Description for Cell 1

This cell contains the following command:

```
!which python
```

Description:

This command is used in Jupyter Notebooks to determine the path of the Python interpreter being used in the current environment. The `!` at the beginning indicates that it is a shell command. This is useful to verify which Python environment (e.g., system Python, virtual environment, or Anaconda) is being used to execute the notebook.

Description for Cell 2

This cell defines and executes a function to interactively get file paths from the user.

Function Definition: `get_file_paths()`:

- Prompts the user to input file paths for processing.
- Accepts only files with `.png` or `.pdf` extensions.
- Allows the user to enter multiple file paths, pressing **Enter** to confirm each.
- Terminates the input process when the user presses **Enter** without providing a file path.

Global Variable: `files_to_process`:

The list of file paths provided by the user is stored in the global variable `files_to_process`, making it accessible throughout the notebook.

User Feedback:

- If valid file paths are provided, they are printed to confirm the selection.
- If no files are selected, a message informs the user that no files were provided.

This cell primarily serves as a user-friendly interface for gathering input files for further processing in the notebook.

Description for Cell 3

This cell provides documentation and instructions for using the notebook.

Overview:

- Describes the notebook's purpose: processing PNG and PDF files.
- Explains that PDFs are converted to PNG images (one for each page) before processing.

Workflow:

- **Input Files:** Users can upload PNGs directly or PDFs to be converted to PNGs.
- **PDF to PNG Conversion:** Uses the `pdf2image` library to convert PDF pages into PNGs, which are saved in a `converted_files` folder with descriptive filenames. The converted files are automatically added to the `files_to_process` list.
- **Processing:** Combines all PNGs, regardless of origin, for uniform processing.

Prerequisites:

- Lists required Python libraries (`pdf2image`, `opencv-python-headless`, `numpy`) and tools (Poppler for PDF conversion), with platform-specific installation instructions for Poppler.

Execution Steps:

1. Cell 1: Input file paths.
2. Cell 2: Convert PDFs to PNGs and prepare all file paths for processing.
3. Cell 3: Processes the files.

Notes:

- Highlights the need for correct file paths.
- Mentions error logging and the location of processed output files.

This cell serves as a comprehensive guide for users, ensuring they understand the workflow, dependencies, and usage of the notebook.

Description for Cell 4

This cell defines functions for handling PDF to PNG conversion and processing multiple PDFs. It also integrates with the global workflow by updating the `files_to_process` list.

```
\subsection*{Function: \texttt{convert\_pdf\_to\_png(file\_path, dpi=900)}:}
\begin{itemize}
    \item Converts a single PDF into PNG images at 900 DPI.
    \item Saves each page of the PDF as a separate PNG in a \texttt{converted\_files}
    directory located next to the original PDF file.
    \item Uses a descriptive naming convention for the output images: \texttt{<original\
    _filename>\_convertedPNG\_pg<page\_number>.png}.
    \item Logs messages for each saved file or errors encountered during the conversion.
\end{itemize}
```

Function: `process_pdfs(file_paths)`:

- Processes multiple PDF files provided as a list.
- Calls `convert_pdf_to_png` for each file to handle the conversion.
- Updates the global `files_to_process` list with the paths of all generated PNGs for further processing.
- Provides error messages for missing or inaccessible files.

Integration with `files_to_process`:

- Filters the global `files_to_process` list to extract PDF files.
- Converts the filtered PDFs into PNGs and updates the list with the resulting file paths.
- Checks if `files_to_process` is defined; if not, prompts the user to run Cell 1 first.

Purpose:

This cell is essential for preparing PNG images from PDF files, ensuring that the workflow can seamlessly process both file types. It incorporates error handling and logging to improve robustness.

Cell 5 Description

This cell defines functions to process PNG images, focusing on identifying and filtering specific visual elements, and then generates temporary processed images.

Function: `process_image(image_path)`

Processes a single PNG image to filter and highlight contours (e.g., flowchart components).

- **Steps:**
  - Loads the image using OpenCV.
  - Converts the image to grayscale and applies binary thresholding.
  - Detects contours in the binary image.
  - Filters out unwanted contours based on size and shape criteria (e.g., ignoring text-like regions and small lines).
  - Draws filtered contours onto a blank canvas.
  - Saves the processed image (`temp.png`) in a `processed_files` folder near the original file.
  - Includes error handling for invalid or missing images.

Function: `process_files(file_paths)`

Processes multiple PNG files by calling `process_image` for each.

- Filters the input file list to include only PNG files.
- Logs warnings for unsupported or missing files.
- Collects the paths of all successfully processed temporary images for further steps.

Integration with `files_to_process`

- Processes all PNG files listed in the global `files_to_process`.
- Outputs the paths of processed files for use in subsequent cells.
- Includes a check to ensure `files_to_process` is defined, prompting the user to run Cell 1 if necessary.

Purpose

This cell is the core of the image processing workflow, applying computer vision techniques to prepare and filter visual elements (e.g., flowcharts) from PNG files. It integrates seamlessly with earlier steps to process files provided or converted from PDFs.

Cell 6 Description

This cell focuses on extracting text from labelled rectangles in processed images (`temp.png`) using OCR (Optical Character Recognition) and displaying/debugging the detected regions.

Key Components

1. **Initialization of PaddleOCR:** The PaddleOCR model is initialized with English language support (`lang='en'`) and angle classification enabled for rotated text detection.
2. **Helper Function:** `is_rectangle(approx)`: Determines if a given contour approximates a rectangle based on:
   - Having 4 vertices.
   - Convexity.

- Approximate 90° internal angles (with a tolerance of ±10°).

3. **Function:** `process_image(image_path)`: Processes a single image to:
   - Detect rectangular regions in the image using edge detection (Canny) and contour approximation.
   - Extract text from these regions using OCR.
4. **Workflow:**
   - **Edge Detection:** Detects edges in the grayscale image using Canny edge detection.
   - **Contour Detection:** Finds contours and filters them to identify rectangles.
   - **Text Extraction:**
     - Crops each detected rectangle.
     - Runs OCR within the rectangle using PaddleOCR.
     - Stores the extracted text along with the rectangle's number.
5. **Output:**
   - Saves extracted text to a `.txt` file.
   - Creates a debug image with labelled rectangles and their numbers.
   - Displays the debug image with `matplotlib` for visual verification.
6. **Function:** `process_files(temp_image_paths)`: Processes multiple `temp.png` images by calling `process_image` for each.
   - Handles missing or invalid files gracefully with logging.
   - Ensures all files in `temp_image_paths` are processed sequentially.
7. **Integration with `temp_files`:**
   - Processes the `temp.png` files stored in the global `temp_files` list (generated in Cell 5).
   - Logs an error if `temp_files` is not defined, prompting the user to run the preceding cell.

Purpose

This cell automates the detection of rectangular regions in processed images and extracts text using OCR. It provides clear visual and textual outputs, aiding in debugging and ensuring the workflow's accuracy.

Cell 7 Description

This cell builds upon the previous processing steps to analyse both original images and their corresponding temporary processed images (`temp.png`). It extracts text and rectangle coordinates and outputs the results in JSON files.

Key Components

1. **Initialization of PaddleOCR:** PaddleOCR is re-initialized for extracting text from identified rectangular regions.
2. **Global List:**
   - `rectangle_json_files` is a global list to store paths of JSON files generated in this step.
3. **Helper Function:** `is_rectangle(approx)`: Checks whether a detected contour approximates a rectangle by:
   - Having 4 vertices.
   - Convexity.

- Approximately 90° internal angles (tolerance ±10°).

4. **Function:** `process_file(original_image_path, temp_image_path)`: Processes one pair of original and temporary images.
   - **Workflow:**
     - **Rectangle Detection:**
       * Detects rectangular contours in the temporary processed image using edge detection and contour approximation.
     - **Text Extraction:**
       * Crops regions corresponding to the detected rectangles from the original image.
       * Uses OCR to extract text from each region.
       * Stores text along with rectangle labels and coordinates.
     - **JSON Output:**
       * Saves rectangle text and coordinates as a JSON file in the `processed_files` directory.
       * Adds the JSON file path to the global `rectangle_json_files` list.
     - **Debug Image:**
       * Annotates the temporary image with labelled rectangles for visual verification.
       * Saves the labelled image and displays it using `matplotlib`.
     - **Console Output:**
       * Prints the extracted text and coordinates for each rectangle.

5. **Function:** `process_files(original_image_paths, temp_image_paths)`:
   - Processes multiple image pairs by calling `process_file` for each pair of original and temporary images.
   - Verifies the existence of the provided file paths before processing.
   - Handles missing files with error messages.

6. **Integration:**
   - Processes all image pairs listed in `files_to_process` (original images) and `temp_files` (temporary processed images).
   - Logs an error if required variables are not defined, prompting users to execute prior cells.

Purpose

This cell performs the final steps of the workflow, extracting text and rectangle coordinates for each image, and outputs the results in JSON format. It ensures robust error handling, provides visual verification, and prepares data for subsequent analysis or documentation.

Cell 8 Description

This cell introduces advanced processing to detect rectangles and lines in images and clean up the visual representation, preparing data for further use or debugging.

Key Components

1. **Global Variables:**
   - `processed_file_results`: Stores metadata and paths for processed images.
   - `detected_line_files`: Stores paths to images with detected lines.
   - `debug_images`: Holds debug images for visual inspection.

2. **Helper Function:** `is_rectangle(approx)`: Checks if a contour approximates a rectangle by verifying:

- Convexity.
- Four vertices with approximate 90° internal angles (tolerance ±10°).

3. **Function:** `process_file(image_path)`: Processes a single image for rectangle and line detection.
   - **Workflow:**
     - **Rectangle Detection:**
       * Detects and filters rectangular contours in the image.
       * Masks and labels detected rectangles on a debug image.
       * Generates an inverted mask to remove detected rectangles.
     - **Image Cleanup:**
       * Dilates the rectangle mask to refine boundaries.
       * Applies the refined mask to clean up the image, leaving non-rectangular elements.
     - **Line Detection:**
       * Uses the cleaned image for edge detection and applies Hough Line Transform to detect straight lines.
       * Annotates detected lines on the image.
     - **Output:**
       * Saves processed images, including:
         · Debug image with rectangles.
         · Image without rectangles.
         · Cleaned image.
         · Image with detected lines.
       * Appends metadata to `processed_file_results`.
     - **Visualization:**
       * Displays the four processed images in a grid layout using `matplotlib`.
4. **Function:** `process_files(temp_image_paths)`:
   - Iterates through a list of image paths and processes each using `process_file`.
   - Stores paths to images with detected lines in `detected_line_files`.
5. **Integration:**
   - Processes the `temp_files` list (generated in earlier cells).
   - Logs an error if `temp_files` is undefined, prompting users to execute prior cells.

Purpose

This cell enhances the processing pipeline by:

- Detecting and isolating rectangles.
- Cleaning up images by removing rectangular regions.
- Detecting and annotating straight lines.
- Preparing outputs for debugging or further analysis. The combination of robust visualization, file outputs, and comprehensive logging makes this step pivotal for ensuring accuracy in visual processing.

Cell 9 Description

This cell focuses on detecting, combining, and analyzing lines within images using Hough Line Transform, followed by saving results in a structured JSON format.

Key Components

1. **Helper Functions:**
   - `distance(p1, p2)`: Computes the Euclidean distance between two points.
   - `midpoint(p1, p2)`: Calculates the midpoint of a line segment.
   - `points_are_close(p1, p2, distance_thresh)`: Checks if two points are within a defined distance threshold.
   - `should_combine(line1, line2, distance_thresh)`: Determines whether two lines should be merged based on proximity.
   - `combine_lines(line1, line2)`: Merges two lines into one by determining the farthest start and end points.
2. **Global Variables:**
   - `json_files`: A global list storing paths of generated JSON files with line data.
3. **Function:** `process_file(original_image_path)`:
   - **Workflow:**
     - **Line Detection:**
       - Detects lines in an image using Hough Line Transform (`cv2.HoughLinesP`).
       - Annotates detected lines with labels on the image for visualization.
     - **Line Combination:**
       - Iterates through detected lines and merges lines that are close or overlap.
       - Generates a separate image displaying the combined lines.
     - **Output:**
       - Saves:
         - Image with all detected lines.
         - Image with combined lines.
       - Outputs a JSON file with:
         - Original line data (start and end points).
         - Combined line data.
     - **Visualization:** Displays both detected and combined lines using `matplotlib`.
4. **Function:** `process_files(image_paths)`:
   - Iterates through a list of image paths and calls `process_file` for each.
   - Handles missing files with error messages.
5. **Integration:**
   - Processes images listed in the global variable `detected_line_files` (generated in previous steps).
   - Prompts the user to run earlier cells if required data is unavailable.

Purpose

This cell refines the analysis by detecting and merging lines in images. It outputs both visual and JSON representations of the data, enabling detailed insights and further processing. The structured workflow and robust error handling ensure smooth integration with the overall pipeline.

Cell 10 Description

This cell serves as a debugging utility to verify the existence of files in the `temp_files` and `json_files` global variables.

Key Components

- **Check for `temp_files`:**
  - Iterates through the `temp_files` list (if defined) and checks if each file exists on the disk.
  - Logs the results:
    - `[FOUND]`: The file exists.
    - `[MISSING]`: The file is missing.
  - Logs an error if `temp_files` is not defined.
- **Check for `json_files`:**
  - Iterates through the `json_files` list (if defined) and performs the same existence check.
  - Logs the results:
    - `[FOUND]`: The file exists.
    - `[MISSING]`: The file is missing.
  - Logs an error if `json_files` is not defined.

Purpose

This cell helps ensure that the intermediate and final outputs of the pipeline (`temp_files` and `json_files`) are present and accessible. It identifies missing files for troubleshooting and improves the robustness of the workflow.

Cell 11 Description

This cell extends the line detection workflow by processing temporary images (`temp_files`) and their corresponding JSON files containing line data. It visualizes, combines, and annotates the detected lines.

Key Components

1. **Helper Functions:**
   - `distance(p1, p2)`: Computes the Euclidean distance between two points.
   - `midpoint(p1, p2)`: Finds the midpoint of a line segment.
   - `points_are_close(p1, p2, distance_thresh)`: Checks if two points are within a certain threshold.
   - `should_combine(line1, line2, distance_thresh)`: Determines whether two lines should be merged based on proximity.
   - `combine_lines(line1, line2)`: Merges two lines by finding the farthest points that define the new line.
2. **Function:** `process_file(temp_image_path, json_file_path)`:
   - **Input:** A temporary image and its corresponding JSON file containing detected line data.
   - **Workflow:**
     - **Line Retrieval:**
       - Reads line data (start and end points) from the JSON file.
     - **Line Visualization:**
       - Draws detected lines on the image.
       - Annotates each line with its index.
     - **Line Combination:**
       - Combines lines that are close or overlap based on proximity criteria.
       - Draws combined lines with new annotations.
     - **Output:**

        * Saves an image showing combined lines.
        * Displays the combined lines with annotations.
        * Logs the start and end points of combined lines in the console.

3. **Function:** `process_files(temp_files, json_files)`:
    - Iterates through pairs of temporary images and JSON files, calling `process_file` for each.
    - Verifies the existence of each file before processing.
    - Handles missing files with error messages.

4. **Integration:**
    - Processes the global variables `temp_files` and `json_files` generated in earlier steps.
    - Logs an error if required variables are undefined, prompting users to run preceding cells.

Purpose

This cell enhances the line processing workflow by combining and annotating detected lines, ensuring a cleaner and more informative output. It provides visual results for debugging and logs key data points, making it a critical step for refining the analysis pipeline.

Cell 12 Description

This cell filters and updates rectangle-related data from the `processed_files` directory by processing associated JSON files. It focuses on cleaning up OCR results by removing entries with errors.

Key Components

1. **Function:** `find_processed_files_directory(file_paths)`:
    - **Purpose:** Dynamically locates the `processed_files` directory relative to the parent folder of the first file in the provided list.
    - **Workflow:**
        - Derives the parent directory of the input files.
        - Constructs the path to the `processed_files` directory.
        - Checks if the directory exists and returns its path; otherwise, logs an error.

2. **Function:** `filter_and_update_rectangles_for_files(file_paths)`:
    - **Purpose:** Iterates through the provided image file paths, filters out rectangles with OCR errors from corresponding JSON files, and updates the JSON files.
    - **Workflow:**
        - **Locate the `processed_files` Directory:**
            * Calls `find_processed_files_directory` to locate the directory dynamically.
            * Halts processing if the directory cannot be found.
        - **Process Each File:**
            * Skips non-PNG files.
            * Constructs the path to the JSON file for the image.
            * Logs an error and skips the file if the JSON file does not exist.
        - **Filter Rectangles:**
            * Reads the JSON file and filters out rectangles where the text contains "Error during OCR."
            * Logs the filtered rectangle text and its count for verification.
        - **Update JSON:**
            * Updates the JSON data with the filtered rectangles.
            * Saves the updated JSON file back to the same path.

- **Feedback:**
  - Provides detailed logs for each processed file, including success and errors.

3. **Integration:**
   - Processes files listed in the global variable `files_to_process`.
   - Applies dynamic path handling to ensure flexibility across environments.

Purpose

This cell is crucial for refining the rectangle data by removing OCR errors. It dynamically locates the required directory, updates the corresponding JSON files, and provides detailed logs, ensuring the data's integrity and usability for subsequent steps.

Cell 13 Description

This cell provides a debugging utility to inspect JSON files stored in the `rectangle_json_files` global variable, ensuring that the key `"rectangles"` exists and contains valid data.

Key Components

1. **Function:** `debug_check_rectangle_json_files()`
   - **Purpose:** Verifies the integrity of JSON files by checking for the presence of the `"rectangles"` key and examining its contents.
   - **Workflow:**
     - **Global Variable Validation:**
       - Ensures `rectangle_json_files` is defined and not empty.
       - Logs an error and halts if the variable is undefined or empty.
     - **File Inspection:**
       - Iterates through each JSON file in `rectangle_json_files`.
       - Logs an error if a file is missing.
     - **Data Verification:**
       - Opens and parses each JSON file.
       - Checks if the `"rectangles"` key exists:
         - Logs success if the key is found, including the count of rectangles.
         - Optionally logs details of the first rectangle for further verification.
       - Logs available keys in the JSON if the `"rectangles"` key is missing.
     - **Error Handling:**
       - Logs errors for JSON decoding failures or unexpected exceptions.

2. **Integration:**
   - The function is invoked if `rectangle_json_files` exists in the global scope.
   - Logs an error if `rectangle_json_files` is not defined, prompting the user to generate the JSON files first.

Purpose

This cell is an essential debugging tool to validate the structure and content of JSON files generated in earlier processing steps. It provides detailed logs and insights, ensuring data consistency and aiding in troubleshooting potential issues.

Cell 14 Description

This cell provides a comprehensive debugging framework to validate the environment, file existence, and JSON file integrity. It focuses on ensuring that all required variables and files are correctly defined, accessible, and structured.

Key Components

1. **Function:** `debug_environment()`
   - **Purpose:** Checks the existence and contents of the global variables `rectangle_json_files` and `json_files`, which store paths to JSON files for rectangle and line data.
   - **Workflow:**
     - **Check for `rectangle_json_files`:**
       * Verifies if the variable is defined and not empty.
       * Logs the status of each file in the list, flagging missing files with an error.
     - **Check for `json_files`:**
       * Performs similar validation as above for line-related JSON files.
     - **Summary:**
       * Logs all errors or confirms that the environment is correctly set up.
2. **Function:** `validate_json_files(json_file_list, required_keys)`
   - **Purpose:** Validates the contents of a list of JSON files to ensure they include specific required keys.
   - **Workflow:**
     - **File Validation:**
       * Checks if each file in the list exists.
     - **Key Validation:**
       * Opens and parses each JSON file.
       * Checks for the presence of the `required_keys` and logs any missing keys.
     - **Error Handling:**
       * Catches and logs JSON decoding errors or unexpected exceptions.
3. **Integration:**
   - The environment is validated by calling `debug_environment()`.
   - Rectangle-related JSON files (`rectangle_json_files`) are validated for the presence of the `"rectangles"` key.
   - Line-related JSON files (`json_files`) are validated for the presence of the `"combined_lines"` key.

Purpose

This cell ensures the robustness of the processing pipeline by validating critical files and their contents. It systematically identifies missing or improperly structured files, helping to prevent downstream errors. The clear logging and modular design make it a powerful tool for debugging and quality control.

Cell 15 Description

This cell computes adjacency matrices between rectangles based on the proximity of line endpoints to rectangle edges. It processes multiple rectangle and line JSON files to establish connectivity relationships between rectangles.

Key Components

1. **Helper Functions:**
   - `point_to_line_distance(line_start, line_end, point)`:
     - Calculates the shortest distance from a point to a line segment.
     - Handles edge cases like degenerate lines (where start equals end).
   - `is_point_close_to_rectangle(point, rectangle, threshold)`:

- Checks if a point is close to any edge of a rectangle, given a distance threshold.
- Iterates over all edges of the rectangle.

2. **Function:** `compute_adjacency_matrix_via_lines(rectangle_json_path, line_json_path, threshold)`
   - **Purpose:** Computes an adjacency matrix that represents connections between rectangles based on lines.
   - **Workflow:**
     - **Load Data:**
       * Reads rectangle data (coordinates) from the JSON file.
       * Reads line data (start and end points) from the JSON file.
     - **Proximity Check:**
       * For each line, checks if its start and end points are close to any rectangles.
       * Determines adjacency between rectangles based on shared proximity to line endpoints.
     - **Matrix Construction:**
       * Creates a square adjacency matrix where a `True` value indicates a connection between two rectangles.
     - **Output:**
       * Saves the adjacency matrix as a JSON file.
       * Adds the output path to the global `adjacency_matrix_files` list.
3. **Function:** `compute_adjacency_matrices_via_lines(rectangle_json_files, line_json_files, threshold)`
   - **Purpose:** Processes multiple rectangle and line JSON files in pairs to compute adjacency matrices.
   - **Workflow:**
     - **Validation:**
       * Ensures both file lists are non-empty and have matching lengths.
       * Skips missing files with appropriate logging.
     - **Matrix Computation:**
       * Calls `compute_adjacency_matrix_via_lines` for each pair of rectangle and line JSON files.
     - **Summary:**
       * Logs all generated adjacency matrix file paths.
4. **Integration:**
   - The function processes files from `rectangle_json_files` and `json_files` global variables, generated in previous steps.
   - Logs an error if required variables or files are missing.

Purpose

This cell establishes relationships between rectangles using adjacency matrices, enabling analysis of connections within the data. It handles multiple files systematically, ensuring robust validation, processing, and output logging. This is a crucial step for applications like flowchart analysis or visual structure understanding.

Cell 18 Description

This cell generates GraphML files to represent relationships between rectangles using the adjacency matrix. These GraphML files are used for network visualization and analysis.

Key Components

```
\item \textbf{Function: \texttt{generate\_graphml(rectangle\_json\_file,
adjacency\_matrix\_file, output\_directory="graphml\_files")}}
```

- **Purpose:** Creates a GraphML file for a single pair of rectangle JSON and adjacency matrix file.
- **Workflow:**
  - **Load Rectangle Data:**
    - Reads rectangle data from the JSON file.
    - Maps rectangle numbers to their associated text for use as node attributes.
  - **Load Adjacency Matrix:**
    - Reads the adjacency matrix, determining connections between rectangles.
  - **Graph Construction:**
    - Adds nodes to the graph, labeled with rectangle numbers and text.
    - Adds edges based on the adjacency matrix, indicating connections.
  - **Output:**
    - Saves the graph as a GraphML file in the specified directory.
    - Appends the file path to the global `graphml_files` list.

```
\item \textbf{Function: \texttt{generate\_graphml\_files(rectangle\_json\_files,
 adjacency\_matrix\_files, output\_directory="graphml\_files")}}
\begin{itemize}
    \item \textbf{Purpose:} Processes multiple pairs of rectangle JSON and
adjacency matrix files to generate GraphML files.
    \item \textbf{Workflow:}
    \begin{itemize}
        \item \textbf{Validation:}
        \begin{itemize}
            \item Ensures both file lists are non-empty and have matching
lengths.
            \item Logs errors for missing or mismatched files.
        \end{itemize}
        \item \textbf{File Processing:}
        \begin{itemize}
            \item Iterates through the file pairs, calling \texttt{generate\
_graphml} for each.
        \end{itemize}
        \item \textbf{Summary:}
        \begin{itemize}
            \item Logs the paths of all generated GraphML files.
        \end{itemize}
    \end{itemize}
\end{itemize}
```

1. **Integration:**
   - Processes files from the global variables `rectangle_json_files` and `adjacency_matrix_files`.
   - Logs an error if required variables are undefined or empty.

Purpose

This cell converts adjacency relationships into GraphML format, enabling sophisticated visualization and analysis with tools like Gephi or Cytoscape. It supports workflow automation by generating GraphML files for all processed data, with detailed logging for traceability.

Cell 19 Description

This cell contains the following command:

```
!pip install pandas
```

Purpose

- Installs the Pandas library, which is a powerful Python library used for data manipulation and analysis.
- Ensures that the required dependency is available for any subsequent steps that involve dataframes or tabular data processing.

This cell is essential for setting up the environment if the Pandas library is not already installed.

Cell 20 Description

This cell processes GraphML files to detect and analyze communities within a graph using Girvan-Newman and Louvain community detection algorithms. The results are visualized, saved, and summarized.

Key Components

1. **Helper Functions:**
   - `load_rectangle_text_mapping(rectangle_json_file):`
     - Loads rectangle labels and text from a JSON file for graph nodes.
   - `girvan_newman_communities(G, depth=1):`
     - Performs Girvan-Newman community detection, returning communities after a specified number of splits.
   - `louvain_community_detection(G):`
     - Detects communities using the Louvain algorithm.
   - `calculate_modularity(G, communities):`
     - Computes modularity, a measure of the strength of the community structure.
   - `draw_graph_with_communities(G, communities_gn, communities_lv, node_labels, output_directory, output_filename):`
     - Visualizes the graph with Girvan-Newman and Louvain communities.
     - Saves the plot as an image file.
2. **Function:** `process_graph(graphml_file, rectangle_json_file, output_directory="community_results", depth=1)`
   - **Purpose:** Processes a single graph to detect and analyze communities.
   - **Workflow:**
     - **Load Graph and Rectangle Data:**
       - Reads the GraphML file and rectangle text mappings.
       - Maps nodes to their respective labels and text.
     - **Community Detection:**
       - Detects communities using both Girvan-Newman and Louvain methods.
       - Calculates modularity for each method.
     - **Visualization:**
       - Creates and saves a visualization of the graph with detected communities.
     - **Output:**

        * Saves the community results (including modularity and community structure) to a JSON file.
        * Logs results in a table format for user reference.

3. **Function:** `process_multiple_files(graphml_files, rectangle_json_files, output_directory="community depth=1)`
    - **Purpose:** Processes multiple GraphML and rectangle JSON files in parallel.
    - **Workflow:**
        - **Validation:**
            * Ensures the number of GraphML files matches the number of rectangle JSON files.
            * Logs errors for missing or mismatched files.
        - **Batch Processing:**
            * Calls `process_graph` for each file pair.
        - **Summary:**
            * Logs a summary of results for all processed files.

4. **Integration:**
    - Processes files from the global variables `graphml_files` and `rectangle_json_files`.
    - Logs an error if required variables are undefined or empty.

Purpose

This cell identifies and analyzes community structures within a graph. The results are visualized for better interpretability and saved for further analysis. It supports batch processing, ensuring scalability and robustness. This step is vital for understanding graph connectivity and clustering.

Cell 21 Description

This cell contains the following command:

```
!pip install node2vec
```

Purpose

- Installs the `node2vec` library, which is used for generating embeddings for nodes in a graph.
- These embeddings can be used for machine learning tasks, clustering, and graph-based analysis.

This cell ensures that the required library is available in the environment for subsequent steps involving node embeddings.

Cell 22 Description

This cell dynamically loads GraphML files into a dictionary for further processing.

Key Components

- **Workflow:**
    - Iterates through the global list `graphml_files`, which contains paths to GraphML files.
    - For each file:
        * Extracts the file name (without extension) to use as the graph's identifier.
        * Loads the GraphML file into a NetworkX graph object.
        * Stores the graph object in the dictionary `graphs`, with the file name as the key.
    - **Logging:**
        * Prints informative messages for each file being loaded, including its path and identifier.

Purpose

This cell organizes all the GraphML data into a Python dictionary for easy access and manipulation in subsequent steps. The dynamic loading ensures scalability and reusability for different datasets.

Cell 23 Description

This cell performs first-level community detection on graphs using the Girvan-Newman algorithm, replacing node identifiers with their corresponding text content.

Key Components

1. **Function:** `load_rectangle_text_mapping(graph_name, processed_files_directory)`
   - **Purpose:** Loads rectangle text mappings for nodes in a graph.
   - **Workflow:**
     - Constructs the path to the JSON file containing rectangle mappings.
     - Loads rectangle data and maps node identifiers to text content.
     - Logs an error if the file is not found or improperly formatted.
2. **Function:** `girvan_newman_first_level_with_texts(G, rectangle_text_mapping)`
   - **Purpose:** Detects first-level communities in a graph and maps nodes to their corresponding text content.
   - **Workflow:**
     - Runs the Girvan-Newman algorithm to detect the first split of communities.
     - Replaces node identifiers with text mappings for each community.
     - Handles cases where communities cannot be detected and logs warnings.
3. **Function:** `process_all_graphs_first_level_with_texts(graphs, processed_files_directory)`
   - **Purpose:** Processes multiple graphs to detect and log first-level communities with textual representations.
   - **Workflow:**
     - Iterates through the `graphs` dictionary, where each key is a graph name, and each value is a graph object.
     - Loads text mappings for each graph.
     - Runs `girvan_newman_first_level_with_texts` to detect communities.
     - Stores results, including both textual and identifier-based community representations.
     - Logs errors for graphs with missing or invalid text mappings.
4. **Integration and Output:**
   - Dynamically locates the `processed_files` directory.
   - Processes all graphs in the `graphs` dictionary.
   - Saves the results, including community structures with textual representations, to a JSON file (`graph_first_level_communities_with_texts.json`).

Purpose

This cell integrates graph-based community detection with textual insights, enabling a more interpretable representation of detected communities. The results are saved for further analysis, providing both the textual content and underlying graph identifiers. This step is essential for understanding graph structures in context.

Cell 24 Description

This cell calculates the modularity of graphs based on detected communities, providing insights into the strength of the community structure within the graphs.

Key Components

1. **Function:** `calculate_modularity(G, communities)`
   - **Purpose:** Computes the modularity of a graph based on a given set of communities.
   - **Workflow:**
     - Ensures that communities are passed as sets of node identifiers.
     - Uses NetworkX's modularity function to calculate the modularity score.
2. **Modularity Calculation for Each Graph:**
   - **Workflow:**
     - Iterates through graphs:
       - Processes each graph stored in the `graphs` dictionary.
     - **Loads Rectangle Text Mapping:**
       - Retrieves text mappings for nodes using `load_rectangle_text_mapping`.
       - Skips graphs with missing mappings.
     - **Detects Communities:**
       - Runs `girvan_newman_first_level_with_texts` to detect first-level communities and extract node identifiers.
     - **Modularity Calculation:**
       - Filters valid communities to include only subsets of existing graph nodes.
       - Calculates modularity for each graph using the detected communities.
       - Logs the modularity value for each graph.
     - **Logs Errors:**
       - Handles graphs where communities cannot be detected.
3. **Aggregate Results:**
   - Stores all modularity scores in the `modularities` list.
   - Computes the average modularity across all graphs.
   - Logs the average modularity or a warning if no scores are available.

Purpose

This cell evaluates the quality of community detection by calculating modularity scores for graphs. The results provide a quantitative measure of how well the detected communities partition the graph, aiding in the assessment and comparison of community structures.

Cell 25 Description

This cell collects community information for all graphs, including node and edge counts, community structures, and their textual mappings. The results are saved to a JSON file for further analysis.

Key Components

1. **Function:** `collect_graph_community_information(graphs, processed_files_directory)`
   - **Purpose:** Gathers community-related data for each graph, including textual and identifier-based community structures.
   - **Workflow:**

- Iterates through graphs:
  - Processes each graph stored in the `graphs` dictionary.
- **Loads Rectangle Text Mapping:**
  - Retrieves textual data for nodes using `load_rectangle_text_mapping`.
- **Detects Communities:**
  - Runs `girvan_newman_first_level_with_texts` to detect first-level communities.
  - Extracts both node identifiers and textual mappings for detected communities.
- **Stores Results:**
  - Collects:
    - Node and edge counts for the graph.
    - Communities in both identifier and textual formats.
  - Logs success or warnings for each graph.

2. **Results Aggregation:**

- **Stores Results:**
  - Aggregates all collected data in a dictionary, with graph names as keys and community details as values.
- **Saves Results:**
  - Writes the aggregated data to a JSON file named `graph_community_information.json`.

3. **Output:**

- Saves the community information, including:
  - Number of nodes and edges.
  - Communities with both identifiers and textual representations.

Purpose

This cell centralizes graph and community data, providing a detailed, structured overview of community structures for all graphs. The saved JSON file facilitates downstream analysis or reporting.

Cell 28 Description

This cell prepares community embeddings for clustering by reducing their dimensionality using t-SNE (t-Distributed Stochastic Neighbor Embedding) and standardizing the results.

Key Components

1. **Function:** `prepare_clustering_data(community_avg_embeddings)`

- **Purpose:** Converts community average embeddings into a format suitable for clustering.
- **Workflow:**
  - Iterates through `community_avg_embeddings`:
    - For each graph and its communities, appends embeddings to a data array.
    - Creates corresponding labels using the graph and community names (e.g., "Graph-Name_Community1").
  - **Returns:**
    - A NumPy array of embeddings (`data`).
    - A list of corresponding labels.

2. **Dimensionality Reduction with t-SNE:**

- **Purpose:** Reduces high-dimensional embeddings to 2D for clustering and visualization.
- **Workflow:**

- Configures t-SNE to reduce embeddings to 2 dimensions (`n_components=2`) with a fixed random seed for reproducibility.
- Fits t-SNE on the embeddings and transforms the data.

3. **Standardization with StandardScaler:**
   - **Purpose:** Standardizes the reduced 2D embeddings to ensure a consistent scale for clustering algorithms.
   - **Workflow:**
     - Fits a StandardScaler to the t-SNE-transformed data.
     - Scales the data to have a mean of 0 and standard deviation of 1.
4. **Output:**
   - Prepares the `data_tsne` variable containing the 2D, standardized embeddings for clustering.
   - Logs a success message upon completion.

Purpose

This cell processes community embeddings for clustering and visualization by reducing their dimensionality and standardizing them. The output is ready for tasks like grouping similar communities or visualizing their relationships in 2D space.

Cell 29 Description

This cell performs MeanShift clustering on the prepared community embeddings to group similar communities and evaluates the clustering quality using silhouette scores.

Key Components

1. **Function:** `perform_clustering(data, bandwidth)`
   - **Purpose:** Performs MeanShift clustering on the input data using a specified bandwidth.
   - **Workflow:**
     - Configures and fits a MeanShift model on the data.
     - Returns the clustering model.
2. **Function:** `find_best_bandwidth(data, bandwidth_values)`
   - **Purpose:** Identifies the optimal bandwidth for MeanShift clustering using silhouette scores.
   - **Workflow:**
     - Iterates through a range of bandwidth values:
       * Fits a MeanShift model on the data.
       * Calculates the number of clusters and evaluates the silhouette score if at least two clusters are formed.
       * Logs the silhouette score and bandwidth for reference.
     - Tracks the bandwidth with the highest silhouette score.
     - **Returns:**
       * The best bandwidth.
       * The corresponding silhouette score.
       * A list of silhouette scores for all bandwidths.
3. **Execution:**
   - **Bandwidth Testing:**
     - Tests bandwidth values ranging from 0.5 to 1.0 in 10 equally spaced intervals.
   - **Plot Results:**

- Plots the silhouette scores against the tested bandwidth values to visualize the performance.

4. **Output:**
   - Identifies the best bandwidth for clustering based on silhouette scores.
   - Provides a visual representation of silhouette scores vs. bandwidth.

Purpose

This cell optimizes the MeanShift clustering process by selecting the bandwidth that maximizes clustering quality. The silhouette score ensures that the clusters are well-defined, and the visual plot aids in understanding the performance of different bandwidth values.

Cell 30 Description

This cell performs KMeans clustering on the t-SNE-transformed data and evaluates the clustering quality using the silhouette score.

Key Components

1. **Function:** `evaluate_silhouette_score(data, labels)`
   - **Purpose:** Computes the silhouette score for clustering results.
   - **Workflow:**
     - Calculates the silhouette score using the data and cluster labels.
     - Logs and returns the silhouette score.

2. **Execution:**
   - **KMeans Clustering:**
     - Configures a KMeans model with:
       - `n_clusters=15`: Specifies the number of clusters.
       - `random_state=42`: Ensures reproducibility.
       - `n_init=10`: Runs the KMeans algorithm 10 times and selects the best result.
     - Fits the KMeans model on the t-SNE-transformed data (`data_tsne`).
     - Retrieves the cluster labels for each data point.
   - **Silhouette Score Evaluation:**
     - Calculates the silhouette score to assess clustering quality.
     - Logs success or errors during the process.

3. **Output:**
   - Logs:
     - The silhouette score for the KMeans clustering.
     - Success or error messages based on the execution.

Purpose

This cell evaluates the clustering structure of the data using KMeans and silhouette score. It provides insights into how well the clustering algorithm partitions the data, aiding in determining the appropriateness of the selected number of clusters.

Cell 31 Description

This cell visualizes the KMeans clustering results using the t-SNE-transformed data and provides details about the clusters, including textual community information.

Key Components

1. **Visualization of Clusters:**
   - **t-SNE Scatter Plot:**
     - Configures a large figure size (100x80 inches) to enhance visibility.
     - Iterates through unique cluster labels and plots points corresponding to each cluster using unique colors from the `tab10` colormap.
     - **Customizes the plot:**
       * Adds a title and axis labels with increased font sizes.
       * Adjusts tick label sizes for better readability.
       * Configures a legend to display cluster labels with a larger font size.
2. **Community Information for Clusters:**
   - **Cluster Details Extraction:**
     - Groups community labels (`labels`) by their respective cluster labels (`cluster_labels`).
     - Iterates through each cluster and retrieves textual information about the communities.
     - Splits the label into the graph name and community identifier.
     - Uses `rectangle_text_mapping` to map community identifiers to their corresponding textual descriptions.
   - Logs community details for each cluster.
   - Handles invalid or improperly formatted labels gracefully, logging warnings when necessary.
3. **Output:**
   - Displays a scatter plot showing clusters in the t-SNE-reduced 2D space.
   - Logs community details for each cluster, including textual descriptions.

Purpose

This cell provides a clear visualization of the clustering results and enriches the analysis by linking cluster members to their textual descriptions. The combination of visualization and textual insights aids in interpreting the clustering structure and its implications.