Log-Level Explainability Calculator

By

Syed Ausaf Haider

Under the guidance of

Associate Professor Haider Banka

(Note. This is a draft (to be reviewed) of the prototype highlighting the project and explaining the scripts and functionality of project directory and showcasing a working demo with related screenshots.)

The attached research proposal focuses on enhancing transparency and explainability in generative AI systems, such as GPT, DALL·E, and Stable Diffusion, by identifying current challenges and proposing innovative solutions and future directions. The central prototype idea is a log-level explainability calculator that captures and analyzes model behavior during content generation, transforming raw logs into human-readable explanations.

Core Research Focus

- The research identifies the "black-box" nature of generative models as a major problem, especially in sensitive domains like healthcare and finance, where unexplainable outputs pose ethical and practical risks.
- Traditional interpretability tools (like SHAP, LIME, Grad-CAM) are noted to be largely ineffective for generative models, which require deeper, domain-specific, and user-friendly explanations.

Research Objectives

- Critical Review: Analyze limitations of current interpretability methods for generative models.
- Tool Development: Design scalable, model-agnostic techniques (latent space visualization, feature attribution, counterfactual generation, new transparency metrics).
- Empirical Validation: Apply the new methods to state-of-the-art generative models and measure their effectiveness qualitatively (user studies) and quantitatively (metrics like precision, fairness, clarity).
- Ethical & Societal Impact: Evaluate how improved explainability can reduce risks (bias, hallucinations, misinformation) and support responsible AI deployment in sensitive fields.
- Future Directions: Focus on scalability, multimodal explainability, and human-in-the-loop interpretability for non-experts.

Prototype: Log-Level Explainability Calculator

- Goal: To produce detailed, human-interpretable explanations of generative model outputs by capturing model internals during inference.
- Logging Layer: During content generation, the system records intermediate states, including token probabilities, attention weights, latent activations, inputs, outputs, and timestamps.
- Processing Layer: The logs are analyzed to extract feature attribution (identifying which inputs most influenced the output), attention patterns, and latent space dynamics.
- Explanation Generator: Technical log data is transformed into narrative explanations, for example, quantifying how particular phrases steer the generation process.
- UI/Visualization (optional): Visual tools like tables, graphs, or heatmaps can help illustrate which parts of the input or internal states were most influential.

Summary Table: Prototype Flow

Layer	Function	Output Example
Logging	Capture model internals/steps	Token probs, attention, latents, timestamps
Processing	Analyze logs for attribution and patterns	"Phrase X increased medical term generation by 42%"
Explanation Generator	Convert logs to human-readable narrative	Clear explanation of model's decisions
(Optional) Visualization	Present influential tokens/features visually	Tables, graphs, heatmaps

Project Directory Structure

```
explainability_pkg/
│
├── LICENSE                      # Project/open source license
├── README.md                    # Project overview, install, usage instructions
├── CHANGELOG.md                 # Changelog for tracking version updates
├── CONTRIBUTING.md              # Contribution guidelines for collaborators
├── CODE_OF_CONDUCT.md           # Contributor code of conduct
├── pyproject.toml               # Modern Python build system config (metadata, deps)
├── setup.py                     # Traditional setup script (legacy/compat)
├── setup.cfg                    # Declarative build config (optional, complements setup.py)
├── requirements.txt             # List of required packages (pip install -r)
├── tox.ini                      # Test automation/config (for tox, pytest, lint)
├── .gitignore                   # Exclude build, virtualenv, logs from version control
├── .flake8                      # Python linting config
├── .pre-commit-config.yaml      # Pre-commit hooks setup for code quality
│
├── docs/                        # Project documentation
│   ├── conf.py                  # Sphinx documentation config
│   ├── index.rst                # Documentation entry point
│   ├── usage.md                 # Usage and installation guide
│   ├── api_reference.md         # API-level documentation (autogenerated/manual)
│   └── ...                      # More markdown or rst documentation files
│
├── examples/                    # Sample scripts and Jupyter notebooks
│   ├── run_all_metrics.py       # Script: run all metrics on a log file
│   ├── run_generation.py        # Script: generate text and output logs
│   ├── multi_model_userdefined_run_generation.py   # Interactive CLI user selection for model/config
│   ├── app_streamlit.py         # Streamlit web app UI for generation/logging
│   └── demo_hybrid_metric.ipynb     # (Optional) Notebook demo for the API
│
├── tests/                       # Automated tests for all package modules
```

```
│   ├── __init__.py
│   ├── test_main_logging_layer.py   # Tests for logging layer
│   ├── test_api.py                  # Tests for API module
│   ├── test_data_loader.py          # Tests for data/utils loader
│   └── ...                          # Add more tests as the package grows
│
├── logs/                            # Output logs directory (user-generated)
│   └── generation_log.json          # Example log file generated by logging layer
│
├── scripts/                         # Standalone automation or CLI scripts
│   ├── run_metrics.py               # Script to run metrics batch-wise (optional)
│   ├── generate_example_logs.py     # Script to generate example logs
│   └── ...                          # Other helper scripts
│
└── src/                             # Main source code, recommended src/ layout
    └── explainability_pkg/          # Package root (import explainability_pkg)
        ├── __init__.py              # Imports main API, classes, sets up namespace
        ├── main_logging_layer.py    # Core logging layer (captures/generates logs)
        ├── api.py                   # User-facing API (load logs, run metrics, etc.)
        ├── utils/                   # Utilities: loading, tensor ops, helpers
        │   ├── __init__.py
        │   ├── data_loader.py
        │   └── tensor_utils.py
        └── hybrid_metrics/          # Collection of independent hybrid metric modules
            ├── __init__.py
            ├── attention_weighted_influence.py
            ├── latent_sensitivity.py
            ├── context_consistency.py
            ├── fairness_weighted_influence.py
            ├── surprise_confidence.py
            └── ...                  # To add more hybrid metrics as needed
```

Phase 1. Logging Layer

The logging layer in our explainability package is the core component responsible for capturing, structuring, and storing detailed information about model inference runs. It acts as the foundational data collection mechanism that enables subsequent explainability metrics and analysis.

Purpose of the Logging Layer

- To capture all relevant data during model inference, including inputs, intermediate states, outputs, and metadata like timestamps.
- To structure this information systematically in a standard format (e.g., JSON logs) that can be easily stored, shared, and processed.
- To serve as a data foundation for explainability metrics, which analyze the logged information to compute interpretability scores or explanations.
- To enable reproducibility and auditability of AI model runs by preserving comprehensive execution details.

Key Functionalities

- Input capture: Logs raw inputs fed into models (e.g., text prompts, images).
- Output capture: Logs model outputs such as generated tokens, predictions, or feature weights.
- Intermediate state capture: Records internal states like attention weights, latent activations, or gradients.
- Timestamping: Captures timing information for each step to analyze model latency or performance.
- JSON serialization: Converts all logged data into a well-structured JSON format for easy consumption.
- File output: Writes logs to persistent files (e.g., generation_log.json) for later retrieval.

Role in the Package Ecosystem

- Acts as the underlying data producer for the hybrid metrics in the hybrid_metrics module.
- Provides a standardized input to API methods enabling metric computation, analysis, and visualization.
- Supports streaming or batch logging, allowing flexibility in different usage scenarios like real-time monitoring or offline evaluation.

Architectural Placement

The logging layer is implemented in the main_logging_layer.py file within the package root (src/explainability_pkg/). It is designed to be modular and extensible so that new logging features or data types can be added with minimal disruption.

Metrics generated by logging layer

The logging layer generates a comprehensive set of metrics capturing detailed internal states of the generative AI model during inference. The key metrics produced per generation step include:

Metric Name	Description
Step	Generation step index (starting from 0), indicating the token position in the output sequence.
Timestamp	Unix timestamp recording the exact time when the step was executed, useful for latency analysis.
input_data	The initial input prompt or context provided to the model at the start of generation.
output_token	The token generated by the model at this specific inference step, decoded as text.
token_probabilities	The softmax probability distribution over the entire vocabulary for the predicted next token.
attention_weights	Multi-dimensional arrays (layers × heads × tokens × tokens) representing how the model attends to input tokens.
latent_vectors	Numerical vectors representing the internal activations (hidden states) of the model's layers at this step.
influence_scores	Aggregate scores derived from attention or other methods that quantify how much each input token influenced the generated token.
generation_config	The model generation parameters such as temperature, top-k, top-p, and random seed used for this run.
context_window	The decoded text representing the current context — all tokens processed/generated up to this step.

These metrics collectively capture probabilistic, attentional, latent, temporal, and configurational aspects of model inference, providing a rich foundation for downstream explainability computations and human-interpretable narratives.

Summary of the Logging Layer

- The logging layer tracks every token generated, noting not only the output but also the probabilities and attention patterns associated with it.
- It records internal states like attention weights and latent activations, which reveal which parts of the input influenced the model's decisions most strongly.
- It captures timestamps and generation parameters such as temperature or top-k settings for reproducibility and performance analysis.
- All this information is structured into detailed JSON logs, enabling further use by explainability metrics and audit tools.
- By preserving this data, the logging layer provides a transparent, step-by-step view of the model's behavior, which is critical for understanding, debugging, and trusting generative AI outputs.
- This approach mirrors modern best practices in AI system monitoring—capturing rich contextual data to enable continuous improvement, issue detection, and user-friendly explanations.

In essence, the logging layer is the "memory" of the generation process, storing granular insights that empower downstream explainability and operational monitoring, making complex AI systems more interpretable and accountable.

Phase 2. Processing Layer (Produces Hybrid metric)

The processing layer is the next stage after the logging layer in an explainability system for generative AI. Its core function is to take the raw, detailed logs produced by the logging layer and extract meaningful insights, metrics, and summaries that reveal *how* the model made its decisions.

- Parse Raw Logs: It ingests the structured JSON logs containing token probabilities, attention weights, latent activations, timestamps, and other metrics recorded at each generation step.
- Compute Feature Attribution: Using methods such as attention analysis, gradient-based techniques, or other algorithms, it quantifies the influence that specific input tokens or features had on the predicted outputs. These quantified values are often called *attribution scores* or *influence metrics*.
- Calculate Composite Metrics: It combines multiple raw signals—such as weighting attention scores by token probabilities—to produce *hybrid* or *composite* metrics that better capture the model's internal decision dynamics than any single raw measure.
- Aggregate Over Sequences: It may compute cumulative or temporal metrics analyzing how attention or influence evolves over multiple generation steps, providing broader context and detecting consistency or drift.
- Filter and Normalize Data: To ensure meaningful comparisons, the processing layer normalizes raw activations and probabilities, removes noise, and handles missing or sparse data gracefully.
- Prepare Data for Explanation and Visualization: The computed attributions and metrics form the basis for generating human-readable explanations and driving visualizations like heatmaps, charts, or interactive graphs.

Why the Processing Layer is Important

The raw outputs from a model—probabilities, activations, attentions—are complex and high-dimensional, making direct interpretation difficult. The processing layer distills these raw traces into concise, informative metrics that can be converted into explanations understandable to humans.

This layer bridges the gap between model internals and user-facing interpretability by:

- Highlighting which input elements mattered most
- Identifying influential patterns across the generation
- Quantifying the reliability and relevance of intermediate signals

In essence, the processing layer transforms a *detailed whereabouts log* into a *storyline* of the model's reasoning, enabling downstream modules to explain the AI's outputs clearly and fairly.

This concept aligns with best practices in explainable AI and is crucial for building trustworthy generative systems as detailed in the prototype and literature.

Working of processing layer

The processing layer operates as the analytical engine that transforms raw logs collected by the logging layer into meaningful explanations and insights about the generative AI model's behavior. Here's how it works step-by-step:

Step 1: Input - Parse Raw Logs

- The processing layer starts by loading the detailed JSON logs generated during model inference.
- Each log entry represents one generation step and contains metrics such as token probabilities, attention weights, latent activations, timestamps, and input-output mappings.

Step 2: Extract Relevant Metrics

- From each step, the layer extracts key data like:
    - Attention matrices (indicating how much the model "looked" at each input token)
    - Token probability distributions
    - Latent vector activations (deep internal representations)
- It arranges and aligns these metrics for consistent analysis across generation steps.

Step 3: Calculate Feature Attribution Scores

- Using attention weights and optionally gradient-based or perturbation methods, the layer computes scores that estimate how much each input token influenced the generated token at each step.
- These scores quantify "importance" or "influence," often normalizing values across the input to highlight the most impactful elements.

Step 4: Compute Hybrid/Composite Metrics

- The system integrates multiple sources of information to derive richer, more nuanced metrics.
- For example, it can weight attention-based attributions by token probabilities to estimate the net effect on the generation confidence.
- It may also analyze temporal consistency by comparing attributions across multiple steps to detect stable or shifting focus.

Step 5: Filter and Normalize Data

- The processing layer cleans the data by:
    - Handling missing or incomplete values robustly.
    - Normalizing scores to comparable scales.
    - Removing noise or outliers that may distort interpretations.

Step 6: Aggregate Metrics over the Generation Sequence

- It summarizes metrics at various levels:
    - Per-token importance profiles
    - Cumulative influence over the entire generated text

- Sensitivity or stability of latent activations through the sequence

Step 7: Prepare Outputs for Explanation and Visualization

- The processed metrics are formatted into structured summaries or dataframes.
- These can be fed into:
    - Natural language explanation generators which craft human-readable narratives explaining model reasoning.
    - Visualization modules that create attention heatmaps, influence charts, or stepwise trace diagrams.

Complete Summary of the Processing Layer:

1. Input Handling:
   It ingests structured JSON logs produced by the Logging Layer, which include stepwise data such as token probabilities, attention weights, latent activations, timestamps, and generation parameters.
2. Feature Extraction:
   The layer parses these logs to extract relevant signals at each generation step — notably the attention matrices and token probability distributions that reflect where the model focuses and how confident it is.
3. Attribution Computation:
   Using techniques like aggregated attention analysis and optionally gradient- or perturbation-based methods, it quantifies the influence or importance of each input token on the generated output at each step. This quantification is often called the attribution or influence score.
4. Hybrid and Composite Metrics:
   The layer integrates multiple metrics—such as combining attention weights with token probabilities—to produce richer, hybrid indicators that better capture the nuances of model decisions.
5. Temporal Analysis:
   By examining the evolution of attributions and latent activations across the sequence, the Processing Layer assesses consistency, shifts in focus, and stability of the model's reasoning over time.
6. Data Cleaning and Normalization:
   It applies normalization and filtering to reduce noise and ensure that metrics are comparable and robust. Missing or sparse data are handled gracefully.
7. Output Preparation:
   The computed metrics and summaries are organized and formatted into structures suitable for further steps: generating human-readable explanations and creating visualizations such as heatmaps or influence charts.
8. Role in Explainability:
   Acting as a bridge between raw model internals and end-user interpretable information, the Processing Layer enables transparent inspection, audit, fairness analysis, and debugging of generative AI outputs.

Why It Matters:

The Processing Layer condenses complex, high-dimensional internal model data into accessible, actionable insights that can be communicated to stakeholders across levels of expertise. It transforms opaque neural computations into understandable explanations, fostering trust and accountability in AI systems.

Reference:

This conceptual and functional framework aligns with the research and implementation roadmap detailed in the attached documents, which position the Processing Layer as an indispensable part of scalable, modular, and trustworthy generative AI explainability systems.

Phase 3. Explanation Generator

The Explanation Generator is a critical component in the explainability pipeline that translates complex, multi-dimensional data from the logging and processing layers into clear, human-understandable narratives and insights about the generative AI's behavior.

Purpose & Role

- Converts technical metrics such as token probabilities, attention weights, and feature attributions into natural language explanations.
- Helps end-users—including non-technical stakeholders—comprehend why and how the AI model generated particular outputs.
- Bridges the "language" gap between low-level numeric data and high-level intuitive understanding.

Key Functions

1. Interpretation of Metrics:
   Transforms quantitative signals (e.g., "token X increased likelihood by 42%") into concise descriptive statements.
2. Narrative Construction:
   Crafts coherent narratives that reflect the model's stepwise reasoning, highlighting key influencers, contrasts, and causal links within the generation.
3. Customization for Audience:
   Adapts explanation complexity and terminology to suit different users—from AI practitioners requiring detailed technical insights to general users seeking simple summaries.
4. Template-Based and/or Automated Generation:
   Utilizes:
   - Rule-based templates mapping metric patterns to language constructs.
   - Lightweight Natural Language Generation (NLG) frameworks to automate and diversify explanations.
5. Integration with UI/Visualization:
   Coordinates with visualization layers to provide textual counterparts complementing heatmaps, charts, and interactive tools.

Why It is Important

- Empowers trust and transparency by revealing model decisions in an accessible way.
- Facilitates error analysis and debugging by pinpointing influential inputs and uncertain reasoning.
- Supports ethical AI deployment by making biases and potential failure modes explicit.
- Enhances user engagement and confidence through tailored, insightful communication.

The Explanation Generator is the culminating stage in the explainability framework, responsible for turning the rich, complex data produced by the logging and processing layers into clear, insightful, and user-friendly interpretations of the generative AI model's behavior. Here's a detailed account of its complete workings:

Complete Workings of the Explanation Generator

1. Input Acquisition:
   The generator receives processed metrics and attribution scores from the Processing Layer. These inputs include:
   - Token-level influence scores
   - Attention patterns and their aggregate effects
   - Latent activation insights
   - Contextual summaries (e.g., how the model's focus shifts over time)
   - User-configurable parameters or domain-specific knowledge
2. Interpretation & Mapping:
   It interprets numeric metrics into conceptual meanings by:
   - Identifying key tokens or inputs with high influence on outputs
   - Recognizing patterns such as reinforcement or suppression of probabilities
   - Comparing current outputs with alternative or counterfactual cases if available
   - Mapping raw numeric changes (e.g., a token increasing likelihood by 40%) to verbal expressions
3. Template-Based Narration:
   Utilizes predefined templates or rule-based systems to convert insights into coherent language. Examples include:
   - "The word 'urgent' increased the probability of medical terms by X%."
   - "Attention focused mostly on tokens related to 'hospital' indicating strong domain influence."
   - "This token choice appears less likely given the previous context."
     Templates balance specificity and generality for clarity.
4. Natural Language Generation (NLG):
   - Optionally employs lightweight NLG frameworks to automate narrative generation, providing variation and natural flow.
   - Handles complex constructs such as cause-effect chains, contrasts, or hypothetical explanations.
   - Enables multi-level explanations—from simplified summaries to detailed technical reports.
5. User Adaptation:
   - Adjusts the level of detail and complexity according to the end-user:
     - Technical users get info-rich, data-driven narratives.
     - Lay users receive simplified, analogy-based explanations.
   - Supports configurable verbosity and focus (e.g., highlight biases, uncertainties).

6. Integration with Visualization:
   Coordinates with visual components to create complementary explanations:
   - Textual descriptions accompanying heatmaps or charts
   - Interactive explanations that respond to user inputs or selections
   - Augments graphical views with narrative context for enhanced understanding
7. Output Formatting:
   Packages explanations alongside raw or processed metrics in structured formats:
   - JSON with text fields and references to data points
   - Rich text or HTML for web UIs
   - Reports or dashboards customized for stakeholder needs
8. Feedback & Iteration Loop:
   - Incorporates user feedback or domain expert inputs to refine explanation templates and logic.
   - Uses evaluation metrics (e.g., clarity scores, user trust assessments) to improve quality.

The Explanation Generator is the final, interpretive component in the explainability pipeline that converts detailed, numerical insights from internal model processes into clear, accessible, and user-friendly narratives. Its core purpose is to make the AI model's decision-making transparent and understandable for diverse audiences.

Complete Summary of Explanation Generator

- Input Processing:
  Takes processed metrics and attribution results from the processing layer, including token-level influence scores, attention patterns, latent activation information, and contextual data about the generation process.

- Interpretation and Conceptual Mapping:
  Translates raw numerical metrics into meaningful conceptual insights by identifying significant input tokens, analyzing attention focus, and quantifying the impact of various features on the final output.

- Narrative Construction:
  Employs template-based or rule-driven methods combined optionally with lightweight natural language generation (NLG) to convert technical signals into coherent, readable sentences and explanations. This includes cause-effect relations, key influencers, and comparative statements.

- User-Centric Adaptation:
  Customizes explanations based on the target user's expertise and needs—providing detailed technical reports for AI developers or concise summaries and analogies for general users.

- Integration with Visualization:
  Works alongside visual tools such as heatmaps or influence charts, providing textual descriptions that enhance comprehension and user experience.

- Output Formatting and Delivery:
  Outputs explanations in structured formats suitable for diverse interfaces—ranging from JSON for APIs and dashboards to rich text or HTML for interactive web applications.

- Feedback and Improvement Loop:
  Incorporates user feedback and evaluation to continually refine the quality, clarity, and relevance of generated explanations.

Importance

By bridging complex internal model data and human communication, the Explanation Generator fosters trust, transparency, and accountability in generative AI systems. It empowers users to understand why specific outputs were produced, supports debugging and bias detection, and facilitates responsible AI deployment.

Phase 4. UI/Visualization (to be done)

UI/Visualization (optional): Visual tools like tables, graphs, or heatmaps can help illustrate which parts of the input or internal states were most influential.

Plese go to next page as I have not implement this phase currently.

Inventions (Hybrid metrics)

Based on the logs produced by logging layer, we try to achieve and build new metrics. Currently I have been also to figure out only 2 new Hybrid metrics that are fully working showcasing theory and working implementations as well.
Two new working and implementational hybrid metrics are:

1. Attention-Weighted Token Probability Influence
2. Latent Activation Sensitivity Score

These metrics will be explained in later pages.

The proposed hybrid metrics in this generative AI explainability framework are advanced, composite interpretability measures computed by processing the detailed raw logs produced during model inference. These metrics combine multiple internal signals such as attention weights, token probabilities, and latent state activations to create richer, more nuanced scores that better capture the AI model's decision dynamics than any single raw metric alone.

What Are Hybrid Metrics?

- They are post-hoc computed metrics that analyze the *raw logging data* (token probabilities, attention patterns, latent vectors) collected step-by-step during generation.
- Hybrid metrics integrate multiple sources of information, such as combining attention weights with token probability confidence, or analyzing sensitivity in latent activations relative to output changes.
- Designed to provide interpretable, quantitative scores that highlight influential inputs, model confidence, stability, bias, fairness, and uncertainty aspects.
- They serve as the core analytical tools enabling transparency into the otherwise opaque generative process.

Why Are Hybrid Metrics Important?

- They provide a multidimensional interpretability lens into generative AI decisions, combining complementary signals.
- Help to pinpoint which input features or tokens drove model outputs, how confident the model was, and how robust or biased its internal reasoning is.
- Enable auditing, debugging, bias detection, and trust-building in critical applications like healthcare and finance.
- Are modular and extensible, allowing future addition of new metrics without disrupting the architecture.

How They Fit into the Prototype

- These hybrid metrics are computed in the Processing Layer, consuming JSON logs generated by the Logging Layer.
- Each metric is implemented as an independent, reusable module within the hybrid_metrics/ package.
- The package API allows running one or more registered hybrid metrics on logs seamlessly.
- Metrics may feed into the Explanation Generator and visualization layers for producing human-readable insights and interactive dashboards.

Attention-Weighted Token Probability Influence (1st Hybrid metric Invented)

Explanation of Attention-Weighted Token Probability Influence

What is it?

The Attention-Weighted Token Probability Influence is a hybrid metric that combines two raw metrics from the logging layer:

1. Token Probabilities: These are the probabilities assigned by the model to each possible token in its vocabulary for the next step of generation. They indicate the model's confidence in predicting a specific token.
2. Attention Weights: These represent how much "focus" or importance the model assigns to each input token (or previous tokens) during the generation of the current token.

By combining these two metrics, this hybrid metric quantifies the weighted impact of input tokens on the confidence (probability) of the generated token. Essentially, it tells us which input tokens not only influenced the output but also contributed to the model's certainty about that output.

Why is it important?

This metric helps identify which parts of the input (tokens) had the most significant influence on the model's decision-making process. It provides deeper insights into:

- Model Confidence: Understanding why the model is confident about certain outputs.
- Explainability: Highlighting specific tokens or phrases that guided the model’s reasoning.
- Error Detection: Identifying potentially problematic inputs that overly influenced the model, leading to incorrect or biased outputs.

For example, if the model generates a medical term like "diagnosis" with high confidence, this metric can reveal whether the phrase "in the hospital" or "fever" was the primary driver of that decision.

Explanation of Attention-Weighted Token Probability Influence

What is it?

The Attention-Weighted Token Probability Influence is a hybrid metric that combines two raw metrics from the logging layer:

1. Token Probabilities: These are the probabilities assigned by the model to each possible token in its vocabulary for the next step of generation. They indicate the model's confidence in predicting a specific token.
2. Attention Weights: These represent how much "focus" or importance the model assigns to each input token (or previous tokens) during the generation of the current token.

By combining these two metrics, this hybrid metric quantifies the weighted impact of input tokens on the confidence (probability) of the generated token. Essentially, it tells us which input tokens not only influenced the output but also contributed to the model's certainty about that output.

Why is it important?

This metric helps identify which parts of the input (tokens) had the most significant influence on the model's decision-making process. It provides deeper insights into:

- Model Confidence: Understanding why the model is confident about certain outputs.
- Explainability: Highlighting specific tokens or phrases that guided the model’s reasoning.

- Error Detection: Identifying potentially problematic inputs that overly influenced the model, leading to incorrect or biased outputs.

For example, if the model generates a medical term like "diagnosis" with high confidence, this metric can reveal whether the phrase "in the hospital" or "fever" was the primary driver of that decision.

How does it work?

The formula for this metric is:

$$\text{Attention-Weighted Influence} = \sum_i (\text{Attention Weight}_i \times \text{Token Probability}_i)$$

Here's a breakdown of the formula:

1. Attention Weight (Attention Weight$_i$): This measures how much focus the model places on the i -th input token when generating the current output token. Higher attention weights mean the model pays more attention to that token.
2. Token Probability (Token Probability$_i$): This is the probability assigned by the model to the i -th token in the vocabulary. Higher probabilities indicate greater confidence in that token being the correct choice.
3. Weighted Sum: By multiplying the attention weight and token probability for each token and summing them up, we compute the overall influence of all input tokens on the generated token.

Use Case: Identifying Influential Input Tokens

This metric is particularly useful for answering questions like:

- Which input tokens most strongly influenced the model's confidence in generating the output token?
    - For example, in a sentence completion task, if the model generates "hospital," this metric can show that the phrase "medical emergency" had a higher weighted influence than other parts of the input.
- Are there any tokens that disproportionately influenced the output?
    - If certain tokens have an unusually high influence, it could indicate bias, overfitting, or reliance on spurious correlations.
- How does the model's focus shift over time?
    - By analyzing this metric across multiple steps, you can track how the model shifts its attention as the context evolves.

Example Walkthrough

Let's say the model is generating the word "admitted" in response to the input prompt:
"The patient was brought to the hospital due to severe symptoms."

Step 1: Capture Raw Metrics

- Token Probabilities: The model assigns probabilities to all possible tokens for the next step. For simplicity, let's assume the top three probabilities are:
    - "admitted": 0.7
    - "discharged": 0.2
    - "treated": 0.1
- Attention Weights: The model assigns attention weights to each token in the input prompt. For example:
    - "patient": 0.1
    - "hospital": 0.5
    - "symptoms": 0.4

Step 2: Compute Weighted Influence

Using the formula:

$$\text{Attention-Weighted Influence} = \sum_i (\text{Attention Weight}_i \times \text{Token Probability}_i)$$

For the token "admitted":

$$\text{Influence} = (0.1 \times 0.7) + (0.5 \times 0.7) + (0.4 \times 0.7)$$

$$\text{Influence} = 0.07 + 0.35 + 0.28 = 0.7$$

Step 3: Interpret Results

- The total influence score of 0.7 indicates that the input tokens collectively had a strong impact on the model's confidence in generating "admitted."
- Breaking it down further:
    - "hospital" had the highest influence (0.35), suggesting it was the most critical factor in guiding the model toward "admitted."
    - "symptoms" also played a significant role (0.28).
    - "patient" had a smaller but still meaningful contribution (0.07).

Benefits of Using This Metric

1. Improved Explainability: Provides a clear explanation of how the model arrived at its decision by highlighting influential input tokens.
2. Bias Detection: Helps identify whether the model is relying too heavily on certain tokens, which could indicate bias or unfairness.
3. Debugging: Useful for diagnosing cases where the model generates unexpected or incorrect outputs by pinpointing the source of influence.
4. Trust and Transparency: Builds trust with users by making the model's reasoning process more transparent and interpretable.

Explanation of the Code

1. Input (logs):
    - The function takes a list of log entries (logs), where each entry is a dictionary containing raw metrics like token_probabilities and attention_weights.
2. Processing Each Log Entry:
    - For each step, the function extracts token_probabilities and attention_weights.
    - If these fields are missing or empty, the function appends None to the result list.
3. Dimensionality Handling:
    - Attention weights often come as multi-dimensional arrays (e.g., [layers, heads, seq_len, seq_len]).
    - To simplify computation, the function sums over the layers and heads (axes 0 and 1) to reduce the dimensionality.
4. Weighted Influence Calculation:
    - The formula used is:

$$\text{Attention-Weighted Influence} = \sum(\text{Attention Weights} \times \text{Token Probabilities})$$

    - This computes the weighted impact of input tokens on the confidence (probability) of the generated token.
5. Output:
    - The function returns a list of attention-weighted influence scores, one for each generation step.

Key Notes

1. Scalability:
    - This implementation can handle large vocabularies and multi-dimensional attention weights by reducing dimensions appropriately.
2. Error Handling:
    - The function gracefully handles missing or incomplete data by appending None to the results.
3. Extensibility:
    - You can extend this function to include additional computations, such as normalizing the influence scores or aggregating them across steps.

This code provides a clear and simple implementation of the Attention-Weighted Token Probability Influence metric, aligning with the goals of explainability and transparency in generative AI systems.

Latent Activation Sensitivity Score (2nd Hybrid metric Invented)

Explanation of Latent Activation Sensitivity Score

What is it?

The Latent Activation Sensitivity Score is a hybrid metric that combines two raw metrics:

1. Latent Vectors: These are the internal hidden state activations or representations learned by the model during inference. They capture the transient state of the model as it processes inputs and generates outputs.
2. Token Probabilities: These represent the probabilities assigned by the model to each possible token in the vocabulary for the next step of generation.

This metric quantifies how sensitive the latent activations (hidden states) are to changes in token probabilities. In other words, it measures how much small perturbations in the latent space affect the model's confidence in predicting the next token. This helps identify regions of the latent space that are overly sensitive or unstable, which could indicate issues like hallucinations, bias, or poor generalization.

Why is it important?

Understanding the sensitivity of latent activations is crucial for several reasons:

1. Model Robustness: If the latent space is highly sensitive, even minor changes in the input or latent state could lead to drastically different outputs, signaling brittleness in the model's reasoning.
2. Error Detection: Sensitive regions of the latent space may correspond to areas where the model struggles to generalize, leading to incorrect or unstable outputs.
3. Explainability: By identifying which parts of the latent space have the most influence on token probabilities, this metric provides deeper insights into the model's decision-making process.
4. Bias and Fairness: Overly sensitive regions might disproportionately amplify biases present in the training data, making this metric useful for fairness analysis.

For example, if the model generates "urgent care" with high confidence, this metric can reveal whether the latent activations corresponding to medical concepts are overly sensitive to small variations in the input.

How does it work?

The formula for this metric is:

Latent Sensitivity=∂Latent Vector / ∂Token Probability

Here’s a breakdown of the formula:

1. Token Probability (Token Probability): The probability assigned by the model to a specific token in the vocabulary.
2. Latent Vector (Latent Vector): The internal hidden state representation at a particular step in the generation process.
3. Partial Derivative (∂∂): The partial derivative measures how much the token probability changes with respect to small changes in the latent vector. A higher value indicates greater sensitivity.

In practice, this involves:

- Perturbing the latent vector slightly (e.g., adding small noise).
- Observing how the token probabilities change as a result.

- Calculating the gradient or rate of change between the two.

Use Case: Identifying Unstable or Overly Sensitive Regions

This metric is particularly useful for answering questions like:

- Are there regions of the latent space that are overly sensitive?
    - For example, if slight changes in the latent vector cause large swings in token probabilities, this could indicate instability.
- Which latent dimensions have the most influence on token probabilities?
    - This helps pinpoint specific dimensions or features in the latent space that drive the model's decisions.
- Is the model relying too heavily on certain hidden representations?
    - Over-reliance on specific latent dimensions could make the model brittle or prone to errors.

Example Walkthrough

Let’s say the model is generating the word "admitted" in response to the input prompt:
"The patient was brought to the hospital due to severe symptoms."

1. Step 1: Capture Raw Metrics
    - Token Probabilities: The model assigns probabilities to all possible tokens for the next step. For simplicity, let’s assume the top three probabilities are:
        - "admitted": 0.7
        - "discharged": 0.2
        - "treated": 0.1
    - Latent Vectors: The latent vector for the current step is:
        - [0.1, 0.3, 0.5]
2. Step 2: Perturb the Latent Vector
    - Add small noise to the latent vector:
        - Original latent vector: [0.1, 0.3, 0.5]
        - Perturbed latent vector: [0.11, 0.31, 0.51]
3. Step 3: Observe Changes in Token Probabilities
    - After perturbing the latent vector, the new token probabilities become:
        - "admitted": 0.65
        - "discharged": 0.25
        - "treated": 0.1

4. Step 4: Compute Sensitivity
    - Calculate the partial derivative for each token:
        - For "admitted":

$$\text{Sensitivity} = \frac{\Delta \text{Latent Vector}}{\Delta \text{Token Probability}} = \frac{0.01}{0.65 - 0.7} = -5.0$$

        - Repeat for other tokens.
5. Step 5: Interpret Results
    - A sensitivity score of -5.0 for "admitted" indicates that small changes in the latent vector significantly reduce the model's confidence in this token. This suggests the latent space is sensitive to this region.

Benefits of Using This Metric

1. Improved Robustness: Helps identify and mitigate unstable regions of the latent space, improving the model's reliability.
2. Error Analysis: Useful for diagnosing cases where the model generates unexpected or incorrect outputs.
3. Fairness and Bias Detection: Highlights latent dimensions that disproportionately influence token probabilities, potentially amplifying biases.
4. Trust and Transparency: Builds trust with users by providing insights into the stability and reliability of the model's internal representations.

Conclusion

The Latent Activation Sensitivity Score is a powerful tool for understanding the relationship between latent activations and token probabilities in generative AI models. By quantifying how sensitive the latent space is to changes in token probabilities, it reveals unstable or overly sensitive regions that could impact the model's performance. This insight is invaluable for improving robustness, detecting errors, and ensuring fairness in generative AI systems.

Now showing the implementation demo step by step along with required screenshots:

Log-Level Explainability Calculator
(I have put the workflow of building this project in the attachment called workflow.docx)

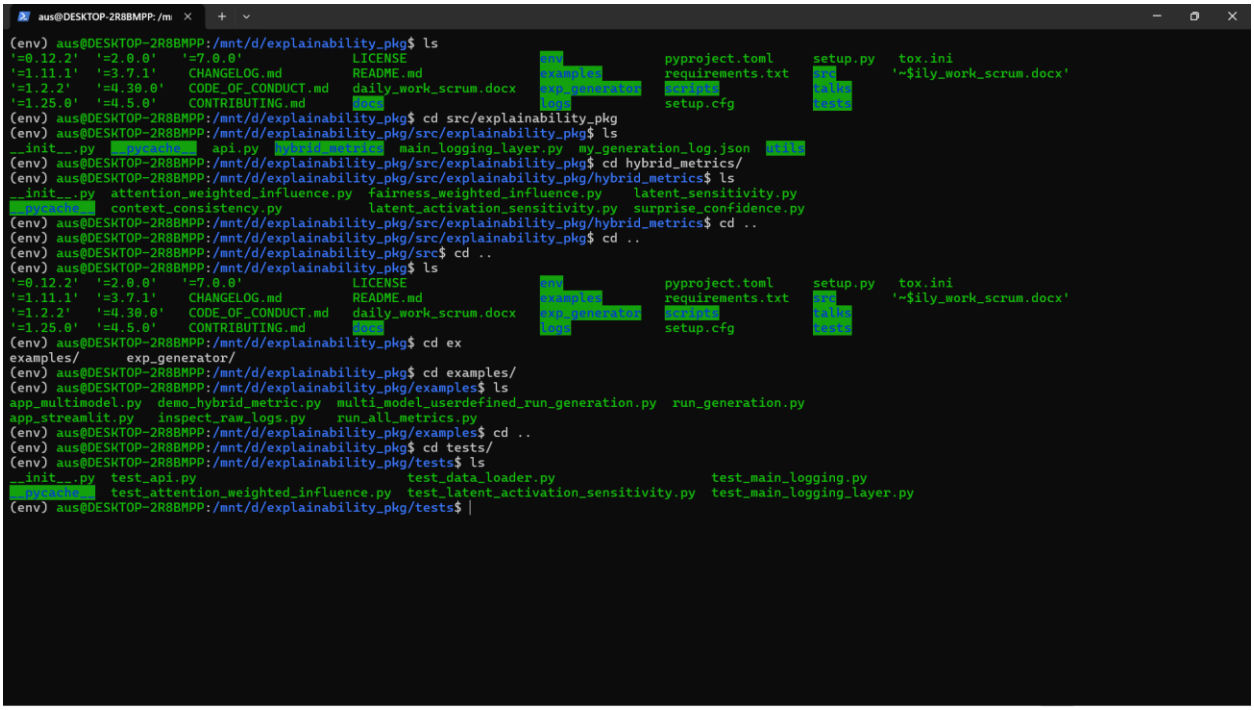
Starting by loading the Python environment env

>> python3 -m venv env

>> source env/bin/activate

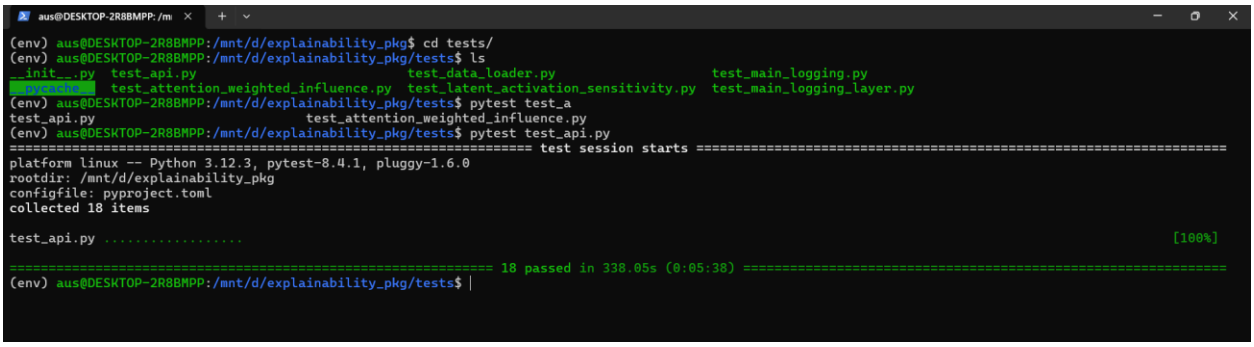
```
PS D:\explainability_pkg> wsl
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Fri Sep  5 15:10:43 UTC 2025

  System load:  2.71                Processes:             77
  Usage of /:   7.1% of 1006.85GB   Users logged in:       0
  Memory usage: 3%                  IPv4 address for eth0: 172.22.204.190
  Swap usage:   0%

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

This message is shown once a day. To disable it please create the
/home/aus/.hushlogin file.
aus@DESKTOP-2R8BMPP:/mnt/d/explainability_pkg$ source env/bin/activate
(env) aus@DESKTOP-2R8BMPP:/mnt/d/explainability_pkg$
```

Showcasing different scripts as per project directory:



Starting testing (It includes stress testing and all unit tests required)

Testing api.py



The test_api.py file in your project is designed to automatically verify the correctness, robustness, and expected behavior of the API layer that provides core functionalities such as loading logs and running hybrid metrics on generative AI model outputs.

Purpose of test_api.py

- To ensure that the main API functions (load_logs, run_metric, run_all_metrics) work as intended.
- To validate correct handling of normal inputs as well as edge cases such as missing files or invalid data.
- To catch errors early in development and integration, maintaining the quality and reliability of the explainability package.
- To facilitate regression testing as new features and metrics are added.

Typical Tests Covered

1. Loading Logs
    - Tests reading valid JSON log files and confirming the correct number and format of entries.
    - Handles invalid or missing log file cases gracefully, confirming the function returns None or raises appropriate errors without crashing.
2. Running Single Metrics (run_metric)
    - Confirms that named registered metrics compute successfully given valid logs.
    - Checks for error handling when an unregistered or misspelled metric name is used.
    - Validates metric output structure and types.
3. Running All Metrics (run_all_metrics)
    - Tests the ability to compute all registered metrics on given logs.
    - Confirms that results include expected keys, and that failures in one metric do not halt the entire batch.
4. Error Handling & Input Validation
    - Tests robustness against malformed log inputs (wrong types, empty data, missing keys).
    - Ensures the API functions log errors and return safe error states.

Why This Matters

- The API layer is the central user interface to the package's explainability functionality.
- Reliable API methods mean users can trust the package to correctly load their data and produce explainability metrics without manual debugging.
- Automated tests enable confidence when extending the package, like adding more hybrid metrics or integrating new explainability features.

Summary

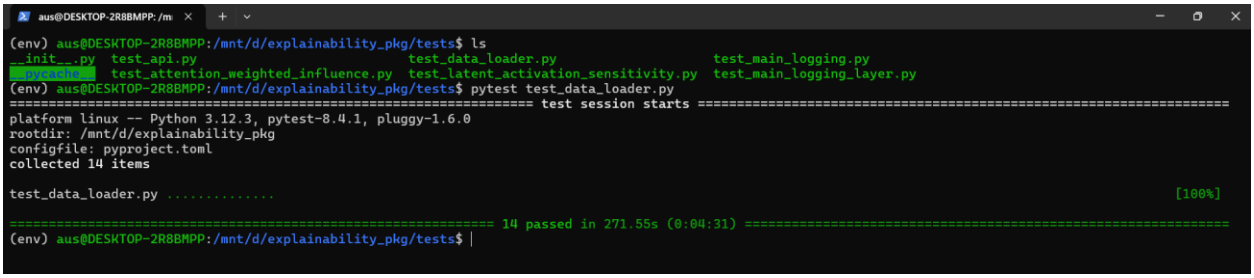
test_api.py is a comprehensive suite of automated tests that exercise the main user-facing API functions of your explainability package. It verifies correct function, input validation, error handling, and output integrity for loading logs and running generative model explainability metrics, underpinning the overall project reliability and maintainability.

explain test_data_loader.py

explain test_data_loader.py

The test_data_loader.py file in your project is focused on testing the data loading utilities and functions designed to read, parse, and validate the raw generative AI logs produced by the logging layer.

test_data_loader.py



Purpose of test_data_loader.py

- To ensure that the data loader modules correctly handle the input data format (typically JSON logs) without errors.
- To validate the integrity and structure of loaded data, making sure required fields are present and formatted as expected.
- To test robustness against corrupt, incomplete, or malformed log files, verifying graceful failure or recovery.
- To confirm consistent parsing behavior across different versions or variants of log files, ensuring compatibility and stability.

Typical Tests Covered

1. Successful Loading
    - Verify that well-formed JSON log files are read and parsed into Python data structures correctly.
    - Check that each log entry contains all required keys for downstream processing (e.g., step, token_probabilities, attention_weights).
2. Data Validation
    - Test that the loader checks for missing or null required fields.
    - Confirm that invalid data types or unexpected structures raise warnings or errors as designed.
3. File Handling Robustness
    - Simulate scenarios such as missing files, empty files, or invalid JSON syntax.
    - Test the data loader's behavior when encountering these to ensure proper exceptions or return values.
4. Edge Case Handling
    - Handle logs with unusual but valid conditions (e.g., zero-length attention vectors, empty token lists).
    - Ensure the loader does not crash or produce incorrect outputs.

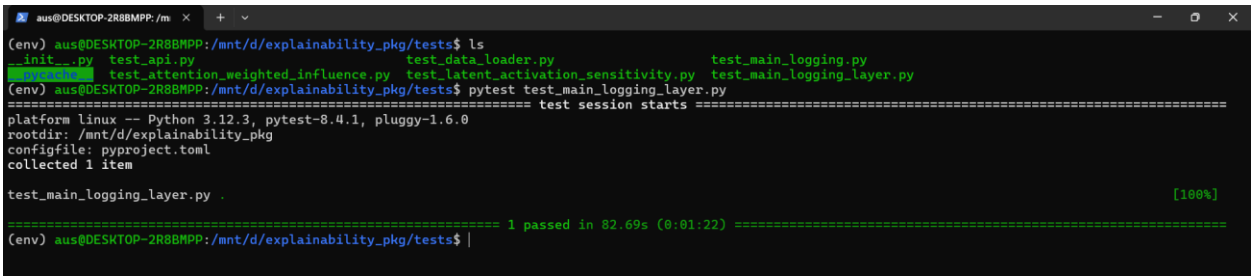
Why This Matters

- The data loader is the gateway converting raw log files into usable data for metrics and explanation generation.
- Any errors or inconsistencies here ripple through the whole explainability pipeline.
- Automated tests here ensure the package will reliably ingest logs from various sources and formats, a critical foundation for correctness and user trust.

Summary

test_data_loader.py provides essential automated validation of the log loading and parsing logic in your explainability package. It verifies that the data loader functions correctly interpret raw JSON logs under normal and exceptional conditions, maintaining data integrity and stability for the downstream process of explaining generative AI behavior.

test_main_logging_layer.py



The test_main_logging_layer.py file is dedicated to testing the core logging layer module of your explainability package, which is responsible for capturing and recording detailed internal metrics during each step of generative AI model inference.

Purpose of test_main_logging_layer.py

- To verify that the logging layer correctly captures key data such as generated tokens, token probabilities, attention weights, latent activations, timestamps, and other metadata.
- To ensure logs produced are structured and formatted properly—usually in standardized JSON suitable for downstream processing.
- To test the integration and compatibility of the logging layer with different generative models or input scenarios.
- To check error handling and robustness, ensuring the logging layer behaves gracefully if certain data is missing or malformed during capture.
- To confirm that logging performance and stepwise traceability meet design requirements.

Typical Tests Covered

1. Basic Log Capture
    - Confirm that generating a sample output triggers the logging mechanism.
    - Validate presence and correctness of fields such as step, output_token, token_probabilities, and attention_weights.
2. Log Format and Content Validation
    - Verify that the logs conform to expected schema and data types.
    - Check that all required fields are never missing when the logging layer runs normally.
3. Handling Edge Cases
    - Test behavior when model outputs are partial, empty, or include unexpected tokens.
    - Ensure logging does not break if some internal metrics (e.g., attention weights) are unavailable.
4. Timestamp and Metadata Accuracy
    - Confirm timing data is recorded and formatted correctly for temporal analysis.
    - Verify generation config data (like temperature, top-k parameters) is captured accurately.
5. Performance and Resource Management

- Optionally, measure that logging overheads remain acceptable.
- Ensure no memory leaks or resource contention occurs during repeated logging.

Why This Matters

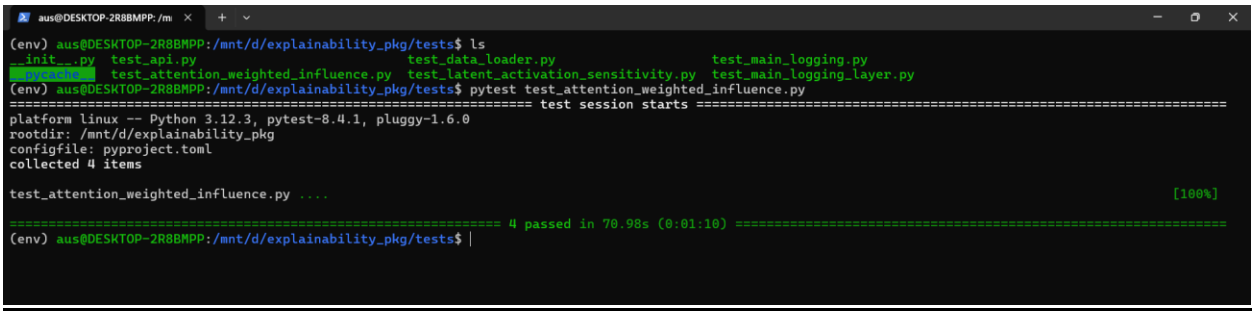
- The logging layer is the foundation of all downstream explainability analyses; without accurate, complete logs, hybrid metrics and explanations cannot be correctly computed.
- Reliable testing here ensures the logging layer is trustworthy, resilient, and fully functional across use cases.
- Solid tests facilitate integration with various generative models and ease future enhancements or debugging.

Summary

test_main_logging_layer.py is a crucial test suite designed to validate the integrity, completeness, and robustness of the core logging infrastructure in your generative AI explainability project. It ensures every inference step is properly recorded with all necessary internal data, forming the backbone for transparent and accurate model explainability.

Now Testing Hybrid Metrics

1. **test_attention_weighted_influence.py**



The test_attention_weighted_influence.py file is a dedicated automated test suite for validating the Attention-Weighted Token Probability Influence hybrid metric module in your explainability package.

Purpose of test_attention_weighted_influence.py

- To verify that the attention-weighted influence metric computes correctly and robustly given various input log scenarios.
- To test that the metric normalizes and processes attention weights properly, including multi-dimensional cases.
- To ensure the module handles edge cases gracefully, such as empty or missing attention or token probability fields.
- To confirm that outputs have the expected structure and data types, supporting downstream compatibility.
- To provide confidence that the metric module behaves as intended before integration into larger workflows.

Typical Tests Included

1. Basic Functionality Test
    - Runs the metric on simulated log entries with 1D and 2D attention weights.
    - Checks that results contain expected keys like step, influence_per_token, total_influence, and valid data types.
2. Empty or Missing Inputs Handling
    - Tests scenarios where attention or token probabilities are empty lists or absent.
    - Verifies the metric returns empty influences or None values without raising exceptions.
3. Attention Shape Handling
    - Provides input with multi-dimensional attention arrays to test averaging and reduction logic.
    - Confirms the metric correctly converts these to usable 1D attention vectors.
4. Attention Normalization
    - Validates that attention weights are properly normalized (summing to 1) before influence calculation.
    - Compares computed influence values against expected mathematical products.

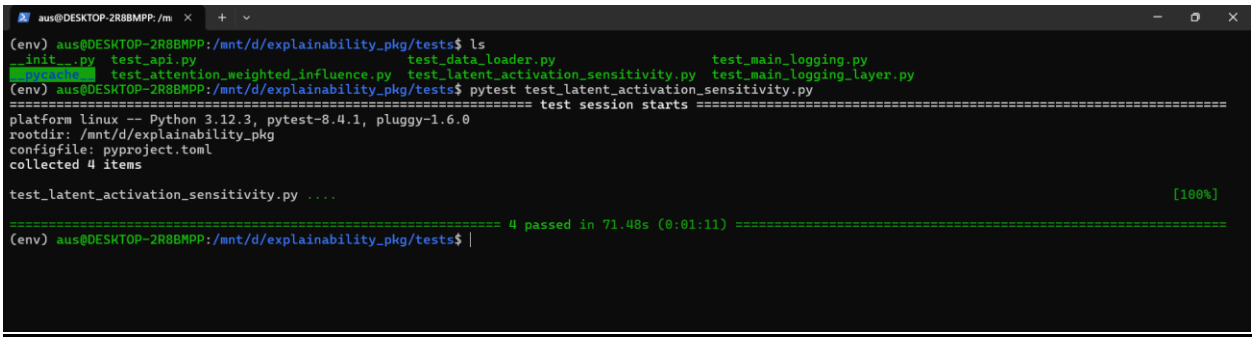
Why This Matters

- The attention-weighted influence metric is a key explainability score quantifying input influence on output generation.
- Comprehensive testing ensures this complex metric reliably reflects model behavior and does not break due to unexpected log shapes or missing data.
- Prevents regressions and ensures smooth integration with the API and explanation generators.

Summary

test_attention_weighted_influence.py is an essential quality assurance module that thoroughly validates the correctness, stability, and error resilience of the Attention-Weighted Token Probability Influence metric. It uses thoughtfully designed test cases to guarantee that this critical component consistently produces valid, interpretable influence scores for generative AI explainability.

2. **test_latent_activation_sensitivity.py**



The test_latent_activation_sensitivity.py file is an automated test suite dedicated to validating the Latent Activation Sensitivity Score hybrid metric module in your explainability package.

Purpose of test_latent_activation_sensitivity.py

- To verify that the latent activation sensitivity metric computes expected results for synthetic or sample log inputs.
- To ensure the metric properly handles various input scenarios, including:
    - Presence or absence of required fields (latent_vectors, token_probabilities).
    - Correct identification and processing of token indices to analyze.
    - Handling out-of-bound token indices gracefully.
- To confirm that the output structure includes expected keys (step, tokens, sensitivities) with appropriate data types.
- To validate that the metric correctly saves its computed results to a JSON file when configured to do so.
- To ensure robustness and error handling without crashes or silent failures.

Typical Tests Included

1. Basic Functionality Test
    - Runs the full sensitivity computation on sample logs with dummy latent vectors and token probabilities.
    - Checks that results match input log length and contain required fields.
2. Token Indices Handling
    - Tests specifying particular token indices for sensitivity computation.
    - Confirms output tokens match requested indices and sensitivities are provided accordingly.
3. Saving Results to JSON
    - Uses temporary directories to verify that computed results are correctly serialized and saved to disk.
    - Checks that saved files contain valid JSON matching the metric output.
4. Handling Missing Fields
    - Tests input log entries where latent_vectors or token_probabilities are missing or null.
    - Confirms such entries are skipped and no incorrect results are produced.

Why This Matters

- The Latent Activation Sensitivity score is an important explainability metric measuring how token prediction probabilities respond to changes in latent space.
- Comprehensive testing ensures the metric works reliably on real and edge-case inputs, correctly processes token selections, and reliably saves results for downstream use.
- Early detection of potential bugs or corner cases during development leads to a robust, production-quality explainability component.

Summary

test_latent_activation_sensitivity.py is a critical quality assurance module providing rigorous validation of the latent activation sensitivity metric's behavior, correctness, input handling, and output serialization. It guarantees that this complex metric consistently delivers interpretable, accurate sensitivity scores and integrates smoothly into the explainability workflow.

(In future I tend to implement more hybrid metrics, it takes time and reseach please be patient)

Now Starting the complete demo.(Please Turn over)

Running The demo screenshot wise :

The multi_model_userdefined_run_generation.py script is an interactive example or utility within your explainability package designed to facilitate generative AI text generation across multiple language models, with user-defined choices for model selection and configuration.

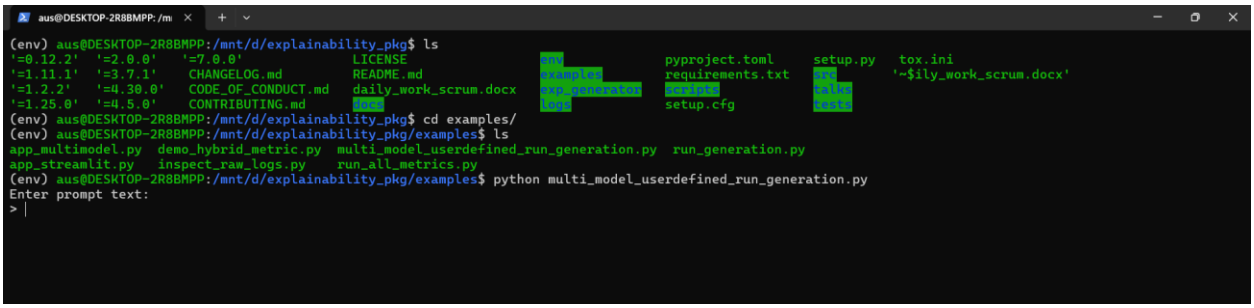
Purpose of multi_model_userdefined_run_generation.py

- To provide a flexible command-line or script interface enabling users to:
    - Choose from multiple available local LLM models or APIs.
    - Configure generation parameters dynamically (e.g., temperature, max tokens, top-k, top-p).
    - Run text generation tasks interactively without hardcoded model specifics.
- To demonstrate multi-model support in your project, showcasing how your explainability tools can work with different generative architectures.
- To capture and log generation outputs and intermediate data compatible with the logging layer, feeding explainability workflows downstream.
- To serve as a practical example for users on how to orchestrate generation workflows with user inputs and configurations in your system.

Key Functionalities

- User Input Interface:
  Prompts the user to select one or more models from a predefined list or configuration file.
- Model Initialization and Loading:
  Dynamically loads and initializes the selected model(s), potentially supporting different backends (e.g., Hugging Face transformers, local or remote APIs).
- Configurable Generation Parameters:
  Accepts user input or command-line arguments to set generation hyperparameters like temperature, max tokens, repetition penalty, and beam width.
- Generation Execution:
  Runs generation with the user-specified settings, processes outputs, and collects raw internal metrics if logging is enabled.
- Logging Integration:
  Records output tokens, token probabilities, attention weights, and other data during generation, producing logs consumable by downstream hybrid metrics.
- Output Display:
  Prints or saves generated text for user inspection, optionally showing generation metadata or confidence scores.

Screenshot 1. The user is asked to enter a text prompt.



Screenshot 2.

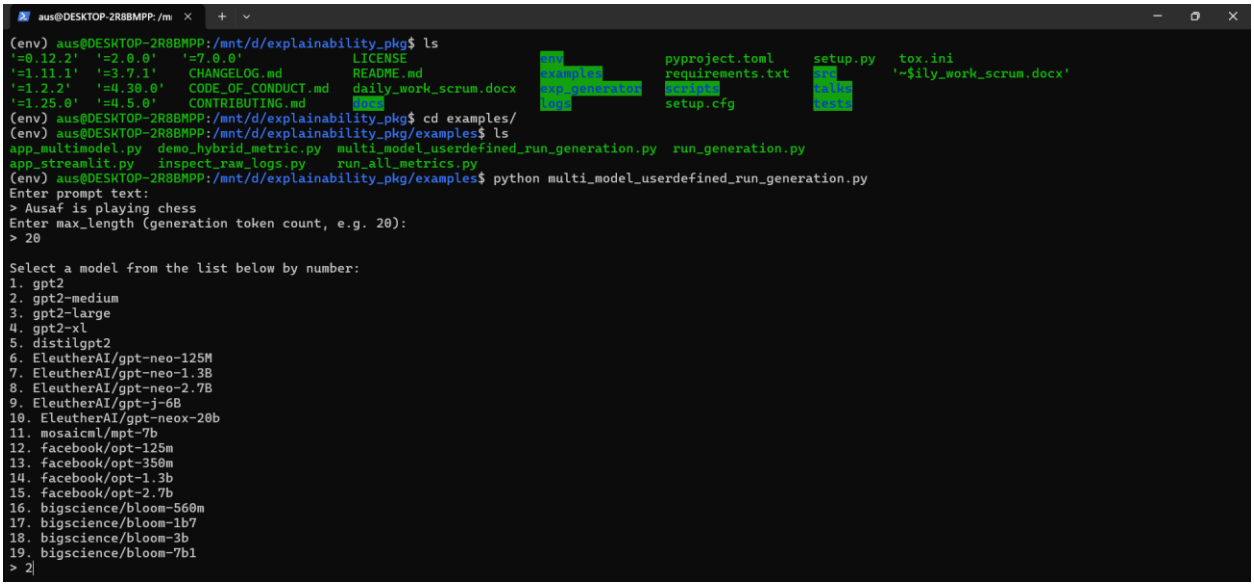
→ The user enters a prompt “Ausaf is playing chess”.

➔ The user is asked to enter max_length (generation token count, e.g. 20)

The user unlike chatgpt or other public llm has full control over the tokens to be generated.

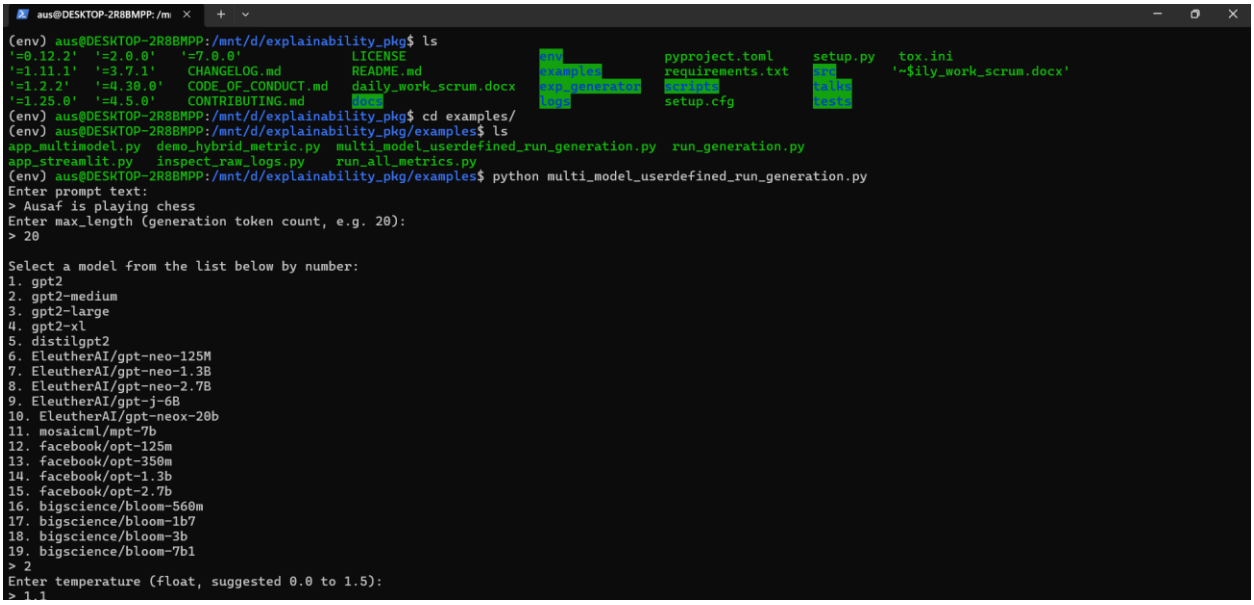
Screenshot 3. User enters ”20” as max token length.

➔ Now user has full control to choose a variety of models for the usecase.
User chooses “2” i.e. 2. gpt2-medium



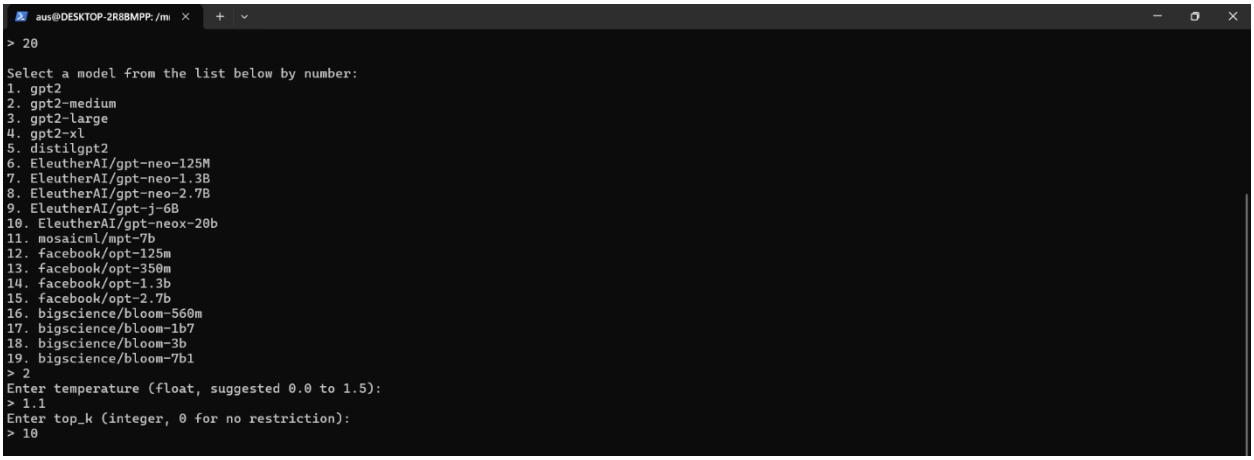
Screenshot 4. User is asked to enter temperature (a float value ranging 0.0 t0 1.5)

➔ User enters ‘1.1’



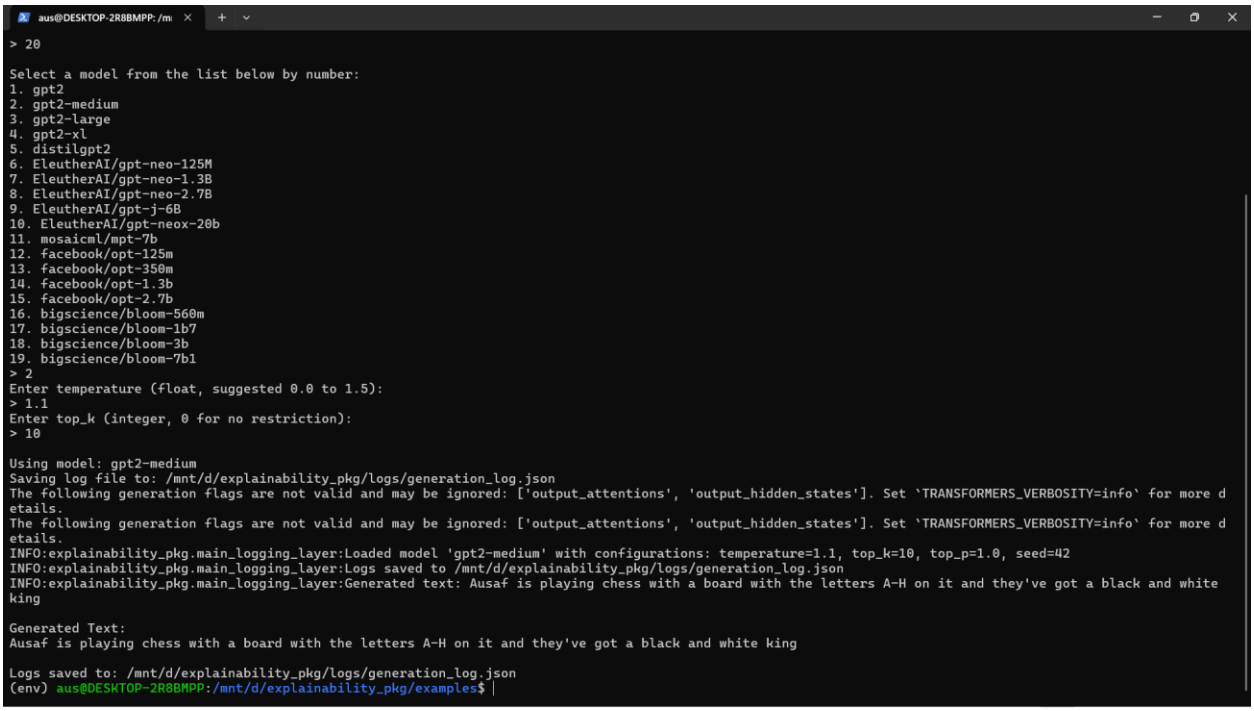
Screenshot 5. Now user is asked to enter top k

➔ User enters “10”

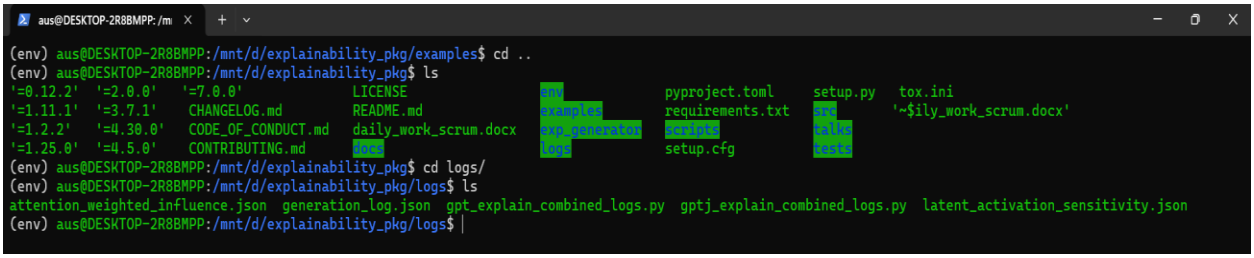


Screenshot 6. Now the complete expression having 20 token length is generated.\

➔ A relative log file is generated. (generation_log.json)



Screenshot 7.Lets visit the log folder to check whether the log generation_log.json is present.

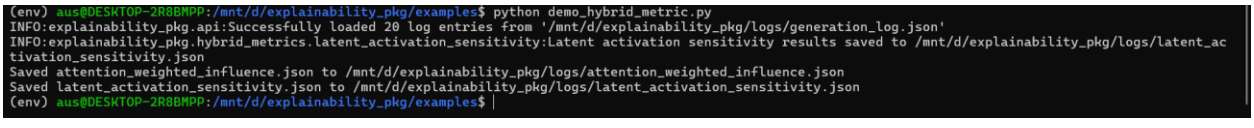


Screenshot 8.

Now Generating logs from Hybrid metrics.

Hybrid metric will fetch logs of logginglayer and then compute them respectively.

This screenshot show the working of Phase 2. Processing Layer.



Lets check the log folder where 2 new json logs of hybrid metrics are produced.

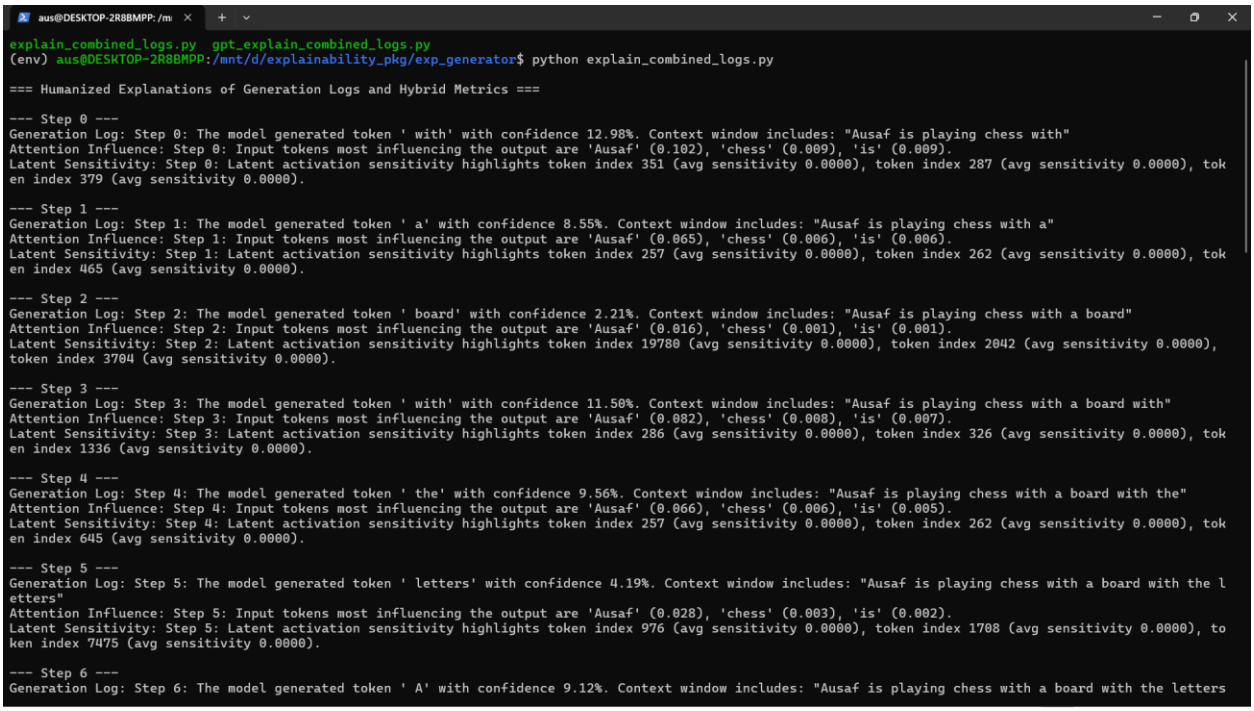
Screenshot 9. Shows the two new logs generated namely:

1. attention_weighted_influence.json
2. latent_activation_sensitivity.json

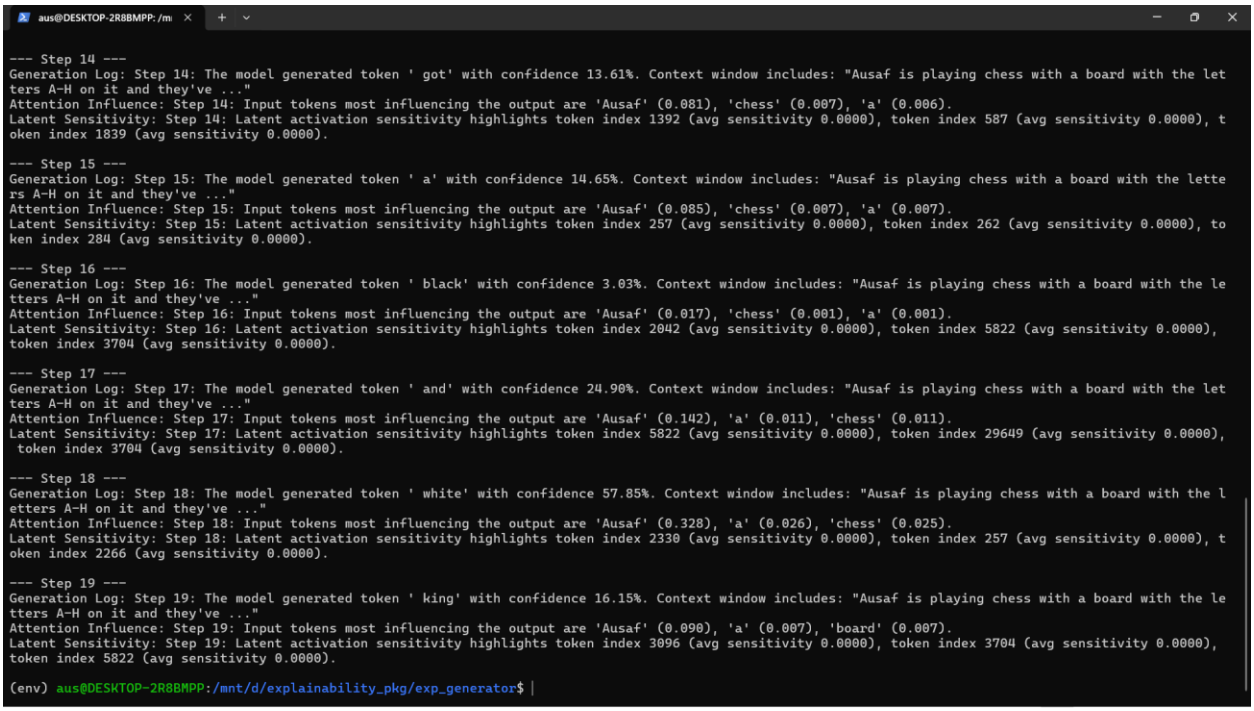
Now going for the phase 3. i.e explanation generator

This explaination generator phase (Phase 3) takes all the logs and generates Human understandable concepts, explaining what is happing at each step of token computation.
I am using GPT2 model since my laptop has low specifications to handle so output might not be expressable.

Screenshot 10-14.




aus@DESKTOP-2R8BMPP: /m
--- Step 12 ---
Generation Log: Step 12: The model generated token ' they' with confidence 10.52%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they"
Attention Influence: Step 12: Input tokens most influencing the output are 'Ausaf' (0.064), 'chess' (0.005), 'a' (0.005).
Latent Sensitivity: Step 12: Latent activation sensitivity highlights token index 262 (avg sensitivity 0.0000), token index 257 (avg sensitivity 0.0000), token index 317 (avg sensitivity 0.0000).
--- Step 13 ---
Generation Log: Step 13: The model generated token ''ve' with confidence 21.60%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've"
Attention Influence: Step 13: Input tokens most influencing the output are 'Ausaf' (0.130), 'chess' (0.011), 'a' (0.010).
Latent Sensitivity: Step 13: Latent activation sensitivity highlights token index 389 (avg sensitivity 0.0000), token index 821 (avg sensitivity 0.0000), token index 423 (avg sensitivity 0.0000).
--- Step 14 ---
Generation Log: Step 14: The model generated token ' got' with confidence 13.61%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 14: Input tokens most influencing the output are 'Ausaf' (0.081), 'chess' (0.007), 'a' (0.006).
Latent Sensitivity: Step 14: Latent activation sensitivity highlights token index 1392 (avg sensitivity 0.0000), token index 587 (avg sensitivity 0.0000), token index 1839 (avg sensitivity 0.0000).
--- Step 15 ---
Generation Log: Step 15: The model generated token ' a' with confidence 14.65%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 15: Input tokens most influencing the output are 'Ausaf' (0.085), 'chess' (0.007), 'a' (0.007).
Latent Sensitivity: Step 15: Latent activation sensitivity highlights token index 257 (avg sensitivity 0.0000), token index 262 (avg sensitivity 0.0000), token index 284 (avg sensitivity 0.0000).
--- Step 16 ---
Generation Log: Step 16: The model generated token ' black' with confidence 3.03%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 16: Input tokens most influencing the output are 'Ausaf' (0.017), 'chess' (0.001), 'a' (0.001).
Latent Sensitivity: Step 16: Latent activation sensitivity highlights token index 2042 (avg sensitivity 0.0000), token index 5822 (avg sensitivity 0.0000), token index 3704 (avg sensitivity 0.0000).
--- Step 17 ---
Generation Log: Step 17: The model generated token ' and' with confidence 24.90%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 17: Input tokens most influencing the output are 'Ausaf' (0.142), 'a' (0.011), 'chess' (0.011).
Latent Sensitivity: Step 17: Latent activation sensitivity highlights token index 5822 (avg sensitivity 0.0000), token index 29649 (avg sensitivity 0.0000), token index 3704 (avg sensitivity 0.0000).
--- Step 18 ---


aus@DESKTOP-2R8BMPP: /m
--- Step 14 ---
Generation Log: Step 14: The model generated token ' got' with confidence 13.61%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 14: Input tokens most influencing the output are 'Ausaf' (0.081), 'chess' (0.007), 'a' (0.006).
Latent Sensitivity: Step 14: Latent activation sensitivity highlights token index 1392 (avg sensitivity 0.0000), token index 587 (avg sensitivity 0.0000), token index 1839 (avg sensitivity 0.0000).
--- Step 15 ---
Generation Log: Step 15: The model generated token ' a' with confidence 14.65%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 15: Input tokens most influencing the output are 'Ausaf' (0.085), 'chess' (0.007), 'a' (0.007).
Latent Sensitivity: Step 15: Latent activation sensitivity highlights token index 257 (avg sensitivity 0.0000), token index 262 (avg sensitivity 0.0000), token index 284 (avg sensitivity 0.0000).
--- Step 16 ---
Generation Log: Step 16: The model generated token ' black' with confidence 3.03%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 16: Input tokens most influencing the output are 'Ausaf' (0.017), 'chess' (0.001), 'a' (0.001).
Latent Sensitivity: Step 16: Latent activation sensitivity highlights token index 2042 (avg sensitivity 0.0000), token index 5822 (avg sensitivity 0.0000), token index 3704 (avg sensitivity 0.0000).
--- Step 17 ---
Generation Log: Step 17: The model generated token ' and' with confidence 24.90%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 17: Input tokens most influencing the output are 'Ausaf' (0.142), 'a' (0.011), 'chess' (0.011).
Latent Sensitivity: Step 17: Latent activation sensitivity highlights token index 5822 (avg sensitivity 0.0000), token index 29649 (avg sensitivity 0.0000), token index 3704 (avg sensitivity 0.0000).
--- Step 18 ---
Generation Log: Step 18: The model generated token ' white' with confidence 57.85%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 18: Input tokens most influencing the output are 'Ausaf' (0.328), 'a' (0.026), 'chess' (0.025).
Latent Sensitivity: Step 18: Latent activation sensitivity highlights token index 2330 (avg sensitivity 0.0000), token index 257 (avg sensitivity 0.0000), token index 2266 (avg sensitivity 0.0000).
--- Step 19 ---
Generation Log: Step 19: The model generated token ' king' with confidence 16.15%. Context window includes: "Ausaf is playing chess with a board with the letters A–H on it and they've ..."
Attention Influence: Step 19: Input tokens most influencing the output are 'Ausaf' (0.090), 'a' (0.007), 'board' (0.007).
Latent Sensitivity: Step 19: Latent activation sensitivity highlights token index 3096 (avg sensitivity 0.0000), token index 3704 (avg sensitivity 0.0000), token index 5822 (avg sensitivity 0.0000).
(env) aus@DESKTOP-2R8BMPP:/mnt/d/explainability_pkg/exp_generator$

Future works that I need to do:

- Study and create new hybrid metrics for better explanations.
- Create the Phase 4. i.e. visualization/ UI.

Please review and share your valueable insights.