Quantizing Large-Language Models for Predicting Flaky Tests

Shanto Rahman
University of Texas at Austin
Austin, TX, USA
shanto.rahman@utexas.edu

Abdelrahman Baz
University of Texas at Austin
Austin, TX, USA
ambaz@utexas.edu

Sasa Misailovic
University of Illinois Urbana-Champaign
Urbana, IL, USA
misailo@illinois.edu

August Shi
University of Texas at Austin
Austin, TX, USA
august@utexas.edu

***Abstract*—A major challenge in regression testing practice is the presence of flaky tests, which non-deterministically pass or fail when run on the same code. Previous research identified multiple categories of flaky tests. Prior research has also developed techniques for automatically detecting which tests are flaky or categorizing flaky tests, but these techniques generally involve repeatedly rerunning tests in various ways, making them costly to use. Although several recent approaches have utilized large-language models (LLMs) to predicting which tests are flaky or predicting flaky-test categories without needing to rerun tests, they are costly to use due to relying on a large neural network to perform feature extraction and prediction.**

We propose FlakyQ to improve the effectiveness of LLM-based flaky-test prediction by quantizing LLM's weights. The quantized LLM can extract features from test code more efficiently. To make up for loss in prediction performance due to quantization, we further train a traditional ML classifier (e.g., a random forest) to learn from the quantized LLM-extracted features and do the same prediction. The final model has similar prediction performance while running faster than the non-quantized LLM.

Our evaluation finds that FlakyQ classifiers consistently improves prediction time over the non-quantized LLM classifier, saving 25.4% in prediction time over all tests, along with a 48.4% reduction in memory usage. Furthermore, prediction performance is equal or better than the non-quantized LLM classifier.

I. Introduction

Regression testing is the common practice of rerunning tests after every change to check whether their changes introduced any bugs [1]–[3], but suffers from the presence of flaky tests. *Flaky tests* are tests that non-deterministically pass or fail when run on the same code [4]. If there are flaky tests, a developer can no longer trust that the test failures during regression testing are due to bugs introduced in code changes. Furthermore, flaky tests are prevalent in open-source software and in industry, with researchers at Facebook even suggesting that everyone should "assume all tests are flaky" [5].

There are many reasons for why tests are flaky. In a large dataset of known flaky tests [6], the flaky tests are categorized based on techniques that researchers previously developed for detecting these flaky tests automatically [7]–[10]: order-dependent (OD), non-idempotent-outcome (NIO), implementation-dependent (ID), non-deterministic order-dependent (NDOD), non-order-dependent (NOD), and unknown dependency (UD). The way researchers previously detected and categorized flaky tests generally involve repeatedly rerunning the tests in various ways [7], [8], [11], [12], which makes them costly to use.

Recent work has focused on predicting flaky tests using machine learning (ML), which does not require rerunning the tests. Alshammari et al. proposed FlakeFlagger, an ML-based approach to detect flaky tests by predicting whether a test is flaky based on features such as test-code smells, historical runtime information, or features based on running the tests once, without continuously rerunning tests [13]. Fatima et al. proposed Flakify to similarly predict flaky tests, but does so using large-language models (LLMs) and predicting based solely on the test code itself [14]. Flakify involves fine-tuning a LLM to extract features from test-code tokens (resulting in a feature vector) that it then passes on to a neural network to predict whether a test is flaky or not. Fatima et al. found that Flakify, by using a LLM, performs better at predicting flaky tests than FlakeFlagger. Akli et al. later proposed FlakyCat to predict a flaky-test category, also based on LLMs as well as with few-shot learning [15]. While we see that this recent trend of using LLMs provides benefits over traditional ML classifiers in terms of prediction performance as well as not requiring to rerun tests like dynamic analyses-based techniques, LLMs tend to be large neural networks, requiring GPUs to both fine-tune and run prediction. Running such LLM-based classifiers can be costly in terms of both runtime and memory usage.

We propose FlakyQ, an approach to train more efficient LLM-based classifiers that predict flaky tests via quantization. *Quantization* for a LLM involves converting the weights in the LLM from a higher precision data type, such as `float`, into a lower precision data type, such as `int8` [16]. A quantized LLM can extract features from test code and perform prediction faster. However, a quantized LLM generally suffers from a loss in prediction capability compared to the non-quantized LLM. We propose to overcome this limitation by additionally training a traditional ML classifier (e.g., a random forest model [17]) to learn from the features (in the form of a feature vector) that the quantized LLM extracts from test code and then performing the same prediction task (either predicting whether a test is flaky or predicting the flaky-test category). The intuition is that a traditional ML classifier

We acknowledge NSF grants no. CCF-2145774 and CCF-2313028, and the Jarmon Innovation Fund.

that uses these LLM-extracted features can enhance prediction accuracy. In the final model, which pairs the quantized LLM for feature extraction with a classifier for predictions, any accuracy loss due to quantization is rectified, achieving the same accuracy as a non-quantized LLM used for similar predictions. Furthermore, a traditional ML classifier tends to run much faster, requiring less resources than a neural network. As such, combining both the traditional ML classifier and the quantized LLM can result in faster prediction time while achieving higher prediction accuracy.

We evaluate FlakyQ on large datasets of tests that contain both flaky and non-flaky tests, and flaky tests that are labeled with flaky-test categories, collected by prior researchers [6], [15]. We fine-tune a pre-trained CodeBERT LLM on this dataset to either predict whether a test is flaky or not, or to predict the correct flaky-test category for a known flaky test. We later quantize the resulting fine-tuned LLM, converting the `float` data-type weights into `int8` data-type weights. We then train five different traditional ML classifiers using the features extracted with the quantized LLM to perform the same type of predictions. We compare the performance of the classifiers created using our approach against the original, non-quantized LLM classifier in terms of precision/recall/F1-score as well as time to perform the prediction.

We find that the FlakyQ classifiers can predict with similar precision/recall/F1-scores as the original, non-quantized LLM, sometimes even surpassing it on these metrics. Furthermore, the FlakyQ classifiers predict faster, e.g., dropping from 100.2 seconds down to 64.7 seconds to predict flaky-test categories across all labeled flaky tests in our dataset, a 35.4% reduction. We also find that the classifier uses 48.4% less memory. The comparison against training the traditional ML classifiers using non-LLM static code features (using bag-of-words or vocabulary-based features extracted from test code) shows that the traditional ML classifiers perform much worse using these other features, suggesting the need to use LLMs to achieve good prediction results. We also perform the same evaluation on different datasets of categorized flaky tests, finding similar trends between datasets, reinforcing our findings that quantization can provide prediction runtime improvements while any loss in accuracy can be masked through additional use of a traditional ML classifier.

This paper makes the following contributions:

- We propose quantizing LLMs to improve the runtime and computational resources needed to predict flaky tests and flaky-test categories, allowing them to run effectively in CPUs. Further, we recover losses in accuracy by additionally training traditional ML classifiers to learn from LLM-extracted features to perform the same predictions.
- We implement our approach and evaluate on large datasets of labeled flaky tests. We compare the classifiers trained using our approach against the non-quantized LLM and traditional ML classifiers trained using bag-of-words and vocabulary-based features, comparing the classifiers in terms of both prediction accuracy metrics and prediction runtime.
- We find that using a quantized LLM to extract features speeds up prediction time, while the additional traditional ML classifier can rectify any prediction loss. Further, classifiers trained using LLM-extracted features perform much better than using bag-of-words or vocabulary-based features, showing the necessity to use LLMs to help with prediction tasks. Our experiment scripts and results can be found at https://sites.google.com/view/flakyq.

II. Background

A. Flaky Tests

Flaky tests are tests that can nondeterministically pass and fail when run on the same version of code [4], [18]. These tests can mislead developers concerning the correctness of their code, and they are prevalent both in open-source software and in industry [5], [19], [20]. Prior work has focused on proactively detecting which tests are flaky within a test suite, such as through machine learning techniques [13], [14] or rerunning the tests in various ways, resulting in a dataset of known flaky tests known as IDoFT [6].

The flaky tests in the IDoFT dataset are also categorized based on the techniques used to detect them. Shi et al. proposed NonDex to detect tests that make assumptions on methods or data structures with nondeterministic specifications, e.g., assuming unordered sets are always iterated over in the same order due to its implementation [8]; these tests are marked as implementation-dependent (ID). Lam et al. proposed iDFlakies to detect tests whose outcomes depend on the order in which they are run (passing in one order but failing in another) [7]; these tests are marked as order-dependent (OD). iDFlakies detects flaky tests by rerunning them in different test orders, but some of the detected flaky tests are not actually OD, i.e., the tests do not consistently fail in one order and pass in another; they categorized these tests as non-order-dependent (NOD). Lam et al. later found that some of these tests are actually order-dependent, but not deterministically, i.e., they fail more often in a specific order, but not always [9]; they then marked such tests as non-deterministic order-dependent (NDOD). Wei et al. studied non-idempotent-outcome tests, which are tests that fail when run twice in the same process [10]. They marked such tests as non-idempotent-outcome (NIO). Finally, remaining flaky tests that were found through reruns, yet difficult to understand or reproduce, are marked as unknown dependency (UD).

Luo et al. previously categorized flaky tests in a different manner, based on the root causes for flakiness based on their inspection of fixed flaky tests in open-source projects [4]. Their categories include Async wait, Concurrency, Test order dependency, Time, and Unordered collections. Barbosa et al. inspected known flaky tests and manually classified them among these categories [21]. Akli et al. later refined this labeled dataset of flaky tests for their own evaluation purposes. Researchers developed techniques for debugging/reproducing flaky-test failures [22], [23] or repairing [24]–[28] flaky tests, but only for specific categories of flaky tests; knowing the category a priori can be useful for guiding which tools to use.

B. Machine Learning Classifiers and Large-Language Models

Traditional machine learning (ML) algorithms can be used to train classifiers or models that categorize some input data into a discrete set of classes [29]. In general, they take as input a training set of labeled data, training a classifier that can parse similar data to predict a label for that data. There are a wide variety of different ML algorithms readily available in libraries such as Scikit-learn [30]. Support Vector Machine (SVM) creates hyperplanes to separate instances of different classes [29], [31], [32]. Random Forest (RF) partitions the feature space into regions, assigning each region to a class [17]. K Nearest Neigbour (KNN) classifies instances based on their similarity to other instances, as measured by distance in the feature space [33]. Multi Layer Perceptron (MLP) is a type of artificial neural network that uses layers of nodes with activation functions to classify instances [34]. Logistic Regression (LR) is a probabilistic classifier that generates predictions of class membership probabilities, modeling the log-odds of the probabilities in a linear fashion [35].

Large-language models (LLMs) are neural networks trained on large quantities of text data. There are pre-trained LLMs that are specifically trained on large amounts of text from source code, e.g., CodeBERT is a LLM pre-trained on over six million lines of code across six programming languages [36], [37]. Given that these LLMs are trained on massive amounts of code data, they can provide a means to parse and process source code as well as general tasks such as predicting the next tokens, based on the large amounts of data they were already trained on. It is common to fine-tune a LLM to focus on a specific prediction task, training on labeled data to construct a better model geared towards the desired task. However, given that LLMs are built on top of neural networks, they can be expensive to train and to use for prediction, with both often requiring GPU resources.

Quantization is the process of reducing the precision of model parameters to save on memory and computation time [16], [38], [39]. For example, one can convert the data-types of weights in a model from `float` to `int8`. By using this less precise data-type, the model can run faster without using as much memory. However, the reduced computational load comes at the cost of loss in prediction accuracy. There has been much work in apply quantization for different models, including for LLMs, trying to balance between computation cost and prediction accuracy.

III. FlakyQ

We propose FlakyQ, an approach for creating an efficient LLM-based classifier that can predict whether a test is flaky or predict the flaky-test category for a known flaky test by just parsing the test-code body, with the goal that it can be run in a CPU environment. The intuition is that we can make the LLM run more efficiently through quantization. While quantization can make a model that runs faster and consumes less memory, the model may have less accurate predictions. FlakyQ further trains of a traditional ML classifier to rectify prediction loss. This traditional ML classifier is trained to take as input the features extracted using the quantized LLM to perform the same prediction task.

We first fine-tune a pre-trained LLM using a given dataset of labeled tests so it can learn how to extract features from test code for use in a specific prediction task. We specifically fine-tune the CodeBERT model [36], [37]. We follow a similar process to fine-tune the LLM as Flakify [14], with some adjustments. Figure 1 shows our overall fine-tuning process.

1) Data Processing: Given a labeled dataset for training, we first partition it into two distinct sets of training and validation (Figure 1, ❶). We have a validation set to evaluate validation loss as we fine-tune the model across several epochs. We make this division early as to ensure no overlap or mixing between the sets. Each of these sets contains the tests' source code along with their respective labels based on the prediction task (flaky or non-flaky when predicting for flaky tests, and flaky-test category when predicting for category). We utilize the tokenizer from the pre-trained CodeBERT model to tokenize the code from each test. After obtaining these tokens, we convert these tokens into tensors (multi-dimensional arrays). We create a sequence tensor and an attention mask for each test. A sequence tensor contains the numerical representations of the token, and the attention mask indicates which positions in the corresponding sequence tensor should get importance and which should be ignored. It is noteworthy that during tokenization if the token size becomes less than our defined token size, then the model adds padding to keep the same token length for all the test code. If a token is padded, then the corresponding index value in the attention mask is set to 0, otherwise it is 1. These tensors therefore represent the input test-code data that are the inputs for the model. By default, CodeBERT accommodates up to 512 tokens, truncating any additional tokens. From our evaluation, we find that 13% of tests contain more than 512 tokens. Indeed, Fatima et al. similarly noted they encountered this limitation when developing Flakify to predict flaky tests when fine-tuning a CodeBERT model [14]. To address this limitation, we segment the complete token list into smaller chunks for processing (Figure 1, ❷), where each chunk has 512 tokens (padded if less than that many tokens).

2) Fine-Tuning: For each chunk set, we utilize one CodeBERT model, as a single CodeBERT model can process a maximum of 512 tokens simultaneously. Therefore, we employ as many models as there are chunks. After creating chunks of tokens taken from test code, we feed them into the pre-trained CodeBERT models, which convert them into vectors of length 768, a CodeBERT design choice [37]. We concatenate all vectors together to create a new 768-length vector (Figure 1, ❸). To further refine our model, we introduce two additional neural network layers. The initial layer, a fully connected one, takes the 768-length vector and outputs a feature vector of length 512. This layer integrates a Rectified Linear Unit (ReLU) activation function [40], followed by a dropout layer set at a rate of 0.2.

The subsequent fully connected prediction layer uses the 512-length vector output of the preceding layer as its input.

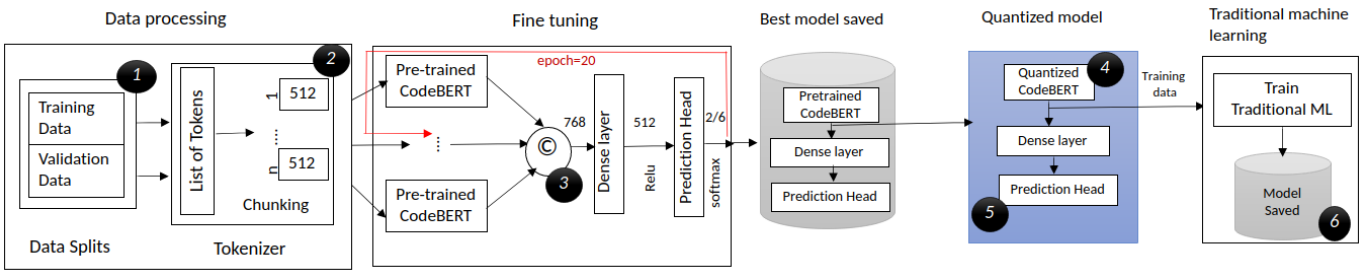

Fig. 1. FlakyQ training process.

This prediction layer's output corresponds to the specific prediction task. For instance, in the context of predicting whether a test is flaky, there are two output units: "flaky" and "non-flaky". In the context of predicting a flaky-test category, the output units match the number of categories, with each unit signifying the likelihood of a test fitting that particular category. The final step involves applying the log-softmax function [41] to generate the logits [42] corresponding to these categories. Using these logits, we compute the training loss with a class-weighted cross-entropy loss function. The class weights are determined by the number of samples in each category, addressing the issue of class imbalance.

We continue this process for 20 epochs. In each epoch, we train the model using the training set and validate on the validation set determined previously (Section III-1). We measure both the training loss and validation loss of the model. We give the model-weights of this new model as input for the next epoch to consult as it trains the model again. During these 20 epochs, we use the AdamW optimizer [43] to optimize the training loss. After 20 epochs, we save the model that resulted in the lowest validation loss.

3) Quantization: After fine-tuning a CodeBERT model, we dynamically quantize this model [44], creating a quantized LLM (Figure 1, ❹). This dynamic quantization process converts the `float` model weights to type `int8`, offering the distinct advantage of eliminating the need for model retraining, unlike static quantization [45]. By converting the model weights from `float` to `int8`, and specifically targeting the linear layers of the CodeBERT model that extracts features, we can improve feature-extraction time, particularly as it requires smaller data transfers from memory and runs on hardware designed for `int8` operations.

4) Rectifying Prediction: While we can continue to use the same prediction head created when fine-tuning the LLM to now predict using the features extracted by the quantized LLM (Figure 1, ❺), the prediction may now no longer be as effective due to the losses in model weight precision. To rectify any prediction accuracy loss, we additionally train a traditional ML classifier (e.g., random forest) to perform prediction based on the features extracted using the quantized LLM.

We train this traditional ML classifier normally to predict whether a test is flaky or to predict a flaky-test category using the entire training dataset. The inputs to the classifiers are the 768-length vectors outputted by the quantized, fine-tuned CodeBERT model. We finally output a classifier, that ultimately uses the quantized LLM to extract features and leverage the traditional ML classifier to do the actual prediction.

Fig. 2. FlakyQ classifier workflow.

Figure 2 illustrates the flow of the final outputted classifiers (Figure 1, ❻), showcasing how a developer would use one of these classifiers. The input is test-code data, which is parsed through a tokenizer to create tokens that are then converted into tensors. This tensor is fed through a quantized CodeBERT model, which extracts features from the tensor and outputs another vector that is used as input to a trained traditional ML classifier to perform the prediction.

IV. Experimental Setup

We address the following research questions:

- **RQ1**: How accurate are FlakyQ classifiers at predicting flaky tests and flaky-test categories?
- **RQ2**: How much time and memory is saved by using FlakyQ classifiers?
- **RQ3**: How effective are the classifiers when evaluated in a per-project evaluation?
- **RQ4**: How effective are the classifiers when trained/evaluated on different datasets?

We address RQ1 to evaluate the loss in prediction accuracy due to quantization as well as how well traditional ML classifiers can rectify that loss. We address RQ2 to compare the amount of time and memory needed by each classifier, i.e., how much time can be saved by using a quantized LLM. A developer may need to run the classifier many times on future code and tests, so a more efficient prediction time can reduce their development costs. We address RQ3 to check whether the prediction results are also applicable when we train on tests from some open-source projects but predict on tests from entirely different projects. This evaluation is focused on a real-world application of these classifiers, namely can a developer effectively reuse the classifiers trained on some other projects for use in their own project that was not part of the training.

TABLE I
ALL CATEGORIES ACROSS TWO DATASETS

Category-Name	#Tests
IDoFT Flaky-vs-NonFlaky	
Flaky	3195
NonFlaky	618
Total	**3813**
IDoFT Flaky Test Category	
NDOD	84
NOD	226
OD	932
NIO	196
ID	1617
UD	140
Total	**3195**
FlakyCat	
Async wait (Asyn.)	125
Concurrency (Conc.)	48
Time	42
Test Order Dependency (OD)	103
Unordered Collections (UC)	51
Total	**369**

Finally, we address RQ4 to see whether our findings also hold on some other dataset of flaky tests.

A. Dataset

We evaluate on a dataset of labeled flaky tests. We start with the dataset that Fatima et al. used to evaluate Flakify [14]. Their dataset includes both flaky and non-flaky tests. We need flaky tests that are also categorized and labeled, given that one of our prediction tasks is to predict a flaky-test category for a known flaky test. A large part of the flaky tests from Fatima et al.'s dataset are taken from IDoFT [6], a public dataset of known flaky tests where each flaky test is labeled with a category based on a prior technique used specifically to detect the flaky test [7]–[10]: order-dependent (OD), non-idempotent-outcome (NIO), implementation-dependent (ID), non-deterministic order-dependent (NDOD), non-order-dependent (NOD), and unknown dependency (UD). Ultimately we use a total of 3813 tests from Fatima et al., where 3195 of them are flaky tests and the rest are non-flaky tests. The 3195 flaky tests are also labeled using categories taken from IDoFT. Table I shows the breakdown of the number of flaky tests among each of these categories within the dataset.

For the sake of RQ4, we also use a different labeled set of flaky tests provided by Akli et al. for their work on FlakyCat [15]. FlakyCat is a technique for categorizing flaky tests, where their categories are based on flaky-test root causes defined by past studies [4], [18]: Async wait, Test order dependency, Unordered collections, Concurrency, and Time. Ultimately, we use a total of 369 labeled flaky tests from this dataset. Table I also shows the breakdown of the number of flaky tests among each of these categories within the dataset.

B. Trained Classifiers

We use FlakyQ to train five different classifiers, each using a different traditional ML classifier that uses the quantized, fine-tuned LLM's extracted features to predict flaky tests. The five traditional ML classifiers we use are KNN, MLP, RF, SVM, and LR (Section II-B).

For comparison purposes, we also evaluate the neural network classifier that uses the fine-tuned LLM for extracting features. This neural network classifier is essentially Flakify [14], except we address its token-length limitation (Section III-1). Additionally, we also train a version of it that can predict the flaky-test category. We refer to such a classifier as Flakify++. We also evaluate this neural network classifier that uses the quantized LLM to extract features, as a means to show whether the additional traditional ML classifiers trained on top of the quantized LLM can rectify any loss in prediction accuracy. We refer to such a classifier as Q-Flakify++.

In addition, we also evaluate the traditional ML classifiers using non-LLM features The goal is to see whether the traditional ML classifiers' prediction results are mainly due to using the LLM-extracted features or whether any form of representing test code is sufficient to create a good classifier. Here, we use bag-of-words and vocabulary-based features.

A bag-of-words is a vector that represents the frequency of word appearance in text while also ignoring any information concerning the order of words that appear. We use a CountVectorizer from the Scikit-learn library [30] to transform the test source code into a token count vector, and we use this vector as the extracted features for training the ML classifier.

We follow a similar approach as Pinto et al. for extracting features in a vocabulary-based approach [46]. We start by tokenizing the code using a word tokenizer [47]. We then apply stemming and calculate the appearance of each token, resulting in a token-occurrence vector for each test. We use this vector as the extracted features for training the ML classifier.

C. Cross Validation

We use a 10-fold cross validation to evaluate the effectiveness of the different classifiers. We construct 10 folds, where in each fold we divide the dataset such that 90% of the tests are used for the training set and the remaining 10% are used for test set. For each fold, we train a classifier on the training set of that fold and then evaluate on the test set (recall that this training set of each fold gets further divided into training and validation set when fine-tuning the model across many epochs, Section III-1). We also use the exact same training set and validation set on each fold for each classifier as to more fairly compare them to one another.

For the task of predicting whether a test is flaky or not, we consider the classifier predicting a test to be flaky as a positive while predicting it as non-flaky to be a negative, allowing us to compute the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). We then compute the precision of the classifier on the data in a fold as $\frac{TP}{TP+FP}$. We compute the recall of the classifier as $\frac{TP}{TP+FN}$. The F1-score is then computed as $2 \times \frac{Precision \times Recall}{Precision+Recall}$. We

compute the averages of these three values across all 10 folds and report these results for answering RQ1.

For the task of predicting the flaky-test category, we evaluate on the known flaky tests with labeled categories, and we separately compute precision, recall, and F1-score for each category, where a positive is that the classifier categorizes the test within the category and a negative as otherwise. We similarly compute an average of each value across all 10 folds per category. Furthermore, for each classifier, we compute the average of the precision, recall, and F1-scores across all categories, resulting in an average for all three values per classifier for comparison purposes.

D. Per-Project Evaluation

To answer RQ3, we break apart our dataset based on the projects the tests belong to. Like the per-project evaluation methodology proposed by Fatima et al. [14], instead of performing a 10-fold cross validation for each classifier, we instead reserve the tests from one project for use as the test set while using the tests from all the other projects for training. The goal is to check how good the classifiers are at predicting flakiness or categories for tests in projects not seen during training. Some projects have very few tests in our dataset, meaning validation on just those few tests can result in misleading results. As such, we purposefully only choose projects that have at least 30 flaky tests, resulting in 27 projects. As such, for each classifier, we train and validate 27 times (essentially performing 27-fold validation), and we compute the precision, recall, and F1-score that each classifier achieves when validating on each individual project.

E. Time Measurements

For each classifier, we measure the time required to extract features from the quantized model using the testing data. Subsequently, we calculate the time each classifier needs to predict labels for all tests within each fold, then average these prediction times over all folds. These times together form the overall prediction time. The results, including the training and quantization times, are detailed in RQ2.

F. Hardware Environment

We fine-tune the CodeBERT model using a Linux machine equipped with a single NVIDIA RTX A5000 GPU and 125GB RAM. We use CUDA version 12.0 with the GPU. We quantize the LLM using Pytorch's built-in quantizer, which can only be run on a CPU. When we train the traditional ML classifier, we run in a CPU environment without a GPU, namely on a 64-bit Ubuntu 20.04.4 desktop with an Intel(R) Xeon(R) W-2245 CPU @3.90GHz. We use the same environment for training the traditional ML classifier when using both LLM-extracted features, bag-of-words features, and vocabulary-based features. When we finally measure prediction time per classifier, we run the classifier (including feature-extraction using the LLM) in the same CPU environment.

V. Evaluation

A. RQ1: Predicting Flaky Tests

Table II presents how effective the classifiers are at predicting flaky tests. Each row corresponds to a different classifier, and each classifier trained using FlakyQ includes the name of the traditional ML classifier. Columns under "Flaky-vs-NonFlaky" show the results of the classifiers trained to predict whether a test is flaky, and columns under "Flaky-Test Category" show the results of the classifiers trained to predict a flaky-test category. The columns "P", "R", and "F1" show precision, recall, and F1-score, respectively.

Overall, the neural network classifier Flakify++ has high precision, recall, and F1-score for all prediction tasks, with on average 93.9 F1-score to predict whether a test is flaky and on average 91.2 F1-score to predict a flaky-test category. Using the quantized LLM decreases the classifier's prediction effectiveness, with F1-score dropping to 93.0 when predicting whether a test is flaky or not and F1-score dropping to 84.3 when predicting the flaky-test category.

However, the classifiers created using FlakyQ rectify the loss that comes from quantization. All FlakyQ classifiers have average F1-scores that are higher than Q-Flakify++, and they can even be higher than those achieved by Flakify++, which does not use the quantized model. For example, FlakyQ_MLP achieves the highest average F1-score for predicting whether a test is flaky, at 94.4, while FlakyQ_LR achieves the highest average F1-score for predicting flaky-test categories, at 93.3.

We also show in Table III the average F1-scores across all folds for each classifier, but broken down across the six flaky-test categories. Different classifiers are better or worse at predicting different flaky-test categories. A priori, we cannot know which classifier would be more effective for specific categories. However, we see the same trend overall, where Q-Flakify++, due to using a quantized LLM, is worse at predictions for almost all categories compared to the non-quantized version in Flakify++. The FlakyQ classifiers can result in F1-scores that rectify that loss, with comparable (often higher) F1-scores than Flakify++.

Table IV shows the same prediction results as in Table II, but for the traditional ML classifiers trained using bag-of-words and vocabulary-based features. In comparison to their corresponding versions that use the quantized LLM, the precision, recall, and F1-scores are all lower. While F1-scores are not lower by much in the case of predicting whether a test is flaky or not, the F1-score is especially low when predicting the flaky-test category. For example, KNN, when trained using LLM features, can achieve an average F1-score of 92.1 for predicting flaky-test categories, but that F1-score drops to 69.1 when using vocabulary-based features. Most other classifiers' F1-scores drop around 10 percentage points when predicting flaky-test categories using such features. Overall, these results highlight how useful LLM-extracted features are towards these prediction tasks, where the traditional ML classifiers can now predict at the same level as the neural network classifier if they are trained to use the same LLM-extracted features.

TABLE II
RESULTS OF CLASSIFIERS PREDICTING FLAKY TESTS.

Classifier	Flaky-vs-NonFlaky Precision (P)	Recall (R)	F1-score (F1)	PT (sec)	Flaky-Test Category Precision (P)	Recall (R)	F1-score (F1)	PT (sec)
Flakify++	94.1	93.9	93.9	123.0	92.6	90.8	91.2	100.2
Q-Flakify++	93.0	93.2	93.0	109.6	88.4	83.9	84.3	96.5
FlakyQ_KNN	93.4	93.6	93.5	78.7	92.4	92.0	92.1	66.9
FlakyQ_MLP	94.4	94.5	94.4	78.3	92.9	92.6	92.6	64.2
FlakyQ_RF	93.2	93.2	93.1	75.9	93.0	92.8	92.6	64.7
FlakyQ_SVM	93.6	93.6	93.5	76.0	93.3	93.1	93.0	65.2
FlakyQ_LR	93.6	93.8	93.6	78.7	93.5	93.3	93.3	65.8

TABLE III
RESULTS OF CLASSIFIERS FOR FLAKY-TEST CATEGORY (THIS TABLE CONTAINS F1-SCORE).

Classifier	NDOD	NOD	OD	NIO	ID	UD
Flakify++	84.9	71.9	91.8	94.6	92.4	81.4
Q-Flakify++	86.7	64.5	84.7	86.7	91.7	72.9
FlakyQ_KNN	85.9	72.6	92.2	95.8	92.5	82.0
FlakyQ_MLP	86.5	75.1	92.9	96.2	91.9	80.6
FlakyQ_RF	87.8	75.2	92.9	96.2	91.6	82.6
FlakyQ_SVM	85.1	77.9	93.6	96.6	91.0	81.9
FlakyQ_LR	86.1	77.4	93.3	96.8	92.1	82.3

TABLE IV
RESULTS OF PREDICTING TEST FLAKINESS USING BAG-OF-WORDS AND VOCABULARY-BASED FEATURES.

Classifier	Flaky-vs-NonFlaky P	R	F1	PT (sec)	Flaky-Test Category P	R	F1	PT (sec)
Bag-Of-Words Features								
KNN	85.8	95.8	90.5	1.7	74.6	73.3	73.0	0.6
MLP	89.4	90.8	90.1	0.3	85.9	85.3	85.3	0.2
RF	90.0	93.8	91.8	0.6	84.4	83.2	82.3	0.4
SVM	92.8	91.6	92.2	3.1	82.5	82.8	82.4	4.6
LR	91.2	92.7	92.0	0.2	83.8	84.1	83.6	0.2
Vocabulary Features								
KNN	85.3	96.3	90.4	3.6	70.5	69.5	69.1	4.1
MLP	92.9	79.0	85.4	2.5	85.9	85.7	85.4	3.3
RF	89.7	91.5	90.6	2.5	83.6	82.3	81.1	3.4
SVM	89.5	93.4	91.4	2.6	83.0	83.2	82.9	9.0
LR	93.2	91.5	92.3	4.6	84.5	84.8	84.2	3.2

B. RQ2: Time and Memory

Table II also shows for each classifier the average prediction time (column "PT (sec)"), namely the number of seconds needed for the classifier to run prediction on all tests in a fold. Note that this prediction time includes the time to extract features as well as the time to do the actual prediction using the extracted features as input.

Flakify++ takes the most time to do prediction, on average needing 123.0 seconds for predicting whether a test is flaky and 100.2 for predicting flaky-test categories. By using a quantized LLM, the time drops down to 109.6 seconds and 96.5 seconds, respectively. The prediction time drops even further for the FlakyQ classifiers, which can indicate how the final prediction can be sped up when using a traditional ML classifier over the neural network.

We observe that the times needed for training all classifiers are similar to each other, needing around 2263.6 seconds for training, the bulk of which is fine-tuning the LLM. Quantization takes an additional 3.3 seconds. We also find that the additional time needed to train the traditional ML classifier on top of the quantized LLM is rather small compared to the rest of the time. The training time for these traditional ML classifiers is quite short, with durations ranging from as little as 0.0034 seconds (for KNN) up to 20.5 seconds (for RF). While training time is rather large, note that training a classifier needs to happen just once, and a developer can reuse the classifier for predicting flaky tests in other projects.

Table IV also shows the prediction time for the classifiers trained using bag-of-words and vocabulary-based features. The prediction time is much faster, due to not using a LLM to extract features. While these classifiers take much less time to run, given that they are not as effective at prediction, a developer would have to consider whether this trade-off between prediction accuracy and prediction time is worth it for their specific development process.

Overall, these results showcase the efficiency of FlakyQ classifiers, given that they can achieve the similar prediction accuracy as Flakify++ while running much faster. For ease of presentation, we only show results for classifiers that use LLM-extracted features for later RQs.

Futhermore, we find that using the quantized LLM also provides reduced memory usage. Our results show that classifiers that use the quantized LLM use 48.4% less memory for all prediction tasks. This large reduction in memory usage can be especially beneficial for larger-scale applications.

C. RQ3: Per-Project Evaluation

Table V shows the results of the per-project evaluation for the task of predicting whether a test is flaky, for the 27 projects that have at least 30 flaky tests. Each row shows the results for each project. The column "Support" shows the number of tests on which we run the classifier. Columns "P", "R", "F1", and "PT (sec)" show the precision, recall, F1-score, and prediction time, respectively, of each classifier for the project. For space reasons, we only show the results for Flakify++, Q-Flakify++, and FlakyQ_RF as a representative of the FlakyQ classifiers. We show in the final row the summary of the results, namely the total number of tests on which we run the classifiers, the

TABLE V
PER-PROJECT ACCURACY OF FLAKY VS NON-FLAKY (EACH PROJECT HAVE AT LEAST 30 FLAKY TESTS).

Project	Support	Flakify++				Q-Flakify++				FlakyQ_RF			
		P	R	F1	PT (sec)	P	R	F1	PT (sec)	P	R	F1	PT (sec)
Chronicle-Wire	63	93.0	87.0	90.0	22.6	94.0	95.0	95.0	20.2	94.0	94.0	94.0	17.6
DataflowTemplates	39	100.0	100.0	100.0	13.9	100.0	100.0	100.0	12.8	100.0	100.0	100.0	7.9
Java-WebSocket	54	100.0	100.0	100.0	18.8	100.0	100.0	100.0	16.9	100.0	100.0	100.0	10.8
Mapper	76	97.0	99.0	98.0	26.3	97.0	99.0	98.0	23.8	97.0	99.0	98.0	15.4
admiral	113	99.0	99.0	99.0	38.6	99.0	99.0	99.0	33.6	99.0	99.0	99.0	22.0
adyen-java-api-library	89	68.0	54.0	43.0	30.6	55.0	54.0	52.0	27.3	64.0	54.0	45.0	18.2
biojava	52	100.0	100.0	100.0	18.2	100.0	100.0	100.0	16.3	96.0	98.0	97.0	10.7
dubbo	186	87.0	88.0	87.0	61.7	84.0	88.0	86.0	55.4	84.0	91.0	88.0	37.5
esper	38	100.0	100.0	100.0	13.4	100.0	100.0	100.0	12.9	100.0	100.0	100.0	7.6
fastjson	109	92.0	91.0	91.0	37.1	93.0	93.0	93.0	34.7	93.0	93.0	93.0	22.1
hadoop	149	99.0	99.0	99.0	50.3	100.0	100.0	100.0	45.7	100.0	100.0	100.0	30.5
hbase	52	99.0	98.0	98.0	18.2	99.0	98.0	98.0	17.3	99.0	98.0	98.0	10.7
hive	42	99.0	98.0	98.0	14.9	99.0	98.0	98.0	14.1	99.0	98.0	98.0	8.9
innodb-java-reader	45	100.0	100.0	100.0	15.7	100.0	100.0	100.0	14.9	100.0	100.0	100.0	9.0
junit-quickcheck	250	99.0	99.0	99.0	80.7	99.0	99.0	99.0	75.1	99.0	99.0	99.0	50.1
mockserver	39	100.0	100.0	100.0	14.0	100.0	100.0	100.0	13.7	100.0	100.0	100.0	8.2
nacos	34	94.0	94.0	94.0	12.1	89.0	94.0	91.0	11.8	89.0	94.0	91.0	6.9
nifi	146	99.0	99.0	99.0	49.4	100.0	100.0	100.0	45.8	100.0	100.0	100.0	29.7
openhtmltopdf	35	100.0	100.0	100.0	12.6	100.0	100.0	100.0	11.9	100.0	100.0	100.0	7.2
ormlite-core	114	100.0	100.0	100.0	38.9	100.0	100.0	100.0	35.7	100.0	100.0	100.0	23.0
riptide	30	100.0	100.0	100.0	10.7	100.0	100.0	100.0	10.1	100.0	100.0	100.0	6.1
spring-boot	48	100.0	100.0	100.0	16.9	100.0	100.0	100.0	15.6	100.0	100.0	100.0	9.7
spring-data-r2dbc	68	100.0	100.0	100.0	23.6	100.0	100.0	100.0	21.8	100.0	100.0	100.0	13.8
spring-hateoas	42	100.0	100.0	100.0	14.9	100.0	100.0	100.0	13.8	100.0	100.0	100.0	8.6
typescript-generator	60	100.0	100.0	100.0	20.9	100.0	100.0	100.0	19.3	100.0	100.0	100.0	12.2
visualee	47	100.0	100.0	100.0	16.6	100.0	100.0	100.0	15.3	100.0	100.0	100.0	9.7
wildfly	85	100.0	100.0	100.0	29.3	98.0	99.0	98.0	26.9	98.0	99.0	98.0	17.3
Total/ Weighted Avg.	2105	96.3	95.6	95.1	721.0	95.6	96.0	**95.6**	662.4	95.8	96.2	95.4	**431.5**

average precision, recall, and F1-score across all projects, and the total prediction time across all tests.

Overall, we see that the classifiers in general are effective at predicting whether a test is flaky, even when predicting on tests from projects not seen during training. We see that the differences in prediction is not that much different between using quantized and non-quantized LLMs, with even a slight increase in recall in using a quantized model, leading to a higher F1-score of 95.6 for Q-Flakify++. When using FlakyQ_RF, the final F1-score is similar to that of Q-Flakify++. These results suggest that, when training classifiers to predict whether a test is flaky by using data from other projects, quantization does not change much the prediction accuracy. However, we once again see a substantial speedup from using quantized models, and the prediction time for FlakyQ_RF ends up much faster than for Flakify++ and Q-Flakify++.

When considering the task of predicting the flaky-test category, the results in a per-project evaluation are a bit different. Table VI is a similar table as before, but shows results for predicting the category. We see here that the average F1-score is worse with quantization when comparing Flakify++ to Q-Flakify++, with the score dropping from 92.0 down to 89.7. However, FlakyQ_RF now makes up for that loss in prediction accuracy, with an average F1-score of 92.4. Once again, quantization saves on prediction time, with the total prediction time being 366.9 seconds for FlakyQ_RF across all tests, in comparison to 633.8 seconds for Flakify++.

D. RQ4: Evaluating on FlakyCat Dataset

Table VII shows the results of running the different classifiers but trained and evaluated on the FlakyCat dataset. We show in the table the average F1-score achieved by each classifier when categorizing the flaky tests into one of the five categories defined in the FlakyCat dataset, namely Async wait (Asyn.), Concurrency (Conc.), Time, Unordered collections (UC), Test order dependency (OD). The column "Weighted Avg." shows the average F1-score across all categories. The final column "PT (sec)" shows the prediction time of the classifier when run across all tests in this dataset.

Flakify++ achieves high F1-scores across all categories, with an average F1-score of 95.6, showing how effective LLMs are at predicting flaky-test categories even for a different dataset of flaky tests and flaky-test categories. Q-Flakify++'s F1-score once again drops due to using a quantized LLM, down to 93.6. The prediction time for Q-Flakify++ is lower than for Flakify++, at 11.3 seconds vs 14.0 seconds. We once again see that the FlakyQ classifiers almost all rectify the prediction loss. Furthermore, the FlakyQ classifiers have comparable prediction time as Q-Flakify++.

We also evaluate FlakyCat itself on this same dataset. FlakyCat also uses a pre-trained CodeBERT model, but it relies on few-shot learning (FSL) for training to do the prediction. We show the results for FSL in the table as well. The prediction results for FSL are not as high, similar to the findings in the original work [15]. We note that the main difference comes

TABLE VI
PER-PROJECT ACCURACY OF FLAKY-TEST CATEGORY (THE PROJECTS THAT HAVE AT LEAST 30 TESTS).

Project	Support	Flakify++				Q-Flakify++				FlakyQ_RF			
		P	R	F1	PT (sec)	P	R	F1	PT (sec)	P	R	F1	PT (sec)
Chronicle-Wire	59	93.0	85.0	89.0	21.3	92.0	63.0	74.0	20.8	93.0	86.0	90.0	13.5
DataflowTemplates	39	100.0	100.0	100.0	13.8	100.0	90.0	95.0	13.5	100.0	100.0	100.0	7.5
Java-WebSocket	54	88.0	87.0	87.0	19.1	88.0	87.0	87.0	18.2	88.0	87.0	87.0	11.0
Mapper	75	95.0	95.0	93.0	26.2	86.0	75.0	80.0	24.7	99.0	99.0	99.0	15.0
admiral	109	90.0	87.0	85.0	37.8	77.0	80.0	77.0	34.5	91.0	91.0	88.0	22.1
adyen-java-api-library	45	100.0	100.0	100.0	16.1	100.0	100.0	100.0	15.1	100.0	100.0	100.0	8.8
biojava	51	24.0	16.0	19.0	18.2	81.0	10.0	16.0	17.0	28.0	37.0	32.0	10.4
dubbo	170	79.0	77.0	77.0	57.6	81.0	78.0	77.0	50.7	80.0	73.0	73.0	34.8
esper	38	97.0	97.0	97.0	13.6	97.0	97.0	97.0	12.6	95.0	97.0	96.0	7.7
fastjson	64	91.0	92.0	91.0	22.5	89.0	88.0	88.0	20.9	92.0	95.0	94.0	12.8
hadoop	146	91.0	89.0	90.0	49.9	92.0	86.0	88.0	44.0	92.0	90.0	91.0	29.5
hbase	47	98.0	98.0	98.0	16.8	94.0	96.0	95.0	15.6	98.0	98.0	98.0	9.7
hive	41	98.0	98.0	98.0	14.9	98.0	95.0	96.0	13.5	98.0	98.0	98.0	8.6
innodb-java-reader	45	100.0	100.0	100.0	16.1	100.0	100.0	100.0	14.6	100.0	100.0	100.0	9.0
junit-quickcheck	131	97.0	98.0	98.0	44.8	97.0	98.0	98.0	39.4	97.0	98.0	98.0	26.1
mockserver	30	100.0	100.0	100.0	10.9	100.0	100.0	100.0	10.0	100.0	100.0	100.0	6.2
nacos	32	100.0	100.0	100.0	11.6	97.0	97.0	97.0	10.6	97.0	97.0	97.0	6.6
nifi	139	100.0	100.0	100.0	47.5	100.0	99.0	100.0	42.1	100.0	100.0	100.0	28.2
openhtmltopdf	35	100.0	100.0	100.0	12.6	100.0	100.0	100.0	11.5	100.0	100.0	100.0	7.2
ormlite-core	113	99.0	99.0	99.0	39.1	97.0	97.0	97.0	39.1	97.0	97.0	97.0	23.2
riptide	30	100.0	100.0	100.0	10.9	100.0	100.0	100.0	9.7	100.0	100.0	100.0	6.1
spring-boot	48	100.0	100.0	100.0	17.1	100.0	100.0	100.0	15.0	98.0	98.0	98.0	9.6
spring-data-r2dbc	37	100.0	100.0	100.0	13.4	100.0	100.0	100.0	11.6	100.0	100.0	100.0	7.5
spring-hateoas	41	100.0	100.0	100.0	14.7	100.0	100.0	100.0	12.9	100.0	100.0	100.0	8.3
typescript-generator	60	100.0	100.0	100.0	21.2	100.0	100.0	100.0	18.6	100.0	100.0	100.0	12.1
visualee	47	100.0	100.0	100.0	16.8	100.0	100.0	100.0	14.7	100.0	100.0	100.0	9.1
wildfly	84	98.0	96.0	97.0	29.4	98.0	96.0	97.0	25.3	98.0	98.0	98.0	16.2
Total/ Weighted Avg.	1810	92.9	91.9	92.0	633.8	93.2	88.9	89.7	576.2	93.2	92.6	**92.4**	**366.9**

TABLE VII
PER-CLASSIFIER EVALUATION WITH FLAKYCAT DATASET.

Classifier	Asyn.	Conc.	Time	UC	OD	Weighted Avg.	PT (sec)
Flakify++	94.80	93.31	96.86	96.08	97.09	95.6	14.0
Q-Flakify++	92.62	87.06	96.86	95.02	95.83	93.6	11.3
FlakyQ_KNN	93.14	90.71	95.52	95.02	96.31	94.2	7.8
FlakyQ_MLP	93.97	89.67	95.52	94.82	96.60	94.5	7.8
FlakyQ_RF	94.28	91.54	95.52	94.82	96.60	94.8	7.6
FlakyQ_SVM	93.76	89.04	95.52	93.16	96.12	93.9	7.7
FlakyQ_LR	92.72	89.77	95.52	94.82	96.12	93.9	7.5
FSL	72.00	36.00	75.00	72.00	73.00	67.5	10.6
FSL++	93.66	90.29	97.90	95.88	96.66	91.5	10.6

from the model not being fine-tuned for the prediction task, as they directly use the existing pre-trained CodeBERT model. We enhance their technique to fine-tune the CodeBERT model, similar to how we do for our own approach, to create a better classifier; we show the results for this classifier as FSL++. The F1-scores are comparable to the ones achieved by the other classifiers. These results show the importance of fine-tuning the model for better prediction results.

VI. THREATS TO VALIDITY

To mitigate bias in results, we use 10-fold cross validation and a per-project validation that treats each project's tests as a fold. For fairer comparison between different classifiers, we use the same training set and validation set for each fold across the classifiers.

The dataset we use contains an unbalanced set of tests labeled as flaky and non-flaky, with most tests being labeled as flaky. It can thus bias the true/false positives and true/false negatives during prediction. However, given the strong need for ground truth data for this type of prediction task, we need to use tests that are labeled correctly as flaky or non-flaky, i.e., we cannot just assume all tests that are not labeled as flaky are truly non-flaky. We use the same dataset used in prior work [14], where the authors manually labeled the dataset with this ground truth through inspection.

The traditional ML classifiers have many tunable hyper-parameters, and we use only the recommended defaults for parameters. The performance of each classifier may actually be better if they are better tuned. Our overall results showing that the traditional ML classifiers, when trained using the LLM-extracted features, still results in predictions quite comparable to the original LLM-based neural network classifier. Our conclusions would still be valid if the traditional ML classifiers' results are better after tuning their parameters.

For our study, we choose to use CodeBERT as the LLM for extracting features, as to match prior work in Flakify [14] that uses CodeBERT. Other LLMs may also be used to parse test source code and to extract features for the traditional ML classifier for prediction, and they may lead to different results. Ultimately, we focus just on the effects of quantization and rectifying prediction loss using a traditional ML classifier evaluated specifically on CodeBERT. Future work can investigate

differences that may occur from using different LLMs.

VII. Related Work

Luo et al. performed the first empirical study on flaky tests in open-source projects [4]. They categorized flaky tests by manually inspecting developer-fixed flaky tests. Later researchers would develop techniques to automatically detect flaky tests, guided by the results from this empirical study [7]–[11], [48]. For example, Lam et al. developed iDFlakies that reruns tests in different orders to detect order-dependent flaky tests [7], and Shi et al. developed NonDex to detect tests that assume deterministic implementations of nondeterministic specifications [8]. These researchers collected the results of their techniques into a public dataset of now known flaky tests, called IDoFT [6]. This dataset also contains labels for the flaky tests, categorizing the tests based on the technique that was used to detect them. We use this dataset for our evaluation.

Many flaky-test detection techniques rely on rerunning tests or dynamic analysis, meaning they can be costly to run. Prior work investigated ML techniques to predict whether a test could be flaky. Alshammari et al. developed the first ML-based technique for detecting flaky tests, called FlakeFlagger [13]. They first created a large dataset of flaky tests by rerunning tests 10,000 times, and then they used this dataset to train and evaluate their ML classifier at predicting whether a test is flaky. They train their classifier to use static features like as test smells and lines of code as well as dynamic features like coverage and test runtime. They find their classifier obtains an F1-score of 85% in their evaluation on their dataset. Fatima et al. later proposed Flakify, which uses LLMs to similarly predict test flakiness, training the LLM to extract features from just test code . They find improved performance, with a reported F1-score of 98%. Our work is most similar to Flakify, because we also use the CodeBERT LLM for flaky-test prediction. Our evaluation shows similar high F1-scores. We also find that extracting features using CodeBERT greatly helps traditional ML classifiers at the same task.

Our evaluation on flaky-test categorization is most similar to prior work by Akli et al. and their technique FlakyCat. They obtained a dataset of flaky tests labeled by the reason for flakiness [21], based on the definitions provided by Luo et al. [4]. FlakyCat leverages CodeBERT to extract features from test code, and then it later uses few-shot learning to train a classifier to predict the flaky-test category. They similarly evaluate using some traditional ML classifiers that use CodeBERT features for doing the same prediction, finding that FlakyCat performs better than these traditional ML classifiers. We do the same evaluation on their dataset of flaky tests, but we find that these traditional ML classifiers actually predict effectively when using CodeBERT-extracted features. Akli et al. did not fine-tune their CodeBERT model whereas we do. Our findings show the importance of not just using CodeBERT to extract features but to also fine-tune this model for the specific task.

LLMs are also used for other software engineering tasks. Lemieux et al. proposed CodaMOSA, which uses LLMs to improve automatic test generation by providing hints to the search when traditional search approaches get "stuck" and cannot increase coverage any further [49]. Zhang et al. proposed training a LLM to perform software-edit tasks such as fixing bugs or updating comments, providing better performance over LLMs that are trained for code generation [50]. Xia et al. [51] and Fan et al. [52] both explored using LLMs for automatic program repair. Lee et al. proposed using LLMs for bug triaging. Researchers are quickly exploring more ways to use LLMs, and we investigate expanding its use in flaky-test detection and categorization tasks while combined with traditional ML classifiers.

Quantization is a widely used compression technique to reduce the precision of model parameters and activations to save memory and computation [16], [38], [39]. Yao et al. [53] proposed an end-to-end quantization and inference pipeline that can compress large Transformer-based models with minimal accuracy impact, with up to 5.19x/4.16x speedup on BERT and GPT-3-style models and a 3x memory footprint reduction. Xiao et al. [54] proposed a training-free and accuracy-preserving post-training quantization method for LLMs that migrate. They integrate SmoothQuant into PyTorch and FasterTransformer, achieving up to 1.56× inference acceleration and halving the memory footprint. In our work, we propose quantization that converts `float` data-type model weights to `int8`, which we find to improve the model's runtime and memory usage, reducing the cost of the LLM.

VIII. Conclusions

LLMs that predict whether tests are flaky or flaky-test categories without running them require heavy computational resources to execute deep neural networks. We propose FlakyQ to make LLM-based classifiers more efficient via quantization. We rectify any prediction loss form using a quantized model by training a traditional, computationally inexpensive, ML classifier that learns from features extracted using the quantized LLM to perform the prediction. Our evaluation shows that a fine-tuned LLM-based classifier's prediction time can be significantly reduced by applying quantization. Further, the additional traditional ML classifier trained on top of the quantized LLM masks prediction loss and achieves a similar F1-scores as the original LLM-based classifier. We find that LLM-extracted features are key to training an effective traditional ML classifier for these prediction tasks, as using features such as bag-of-words or vocabulary-based features out of test code does not achieve as high F1-scores.

In the future, we plan to explore the use of both LLMs and traditional ML classifiers to explain their predictions, as explanations may help developers understand and debug their flaky tests. We also plan on exploring use of additional features beyond those extractable from test source code. Finally, we plan on exploring how to combine different classifiers together, forming a "meta" classifier, to choose which classifier to use for specific tasks.

REFERENCES

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *International Conference on Automated Software Engineering*, 2016.

[3] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017.

[4] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014.

[5] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *International Working Conference on Source Code Analysis and Manipulation*, 2018.

[6] "IDoFT," http://mir.cs.illinois.edu/flakytests.

[7] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *International Conference on Software Testing, Verification, and Validation*, 2019.

[8] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *International Conference on Software Testing, Verification, and Validation*, 2016.

[9] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *International Symposium on Software Reliability Engineering*, 2020.

[10] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam, "Preempting flaky tests via non-idempotent-outcome tests," in *International Conference on Software Engineering*, 2022.

[11] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis*, 2014.

[12] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and M. Sasa, "Detecting flaky tests in probabilistic and machine learning applications," in *International Symposium on Software Testing and Analysis*, 2020.

[13] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *International Conference on Software Engineering*, 2021.

[14] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, 2023.

[15] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, "FlakyCat: Predicting flaky tests categories using few-shot learning," in *International Conference on Automation of Software Test*, 2023.

[16] X. Wu, C. Li, R. Y. Aminabadi, Z. Yao, and Y. He, "Understanding int4 quantization for transformer models: Latency speedup, composability, and failure cases," *arXiv preprint arXiv:2301.12017*, 2023.

[17] L. Breiman, "Random forests," *Machine learning*, vol. 45, 2001.

[18] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[19] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *International Conference on Software Engineering, Software Engineering in Practice*, 2019.

[20] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *International Conference on Software Engineering, Software Engineering in Practice*, 2020.

[21] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, 2023.

[22] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell, "Automatically reproducing timing-dependent flaky-test failures," in *International Conference on Software Testing, Verification, and Validation*, 2024.

[23] T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[24] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[25] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *International Conference on Software Engineering*, 2021.

[26] S. Dutta, A. Shi, and S. Misailovic, "FLEX: Fixing flaky tests in machine-learning projects by updating assertion bounds," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[27] C. Li, C. Zhu, W. Wang, and A. Shi, "Repairing order-dependent flaky tests via test generation," in *International Conference on Software Engineering*, 2022.

[28] S. Rahman and A. Shi, "FlakeSync: Automatically repairing async flaky tests," in *International Conference on Software Engineering*, 2024.

[29] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009, vol. 2.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *The Journal of Machine Learning Research*, vol. 12, 2011.

[31] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[32] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and computing*, vol. 14, 2004.

[33] J. Goldberger, S. Roweis, G. Hinton, and R. Salakhutdinov, "Neighbourhood components analysis," in *Advances in Neural Information Processing Systems*, 2005.

[34] G. E. Hinton, "Connectionist learning procedures," *Artificial Intelligence*, vol. 40, no. 1, 1989.

[35] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.

[36] "Code pretraining models," https://github.com/microsoft/CodeBERT, 2023.

[37] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," Tech. Rep., 2020.

[38] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[39] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao, "Post-training quantization for vision transformer," in *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[40] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *International Conference on Machine Learning*, 2010.

[41] A. de Brébisson and P. Vincent, "An exploration of softmax alternatives belonging to the spherical loss family," in *International Conference on Learning Representations*, 2016.

[42] A. Demaris, *Logit modeling: Practical applications*. Sage, 1992, no. 86.

[43] Z. Zhuang, M. Liu, A. Cutkosky, and F. Orabona, "Understanding adamw through proximal methods and scale-freeness," *Transactions on Machine Learning Research*, 2022.

[44] L. A. Montestruque and P. J. Antsaklis, "Static and dynamic quantization in model-based networked control systems," *International Journal of Control*, vol. 80, no. 1, 2007.

[45] H. Fan, G. Wang, M. Ferianc, X. Niu, and W. Luk, "Static block floating-point quantization for convolutional neural networks on FPGA," in *International Conference on Field-Programmable Technology*, 2019.

[46] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR*, 2020.

[47] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[48] R. Wang, Y. Chen, and W. Lam, "iPFlakies: A framework for detecting and fixing python order-dependent flaky tests," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2022.

[49] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMOSA: Escaping coverage plateaus in test generation with pre-trained large language models," in *International Conference on Software Engineering*, 2023.

[50] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *International Conference on Automated Software Engineering*, 2022.

[51] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *International Conference on Software Engineering*, 2023.
[52] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *International Conference on Software Engineering*, 2023.
[53] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," in *Advances in Neural Information Processing Systems*, vol. 35, 2022.
[54] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*. PMLR, 2023.