1. About the Dataset

- **FakeNewsNet**
  The dataset used here is a subset of FakeNewsNet focused on two news sources:
  - **Politifact**
  - **GossipCop**
- Each news piece (fake or real) is associated with a hierarchical propagation structure: how tweets (and retweets or replies) spread this news over time on Twitter.
- You can access the dataset at https://github.com/mdepak/fake-news-propagation/
- **Privacy and Structure**
  To comply with Twitter's privacy policy, personally identifiable information is anonymized, and tweet contents are not shared. Instead, each tweet is identified by:
  - A **random tweet id**
  - A **timestamp** in epoch format
  - A **random user id**
  - Additional attributes (e.g., bot scores in the retweet networks, sentiment scores in the reply-chain networks)

JSON Format
Each file is a JSON describing a single "diffusion tree" for one piece of news. The top-level JSON object (the root) represents the originating tweet, and it recursively contains children representing retweets or replies.
For example, a node in the JSON might look like:

```
{
  "id": "random_tweet_id",
  "time": 1623456789,
  "children": [
    { "id": "child_tweet_id_1", "time": 1623456880, "children": [...] },
    { "id": "child_tweet_id_2", "time": 1623456940, "children": [...] }
  ]
}
```

- This nested structure can be represented as a directed acyclic graph (DAG), or more specifically a tree, where edges point from a tweet to its retweets (or replies).
- **Directory Organization**
  In the code, `data_dir` is the root path to your local copy of the dataset. Subfolders are named after the source and label (e.g., `politifact_fake`, `politifact_real`, `gossipcop_fake`, `gossipcop_real`). Each subfolder contains multiple JSON files, each file corresponding to one piece of news and its diffusion tree.

---

2. High-Level Code Overview

The code is organized into several sections (labeled 0–8). These sections follow a workflow:

1. **Plotting a Single Sample Tree** (for visualization/demonstration).
2. **Setup** (defining data paths, source/label lists).
3. **Helper Functions** for building diffusion trees (as NetworkX graphs) and computing:
    - Depth of the tree
    - Branching factor
    - Number of retweets (node counts)
4. **Analysis** of all diffusion trees in a given folder to gather distributions of depth, retweet counts, and branching factors.
5. **Plotting** distributions (e.g., retweet counts) on a log-log scale.
6. **Plotting** other distributions (depth, branching factor).
7. **Diffusion Speed Analysis** (time-based analysis of how quickly nodes get “infected”/reach the news).
8. **Plotting** the diffusion speed results on a log-log scale.

Below is a step-by-step explanation of each part of the code.

---

3. Detailed Step-by-Step Explanation

(0) Plotting a Sample Diffusion Tree

```
def plot_sample_tree(source, label, data_dir, max_nodes=20):
    ...
```

- **Purpose**: Load one JSON file from the specified folder (e.g., `politifact_fake`) and build a directed graph (DiGraph) in NetworkX to visualize the diffusion structure.
- **Key Steps**:
    1. **Folder Path**: Combines `data_dir`, the `source` string, and the `label` string (e.g., `"politifact_" + "fake"` = `"politifact_fake"`).
    2. **List JSON Files**: Looks for any file ending with `.json`. The code picks the *first* JSON file found.
    3. **Build DiGraph**:
        - Defines a nested function `add_edges(node)` that reads a node’s children and adds edges `(parent -> child)` in the NetworkX graph.
        - Recursively calls `add_edges` on each child to traverse the entire tree.
    4. **Node Limit**: If the tree is very large, it sub-samples the first `max_nodes` nodes for clarity.

5. **Plotting**: Uses `nx.spring_layout` to determine node positions, and `nx.draw_networkx_nodes`/`nx.draw_networkx_edges` to draw a directed graph.

Outcome: A *visual representation* of one diffusion tree, showing how an original post branched out into multiple retweets or replies.

---

(1) Basic Setup

```
data_dir = ""  # Replace with your actual dataset path
sources = ["politifact_", "gossipcop_"]
labels = ["fake", "real"]
plot_sample_tree(sources[0], labels[0], data_dir)
```

- Here, you set `data_dir` to wherever your JSON data resides.
- `sources` and `labels` are simple lists specifying the news source and label categories to iterate over.
- Finally, the code calls `plot_sample_tree` with the first `(source, label)` pair—e.g., `politifact_fake`—as a quick demonstration.

---

(2) Helper Functions for Basic Analysis

`compute_tree_depth(graph)`

```
def compute_tree_depth(graph):
    """Compute depth (longest path) in a directed acyclic graph."""
    if graph.number_of_nodes() == 0:
        return 0
    return nx.dag_longest_path_length(graph)
```

- **What it does**: Finds the longest path in the DAG. In diffusion terms, this is the "depth" of the propagation tree (the number of hops from the root tweet to the furthest retweet).

`compute_branching_factor(graph)`

```
def compute_branching_factor(graph):
    """
    Compute the average out-degree over all nodes that actually have
children (out_degree > 0).
    """
```

```
    internal_nodes = [n for n in graph.nodes() if
graph.out_degree(n) > 0]
    if not internal_nodes:
        return 0.0
    return sum(graph.out_degree(n) for n in internal_nodes) /
len(internal_nodes)
```

- **What it does**: Calculates how "wide" the tree branches on average. Specifically, it takes the average out-degree for any node that has at least one child.

Interpreting Branching Factor:

- A branching factor of 1 suggests a "chain-like" spread (each user retweets to exactly one other user on average).
- A branching factor higher than 1 suggests more "viral" spreading behavior (one user's post leads to multiple retweets).

---

(3) Analyzing All Diffusion Trees in a Folder

```
def analyze_diffusion_trees(source, label):
    ...
```

- **Purpose**: Reads *all* the JSON files in a particular folder (e.g., `politifact_fake`) and builds a diffusion tree for each. Then, it computes:
    1. `depth_list`: A list of the depth of each tree.
    2. `retweet_counts`: A list of the total number of nodes in each tree (i.e., how many tweets are involved in the diffusion).
    3. `branching_factors`: A list of the average branching factor in each tree.
- **Key Steps**:
    1. Loop through every `.json` in the `source+label` directory.
    2. Build a `nx.DiGraph` by recursively adding edges.
    3. Compute `compute_tree_depth(G)`, `len(G.nodes())`, and `compute_branching_factor(G)` for each graph.
    4. Return three lists holding these computed values across all files.

---

(4) Gather Data and Print Results

```
analysis_results = {}
for source in sources:
    for label in labels:
        key = f"{source}{label}"
```

```
        depth_list, retweet_counts, branching_factors =
analyze_diffusion_trees(source, label)
        analysis_results[key] = (depth_list, retweet_counts,
branching_factors)
        print(f"Processed {key} -> #Trees: {len(depth_list)}")
```

- **Goal**: For each `(source, label)` combination:
  - Run `analyze_diffusion_trees`.
  - Store the results in a dictionary `analysis_results`, indexed by the string key (like `"politifact_fake"`).
  - Print how many diffusion trees were processed.

Outcome: A single data structure, `analysis_results`, now holds the basic metrics (depth, retweet count, branching factor) for each tree in each category.

---

(5) Plot Distribution of Number of Retweets (Tree Size) on Log-Log Scale

Why log-log plots?
When dealing with diffusion or popularity distributions, data often follows heavy-tailed or power-law-like patterns. Log-log plots help visualize these distributions more clearly.

`log_binning(data, num_bins=10)`

```
def log_binning(data, num_bins=10):
    ...
```

- **What it does**:
  1. Removes zero values (which are invalid for log).
  2. Creates log-spaced bins between the minimum and maximum of `data`.
  3. Uses `np.histogram` to compute the probability density for each bin.
  4. Returns the geometric center of each bin as `bin_centers`, along with the `hist` (density values).

`plot_log_log_distribution(data_dict, title, xlabel)`

```
def plot_log_log_distribution(data_dict, title, xlabel):
    ...
```

- **What it does**:
  1. For each key in `data_dict` (e.g., `"politifact_fake"`), retrieves the associated list of values (e.g., retweet counts).
  2. Applies `log_binning` to get bin centers and densities.

3. Plots them on a log-log scale (`plt.loglog(...)`).
4. Adds legends, labels, and gridlines.

Actual Plot Call:

```
plot_log_log_distribution(
    {k: v[1] for k, v in analysis_results.items()},  # v[1] is the
retweet_counts
    "Log-Log Distribution of Number of Retweets (Tree Size)",
    "Number of Retweets"
)
```

- `v[1]` is the list of retweet counts for each tree.
- You'll see multiple lines on the plot (one per key), each showing how retweet sizes are distributed.

---

(6) Plot Depth Distribution and Branching Factor Distribution

Depth Distribution

```
plot_line_distribution(
    {k: v[0] for k, v in analysis_results.items()},  # v[0] =
depth_list
    title="Depth Distribution of Tree Depth",
    xlabel="Tree Depth",
    log_y=True
)
```

1.
    - This uses a simple line histogram (with `np.histogram`) to show how tree depths are distributed.
    - `log_y=True` sets the y-axis to a log scale.

Branching Factor Distribution

```
plot_log_log_branching_factor_distribution(
    {k: v[2] for k, v in analysis_results.items()},
    title="Branching Factor Distribution (Log-Log)"
)
```

2.
    - Similar approach but specifically for the branching factor values.
    - Uses the same `log_binning` approach and plots them on a log-log scale.

Outcome: You can observe whether real/fake news tends to have deeper diffusion trees, how often it exhibits high branching, etc.

---

(7) Diffusion Speed Analysis (Log-Log)

This section introduces time-based analysis. We want to see how quickly a certain piece of news “infects” or reaches new accounts over time.

`build_graph_and_times(data)`

```
def build_graph_and_times(data):
    """
    Build a DiGraph and gather timestamps of each node.
    Returns:
      G: DiGraph
      times: dict { node_id: timestamp }
      root_time: the earliest timestamp among all nodes (or None if
invalid)
    """
    ...
```

- **Steps**:
  1. Recursively parse the JSON to add edges into a NetworkX DiGraph.
  2. Store each node’s time in a dictionary `times[node_id] = node_time`.
  3. Determine `root_time`, which is the minimum (earliest) valid timestamp in the tree.

`analyze_diffusion_speed(source, label, num_time_bins=50)`

```
def analyze_diffusion_speed(source, label, num_time_bins=50):
    ...
```

- **What it does**:
  1. For each JSON file, build the graph and extract timestamps.
  2. Shift all timestamps so that the earliest node is at time `t=1` (avoiding zero or negative times for log-scale).
  3. Sort the times for that tree and accumulate them in `time_diffs`.
  4. Collect these across all trees in a given folder.
  5. Use `np.logspace` to create log-spaced time bins between `1.0` and the maximum observed time difference (`max_time_diff`).
  6. For each tree’s `time_diffs`, compute how many nodes appear (are “infected”) by each time bin.

7. Aggregate these to get an average curve (`avg_curve`) showing how on average, the news propagation accumulates retweets over (log-scaled) time.
8. Return the time bins and the average ± standard deviation curve.

---

(8) Plotting Diffusion Speed

```
analysis_speed_results = {}
for source in sources:
    for label in labels:
        key = f"{source}{label}"
        time_bins, (avg_curve, std_curve) =
analyze_diffusion_speed(source, label, num_time_bins=50)
        analysis_speed_results[key] = (time_bins, (avg_curve,
std_curve))

plot_diffusion_speeds(analysis_speed_results)
```

- **Storing Results**: Similar loop over `(source, label)` as before, calling `analyze_diffusion_speed`.
- **Plot**:
  - For each dataset, we plot the average diffusion curve (`avg_curve`) in log-log space:
    - **x-axis**: Time since earliest node (log scale).
    - **y-axis**: Average number of infected nodes (log scale).

Interpretation: This tells you how quickly a piece of news tends to spread over time—whether growth is rapid (steep slope early on) or more gradual.

---

4. What the Outputs Show You

1. **Sample Diffusion Tree**: A direct visualization of a single news item's propagation structure.
2. **Retweet Size Distribution (Log-Log)**: Shows how many diffusion trees are small vs. extremely large. Often you might see a heavy-tailed pattern (fewer very large trees).
3. **Depth Distribution**: Tells you whether news typically spreads in many "hops." A larger average depth might indicate longer chains of retweets or replies.
4. **Branching Factor Distribution**: Indicates whether retweets often fan out widely or remain fairly linear.
5. **Diffusion Speed**: Reflects how quickly new accounts adopt the news.