Fine-tuning Code LLMs to Generate Branch Prediction Attributes in C++

Pranav Yandamuri
Univeristy of Illinois
Urbana-Champaign
pyanda2@illinois.edu

Lilesh Kurella
University of Illinois
Urbana-Champaign
lileshk2@illinois.edu

Justin Lin
University of Illinois
Urbana-Champaign
jklin6@illinois.edu

Abhi Sathya
University of Illinois
Urbana-Champaign
asathya2@illinois.edu

Rohit Chatterjee
University of Illinois
Urbana-Champaign
rohitc3@illinois.edu

KEYWORDS

Natural Language Processing, Code Classification

1 ABSTRACT

As compilers evolve, optimizing execution time, storage size, and power consumption of software remains a challenge, particularly in the realm of branch prediction. Branch prediction has typically relied on static analysis and heuristics, which can fail to capture the dynamic nature of code behavior. The introduction of the [[likely]] and [[unlikely]] attributes in C++ 20 have provided developers with a manual approach to hint the compiler for proper branch prediction.

This paper addresses the need for an automated and trustworthy protocol to accurately predict execution likelihood for different if statements. We approach this through fine tuning a classification LLM that parses segments of C++ code.

2 PROBLEM

Compilers implement various optimization strategies as they evolve through time to enhance performance and efficiency. The goal for compiler implementation is to minimize execution time, storage size, and power consumed.

Authors' addresses: Pranav Yandamuri, Univeristy of Illinois Urbana-Champaign, pyanda2@illinois.edu; Lilesh Kurella, University of Illinois Urbana-Champaign, lileshk2@illinois.edu; Justin Lin, University of Illinois Urbana-Champaign, jklin6@illinois.edu; Abhi Sathya, University of Illinois Urbana-Champaign, asathya2@illinois.edu; Rohit Chatterjee, University of Illinois Urbana-Champaign, rohitc3@illinois.edu.

A core area of this challenge lies within the CPU's ability to predict and pre-fetch instructions before their execution paths are definitively known, a process known as branch prediction. Branch prediction involves selecting the clause the executable file will most likely follow to reduce the execution time in performance-critical software. Traditional compilers employ static analysis and heuristics to optimize branch prediction, yet these methods often fall short of capturing the dynamic nature of runtime behavior. Branch prediction has been heavily studied before the rise in AI and ML as well as significantly after. There have been works on reinforcement learning [3] to figure out the correct path, as well as deep learning into very low level coding structures of chip design and much more [2].

In the scope of branch prediction, the if statement, in C++ and most languages, is a fundamental structure that dictates the direction in which a compiler runs its executable files. Its execution can lead to overestimation of CPU resources. C++ 20 introduced the [[likely]] and [[unlikely]] attributes, which help manually hint the compiler at the expected path of branch execution [1]. The concern with its application in comparison to current built-in predictions is that these attributes hinge upon the developers' ability to properly predict the likelihood of an executable path being selected. This can lead to scenarios of error-prone labeling and are unfeasible for large and verbose codebases.

This concern leads to a critical need in C++ 20: a more profound and automated protocol to predict an if statement's likelihood of putting the executable path down its particular branch. This protocol would not

only enhance compiler optimizations more effectively but also alleviate the burden on developers to manually annotate branches with the [[likely]] and [[unlikely]] attributes.

3 SOLUTION

To solve the problem of code branch prediction, our solution consists of two parts. As previous datasets revolving around Code Branch prediction have not been created, a custom dataset was required to fine-tune our desired models for this task. With the custom dataset that consists of C++ code examples, we are able to fine-tune existing Code LLMs. Link to GitHub: https://github.com/pyanda2/Code-Branch-Prediction

3.1 Dataset

We created the dataset by scraping all public C++ code files on GitHub that have [[likely]] or [[unlikely]] tags, using the GitHub code search API. Altogether, we gathered 13,937 code files. Our final dataset consists of 30,458 examples pulled from those files. Each example consists an if statement and the 50 lines of code that immediately precede it (fewer than that if the if statement is close to the start of the file). In the data, we observed that [[likely]] and [[unlikely]] tags were mostly used with if statements, so to simplify the dataset we did not include examples of other kinds of statements where those tags could be used.

The dataset is further preprocessed before training. Each example is given one of two labels ([[likely]], [[unlikely]]) based on which tag (if any) the ending if statement has. A total of 7,923 examples were labeled [[likely]] and 22,535 [[unlikely]]. Next, all [[likely]] and [[unlikely]] tags are stripped from the example, so that the model must predict the label based on the logic of the code.

3.1.1 Dataset Adjustment. We hypothesized that any code prior to the if statement is unnecessary because the branch the compiler takes is solely dependent on code prior to the branches. After feedback, we've realized that an LLM can pick up on patterns from within a branch's code to determine likelihood. Thus, we include all code from the beginning to the end of the if body.

Additionally, instead of scraping only 50 lines of code prior to the if statement, we locate the beginning scope (generally a function or class) that the if statement belongs in. This way, we do not segment parts of code but instead have a continuous piece. We believe this facilitates interpretation for the LLM.

Given our two adjustments, we run evaluation on both datasets and compare the results.

3.2 Fine-tuned Models

Using our datasets, we fine-tune two Code LLMs: CodeBERT-base, and StarCoderBase-1b. For this project, using already pre-trained Code LLMs is useful because the models have knowledge of code semantics and syntax from the pre-training process, which would help them learn to classify likely and unlikely branches. This approach, transfer learning, allows faster convergence for our specific task.

Both models are trained using the Hugging Face Trainer API. Since both models are pre-trained to generate code, we load the models with the AutoModelForSequenceClassification loader provided by Hugging Face, which adds an extra output later of dimension k, where k is the number of classes the model can predict (in our case 2). This allows us to use the models for classification.

Our models take as input a section of code containing an if statement (marked with a sentinel) and output a class corresponding to one of two labels: [[likely]] and [[unlikely]]. Since each example in the training dataset has a ground truth label of [[likely]] or [[unlikely]], we use cross-entropy loss to measure the discrepancy between the model's predictions and the ground-truth labels.

We use the same datasets, training pipeline, and hyperparameters for training both CodeBERT and StarCoder, with the only difference being a batch size of 6 for StarCoder (versus 16 for CodeBert) due to memory limitations. We also use each model's existing tokenizer without modification. Additionally, we use DeepSpeed ZeRO Stage 2 along with mixed-precision training (FP16) to allow training of StarCoder on memory-constrained Google Colab instances, although this does not affect training results.

Each model is trained for 4 epochs. The learning rate increases linearly from 0 to 3e-4 for the first 500 steps, then decreases linearly back to 0 until the end of the 4th epoch. All other hyperparameters are kept as defaults provided by Hugging Face.

3.3 Pipeline

The overall pipeline can be seen in Figure 1 below. We start of by curating our dataset by scraping multiple GitHub pages for likely and unlikely tagged code. We then pre-process this data as mentioned in 3.1. This dataset is then utilized in the to fine-tune the models for classification. Finally, we evaluate both typical classification metrics and runtime analysis of newly generated code corresponding to the classifications outputted.

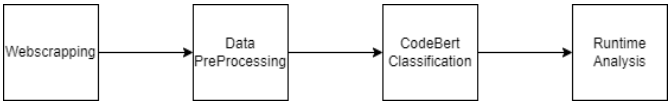

Figure 1: Main pipeline

4 EVALUATION

4.1 General Metrics

We use separate evaluation datasets to test our models. The metrics we measure are training loss, validation loss, precision, recall, and F1 score. The loss function quantifies the difference between our predicted outputs and the actual outputs. Since we have three labels, the `precision_recall_fscore_support` function from the `sklearn.metrics` library finds the recall, precision, and F1 score by treating each class as a "positive" label and calculating per class. Then, a final score is computed as an average of the three.

4.2 Runtime Analysis

We performed runtime analysis on an MP taken from Intro to Computer Science II, which is a freshman-level C++ class at the University of Illinois Urbana Champaign. The MP implements code to perform seam carving, which is the removal of seams of pixels that contain the least information (energy). Seam carving allows images to be resized up to a certain point without the typical squashing, stretching artefacts. The following (Figure 2) is an example of a source image and a carved image (Figure 3) which is the result of removing 170 vertical seams and 50 horizontal seams. We chose Seam Carver for runtime analysis as it has a lot of branches that execute in loops. For each seam that is found and carved a dynamic programming algorithm accumulates energies from the bottom up (right to left for horizontal seams), finding the amount of energy lost for removing

Figure 2: Ocean Source Image (507×285)

Figure 3: Ocean Carved Image (337×235)

the least cost seam starting at pixel (i,j) for every pixel in the image. Then the seam with the least cost, which is the seam starting at the pixel with the minimum accumulated energy at row 0 for vertical seams and column 0 for horizontal seams is traced and removed. The branches in this code execute multiple times while calculating the seams, for each seam. Therefore, we believed that this would allow us to observe the effects of branch prediction.

Testing Setup. We used StarCoder to label the branches in the seam carver code. We only added labels where StarCoder had high confidence in its prediction (difference between logits of 2 or greater). Then we ran seam carver on three images: ocean, mountain, and forest. These images were picked to represent different sizes – ocean (507×285), mountain (728×485), and forest (1300×865). Seam carver was set to carve 4 horizontal and 4 vertical seams from each image, we will henceforth refer to carving 4 horizontal and 4 vertical seams as a run. We performed 20 runs on each image for both

Epoch	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall
1	1.042800	1.001853	0.467305	0.297653	0.218374	0.467305
2	0.972900	0.972951	0.467305	0.297653	0.218374	0.467305
3	0.955000	0.961399	0.467305	0.297653	0.218374	0.467305

Table 1: Metrics from CodeBert training process

our unlabeled and labeled code. We timed each run and averaged the results.

5 RESULT ANALYSIS

Metric	Value
Evaluation Loss	0.8058
Evaluation Accuracy (%)	79.35

Table 2: Evaluation metrics for prefix-only training

Metric	Class 0	Class 1
F1 Score	0.6412	0.8550
Precision	0.6209	0.8664
Recall	0.6628	0.8439

Table 3: Class-specific metrics for prefix-only training

Metric	Value
Evaluation Loss	0.4468
Evaluation Accuracy (%)	86.35

Table 4: Evaluation metrics for prefix + suffix training

Metric	Class 0	Class 1
F1 Score	0.7551	0.9054
Precision	0.7547	0.9055
Recall	0.7555	0.9052

Table 5: Class-specific metrics for prefix + suffix training

5.1 CodeBERT Model Evaluation

CodeBERT yielded very poor results (Table 1). Although we re-ran the training process multiple times, each time it learned to predict the same label for all examples. Because StarCoder yielded much better results using the same training process, this suggests that CodeBERT is not a powerful enough model for our task. Because of CodeBERT's poor performance, we focus on StarCoder for the rest of this section.

5.2 StarCoder Model Evaluation

Tables 2 & 3 both show classification metrics that we get from training StarCoder on examples where code inside the if statement (suffix) is excluded. We can see an improvement moving towards Tables 4 & 5. The model accuracy and loss get better and so do class quantitative scores (f1, precision, and recall).

We see a significant improvement for prefix + suffix training versus prefix-only training, which suggests that even though the code inside an if statement does not logically influence its likelihood, it provides hints which help StarCoder make more accurate predictions.

5.3 Runtime Analysis

Figures 4, 5 show the difference in runtime between unlabeled and labeled data and the percentage difference in runtime between unlabeled and labeled data. The results show that there is a speedup across all three images on labeled data. The best speedup is observed in the ocean image. We suspect that this is caused by a higher similarity between seams in the ocean image, leading to more branches being predicted accurately.

We performed a single tailed unpaired t-test on data from all images to determine if the means for unlabeled and labeled data were statistically significantly different. We used an alpha of 0.05. Of the three images the only statistically significant speedups were for the large case on the forest image with a p value of 0.00021, and the small case on the ocean image with a p value of 6.89e-06.

Therefore, we can conclude that our finetuned StarCoder model is able to effectively predict [[likely]], and [[unlikely]] attributes for C++ branches resulting in a statistically significant speedup.

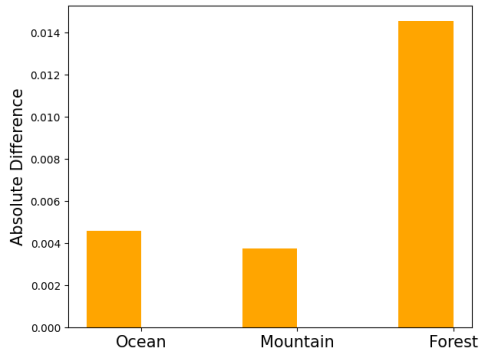

Figure 4: Absolute difference in run times for Seam Carver

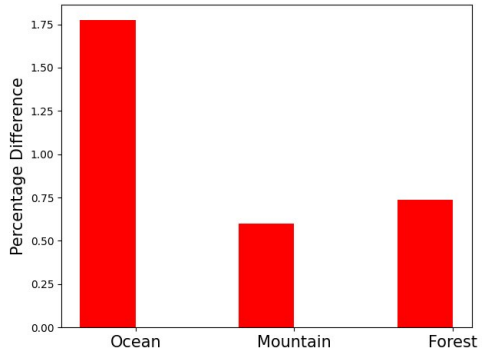

Figure 5: Percentage difference in run times for Seam Carver

6 FUTURE PLAN

A major drawback of our approach can be seen in the data. The more context given in the dataset with the prefix and suffix to the branches could help the model generalize and predict the [[likely]] and [[unlikely]] tags more easily. Additionally adding more data with different varieties of branch behaviors could help the model be more flexible to different code structures. These were all a drawback due to scraping large code files in GitHub repositories and running into token limits that limited the context we could provide during training. Finding more suitable case data could have aided in better predictions with more context.

Finally to validate the efficacy of our approach a more comprehensive large-scale bench marking method would be needed. This method would test our branch prediction methodology across diverse datasets and workloads allowing us to gauge the real-world applicability and effectiveness. By rigorously testing in large-scale scenarios we aim to establish a robust foundation for future advancements in branch prediction techniques.

More generally, the application of LLMs for other types of compiler hints, such as the C++ inline and register keywords, could be explored. These keywords face the same problem as what we seek to address about the [[likely]] and [[unlikely]] tags: poor understanding and incorrect application by developers. However, running a LLM (even the 1 billion parameter StarCoder) remains prohibitively computationally intensive on a consumer laptop or desktop. Nevertheless, the increasing adoption and advancement of ML accelerators on mobile chipsets, as well as other, unforeseen advancements in hardware or software, could make their adoption in compilers more feasible in the future.

REFERENCES

[1] cppreference.com contributors. 2024. Attributes: likely and unlikely. https://en.cppreference.com/w/cpp/language/attributes/likely. Accessed: 2024-04-01.

[2] Rinu Joseph. 2021. A Survey of Deep Learning Techniques for Dynamic Branch Prediction. *arXiv preprint arXiv:2112.14911* (2021).

[3] Anastasios Zouzias, Kleovoulos Kalaitzidis, and Boris Grot. 2021. Branch prediction as a reinforcement learning problem: Why, how and case studies. *arXiv preprint arXiv:2106.13429* (2021).