import streamlit as st

import pandas as pd

from typing import Dict, List, Tuple, Optional, Any

import os

from datetime import datetime

import datacompy

from ydata\_profiling import ProfileReport

import streamlit.components.v1 as components

# Import our modules

from data\_reader import DataReader

from mapping\_manager import MappingManager

from report\_generator import ReportGenerator

from db\_connector import DatabaseConnector

from api\_fetcher import APIFetcher

from utils import log\_error, format\_timestamp

# Set page config

st.set\_page\_config(

page\_title="Data Comparison Framework",

page\_icon="📊",

layout="wide"

)

# Define source types

SOURCE\_TYPES = [

"CSV File",

"DAT File",

"SQL Server",

"Stored Procedure",

"Teradata",

"API",

"Parquet File",

"Zipped Flat Files"

]

def main():

"""Main application function"""

# Add modern header with enhanced styling

st.markdown(

"""

<div style='text-align: center; background-color: #f8f9fa; padding: 2rem;

border-radius: 10px; margin-bottom: 2rem; border: 1px solid #dee2e6;'>

<h1 style='color: #1f77b4; margin-bottom: 0.5rem; font-size: 2.5em;'>

Data Comparison Framework

</h1>

<p style='color: #6c757d; font-size: 1.1em; margin: 1rem 0;'>

Compare data between SQL Server databases and feed files with advanced mapping capabilities

</p>

<div style='display: flex; justify-content: center; gap: 1rem; margin-top: 1rem;'>

<div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

<span style='color: #1f77b4;'>✨ Automatic Column Mapping</span>

</div>

<div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

<span style='color: #1f77b4;'>🔍 Smart Join Detection</span>

</div>

<div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

<span style='color: #1f77b4;'>📊 Detailed Reports</span>

</div>

</div>

</div>

""",

unsafe\_allow\_html=True

)

# Add error handling for the entire app

try:

# Initialize session state variables

if 'source\_df' not in st.session\_state:

st.session\_state.source\_df = None

if 'target\_df' not in st.session\_state:

st.session\_state.target\_df = None

if 'column\_mapping' not in st.session\_state:

st.session\_state.column\_mapping = {}

if 'excluded\_columns' not in st.session\_state:

st.session\_state.excluded\_columns = []

if 'report\_paths' not in st.session\_state:

st.session\_state.report\_paths = {}

# Create two columns for source and target selection with enhanced styling

st.markdown(

"""

<div style='display: flex; gap: 2rem; margin: 1rem 0;'>

<div style='flex: 1; background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; border: 1px solid #dee2e6;'>

<h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Source Configuration</h3>

<p style='color: #6c757d; margin-bottom: 1rem;'>Select and configure your source data</p>

</div>

<div style='flex: 1; background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; border: 1px solid #dee2e6;'>

<h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Target Configuration</h3>

<p style='color: #6c757d; margin-bottom: 1rem;'>Select and configure your target data</p>

</div>

</div>

""",

unsafe\_allow\_html=True

)

col1, col2 = st.columns(2)

with col1:

source\_type = st.selectbox(

"Select Source Type",

SOURCE\_TYPES,

key="source\_type",

help="Choose the type of source data you want to compare"

)

source\_data = handle\_data\_source(source\_type, "source")

if source\_data is not None:

st.success(f"✅ Source data loaded successfully: {len(source\_data)} rows, {len(source\_data.columns)} columns")

with col2:

target\_type = st.selectbox(

"Select Target Type",

SOURCE\_TYPES,

key="target\_type",

help="Choose the type of target data you want to compare"

)

target\_data = handle\_data\_source(target\_type, "target")

if target\_data is not None:

st.success(f"✅ Target data loaded successfully: {len(target\_data)} rows, {len(target\_data.columns)} columns")

except Exception as e:

st.error(f"❌ An error occurred: {str(e)}")

log\_error(f"Application error: {str(e)}")

return

# Check if we have data either from direct loading or session state

if ((source\_data is not None and target\_data is not None) or

(isinstance(st.session\_state.source\_df, pd.DataFrame) and isinstance(st.session\_state.target\_df, pd.DataFrame))):

try:

# Get data from direct load or session state

from utils import clean\_df\_columns

# Validate and prepare source data

source\_df = source\_data if source\_data is not None else st.session\_state.get('source\_df')

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

st.error("❌ Source data is not available or invalid. Please load source data first.")

return

# Convert all columns to string type for consistent comparison

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

# Validate and prepare target data

target\_df = target\_data if target\_data is not None else st.session\_state.get('target\_df')

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

st.error("❌ Target data is not available or invalid. Please load target data first.")

return

# Convert all columns to string type for consistent comparison

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

try:

# Normalize columns for consistent mapping

cleaned\_source = clean\_df\_columns(source\_df)

cleaned\_target = clean\_df\_columns(target\_df)

if cleaned\_source.empty or cleaned\_target.empty:

st.error("❌ One or both datasets are empty. Please check your data.")

return

# Reset index to ensure consistent comparison

cleaned\_source = cleaned\_source.reset\_index(drop=True)

cleaned\_target = cleaned\_target.reset\_index(drop=True)

except Exception as e:

st.error(f"❌ Error cleaning data: {str(e)}")

return

# Store cleaned DataFrames in session state if they came from direct load

if source\_data is not None:

st.session\_state.source\_df = cleaned\_source

if target\_data is not None:

st.session\_state.target\_df = cleaned\_target

# Debug information

st.write("Source columns:", list(cleaned\_source.columns))

st.write("Target columns:", list(cleaned\_target.columns))

# Force re-mapping if source columns have changed

current\_source\_cols = list(cleaned\_source.columns)

if (not st.session\_state.get('column\_mapping') or

st.session\_state.get('last\_source\_columns') != current\_source\_cols):

# Clean column names before mapping

cleaned\_source.columns = [col.strip().lower() for col in cleaned\_source.columns]

cleaned\_target.columns = [col.strip().lower() for col in cleaned\_target.columns]

st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(cleaned\_source, cleaned\_target)

st.session\_state.last\_source\_columns = current\_source\_cols

# Debug mapping results

st.write("Mapped columns:", st.session\_state.column\_mapping)

if st.session\_state.column\_mapping:

st.success(f"✅ Successfully mapped {len(st.session\_state.column\_mapping)} columns automatically!")

else:

st.warning("⚠️ No automatic column mappings found. Please map columns manually below.")

except Exception as e:

st.error(f"❌ Error initializing data: {str(e)}")

log\_error(f"Data initialization error: {str(e)}")

return

# Show column mapping interface

st.subheader("Column Mapping")

show\_column\_mapping\_interface(source\_data, target\_data)

# Add join key selection with enhanced styling

st.markdown(

"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin: 2rem 0; border: 1px solid #dee2e6;'>

<h3 style='margin: 0; color: #1f77b4;'>Join Key Selection</h3>

<p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

Select columns to use as join keys for comparing records between source and target data.

</p>

</div>

""",

unsafe\_allow\_html=True

)

# Get mapped columns (where source and target are mapped)

mapped\_columns = {source\_col: target\_col

for source\_col, target\_col in st.session\_state.column\_mapping.items()

if source\_col in source\_data.columns and target\_col in target\_data.columns}

if not mapped\_columns:

st.warning("⚠️ No mapped columns available for join keys. Please map at least one column first.")

join\_keys = None

else:

# Create columns for join key selection and preview

col1, col2 = st.columns([1, 2])

with col1:

# Select join keys from mapped columns with enhanced styling

st.markdown("##### Select Join Key Columns")

selected\_keys = st.multiselect(

"Choose columns that uniquely identify records",

options=list(mapped\_columns.keys()),

help="These columns will be used to match records between source and target data"

)

if selected\_keys:

join\_keys = [(source\_col, mapped\_columns[source\_col]) for source\_col in selected\_keys]

st.markdown(

"""

<div style='background-color: #e8f4f8; padding: 0.75rem; border-radius: 5px; margin-top: 1rem;'>

<p style='margin: 0; color: #1f77b4;'>Selected Join Keys:</p>

</div>

""",

unsafe\_allow\_html=True

)

for source\_col, target\_col in join\_keys:

st.markdown(f"- {source\_col} → {target\_col}")

else:

st.info("ℹ️ No join keys selected. Comparison will be done row by row.")

join\_keys = None

with col2:

# Show preview of mapped columns with samples

st.markdown("##### Join Keys Preview")

mapped\_preview = []

for source\_col, target\_col in mapped\_columns.items():

source\_sample = source\_data[source\_col].head(3).tolist()

target\_sample = target\_data[target\_col].head(3).tolist()

mapped\_preview.append({

'Source Column': source\_col,

'Target Column': target\_col,

'Source Sample': str(source\_sample),

'Target Sample': str(target\_sample)

})

if mapped\_preview:

st.dataframe(

pd.DataFrame(mapped\_preview),

use\_container\_width=True,

height=200

)

# Store join keys in session state

st.session\_state.join\_keys = join\_keys

# Compare button section with enhanced styling

st.markdown(

"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin: 2rem 0;

border: 1px solid #dee2e6; text-align: center;'>

<h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Generate Comparison Reports</h3>

</div>

""",

unsafe\_allow\_html=True

)

# Center the compare button

col1, col2, col3 = st.columns([2, 1, 2])

with col2:

if st.button("🔄 Compare Data", type="primary", use\_container\_width=True):

with st.spinner("Generating comparison reports..."):

perform\_comparison()

def handle\_data\_source(source\_type: str, prefix: str) -> Optional[pd.DataFrame]:

"""Handle different types of data sources"""

try:

if source\_type in ["CSV File", "DAT File", "Parquet File", "Zipped Flat Files"]:

return handle\_file\_upload(source\_type, prefix)

elif source\_type in ["SQL Server", "Teradata", "Stored Procedure"]:

return handle\_database\_connection(source\_type, prefix)

elif source\_type == "API":

return handle\_api\_connection(prefix)

except Exception as e:

log\_error(f"Error handling {source\_type}: {str(e)}")

st.error(f"Error processing {source\_type}: {str(e)}")

return None

def handle\_file\_upload(file\_type: str, prefix: str) -> Optional[pd.DataFrame]:

"""Handle file upload for different file types with enhanced UI and error handling"""

# Add file upload UI with modern styling

st.markdown(

f"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

<h4 style='margin: 0; color: #1f77b4;'>{file\_type} Upload</h4>

<p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

Upload your {file\_type.lower()} and configure import settings below.

</p>

</div>

""",

unsafe\_allow\_html=True

)

# File upload section

uploaded\_file = st.file\_uploader(

f"Choose {file\_type}",

key=f"{prefix}\_file",

help=f"Select a {file\_type.lower()} to upload"

)

if uploaded\_file:

try:

# Show file info

file\_details = {

"Filename": uploaded\_file.name,

"File size": f"{uploaded\_file.size / 1024:.2f} KB",

"File type": uploaded\_file.type

}

st.markdown("##### File Details")

for key, value in file\_details.items():

st.text(f"{key}: {value}")

# Configuration options based on file type

with st.expander("Import Configuration", expanded=True):

if file\_type in ["CSV File", "DAT File"]:

col1, col2 = st.columns(2)

with col1:

delimiter = st.text\_input(

"Delimiter",

value=',' if file\_type == "CSV File" else '|',

key=f"{prefix}\_delimiter",

help="Enter the character used to separate columns"

)

with col2:

encoding = st.selectbox(

"File Encoding",

["utf-8", "latin1", "ascii"],

key=f"{prefix}\_encoding",

help="Select the file encoding"

)

# Load the file with progress indicator

with st.spinner(f"Loading {file\_type}..."):

if file\_type == "CSV File":

df = DataReader.load\_csv(uploaded\_file, delimiter=delimiter)

elif file\_type == "DAT File":

df = DataReader.load\_dat(uploaded\_file, delimiter=delimiter)

elif file\_type == "Parquet File":

df = DataReader.load\_parquet(uploaded\_file)

elif file\_type == "Zipped Flat Files":

df = DataReader.load\_zipped\_flat\_files(uploaded\_file, separator=delimiter)

if df is not None:

st.success(f"✅ {file\_type} loaded successfully!")

st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

# Show data preview in an expander

with st.expander("Data Preview"):

st.dataframe(

df.head(5),

use\_container\_width=True,

height=200

)

return df

else:

st.error(f"❌ Failed to load {file\_type}. Please check the file format and try again.")

except Exception as e:

st.error(f"❌ Error reading {file\_type}: {str(e)}")

log\_error(f"File upload error ({file\_type}): {str(e)}")

# Show detailed error message in an expander

with st.expander("Error Details"):

st.code(str(e))

return None

def handle\_database\_connection(db\_type: str, prefix: str) -> Optional[pd.DataFrame]:

"""Handle database connections with enhanced UI and error handling"""

# Add database connection UI with modern styling

st.markdown(

f"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

<h4 style='margin: 0; color: #1f77b4;'>{db\_type} Connection</h4>

<p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

Enter your {db\_type} connection details below. Column names will be automatically standardized for better mapping.

</p>

</div>

""",

unsafe\_allow\_html=True

)

# Create tabs for connection and query

conn\_tab, query\_tab = st.tabs(["Connection Details", "Query Configuration"])

with conn\_tab:

# Connection details with improved layout

col1, col2 = st.columns(2)

with col1:

host = st.text\_input(

"Server/Host",

key=f"{prefix}\_host",

help="Enter the server name or IP address"

)

with col2:

database = st.text\_input(

"Database",

key=f"{prefix}\_database",

help="Enter the database name"

)

# Authentication section

st.markdown("##### Authentication")

if db\_type in ["SQL Server", "Stored Procedure"]:

use\_windows\_auth = st.checkbox(

"Use Windows Authentication",

value=True,

key=f"{prefix}\_use\_windows\_auth",

help="Check to use Windows Authentication, uncheck for SQL Authentication"

)

if not use\_windows\_auth:

col1, col2 = st.columns(2)

with col1:

username = st.text\_input(

"Username",

key=f"{prefix}\_username",

help="Enter your SQL Server username"

)

with col2:

password = st.text\_input(

"Password",

type="password",

key=f"{prefix}\_password",

help="Enter your SQL Server password"

)

else:

use\_windows\_auth = False

col1, col2 = st.columns(2)

with col1:

username = st.text\_input(

"Username",

key=f"{prefix}\_username"

)

with col2:

password = st.text\_input(

"Password",

type="password",

key=f"{prefix}\_password"

)

with query\_tab:

if db\_type == "Stored Procedure":

proc\_name = st.text\_input(

"Stored Procedure Name",

key=f"{prefix}\_proc",

help="Enter the name of the stored procedure"

)

params = st.text\_area(

"Parameters (JSON format)",

key=f"{prefix}\_params",

help="Enter parameters as JSON, e.g., {\"param1\": \"value1\"}"

)

else:

query = st.text\_area(

"SQL Query",

key=f"{prefix}\_query",

height=150,

help="Enter your SQL query here. The query will be executed to fetch the data."

)

# Connection button with loading state

if st.button("🔌 Connect to Database", key=f"{prefix}\_connect", use\_container\_width=True):

with st.spinner(f"Connecting to {db\_type}..."):

try:

# Build connection parameters

conn\_params = {

'host': host,

'database': database,

'use\_windows\_auth': use\_windows\_auth if db\_type in ["SQL Server", "Stored Procedure"] else False

}

# Add username/password only if not using Windows Auth

if not (db\_type in ["SQL Server", "Stored Procedure"] and use\_windows\_auth):

if not username or not password:

st.error("❌ Username and password are required for SQL Authentication")

return None

conn\_params.update({

'username': username,

'password': password

})

# Validate connection parameters

if not host or not database:

st.error("❌ Server/Host and Database are required")

return None

# Execute query based on database type

if db\_type == "SQL Server":

if not query:

st.error("❌ SQL Query is required")

return None

try:

df = DatabaseConnector.get\_sqlserver\_data(conn\_params, query)

if df is not None and isinstance(df, pd.DataFrame) and not df.empty:

st.success("✅ SQL Server connection successful!")

st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

# Store DataFrame in session state

if prefix == "source":

st.session\_state['source\_df'] = df.copy()

else:

st.session\_state['target\_df'] = df.copy()

# Verify the DataFrame was stored correctly

stored\_df = st.session\_state.get('source\_df' if prefix == "source" else 'target\_df')

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

st.error("❌ Error storing DataFrame in session state")

return None

return df

else:

st.error("❌ No data returned from SQL Server or empty result")

return None

except Exception as e:

st.error(f"❌ Error processing SQL Server data: {str(e)}")

return None

elif db\_type == "Teradata":

if not query:

st.error("❌ SQL Query is required")

return None

try:

df = DatabaseConnector.get\_teradata\_data(conn\_params, query)

if df is not None and isinstance(df, pd.DataFrame) and not df.empty:

st.success("✅ Teradata connection successful!")

st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

# Store DataFrame in session state

if prefix == "source":

st.session\_state['source\_df'] = df.copy()

else:

st.session\_state['target\_df'] = df.copy()

# Verify the DataFrame was stored correctly

stored\_df = st.session\_state.get('source\_df' if prefix == "source" else 'target\_df')

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

st.error("❌ Error storing DataFrame in session state")

return None

return df

else:

st.error("❌ No data returned from Teradata or empty result")

return None

except Exception as e:

st.error(f"❌ Error processing Teradata data: {str(e)}")

return None

elif db\_type == "Stored Procedure":

if not proc\_name:

st.error("❌ Stored Procedure name is required")

return None

try:

df = DatabaseConnector.get\_data\_from\_stored\_proc(

conn\_params, proc\_name, eval(params) if params else None)

if df is not None and isinstance(df, pd.DataFrame) and not df.empty:

st.success("✅ Stored Procedure executed successfully!")

st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

# Store DataFrame in session state

if prefix == "source":

st.session\_state['source\_df'] = df.copy()

else:

st.session\_state['target\_df'] = df.copy()

# Verify the DataFrame was stored correctly

stored\_df = st.session\_state.get('source\_df' if prefix == "source" else 'target\_df')

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

st.error("❌ Error storing DataFrame in session state")

return None

return df

else:

st.error("❌ No data returned from Stored Procedure or empty result")

return None

except Exception as e:

st.error(f"❌ Error processing Stored Procedure data: {str(e)}")

return None

except Exception as e:

st.error(f"❌ Database connection error: {str(e)}")

log\_error(f"Database connection error ({db\_type}): {str(e)}")

return None

def handle\_api\_connection(prefix: str) -> Optional[pd.DataFrame]:

"""Handle API connections with enhanced UI and error handling"""

# Add API connection UI with modern styling

st.markdown(

"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

<h4 style='margin: 0; color: #1f77b4;'>API Connection</h4>

<p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

Configure your API connection details and parameters below.

</p>

</div>

""",

unsafe\_allow\_html=True

)

# Create tabs for basic and advanced configuration

basic\_tab, advanced\_tab = st.tabs(["Basic Configuration", "Advanced Settings"])

with basic\_tab:

# URL and Method

col1, col2 = st.columns([3, 1])

with col1:

api\_url = st.text\_input(

"API URL",

key=f"{prefix}\_api\_url",

help="Enter the complete API endpoint URL",

placeholder="https://api.example.com/data"

)

with col2:

method = st.selectbox(

"Method",

["GET", "POST"],

key=f"{prefix}\_method",

help="Select the HTTP method to use"

)

with advanced\_tab:

# Headers

st.markdown("##### Headers")

headers = st.text\_area(

"Headers (JSON format)",

key=f"{prefix}\_headers",

help="Enter headers as JSON object, e.g., {\"Authorization\": \"Bearer token\"}",

height=100,

placeholder="""

{

"Authorization": "Bearer your\_token",

"Content-Type": "application/json"

}"""

)

# Parameters

st.markdown("##### Parameters")

params = st.text\_area(

"Parameters (JSON format)",

key=f"{prefix}\_params",

help="Enter query parameters or POST body as JSON object",

height=100,

placeholder="""

{

"limit": 1000,

"offset": 0

}"""

)

# Connection button with loading state

if st.button("🔌 Connect to API", key=f"{prefix}\_connect", use\_container\_width=True):

with st.spinner("Connecting to API..."):

try:

# Validate inputs

if not api\_url:

st.error("❌ API URL is required")

return None

# Parse JSON inputs

try:

headers\_dict = eval(headers) if headers else None

params\_dict = eval(params) if params else None

except Exception as e:

st.error(f"❌ Invalid JSON format: {str(e)}")

return None

# Fetch data

df = APIFetcher.fetch\_api\_data(

api\_url=api\_url,

method=method,

headers=headers\_dict,

params=params\_dict

)

if df is not None:

st.success("✅ API connection successful!")

st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

# Show data preview in an expander

with st.expander("Data Preview"):

st.dataframe(

df.head(5),

use\_container\_width=True,

height=200

)

return df

else:

st.error("❌ No data received from API")

except Exception as e:

st.error(f"❌ API connection error: {str(e)}")

log\_error(f"API connection error: {str(e)}")

# Show detailed error message in an expander

with st.expander("Error Details"):

st.code(str(e))

return None

def show\_column\_mapping\_interface(source\_df: pd.DataFrame, target\_df: pd.DataFrame):

"""Show interface for column mapping"""

if not isinstance(source\_df, pd.DataFrame) or not isinstance(target\_df, pd.DataFrame):

st.error("❌ Source or target data is not available for mapping")

return

st.markdown(

"""

<div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

<h3 style='margin: 0; color: #1f77b4;'>Column Mapping Configuration</h3>

<p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

Map columns between source and target data sources. Use auto-mapping or manually adjust the mappings below.

</p>

</div>

""",

unsafe\_allow\_html=True

)

# Create tabs for auto and manual mapping

auto\_tab, manual\_tab = st.tabs(["Automatic Mapping", "Manual Mapping"])

with auto\_tab:

# Create two columns for the auto-map button and mapping status

col1, col2 = st.columns([1, 2])

with col1:

if st.button("🔄 Auto-Map Columns", key="auto\_map\_btn", use\_container\_width=True):

with st.spinner("Mapping columns..."):

try:

st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(source\_df, target\_df)

if st.session\_state.column\_mapping:

st.success(f"✅ Successfully mapped {len(st.session\_state.column\_mapping)} columns!")

else:

st.warning("⚠️ No automatic matches found. Please map columns manually.")

except Exception as e:

st.error(f"❌ Error during auto-mapping: {str(e)}")

with col2:

if st.session\_state.get('column\_mapping'):

total\_cols = len(source\_df.columns)

mapped\_cols = len(st.session\_state.column\_mapping)

mapping\_percentage = (mapped\_cols / total\_cols) \* 100

st.progress(mapping\_percentage / 100, text=f"Mapped {mapped\_cols} of {total\_cols} columns ({mapping\_percentage:.1f}%)")

with manual\_tab:

st.markdown("### Manual Column Mapping")

st.markdown("Select target columns for each source column below:")

# Initialize column mapping in session state if not exists

if 'column\_mapping' not in st.session\_state:

st.session\_state.column\_mapping = {}

# Create mapping interface for each source column

for source\_col in source\_df.columns:

col1, col2 = st.columns([2, 3])

with col1:

st.markdown(f"\*\*Source:\*\* {source\_col}")

st.caption(f"Sample: {str(source\_df[source\_col].head(2).tolist())}")

with col2:

# Get current mapping for this column

current\_mapping = st.session\_state.column\_mapping.get(source\_col, '')

# Create dropdown with target columns

target\_options = [''] + list(target\_df.columns)

selected\_target = st.selectbox(

"Map to target column",

options=target\_options,

index=target\_options.index(current\_mapping) if current\_mapping in target\_options else 0,

key=f"manual\_mapping\_{source\_col}"

)

# Update mapping if changed

if selected\_target:

if selected\_target != current\_mapping:

st.session\_state.column\_mapping[source\_col] = selected\_target

# Show sample of selected target column

st.caption(f"Target sample: {str(target\_df[selected\_target].head(2).tolist())}")

elif source\_col in st.session\_state.column\_mapping:

del st.session\_state.column\_mapping[source\_col]

st.divider()

# Show current mapping summary

if st.session\_state.column\_mapping:

st.success(f"✅ Currently mapped: {len(st.session\_state.column\_mapping)} columns")

with st.expander("View Current Mappings"):

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

st.write(f"{source\_col} → {target\_col}")

# Show mapping table with enhanced styling

st.markdown("### Current Mapping Overview")

# Create a DataFrame to display the mapping

mapping\_data = []

for source\_col in source\_df.columns:

source\_sample = str(source\_df[source\_col].head(2).tolist())

target\_col = st.session\_state.column\_mapping.get(source\_col, '')

target\_sample = str(target\_df[target\_col].head(2).tolist()) if target\_col else ''

mapping\_data.append({

'Source Column': source\_col,

'Source Sample': source\_sample,

'Target Column': target\_col,

'Target Sample': target\_sample,

'Status': '✅' if target\_col else '❌'

})

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

# Display the mapping table with custom styling

st.dataframe(

mapping\_df,

use\_container\_width=True,

height=400,

column\_config={

"Status": st.column\_config.Column(

"Mapping Status",

help="✅: Mapped | ❌: Unmapped",

width="small"

)

}

)

def perform\_comparison():

"""Perform the comparison and generate reports"""

try:

# Initial validation

if not isinstance(st.session\_state.get('source\_df'), pd.DataFrame):

st.error("❌ Source data is not available. Please load source data first.")

return

if not isinstance(st.session\_state.get('target\_df'), pd.DataFrame):

st.error("❌ Target data is not available. Please load target data first.")

return

if not st.session\_state.get('column\_mapping'):

st.error("❌ No column mappings defined. Please map columns before comparing.")

return

# Validate column mappings

source\_df = st.session\_state.source\_df

target\_df = st.session\_state.target\_df

invalid\_mappings = []

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

if source\_col not in source\_df.columns:

invalid\_mappings.append(f"Source column '{source\_col}' not found")

if target\_col not in target\_df.columns:

invalid\_mappings.append(f"Target column '{target\_col}' not found")

if invalid\_mappings:

st.error("❌ Invalid column mappings detected:")

for msg in invalid\_mappings:

st.write(f"- {msg}")

return

st.markdown("### Comparison Results")

# Create reports directory with full path

reports\_dir = os.path.join(os.getcwd(), "reports")

os.makedirs(reports\_dir, exist\_ok=True)

timestamp = format\_timestamp()

# Log report generation start

st.info("📊 Generating comparison reports...")

# Generate reports

join\_keys = st.session\_state.get('join\_keys', None)

with st.spinner("Generating comparison reports..."):

try:

from ydata\_profiling import ProfileReport

import streamlit.components.v1 as components

# Generate Y-Data HTML profiling reports

st.write("Generating Y-Data Profiling Reports...")

# Source profiling

source\_profile = ProfileReport(

st.session\_state.source\_df,

title="Source Data Profile Report",

minimal=True

)

source\_html = os.path.join(reports\_dir, f"SourceProfile\_{timestamp}.html")

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

st.success("✅ Source profile report generated")

# Target profiling

target\_profile = ProfileReport(

st.session\_state.target\_df,

title="Target Data Profile Report",

minimal=True

)

target\_html = os.path.join(reports\_dir, f"TargetProfile\_{timestamp}.html")

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

st.success("✅ Target profile report generated")

# Generate other reports

st.write("Generating Comparison Reports...")

# Generate difference report

diff\_df, has\_differences = ReportGenerator.generate\_diff\_report(

st.session\_state.source\_df,

st.session\_state.target\_df,

st.session\_state.column\_mapping,

st.session\_state.excluded\_columns,

join\_keys=join\_keys

)

# Generate profiling report

profile\_df = ReportGenerator.generate\_profiling\_report(

st.session\_state.source\_df,

st.session\_state.target\_df,

st.session\_state.column\_mapping,

join\_keys=join\_keys

)

# Generate regression report (independent of join keys)

regression\_path = f"reports/RegressionReport\_{timestamp}.xlsx"

ReportGenerator.generate\_regression\_report(

st.session\_state.source\_df,

st.session\_state.target\_df,

st.session\_state.column\_mapping,

regression\_path

)

# Generate side-by-side report

side\_by\_side\_path = f"reports/DifferenceReport\_{timestamp}.xlsx"

side\_by\_side\_df, \_ = ReportGenerator.generate\_side\_by\_side\_report(

st.session\_state.source\_df,

st.session\_state.target\_df,

st.session\_state.column\_mapping,

side\_by\_side\_path,

join\_keys=join\_keys

)

# Display Y-Data Profiling Reports in expandable sections

st.subheader("Y-Data Profiling Reports")

# Generate comparison profile with side-by-side data

comparison\_df = pd.DataFrame()

# Process each column in the mapping

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

if source\_col in st.session\_state.source\_df.columns and target\_col in st.session\_state.target\_df.columns:

comparison\_df[f'Source\_{source\_col}'] = st.session\_state.source\_df[source\_col]

comparison\_df[f'Target\_{target\_col}'] = st.session\_state.target\_df[target\_col]

# Generate profile report with enhanced configuration

comparison\_profile = ProfileReport(

comparison\_df,

title="Source vs Target Comparison Profile",

minimal=False, # Use full report for more detailed comparison

correlations={"cramers": True}, # Add correlation analysis

vars={"num": {"low\_categorical\_threshold": 0}}, # Treat numeric columns as continuous

interactions={"continuous": True} # Show interactions between continuous variables

)

comparison\_html = os.path.join(reports\_dir, f"ComparisonProfile\_{timestamp}.html")

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

st.success("✅ Comparison profile report generated")

# Generate DataCompy comparison report

try:

# Prepare join keys for DataCompy

join\_columns = []

source\_df = st.session\_state.source\_df.copy()

target\_df = st.session\_state.target\_df.copy()

if join\_keys:

join\_columns = [key[0] for key in join\_keys] # Use source column names

else:

# If no join keys specified, use index

source\_df['\_index'] = source\_df.index

target\_df['\_index'] = target\_df.index

join\_columns = ['\_index']

# Create DataCompy comparison

comparison = datacompy.Compare(

source\_df,

target\_df,

join\_columns=join\_columns,

df1\_name='Source',

df2\_name='Target'

)

# Get column statistics safely

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

try:

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

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

not comparison.column\_stats.empty):

column\_stats\_html = comparison.column\_stats.to\_html()

except Exception as e:

st.warning(f"Could not generate column statistics: {str(e)}")

# Generate HTML report

datacompy\_html = os.path.join(reports\_dir, f"DataCompyReport\_{timestamp}.html")

with open(datacompy\_html, '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>{comparison.report()}</pre>

</div>

<div class="section">

<h2>Detailed Statistics</h2>

<h3>Matches</h3>

<div class="match">

<p>Number of rows match: {comparison.count\_matching\_rows()}</p>

<p>Number of columns match: {len(set(source\_df.columns).intersection(target\_df.columns))}</p>

</div>

<h3>Mismatches</h3>

<div class="mismatch">

<p>Rows only in Source: {len(comparison.df1\_unq\_rows)}</p>

<p>Rows only in Target: {len(comparison.df2\_unq\_rows)}</p>

<p>Source-only columns: {len(set(source\_df.columns) - set(target\_df.columns))}</p>

<p>Target-only columns: {len(set(target\_df.columns) - set(source\_df.columns))}</p>

</div>

</div>

<div class="section">

<h2>Column Statistics</h2>

{column\_stats\_html}

</div>

</div>

</body>

</html>

""")

except Exception as e:

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

datacompy\_html = None

# Store DataCompy report path

st.session\_state.report\_paths['datacompy\_html'] = datacompy\_html

# Store report paths in session state to prevent page reset

if 'report\_paths' not in st.session\_state:

st.session\_state.report\_paths = {}

st.session\_state.report\_paths.update({

'source\_html': source\_html,

'target\_html': target\_html,

'comparison\_html': comparison\_html

})

tab1, tab2, tab3, tab4 = st.tabs(["Source Profile", "Target Profile", "Comparison Profile", "DataCompy Report"])

with tab1:

with open(source\_html, 'r', encoding='utf-8') as f:

components.html(f.read(), height=600, scrolling=True)

# Download button for source profile

with open(source\_html, 'rb') as f:

st.download\_button(

"Download Source Profile Report",

f,

file\_name=f"SourceProfile\_{timestamp}.html",

mime="text/html",

key="download\_source"

)

with tab2:

with open(target\_html, 'r', encoding='utf-8') as f:

components.html(f.read(), height=600, scrolling=True)

# Download button for target profile

with open(target\_html, 'rb') as f:

st.download\_button(

"Download Target Profile Report",

f,

file\_name=f"TargetProfile\_{timestamp}.html",

mime="text/html",

key="download\_target"

)

with tab3:

with open(comparison\_html, 'r', encoding='utf-8') as f:

components.html(f.read(), height=600, scrolling=True)

# Download button for comparison profile

with open(comparison\_html, 'rb') as f:

st.download\_button(

"Download Comparison Profile Report",

f,

file\_name=f"ComparisonProfile\_{timestamp}.html",

mime="text/html",

key="download\_comparison"

)

with tab4:

if datacompy\_html and os.path.exists(datacompy\_html):

with open(datacompy\_html, 'r', encoding='utf-8') as f:

components.html(f.read(), height=600, scrolling=True)

col1, col2 = st.columns([3, 1])

with col1:

# Download button for DataCompy report

with open(datacompy\_html, 'rb') as f:

st.download\_button(

"📥 Download DataCompy Report",

f,

file\_name=f"DataCompyReport\_{timestamp}.html",

mime="text/html",

key="download\_datacompy",

use\_container\_width=True

)

# Show additional DataCompy insights

with st.expander("📊 DataCompy Insights", expanded=False):

try:

st.write("### Match Statistics")

col1, col2, col3 = st.columns(3)

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

source\_only\_rows = len(comparison.df1\_unq\_rows)

target\_only\_rows = len(comparison.df2\_unq\_rows)

with col1:

st.metric(

"Matching Rows",

matching\_rows,

delta=f"{matching\_rows / len(source\_df) \* 100:.1f}%"

)

with col2:

st.metric(

"Source Only Rows",

source\_only\_rows,

delta=f"{source\_only\_rows / len(source\_df) \* 100:.1f}%"

)

with col3:

st.metric(

"Target Only Rows",

target\_only\_rows,

delta=f"{target\_only\_rows / len(target\_df) \* 100:.1f}%"

)

st.write("### Column Analysis")

# Get column sets

source\_cols = set(source\_df.columns)

target\_cols = set(target\_df.columns)

common\_cols = source\_cols.intersection(target\_cols)

source\_only = source\_cols - target\_cols

target\_only = target\_cols - source\_cols

# Display column sets

col1, col2 = st.columns(2)

with col1:

st.write("Source-only Columns:")

if source\_only:

for col in sorted(source\_only):

st.info(f"- {col}")

else:

st.success("No source-only columns")

with col2:

st.write("Target-only Columns:")

if target\_only:

for col in sorted(target\_only):

st.info(f"- {col}")

else:

st.success("No target-only columns")

st.write("### Common Column Analysis")

if common\_cols:

for col in sorted(common\_cols):

# Calculate match rate for common columns

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

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

common\_values = source\_values.intersection(target\_values)

all\_values = source\_values.union(target\_values)

match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

# Display progress bar with details

st.progress(

match\_rate,

text=f"{col}: {match\_rate\*100:.1f}% unique values match"

)

with st.expander(f"Details for {col}", expanded=False):

st.write(f"- Unique values in Source: {len(source\_values)}")

st.write(f"- Unique values in Target: {len(target\_values)}")

st.write(f"- Common unique values: {len(common\_values)}")

if len(source\_values - target\_values) > 0:

st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

if len(target\_values - source\_values) > 0:

st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

else:

st.warning("No common columns found between source and target datasets")

except Exception as e:

st.error(f"Error displaying insights: {str(e)}")

else:

st.error("❌ DataCompy report generation failed. Please check the data and try again.")

with col2:

st.write("Target-only Columns:")

if target\_only:

for col in sorted(target\_only):

st.info(f"- {col}")

else:

st.success("No target-only columns")

st.write("### Common Column Analysis")

if common\_cols:

for col in sorted(common\_cols):

# Calculate match rate for common columns

source\_values = set(st.session\_state.source\_df[col].dropna())

target\_values = set(st.session\_state.target\_df[col].dropna())

common\_values = source\_values.intersection(target\_values)

all\_values = source\_values.union(target\_values)

match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

# Display progress bar with details

st.progress(

match\_rate,

text=f"{col}: {match\_rate\*100:.1f}% unique values match"

)

with st.expander(f"Details for {col}", expanded=False):

st.write(f"- Unique values in Source: {len(source\_values)}")

st.write(f"- Unique values in Target: {len(target\_values)}")

st.write(f"- Common unique values: {len(common\_values)}")

if len(source\_values - target\_values) > 0:

st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

if len(target\_values - source\_values) > 0:

st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

else:

st.warning("No common columns found between source and target datasets")

except ImportError:

st.warning("Y-Data Profiling package not installed. Please install ydata-profiling package.")

# Continue with other reports...

# Store report paths in session state

st.session\_state.report\_paths.update({

'side\_by\_side\_path': side\_by\_side\_path,

'profiling\_path': f"reports/ProfilingReport\_{timestamp}.xlsx",

'regression\_path': regression\_path

})

# Create zip file containing all reports

st.divider()

zip\_path = f"reports/FinalComparison\_{timestamp}.zip"

try:

import zipfile

with zipfile.ZipFile(zip\_path, 'w') as zipf:

# Add all reports to zip

for report\_name, report\_path in st.session\_state.report\_paths.items():

if report\_path and os.path.exists(report\_path):

zipf.write(report\_path, os.path.basename(report\_path))

# Add download button for zip file

st.markdown("### 📦 Download Complete Comparison Package")

st.caption("Contains all reports: Regression, Side by Side, DataCompy, and Profile Reports")

with open(zip\_path, 'rb') as f:

st.download\_button(

"📥 Download All Reports (ZIP)",

f,

file\_name=f"FinalComparison\_{timestamp}.zip",

mime="application/zip",

key="download\_all",

use\_container\_width=True

)

except Exception as e:

st.error(f"Error creating zip file: {str(e)}")

st.divider()

# Display results and download buttons

col1, col2 = st.columns(2)

with col1:

st.markdown("#### Data Comparison Results")

st.dataframe(diff\_df)

if os.path.exists(side\_by\_side\_path):

with open(side\_by\_side\_path, 'rb') as f:

st.download\_button(

"Download Difference Report",

f,

file\_name=f"DifferenceReport\_{timestamp}.xlsx",

mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

key="download\_difference"

)

with col2:

st.markdown("#### Y-Data Profiling Results")

if not profile\_df.empty:

# Format the dataframe for better display

st.write("Column-wise Statistical Comparison:")

# Create tabs for different views

metrics\_tab, details\_tab = st.tabs(["Main Metrics", "Detailed Statistics"])

with metrics\_tab:

main\_metrics = profile\_df[['Column', 'Source\_Count', 'Target\_Count', 'Match\_Percentage']]

st.dataframe(main\_metrics, use\_container\_width=True)

with details\_tab:

st.dataframe(profile\_df, use\_container\_width=True)

# Save profiling report

profiling\_path = st.session\_state.report\_paths['profiling\_path']

profile\_df.to\_excel(profiling\_path, index=False)

# Download buttons with unique keys

col3, col4 = st.columns(2)

with col3:

with open(profiling\_path, 'rb') as f:

st.download\_button(

"Download Profiling Report",

f,

file\_name=f"ProfilingReport\_{timestamp}.xlsx",

mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

key="download\_profiling"

)

with col4:

if os.path.exists(regression\_path):

with open(regression\_path, 'rb') as f:

st.download\_button(

"Download Regression Report",

f,

file\_name=f"RegressionReport\_{timestamp}.xlsx",

mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

key="download\_regression"

)

else:

st.error("Unable to generate profiling report. Please check the data and try again.")

except Exception as e:

log\_error(f"Error performing comparison: {str(e)}")

st.error(f"Error performing comparison: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

main()