Midterm Project Report

1. Introduction & Research Shift

Original Focus

- Evaluating the efficiency of LLM-generated code using metrics like runtime, memory consumption, and algorithmic complexity.
- Primary goals of the initial project: dataset curation, defining relevant metrics, and building automated tools for analysis.

Motivation for Change

- The original premise was very broad (likely outside the scope of one semester) and relatively unfocused. Additionally, many papers covered a holistic goal like ours (evaluation frameworks that span all non-functional requirements such as NoFunEval).
- New focus: **fine-tuning an LLM to improve runtime efficiency & memory utilization of generated code**, while balancing the two and comparing the tradeoffs in generated code, especially due to being fine tuned on competitive programming (memory utilization tends to be irrelevant).

2. Progress Update

I. Literature Review & Understanding the Research Landscape

- Identification of gaps in previous efficiency evaluations of LLMs.
- A deeper understanding of state-of-the-art fine-tuning procedures and challenges.
- Optimizing our approach to keep computational costs to a minimum (e.g. exploration of Parameter Efficient Fine Tuning Methods, Dataset Sampling)
- Finalizing datasets that match our requirements for research (described below).

II. Formulating New Research Hypotheses

1. Can fine-tuning improve the **runtime efficiency** of LLM-generated code?
2. Can fine-tuning improve the **memory utilization** of LLM-generated code?
3. What trade-offs in memory usage are expected as a result of optimizing for runtime and vice-versa?
4. Can fine-tuning balance both runtime efficiency and memory consumption to produce high-quality code?

III. Dataset Curation for Runtime Efficiency

- **Runtime Efficiency**: We are using instances from NoFunEdit's Python test split, specifically drawn from the PIE dataset, which includes a rich set of problem pairs that emphasize performance improvements.
- **Memory Utilization**: We are leveraging NoFunEdit's 'Resource Utilization' dataset, curated from GitHub commits on non-functional changes within Android applications. These commits highlight optimizations aimed at reducing memory consumption, which will allow us to train the model to generate more resource-efficient code.

IV. Code Model Preparation

- We are leveraging an established code model, **CodeT5**, as the base for our experiments. Codex is a popular choice in the literature due to it being available for open-source use. For our initial fine-tuning, we are focusing on a smaller, manageable sample using Google Colab. This allows for quick iterations and testing. For larger-scale experiments, however, we plan to utilize virtual machine credits to run on the Google Cloud environment that can handle more intensive computational loads.

V. Fine-tuning Strategy

- We are employing a task structure similar to the Performance-conditioned Generation approach used in the PIE dataset study. This involves training the model to predict performance-improving edits using a mix of both minor and significant improvements from the dataset.
- To give the model a nuanced understanding of performance optimization, we associate performance tags, but only categorize the level of optimization using binary classification.

Example Prompt Structure

```
Problem: Optimize the function to reduce runtime in large inputs by
minimizing redundant operations.

Initial Code:

def quicksort(arr):

    if len(arr) <= 1:

        return arr
```

```
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
Performance Tag: 0


Expected Optimized Output:
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left, right = [], []
    for x in arr:
        (left if x < pivot else right).append(x)
    return quicksort(left) + [pivot] + quicksort(right)
Performance Tag: 1
```

This format allows the model to identify patterns between code improvements and performance tags, fostering a more nuanced understanding of efficient coding practices over time. During inference, we guide the model by using a test input coupled with this performance tag, instructing it to generate code that prioritizes high performance.

VI. Preliminary Fine-tuning runs:

The following early-stage fine-tuning experiment serves as a testbed to verify the feasibility of our approach before we expand to larger datasets and conduct a more thorough performance analysis.

- Model: Using a subset of the pre-trained CodeT5 model.
- Training Batch Size: 16 examples per batch for sorting tasks.
- Learning Rate: 5e-5, to ensure stability during fine-tuning.

- Epochs: 3 epochs for initial testing.
- Objective: The model should learn to prioritize implementations that reduce runtime while maintaining correctness, guided by the performance tags.

3. Future steps: Experimental Design & Planning

Fine-Tuning Procedure

1. Conducting an official fine tuning of the base LLM on the runtime efficiency dataset.
2. Implementing a second phase of fine-tuning to incorporate memory-efficient code.
3. Investigating the impact of this sequential, two-step fine-tuning on code quality, memory consumption, and overall efficiency.
4. If time permits, fine-tune the LLM on each dataset individually and compare outcomes for each against memory consumption & runtime efficiency.

Experimental Evaluation Framework

- Use Effibench, a benchmark of 1,000 efficiency-critical LeetCode coding problems, each paired with an optimal human-written solution. This will provide us with a framework to assess the following metrics: Memory and Runtime-related metrics: Execution Time (ET), Normalized Execution Time (NET), Max Memory Usage (MU), Normalized Max Memory Usage (NMU), Total Memory Usage (TMU), and Normalized Total Memory Usage (NTMU).

- Compare the fine-tuned model's performance to the base model and conduct regular evaluations post each training phase to track improvements or regressions.

4. Challenges Predicted & Mitigation Strategies

(i) Dataset Integration & Curation

- Challenges:
  - Merging two datasets with distinct focuses (runtime efficiency and memory efficiency).
  - Ensuring that the dataset diversity covers a wide range of programming challenges.
- Mitigation:
  - Implementing robust filtering criteria to ensure that training data aligns with dual optimization goals.
  - Developing a preprocessing pipeline to normalize data inputs across both datasets.

(ii) Fine-Tuning Stability

- Challenges:
    - Risks of catastrophic forgetting during multi-phase fine-tuning.
    - Trade-offs in model performance due to the focus on different efficiency metrics.
- Mitigation:
    - Implementing early stopping during training to prevent overfitting.
    - Monitoring performance metrics continuously to adjust fine-tuning parameters dynamically.