"""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

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\_comparison\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

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

                               output\_path: str) -> None:

        """Generate pandas-based 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']

        # 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

        # Initialize comparison results

        comparison\_results = {

            'total\_rows\_source': len(source\_compare\_df),

            'total\_rows\_target': len(target\_compare\_df),

            'matching\_rows': 0,

            'source\_only\_rows': 0,

            'target\_only\_rows': 0,

            'differing\_rows': 0,

            'source\_only\_df': pd.DataFrame(),

            'target\_only\_df': pd.DataFrame(),

            'diff\_df': pd.DataFrame()

        }

        try:

            # Prepare join keys with type checking

            valid\_join\_keys = []

            # Default to index-based comparison

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

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

            valid\_join\_keys = ['\_index']

            # Try to use provided join keys if available

            if join\_keys:

                try:

                    # Convert join\_keys to list if it's not already

                    join\_key\_list = list(join\_keys) if isinstance(join\_keys, (list, tuple)) else []

                    temp\_join\_keys = []

                    for key in join\_key\_list:

                        if isinstance(key, (list, tuple)) and len(key) == 2:

                            source\_key = str(key[0])

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

                                # Convert columns to string and handle 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)

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

                    # Only use custom join keys if valid ones were found

                    if temp\_join\_keys:

                        valid\_join\_keys = temp\_join\_keys

                        # Remove index columns if we're using custom join keys

                        if '\_index' in source\_compare\_df.columns:

                            source\_compare\_df.drop('\_index', axis=1, inplace=True)

                        if '\_index' in target\_compare\_df.columns:

                            target\_compare\_df.drop('\_index', axis=1, inplace=True)

                except Exception as e:

                    print(f"Warning: Error processing join keys, falling back to index-based comparison: {str(e)}")

            try:

                # Perform merge to find matching and non-matching rows

                merged\_df = pd.merge(

                    source\_compare\_df,

                    target\_compare\_df,

                    on=valid\_join\_keys,

                    how='outer',

                    indicator=True

                )

            except Exception as e:

                raise ValueError(f"Error performing merge operation: {str(e)}")

            # Calculate comparison statistics

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

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

            matching\_rows = merged\_df[merged\_df['\_merge'] == 'both']

            comparison\_results['source\_only\_rows'] = len(comparison\_results['source\_only\_df'])

            comparison\_results['target\_only\_rows'] = len(comparison\_results['target\_only\_df'])

            comparison\_results['matching\_rows'] = len(matching\_rows)

            # Calculate match percentage

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

            match\_percentage = (comparison\_results['matching\_rows'] / total\_rows \* 100) if total\_rows > 0 else 0

            # Generate summary

            summary = f"""

            Comparison Summary:

            - Total Rows in Source: {comparison\_results['total\_rows\_source']}

            - Total Rows in Target: {comparison\_results['total\_rows\_target']}

            - Matching Rows: {comparison\_results['matching\_rows']}

            - Source Only Rows: {comparison\_results['source\_only\_rows']}

            - Target Only Rows: {comparison\_results['target\_only\_rows']}

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

            """

            # Generate column statistics

            stats\_data = []

            for col in source\_compare\_df.columns:

                if col != '\_index':  # Skip index column if it exists

                    source\_stats = {

                        'Column': col,

                        'Source\_Count': len(source\_compare\_df[col].dropna()),

                        'Source\_Unique': source\_compare\_df[col].nunique(),

                        'Target\_Count': len(target\_compare\_df[col].dropna()),

                        'Target\_Unique': target\_compare\_df[col].nunique(),

                        'Match\_Status': 'Match' if source\_compare\_df[col].equals(target\_compare\_df[col]) else 'Mismatch'

                    }

                    stats\_data.append(source\_stats)

            col\_stats = pd.DataFrame(stats\_data)

            col\_stats\_html = col\_stats.to\_html(classes='table table-striped', index=False)

        except Exception as e:

            raise Exception(f"Error performing comparison: {str(e)}")

        # Generate HTML report

        try:

            # Generate HTML for source-only records

            source\_only\_html = comparison\_results['source\_only\_df'].to\_html(

                classes='table table-striped',

                index=False

            ) if not comparison\_results['source\_only\_df'].empty else '<p>No records unique to source</p>'

            # Generate HTML for target-only records

            target\_only\_html = comparison\_results['target\_only\_df'].to\_html(

                classes='table table-striped',

                index=False

            ) if not comparison\_results['target\_only\_df'].empty else '<p>No records unique to target</p>'

            # Write the HTML report

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

                f.write(f"""

                <!DOCTYPE html>

                <html lang="en">

                <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: {comparison\_results['matching\_rows']}</p>

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

                        </div>

                        <h3>Mismatches</h3>

                        <div class="mismatch">

                            <p>Rows only in Source: {comparison\_results['source\_only\_rows']}</p>

                            <p>Rows only in Target: {comparison\_results['target\_only\_rows']}</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>

                            {source\_only\_html}

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

                            {target\_only\_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)}")