# Code Notes: Log File Reading and Preview

---

## Imports

- `from pathlib import Path`: Used for handling file and directory paths in a platform-independent way.
- `import chardet`: Library for detecting the character encoding of files, ensuring logs are read correctly regardless of their original encoding.

---

## Path Setup

- `base_path = Path('../datasets/raw_drift_dataset')`
  - Sets the base directory where all raw log datasets are stored.
- Individual log file paths (e.g., `hdfs_path`, `apache_path`, etc.)
  - Each variable defines the full path to a specific log file by joining the base path with the dataset and file name.

---

## Encoding Detection Function

- `def detect_encoding(file_path):`
  - Uses `chardet` to read the first 10,000 bytes of a file and detect its encoding.
  - Returns the detected encoding as a string (e.g., 'utf-8', 'ISO-8859-1').
  - **Purpose:** Ensures that log files with different encodings can be read without errors.

---

# Log File Reading Function

- `def read_log_file(file_path):`
    - Calls `detect_encoding` to determine the file's encoding.
    - Opens the file in text mode with the detected encoding, replacing any problematic characters.
    - Reads the file line by line, strips whitespace, and skips empty lines.
    - Returns a list of non-empty log entry strings.
    - **Purpose:** Provides a clean list of log entries for further analysis.

---

# Previewing Log Entries

- The `for` loop iterates over each dataset and its corresponding log file path.
    - Checks if the log file exists.
    - If it exists:
        - Prints the dataset name and file name.
        - Reads the first 5 non-empty lines using `read_log_file` and prints them as sample entries.
    - If the file does not exist:
        - Prints a message indicating the file was not found.
- **Purpose:** Quickly preview the structure and content of each log file to verify data quality and format before deeper analysis.

---

# Variables and Their Roles

- `base_path`: Root directory for all log datasets.
- `*_path` variables: Full paths to each dataset's log file.
- `encoding`: Detected character encoding for each file.
- `lines`: List of non-empty, stripped log entries from a file.
- `sample_lines`: First 5 log entries used for preview.

---

## Summary

- This code ensures robust, encoding-aware reading of log files from multiple sources.
- It provides a quick way to check data quality and log structure before proceeding to EDA or feature extraction.
- Modular functions ( `detect_encoding` , `read_log_file` ) make it easy to reuse and adapt for other datasets or workflows.

---

# Code Notes: Log Entry Length Analysis and Visualization

## Imports

- `import matplotlib.pyplot as plt` : Used for creating and customizing plots.
- `import seaborn as sns` : Statistical data visualization library, used here for histograms with KDE (Kernel Density Estimate).
- `from pathlib import Path` : For handling file and directory paths.
- `import pandas as pd` : Used for statistical calculations (e.g., standard deviation).

---

## Plot Directory Setup

- `plot_dir = Path("eda_plots")`
    - Defines the directory where EDA plots will be saved.
    - `plot_dir.mkdir(exist_ok=True)` : Creates the directory if it doesn't already exist.

---

# Function: explore_entry_length

- `def explore_entry_length(logs, name):`
  - **Purpose:** Analyze and visualize the length (in characters) of log entries for a given dataset.
  - **Parameters:**
    - `logs`: List of log entry strings.
    - `name`: Name of the dataset (for labeling plots/files).
  - **Workflow:**
    - Computes the length of each log entry.
    - Prints summary statistics: total entries, min/max/mean/std of entry lengths.
    - Plots a histogram (with KDE) of entry lengths using seaborn/matplotlib.
    - Saves the plot as a PNG image in the `eda_plots` directory.
    - Displays the plot.
  - **Interpretation:**
    - The histogram shows the distribution of log entry lengths (e.g., most logs are short, some are very long).
    - Summary stats help identify outliers or unusual verbosity/conciseness in logs.

---

# Reading Log Files

- Calls `read_log_file` for each dataset path to get lists of log entries:
  - `hdfs_logs`, `apache_logs`, `healthapp_logs`, `bgl_logs`, `hpc_logs`, `linux_logs`, `mac_logs`
- **Purpose:** Prepares the data for analysis and visualization.

---

# Running the Analysis

- Calls `explore_entry_length` for each dataset:
  - e.g., `explore_entry_length(apache_logs, 'APACHE')`
- **Purpose:** Generates and saves a histogram plot for each dataset, and prints summary

statistics to the console.

---

## Variables and Their Roles

- `logs` : List of log entry strings for a dataset.
- `name` : Dataset name (used in plot titles and filenames).
- `lengths` : List of character counts for each log entry.
- `plot_dir` : Directory where plots are saved.

---

## Summary

- This code provides a quick, visual overview of log entry length distributions for multiple datasets.
- Helps identify structural differences, verbosity, and outliers in log data.
- Plots and statistics are saved for later review and inclusion in reports or presentations.

---

# Code Notes: Checking for Missing or Malformed Log Entries

## Function: check_missing_entries

```python
def check_missing_entries(logs, name):
    """
    Check for empty or obviously malformed entries.
    """
    empty_count = sum(1 for line in logs if not line)
    print(f"{name}: {empty_count} empty entries found.")
```

## Purpose

- Quickly checks a list of log entries for empty strings (i.e., missing or blank entries).
- Prints the number of empty entries found for each dataset.

## How it Works

- Iterates through the `logs` list.
- Counts entries that are empty ( `not line` ).
- Prints a summary for the dataset.

## Usage Example

```
check_missing_entries(hdfs_logs, "HDFS")
check_missing_entries(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- If `empty_count` is 0, the dataset has no missing or blank entries (high data quality).
- If nonzero, further investigation is needed to handle or clean these entries.

---

# Code Notes: Analyzing Message Type Distribution

## Function: analyze_message_types

```
def analyze_message_types(logs, name):
    """
    Analyze the frequency of different message types (INFO, ERROR, etc.).
    Saves the plot as an image in the 'eda_plots' directory.
    """
    patterns = {
        'ERROR': r'error|ERROR|Error|FAIL|fail|Fail|EXCEPTION|exception|Exception',
        'WARNING': r'warn|WARN|Warn|WARNING|warning|Warning',
        'INFO': r'info|INFO|Info|NOTICE|notice|Notice',
        'DEBUG': r'debug|DEBUG|Debug|TRACE|trace|Trace',
        'CRITICAL': r'critical|CRITICAL|Critical|FATAL|fatal|Fatal|EMERGENCY|emergency|Eme
    }
    type_counts = Counter()
    for log in logs:
        for type_name, pattern in patterns.items():
            if re.search(pattern, log):
                type_counts[type_name] += 1
                break
        else:
            type_counts['OTHER'] += 1
    print(f"\n{name} Message Type Distribution:")
    for t, c in type_counts.items():
        print(f"  {t}: {c} ({c/len(logs)*100:.2f}%)")
    # Visualize
    plt.figure(figsize=(8, 4))
    sns.barplot(x=list(type_counts.keys()), y=list(type_counts.values()))
    plt.title(f"{name} Message Type Distribution")
    plt.ylabel("Count")
    plt.tight_layout()
    plt.savefig(plot_dir / f"{name.lower()}_message_types.png")
    plt.show()
```

## Purpose

- Categorizes log entries by message type (e.g., ERROR, WARNING, INFO).
- Counts and visualizes the frequency of each type.
- Saves a bar plot for reporting and further analysis.

## How it Works

- Defines regex patterns for each message type.
- Iterates through each log entry, searching for a match with each pattern.
- Increments the corresponding type count.
- If no pattern matches, counts as 'OTHER'.
- Prints the count and percentage for each type.
- Plots and saves a bar chart of the distribution.

## Usage Example

```
analyze_message_types(hdfs_logs, "HDFS")
analyze_message_types(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- Reveals the dominant message types in each dataset (e.g., mostly INFO, mostly ERROR).
- Helps identify unusual log behavior (e.g., high ERROR rate).
- The 'OTHER' category captures entries that don't match standard types, which may need further review.

---

# Summary

- These functions are essential for EDA:
  - `check_missing_entries` ensures data completeness.
  - `analyze_message_types` provides insight into log content and potential issues.
- Both print summary statistics and, for message types, generate visualizations for

inclusion in reports and presentations.

---

# Code Notes: Analyzing Temporal Patterns in Log Data

## Imports

- `from datetime import datetime`: For parsing and handling timestamps.
- `import matplotlib.pyplot as plt`: For creating visualizations.
- `import numpy as np`: For numerical calculations (e.g., mean time differences).
- `from collections import Counter`: For counting occurrences of hours, days, etc.
- `from pathlib import Path`: For handling file paths (used for plot saving).

## Function: analyze_temporal_patterns

```python
def analyze_temporal_patterns(logs, name, plot_dir=Path("eda_plots")):
    """
    Extract and analyze timestamps from log entries using multiple common patterns.
    Saves visualizations to the specified plot directory.
    """
    # ... function code ...
```

### Purpose

- Extracts timestamps from log entries using several common log formats.
- Analyzes temporal patterns such as activity by hour, time between logs, and cumulative log volume.
- Prints summary statistics and saves multiple visualizations for each dataset.

## How it Works

- Defines a list of regex patterns and corresponding datetime formats to match various log timestamp styles (e.g., HDFS, ISO, syslog, Apache).
- Iterates through each log entry, searching for a matching timestamp pattern.
- Parses and collects valid timestamps into a list.
- If timestamps are found:
    - Prints the number of timestamps and the time range covered.
    - Calculates time differences between consecutive logs and prints summary stats (average, min, max time between logs).
    - Visualizes:
        - Hourly distribution of log entries (histogram).
        - Distribution of time between logs (histogram).
        - Cumulative log count over time (line plot).
    - Saves the combined plot as a PNG in the plot directory.
    - Prints the most common hours, days, months, and weekdays for log activity.
- If no timestamps are found, prints a message and shows sample log entries for debugging.

## Usage Example

```
analyze_temporal_patterns(hdfs_logs, "HDFS")
analyze_temporal_patterns(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- Reveals temporal activity patterns (e.g., peak hours, daily/weekly cycles, bursts of activity).
- Time-between-logs histogram can highlight bursts, outages, or regular intervals.
- Cumulative plot shows log volume growth and can reveal periods of increased or decreased activity.
- Most common hours/days/months help identify operational cycles or anomalies.
- If no timestamps are found, may indicate unusual log format or need for pattern adjustment.

---

## Summary

- `analyze_temporal_patterns` is a key EDA tool for understanding when log events occur and how activity changes over time.
- Supports robust drift detection and incident analysis by revealing temporal structure in log data.
- Visualizations and statistics are useful for reports, presentations, and further analysis.

---

# Code Notes: Extracting and Visualizing Timestamps from Log Entries

## Imports

- `import re`: For regular expression matching to extract timestamps from log entries.
- `from collections import Counter`: For counting occurrences if needed (not used directly in this function but often paired in EDA workflows).
- `from datetime import datetime`: For parsing timestamp strings into datetime objects.
- `import matplotlib.pyplot as plt`: For plotting histograms of log entry dates.
- `import seaborn as sns`: For enhanced statistical visualizations (used for histogram plotting).

# Function: extract_timestamps

```
def extract_timestamps(logs, time_format, name):
    timestamps = []
    for log in logs:
        try:
            # Adjusted regex for Apache logs
            match = re.search(r'\[(\d{2}/[A-Za-z]{3}/\d{4}:\d{2}:\d{2}:\d{2})', log)
            if match:
                timestamps.append(datetime.strptime(match.group(1), time_format))
        except Exception:
            continue
    if timestamps:
        print(f"\n{name} Temporal Coverage:")
        print(f"  Earliest: {min(timestamps)}")
        print(f"  Latest: {max(timestamps)}")
        plt.figure(figsize=(12, 4))
        sns.histplot([t.date() for t in timestamps], bins=30)
        plt.title(f"{name} Log Entries Over Time")
        plt.xlabel("Date")
        plt.ylabel("Count")
        plt.tight_layout()
        plt.savefig(plot_dir / f"{name.lower()}_temporal_coverage.png")
        plt.show()
    else:
        print(f"{name}: No timestamps found or format mismatch.")
```

## Purpose

- Extracts timestamps from log entries using a regex pattern (specifically for Apache-style logs).
- Parses the extracted timestamp strings into datetime objects using the provided format.
- Visualizes the distribution of log entries over time as a histogram.
- Prints the earliest and latest timestamps found for temporal coverage assessment.

## How it Works

- Iterates through each log entry, searching for a timestamp using a regex pattern (e.g., `[10/Oct/2000:13:55:36`).
- If a match is found, parses the timestamp string into a datetime object using the specified format (e.g., `%d/%b/%Y:%H:%M:%S`).
- Collects all valid timestamps into a list.
- If timestamps are found:
    - Prints the earliest and latest timestamps (temporal coverage).
    - Plots a histogram of log entry dates (daily granularity) using seaborn.
    - Saves the plot as a PNG in the plot directory.
- If no timestamps are found, prints a warning message.

## Usage Example

```
extract_timestamps(apache_logs, "%d/%b/%Y:%H:%M:%S", "Apache")
extract_timestamps(bgl_logs, "%d/%b/%Y:%H:%M:%S", "BGL")
# ...repeat for other datasets
```

## Interpretation

- The histogram shows the volume of log entries per day, revealing periods of high or low activity.
- The earliest and latest timestamps indicate the time span covered by the dataset.
- If no timestamps are found, it may indicate a format mismatch or unusual log structure.

---

# Summary

- `extract_timestamps` is a focused EDA tool for quickly assessing the temporal coverage and daily activity of log datasets with Apache-style timestamps.
- Useful for identifying data gaps, bursts, or periods of interest before deeper temporal or drift analysis.
- The approach can be adapted for other timestamp formats by changing the regex and

format string.

---

# Code Notes: Dataset-Specific Timestamp Extraction and Visualization

## Imports

- `import re`: For regular expression matching to extract timestamps from log entries.
- `from datetime import datetime`: For parsing timestamp strings into datetime objects.
- `import matplotlib.pyplot as plt`: For plotting histograms of log entry dates.
- `import seaborn as sns`: For enhanced statistical visualizations (used for histogram plotting).
- `from pathlib import Path`: For handling file paths (used for plot saving).

## Plot Directory Setup

- `plot_dir = Path("eda_plots")`: Defines the directory for saving plots.
- `plot_dir.mkdir(exist_ok=True)`: Ensures the directory exists before saving plots.

# Function: plot_timestamps

```
def plot_timestamps(timestamps, name):
    if timestamps:
        print(f"\n{name} Temporal Coverage:")
        print(f"  Earliest: {min(timestamps)}")
        print(f"  Latest: {max(timestamps)}")
        plt.figure(figsize=(12, 4))
        sns.histplot([t.date() for t in timestamps], bins=30)
        plt.title(f"{name} Log Entries Over Time")
        plt.xlabel("Date")
        plt.ylabel("Count")
        plt.tight_layout()
        plt.savefig(plot_dir / f"{name.lower()}_temporal_coverage.png")
        plt.show()
    else:
        print(f"{name}: No timestamps found or format mismatch.")
```

## Purpose

- Centralized function to visualize the distribution of log entry timestamps for any dataset.
- Prints the earliest and latest timestamps and saves a histogram plot of daily log entry counts.

# Dataset-Specific Timestamp Extraction Functions

Each function is tailored to the unique timestamp format of a specific dataset. They all:

- Use a regex pattern to extract the timestamp substring from each log entry.
- Parse the substring into a datetime object using the appropriate format string.
- Collect valid timestamps and pass them to `plot_timestamps` for visualization.

## Examples

- `extract_apache_timestamps`: Matches `[Thu Jun 09 06:07:04 2005]` and parses with `%a %b %d %H:%M:%S %Y`.

- `extract_bgl_timestamps`: Matches `2005-06-03-15.42.50.363779` and parses with `%Y-%m-%d-%H.%M.%S.%f`.
- `extract_healthapp_timestamps`: Matches `20171223-22:15:29:606` and parses with `%Y%m%d-%H:%M:%S`.
- `extract_hpc_timestamps`: Matches Unix timestamps (10 digits) and converts with `datetime.fromtimestamp`.
- `extract_linux_timestamps` / `extract_mac_timestamps`: Matches `Jun  9 06:06:20` and parses with `%b %d %H:%M:%S %Y` (assumes year 2000).
- `extract_hdfs_timestamps`: Matches `081109 203615` and parses with `%y%m%d %H%M%S`.

## Usage Example

```
extract_apache_timestamps(apache_logs)
extract_bgl_timestamps(bgl_logs)
extract_healthapp_timestamps(healthapp_logs)
extract_hpc_timestamps(hpc_logs)
extract_linux_timestamps(linux_logs)
extract_mac_timestamps(mac_logs)
extract_hdfs_timestamps(hdfs_logs)
```

## Interpretation

- The histogram shows the volume of log entries per day, revealing periods of high or low activity.
- The earliest and latest timestamps indicate the time span covered by the dataset.
- If no timestamps are found, it may indicate a format mismatch or unusual log structure.

---

# Note: Why Dataset-Specific Extraction Works

- **Log format diversity:** Each dataset uses a different timestamp format and log structure. Generic extraction functions may miss timestamps or misinterpret data due to these differences.
- **Regex and format matching:** By writing a dedicated extraction function for each dataset, we ensure the regex and datetime format string precisely match the log's

structure, leading to accurate extraction and parsing.

- **Result:** This approach reliably extracts timestamps and enables temporal analysis for all datasets, whereas a one-size-fits-all method may fail or miss data due to format mismatches.

---

## Summary

- Dataset-specific timestamp extraction is essential for robust EDA when working with heterogeneous log sources.
- The shared `plot_timestamps` function standardizes visualization and reporting.
- This approach ensures accurate temporal coverage analysis and supports further time-based exploration and drift detection.

---

# Code Notes: Extensive Component Extraction and Analysis

## Imports

- `import re`: For regular expression matching to extract component names and identifiers from log entries.
- `from collections import Counter`: For counting occurrences of each component.
- `import matplotlib.pyplot as plt`: For plotting bar charts of component frequencies.
- `import seaborn as sns`: For enhanced statistical visualizations (used for bar plotting).

# Function: analyze_components_extensive

```python
def analyze_components_extensive(logs, name):
    """
    Extract and analyze component names from log entries using a comprehensive set of rege
    Saves the plot as an image in the 'eda_plots' directory.
    """
    patterns = [
        r'\[(.*?)\]',                                # [Component]
        r'^([A-Za-z0-9_-]+):',                       # Component:
        r'(?:INFO|ERROR|WARN)\s+([^:]+):',           # INFO Component:
        r'\(([\w.-]+)\)',                            # (Component)
        r'(?:daemon|server|client)\s+([\w.-]+)',  # daemon/server/client names
        r'blk_[-\d]+',                               # HDFS block IDs
        r'BP-[\d\-]+',                               # HDFS block pool IDs
        r'DFSClient_[\w.-]+',                        # DFS Client IDs
        r'NameNode',                                 # NameNode references
        r'DataNode',                                 # DataNode references
        r'FSNamesystem',                             # FSNamesystem references
        r'PacketResponder',                          # PacketResponder references
    ]
    components = []
    for log in logs:
        for pattern in patterns:
            found = re.findall(pattern, log)
            if found:
                if isinstance(found, list):
                    components.extend([comp for comp in found if comp])
                else:
                    components.append(found)
    if components:
        comp_counts = Counter(components)
        print(f"\n{name} Top Components (extensive extraction):")
        for comp, count in comp_counts.most_common(10):
            print(f"  {comp}: {count}")
        plt.figure(figsize=(10, 4))
        sns.barplot(x=list(comp_counts.keys())[:10], y=list(comp_counts.values())[:10])
        plt.title(f"{name} Top 10 Components (Extensive Extraction)")
        plt.ylabel("Count")
        plt.xticks(rotation=45)
        plt.tight_layout()
```

```
        plt.savefig(plot_dir / f"{name.lower()}_components.png")
        plt.show()
    else:
        print(f"{name}: No components found with extensive patterns.")
```

## Purpose

- Extracts component names, IDs, and references from log entries using a comprehensive set of regex patterns.
- Counts and visualizes the most frequent components for each dataset.
- Saves a bar plot of the top 10 components for reporting and further analysis.

## How it Works

- Defines a list of regex patterns to match various component formats (e.g., `[Component]`, `Component:`, block IDs, system references).
- Iterates through each log entry and applies all patterns, collecting all matches.
- Flattens and filters the results to build a list of extracted components.
- Uses `Counter` to tally the frequency of each component.
- Prints the top 10 components and their counts.
- Plots and saves a bar chart of the top 10 components in the plot directory.
- If no components are found, prints a warning message.

## Usage Example

```
analyze_components_extensive(mac_logs, "Mac")
analyze_components_extensive(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- The bar chart shows the most common components, IDs, or system references in the logs.
- Helps identify which parts of the system are most active, error-prone, or relevant for further analysis.
- Useful for understanding log structure, system architecture, and for feature engineering.

- If no components are found, it may indicate a need to adjust or expand the regex patterns for that dataset.

---

## Summary

- `analyze_components_extensive` is a powerful EDA tool for extracting and visualizing the diversity and frequency of components in heterogeneous log datasets.
- Supports deeper analysis of system behavior, anomaly detection, and feature extraction for downstream tasks.

---

# Code Notes: Log Entry Word Count Analysis and Visualization

## Imports

- `import numpy as np`: For calculating mean and standard deviation of word counts.
- `import matplotlib.pyplot as plt`: For plotting histograms of word counts.
- `import seaborn as sns`: For enhanced statistical visualizations (used for histogram plotting).

# Function: explore_word_count

```
def explore_word_count(logs, name):
    word_counts = [len(line.split()) for line in logs]
    print(f"\n{name} Word Count Stats:")
    print(f"  Min: {min(word_counts) if word_counts else 0}")
    print(f"  Max: {max(word_counts) if word_counts else 0}")
    print(f"  Mean: {np.mean(word_counts) if word_counts else 0:.2f}")
    print(f"  Std: {np.std(word_counts) if word_counts else 0:.2f}")
    plt.figure(figsize=(12, 4))
    sns.histplot(word_counts, bins=50, kde=True)
    plt.title(f"{name} Log Entry Word Count Distribution")
    plt.xlabel("Word Count")
    plt.ylabel("Count")
    plt.tight_layout()
    plt.savefig(plot_dir / f"{name.lower()}_word_count.png")
    plt.show()
```

## Purpose

- Analyzes the distribution of word counts in log entries for a given dataset.
- Prints summary statistics (min, max, mean, std) for word counts.
- Visualizes the distribution as a histogram with a KDE (Kernel Density Estimate) overlay.
- Saves the plot for reporting and further analysis.

## How it Works

- Splits each log entry into words and counts them, building a list of word counts.
- Calculates and prints summary statistics: minimum, maximum, mean, and standard deviation.
- Plots a histogram (with KDE) of word counts using seaborn/matplotlib.
- Saves the plot as a PNG in the plot directory.

### Usage Example

```
explore_word_count(hdfs_logs, "HDFS")
explore_word_count(apache_logs, "APACHE")
# ...repeat for other datasets
```

### Interpretation

- The histogram shows the typical length of log entries in terms of word count.
- Summary statistics help identify outliers (very short or very long entries) and structural differences between datasets.
- Useful for understanding log verbosity, structure, and for feature engineering.

---

## Summary

- `explore_word_count` is a straightforward EDA tool for quantifying and visualizing the textual complexity of log entries.
- Helps identify dataset-specific patterns and supports downstream analysis and feature selection.

---

# Code Notes: Special Character Count Analysis and Visualization

## Imports

- `import string`: For accessing lists of ASCII letters and digits to identify special characters.
- `import numpy as np`: For calculating mean and standard deviation of special character counts.

- `import matplotlib.pyplot as plt`: For plotting histograms of special character counts.
- `import seaborn as sns`: For enhanced statistical visualizations (used for histogram plotting).

# Function: explore_special_characters

```
def explore_special_characters(logs, name):
    special_counts = [sum(1 for c in line if c not in string.ascii_letters + string.digits
    print(f"\n{name} Special Character Stats:")
    print(f"  Min: {min(special_counts) if special_counts else 0}")
    print(f"  Max: {max(special_counts) if special_counts else 0}")
    print(f"  Mean: {np.mean(special_counts) if special_counts else 0:.2f}")
    print(f"  Std: {np.std(special_counts) if special_counts else 0:.2f}")
    plt.figure(figsize=(12, 4))
    sns.histplot(special_counts, bins=50, kde=True)
    plt.title(f"{name} Special Character Count Distribution")
    plt.xlabel("Special Character Count")
    plt.ylabel("Count")
    plt.tight_layout()
    plt.savefig(plot_dir / f"{name.lower()}_special_chars.png")
    plt.show()
```

## Purpose

- Analyzes the distribution of special (non-alphanumeric) characters in log entries for a given dataset.
- Prints summary statistics (min, max, mean, std) for special character counts.
- Visualizes the distribution as a histogram with a KDE (Kernel Density Estimate) overlay.
- Saves the plot for reporting and further analysis.

## How it Works

- For each log entry, counts the number of characters that are not ASCII letters, digits, or spaces.
- Builds a list of special character counts for all entries.

- Calculates and prints summary statistics: minimum, maximum, mean, and standard deviation.
- Plots a histogram (with KDE) of special character counts using seaborn/matplotlib.
- Saves the plot as a PNG in the plot directory.

### Usage Example

```
explore_special_characters(hdfs_logs, "HDFS")
explore_special_characters(apache_logs, "APACHE")
# ...repeat for other datasets
```

### Interpretation

- The histogram shows how many special characters are typically present in log entries.
- Summary statistics help identify outliers (entries with many special characters) and structural differences between datasets.
- Useful for understanding log formatting, presence of encoded data, or unusual symbols that may affect parsing or analysis.

---

## Summary

- `explore_special_characters` is a useful EDA tool for quantifying and visualizing the presence of non-alphanumeric characters in log entries.
- Helps identify formatting patterns, anomalies, and supports preprocessing and feature engineering for downstream analysis.

---

# Code Notes: Unique Component Count Analysis

## Imports

- `import re`: For regular expression matching to extract component names and identifiers from log entries.

## Function: unique_component_count

```python
def unique_component_count(logs, name):
    # Use the same extraction logic as your component diversity function
    patterns = [
        r'\[(.*?)\]', r'^([A-Za-z0-9_-]+):', r'(?:INFO|ERROR|WARN)\s+([^:]+):',
        r'\(([\w.-]+)\)', r'(?:daemon|server|client)\s+([\w.-]+)', r'blk_[-\d]+',
        r'BP-[\d\-]+', r'DFSClient_[\w.-]+', r'NameNode', r'DataNode',
        r'FSNamesystem', r'PacketResponder'
    ]
    components = set()
    for log in logs:
        for pattern in patterns:
            found = re.findall(pattern, log)
            if found:
                if isinstance(found, list):
                    components.update([comp for comp in found if comp])
                else:
                    components.add(found)
    print(f"{name}: {len(components)} unique components found.")
```

## Purpose

- Counts the number of unique components, IDs, or system references present in a log dataset.
- Uses a comprehensive set of regex patterns to capture a wide variety of component

formats.
- Provides a measure of component diversity for each dataset.

## How it Works

- Defines a list of regex patterns to match different component formats:
    - `r'\[(.*?)\]'` : Matches anything inside square brackets (e.g., `[Component]`).
    - `r'^([A-Za-z0-9_-]+):'` : Matches a component at the start of a line followed by a colon (e.g., `Component:`).
    - `r'(?:INFO|ERROR|WARN)\s+([^:]+):'` : Matches log level followed by a component and colon (e.g., `INFO Component:`).
    - `r'\(([\w.-]+)\)'` : Matches anything inside parentheses (e.g., `(Component)`).
    - `r'(?:daemon|server|client)\s+([\w.-]+)'` : Matches daemon/server/client names (e.g., `server Component`).
    - `r'blk_[-\d]+'` : Matches HDFS block IDs (e.g., `blk_-12345`).
    - `r'BP-[\d\-]+'` : Matches HDFS block pool IDs (e.g., `BP-1234-5678`).
    - `r'DFSClient_[\w.-]+'` : Matches DFS client IDs (e.g., `DFSClient_abc123`).
    - `r'NameNode'` , `r'DataNode'` , `r'FSNamesystem'` , `r'PacketResponder'` : Matches specific system component names.
- Iterates through each log entry and applies all patterns, collecting all unique matches in a set.
- Prints the total number of unique components found.

## Usage Example

```
unique_component_count(hdfs_logs, "HDFS")
unique_component_count(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- The count reflects the diversity of components referenced in the logs.
- High uniqueness may indicate a complex or highly modular system, or a large number of unique IDs.
- Low uniqueness may indicate a focused or repetitive log structure.

- Useful for feature engineering, anomaly detection, and understanding system architecture.

---

## Summary

- `unique_component_count` is a quick EDA tool for quantifying the diversity of components in log datasets.
- The comprehensive regex patterns ensure broad coverage of different log formats and component types.
- Supports deeper analysis of system structure and log variability.

---

# Code Notes: Numerical Value Extraction and Analysis

## Imports

- `import re`: For regular expression matching to extract numerical values from log entries.
- `import numpy as np`: For calculating mean and standard deviation of numerical value counts.
- `import matplotlib.pyplot as plt`: For plotting histograms of numerical value counts.
- `import seaborn as sns`: For enhanced statistical visualizations (used for histogram plotting).

## Function: extract_numerical_values

```
def extract_numerical_values(message):
    return [float(num) for num in re.findall(r'\d+(?:\.\d+)?', message)]
```

## Purpose

- Extracts all numerical values (integers and decimals) from a log entry string.
- Uses a regex pattern to match both whole numbers and floating-point numbers.

## Regex Pattern Explanation

- `r'\d+(?:.\d+)?':
  - `\d+` matches one or more digits (an integer part).
  - `(?:\.\d+)?` is a non-capturing group for an optional decimal part (a period followed by one or more digits).
  - This pattern matches numbers like `42`, `3.14`, `1000`, etc.

# Function: analyze_numerical_values

```
def analyze_numerical_values(logs, name):
    num_counts = [len(extract_numerical_values(line)) for line in logs]
    print(f"\n{name} Numerical Value Stats:")
    print(f"  Min: {min(num_counts) if num_counts else 0}")
    print(f"  Max: {max(num_counts) if num_counts else 0}")
    print(f"  Mean: {np.mean(num_counts) if num_counts else 0:.2f}")
    print(f"  Std: {np.std(num_counts) if num_counts else 0:.2f}")
    plt.figure(figsize=(12, 4))
    sns.histplot(num_counts, bins=50, kde=True)
    plt.title(f"{name} Numerical Value Count Distribution")
    plt.xlabel("Numerical Value Count")
    plt.ylabel("Count")
    plt.tight_layout()
    plt.savefig(plot_dir / f"{name.lower()}_numerical_values.png")
    plt.show()
```

## Purpose

- Analyzes the distribution of numerical value counts in log entries for a given dataset.
- Prints summary statistics (min, max, mean, std) for the number of numerical values per entry.
- Visualizes the distribution as a histogram with a KDE (Kernel Density Estimate) overlay.

- Saves the plot for reporting and further analysis.

## How it Works

- For each log entry, uses `extract_numerical_values` to find all numbers and counts them.
- Builds a list of numerical value counts for all entries.
- Calculates and prints summary statistics: minimum, maximum, mean, and standard deviation.
- Plots a histogram (with KDE) of numerical value counts using seaborn/matplotlib.
- Saves the plot as a PNG in the plot directory.

## Usage Example

```
analyze_numerical_values(hdfs_logs, "HDFS")
analyze_numerical_values(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- The histogram shows how many numerical values are typically present in log entries.
- Summary statistics help identify outliers (entries with many numbers) and structural differences between datasets.
- Useful for understanding log content, identifying parameter-rich entries, and for feature engineering.

---

# Summary

- `extract_numerical_values` and `analyze_numerical_values` are useful EDA tools for quantifying and visualizing the presence of numerical data in log entries.
- Helps identify formatting patterns, anomalies, and supports preprocessing and feature engineering for downstream analysis.

---

# Code Notes: Rare Component Detection in Log Entries

## Imports

- `import re`: For regular expression matching to extract component names and identifiers from log entries.
- `from collections import Counter`: For counting occurrences of each component.

## Function: detect_rare_components

```python
def detect_rare_components(logs, name, min_count=2):
    patterns = [
        r'\[(.*?)\]', r'^([A-Za-z0-9_-]+):', r'(?:INFO|ERROR|WARN)\s+([^:]+):',
        r'\(([\w.-]+)\)', r'(?:daemon|server|client)\s+([\w.-]+)', r'blk_[-\d]+',
        r'BP-[\d\-]+', r'DFSClient_[\w.-]+', r'NameNode', r'DataNode',
        r'FSNamesystem', r'PacketResponder'
    ]
    components = []
    for log in logs:
        for pattern in patterns:
            found = re.findall(pattern, log)
            if found:
                if isinstance(found, list):
                    components.extend([comp for comp in found if comp])
                else:
                    components.append(found)
    comp_counts = Counter(components)
    rare = [comp for comp, count in comp_counts.items() if count <= min_count]
    print(f"{name}: {len(rare)} rare components (appearing <= {min_count} times)")
    if rare:
        print("Sample rare components:", rare[:10])
```

## Purpose

- Identifies and counts components that appear infrequently (rare components) in a log dataset.
- Uses a comprehensive set of regex patterns to extract a wide variety of component formats.
- Helps highlight unusual, infrequent, or potentially anomalous system elements.

## How it Works

- Defines a list of regex patterns to match different component formats (see previous notes for pattern explanations).
- Iterates through each log entry and applies all patterns, collecting all matches in a list.
- Uses `Counter` to tally the frequency of each component.
- Identifies components whose count is less than or equal to `min_count` (default: 2).
- Prints the number of rare components and a sample list of up to 10 rare component names.

## Usage Example

```
detect_rare_components(hdfs_logs, "HDFS")
detect_rare_components(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- The count of rare components reflects the presence of infrequent or unique system elements in the logs.
- Rare components may indicate:
  - Outliers or anomalies
  - Rarely used features or modules
  - Typos or inconsistent naming
- Useful for anomaly detection, system auditing, and understanding the long tail of system activity.

---

## Summary

- `detect_rare_components` is a valuable EDA tool for surfacing infrequent or unique components in log datasets.
- Supports anomaly detection, system health monitoring, and deeper investigation of unusual log activity.

---

# Code Notes: Log Entry Uniqueness Analysis

## Function: analyze_uniqueness

```
def analyze_uniqueness(logs, name):
    unique_count = len(set(logs))
    total_count = len(logs)
    print(f"{name}: {unique_count}/{total_count} unique entries ({(unique_count/total_cou
```

### Purpose

- Calculates the number and percentage of unique log entries in a dataset.
- Provides a quick measure of redundancy, repetition, or diversity in log data.

### How it Works

- Converts the list of log entries to a set to remove duplicates and count unique entries.
- Calculates the total number of entries and the number of unique entries.
- Prints the count and percentage of unique entries for the dataset.

## Usage Example

```
analyze_uniqueness(hdfs_logs, "HDFS")
analyze_uniqueness(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- A high percentage of unique entries suggests high variability, possibly due to dynamic content, parameters, or diverse events.
- A low percentage indicates repetitive or templated logs, which may be easier to cluster or compress.
- Useful for understanding log structure, guiding feature engineering, and identifying datasets with high or low entropy.

---

## Summary

- `analyze_uniqueness` is a simple but effective EDA tool for quantifying the diversity of log entries in a dataset.
- Helps guide downstream analysis, such as clustering, deduplication, or anomaly detection.

---

# Code Notes: Log Entry Clustering with TF-IDF, SVD, and DBSCAN

## Imports

- `import numpy as np`: For numerical operations, random subsampling, and array indexing.

- `import matplotlib.pyplot as plt`: For plotting cluster size distributions.
- `from sklearn.feature_extraction.text import TfidfVectorizer`: For converting log messages into TF-IDF feature vectors.
- `from sklearn.decomposition import TruncatedSVD`: For dimensionality reduction of high-dimensional TF-IDF vectors.
- `from sklearn.cluster import DBSCAN`: For density-based clustering of log entries.

# Function: cluster_log_entries_dbscan

```
def cluster_log_entries_dbscan(
    logs, name, eps=0.7, min_samples=10, max_samples=5000, max_features=300, use_svd=True
):
    if not logs:
        print(f"No logs to cluster for {name}.")
        return
    # Subsample for speed
    if len(logs) > max_samples:
        idx = np.random.choice(len(logs), max_samples, replace=False)
        logs_sample = [logs[i] for i in idx]
        print(f"Subsampling {max_samples} entries from {len(logs)} for {name}.")
    else:
        logs_sample = logs

    # TF-IDF vectorization
    vectorizer = TfidfVectorizer(max_features=max_features, stop_words='english')
    X = vectorizer.fit_transform(logs_sample)

    # Optional dimensionality reduction
    if use_svd and X.shape[1] > n_components:
        svd = TruncatedSVD(n_components=n_components, random_state=42)
        X_reduced = svd.fit_transform(X)
    else:
        X_reduced = X

    # DBSCAN clustering
    dbscan = DBSCAN(eps=eps, min_samples=min_samples, metric='cosine', n_jobs=-1)
    labels = dbscan.fit_predict(X_reduced)
    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    n_noise = list(labels).count(-1)
    print(f"\n{name} DBSCAN Clustering Results:")
    print(f"Clusters found: {n_clusters}")
    print(f"Noise points (potential anomalies): {n_noise}")

    for i in set(labels):
        if i == -1:
            continue
        cluster_indices = np.where(labels == i)[0]
        print(f"\nCluster {i} - Size: {len(cluster_indices)}")
```

```
        for idx2 in cluster_indices[:3]:
            print(f"  - {logs_sample[idx2][:100]}{'...' if len(logs_sample[idx2]) > 100 el

    cluster_sizes = [np.sum(labels == i) for i in set(labels) if i != -1]
    plt.figure(figsize=(8, 4))
    plt.bar(range(len(cluster_sizes)), cluster_sizes)
    plt.title(f"{name} DBSCAN Cluster Sizes (Excluding Noise)")
    plt.xlabel("Cluster")
    plt.ylabel("Number of Entries")
    plt.tight_layout()
    plt.show()
```

## Purpose

- Groups log entries into clusters based on textual similarity using TF-IDF features, optional SVD dimensionality reduction, and DBSCAN clustering.
- Identifies dense regions (clusters) and outliers (noise/anomalies) in the log data.
- Visualizes the size of each cluster (excluding noise) and prints sample messages for interpretation.

## How it Works

- **Subsampling for Large Data:** If the dataset is very large, randomly selects up to `max_samples` entries to reduce computation time and memory usage. This makes clustering feasible on large log datasets.
- **TF-IDF Vectorization:** Converts log messages into a matrix of TF-IDF features (up to `max_features` features, with English stop words removed).
- **SVD Dimensionality Reduction:** Optionally reduces the dimensionality of the TF-IDF matrix to `n_components` using TruncatedSVD, which speeds up clustering and reduces noise in high-dimensional data.
- **DBSCAN Clustering:** Applies DBSCAN with cosine distance, grouping similar entries into clusters and labeling outliers as noise (`-1`).
- Prints the number of clusters found, the number of noise points, and sample messages from each cluster.
- Plots and displays a bar chart of cluster sizes (excluding noise).

## Parameter Rationale

- `logs` : The list of log entries to cluster.
- `name` : Dataset name for labeling outputs.
- `eps=0.7` : DBSCAN neighborhood radius; controls cluster density sensitivity (tune for your data).
- `min_samples=10` : Minimum number of points to form a cluster (tune for your data size and expected cluster size).
- `max_samples=5000` : Maximum number of log entries to use for clustering (prevents memory/computation issues on large datasets).
- `max_features=300` : Maximum number of TF-IDF features (reduces dimensionality and noise).
- `use_svd=True` : Whether to apply SVD dimensionality reduction (recommended for high-dimensional data).
- `n_components=50` : Number of SVD components to keep (controls the reduced feature space size).

## Usage Example

```
cluster_log_entries_dbscan(hdfs_logs, "HDFS")
cluster_log_entries_dbscan(apache_logs, "APACHE")
# ...repeat for other datasets
```

## Interpretation

- Each cluster represents a group of log entries with similar content or structure.
- Noise points (label `-1`) are potential anomalies or outliers.
- The size of each cluster indicates the prevalence of that log type or pattern.
- Sample messages help interpret the nature of each cluster (e.g., errors, info, specific events).
- Useful for summarizing log data, identifying dominant patterns, and supporting downstream tasks like anomaly detection or drift analysis.

---

## Summary

- `cluster_log_entries_dbscan` is a robust EDA tool for unsupervised grouping of log messages, especially in large datasets.
- Subsampling, TF-IDF, SVD, and DBSCAN together provide scalable, interpretable clustering with anomaly detection capabilities.
- Visual and textual outputs support reporting, interpretation, and further analysis.

---

# Code Notes: t-SNE Visualization of Log Entry Clusters

## Imports

- `import numpy as np`: For numerical operations, random subsampling, and array indexing.
- `import matplotlib.pyplot as plt`: For plotting the t-SNE scatter plot.
- `from sklearn.feature_extraction.text import TfidfVectorizer`: For converting log messages into TF-IDF feature vectors.
- `from sklearn.decomposition import TruncatedSVD`: For dimensionality reduction of high-dimensional TF-IDF vectors.
- `from sklearn.cluster import KMeans`: For clustering log entries before visualization.
- `from sklearn.manifold import TSNE`: For non-linear dimensionality reduction and visualization.
- `from pathlib import Path`: For handling file paths (used for plot saving).

# Function: plot_tsne

```python
def plot_tsne(logs, name, n_clusters=5, max_samples=2000, max_features=200, svd_components
    # Subsample for speed/memory
    if len(logs) > max_samples:
        idx = np.random.choice(len(logs), max_samples, replace=False)
        logs_sample = [logs[i] for i in idx]
        print(f'Subsampling {max_samples} entries from {len(logs)} for {name}.')
    else:
        logs_sample = logs

    # TF-IDF vectorization
    vectorizer = TfidfVectorizer(max_features=max_features, stop_words='english')
    X = vectorizer.fit_transform(logs_sample)

    # SVD for further reduction
    if X.shape[1] > svd_components:
        svd = TruncatedSVD(n_components=svd_components, random_state=42)
        X_reduced = svd.fit_transform(X)
    else:
        X_reduced = X.toarray()

    # KMeans clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X_reduced)

    # t-SNE embedding
    tsne = TSNE(n_components=2, random_state=42, init='pca', learning_rate='auto')
    X_embedded = tsne.fit_transform(X_reduced)

    # Plot
    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=labels, cmap='tab10', alpl
    plt.title(f'{name} Log Entry Clusters (t-SNE Projection)')
    plt.xlabel('t-SNE 1')
    plt.ylabel('t-SNE 2')
    plt.tight_layout()
    plt.savefig(plot_dir / f'{name.lower()}_tsne_clusters.png')
    plt.show()
```

## Purpose

- Visualizes the structure of log entry clusters in two dimensions using t-SNE, after clustering with KMeans.
- Helps interpret the separability and relationships between clusters in high-dimensional log data.
- Provides an intuitive, visual summary of log diversity and cluster structure.

## How it Works

- **Subsampling for Large Data:** If the dataset is very large, randomly selects up to `max_samples` entries to reduce computation time and memory usage. This makes t-SNE feasible and the plot interpretable.
- **TF-IDF Vectorization:** Converts log messages into a matrix of TF-IDF features (up to `max_features` features, with English stop words removed).
- **SVD Dimensionality Reduction:** Optionally reduces the dimensionality of the TF-IDF matrix to `svd_components` using TruncatedSVD, which speeds up t-SNE and reduces noise in high-dimensional data.
- **KMeans Clustering:** Assigns each log entry to one of `n_clusters` clusters for coloring in the plot.
- **t-SNE Embedding:** Projects the reduced feature matrix into two dimensions for visualization, preserving local structure.
- Plots and saves a scatter plot of the t-SNE embedding, colored by cluster label.

## Parameter Rationale

- `logs`: The list of log entries to visualize.
- `name`: Dataset name for labeling outputs.
- `n_clusters=5`: Number of clusters for KMeans (for coloring in the plot).
- `max_samples=2000`: Maximum number of log entries to use for visualization (prevents memory/computation issues on large datasets).
- `max_features=200`: Maximum number of TF-IDF features (reduces dimensionality and noise).
- `svd_components=50`: Number of SVD components to keep (controls the reduced feature space size for t-SNE).

## Usage Example

```
plot_tsne(hdfs_logs, 'HDFS')
plot_tsne(apache_logs, 'APACHE')
# ...repeat for other datasets
```

## Interpretation

- Each point represents a log entry, colored by its assigned cluster.
- Well-separated clusters indicate distinct log types or patterns.
- Overlapping or diffuse clusters may indicate ambiguity or noise in the data.
- Useful for summarizing log data, identifying dominant patterns, and supporting downstream tasks like anomaly detection or drift analysis.

---

## Summary

- `plot_tsne` is a powerful EDA tool for visualizing the structure and diversity of log datasets in two dimensions.
- Subsampling, TF-IDF, SVD, KMeans, and t-SNE together provide scalable, interpretable visualizations for large and complex log data.
- Visual outputs support reporting, interpretation, and further analysis.

---

# Code Notes: Log DataFrame Construction and Feature Extraction

## Imports

- `import pandas as pd`: For building and manipulating the DataFrame.
- `import re`: For regular expression matching to extract features from log entries.

- `from datetime import datetime`: For parsing and handling timestamps.
- `from pathlib import Path`: For file path handling (not directly used in this snippet, but useful for file operations).

# Feature Extraction Functions

- `extract_timestamp(log)`: Extracts a timestamp from a log entry using multiple regex patterns and formats. Returns a `datetime` object or `pd.NaT` if not found.
- `extract_message_type(log)`: Extracts the message type (INFO, ERROR, etc.) from a log entry. Returns 'OTHER' if not found.
- `extract_component(log)`: Extracts the component name from a log entry if present (e.g., `component=XYZ`). Returns 'Unspecified' if not found.

# Workflow: Building the DataFrame

1. **Combine All Logs:**
   - Iterates over all datasets, creating a list of dictionaries with fields: `dataset` (name) and `raw` (log entry).
2. **Create DataFrame:**
   - Converts the list of dictionaries into a pandas DataFrame (`df`).
3. **Feature Extraction:**
   - Adds columns to the DataFrame by applying extraction functions to the `raw` log entry:
     - `timestamp`: Extracted timestamp (datetime or NaT).
     - `message_type`: Extracted message type (INFO, ERROR, etc.).
     - `component`: Extracted component name or 'Unspecified'.
     - `entry_length`: Length of the log entry (number of characters).
     - `word_count`: Number of words in the log entry.
4. **Timestamp Cleaning:**
   - Drops rows with missing timestamps and ensures all timestamps are in pandas datetime format.

# DataFrame Fields/Columns

- `dataset`: Name of the dataset (e.g., 'HDFS', 'Apache').
- `raw`: The original log entry string.
- `timestamp`: Parsed datetime object for the log entry (or NaT if not found).
- `message_type`: Log message type (INFO, ERROR, WARNING, DEBUG, CRITICAL, OTHER).
- `component`: Extracted component name or 'Unspecified'.
- `entry_length`: Number of characters in the log entry.
- `word_count`: Number of words in the log entry.

# Expected Shape

- **Rows:** One per log entry with a valid timestamp (after dropping NaT timestamps). The total number of rows is the sum of all log entries across all datasets, minus those without a valid timestamp.
- **Columns:** 7 (dataset, raw, timestamp, message_type, component, entry_length, word_count).

# Purpose and Role in EDA

- The DataFrame provides a structured, tabular view of all log entries and their extracted features, enabling efficient analysis, visualization, and modeling.
- Facilitates filtering, grouping, and statistical analysis by dataset, message type, component, or time.
- Serves as a foundation for downstream tasks such as clustering, drift detection, anomaly detection, and reporting.

---

# Summary

- This DataFrame construction step is essential for transforming raw, heterogeneous log data into a unified, feature-rich format suitable for comprehensive EDA and machine

learning workflows.
- The resulting DataFrame enables rapid exploration, visualization, and interpretation of log data across multiple datasets and feature dimensions.

---

# Code Notes: Error Flag and Windowed Feature Extraction

## Error Flag Column

- `df['error_flag'] = (df['message_type'] == 'ERROR').astype(int)`
    - Creates a binary column indicating whether each log entry is an error (1) or not (0).
    - **Purpose:** Enables calculation of error rates over time windows, which is a key indicator of system health and operational issues.
    - **Why use error_flag?**
        - Error rates are a direct, interpretable measure of system problems or incidents.
        - Tracking error frequency over time helps detect drifts, spikes, or anomalies in log behavior.
        - Binary encoding makes it easy to aggregate and compute statistics (e.g., mean error rate per window).

# Function: extract_windowed_features

```
def extract_windowed_features(df, time_col='timestamp', window='1D'):
    df = df.copy()
    df[time_col] = pd.to_datetime(df[time_col])
    df = df.sort_values(time_col)
    df.set_index(time_col, inplace=True)
    features = df.resample(window).agg({
        'word_count': ['mean', 'std'],
        'error_flag': 'mean',  # error rate
        'message_type': lambda x: x.value_counts().to_dict()
    })
    return features
```

## Purpose

- Aggregates log features over fixed time windows (e.g., daily) to enable time series analysis and drift detection.
- Computes summary statistics for each window, such as average word count, error rate, and message type distribution.

## How it Works

- Converts the timestamp column to datetime and sorts the DataFrame.
- Sets the timestamp as the index for resampling.
- Resamples the DataFrame by the specified window (default: 1 day).
- Aggregates:
  - `word_count`: Mean and standard deviation (measures verbosity and variability).
  - `error_flag`: Mean (proportion of error entries, i.e., error rate).
  - `message_type`: Distribution of message types in each window (as a dictionary).
- Returns a DataFrame indexed by time window, with aggregated features as columns.

## Why Focus on Error Rate?

- The error rate is a highly interpretable, actionable metric for system monitoring.
- Spikes or changes in error rate often correspond to operational incidents, failures, or

drifts.
- While other features (e.g., component diversity, word count) are useful, error rate provides a direct signal for alerting and root cause analysis.
- Focusing on a single, meaningful component simplifies visualization and interpretation, especially for time series and drift detection tasks.

## Resulting DataFrame

- **Index:** Time window (e.g., each day).
- **Columns:**
    - `word_count_mean`, `word_count_std`: Average and variability of log entry length per window.
    - `error_flag_mean`: Proportion of error entries (error rate) per window.
    - `message_type_<lambda>`: Dictionary of message type counts per window.

# Role in EDA and Drift Detection

- Enables time series analysis of log features, supporting detection of drifts, anomalies, or operational changes.
- Facilitates visualization of trends (e.g., error rate over time) and correlation with incidents.
- Provides a compact, interpretable summary of log behavior for reporting and further analysis.

---

# Summary

- Creating an `error_flag` and extracting windowed features are essential steps for time-based log analysis and drift detection.
- The approach balances interpretability (error rate) with flexibility (other features), supporting robust monitoring and incident response.

---

# Code Notes: Kolmogorov-Smirnov Test for Word Count Drift

## Imports

- `from scipy.stats import ks_2samp`: For performing the two-sample Kolmogorov-Smirnov (KS) test to compare distributions.

## Function: ks_test_word_count

```
def ks_test_word_count(df, time_col='timestamp', window='1D'):
    df = df.copy()
    df[time_col] = pd.to_datetime(df[time_col])
    df = df.sort_values(time_col)
    df.set_index(time_col, inplace=True)
    windows = list(df.resample(window))
    print(f"KS test for word count drift between consecutive {window} windows:")
    for i in range(len(windows)-1):
        w1 = windows[i][1]['word_count']
        w2 = windows[i+1][1]['word_count']
        if len(w1) > 0 and len(w2) > 0:
            stat, p = ks_2samp(w1, w2)
            print(f"Window {i} vs {i+1}: KS stat={stat:.3f}, p={p:.3f}")
```

### Purpose

- Detects distributional drift in log entry word counts over time using the Kolmogorov-Smirnov (KS) test.
- Compares the distribution of word counts between consecutive time windows (e.g., days).
- Identifies statistically significant changes in log structure or verbosity.

## How it Works

- Converts the timestamp column to datetime, sorts, and sets it as the index.
- Resamples the DataFrame into consecutive time windows (default: 1 day).
- For each pair of consecutive windows, extracts the `word_count` series.
- Applies the two-sample KS test ( `ks_2samp` ) to compare the distributions of word counts between the two windows.
- Prints the KS statistic and p-value for each window pair.

## Kolmogorov-Smirnov Test

- The KS test is a non-parametric test that measures the maximum difference between the empirical cumulative distribution functions (ECDFs) of two samples.
- The test statistic ( `stat` ) quantifies the difference between distributions; the p-value ( `p` ) indicates whether the difference is statistically significant.
- A low p-value (e.g., $< 0.05$) suggests a significant drift or change in the distribution between windows.

## Usage Example

```
ks_test_word_count(df)
```

## Interpretation

- High KS statistic and low p-value between windows indicate a significant change in log entry word count distribution (potential drift).
- Useful for detecting operational changes, incidents, or shifts in log generation patterns.
- Can be extended to other features (e.g., entry length, error rate) for comprehensive drift detection.

---

# Summary

- `ks_test_word_count` is a practical tool for statistical drift detection in log data, leveraging the KS test to compare feature distributions over time.

- Supports robust monitoring, anomaly detection, and incident analysis in EDA workflows.

---

# Code Notes: Chi-Squared Test for Message Type Drift

## Imports

- `from scipy.stats import chi2_contingency`: For performing the chi-squared test of independence on categorical data.
- `import numpy as np`: For numerical operations and array construction.

# Function: chi2_test_message_type

```
def chi2_test_message_type(df, time_col='timestamp', window='1D'):
    df = df.copy()
    df[time_col] = pd.to_datetime(df[time_col])
    df = df.sort_values(time_col)
    df.set_index(time_col, inplace=True)
    windows = list(df.resample(window))
    print(f"Chi-squared test for message type drift between consecutive {window} windows:"
    for i in range(len(windows)-1):
        w1 = windows[i][1]['message_type'].value_counts()
        w2 = windows[i+1][1]['message_type'].value_counts()
        all_types = set(w1.index).union(w2.index)
        obs = np.array([
            [w1.get(t, 0) for t in all_types],
            [w2.get(t, 0) for t in all_types]
        ])
        # Remove columns where both windows have zero counts
        obs = obs[:, ~(obs == 0).all(axis=0)]
        if obs.shape[1] > 0 and obs.sum() > 0:
            try:
                chi2, p, _, _ = chi2_contingency(obs)
                print(f"Window {i} vs {i+1}: Chi2={chi2:.2f}, p={p:.3f}")
            except ValueError as e:
                print(f"Window {i} vs {i+1}: Skipped due to error: {e}")
```

## Purpose

- Detects drift in the distribution of log message types (e.g., INFO, ERROR, WARNING) over time using the chi-squared test of independence.
- Compares the frequency distribution of message types between consecutive time windows.
- Identifies statistically significant changes in log event composition.

## How it Works

- Converts the timestamp column to datetime, sorts, and sets it as the index.

- Resamples the DataFrame into consecutive time windows (default: 1 day).
- For each pair of consecutive windows, counts the occurrences of each message type.
- Constructs a contingency table (2 x N) for the two windows, where N is the number of unique message types.
- Removes columns where both windows have zero counts to avoid errors.
- Applies the chi-squared test ( `chi2_contingency` ) to the contingency table.
- Prints the chi-squared statistic and p-value for each window pair.
- Handles errors gracefully if the contingency table is not valid.

## Chi-Squared Test

- The chi-squared test of independence assesses whether the distribution of categorical variables (message types) differs between two samples (windows).
- The test statistic ( `chi2` ) quantifies the difference between observed and expected frequencies; the p-value ( `p` ) indicates whether the difference is statistically significant.
- A low p-value (e.g., < 0.05) suggests a significant drift or change in message type distribution between windows.

## Usage Example

```
chi2_test_message_type(df)
```

## Interpretation

- High chi-squared statistic and low p-value between windows indicate a significant change in the distribution of message types (potential drift).
- Useful for detecting operational changes, incidents, or shifts in log event composition.
- Can be used alongside other drift detection methods (e.g., KS test) for comprehensive monitoring.

---

# Summary

- `chi2_test_message_type` is a practical tool for categorical drift detection in log data,

leveraging the chi-squared test to compare message type distributions over time.
- Supports robust monitoring, anomaly detection, and incident analysis in EDA workflows.

---

# Code Notes: Change Point Detection on Error Rate

## Imports

- `import ruptures as rpt`: For performing change point detection in time series data.
- `import matplotlib.pyplot as plt`: For plotting error rate and detected change points.

# Function: change_point_detection_error_rate

```
def change_point_detection_error_rate(df, time_col='timestamp', window='7D', pen=5, model=
    df = df.copy()
    df[time_col] = pd.to_datetime(df[time_col])
    df = df.sort_values(time_col)
    df.set_index(time_col, inplace=True)
    error_rate = df['error_flag'].resample(window).mean().fillna(0).values
    if downsample > 1:
        error_rate = error_rate[::downsample]
    algo = rpt.Pelt(model=model).fit(error_rate)
    result = algo.predict(pen=pen)
    plt.figure(figsize=(10, 4))
    plt.plot(error_rate, label='Error Rate')
    for cp in result[:-1]:
        plt.axvline(cp, color='red', linestyle='--')
    plt.title('Change Point Detection on Error Rate')
    plt.xlabel('Window')
    plt.ylabel('Error Rate')
    plt.legend()
    plt.tight_layout()
    plt.show()
    print(f"Detected change points at windows: {result}")
```

## Purpose

- Detects abrupt changes (change points) in the error rate time series using the `ruptures` library.
- Identifies periods where the system's error behavior shifts, which may correspond to incidents or operational changes.
- Visualizes error rate over time with detected change points marked.

## How it Works

- Converts the timestamp column to datetime, sorts, and sets it as the index.
- Resamples the DataFrame to compute mean error rate per window (default: 7 days).
- Optionally downsamples the error rate series for computational efficiency.

- Uses the PELT algorithm from `ruptures` to detect change points in the error rate series, with a specified penalty ( `pen` ) and model ( `model` ).
- Plots the error rate time series and overlays vertical lines at detected change points.
- Prints the indices of detected change points.

## Parameter Rationale

- `df` : The DataFrame containing log data and error flags.
- `time_col` : The column containing timestamps (default: 'timestamp').
- `window` : Resampling window for error rate calculation (e.g., '7D' for 7 days).
- `pen` : Penalty value for change point detection (higher = fewer change points).
- `model` : Cost function model for change point detection (e.g., 'l2' for least squares).
- `downsample` : Factor to reduce the number of points for faster computation (default: 1, i.e., no downsampling).

## Usage Example

```
change_point_detection_error_rate(df, window='7D', pen=5, model='l2', downsample=1)
```

## Interpretation

- Vertical red dashed lines indicate detected change points—moments where the error rate behavior shifts.
- Useful for pinpointing incidents, regime changes, or periods of instability in system logs.
- The number and location of change points can be tuned by adjusting the penalty ( `pen` ).
- Supports root cause analysis and incident response by highlighting when and where log behavior changes.

---

# Summary

- `change_point_detection_error_rate` is a powerful tool for time series drift and incident detection in log data.
- Combines error rate analysis with robust change point detection and clear visualizations

for actionable insights.

# Code Notes: Combined Drift Window Detection (KS + Chi2)

## Function: detect_drift_windows

```python
def detect_drift_windows(df, time_col='timestamp', window='1D', alpha=0.05):
    df = df.copy()
    df[time_col] = pd.to_datetime(df[time_col])
    df = df.sort_values(time_col)
    df.set_index(time_col, inplace=True)
    windows = list(df.resample(window))
    drift_windows = []
    for i in range(len(windows)-1):
        w1 = windows[i][1]
        w2 = windows[i+1][1]
        # Only run KS test if both windows have at least 2 entries
        if len(w1['word_count']) < 2 or len(w2['word_count']) < 2:
            ks_stat, ks_p = np.nan, np.nan
        else:
            ks_stat, ks_p = ks_2samp(w1['word_count'], w2['word_count'])
        # Chi2 test for message type
        w1_counts = w1['message_type'].value_counts()
        w2_counts = w2['message_type'].value_counts()
        all_types = set(w1_counts.index).union(w2_counts.index)
        obs = np.array([
            [w1_counts.get(t, 0) for t in all_types],
            [w2_counts.get(t, 0) for t in all_types]
        ])
        obs = obs[:, ~(obs == 0).all(axis=0)]
        chi2_p = 1.0
        if obs.shape[1] > 0 and obs.sum() > 0:
            try:
                _, chi2_p, _, _ = chi2_contingency(obs)
            except Exception:
                pass
        # Only append if at least one test is significant and not NaN
        if ((ks_p is not np.nan and ks_p < alpha) or (chi2_p < alpha)):
            drift_windows.append((i, windows[i][0], windows[i+1][0], ks_stat, ks_p, chi2_
    return drift_windows
```

## Purpose

- Detects time windows where significant drift occurs in log data by combining the Kolmogorov-Smirnov (KS) test (for word count distribution) and the chi-squared test (for

message type distribution).
- Flags windows where either test indicates a statistically significant change.

## How it Works

- Converts the timestamp column to datetime, sorts, and sets it as the index.
- Resamples the DataFrame into consecutive time windows (default: 1 day).
- For each pair of consecutive windows:
    - Runs the KS test on word count distributions (if both windows have at least 2 entries).
    - Runs the chi-squared test on message type distributions.
    - If either test is significant (p < alpha), records the window pair and test statistics.
- Returns a list of window pairs with detected drift, including the window indices, timestamps, KS statistic, KS p-value, and chi2 p-value.

## Usage Example

```
drift_windows = detect_drift_windows(df)
print("Drift detected in the following window pairs:")
for i, t1, t2, ks_stat, ks_p, chi2_p in drift_windows:
    print(f"Window {i}: {t1} to {t2} | KS p={ks_p:.3g}, Chi2 p={chi2_p:.3g}")
```

## Interpretation

- Each reported window pair marks a period where a significant change (drift) was detected in either word count or message type distribution.
- Low p-values (KS or Chi2) indicate the type of drift:
    - **KS p < alpha:** Drift in word count distribution (structural/log content change).
    - **Chi2 p < alpha:** Drift in message type distribution (event type change).
- Useful for pinpointing when and where log behavior changes, supporting incident analysis and root cause investigation.

---

## Summary

- `detect_drift_windows` is a comprehensive drift detection tool that combines statistical tests for both continuous and categorical log features.
- Enables robust, interpretable detection of operational changes, incidents, or anomalies in log data over time.

---

# Code Notes: Visualizing Log Feature Trends with Drift Points

## Imports

- `import matplotlib.pyplot as plt`: For plotting time series and marking drift points.

## Workflow: Plotting Feature Trends and Drift Points

- **Aggregate Features:**
  - Resample the DataFrame by day (`'1D'`) and compute the mean word count and mean error rate for each day.
  - `windowed = df.resample('1D').agg({'word_count': 'mean', 'error_flag': 'mean'})`
- **Mark Drift Points:**
  - Extract the end date of each detected drift window from `drift_windows`.
  - `drift_days = [t2 for _, _, t2, _, _, _ in drift_windows]`
- **Plot:**
  - Plot the time series of mean word count and error rate.
  - Overlay vertical red dashed lines at each drift point to highlight detected changes.
  - Add legend, title, axis labels, and layout adjustments for clarity.

## Purpose

- Visualizes the temporal trends of key log features (mean word count and error rate) alongside detected drift points.
- Helps correlate statistical drift detection with observable changes in log behavior.
- Provides an intuitive summary for reports and presentations.

## Usage Example

```
plt.figure(figsize=(12, 6))
plt.plot(windowed.index, windowed['word_count'], label='Mean Word Count')
plt.plot(windowed.index, windowed['error_flag'], label='Error Rate')
for d in drift_days:
    plt.axvline(d, color='red', linestyle='--', alpha=0.5)
plt.legend()
plt.title('Log Feature Trends with Drift Points')
plt.xlabel('Date')
plt.ylabel('Value')
plt.tight_layout()
plt.show()
```

## Interpretation

- The plot shows how log entry structure (word count) and error rate evolve over time.
- Vertical red dashed lines indicate time points where significant drift was detected (by KS or chi-squared tests).
- Sudden changes or trends in the plotted features, especially near drift points, may correspond to operational events, incidents, or changes in system behavior.
- Useful for communicating findings and supporting root cause analysis.

---

## Summary

- This visualization integrates statistical drift detection with feature trend analysis, providing a clear, actionable overview of log data evolution over time.

- Supports both exploratory analysis and reporting in EDA workflows.