CodeReviewer.AI

Mid PPT

Team members:

Yogev Cohen

Dudi Ohayon

Romy Somkin

Description

Transferable Code Review via Language-Agnostic Augmentation with LLMs

<u>Task:</u>

Binary Code Review Classification

- **Input**: A code change (composed of the original code **oldf** and the **patch**
- **Output**: A binary label:
  - o **0** – High quality, no further review needed
  - o **1** – Low quality, requires further review

<u>Data Sources</u>:

Dataset– https://huggingface.co/datasets/fasterinnerlooper/codereviewer

- **Real Data**: Java and C++ code review examples (from GitHub PRs)
- **Synthetic Data**: Java examples translated to C++ using LLM-based code translation

<u>Model Choice Rationale:</u>

We use CodeBERT as our base model because it was **not pre-trained on C++**, making it ideal for **transfer learning**. This allows us to evaluate how well it generalizes when fine-tuned on both real and augmented C++ data, enabling a clear evaluation of **language transfer** and **augmentation impact**.

Prior Art

Source/Title	CodeReviewGPT (Shippie)	Amazon CodeWhisperer	LLaMA-Reviewer
Approach/Model	Custom GPT model fine-tuned on code reviews, integrates LLM feedback and heuristics	Transformer-based model trained on billions of lines of code from open-source and Amazon	LLaMA-based model trained to mimic expert reviewers using reinforcement learning
Data	Real-world pull requests + human review comments from GitHub (multi-language)	Open-source codebases + proprietary Amazon data	Curated reviews with expert-labeled decisions from multiple repositories
Metrics	Agreement with human reviewers, reduction in review time, precision-recall on classification	Suggestion relevance, developer acceptance rate	Accuracy, F1 score vs expert labels, time saved
Results	Achieves human-level agreement in 68% of cases, reduces review time by ~30%	High developer adoption, speeds up review, improves code quality	81% agreement with experts, improves decision confidence by 25%

Steps

<u>Preprocessing</u>:

o Extract code diffs (oldf + patch)

o Format examples for model input - Concatenate the original code (oldf) and its corresponding patch into a unified input sequence to provide the model with contextualized code changes

<u>Data Augmentation</u>:

o Use LLM to translate Java diffs into C++ for synthetic training data

<u>Model Training (Transfer Learning):</u>

o Base Model: CodeBERT

o M_source: Fine-tuned on real C++ data

o M_aug: Fine-tuned on translated Java → C++ data

<u>Evaluation:</u>

o Test set: Evaluate both models on a held-out C++ test set and compare their performance.

o Metrics per model: Accuracy, Precision, Recall, F1-score

<u>Expected Output (per step):</u>

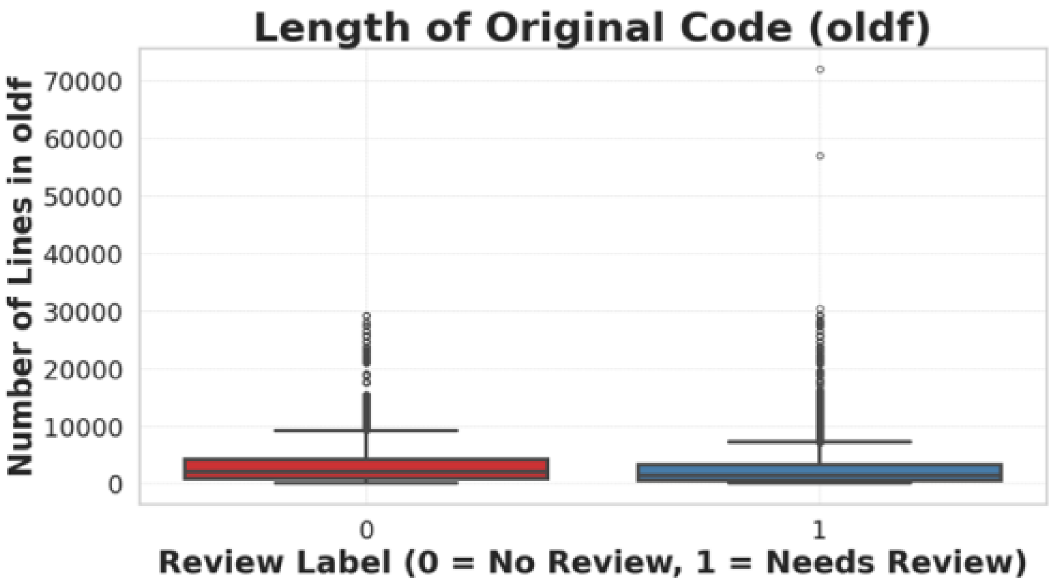
- **Input:** Code diff → Tokenized format
- **Model Output**: Binary label (0 or 1)
- **Final Output:** Performance metrics comparison across models

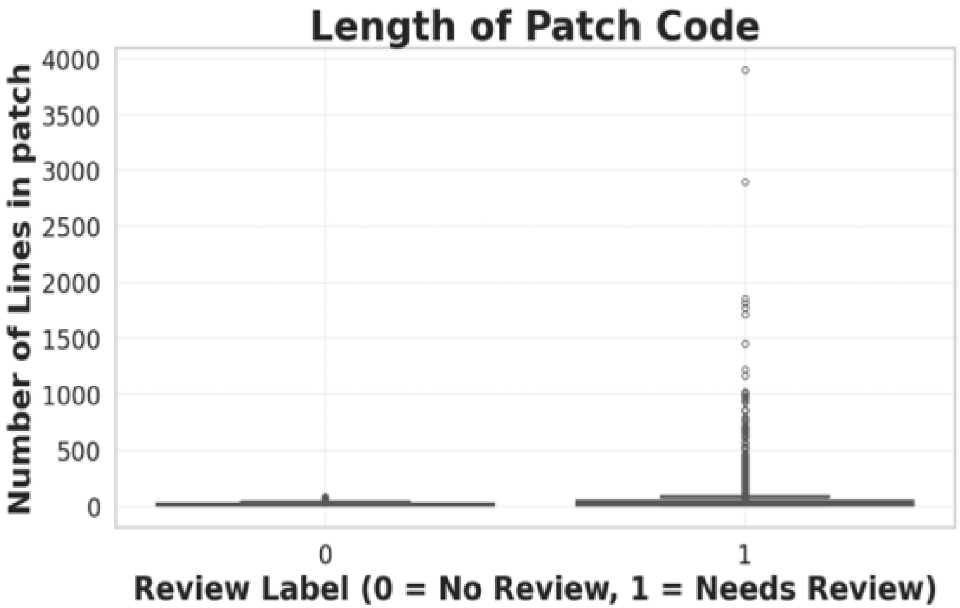
Exploration and Baseline

Dataset

Class Distribution

Code Length Distribution





Baseline Setup

Training: 3 epoch on 50% of real C++ training data.

Initial Results:

- **Accuracy**: 46.9%
- **F1 Score:** 0.423
- **Precision:** 0.359
- **Recall:** 0.514

Conclusions

•**Combining real and synthetic data improves model robustness**, especially in cross-language scenarios where labeled data may be scarce.

•**Evaluating models on held-out C++ test sets reveals the impact of augmentation**, showing the performance trade-offs between training with only real data versus including translated code.

•**Increase training epochs** to allow better convergence, especially for complex code sequences.

• **Balance the dataset or apply class weighting** to mitigate possible class imbalance.