# Report on the Functions and Structure of `Final.ipynb`

**Abstract**

This report summarizes the main programmatic components of the Jupyter notebook `Final.ipynb`, which implements a series of experiments on cross-entropy loss sensitivity and soft errors in GPT-2. The focus is on the helper functions and classes defined in the notebook, their purpose, and how they interact to carry out gradient-based sensitivity analysis and bit-flip experiments.

## 1 High-Level Overview

The notebook builds several experimental pipelines around a GPT-2 language model:

- loading GPT-2 and the WikiText-103 dataset;
- scanning gradients to identify the parameters with the largest absolute cross-entropy loss gradients;
- flipping individual bits of selected parameters in floating-point representation;
- generating text under clean and corrupted models;
- computing text-similarity and quality metrics to quantify the effect of bit-flips; and
- saving detailed and aggregated results to CSV files (optionally to Google Drive for Colab runs).

These tasks are organized through a set of utility functions and custom logits processors, described in the following sections.

## 2 Data and Batching Utilities

### 2.1 `chunk_generator()`

The function `chunk_generator()` streams documents from the WikiText-103 training split and tokenizes them with the GPT-2 tokenizer. It maintains a cache of token IDs and, once there are at least `SEQ_LEN + 1` tokens, yields overlapping input–target windows:

- input sequence: the first `SEQ_LEN` tokens;
- target sequence: the same sequence shifted by one position.

This provides training-style language modeling windows for loss and gradient computation.

### 2.2 `get_batch(gen, bs)`

The function `get_batch(gen, bs=BATCH_SIZE)` consumes windows from `chunk_generator()` (or any compatible generator), groups them into mini-batches of size `bs`, and returns each batch as a tensor on the configured device. It is used to feed mini-batches into GPT-2 when scanning gradients.

## 3 Gradient Scan and Bit Selection

### 3.1 `normalize_v_select(sel, k)`

The helper `normalize_v_select(sel, k)` normalizes user-configured rank selections for sensitivity analysis. Allowed forms include:

- the string `"all"`, which expands to all ranks `1, \dots, k`;
- an integer, interpreted as a single rank; and
- a list or tuple of integers, returned as a list.

Invalid inputs raise a `ValueError`. The normalized list of ranks is used to choose which of the top-$K$ sensitive scalars will be tested with bit-flips.

### 3.2 `flip_bit(val_tensor, bit)` and `BIT_CLASSES`

The function `flip_bit(val_tensor: torch.Tensor, bit: int)` implements low-level bit manipulation for 32-bit floating-point tensors:

- it expects a tensor of dtype `float32`; otherwise a `TypeError` is raised;
- the bit index must be between 0 and 31, inclusive; otherwise a `ValueError` is raised;
- values are moved to CPU and viewed as `uint32`; the selected bit is flipped via XOR; and
- the result is converted back to `float32` and returned on the original device with the same shape.

The dictionary `BIT_CLASSES` groups bit positions into semantic classes:

- `"sign"`: the sign bit at position 31;
- `"exponent"`: bits 23–30; and
- `"mantissa"`: bits 0–22.

During experiments, the notebook samples random bit indices from these classes when corrupting selected parameters.

## 4 Metrics and Scoring

### 4.1 `edit_distance(a, b)`

The function `edit_distance(a: str, b: str)` computes the Levenshtein edit distance between two strings using a dynamic programming algorithm implemented in a single-row style for space efficiency. It returns the minimum number of insertions, deletions, or substitutions needed to transform `a` into `b`.

### 4.2 `score_pair(clean, corrupt)`

The function `score_pair(clean: str, corrupt: str)` computes a dictionary of similarity and quality metrics between a clean reference completion and a corrupted completion:

- raw and length-normalized edit distance;
- BLEU score via `sacrebleu`;
- METEOR score (if enabled via the `evaluate` library);
- BERTScore $F_1$ (contextual similarity);
- ROUGE-1, ROUGE-2, and ROUGE-L $F_1$ scores; and
- optionally BLEURT if the corresponding flag is enabled.

Some variants wrap metric computation in `try/except` blocks to handle occasional errors gracefully. The returned dictionary is merged into the per-trial result rows that later form detailed and aggregated CSV tables.

### 4.3 `llm_judge_score(...)`

The placeholder `llm_judge_score(prompt, clean, corrupt, rubric=None)` is defined to optionally integrate an external LLM-based judge. In the current version, when the `ENABLE_LLM_JUDGE` flag is false, it returns an empty dictionary and does not affect downstream computations.

## 5 Logits Processors and Generation Helpers

### 5.1 Clamp and Detection Processors

Several custom subclasses of `transformers.generation.logits_process.LogitsProcessor` are defined to control and monitor the token-level logits during decoding:

- `NanInfClampProcessor` replaces NaNs and infinities in the logits with zeros and clamps all logits to a configurable range. It can also set a flag in a shared dictionary if any NaN or Inf is detected.
- `NanInfDetector` only detects the presence of NaNs or Infs in the logits and records this in a flag dictionary, without modifying the logits.

### 5.2 Repetition Guards

To reduce degenerate repetitive outputs, the notebook defines two related processors:

- `MaxConsecutiveRepeatProcessor` counts how many times the last generated token has been repeated consecutively in the sequence. If this count exceeds a threshold, it sets the corresponding logit to a large negative value, effectively banning that token.
- `MaxRepeatGuard` implements the same idea in other experimental blocks, again limiting excessive consecutive repetition of the last token.

### 5.3 Clean and Corrupt Generation Functions

The notebook defines several helper functions for text generation from GPT-2:

- `generate_tail_clean(prompt, max_new_tokens)` runs greedy decoding without custom processors and returns only the continuation tokens following the prompt.
- `generate_tail_corrupt(prompt, max_new_tokens)` uses a `LogitsProcessorList` that includes `NanInfClampProcessor` and `MaxConsecutiveRepeatProcessor`, and returns both the generated continuation and a boolean flag indicating whether NaNs were encountered.
- `generate_clean(prompt)` and `generate_corrupt(prompt)` implement similar logic for other experimental configurations, using `NanInfDetector` and `MaxRepeatGuard`.
- The internal helper `_generate(prompt, corrupt=False)` wraps all decoding strategy options (greedy, top-$k$, top-$p$, and temperature) and is used to define `generate_clean` and `generate_corrupt` via lightweight wrappers.

All of these helpers ensure that only the newly generated tokens beyond the prompt are returned, allowing straightforward comparison between clean and corrupted outputs.

## 6 Result Aggregation and Output Helpers

### 6.1 `_truncate(s, w)`

The function `_truncate(s, w=MAX_COL_WIDTH)` shortens long string fields (such as prompts and completions) to a maximum width $w$, appending an ellipsis character when truncation occurs. It is used solely for prettier console tables and does not affect the underlying CSV data.

### 6.2 DataFrame Construction and Saving

After running the bit-flip experiments, the notebook builds a `pandas.DataFrame` in which each row contains:

- metadata about the flipped parameter (rank, tensor name, index, bit class, bit index, trial index);
- the prompt, clean continuation, and corrupted continuation; and
- all scalar metrics produced by `score_pair` (and, in principle, `llm_judge_score`).

Helper code then:

- prints a preview table using `tabulate`;
- saves per-trial results to `bitflip_per_trial.csv`;
- aggregates metrics across trials using group-by and summary statistics (mean, median, standard deviation); and
- saves aggregated results to `bitflip_aggregated.csv`, often both locally and to a Google Drive folder when running in Colab.

## 7 Summary

In summary, `Final.ipynb` is structured around a reusable set of functions and custom logits processors that together:

1. construct language modeling mini-batches from WikiText-103;
2. scan gradients to identify the most sensitivity-critical GPT-2 parameters;
3. apply controlled bit-flips to those parameters;
4. generate clean and corrupted completions under multiple decoding strategies; and
5. quantify and record the impact of soft errors using a rich set of text similarity metrics.

The described functions encapsulate these responsibilities cleanly, making it straightforward to run new sensitivity experiments or extend the analysis with additional metrics and decoding schemes.