"""Module for generating various comparison reports"""

import pandas as pd

import numpy as np

from typing import Dict, List, Tuple, Optional

import os

from datetime import datetime

from ydata\_profiling import ProfileReport

import datacompy

from openpyxl.styles import PatternFill, Font

from openpyxl.utils import get\_column\_letter

from config import EXCEL\_STYLES, REPORTS\_DIR

class ReportGenerator:

    @staticmethod

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

                                 column\_mapping: Dict[str, str], output\_path: str) -> None:

        """Generate regression report with multiple checks"""

        writer = pd.ExcelWriter(output\_path, engine='openpyxl')

        # Generate Aggregation Check

        ReportGenerator.\_generate\_aggregation\_check(source\_df, target\_df, column\_mapping, writer)

        # Generate Count Check

        ReportGenerator.\_generate\_count\_check(source\_df, target\_df, writer)

        # Generate Distinct Check

        ReportGenerator.\_generate\_distinct\_check(source\_df, target\_df, column\_mapping, writer)

        writer.save()

    @staticmethod

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

                                  column\_mapping: Dict[str, str], writer: pd.ExcelWriter) -> None:

        """Generate aggregation check for numeric columns"""

        agg\_data = []

        for source\_col, target\_col in column\_mapping.items():

            # Check if columns are numeric

            if pd.api.types.is\_numeric\_dtype(source\_df[source\_col]) and pd.api.types.is\_numeric\_dtype(target\_df[target\_col]):

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

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

                result = 'PASS' if np.isclose(source\_sum, target\_sum, rtol=1e-05) else 'FAIL'

                agg\_data.append({

                    'Source Column': source\_col,

                    'Target Column': target\_col,

                    'Source Sum': source\_sum,

                    'Target Sum': target\_sum,

                    'Result': result

                })

        if agg\_data:

            df\_agg = pd.DataFrame(agg\_data)

            df\_agg.to\_excel(writer, sheet\_name='AggregationCheck', index=False)

            # Apply conditional formatting

            worksheet = writer.sheets['AggregationCheck']

            for idx, row in enumerate(df\_agg['Result'], start=2):  # start=2 to skip header

                cell = worksheet[f'E{idx}']

                style = EXCEL\_STYLES['PASS'] if row == 'PASS' else EXCEL\_STYLES['FAIL']

                cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

                cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                            writer: pd.ExcelWriter) -> None:

        """Generate count check report"""

        count\_data = {

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

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

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

        }

        df\_count = pd.DataFrame([count\_data])

        df\_count.to\_excel(writer, sheet\_name='CountCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['CountCheck']

        cell = worksheet['C2']  # Result column, first data row

        style = EXCEL\_STYLES['PASS'] if count\_data['Result'] == 'PASS' else EXCEL\_STYLES['FAIL']

        cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

        cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                               column\_mapping: Dict[str, str], writer: pd.ExcelWriter) -> None:

        """Generate distinct value check for non-numeric columns"""

        distinct\_data = []

        for source\_col, target\_col in column\_mapping.items():

            # Check if columns are non-numeric

            if not pd.api.types.is\_numeric\_dtype(source\_df[source\_col]):

                source\_distinct = source\_df[source\_col].nunique()

                target\_distinct = target\_df[target\_col].nunique()

                source\_values = set(source\_df[source\_col].dropna().unique())

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

                values\_match = source\_values == target\_values

                count\_match = source\_distinct == target\_distinct

                result = 'PASS' if values\_match and count\_match else 'FAIL'

                distinct\_data.append({

                    'Source Column': source\_col,

                    'Target Column': target\_col,

                    'Source Distinct Count': source\_distinct,

                    'Target Distinct Count': target\_distinct,

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

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

                    'Result': result

                })

        if distinct\_data:

            df\_distinct = pd.DataFrame(distinct\_data)

            df\_distinct.to\_excel(writer, sheet\_name='DistinctCheck', index=False)

            # Apply conditional formatting

            worksheet = writer.sheets['DistinctCheck']

            for idx, row in enumerate(df\_distinct['Result'], start=2):  # start=2 to skip header

                cell = worksheet[f'G{idx}']

                style = EXCEL\_STYLES['PASS'] if row == 'PASS' else EXCEL\_STYLES['FAIL']

                cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

                cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                                   column\_mapping: Dict[str, str], join\_keys: List[Tuple[str, str]],

                                   output\_path: str) -> None:

        """Generate side by side comparison report"""

        # Prepare DataFrames for comparison

        source\_cols = list(column\_mapping.keys())

        target\_cols = [column\_mapping[col] for col in source\_cols]

        # Merge DataFrames based on join keys

        if join\_keys:

            source\_join\_cols = [key[0] for key in join\_keys]

            target\_join\_cols = [key[1] for key in join\_keys]

            merged\_df = pd.merge(

                source\_df[source\_cols],

                target\_df[target\_cols],

                left\_on=source\_join\_cols,

                right\_on=target\_join\_cols,

                how='outer',

                indicator=True

            )

            # Find differences

            diff\_mask = merged\_df['\_merge'] != 'both'

            diff\_df = merged\_df[diff\_mask].copy()

            if not diff\_df.empty:

                # Rename columns to show source/target

                diff\_df.columns = [f'Source\_{col}' if col in source\_cols else f'Target\_{col}'

                                 for col in diff\_df.columns]

                # Save to Excel

                diff\_df.to\_excel(output\_path, index=False)

                return True

            else:

                # Create empty DataFrame with message

                pd.DataFrame({'Message': ['No differences found']}).to\_excel(output\_path, index=False)

                return False

        else:

            # If no join keys, compare row by row

            comparison\_df = pd.DataFrame()

            for s\_col, t\_col in column\_mapping.items():

                comparison\_df[f'Source\_{s\_col}'] = source\_df[s\_col]

                comparison\_df[f'Target\_{t\_col}'] = target\_df[t\_col]

            comparison\_df.to\_excel(output\_path, index=False)

            return True

    @staticmethod

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

                               timestamp: str) -> Dict[str, str]:

        """Generate Y-Data profiling reports"""

        reports = {}

        # Create reports directory

        os.makedirs(REPORTS\_DIR, exist\_ok=True)

        # Generate source profile

        source\_profile = ProfileReport(

            source\_df,

            title="Source Data Profile Report",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        source\_path = os.path.join(REPORTS\_DIR, f"SourceProfile\_{timestamp}.html")

        source\_profile.to\_file(source\_path)

        reports['source'] = source\_path

        # Generate target profile

        target\_profile = ProfileReport(

            target\_df,

            title="Target Data Profile Report",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        target\_path = os.path.join(REPORTS\_DIR, f"TargetProfile\_{timestamp}.html")

        target\_profile.to\_file(target\_path)

        reports['target'] = target\_path

        # Generate comparison profile

        comparison\_df = pd.DataFrame()

        for col in source\_df.columns:

            comparison\_df[f'Source\_{col}'] = source\_df[col]

        for col in target\_df.columns:

            comparison\_df[f'Target\_{col}'] = target\_df[col]

        comparison\_profile = ProfileReport(

            comparison\_df,

            title="Source vs Target Comparison Profile",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        comparison\_path = os.path.join(REPORTS\_DIR, f"ComparisonProfile\_{timestamp}.html")

        comparison\_profile.to\_file(comparison\_path)

        reports['comparison'] = comparison\_path

        return reports

    @staticmethod

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

                                column\_mapping: Dict[str, str], join\_keys: List[Tuple[str, str]],

                                output\_path: str) -> None:

        """Generate DataCompy comparison report"""

        # Prepare DataFrames with mapped columns

        source\_cols = list(column\_mapping.keys())

        target\_cols = [column\_mapping[col] for col in source\_cols]

        if join\_keys:

            source\_join\_cols = [key[0] for key in join\_keys]

            target\_join\_cols = [key[1] for key in join\_keys]

        else:

            # If no join keys, use index

            source\_df = source\_df.copy()

            target\_df = target\_df.copy()

            source\_df['\_index'] = range(len(source\_df))

            target\_df['\_index'] = range(len(target\_df))

            source\_join\_cols = ['\_index']

            target\_join\_cols = ['\_index']

        try:

            # Prepare DataFrames with mapped columns

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

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

            # Convert all columns to string type to avoid type comparison issues

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

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

            # Rename target columns to match source for proper comparison

            target\_compare\_df.columns = source\_cols

            # Ensure join keys exist in both DataFrames and prepare them

            if join\_keys:

                valid\_join\_keys = []

                for key in join\_keys:

                    source\_key = key[0]

                    if source\_key in source\_compare\_df.columns:

                        # Ensure join columns have matching data types and no nulls

                        source\_compare\_df[source\_key] = source\_compare\_df[source\_key].fillna('').astype(str)

                        target\_compare\_df[source\_key] = target\_compare\_df[source\_key].fillna('').astype(str)

                        valid\_join\_keys.append(source\_key)

                if not valid\_join\_keys:

                    raise ValueError("No valid join keys found")

                try:

                    # Create DataCompy comparison with valid join keys

                    comparison = datacompy.Compare(

                        source\_compare\_df,

                        target\_compare\_df,

                        join\_columns=valid\_join\_keys,

                        df1\_name='Source',

                        df2\_name='Target'

                    )

                except Exception as e:

                    raise ValueError(f"Failed to create comparison with join keys: {str(e)}")

            else:

                # If no join keys specified, use index-based comparison

                source\_compare\_df['\_index'] = range(len(source\_compare\_df))

                target\_compare\_df['\_index'] = range(len(target\_compare\_df))

                comparison = datacompy.Compare(

                    source\_compare\_df,

                    target\_compare\_df,

                    join\_columns=['\_index'],

                    df1\_name='Source',

                    df2\_name='Target'

                )

            # Verify comparison object was created successfully

            if not comparison or not hasattr(comparison, 'report'):

                raise ValueError("Failed to create valid comparison object")

        except Exception as e:

            raise Exception(f"Failed to initialize comparison: {str(e)}")

        try:

            # Generate report content with enhanced error handling

            try:

                summary = comparison.report() if hasattr(comparison, 'report') and callable(comparison.report) else "Summary not available"

            except Exception as e:

                summary = f"Error generating summary: {str(e)}"

            try:

                matching\_rows = (comparison.count\_matching\_rows()

                               if hasattr(comparison, 'count\_matching\_rows') and callable(comparison.count\_matching\_rows)

                               else 0)

            except Exception as e:

                matching\_rows = 0

            try:

                total\_rows = len(source\_compare\_df)

                match\_percentage = (matching\_rows / total\_rows \* 100) if total\_rows > 0 else 0

                summary = f"""

                Comparison Summary:

                - Total Rows: {total\_rows}

                - Matching Rows: {matching\_rows}

                - Match Percentage: {match\_percentage:.2f}%

                {summary}

                """

            except Exception as e:

                summary += f"\nError calculating statistics: {str(e)}"

            # Handle column statistics with type checking

            try:

                def safe\_get\_stats():

                    if not hasattr(comparison, 'column\_stats'):

                        return pd.DataFrame()

                    stats = comparison.column\_stats

                    if isinstance(stats, bool) or stats is None:

                        return pd.DataFrame()

                    if isinstance(stats, pd.DataFrame):

                        return stats

                    try:

                        if isinstance(stats, (list, dict)):

                            return pd.DataFrame(stats)

                        return pd.DataFrame()

                    except:

                        return pd.DataFrame()

                col\_stats = safe\_get\_stats()

                if not col\_stats.empty:

                    try:

                        col\_stats\_html = col\_stats.to\_html()

                    except:

                        col\_stats\_html = '<p>Error formatting column statistics.</p>'

                else:

                    col\_stats\_html = '<p>No column statistics available.</p>'

            except Exception as e:

                col\_stats\_html = f'<p>Error generating column statistics: {str(e)}</p>'

            # Handle unique records with type checking

            try:

                # Safely get unique records with type checking

                def safe\_get\_df(attr\_name):

                    if hasattr(comparison, attr\_name):

                        attr\_value = getattr(comparison, attr\_name)

                        if isinstance(attr\_value, pd.DataFrame):

                            return attr\_value

                        elif isinstance(attr\_value, bool):

                            return pd.DataFrame()  # Return empty DataFrame if boolean

                        elif attr\_value is None:

                            return pd.DataFrame()

                        else:

                            try:

                                return pd.DataFrame(attr\_value)

                            except:

                                return pd.DataFrame()

                    return pd.DataFrame()

                df1\_unq = safe\_get\_df('df1\_unq\_rows')

                df2\_unq = safe\_get\_df('df2\_unq\_rows')

                intersect = safe\_get\_df('intersect\_rows')

                # Generate HTML with safety checks

                def safe\_to\_html(df, empty\_message):

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

                        try:

                            return df.to\_html()

                        except:

                            return f'<p>Error converting data to HTML</p>'

                    return f'<p>{empty\_message}</p>'

                df1\_html = safe\_to\_html(df1\_unq, 'No records unique to source')

                df2\_html = safe\_to\_html(df2\_unq, 'No records unique to target')

                intersect\_html = safe\_to\_html(intersect, 'No records with differences')

            except Exception as e:

                df1\_html = f'<p>Error processing source unique records: {str(e)}</p>'

                df2\_html = f'<p>Error processing target unique records: {str(e)}</p>'

                intersect\_html = f'<p>Error processing differing records: {str(e)}</p>'

            # Generate HTML report

            with open(output\_path, 'w') as f:

                f.write(f"""

                <html>

                <head>

                    <title>DataCompy Comparison Report</title>

                    <style>

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

                        .report {{ max-width: 1200px; margin: 0 auto; }}

                        .section {{ margin: 20px 0; padding: 20px; border: 1px solid #ddd; border-radius: 5px; }}

                        .match {{ color: green; }}

                        .mismatch {{ color: red; }}

                        table {{ border-collapse: collapse; width: 100%; }}

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

                        th {{ background-color: #f5f5f5; }}

                    </style>

                </head>

                <body>

                    <div class="report">

                        <h1>DataCompy Comparison Report</h1>

                        <div class="section">

                            <h2>Summary</h2>

                            <pre>{summary}</pre>

                        </div>

                        <div class="section">

                            <h2>Detailed Statistics</h2>

                            <h3>Matches</h3>

                            <div class="match">

                                <p>Number of rows match: {matching\_rows}</p>

                                <p>Number of columns match: {len(set(source\_cols))}</p>

                            </div>

                            <h3>Mismatches</h3>

                            <div class="mismatch">

                                <p>Rows only in Source: {len(df1\_unq) if isinstance(df1\_unq, pd.DataFrame) else 'N/A'}</p>

                                <p>Rows only in Target: {len(df2\_unq) if isinstance(df2\_unq, pd.DataFrame) else 'N/A'}</p>

                            </div>

                        </div>

                        <div class="section">

                            <h2>Column Statistics</h2>

                            {col\_stats\_html}

                        </div>

                        <div class="section">

                            <h2>Mismatched Records</h2>

                            <h3>Records only in Source:</h3>

                            {df1\_html}

                            <h3>Records only in Target:</h3>

                            {df2\_html}

                            <h3>Records with Different Values:</h3>

                            {intersect\_html}

                        </div>

                    </div>

                </body>

                </html>

                """)

        except Exception as e:

            # Create a simple error report if something goes wrong

            with open(output\_path, 'w') as f:

                f.write(f"""

                <html>

                <head><title>Comparison Error Report</title></head>

                <body>

                    <h1>Error Generating Comparison Report</h1>

                    <p>An error occurred while generating the comparison report: {str(e)}</p>

                </body>

                </html>

                """)

            raise Exception(f"Error generating DataCompy report: {str(e)}")