AI-Based Coverage Analysis Leveraging LLMs

Hin Kit Eric Wong
Dept. of CSE, BCOE
University of California Riverside
Riverside, USA
hwong051@ucr.edu
0000-0001-8668-0462

Harsh Vardhan Sharma
Dept. of CSE, BCOE
University of California Riverside
Riverside, USA
hshar021@ucr.edu

Harsh Patel
Dept. of CSE, BCOE
University of California Riverside
Riverside, USA
hpate064@ucr.edu

***Index Terms*—Coverage Analysis, BERT, Large Language Models, Input Coverage**

I. Introduction

Work Distribution for Team 27:
Hin Kit Eric Wong – 40%
Harsh Vardhan Sharma – 35%
Harsh Patel – 25%

II. Project Introduction

A. Project Choice

We opted to choose Option 3: AI-based Coverage Analysis. This project integrates AI into software testing by employing Bidirectional Encoder Representations from Transformers (BERT), a state-of-the-art Natural Language Processing model, to predict program branch coverage. Traditionally, ensuring comprehensive branch coverage is a meticulous manual task. By adapting BERT to understand text inputs and their corresponding impacts on code execution paths, we can automate the detection of which branches are exercised during tests. This approach not only streamlines the testing process but also enhances the accuracy and efficiency of coverage analysis, leading to more reliable and faster software deployment.

B. Project Description

The project involves training BERT to predict the edge coverage of a word frequency counter program (referred to as wf) when processing a given text file. The input to the model is a string containing text file content, and the output is a list of binary values representing whether a particular edge in the program's control flow is exercised by the input.

We fine-tuned the pre-trained transformer model, BERT, to predict the software branch coverage of a word frequency counting program, wf (provided), when processing a given text file.

The model was initially pre-trained using a massive corpus, where it learned context through tasks like next sentence prediction and masked token prediction. This foundational training enabled BERT to grasp the subtle nuances of language, setting the stage for its specialized application in software testing.

C. Experimental Settings

We operated on Google Colab so that we could work simultaneously and not have our computer specifications be a factor in the training and validation of BERT.

- Python 3 Google Compute Engine backend (GPU)
- T4 GPU while training and running the model

D. Project team folder (contains all relevant files):

Team 27-Project #3

E. Link to source code:

Team 27 Source Code - colab.research.google.com

III. Implementation Documentation

A. Reading and Preprocessing Files

In this section of the code, we define the data frame, the new_df, that contains the columns of input-text and coverage. Also, we print out the number of edges in our dataset so that we can verify. After we have preprocessed the data, we can then create our dataloader.

B. Data Preprocessing

- Data Import: A dataset named training-data.csv, containing $2,000$ text files and their corresponding edge coverages, is imported into a pandas DataFrame.
- Data Processing: The coverage data, spanning multiple columns (from ”branch1” to ”branch59”), is consolidated into a single list for each row, creating a new column named ”coverage”.
- New DataFrame Creation: A new DataFrame new _df is created with two columns: ”input-text” (the content of each text file) and ”coverage” (the corresponding edge coverage list).

C. Setting up hyperparameters(model, number of edges, training and validation batch size, learning rate)

Hyperparameters:

- model
- number of edges
- training and validation batch size
- learning rate

We have configured the setup for fine-tuning a pre-trained BERT model. The setup specifies the BERT model variant to use, the maximum length of input text sequences, and computes the number of features that will define the output layer of the model, crucial for matching the prediction task at hand. It also sets the batch sizes for training and validation, ensuring consistent data processing.It also defines the learning rate, a key hyperparameter that controls the optimization process during model training. These values are set by repeated testing as evidenced by the *testing document*.

D. Dataset loader

A class of CustomDataset loader is defined, which has 5 attributes:

- texts – contains the input text or the inputs given to the program
- targets – contains the target bitmap of input coverage for that specific input
- tokenizer – the tokenizer we use to tokenize our input (in our case, Bert's default tokenizer)
- data – the data frame we processed earlier
- max_len – the max length we want any input to be

Here we transform the raw text data into a format suitable for training and evaluating the BERT model, adhering to the required input specifications of the transformer architecture.

We initialize with three parameters: a Pandas DataFrame *(data _frame*, a BERT tokenizer *(tokenizer)*, and the maximum token length *(max _len)*. Within the class, *self.texts* extracts text inputs from the DataFrame, and *self.targets* retrieves corresponding edge coverage data, formatted as lists of binary values indicating whether certain edges in a program's control flow are exercised by the text input.

The *__len __* method returns the dataset's size, equating to the number of text entries. The *__getitem __* method is a crucial component; it processes individual data items for the model. Given an index, it extracts the specific text *(curr _text)* using *iloc*, ensuring reliable data retrieval. It then converts the associated edge coverage into a PyTorch tensor *(curr _target)*. The text is tokenized using the BERT tokenizer, ensuring consistency in input length through padding and truncation. The resulting tokenized data includes *input _ids* (the tokenized representation of text), *attention _mask* (indicating significant tokens versus padding), and *token _type _ids* (distinguishing between different sentences in cases of multiple sentences within a single input). These components are flattened and packaged into a dictionary with the target tensor, forming a complete processed sample for the model.

This setup facilitates the efficient transformation of raw text data and corresponding targets into a format suitable for training a BERT-based model, streamlining the workflow from raw data to model-ready inputs and outputs.

E. Splitting up dataset into training and validation

Now, we are going to create two subsets, one for training (to fine tune the model) and one for validation (to evaluate the performance of the model). Given that we decided on a training ratio of 0.9, our training data set contains 90% of the input data, whereas the rest is given to the validation data set.

F. Setting parameters for training and validation

Setting the parameters for both training and validation datasets involved configuring and creating data loaders in a PyTorch machine learning workflow. The *train _params* and *valid _params* dictionaries define parameters for the respective data loaders. Both dictionaries set *batch _size* to predefined values (*TRAIN _BATCH _SIZE* and *VALID _BATCH _SIZE*), indicating the number of samples to be processed in a single batch. The *shuffle* parameter is set to *True*, ensuring that the dataset is shuffled at every epoch, which is a common practice to reduce model overfitting and improve generalization. The *num _workers* parameter, set to 2, specifies the number of subprocesses to use for data loading; adjusting this number can optimize the data loading process and is subject to experimentation for efficiency.

The *DataLoader* objects, *train _data _loader* and *valid _data _loader*, are then instantiated using the *Custom-Dataset* instances (*train _dataset _for _model* and *valid _dataset_for_model*) and their corresponding parameter dictionaries. These data loaders automate the process of batching, shuffling, and loading the data into the model during training and validation phases.

G. Dropout Layer

We set the dropout_layer to 0.3. We tested out dropouts at 0.1, 0.2, 0.5, and 0.6. As evidenced in the testing document, the model's overall performance is best when a dropout layer of 0.3 is used compared to the others.

H. Training and Preparation

BERTClass is a custom neural network model for edge coverage prediction, built on PyTorch and incorporating a pre-trained BERT model. The *dropout _layer _rate* is set to 0.3, determining the dropout rate for regularization during training. In the class's initialization (*__init __* method), *self.l1* initializes the BERT model, *self.l2* adds a dropout layer for regularization, and *self.l3* is a linear layer tailored to map the BERT output to the desired number of output features (*NUM _EDGE*).

In the 'forward' method, the model processes input data through the BERT's first layer (*self.l1*). The output from this layer is then passed through the dropout layer (*self.l2*) to reduce overfitting. Finally, the linear layer (*self.l3*) processes this output to produce the final prediction. This structure allows the model to leverage BERT's powerful text processing capabilities while tailoring the output for the specific task of edge coverage prediction.

We decided to use HuberLoss (appropriate for the binary classification task) as our loss function, as it is an amalgamation of MSE and MAE.

I. Optimizer

Optimizers guide the update of weights to minimize the loss function, which can significantly influence the convergence rate and the quality of the final model. We employed two distinct optimizers: SGD and Adam. SGD, known for its simplicity and effectiveness in large-scale and sparse machine learning problems, was initially considered for its robustness. However, we leaned towards Adam due to its adaptive learning rate capabilities, which are particularly suited for problems with large parameter spaces and data sets. Adam's computation efficiency and convergence properties have proven advantageous in handling the intricate landscape of BERT's parameter optimization, contributing significantly to our model's performance.

J. Model training

In refining our model training process, we established the number of epochs—a complete pass through the entire training dataset—as a crucial hyperparameter. After conducting multiple trials with epoch values of 13, 17, 20, and 25, we observed optimal results at 20 epochs, where the model achieved the lowest loss, indicating an effective learning rate without excessive computational demand. Further increasing the epoch count yielded diminishing returns, with a negligible decrease in loss at the expense of significantly longer training times, hence solidifying our decision to set the epoch parameter to 20 for the balance of efficiency and performance.

In order to train with the epoch, Data extraction from the DataLoader yields tensors for *input _ids*, *attention _mask*, *token _type _ids*, and *targets*. These variables represent batches of data in tensor format, which is essential for the model to process. These variables are cast to torch.long (64-bit integers) to match the data type expected by BERT for index-based inputs.

- *input _ids*: integer sequences corresponding to tokenized text.
- *attention _mask*: *attention _mask* is a binary tensor indicating which tokens within *input _ids* are meaningful and which ones are padding. This mask tells the model to pay attention to the relevant tokens during processing.
- *token _type _ids*: *token _type _ids* are used to distinguish different sequences within a single input. For models that handle multiple sequences, like BERT's question answering tasks where both a question and an answer passage are provided, *token _type _ids* specify which part of the input belongs to which sequence.
- *targets*: representing the desired model output for branch coverage, *targets* are converted to torch.float32 (32-bit floating-point) to enable precise computation of gradients during the optimization process, as the loss function operates on floating-point arithmetic for its differential properties.

In the backward pass and optimization step of the training function, we engage in a sequence of operations critical for neural network weight adjustment. This sequence, with the specifics mentioned below, is repeated across epochs to iteratively refine the model's parameters towards optimal predictive performance.

- *optimizer.zero_grad()*: after the forward pass, this function clears old gradients to prevent accumulation, which could otherwise lead to incorrect updates.
- *loss.backward()*: *loss.backward()* computes the gradient of the loss function with respect to each weight.
- *optimizer.step()*: *optimizer.step()* updates the weights in the direction that minimizes the loss, using the previously calculated gradients.

K. Model Evaluation/Validation

In validation, we systematically assess the model's predictive capability on a validation dataset that has not been exposed to the model during training to ensure that our performance metrics reflect the model's ability to generalize. We replicate the data preprocessing steps used in training to prepare our validation data.

After prediction, we transform the model's continuous output into a binary format to match our target labels. This means that a threshold value is used to convert all output values above a certain threshold (set to 0.5 in our case) to 1 and all others to 0.

This binary conversion is necessary for calculating classification-specific metrics, such as F1 score, precision, and recall. These metrics are essential for understanding the model's performance, with the F1 score providing a balance between precision and recall, especially in datasets where class imbalance might be present.

IV. Results

A. Metrics used

Performance metrics, including F1 score, precision, and recall, are computed in both micro and macro averages to assess the model's overall performance. The F1 score combines precision and recall into a single metric, providing a balanced view of the model's accuracy, especially for imbalanced datasets. Micro averaging computes metrics globally by counting the total true positives, false negatives, and false positives, making it more suitable when class imbalance is present. Macro averaging computes metrics for each label and finds their unweighted mean, treating all classes equally.

The calculated values for these metrics are then printed. The micro and macro scores for F1, precision, and recall provide a comprehensive understanding of the model's performance across different aspects: overall accuracy (F1 score), the correctness of positive predictions (precision), and the model's ability to identify all positive samples (recall). This detailed evaluation helps in understanding the strengths and weaknesses of the model, particularly in how it handles different classes within the data.

B. Observations

Figure 1 illustrates our model's training loss over time. Initially, there is a sharp decline in loss, indicating that the model quickly learned from the training data. As training progresses, the reduction in loss slows down and plateaus, suggesting that the model has started to converge to its optimal parameters. The tail of the plot shows that the loss remains relatively stable, with no significant fluctuations, which often

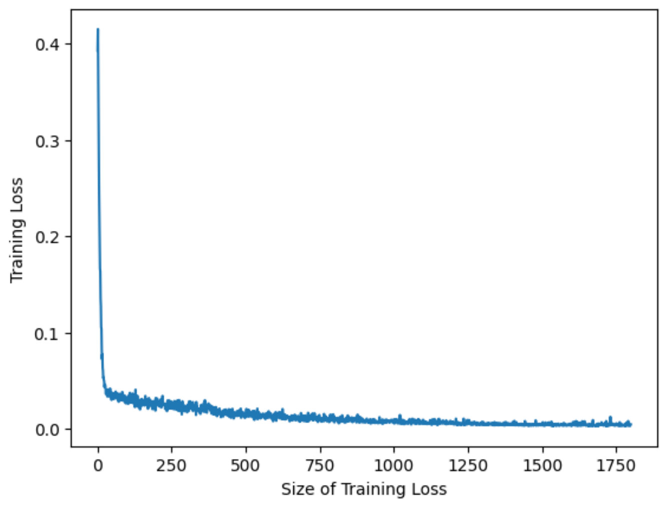

Fig. 1. Plot of the training loss over time.

signifies that further training may not yield substantial improvements and the model is close to its best performance on the training dataset. This kind of loss curve is typical and desirable as it indicates effective learning and the proper convergence behavior of the training process.

Next the testing results from our project indicate a successful implementation of the BERT model in predicting edge coverage. The model was validated with 200 samples, as indicated by the matching lengths of outputs and targets.

TABLE I
VALIDATION RESULTS

	F1 score	Precision	Recall
Micro	0.96585468511601	0.96631224339404	0.96539755994952
Macro	0.81600100209248	0.84235838854531	0.82634341238768

The evaluation metrics show impressive results, particularly with the micro-averaged scores: the F1 score (micro) is approximately 96.59%, indicating a high overall accuracy and balance between precision and recall. Precision (micro) at around 96.63% suggests that the model's positive predictions are highly accurate, and the recall (micro) of approximately 96.54% indicates the model's strong ability to identify most positive samples.

In terms of macro-averaged scores, which treat each class equally, the results are also noteworthy, though slightly lower than the micro-averaged scores, which is common in datasets with class imbalances. The F1 score (macro) is about 81.60%, precision (macro) is approximately 84.24%, and recall (macro) is around 82.63%. These scores demonstrate the model's robustness across different classes, ensuring that it performs well not just on the majority class but across various classes in the dataset.

Overall, these results signify that the model is highly effective in its task, with strong predictive performance and reliability across various aspects of the data.

C. *Challenges Encountered*

Our team grappled with fine-tuning hyperparameters, such as learning rate, epoch size, and batch size, to balance underfitting and overfitting. We aimed to improve macro scores without sacrificing micro scores, a challenging task indicating class imbalance issues. During training epochs, we had to ensure the data transformation process aligned with BERT's specific input requirements, which was not without its trials, particularly in maintaining the delicate architecture BERT demands for optimal function.

D. *Potential Optimizations*

To optimize further, we are considering implementing a more robust hyperparameter search strategy to navigate the hyperparameter space more efficiently. Additionally, exploring alternative loss functions and regularization techniques could potentially improve our model's performance on validation metrics.

REFERENCES

[1] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in 2019 IEEE Symposium on Security and Privacy (SP), May 2019, pp. 803-817.
[2] Q. Zhang, "Input coverage," presented as part of CS182, University of California Riverside, Riverside, USA, October 20, 2023.