatches

            try:

                # Try the new method name first

                if hasattr(comparison, 'get\_sample\_mismatch'):

                    mismatches = comparison.get\_sample\_mismatch()

                # Fall back to old method name

                elif hasattr(comparison, 'sample\_mismatch'):

                    mismatches = comparison.sample\_mismatch()

                else:

                    mismatches = None

                if isinstance(mismatches, pd.DataFrame) and not mismatches.empty:

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

                elif isinstance(mismatches, list) and mismatches:

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

                else:

                    pd.DataFrame({'Message': ['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({'Message': ['Error generating sample mismatches']}).to\_excel(

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

        output.seek(0)

        return 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) -> BytesIO:

    """

    Generate Y-Data Profiling comparison report.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        mapping\_df: DataFrame containing column mapping information

    Returns:

        BytesIO object containing the report

    """

    try:

        # Create mapping dictionary from mapping\_df

        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

        }

        # Prepare DataFrames for comparison

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

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

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

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

        # Rename target columns to match source columns for comparison

        target\_compare.columns = source\_cols

        # Generate profiles

        source\_profile = ProfileReport(source\_compare, title="Source Data Profile")

        target\_profile = ProfileReport(target\_compare, title="Target Data Profile")

        # Compare profiles

        comparison\_report = source\_profile.compare(target\_profile)

        # Save to BytesIO

        output = BytesIO()

        comparison\_report.to\_file(output)

        output.seek(0)

        return 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 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.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        mapping\_df: DataFrame containing column mapping information

        dtype\_mapping: Dictionary mapping columns to their desired data types

    Returns:

        BytesIO object containing the report

    """

    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)

    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)

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)

    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)

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

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

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

    """

    Generate side-by-side difference report.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        join\_columns: List of source columns to join on

        mapping\_df: DataFrame containing column mapping information

        join\_mappings: Dictionary mapping source join columns to target join columns

    Returns:

        BytesIO object containing the report

    """

    try:

        output = BytesIO()

        # Create mapping dictionary from mapping\_df

        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

        }

        # Prepare DataFrames for comparison

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

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

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

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

        # Get the target join column names

        target\_join\_columns = [join\_mappings[src] for src in join\_columns]

        # Merge datasets

        merged = pd.merge(

            source\_compare, target\_compare,

            left\_on=join\_columns,

            right\_on=target\_join\_columns,

            how='outer',

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

            indicator=True

        )

        # Find differences

        differences = merged[merged['\_merge'] != 'both']

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

            if len(differences) > 0:

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

            else:

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

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

        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\_report: BytesIO,

                                ydata\_report: BytesIO,

                                regression\_report: BytesIO,

                                difference\_report: BytesIO) -> BytesIO:

    """

    Create a ZIP file containing individual reports in separate folders.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_report: Y-Data Profiling report as BytesIO

        regression\_report: Regression report as BytesIO

        difference\_report: Difference report as BytesIO

    Returns:

        BytesIO object containing the ZIP file with individual reports

    """

    try:

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

        output = BytesIO()

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

            # DataCompy Report

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

                       datacompy\_report.getvalue())

            # Y-Data Profile

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

                       ydata\_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 = """

Data Comparison Reports

1. datacompy/ - Contains detailed comparison report

2. ydata\_profile/ - Contains comprehensive data profiling

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

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

Generated on: {timestamp}

            """.format(timestamp=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)}")

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

                             ydata\_report: BytesIO,

                             regression\_report: BytesIO,

                             difference\_report: BytesIO) -> BytesIO:

    """

    Combine all reports into a single consolidated report.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_report: Y-Data Profiling report as BytesIO

        regression\_report: Regression report as BytesIO

        difference\_report: Difference report as BytesIO

    Returns:

        BytesIO object containing the consolidated ZIP file

    """

    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\_report.getvalue())

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

                       ydata\_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 = """

Consolidated Data Comparison Report

This ZIP file contains the following reports:

1. datacompy\_report - Detailed comparison of datasets

2. ydata\_profile - Comprehensive data profiling

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

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

Generated on: {timestamp}

            """.format(timestamp=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)}")