import pandas as pd

import numpy as np

import datacompy

from ydata\_profiling import ProfileReport, compare

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]) -> Tuple[BytesIO, 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

            # Create summary DataFrame

            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') 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:

                # Get mismatched rows

                df1\_unq = comparison.df1\_unq\_rows if hasattr(comparison, 'df1\_unq\_rows') else pd.DataFrame()

                df2\_unq = comparison.df2\_unq\_rows if hasattr(comparison, 'df2\_unq\_rows') else pd.DataFrame()

                # Create sample mismatches DataFrame

                if not df1\_unq.empty or not df2\_unq.empty:

                    mismatches = pd.concat([

                        df1\_unq.assign(Source='Source Only').head(5),

                        df2\_unq.assign(Source='Target Only').head(5)

                    ], ignore\_index=True)

                    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': [f'Error generating sample mismatches: {str(e)}']}).to\_excel(

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

        # Generate HTML report

        html\_output = BytesIO()

        html\_report = f"""

        <html>

        <head>

            <title>DataCompy Comparison Report</title>

            <style>

                body {{ font-family: Arial, sans-serif; margin: 20px; }}

                table {{ border-collapse: collapse; width: 100%; margin-bottom: 20px; }}

                th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}

                th {{ background-color: #f2f2f2; }}

                .pass {{ color: green; }}

                .fail {{ color: red; }}

                .section {{ margin-bottom: 30px; }}

            </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>

                {comparison.column\_stats.to\_html() if hasattr(comparison, 'column\_stats') and isinstance(comparison.column\_stats, pd.DataFrame) else '<p>No column statistics available</p>'}

            </div>

            <div class="section">

                <h2>Sample Mismatches</h2>

                {pd.concat([

                    comparison.df1\_unq\_rows.assign(Source='Source Only').head(5) if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty else pd.DataFrame(),

                    comparison.df2\_unq\_rows.assign(Source='Target Only').head(5) if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty else pd.DataFrame()

                ]).to\_html() if (hasattr(comparison, 'df1\_unq\_rows') and hasattr(comparison, 'df2\_unq\_rows')) else '<p>No mismatches found</p>'}

            </div>

        </body>

        </html>

        """

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

        html\_output.seek(0)

        output.seek(0)

        return 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 Y-Data Profiling reports including individual profiles and comparison.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        mapping\_df: DataFrame containing column mapping information

    Returns:

        Tuple of (source\_profile, target\_profile, comparison\_profile) as BytesIO objects

    """

    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

        # Convert problematic data types to string

        for col in source\_compare.columns:

            if source\_compare[col].dtype.name not in ['int64', 'float64', 'bool', 'datetime64[ns]', 'object']:

                source\_compare[col] = source\_compare[col].astype(str)

            if target\_compare[col].dtype.name not in ['int64', 'float64', 'bool', 'datetime64[ns]', 'object']:

                target\_compare[col] = target\_compare[col].astype(str)

        # Handle null values

        source\_compare = source\_compare.fillna(pd.NA)

        target\_compare = target\_compare.fillna(pd.NA)

        try:

            # Generate source profile with enhanced configuration

            source\_profile = ProfileReport(

                source\_compare,

                title="Source Data Profile",

                minimal=False,

                explorative=True,

                show\_variable\_description=True,

                pool\_size=0,  # Disable multiprocessing for stability

                correlations={

                    "auto": True,

                    "pearson": True,

                    "spearman": True,

                    "kendall": True,

                    "phi\_k": True,

                    "cramers": True,

                    "recoded": True

                },

                html={

                    'style': {'full\_width': True},

                    'minify\_html': False

                },

                interactions={

                    "continuous": True,

                    "targets": []  # Enable all possible interactions

                },

                samples=None,

                missing\_diagrams={

                    "bar": True,

                    "matrix": True,

                    "heatmap": True,

                    "dendrogram": True

                },

                duplicates={

                    "head": 10,

                    "report": True

                },

                plot={

                    "correlation": {

                        "cmap": "RdBu",

                        "bad": "#000000"

                    },

                    "missing": True,

                    "histogram": {

                        "bins": 50,

                        "bayesian\_blocks\_bins": True

                    },

                    "scatter": True,

                    "value\_counts": True,

                    "pie": True

                },

                vars={

                    "num": {

                        "quantiles": [0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95],

                        "chi\_squared\_threshold": 0.999,

                        "skewness\_threshold": 20,

                        "low\_categorical\_threshold": 5,

                        "check\_composition": True,

                        "histogram\_bins": 50,

                        "statistics": ["mean", "std", "variance", "kurtosis", "skewness", "sum", "mad", "min", "max", "zeros\_count", "zeros\_perc"]

                    },

                    "cat": {

                        "length": True,

                        "characters": True,

                        "words": True,

                        "cardinality\_threshold": 50,

                        "chi\_squared\_threshold": 0.999,

                        "coerce\_str\_to\_date": True,

                        "redact": False,

                        "histogram\_bins": 50,

                        "check\_composition": True,

                        "length\_stats": True

                    },

                    "bool": {

                        "imbalance\_threshold": 0.7,

                        "histogram\_bins": 50

                    },

                    "path": {

                        "active": True

                    },

                    "file": {

                        "active": True

                    },

                    "image": {

                        "active": True

                    },

                    "url": {

                        "active": True

                    }

                },

                descriptions={},

                report={

                    "precision": 10,

                    "show\_type": True,

                    "show\_description": True,

                    "show\_composition": True,

                    "show\_extreme": True,

                    "show\_missing": True,

                    "show\_correlation\_plot": True,

                    "show\_scatter\_matrix": True,

                    "show\_histogram": True,

                    "show\_value\_counts": True,

                    "show\_length": True,

                    "show\_imbalance": True

                }

            )

            # Generate target profile with the same enhanced configuration

            target\_profile = ProfileReport(

                target\_compare,

                title="Target Data Profile",

                minimal=False,

                explorative=True,

                show\_variable\_description=True,

                pool\_size=0,  # Disable multiprocessing for stability

                correlations={

                    "auto": True,

                    "pearson": True,

                    "spearman": True,

                    "kendall": True,

                    "phi\_k": True,

                    "cramers": True,

                    "recoded": True

                },

                html={

                    'style': {'full\_width': True},

                    'minify\_html': False

                },

                interactions={

                    "continuous": True,

                    "targets": []  # Enable all possible interactions

                },

                samples=None,

                missing\_diagrams={

                    "bar": True,

                    "matrix": True,

                    "heatmap": True,

                    "dendrogram": True

                },

                duplicates={

                    "head": 10,

                    "report": True

                },

                plot={

                    "correlation": {

                        "cmap": "RdBu",

                        "bad": "#000000"

                    },

                    "missing": True,

                    "histogram": {

                        "bins": 50,

                        "bayesian\_blocks\_bins": True

                    },

                    "scatter": True,

                    "value\_counts": True,

                    "pie": True

                },

                vars={

                    "num": {

                        "quantiles": [0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95],

                        "chi\_squared\_threshold": 0.999,

                        "skewness\_threshold": 20,

                        "low\_categorical\_threshold": 5,

                        "check\_composition": True,

                        "histogram\_bins": 50,

                        "statistics": ["mean", "std", "variance", "kurtosis", "skewness", "sum", "mad", "min", "max", "zeros\_count", "zeros\_perc"]

                    },

                    "cat": {

                        "length": True,

                        "characters": True,

                        "words": True,

                        "cardinality\_threshold": 50,

                        "chi\_squared\_threshold": 0.999,

                        "coerce\_str\_to\_date": True,

                        "redact": False,

                        "histogram\_bins": 50,

                        "check\_composition": True,

                        "length\_stats": True

                    },

                    "bool": {

                        "imbalance\_threshold": 0.7,

                        "histogram\_bins": 50

                    },

                    "path": {

                        "active": True

                    },

                    "file": {

                        "active": True

                    },

                    "image": {

                        "active": True

                    },

                    "url": {

                        "active": True

                    }

                },

                descriptions={},

                report={

                    "precision": 10,

                    "show\_type": True,

                    "show\_description": True,

                    "show\_composition": True,

                    "show\_extreme": True,

                    "show\_missing": True,

                    "show\_correlation\_plot": True,

                    "show\_scatter\_matrix": True,

                    "show\_histogram": True,

                    "show\_value\_counts": True,

                    "show\_length": True,

                    "show\_imbalance": True

                }

            )

            # Generate comparison profile

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

            # Create BytesIO objects for reports

            source\_output = BytesIO()

            target\_output = BytesIO()

            comparison\_output = BytesIO()

            try:

                # Generate complete HTML reports with all visualizations

                source\_profile.to\_file(source\_output, silent=True)

                target\_profile.to\_file(target\_output, silent=True)

                comparison\_profile.to\_file(comparison\_output, silent=True)

                # 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 profile reports: {str(e)}")

                raise Exception(f"Failed to generate complete profile reports: {str(e)}")

        except Exception as e:

            logger.error(f"Error in profile generation: {str(e)}")

            # Fallback to basic HTML reports

            source\_output = BytesIO()

            target\_output = BytesIO()

            comparison\_output = BytesIO()

            source\_report = f"""

            <html><head><title>Source Data Profile</title></head>

            <body><h1>Source Data Profile</h1>{source\_compare.describe().to\_html()}</body></html>

            """

            target\_report = f"""

            <html><head><title>Target Data Profile</title></head>

            <body><h1>Target Data Profile</h1>{target\_compare.describe().to\_html()}</body></html>

            """

            comparison\_report = f"""

            <html>

            <head><title>Data Comparison Report</title></head>

            <body>

            <h1>Data Comparison Report</h1>

            <h2>Source Data Summary</h2>{source\_compare.describe().to\_html()}

            <h2>Target Data Summary</h2>{target\_compare.describe().to\_html()}

            </body></html>

            """

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

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

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

            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 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 enhanced side-by-side difference report with highlighted differences.

    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

        )

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

            workbook = writer.book

            # Create formats

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

            header\_format = workbook.add\_format({

                'bold': True,

                'bg\_color': '#D3D3D3',

                'border': 1

            })

            # Process and write each category

            for category in ['left\_only', 'right\_only', 'both']:

                if category == 'left\_only':

                    sheet\_name = 'Source Only'

                    data = merged[merged['\_merge'] == 'left\_only']

                elif category == 'right\_only':

                    sheet\_name = 'Target Only'

                    data = merged[merged['\_merge'] == 'right\_only']

                else:

                    sheet\_name = 'Matching Records'

                    data = merged[merged['\_merge'] == 'both']

                if not data.empty:

                    # Write data to sheet

                    data.to\_excel(writer, sheet\_name=sheet\_name, index=False)

                    worksheet = writer.sheets[sheet\_name]

                    # Apply header format

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

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

                    # For matching records, highlight differences

                    if category == 'both':

                        for row\_idx in range(len(data)):

                            for src\_col in source\_cols:

                                if src\_col not in join\_columns:  # Skip join columns

                                    tgt\_col = valid\_columns[src\_col]

                                    src\_val = data.iloc[row\_idx][f"{src\_col}\_source"]

                                    tgt\_val = data.iloc[row\_idx][f"{src\_col}\_target"]

                                    # Compare values and highlight differences

                                    if pd.notna(src\_val) and pd.notna(tgt\_val):

                                        if str(src\_val) != str(tgt\_val):

                                            col\_idx\_src = data.columns.get\_loc(f"{src\_col}\_source")

                                            col\_idx\_tgt = data.columns.get\_loc(f"{src\_col}\_target")

                                            worksheet.write(row\_idx + 1, col\_idx\_src, src\_val, diff\_format)

                                            worksheet.write(row\_idx + 1, col\_idx\_tgt, tgt\_val, diff\_format)

                    # Adjust column widths

                    for idx, col in enumerate(data.columns):

                        max\_length = max(

                            data[col].astype(str).apply(len).max(),

                            len(str(col))

                        )

                        worksheet.set\_column(idx, idx, max\_length + 2)

        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.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_source\_report: Source data profile as BytesIO

        ydata\_target\_report: Target data profile as BytesIO

        ydata\_comparison\_report: Profile comparison 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 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 = """

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\_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.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_source\_report: Source data profile as BytesIO

        ydata\_target\_report: Target data profile as BytesIO

        ydata\_comparison\_report: Profile comparison 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

            # DataCompy Reports

            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 = """

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)}")