Data Science Without Borders

Profiling the Source Data

Contents

Data Profile

Data profiling is a critical first step in any data analysis or machine learning project. It involves examining the dataset at a granular level to understand its structure, quality, and statistical properties. For the source dataset in this case (the dataset used here is Fake), this profiling will help us:

a) Understand the data's composition: types, distributions, patterns

b) Identify data quality issues: missing values, outliers, inconsistencies

c) Discover relationships between variables

d) Establish baselines for comparison with future synthetic data

e) Guide preprocessing decisions for machine learning

This detailed profiling will cover all aspects of the dataset using Python's most powerful data analysis libraries. It will also provide clear explanations of each check and its importance.

PostgreSQL Table

synthetic_dswb_training.source_dataset_fake

Table Description

```
CREATE TABLE IF NOT EXISTS synthetic_dswb_training.source_dataset_fake
(
    id character varying(10)  NOT NULL,
    first_name character varying(30),
    last_name character varying(30),
    sex character varying(6),
    dob date,
    village_name character varying(30),
    occupation character varying(30),
    covid_19_first_vacc_date date,
    age_on_first_vacc integer,
    vacc_manufacturer character varying(30),
    CONSTRAINT source_dataset_fake_pkey PRIMARY KEY (id)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS synthetic_dswb_training.source_dataset_fake
    OWNER to postgres;
```

The table synthetic_dswb_training.source_dataset_fake is a structured dataset that stores records related to individuals, particularly for training purposes within the Data Science Without Borders (DSWB) initiative. It captures demographic, geographic, and vaccination-related information. Below is a textual description of its structure and purpose:

a) **id** (*character varying(10)*), NOT NULL, PRIMARY KEY): A unique identifier for each individual in the dataset.
b) **first_name** (*character varying(30)*): The given name of the individual.
c) **last_name** (*character varying(30)*): The surname or family name of the individual.
d) **sex** (*character varying(6)*): The gender of the individual, typically represented as "Male" or "Female."
e) **dob** (*date*): The date of birth of the individual.
f) **village_name** (*character varying(30)*): The name of the village where the individual resides.
g) **occupation** (*character varying(30)*): The primary occupation of the individual.
h) **covid_19_first_vacc_date** (*date*): The date when the individual received their first COVID-19 vaccination.
i) **age_on_first_vacc** (*integer*): The individual's age at the time of receiving the first COVID-19 vaccine.
j) **vacc_manufacturer** (*character varying(30)*): The manufacturer of the COVID-19 vaccine received by the individual (e.g., Pfizer, Moderna, AstraZeneca).

The table is likely to be used to support analysis and generate synthetic data for training purposes. The inclusion of vaccination details alongside demographic information suggests its relevance in public health research.

The Python Code for Data Profiling

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import psycopg2
from sqlalchemy import create_engine
from scipy import stats
import missingno as msno
import warnings
from docx import Document
from docx.shared import Inches
import io

warnings.filterwarnings('ignore')


def create_word_report():
    """Initialize and configure Word document"""
    doc = Document()
    doc.add_heading('Comprehensive Data Profile Report', 0)
    doc.add_paragraph(f"Report generated on: {pd.Timestamp.now().strftime('%Y-%m-%d %H:%M:%S')}")
    return doc
```

```
def add_to_report(doc, content, content_type='text', fig=None, fig_width=6):
    """Add content to Word document"""
    if content_type == 'text':
        doc.add_paragraph(content)
    elif content_type == 'table':
        doc.add_paragraph(content)
    elif content_type == 'figure':
        if fig:
            # Save figure to bytes buffer
            buf = io.BytesIO()
            fig.savefig(buf, format='png', dpi=300, bbox_inches='tight')
            buf.seek(0)

            # Add to Word doc
            doc.add_picture(buf, width=Inches(fig_width))
            plt.close(fig)
        doc.add_paragraph()


# 1. Database Connection
def connect_to_postgres():
    try:
        conn_string = "postgresql://postgres:password1234@localhost:5432/dswb_training" #Change it accordingly
        engine = create_engine(conn_string)
        conn = engine.connect()
        print(" ✅ Successfully connected to PostgreSQL")
        return conn, engine
    except Exception as e:
        print(f" ❌ Connection failed: {str(e)}")
        return None, None


# 2. Data Loading
def load_full_data(conn, table_name):
    try:
        query = f"SELECT * FROM {table_name}"
        df = pd.read_sql(query, conn)
        print(f" 📊 Successfully loaded {len(df):,} records from {table_name}")
        return df
    except Exception as e:
        print(f" ❌ Data loading failed: {str(e)}")
        return None


# 3. Outlier Detection
def detect_outliers(series):
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1
    return series[(series < (Q1 - 1.5 * IQR)) | (series > (Q3 + 1.5 * IQR))]
```

```
# 4. Main Analysis Function
def analyze_full_dataset(df, doc):
    add_to_report(doc, "\n=== 🌏 COMPREHENSIVE DATA PROFILE ===", 'text')
    add_to_report(doc, f" 📝 Total records: {len(df):,}", 'text')
    add_to_report(doc, f" 📋 Columns ({len(df.columns)}): {list(df.columns)}", 'text')

    # Skip ID columns from analysis
    id_cols = [col for col in df.columns if 'id' in col.lower()]
    if id_cols:
        add_to_report(doc, f"\n ⏭️ Skipping ID columns: {id_cols}", 'text')
        df = df.drop(columns=id_cols)

    # A. Basic Metadata
    add_to_report(doc, "\nA. 📌 BASIC METADATA", 'text')
    metadata = pd.DataFrame({
        'Data Type': df.dtypes,
        'Missing Values': df.isna().sum(),
        'Missing %': (df.isna().mean() * 100).round(2),
        'Unique Values': df.nunique()
    })
    add_to_report(doc, metadata.to_string(), 'table')

    # B. Advanced Quality Checks
    add_to_report(doc, "\nB. 🔎 ADVANCED QUALITY CHECKS", 'text')

    # Duplicates
    duplicates = df.duplicated().sum()
    add_to_report(doc, f"\n 🔹 Duplicate Rows: {duplicates} ({duplicates / len(df):.2%})", 'text')
    if duplicates > 0:
        add_to_report(doc, "Sample duplicates:", 'text')
        add_to_report(doc, df[df.duplicated(keep=False)].sample(min(3, duplicates)).to_string(), 'table')

    # Outliers
    num_cols = df.select_dtypes(include=np.number).columns
    if len(num_cols) > 0:
        add_to_report(doc, "\n 🔹 Outlier Report:", 'text')
        outlier_report = []
        for col in num_cols:
            outliers = detect_outliers(df[col])
            outlier_report.append({
                'Column': col,
                'Outliers': len(outliers),
                '% Outliers': f"{len(outliers) / len(df):.2%}",
                'Min': outliers.min() if len(outliers) > 0 else None,
                'Max': outliers.max() if len(outliers) > 0 else None
            })
        add_to_report(doc, pd.DataFrame(outlier_report).to_string(index=False), 'table')

    # Constant Features
    constant_cols = [col for col in df.columns if df[col].nunique() == 1]
    add_to_report(doc, f"\n 🔹 Constant Features: {constant_cols or 'None found'}", 'text')
```

```
# High-Cardinality Features
high_card_cols = [col for col in df.select_dtypes(include=['object', 'category'])
            if df[col].nunique() > 50]
add_to_report(doc, f"  🔹 High-Cardinality (>50 unique): {high_card_cols or 'None found'}", 'text')

# C. Numerical Analysis
if len(num_cols) > 0:
  add_to_report(doc, "\nC. 🔢 NUMERICAL FEATURES", 'text')
  num_stats = df[num_cols].describe().T
  num_stats['skewness'] = df[num_cols].skew()
  num_stats['kurtosis'] = df[num_cols].kurt()
  num_stats['outliers'] = [len(detect_outliers(df[col])) for col in num_cols]
  add_to_report(doc, num_stats.to_string(), 'table')

  # Plot numerical distributions
  for col in num_cols:
    fig, ax = plt.subplots(1, 2, figsize=(12, 4))
    sns.histplot(df[col], kde=True, ax=ax[0])
    ax[0].set_title(f"Distribution of {col}")
    sns.boxplot(x=df[col], ax=ax[1])
    ax[1].set_title(f"Spread of {col}")
    add_to_report(doc, f"\nDistribution and Spread of {col}:", 'text', fig, fig_width=6)

# D. Categorical Analysis
cat_cols = df.select_dtypes(include=['object', 'category']).columns
if len(cat_cols) > 0:
  add_to_report(doc, "\nD. 🔑 CATEGORICAL FEATURES", 'text')
  for col in cat_cols:
    add_to_report(doc, f"\n{col} Distribution:", 'text')
    freq = df[col].value_counts(dropna=False)
    perc = df[col].value_counts(normalize=True, dropna=False).mul(100).round(2)
    cat_stats = pd.DataFrame({'Count': freq, 'Percentage': perc})
    add_to_report(doc, cat_stats.to_string(), 'table')

    # Plot top categories
    plt.figure(figsize=(10, 5))
    if df[col].nunique() > 20:
      top_cats = freq.head(20)
      sns.barplot(x=top_cats.values, y=top_cats.index)
      plt.title(f"Top 20 {col} Categories")
    else:
      sns.countplot(y=col, data=df, order=freq.index)
      plt.title(f"{col} Distribution")
    add_to_report(doc, "", 'figure', plt.gcf(), fig_width=6)

# E. Temporal Analysis
date_cols = [col for col in df.columns if any(kw in col.lower() for kw in ['date', 'time', 'year'])]
if date_cols:
  add_to_report(doc, "\nE. ⏳ TEMPORAL ANALYSIS", 'text')
  for col in date_cols:
    try:
```

```
            df[col] = pd.to_datetime(df[col])
            add_to_report(doc, f"\n{col} Analysis:", 'text')
            add_to_report(doc, f"Time range: {df[col].min()} to {df[col].max()}", 'text')
            add_to_report(doc, f"Missing values: {df[col].isna().sum()} ({df[col].isna().mean():.2%})", 'text')

            # Temporal distribution plot
            plt.figure(figsize=(12, 5))
            if (df[col].max() - df[col].min()).days > 365:  # More than 1 year
                time_series = df[col].dt.to_period('M').value_counts().sort_index()
                time_series.plot(kind='line', marker='o')
                plt.title(f"Monthly Distribution of {col}")
            else:
                time_series = df[col].dt.to_period('D').value_counts().sort_index()
                time_series.plot(kind='line')
                plt.title(f"Daily Distribution of {col}")
            plt.grid(True)
            add_to_report(doc, "", 'figure', plt.gcf(), fig_width=6)

        except Exception as e:
            add_to_report(doc, f"⚠️ Could not analyze {col}: {str(e)}", 'text')
    else:
        add_to_report(doc, "\nE. ⏳ No temporal columns found", 'text')

    # F. Correlation & Relationship Analysis
    add_to_report(doc, "\nF. 🔗 CORRELATION ANALYSIS", 'text')

    # Numerical Correlations
    if len(num_cols) > 1:
        add_to_report(doc, "\n 🔹 Numerical Feature Correlations:", 'text')
        corr_matrix = df[num_cols].corr()
        plt.figure(figsize=(12, 8))
        sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0, fmt=".2f")
        plt.title("Numerical Feature Correlation Matrix")
        add_to_report(doc, "", 'figure', plt.gcf(), fig_width=6)

        # Top correlations
        upper = corr_matrix.where(np.triu(np.ones_like(corr_matrix, dtype=bool)))
        top_corr = upper.stack().sort_values(key=abs, ascending=False).head(10)
        add_to_report(doc, "Top Correlations:", 'text')
        add_to_report(doc, top_corr.to_string(), 'table')

    # Categorical-Numerical Relationships
    if len(num_cols) > 0 and len(cat_cols) > 0:
        add_to_report(doc, "\n 🔹 Categorical-Numerical Relationships:", 'text')
        for num_col in num_cols[:3]:  # Limit to top 3 numerical for brevity
            for cat_col in cat_cols[:3]:  # Limit to top 3 categorical
                if df[cat_col].nunique() < 10:  # Avoid high-cardinality
                    plt.figure(figsize=(10, 5))
                    sns.boxplot(x=cat_col, y=num_col, data=df)
                    plt.title(f"{num_col} by {cat_col}")
                    plt.xticks(rotation=45)
                    add_to_report(doc, f"Boxplot of {num_col} by {cat_col}:", 'text', plt.gcf(), fig_width=6)
```

```
# 5. Main Execution
if __name__ == "__main__":
    # Initialize Word document
    doc = create_word_report()

    # Connect to PostgreSQL
    conn, engine = connect_to_postgres()

    if conn:
        # Load your actual data
        table_name = "synthetic_dswb_training.source_dataset_fake"
        df = load_full_data(conn, table_name)

        if df is not None:
            # Perform full analysis and add to report
            analyze_full_dataset(df, doc)

        # Clean up
        conn.close()
        engine.dispose()
        print("\n ✅ Analysis completed successfully")

        # Save the Word report
        report_path = "D:/Platform/Source_Data_Analysis_Report.docx"
        doc.save(report_path)
        print(f" 📄 Report saved to: {report_path}")
```


Explanation

This Python script performs a comprehensive profiling on a dataset loaded from a PostgreSQL database and generates a detailed report in a Microsoft Word document.

Library Imports:

The script begins by importing the necessary Python libraries:

a) **pandas as pd**: Used for data manipulation and analysis, providing the DataFrame structure which is central to working with tabular data.
b) **numpy as np**: Provides support for numerical operations, including array manipulation and mathematical functions.
c) **matplotlib.pyplot as plt**: A plotting library used for creating static, interactive, and animated visualizations in Python.
d) **seaborn as sns**: A data visualization library built on top of Matplotlib, offering a higher-level interface for creating informative and aesthetically pleasing statistical graphics.
e) **psycopg2**: A PostgreSQL adapter for Python, allowing the script to connect to and interact with PostgreSQL databases.

f) **sqlalchemy**: A powerful SQL toolkit and Object-Relational Mapper (ORM). Here, it's used specifically through its create_engine function to establish a connection to the PostgreSQL database using a connection string.
g) **scipy.stats**: A module from the SciPy library containing various statistical functions and distributions. While imported, it's not directly used in the current version of the script.
h) **missingno as msno**: A library for visualizing missing data patterns in datasets. It's imported but not directly used in the current version.
i) **warnings**: A built-in Python module used to handle warning messages.
j) **docx**: The core library for creating and modifying Microsoft Word .docx files.
k) **docx.shared.Inches**: A utility class from the docx library used to specify measurements in inches, particularly useful for setting the width of images in the Word document.
l) **io**: A module providing tools for working with various types of I/O (input/output). Here, io.BytesIO is used to handle in-memory binary streams for saving Matplotlib figures before embedding them in the Word document.

Warning Handling:

This is used for the warnings module to ignore all warning messages that might be generated during the script's execution. This can be helpful to suppress less critical warnings and keep the output and the generated report cleaner. However, in a production environment, it's often better to handle warnings appropriately.

Word Report Initialization (create_word_report()):

This function is responsible for setting up the initial Word document:

a) doc = Document(): Creates a new, empty Microsoft Word document object using the docx library.
b) doc.add_heading('Comprehensive Data Analysis Report', 0): Adds the main title "Comprehensive Data Analysis Report" to the document. The 0 indicates that this is a level 0 heading, which is typically the main title of the document.
c) doc.add_paragraph(f"Report generated on: {pd.Timestamp.now().strftime('%Y-%m-%d %H:%M:%S')}"): Adds a paragraph containing the date and time when the report was generated. pd.Timestamp.now() gets the current timestamp, and .strftime('%Y-%m-%d %H:%M:%S') formats it into a readable string.
d) return doc: Returns the initialized docx.Document object, which will be used to add further content.

Adding Content to the Word Report (add_to_report()):

This function provides a flexible way to add different types of content to the Word document:

a) doc: The docx.Document object to which content will be added.

b) content: The actual content to be added. Its type depends on content_type.

c) content_type: A string specifying the type of content. It can be:

   i. 'text' (default): The content is treated as a plain text string and added as a new paragraph.

   ii. 'table': The content is expected to be a string representation of a table (typically generated by pandas.DataFrame.to_string()). It's added as a new paragraph of text.

   iii. 'figure': Indicates that a Matplotlib figure needs to be added. In this case, the fig argument should hold the Matplotlib figure object.

d) fig: An optional Matplotlib figure object. This is used only when content_type is 'figure'.

e) fig_width: An optional float specifying the desired width of the figure in the Word document, measured in inches. It defaults to 6 inches.

The function's logic is as follows:

a) If content_type is 'text', it directly adds the content as a new paragraph to the Word document using doc.add_paragraph().

b) If content_type is 'table', it also adds the content as a new paragraph. This assumes that the string representation of the table is suitable for inclusion in the text.

c) If content_type is 'figure' and a fig object is provided:

   i. buf = io.BytesIO(): Creates an in-memory binary stream using io.BytesIO. This acts as a temporary file in memory.

   ii. fig.savefig(buf, format='png', dpi=300, bbox_inches='tight'): Saves the Matplotlib figure (fig) to the in-memory buffer (buf) in PNG format.

      a. format='png': Specifies the image format. PNG is a good choice for embedding in documents.

      b. dpi=300: Sets the dots per inch for the image, resulting in a higher-resolution image.

      c. bbox_inches='tight': Attempts to save the figure with minimal whitespace around the plot.

   iii. buf.seek(0): Resets the position of the buffer to the beginning so that the image data can be read from the start.

   iv. doc.add_picture(buf, width=Inches(fig_width)): Adds the image from the buffer (buf) to the Word document. width=Inches(fig_width) sets the width of the image in the document to the specified fig_width. The height is automatically adjusted to maintain the aspect ratio.

   v. plt.close(fig): Closes the Matplotlib figure to free up memory. It's important to close figures when you are done with them, especially in loops where many figures might be created.

d) Finally, doc.add_paragraph() is called without any arguments. This adds an empty paragraph after each piece of content, providing some visual separation in the generated Word document.

Database Connection (connect_to_postgres()):

This function handles the connection to a PostgreSQL database:

a) It defines conn_string: This string contains the necessary information to connect to the PostgreSQL database. It follows the format postgresql://username:password@host:port/database_name. Important: You need to replace "postgres:password1234@localhost:5432/PhD" with your actual database credentials (username, password, host address, port number, and database name).

b) engine = create_engine(conn_string): Uses SQLAlchemy's create_engine() function to create a database engine. The engine is a factory that can create database connections. The first part of the connection string (postgresql+psycopg2) specifies the database dialect (PostgreSQL with the psycopg2 driver).

c) conn = engine.connect(): Establishes an actual connection to the database using the created engine. The connect() method returns a connection object.

d) print("✅ Successfully connected to PostgreSQL"): If the connection is successful, a confirmation message is printed to the console.

e) return conn, engine: The function returns both the database connection object (conn) and the database engine object (engine). The engine might be useful for other database operations later.

f) The try...except block handles potential exceptions that might occur during the connection process (e.g., incorrect credentials, database server not running). If an error occurs, an error message is printed to the console, and the function returns None, None.

Data Loading (load_full_data()):

This function loads data from a specified table in the PostgreSQL database into a Pandas DataFrame:

a) It takes the database connection object (conn) and the name of the table (table_name) as arguments.

b) query = f"SELECT * FROM {table_name}": Constructs a SQL query to select all columns (*) from the specified table_name. Using an f-string makes it easy to embed the table_name into the query. In this case, the table name is hardcoded.

c) df = pd.read_sql(query, conn): Uses the read_sql() function from the pandas library to execute the SQL query against the provided database connection (conn) and load the results into a Pandas DataFrame (df).

d) print(f"📊 Successfully loaded {len(df):,} records from {table_name}"): If the data loading is successful, a message indicating the number of records loaded and the table name is printed to the console.

e) return df: The function returns the DataFrame containing the loaded data.

f) The try...except block handles potential exceptions that might occur during the data loading process (e.g., the table not existing, insufficient permissions). If an error occurs, an error message is printed, and the function returns None.

Outlier Detection (detect_outliers()):

This function implements a common method for detecting outliers in a numerical Pandas Series using the Interquartile Range (IQR):

a) It takes a Pandas Series (series) as input, which is expected to be a column of numerical data.

b) Q1 = series.quantile(0.25): Calculates the first quartile (25th percentile) of the data in the series.

c) Q3 = series.quantile(0.75): Calculates the third quartile (75th percentile) of the data in the series.

d) IQR = Q3 - Q1: Computes the Interquartile Range, which is the difference between the third and first quartiles. The IQR represents the spread of the middle 50% of the data.

e) return series[(series < (Q1 - 1.5 * IQR)) | (series > (Q3 + 1.5 * IQR))]: This line uses boolean indexing to filter the original series and return a new Series containing only the values that are considered outliers. A data point is considered an outlier if it is less than Q1 - 1.5 * IQR (below the lower bound) or greater than Q3 + 1.5 * IQR (above the upper bound). The 1.5 multiplier is a common convention for this method.

Main Analysis Function (analyze_full_dataset()):

This function performs the core data profiling steps and adds the results to the Word document (docx). It takes the loaded Pandas DataFrame (df) and the Word document object as input.

a) Initial Report Information:

   i. It adds a main section heading "COMPREHENSIVE DATA PROFILE" and basic information like the total number of records and the list of columns to the Word document.

b) Skipping ID Columns:

   i. It identifies columns that have "id" in their name (case-insensitive) and stores them in id_cols.

   ii. If any ID columns are found, it adds a message to the report indicating which columns are being skipped and then drops these columns from the DataFrame to avoid analyzing them.

c) Basic Metadata:

   i. It adds a section heading "BASIC METADATA".

   ii. It creates a Pandas DataFrame metadata containing:

      a. Data Type: The data type of each column in the DataFrame.

      b. Missing Values: The number of missing (NaN) values in each column.

      c. Missing %: The percentage of missing values in each column, rounded to two decimal places.

      d. Unique Values: The number of unique values in each column.

   iii. It then adds this metadata DataFrame to the Word document as a table using add_to_report() with content_type='table'.

d) Advanced Quality Checks:

It adds a section heading "ADVANCED QUALITY CHECKS".

   i. Duplicates:

      a. df.duplicated().sum(): Calculates the total number of duplicate rows in the DataFrame.

      b. It adds the count and percentage of duplicate rows to the report.

      c. If duplicates exist, it adds a message and displays a sample of the duplicate rows (up to 3) to the report. df.duplicated(keep=False) marks all occurrences of duplicate rows as True, and .sample(min(3, duplicates)) selects a random sample.

   ii.Outliers**:**

      a. It selects numerical columns from the DataFrame using df.select_dtypes(include=np.number).

      b. If there are numerical columns, it adds an "Outlier Report" heading.

      c. It iterates through each numerical column:

i. detect_outliers(df[col]): Calls the detect_outliers() function to get a Series of outlier values for the current column.

ii. It appends a dictionary containing the column name, the number of outliers, the percentage of outliers, and the minimum and maximum outlier values (if any) to the outlier_report list.

iii. Finally, it converts the outlier_report list into a Pandas DataFrame and adds it to the Word document as a table.

e) Constant Features:

i. It identifies columns where the number of unique values is 1.

ii. It adds a message indicating the constant features found (or "None found").

f) High-Cardinality Features:

i. It identifies categorical (object or category dtype) columns where the number of unique values is greater than 50.

ii. It adds a message indicating the high-cardinality features found (or "None found").

g) Numerical Analysis:

If there are numerical columns:

i. It adds a section heading "NUMERICAL FEATURES".

ii. df[num_cols].describe().T: Calculates descriptive statistics (count, mean, std, min, 25th percentile, median, 75th percentile, max) for all numerical columns and transposes the result so that each row represents a column.

iii. It adds 'skewness' and 'kurtosis' to the descriptive statistics DataFrame using .skew() and .kurt() respectively.

iv. It also calculates the number of outliers for each numerical column using the detect_outliers() function and adds it as an 'outliers' column to the num_stats DataFrame.

v. This extended num_stats DataFrame is added to the Word document as a table.

vi. It then iterates through each numerical column and generates a pair of plots:

a. A histogram (sns.histplot) with a Kernel Density Estimate (KDE) to visualize the distribution of the data.

b. A box plot (sns.boxplot) to show the spread and identify potential outliers visually.

c. Each plot is added to the Word document as a figure using add_to_report(). plt.gcf() gets the current figure object.

h) Categorical Analysis:

i. If there are categorical columns

a. It adds a section heading "CATEGORICAL FEATURES".

b. It iterates through each categorical column:

i. It adds the column name as a subheading.

ii. df[col].value_counts(dropna=False): Calculates the frequency of each unique value in the column, including missing values.
iii. df[col].value_counts(normalize=True, dropna=False).mul(100).round(2): Calculates the percentage of each unique value, including missing values, rounded to two decimal places.
iv. It creates a DataFrame cat_stats with the counts and percentages of each category and adds it to the Word document as a table.
v. It then generates a bar plot to visualize the distribution of categories:
    1. If the number of unique categories is greater than 20, it plots only the top 20 most frequent categories using sns.barplot.
    2. Otherwise, it plots all categories using sns.countplot.
vi. The generated plot is added to the Word document as a figure.

i) Temporal Analysis:
   i. It identifies columns whose names contain keywords like "date", "time", or "year" (case-insensitive) as potential temporal columns.
   ii. If temporal columns are found:
      a. It adds a section heading "TEMPORAL ANALYSIS".
      b. It iterates through each identified column:
         i. It attempts to convert the column to datetime objects using pd.to_datetime().
         ii. It adds information about the time range (min and max dates) and the number and percentage of missing values to the report.
         iii. It then generates a time series plot to visualize the distribution of the temporal data over time:
            1. If the time range is more than one year, it plots the monthly distribution.
            2. Otherwise, it plots the daily distribution.
         iv. The plot is added to the Word document as a figure.
         v. A try...except block is used to handle potential errors during datetime conversion or analysis.
   iii. If no temporal columns are found, a message is added to the report.
j) Correlation & Relationship Analysis:
   i. It adds a section heading "CORRELATION ANALYSIS".
   ii. Numerical Correlations:
      a. If there is more than one numerical column:
         i. It calculates the Pearson correlation matrix between all numerical columns using df[num_cols].corr().
         ii. It generates a heatmap of the correlation matrix using sns.heatmap with annotations and a 'coolwarm' colormap. The heatmap is added to the Word document as a figure.
         iii. It then extracts the top 10 most strongly correlated pairs of features (excluding self-correlations) from the upper triangle of the correlation matrix and adds them to the report as a table.
   iii. Categorical-Numerical Relationships:
      a. If there are both numerical and categorical columns:
         i. It iterates through the first 3 numerical columns and the first 3 categorical columns (for brevity).
         ii. For each combination where the categorical column has fewer than 10 unique values (to avoid overly complex box plots), it generates a box plot using

sns.boxplot to visualize the distribution of the numerical variable for each category of the categorical variable.

iii. Each box plot is added to the Word document as a figure.

k) Main Execution Block (if __name__ == "__main__":)

This block of code is executed when the script is run directly:

i. doc = create_word_report(): Initializes the Word document by calling the create_word_report() function.

ii. conn, engine = connect_to_postgres(): Establishes a connection to the PostgreSQL database.

iii. if conn:: Checks if the connection was successful.

a. table_name = "synthetic_dswb_training.source_dataset_fake": Defines the name of the table to load data from. Make sure this table exists in your PostgreSQL database.

b. df = load_full_data(conn, table_name): Loads the data from the specified table into a Pandas DataFrame.

c. if df is not None:: Checks if the data loading was successful.

i. analyze_full_dataset(df, doc): Calls the main analysis function, passing the loaded DataFrame and the Word document object. This performs all the data analysis and adds the results to the Word document.

d. conn.close(): Closes the database connection to release resources.

e. engine.dispose(): Disposes of the SQLAlchemy engine.

f. print("\n ✅ Analysis completed successfully"): Prints a success message to the console.

iv. report_path = "D:/Platform/Source_Data_Profile_Report.docx": Defines the path where the generated Word report will be saved. You should adjust this path to your desired location.

v. doc.save(report_path): Saves the content of the doc object to a Microsoft Word file at the specified report_path.

vi. print(f" 📄 Report saved to: {report_path}"): Prints a message indicating where the report has been saved.

Summary

In summary, this connects to a PostgreSQL database, loads a specific table, performs a wide range of data profiling tasks (including metadata inspection, quality checks, numerical and categorical analysis, temporal analysis, and correlation analysis), visualizing the data using Matplotlib and Seaborn, and finally, compiling all the analysis results, tables, and plots into a comprehensive Microsoft Word report.