Data Analysis Pipeline Documentation

Pramit, Shraddha, Safna

December 17, 2024

Contents

1 Introduction

This document provides a detailed explanation of a Python-based data analysis pipeline. It includes step-by-step descriptions of the functions used, their arguments, and outputs.

2 Setup

Before using the pipeline, ensure you have the required libraries installed:

```python
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 from scipy.stats import spearmanr, ttest_ind
6 from scipy.signal import savgol_filter
7 from sklearn.decomposition import PCA
```

Explanation:

- **Line 1:** `import numpy as np` The `NumPy` library is imported and aliased as `np`. It provides support for numerical computations and efficient handling of arrays and matrices, such as `np.array()` and `np.mean()`.
- **Line 2:** `import pandas as pd` The `Pandas` library is imported and aliased as `pd`. It is used for handling structured data, particularly for creating and manipulating `DataFrame` objects and performing operations like filtering, aggregation, and transformation.
- **Line 3:** `import matplotlib.pyplot as plt` The `pyplot` module of the `Matplotlib` library is imported and aliased as `plt`. This module is used for data visualization, such as creating scatter plots, line graphs, and bar charts.
- **Line 4:** `import seaborn as sns` The `Seaborn` library is imported as `sns`. It extends `Matplotlib` to simplify the creation of statistically informative visualizations like heatmaps, pair plots, and violin plots.
- **Line 5:** `from scipy.stats import spearmanr, ttest_ind` Specific functions from `scipy.stats` are imported:
  - `spearmanr`: Calculates the Spearman rank correlation, useful for measuring the monotonic relationship between two variables.
  - `ttest_ind`: Performs an independent two-sample t-test to compare the means of two datasets.
- **Line 6:** `from scipy.signal import savgol_filter` The `Savitzky-Golay filter` is imported from `scipy.signal`. This filter is used for smoothing data by fitting successive polynomial curves, preserving important trends while reducing noise.
- **Line 7:** `from sklearn.decomposition import PCA` The `PCA` (Principal Component Analysis) module from `Scikit-learn` is imported. PCA is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional space while preserving as much variance as possible.

3 Function Definitions

3.1 Reading CSV Files

```python
def processCsv(filepath):
    """
    Reads a CSV file and returns a pandas DataFrame.

    Args:
        filepath (str): The path to the CSV file.

    Returns:
        DataFrame: The loaded dataset as a pandas DataFrame.
    """
    try:
        df = pd.read_csv(filepath)
    except FileNotFoundError:
        print(f"File not found: {filepath}")
        return None
    except Exception as e:
        print(f"Error reading CSV file: {filepath}")
        print(e)
        return None
    return df
```

Explanation:

- **Function Definition:** `def processCsv(filepath):` This defines a function named `processCsv` that takes a single parameter, `filepath`, which is a string representing the path to the CSV file.

- **Docstring:** The docstring provides a description of the function's purpose, arguments, and return value.

  - **Args:** `filepath (str)`: Specifies the path to the CSV file to be read.
  - **Returns:** `DataFrame`: A pandas DataFrame containing the data from the CSV file.

- **Try Block:** The `try` block attempts to execute the code to read the CSV file:

  - `df = pd.read_csv(filepath)`: This uses `pandas.read_csv()` to load the CSV file into a pandas DataFrame.

- **Exception Handling:** The `except` blocks handle errors that may occur while reading the file:

  - `FileNotFoundError`: If the file at the specified path does not exist, an error message is printed: `"File not found: {filepath}"`, and `None` is returned.
  - `Exception as e`: Captures any other exceptions, prints an error message, and returns `None`.

- **Return Statement:** If no exception occurs, the loaded DataFrame `df` is returned as the output of the function.

3.2 User Input for Wavelength Range

```
def getWavelengthRange():
    """
    Prompt the user to input the desired wavelength range for segmentation.

    Returns:
        tuple: A tuple containing the lower and upper bounds of the wavelength range.
    """
    while True:
        try:
            lower_bound = float(input("Enter the lower bound of the wavelength range: "))
            upper_bound = float(input("Enter the upper bound of the wavelength range: "))
            print("Lower bound:", lower_bound)
            print("Upper bound:", upper_bound)
            if lower_bound >= upper_bound:
                print("Lower bound must be less than the upper bound. Please try again.")
            else:
                return lower_bound, upper_bound
        except ValueError:
            print("Invalid input. Please enter numeric values.")
```

Explanation:

- **Function Definition:** `def getWavelengthRange():` This defines a function named `getWavelengthRange` that prompts the user for input and returns a tuple representing the lower and upper bounds of a wavelength range.

- **Docstring:** The docstring provides a brief overview of the function's purpose and its return value.

  - **Returns:** `tuple`: A tuple containing two floating-point values (`lower_bound`, `upper_bound`) that represent the wavelength range.

- **While Loop:** `while True:` This loop ensures the user is continuously prompted until valid input is provided. It keeps the program running until the user enters acceptable values.

- **Try Block:** The `try` block handles user input and potential errors:

  - `lower_bound = float(input(...))`: Prompts the user to input the lower bound of the wavelength range, converting the input to a float.

  - `upper_bound = float(input(...))`: Prompts the user to input the upper bound of the wavelength range, converting the input to a float.

- **Validation Check:** `if lower_bound >= upper_bound:` Ensures that the lower bound is less than the upper bound. If not, an error message is printed:

  - "Lower bound must be less than the upper bound. Please try again."

- **Successful Input:** `return lower_bound, upper_bound` If the inputs are valid and the lower bound is less than the upper bound, the function returns a tuple containing the two values.
- **Exception Handling:** `except ValueError:` If the user inputs a non-numeric value, a `ValueError` is raised. An error message is displayed:
  - "Invalid input. Please enter numeric values."

3.3 Dataset Preparation

```
def prepareDataset(df, filepath):
    """
    Cleans and preprocesses the dataset by renaming columns,
       filtering rows, and converting data types.

    Args:
        df (DataFrame): The raw dataset.
        filepath (str): The path to the CSV file (used for error
           logging).

    Returns:
        DataFrame: A cleaned and preprocessed dataset.
    """
    print("Data cleaning....")
    column_names = list(df.iloc[0])  # Extract column names from
       the first row
    column_names[0] = "Wavelength (nm)"  # Rename the first
       column for clarity
    filtered_data = df[1:].copy()  # Exclude the first row used
       for column names
    filtered_data.columns = column_names  # Assign column names

    # Convert "Wavelength (nm)" to numeric
    try:
        filtered_data["Wavelength (nm)"] = pd.to_numeric(
           filtered_data["Wavelength (nm)"], errors='coerce')
    except Exception as e:
        print(f"Error converting 'Wavelength (nm)' to numeric in
           file: {filepath}")
        print(e)
        return None

    # Drop rows with invalid "Wavelength (nm)" values
    filtered_data = filtered_data.dropna(subset=["Wavelength (nm)
       "])

    # Get wavelength range from the user
```

```python
    print("Please specify the wavelength range for segmentation.")
    lower_bound, upper_bound = getWavelengthRange()

    # Filter data within the user-specified range
    filtered_data = filtered_data[(filtered_data["Wavelength (nm)"] >= lower_bound) &
                                  (filtered_data["Wavelength (nm)"] <= upper_bound)]

    # Convert absorption columns to numeric
    valid_columns = ["Wavelength (nm)"]  # Keep track of valid columns
    for col in filtered_data.columns[1:]:
        try:
            filtered_data[col] = pd.to_numeric(filtered_data[col], errors='coerce')
            valid_columns.append(col)
        except Exception as e:
            print(f"Skipping invalid column '{col}' in file: {filepath}")
            print(e)

    # Keep only valid columns
    filtered_data = filtered_data[valid_columns]
    print("Data is prepared now!")
    return filtered_data
```

Explanation:

- **Function Definition:** `def prepareDataset(df, filepath):` This function cleans and preprocesses a dataset provided as a `pandas DataFrame`, using the file path for logging purposes in case of errors.

- **Docstring:** The docstring explains the function's purpose, arguments, and return value.

  - **Args:** `df (DataFrame)`: The raw dataset to be cleaned and processed. `filepath (str)`: The path to the CSV file, used for error messages.
  - **Returns:** `DataFrame`: A cleaned and preprocessed dataset.

- **Column Renaming and Filtering:**

  - `column_names = list(df.iloc[0])`: Extracts column names from the first row of the dataset.
  - `column_names[0] = "Wavelength (nm)"`: Renames the first column for clarity.
  - `filtered_data = df[1:].copy()`: Excludes the first row, which contains the column names, and creates a copy of the remaining data.
  - `filtered_data.columns = column_names`: Assigns the extracted column names to the dataset.

- **Numeric Conversion of Wavelength Column:**
  - `filtered_data["Wavelength (nm)"] = pd.to_numeric(...)`: Converts the "Wavelength (nm)" column to numeric, coercing invalid values to `NaN`.
  - If an error occurs during conversion, the exception is caught, and an error message is printed with the file path.
- **Dropping Invalid Rows:**
  - `filtered_data = filtered_data.dropna(subset=["Wavelength (nm)"])`: Removes rows where "Wavelength (nm)" contains `NaN`.
- **Wavelength Range Filtering:**
  - The `getWavelengthRange()` function is called to prompt the user for the lower and upper bounds of the wavelength range.
  - `filtered_data = filtered_data[(filtered_data["Wavelength (nm)"] >= lower_bound) & (filtered_data["Wavelength (nm)"] <= upper_bound)]`: Filters the dataset to include only rows within the specified range.
- **Numeric Conversion of Absorption Columns:**
  - A list `valid_columns` is initialized with `"Wavelength (nm)"`.
  - For each column other than "Wavelength (nm)", the function attempts to convert it to numeric using `pd.to_numeric(...)`.
  - If a column cannot be converted, an error message is printed, and the column is skipped.
- **Keeping Only Valid Columns:**
  - `filtered_data = filtered_data[valid_columns]`: Keeps only the valid columns in the final dataset.
- **Return Statement:**
  - If all operations are successful, the cleaned and preprocessed dataset is returned.
- **Output Messages:**
  - Messages like `"Data cleaning...."` and `"Data is prepared now!"` provide feedback to the user during processing.

3.4 Statistical calculation and graph plot for the dataset

```python
def statForDataset(df):
    """
    Computes and displays statistical summaries for the dataset.

    Args:
        df (DataFrame): The dataset to analyze.
```

```
    """
    print("Dataset Statistics:")
    # Compute basic statistics for all columns except the wavelength
    data_no_wavelength = df.iloc[:, 1:]  # Exclude the first column (wavelength)

    # Compute statistics
    statistics = {
    "Mean": data_no_wavelength.mean(),
    "Median": data_no_wavelength.median(),
    "Standard Deviation": data_no_wavelength.std(),
    "Min": data_no_wavelength.min(),
    "Max": data_no_wavelength.max(),
    "Range": data_no_wavelength.max() - data_no_wavelength.min(),
    }

    # Print each statistic
    for stat_name, stat_values in statistics.items():
        print(f"\n{stat_name}:")
        print(stat_values.to_string(index=True))

    # Plotting
    plt.figure(figsize=(15, 8))  # Increased figure size

    # Create subplots for each statistic
    for i, stat_name in enumerate(statistics, 1):
        plt.subplot(3, 3, i)
        statistics[stat_name].plot(kind='bar',
                                   color='skyblue',
                                   edgecolor='black')
        plt.title(f'{stat_name} Across Columns', fontsize=12)
        plt.xticks(rotation=45, ha='right', fontsize=8)
        plt.ylabel(stat_name, fontsize=10)
        plt.grid(axis='y', linestyle='--', alpha=0.7)

    plt.suptitle('Statistical Measures Comparison', fontsize=16)
    plt.tight_layout()
    plt.show()
```

Explanation:

- **Function Definition and Purpose:**
  - Function `statForDataset(df)` is designed to perform comprehensive statistical analysis on a pandas DataFrame
  - Takes a DataFrame as input and excludes the first column (assumed to be wavelength)
- **Key Code Analysis:**
  - `df.iloc[:, 1:]`: Slices DataFrame, removing first column

- Statistical computations use pandas built-in methods:
    - `.mean()`: Calculates column-wise average
    - `.median()`: Finds middle value of each column
    - `.std()`: Computes standard deviation
    - `.min()/.max()`: Determines minimum/maximum values

- **Statistical Dictionary Creation:**
  - Creates `statistics` dictionary with multiple statistical measures
  - Enables flexible, comprehensive data exploration
  - Computes range using `max() - min()` subtraction

- **Visualization Approach:**
  - Uses matplotlib for creating subplots
  - `plt.subplot(3, 3, i)`: Creates 3x3 grid of statistical visualizations
  - Bar plot with customized aesthetics:
    - Color: skyblue
    - Edge color: black
    - Rotated x-axis labels for readability

3.5 For visualizing the wavelength vs absorption data

```python
def visualizeTrends(df):
    """
    Plots the absorption trends for each column in the dataset.

    Args:
        df (DataFrame): The dataset containing wavelength and
           absorption data.
    """
    plt.figure(figsize=(15, 8))
    for col in df.columns[1:]:
        plt.plot(df["Wavelength (nm)"], df[col], label=col)
    plt.title("Absorption Trends")
    plt.xlabel("Wavelength (nm)")
    plt.ylabel("Absorption")
    plt.legend()
    plt.grid()
    plt.show()
```

Explanation:

- **Function Signature:** `visualizeTrends(df)` A specialized data visualization function designed to plot absorption trends across multiple spectroscopic measurements.

- **Function Objectives:**

  - Generate comprehensive absorption trend visualization
  - Display multiple dataset columns simultaneously
  - Provide comparative spectral analysis

- **Visualization Methodology:**
  - **Matplotlib Configuration:**
    * `plt.figure(figsize=(15, 8))`: Large visualization canvas
    * Enables detailed, high-resolution plotting
  - **Plotting Strategy:**
    * Iterates through all columns except first (wavelength)
    * `df.columns[1:]`: Selects absorption data columns
    * Uses `plt.plot()` for line representation

- **Graphical Elements:**
  - X-axis: Wavelength (nm)
  - Y-axis: Absorption intensity
  - Automatic legend generation
  - Grid lines for enhanced readability

3.6 Pattern detection using monotonicity and correlations

```python
def patternDetection(df):
    """
    Analyzes patterns using monotonicity (Spearman correlation)
       and visualizes inter-column correlations.

    Args:
        df (DataFrame): The dataset for analysis.
    """
    print("Monotonicity and Correlation Analysis:")

    # Compute monotonicity using Spearman's rank correlation
    monotonicity_results = {}
    for col in df.columns[1:]:
        monotonicity, _ = spearmanr(df["Wavelength (nm)"], df[col
           ])
        monotonicity_results[col] = monotonicity
        print(f"Monotonicity (Spearman's rho) for {col}: {
           monotonicity:.2f}")

    # Plot correlation heatmap
    corr_df = df.iloc[:, 1:] # drop the 'Wavelength (nm)'
    correlations = corr_df.corr(method='spearman')
    plt.figure(figsize=(15, 8))
    sns.heatmap(correlations, annot=True, cmap="coolwarm", vmin
       =-1, vmax=1)
```

```
    plt.title("Correlation Heatmap")
    plt.show()
```

Explanation

- **Function Signature:** `patternDetection(df)` A sophisticated statistical analysis function designed to detect and visualize complex data patterns using advanced correlation techniques.
- **Analytical Objectives:**
  - Compute monotonicity using Spearman's rank correlation
  - Visualize inter-column correlational relationships
  - Provide comprehensive pattern detection insights
- **Monotonicity Analysis:**
  - **Computational Approach:**
    * Uses `spearmanr()` for non-linear correlation detection
    * Compares wavelength against each absorption column
    * Captures monotonic trends independent of linear relationships
  - **Correlation Interpretation:**
    * Range: -1 to 1
    * 0: No monotonic relationship
    * Close to ± 1: Strong monotonic trend
- **Visualization Techniques:**
  - **Correlation Heatmap:**
    * Uses seaborn's `heatmap()` for visualization
    * Color scheme: "coolwarm" (-1 to 1 range)
    * Annotated correlation coefficients
  - **Graphical Configuration:**
    * `figsize=(15, 8)`: Large, detailed visualization
    * Comprehensive inter-column correlation representation

3.7 Data smoothing

```
def dataSmoothing(df):
    """
    Applies Savitzky-Golay filtering to smooth absorption data.

    Args:
        df (DataFrame): The dataset to smooth.

    Returns:
        DataFrame: The smoothed dataset.
```

```
    """
    smoothed_data = df.copy()
    # Exclude the first column from smoothing
    for col in smoothed_data.columns[1:]:
        smoothed_data[col] = savgol_filter(smoothed_data[col],
           window_length=11, polyorder=3)
    if len(smoothed_data.columns) > 1:
        second_column_name = smoothed_data.columns[1]
        filtered_data = smoothed_data.drop(columns=[
           second_column_name])
        print(f"Dropped second column: {second_column_name}")
    visualizeTrends(filtered_data)
    return smoothed_data
```

Explanation

- **Function Signature:** `dataSmoothing(df)` A sophisticated signal processing function designed to apply advanced noise reduction techniques on spectroscopic datasets.
- **Smoothing Methodology:**
  - **Algorithm:** Savitzky-Golay Filter
    - Preserves higher-order moments
    - Maintains signal characteristics
    - Reduces random noise
  - **Filter Parameters:**
    - `window_length=11`: Sliding window size
    - `polyorder=3`: Polynomial approximation order
    - Balances noise reduction and signal preservation
- **Implementation Strategy:**
  - Creates deep copy of input DataFrame
  - Applies smoothing to all columns except wavelength
  - Uses `savgol_filter()` for computational efficiency
- **Post-Processing Actions:**
  - Conditional column dropping mechanism
  - Calls `visualizeTrends()` for result visualization
  - Returns smoothed dataset for further analysis

3.8 T-test relation between balnks and samples

```
def testRelationBetweenBlanksAndSamples(df):
    """
    Performs t-tests to compare blanks and samples for
       statistical differences.

```

```python
    Args:
        df (DataFrame): The dataset containing blanks and sample data.
    """

    # Explicitly identify blank and sample columns based on known naming patterns
    blank_columns = [col for col in df.columns if col.startswith("blank")]
    sample_columns = [col for col in df.columns if col.endswith("mg")]

    if not blank_columns or not sample_columns:
        print("No blank or sample columns found in the dataset.")
        return

    for blank_col in blank_columns:
        for sample_col in sample_columns:
            try:
                blank_data = df[blank_col].dropna()
                sample_data = df[sample_col].dropna()

                if blank_data.empty or sample_data.empty:
                    print(f"No data to compare between {blank_col} and {sample_col}.")
                    continue

                stat, p_value = ttest_ind(blank_data, sample_data)
                print(f"T-test between {blank_col} and {sample_col}: p-value = {p_value:.4f}")
            except Exception as e:
                print(f"Error performing t-test between {blank_col} and {sample_col}: {e}")
```

Explanation

- **Function Signature:** `testRelationBetweenBlanksAndSamples(df)` A comprehensive statistical analysis function designed to compare blank and sample measurements using independent t-tests.
- **Column Identification Strategy:**
  - **Blank Column Detection:**
    * Uses prefix-based identification
    * `col.startswith("blank")`: Identifies blank columns
  - **Sample Column Detection:**
    * Uses suffix-based identification
    * `col.endswith("mg")`: Identifies sample columns
- **Statistical Testing Methodology:**

- **T-Test Configuration:**
    - Independent two-sample t-test
    - Compares means between blank and sample datasets
    - Assesses statistical significance
- **Data Preparation:**
    - Removes NaN values using `.dropna()`
    - Ensures valid data for comparison

- **Comparative Analysis Approach:**
  - Nested loop for comprehensive cross-comparison
  - Pairwise t-test between all blank and sample columns
  - Prints p-values for statistical interpretation

- **Error Handling Mechanisms:**
  - Checks for empty datasets
  - Catches and reports exceptions during testing
  - Provides informative error messages

3.9 Principal Component Analysis (PCA)

```python
def applyPCA(df):
    """
    Applies Principal Component Analysis (PCA) to reduce
       dimensionality and visualize data in 2D.

    Args:
        df (DataFrame): The dataset for PCA.

    Displays:
        Scatter plot of the first two principal components.
    """
    print("Applying PCA...")
    pca = PCA(n_components=2)
    absorption_data = df.iloc[:, 1:].values
    principal_components = pca.fit_transform(absorption_data)
    pca_df = pd.DataFrame(principal_components, columns=["PC1", "
       PC2"])
    print("PCA applied!And we are getting this scatter graph.")
    plt.figure(figsize=(12, 8))
    plt.scatter(pca_df["PC1"], pca_df["PC2"])
    plt.title("PCA of Absorption Data")
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.grid()
    plt.show()
```

Explanation

- **Function Signature:** `applyPCA(df)` A dimensionality reduction function utilizing Principal Component Analysis (PCA) for spectroscopic data visualization.
- **Dimensionality Reduction Strategy:**
  - **PCA Configuration:**
    * `n_components=2`: Reduces to 2-dimensional space
    * Captures maximum variance in minimal dimensions
  - **Data Preparation:**
    * `df.iloc[:, 1:]`: Excludes wavelength column
    * Converts to numerical numpy array
- **Computational Workflow:**
  - Applies `fit_transform()` for PCA computation
  - Generates principal component DataFrame
  - Creates two-dimensional representation
- **Visualization Technique:**
  - **Scatter Plot Configuration:**
    * X-axis: First Principal Component
    * Y-axis: Second Principal Component
    * Grid for enhanced readability

3.10 Get valid segment count

```python
def getValidSegmentCount():
    """
    Prompt user to enter the number of segments with input
       validation.

    Returns:
        int: Number of segments to process
    """
    while True:
        try:
            segment_count = int(input("How many segments would
               you like to analyze? (1-5): "))

            # Validate segment count
            if 1 <= segment_count <= 5:
                return segment_count
            else:
                print("Please enter a number between 1 and 5.")

        except ValueError:
            print("Invalid input. Please enter a valid integer.")
```

Explanation

- **Function Signature:** `getValidSegmentCount()` An interactive input validation function designed to collect user-specified segment count with robust error handling.
- **Input Validation Strategy:**
  - **Input Mechanism:**
    - Uses `input()` for user interaction
    - Prompts for segment count between 1-5
  - **Validation Techniques:**
    - Infinite `while` loop for continuous prompting
    - `int()` conversion for numeric validation
    - Range check: $1 \leq segment\_count \leq 5$
- **Error Handling Mechanisms:**
  - Catches `ValueError` for non-integer inputs
  - Provides user-friendly error messages
  - Ensures valid integer input
- **Return Behavior:**
  - Returns validated segment count
  - Exits loop upon successful input

4 Main Execution Flow

The main script uses the defined functions to analyze the dataset and produce visualizations:

```python
if __name__ == "__main__":
    filepath = '/content/set01.csv'
    df = processCsv(filepath)

    if df is not None:
        # Get number of segments from user
        num_segments = getValidSegmentCount()

        # Lists to store segment data
        segments = []
        smoothed_segments = []

        # Process each segment
        for i in range(num_segments):
            print(f"\nEnter details for segment {i+1}:")
            current_segment = prepareDataset(df, filepath)

            if current_segment is not None:
```

```
                segments.append(current_segment)

                # Visualization
                print(f"Visualizing segment {i+1}...")
                visualizeTrends(current_segment)

                # Statistical Analysis
                print(f"Statistical info on segment {i+1}...")
                statForDataset(current_segment)

                # Pattern Detection
                print(f"Applying pattern detection on segment {i+1}...")
                patternDetection(current_segment)

                # Smoothing
                print(f"Smoothing segment {i+1}...")
                smoothed_segment = dataSmoothing(current_segment)
                smoothed_segments.append(smoothed_segment)

                # Re-apply Pattern Detection on Smoothed Data
                print(f"Re-applying pattern detection on smoothed segment {i+1}...")
                patternDetection(smoothed_segment)

                # Test Blanks and Samples Relation
                print(f"Testing relation between blanks and samples for segment {i+1}...")
                testRelationBetweenBlanksAndSamples(current_segment)

                # PCA
                print(f"Applying PCA on segment {i+1}...")
                applyPCA(current_segment)
```

Explanation

- **Main Execution Flow:** A comprehensive data analysis pipeline designed for multi-segment spectroscopic data processing.
- **Initialization Stage:**
  - **Data Loading:**
    - Uses `processCsv()` for initial data import
    - Validates successful CSV processing
  - **Segment Configuration:**
    - Calls `getValidSegmentCount()` for user input
    - Determines number of segments to analyze
- **Segment Processing Workflow:**

- **Iterative Analysis:**
  - Processes each segment sequentially
  - Applies multiple analytical techniques
- **Analysis Techniques:**
  - Data Preparation
  - Trend Visualization
  - Statistical Analysis
  - Pattern Detection
  - Data Smoothing
  - Blank-Sample Relationship Testing
  - Principal Component Analysis

- **Data Management:**
  - Maintains separate lists for:
    - Original segments
    - Smoothed segments