Data Science Report

Project: Academic Study-Guide Agent
Model: NoteExtractor (Fine-Tuned google/gemma-2b-it)
Author: Rishabh Kumar

1. Fine-Tuning Setup

This section details the data, method, and results for the project's mandatory fine-tuned model.

A. Data: The dataset.jsonl File

The core of this project was to create a model that could adopt my personal, **"adapted style"** of academic note-taking. A general-purpose model cannot do this, as it does not know my specific mental hierarchy for what constitutes a "main topic," a "detail," or a "child topic."

To solve this, I created a custom dataset named dataset.jsonl.

- **Format:** JSON Lines (.jsonl), where each line is a complete input/output example.
- **Content:** The dataset contains **27 examples** manually created from my own university lecture notes.
- **Structure:** Each example has two keys:
  1. **"input"**: A raw text paragraph from my lecture notes.
  2. **"output"**: The "perfect" hierarchical JSON object that I would manually create for that text, representing the structure of a mind map.

Example from dataset.jsonl:

{"input": "A Binary Search Tree (BST) is a node-based binary tree data structure which has the following properties: ...", "output": {"main_topic": "Binary Search Tree (BST)", "details": "A node-based binary tree...", "children": [{"topic": "Property 1", "details": "Left subtree keys are *lesser*..."}, ...]}}

This dataset was used to teach the model its single, specialized task: **reliably converting unstructured lecture text into my personal, structured JSON format.**

B. Method: LoRA Fine-Tuning

- **Base Model:** google/gemma-2b-it. This model was chosen as it is a powerful, instruction-following model that is small enough to be fine-tuned on a single GPU.
- **Technique:** Parameter-Efficient Fine-Tuning (PEFT) using **LoRA** (Low-Rank Adaptation). This was ideal as it allowed me to train the model on a free Google Colab T4 GPU by only updating a tiny fraction of the model's total parameters (approx. 0.08%).
- **Training:** The model was trained using the SFTTrainer from the TRL library with the

following key parameters:

- num_train_epochs: 3
- per_device_train_batch_size: 1
- gradient_accumulation_steps: 4
- learning_rate: 2e-4
- optim: "paged_adamw_8bit"

C. Results: Training Loss

The fine-tuning was a success. The model learned effectively from the 27 examples, as demonstrated by the **quantitative** decrease in the training loss over 3 epochs. The loss, which represents the model's error rate, dropped from an initial ~2.03 to ~0.95, showing that the model was successfully learning to mimic the target JSON structure.

(Screenshot of the training loss log would be included here)

```
Step 5: Training Loss: 2.034100
Step 10: Training Loss: 1.459500
Step 15: Training Loss: 1.091600
Step 20: Training Loss: 0.957500
...
```

2. Evaluation Methodology and Outcomes

To fulfill the **mandatory requirement** for evaluation, I designed a qualitative test to measure the reliability and quality of the fine-tuned model.

A. Evaluation Methodology

A "hold-out" test was performed. I selected a new paragraph of text from a lecture (on "Python Dictionaries") that was **not** included in the 27 training examples.

This unseen text was fed into the fine-tuned model (Step 6) to evaluate two things:

1. **Quality:** Would the model correctly understand the new topic's hierarchy?
2. **Reliability:** Would the model output a *perfectly valid* JSON object, or would it fail?

B. Outcomes & Iteration

This evaluation proved to be the most critical part of the project, as it uncovered a key "hallucination" flaw that had to be engineered around.

Outcome 1: The Initial Test (Failure)

The model was given the "Python Dictionary" text. It successfully processed the text and generated a well-structured JSON object. However, it also added a single, random word (Darío) after the final closing brace }.

Raw Model Output (Truncated):

```
... "details": "They are defined using curly braces { }."}]}Darío
```

This small hallucination caused the agent's JSON parser to fail, resulting in an ❌ FAILED TO PARSE JSON error.

Outcome 2: The Solution (Success)

This evaluation was a success. It demonstrated that the fine-tuned model *did* learn its task (generating the correct JSON) but was not 100% reliable, as it could add extra, non-JSON text.

This finding led to an engineering improvement in the final agent (agent.py). Instead of naively parsing the model's entire output, the agent's code was made more robust. It now uses string-finding logic to **extract only the text between the first { and the last }**.

Robust Parsing Logic (from agent.py):

```
json_start_index = generated_text.find('{')
json_end_index = generated_text.rfind('}') + 1
json_string = generated_text[json_start_index:json_end_index]
parsed_json = json.loads(json_string)
```

With this fix, the agent successfully ignored the "junk" text (Darío), parsed the JSON, and the entire "Reason, Plan, Execute" pipeline ran to completion, resulting in a ✅ SUCCESS! message.

This iterative process of **Test -> Find Flaw -> Engineer Solution** demonstrates the model's capabilities and limitations, and results in a more robust and reliable final agent.