AI-Generated Code Similarity Prediction Model

Introduction

Large Language Models (LLMs) such as GPT, LLaMA, and others have shown remarkable capabilities in understanding and generating human-like text. While these models perform well on general tasks, customizing their behavior for specific applications or domains often requires fine-tuning. Fine-tuning an LLM allows the model to adapt to domain-specific language, rules, or specialized tasks, enhancing its accuracy and relevance for the desired application.

This document details the methodology implemented for predicting AI-detected scores in the given task. The process involves the following key steps: pre-processing, data preparation, model fine-tunning, evaluation and deployment.

In this assignment I have used LLaMA 3 for fine-tunning using Unsloth which is a toolkit designed to simplify the process of enhancing large language models for specific applications.

Methodology

1. Data Understanding

Dataset Description:

- **Input Data**: A dataset containing coding questions, candidate answers, AI-generated responses, and their corresponding AI-detected scores.
- **Score Range**: The AI-detected scores range from 0 (no AI characteristics detected) to 1 (high AI characteristics detected).
- **Fields**:
  - `coding_problem_id`
  - `llm_answer_id`
  - `candidate_code`
  - `ai_code`
  - `question`
  - `rules`
  - `examples`
  - `programming_language`
  - `plagiarism_score` (Target Variable)

2. Pre-processing

- Data Loading
  - **Objective**: Load the dataset containing candidate code, AI-generated code, question details, and AI-detected scores.
  - **Steps:**
    1. Using python, iterated through the directories, accessed the files and extracted the features needed.

- Data Preparation
  - **Objective**: Creating the structured dataset.
  - **Final Result:**
    - The extracted features are:

```
"coding_problem_id",
"llm_answer_id",
"candidate_code",
"ai_code",
"question",
"rules",
"Examples",
"Programming_language"
```

    - The final dataframe has 378 rows and 8 columns.
    - The Fig1 shows the final structured dataset that has been prepared.

coding_problem_id	llm_answer_id	candidate_code	ai_code	question	rules	examples	programming_language	plagiarism_score
source_code_000	gpt-3.5-turbo_00	fun findLargestElement(array: IntArray) : Int ...	public class LargestElementFinder {\n publi...	Write a program to find the largest element in...	['The array can have duplicate elements.']	Input: [1, 4, 2, 9, 5]\nOutput: 9	Java	0.0
source_code_000	gpt-3.5-turbo_01	fun findLargestElement(array: IntArray) : Int ...	public class Main {\n public static void ma...	Write a program to find the largest element in...	['The array can have duplicate elements.']	Input: [1, 4, 2, 9, 5]\nOutput: 9	Java	0.0
source_code_000	gpt-4-turbo_00	fun findLargestElement(array: IntArray) : Int ...	public class Main {\n public static void ma...	Write a program to find the largest element in...	['The array can have duplicate elements.']	Input: [1, 4, 2, 9, 5]\nOutput: 9	Java	0.0
source_code_000	gpt-4-turbo_01	fun findLargestElement(array: IntArray) : Int ...	public class LargestElement {\n public stat...	Write a program to find the largest element in...	['The array can have duplicate elements.']	Input: [1, 4, 2, 9, 5]\nOutput: 9	Java	0.0
source_code_000	gpt-4_00	fun findLargestElement(array: IntArray) : Int ...	public class Main {\n public static void main...	Write a program to find the largest element in...	['The array can have duplicate elements.']	Input: [1, 4, 2, 9, 5]\nOutput: 9	Java	0.0

Fig1

    - Created train set and test set out of the final dataframe I have prepared.
      - Train set(70%)
      - Test set(30%)

3. Feature Engineering

- Input Formatting
    - Generated prompts combining context (question, rules, examples) and candidate/AI code for training and evaluation. This structure allowed the LLM to focus on the task while understanding the context holistically.

```
"""Below is an instruction that describes a task, paired with an input
that provides further context. Write a response that appropriately
completes the request.

### Instruction:
Analyze the provided details and predict the likelihood that the
Candidate Code is AI-generated. The score ranges from 0 to 1, where:
- 0 indicates no AI characteristics detected.
- 1 indicates a high likelihood of AI characteristics.
Base your analysis on the similarities and differences in syntax,
structure, variable naming, and logical flow between the Candidate Code
and the AI-Generated Code.

### Input:
{{
   "question": "{}",
   "rules": "{}",
   "examples": "{}",
   "programming_language": "{}",
   "candidate_code": "{}",
   "ai_code": "{}"
}}

### Response
{}"""
```

-
    - From the generated prompt, created a new feature to train the model.
    - Then converted the dataset into hugging face dataset.

- Tokenization

  Unsloth internally performs RoPE Scaling, so larger maximum sequence lengths are automatically supported. Otherwise the API is pretty much the same as transformers' from_pretrained, except that `FastLanguageModel.from_pretrained` also returns the model tokenizer for convenience.

Therefore I used the tokenizer from Unsloth itself.

- **Objective**: Convert textual data into a machine-readable format.
- **Steps**:
    1. Leveraged the tokenizer associated with the `unsloth/llama-3-8b-bnb-4bit` model.

4. Model Selection Rationale

a. Selected Model

- Model: `unsloth/llama-3-8b-bnb-4bit`.
- Rationale:
    1. LLaMa 3, a high-performance open-source model, optimized with 4-bit quantization for memory efficiency.
    2. Optimized for efficient inference using 4-bit quantization.
    3. Pre-trained on diverse datasets, making it robust for text understanding and regression tasks.
    4. Capable of processing complex textual inputs like code and natural language descriptions.

b. Fine-Tuning

- Employed PEFT (Parameter-Efficient Fine-Tuning) with LoRA (Low-Rank Adaptation) for efficient model adjustments without retraining the entire network.
- Configured with the Hugging Face `SFTTrainer` using:
    1. Batch size: 2
    2. Max sequence length: 2048
    3. Gradient accumulation: 4
    4. Learning rate: `2e-4`
    5. Warm-up steps: 5
    6. Optimizer: AdamW with 8-bit optimizations.
- Performance monitoring during training included GPU memory utilization and training runtime.

5. Inference

- Custom prompts analyzed syntax, structure, and logical flow between candidate and AI code.
- The model predicted scores within the range [0, 1] for AI similarity likelihood.
- Extracted results using tokenized decoding and regular expressions.

6. Evaluation

a. **Metrics:**

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- R-squared (R^2)

Results from evaluation indicated the model's ability to closely predict the AI-detected score based on the input features.

The current evaluation results:

```
Mean Absolute Error (MAE): 0.2412

Mean Squared Error (MSE): 0.1259

Root Mean Squared Error (RMSE): 0.3548

R-squared (R²): -0.7427
```

The interpretation

- **Good**: The error metrics (MAE and RMSE) might seem low in absolute terms depending on the target variable's scale.
- **Bad**: A negative R^2 indicates that the model is fundamentally flawed and fails to capture the underlying patterns in the data.

b. **Model Saving:**

Saved the fine-tuned LoRA adapters and tokenizer locally for reusability:

```
model.save_pretrained("/path/to/save/lora_model")
tokenizer.save_pretrained("/path/to/save/lora_model")
```

7. Deployment

The code development is still in progress

8. Dependencies and installation

a. **Requirements:**

Python Packages:

- `torch`

- `transformers`
- `trl`
- `unsloth`
- `pandas`
- `Scikit-learn`

b. Installation:

Followed [Unsloth Documentation] for setup:

9. Experiment

- Applied data cleaning for the ai code and candidate code and experimented the results.

Data cleaning steps

1. Removed Comments:
   - Handles single-line (`//`, `#`) and multi-line (`/* */`) comments.
2. Normalized Variable Names:
   - Replaces all variable names with placeholders like `var1`, `var2` for uniformity.
   - A counter ensures unique placeholders for each variable.
3. Standardized Whitespace:
   - Consolidates multiple spaces and newlines for clean formatting.
4. Consistent Formatting:
   - Uses `autopep8` to auto-format Python code. For other languages, similar formatters can be applied (e.g., `Prettier` for JavaScript or `PHP-CS-Fixer` for PHP).

Evaluation results after data cleaning:

```
Mean Absolute Error (MAE): 0.2316

Mean Squared Error (MSE): 0.1070

Root Mean Squared Error (RMSE): 0.3271

R-squared (R²): -0.4816
```

10. Results comparison

Model Performance Metrics Comparison

- **First Model (before data cleaning):**
  - MAE: 0.2412
  - MSE: 0.1259
  - RMSE: 0.3548
  - R^2: -0.7427 (Negative R^2 indicates poor model performance)

- **Second Model (after data cleaning):**
  - MAE: 0.2316
  - MSE: 0.1070
  - RMSE: 0.3271
  - R^2: -0.4816 (Negative R^2, but less negative than the first model)

11. Conclusion

While both models (model before data-cleaning and after data-cleaning) show some potential, their negative R^2 values and error metrics indicate the need for substantial improvements. Focusing on advanced models, better feature engineering, and optimized hyperparameters should help enhance performance and provide more accurate predictions.