ChatGPT

Automated Loop Iteration Prediction using Machine Learning Techniques

Abstract

Real-time embedded systems often demand strict guarantees on timing, making accurate *Worst-Case Execution Time (WCET)* analysis crucial. One of the hardest challenges in WCET estimation is determining how many iterations a loop will execute, since loop bounds can depend on dynamic inputs and conditions. This project explores a machine learning approach to **automate loop iteration prediction**, aiming to reduce manual effort and improve the adaptability of WCET analysis. We generate a synthetic dataset of program loop characteristics and apply supervised learning models to predict loop behavior. A multi-label classification strategy is used to categorize loops by their iteration count and nesting level. The resulting models, especially a Random Forest classifier, achieve high predictive accuracy on the synthetic data, demonstrating the feasibility of using machine learning to estimate loop iteration bounds. This report presents the dataset design, machine learning methodology, evaluation results, and conclusions on the effectiveness of the approach, in a format accessible to developers and researchers interested in WCET and code analysis automation.

Introduction

In safety-critical and real-time systems (such as automotive controllers, medical devices, and industrial automation), it is essential to know how long tasks might take in the worst case. **Worst-Case Execution Time (WCET)** analysis provides an upper bound on execution time for software tasks, ensuring that systems can meet their deadlines reliably. A major factor in WCET is the execution of loops within code. However, accurately predicting the number of iterations a loop will run is notoriously difficult. Loop iteration counts often depend on variables or external inputs that are not known until runtime. For example, a loop may run until a sensor reaches a certain value or until user input meets a condition, meaning the exact iteration count can vary widely across different runs. Traditional WCET analysis methods address this by making conservative assumptions (like assuming maximum feasible iterations), or by requiring manual annotations of loop bounds by developers. These approaches tend to be labor-intensive, error-prone, and often overly pessimistic, potentially leading to suboptimal system performance or excessive resource provisioning.

Recent advances have sparked interest in using **machine learning (ML)** to assist WCET analysis, especially for automating the prediction of loop bounds. Machine learning models can, in principle, learn patterns from numerous examples of code to predict how loops behave without needing explicit human-provided bounds for each loop. Unlike static analysis that may struggle with complex or undecidable loop conditions, a trained ML model might infer likely iteration counts from features of the loop and its context. This project investigates the viability of such an approach. We develop an ML-based framework to predict loop iteration characteristics automatically. In particular, the study focuses on using classification models to predict whether a given loop will execute a small number of iterations and whether it is a nested loop, based on features extracted from the loop's code. By doing so, we aim to demonstrate a proof-of-concept that loop iteration prediction can be automated, which could eventually integrate with WCET tools to reduce manual annotation and adapt to different code patterns or platforms. The following sections describe the dataset creation, machine learning methodology, evaluation results, and conclusions drawn from this implementation.

Dataset and Preprocessing

Because there is no readily available corpus of code loops labeled with their iteration counts, we **constructed a synthetic dataset** to train and evaluate the models. This synthetic approach allows full control over loop parameters and outcomes, ensuring a diverse and balanced dataset. Each synthetic data point represents a loop in a simple abstract form, characterized by features that influence its iteration count. By generating the data programmatically, we can simulate common loop structures and provide correct labels (e.g. actual iteration counts or categories) without manual analysis of code.

Feature Design: Each loop in the dataset is represented by four key features that reflect its behavior:

- **initial_value:** The starting value of the loop's index or counter (e.g., a loop might start at `i = 0`).
- **increment:** The step by which the loop index changes each iteration (e.g., `i += 2` means an increment of 2).
- **exit_condition:** The threshold value at which the loop stops (e.g., loop runs while `i < 50` has an exit condition of 50). This effectively relates to the loop's range or limit.
- **nesting_level:** The depth of nesting for the loop (an integer indicating if the loop is inside other loops). For example, `nesting_level = 0` means it's a top-level loop, `1` means it is inside one outer loop, etc. This feature captures structural context of the loop in a program.

From these features, we can derive the loop's iteration count. In our simulation, the **iteration_count** is computed based on the relationship between the initial value, exit condition, and increment (essentially modeling a loop like `for(i = initial_value; i < exit_condition; i += increment)`). Using integer arithmetic, the iteration count can be approximated as `(exit_condition - initial_value) / increment` (rounded down to an integer). This derived iteration_count varies across the dataset, providing a ground truth for how many times each loop would execute.

Label Definition: We defined two target labels for each loop, framing the prediction problem as a multi-label classification task (each loop may belong to both or either category):

- **Small Iteration Loop (label_small_iter):** Indicates whether the loop runs for a *small number of iterations*. We define "small" as fewer than 10 iterations. If the computed iteration_count of a loop is < 10, this label is 1 (true); otherwise it is 0. This label helps identify loops that execute only a limited number of times.
- **Nested Loop (label_nested):** Indicates whether the loop is nested inside another loop. If `nesting_level > 0` (meaning the loop has at least one outer loop), this label is 1; if `nesting_level = 0` (no nesting, a top-level loop), the label is 0. This captures whether a loop is part of a nested structure, which can be important for understanding worst-case execution patterns.

Each loop instance in the dataset thus has two binary labels associated with it. For example, a loop with iteration_count 5 and nesting_level 1 would be labeled as a small iteration loop (since 5 < 10) and as a nested loop (since nesting_level = 1).

Data Generation and Balance: By generating loops with random parameters within chosen ranges (for instance, initial values between 0 and 20, increments between 1 and 4, exit conditions between 30 and 100, and nesting levels between 0 and 2), we created a dataset of 300 synthetic loop instances. This generation process was designed to ensure a mix of scenarios: some loops run only a few times, others run many times; some loops are nested, others are not. As a result, the two target labels are reasonably balanced across the dataset (roughly half of the loops have fewer than 10 iterations, and a proportion have nesting vs. not). A balanced dataset is beneficial for training classification models, as it prevents bias toward a majority class and allows the models to learn both outcomes effectively. Basic **exploratory data analysis** confirmed that the feature distributions cover the intended ranges (for example, exit_condition values are spread between 30 and 99, etc.) and that label 0/1 frequencies for both labels are not heavily skewed.

Before training, the dataset was saved in a CSV format for reproducibility and easy loading. For model training, we did minimal preprocessing since the data was already numeric and labeled. One consideration was to ensure that when splitting into training and testing sets, both labels remained proportionally represented (especially since *nested loops* were slightly fewer in number). This was addressed by using a stratified splitting strategy tailored for multi-label data, as described next.

Methodology

We formulated the loop iteration prediction as a **multi-label supervised learning** problem. This means each loop sample has two target outputs (the two labels described above), and the model needs to predict both for a given input feature set. The goal is to train a classifier that, given the four features of a loop (`initial_value, increment, exit_condition, nesting_level`), can correctly classify whether that loop is a small-iteration loop and whether it is nested.

Model Selection: We selected two machine learning algorithms to build and compare in this study: **Random Forest** and **K-Nearest Neighbors (KNN)**. These were chosen to provide a contrast between a more complex ensemble method and a simple baseline method, both known to perform well on structured data.

- **Random Forest:** This model is an ensemble of decision trees. It works by building numerous decision tree classifiers and aggregating their results (essentially taking a majority vote for classification). Random Forests are known for their robustness and ability to model non-linear relationships in data. They also tend to resist overfitting due to the averaging of many trees. Importantly, Random Forest provides a measure of **feature importance**, which can tell us which input features most strongly influence the predictions – a useful feature for interpretability in a research context.
- **K-Nearest Neighbors:** This is a straightforward, instance-based classifier. KNN makes predictions by looking at the *k* most similar data points in the training set (the "nearest neighbors" in feature space) and taking a majority vote of their labels. We used KNN as a baseline due to its simplicity and ease of understanding. It has no complex model training; it essentially memorizes the dataset. However, KNN's performance can be sensitive to the choice of the parameter *k* (number of neighbors) and to feature scaling (features with larger numeric ranges can dominate distance calculations). We set *k = 5* for our experiments, and we note that features in our data have different scales (for example, exit_condition values are much larger in magnitude than increment values), which could impact KNN's effectiveness if not addressed.

Training Pipeline: We followed a structured pipeline to train and evaluate the models, ensuring a fair comparison and reliable assessment:

1. **Data Splitting:** The dataset was divided into training and testing subsets using a multi-label stratified approach. We used stratified sampling to maintain the same proportion of each label combination in both the train and test sets. In practice, with 300 samples and two binary labels,

we ensured that the ~20% of data held out for testing had a representative mix of loops (both small-iteration and not, nested and not) similar to the training set. This approach prevents scenarios where, for instance, the test set might accidentally contain very few nested loops, which could skew the evaluation.

2. **Model Training:** We trained each classifier on the training data. For the Random Forest model, we used an ensemble of 100 decision trees (a typical default) and let it build its internal split rules based on the features. For KNN, the training phase simply involved storing the training instances since KNN does not create an explicit model beyond the dataset.
3. **Prediction:** After training, we used each model to predict the labels on the unseen test set. Because this is a multi-label classification, the models output two predictions for each test sample (one for each label). Both the Random Forest and KNN were configured to handle multi-output classification, meaning they can predict both labels in one go for each input.
4. **Evaluation:** We evaluated the predictions using appropriate classification metrics. Specifically, we computed **precision**, **recall**, and **F1-score** for each of the two labels. Precision measures the accuracy of positive predictions (how many of the loops predicted as, say, "small iteration" were truly small iteration loops), while recall measures coverage of actual positives (how many of the actual small iteration loops were correctly identified). The F1-score is the harmonic mean of precision and recall, providing a balanced single measure of model performance on a class. These metrics were chosen instead of simple overall accuracy because our task involves two classes and potential class imbalance; we wanted to ensure the model performs well on both identifying loops that meet the criteria and those that do not. In addition, we examined the confusion matrix for each label (a 2x2 table of predicted vs actual outcomes) to see where the models made mistakes – for example, if a model tended to misclassify some larger loops as "small" or vice versa, or if it ever misidentified nested status. This analysis helps in understanding failure modes of the models.

No complex hyperparameter tuning was performed beyond the basic choices (like number of trees for Random Forest or neighbors for KNN), as the focus was on baseline performance comparison. Both models were implemented using standard libraries (scikit-learn in Python), and training was relatively quick given the small size of the dataset.

Evaluation Metrics and Results

Model Performance: The Random Forest classifier demonstrated stronger performance than KNN in predicting both target labels. On the held-out test set, the Random Forest achieved high precision and recall for both **small-iteration loop** and **nested loop** predictions. In quantitative terms, Random Forest obtained an F1-score of approximately **92%** on the small-iteration classification and about **88%** on the nested loop classification. This indicates that it was very effective at identifying loops with fewer than 10 iterations, and also quite reliable in determining whether a loop is nested. The precision and recall for Random Forest were similarly high (on the order of ~0.9 for each label), meaning it made few false-positive errors and also caught most of the true positive cases in both categories.

The **KNN classifier**, while still performing respectably, lagged behind Random Forest. KNN achieved roughly **84% F1-score** for the small-iteration loops and about **80% F1-score** for nested loops. In practice, this means KNN was a bit less accurate and consistent: it missed more of the small-iteration loops (lower recall) and also sometimes misclassified larger-iteration loops as small (lower precision), and similarly for nested loops. The gap in performance shows that the more sophisticated ensemble method (Random Forest) was better suited to this task than the simple distance-based approach. We suspect that one reason for KNN's lower accuracy is the differing scales of the features: for instance, `exit_condition` values (ranging up to ~100) can dominate the distance computation compared to `nesting_level` (which ranges only 0–2). Without explicit feature scaling, this can bias KNN's neighbor comparisons. In our results, the confusion matrices for KNN showed more misclassifications, particularly confusing some non-nested loops as nested or vice versa, likely due to such feature distance issues. By contrast, Random Forest inherently handles features in their own splits and was able to more cleanly separate the classes.

Analysis and Insights: Beyond raw performance metrics, examining the Random Forest model provided insights into which features were most important for predictions. The Random Forest's **feature importance analysis** indicated that the loop's exit condition was the most influential feature for deciding if a loop has a small number of iterations, followed by the initial value and increment. This makes intuitive sense: a loop that runs from a low initial value to a relatively low exit condition, or that increments in large steps, will have fewer iterations. These factors directly determine the iteration count. The nesting level feature was comparatively less important for the small-iteration prediction, which again is logical – nesting doesn't directly affect how many times a loop runs, it's more about where the loop appears in code structure. However, for the **nested loop classification task**, the model understandably relied heavily on the `nesting_level` feature (since a non-nested loop has nesting_level 0 by definition). In fact, identifying a nested loop is almost directly solved by checking nesting_level > 0. The high performance of Random Forest on the nested loop label suggests it effectively learned that rule, while KNN likely also picked up on it but with slightly less consistency. Overall, the machine learning approach aligned with expectations: the model learned the relationships like *"if exit_condition minus initial_value is small relative to increment, then likely a small-iteration loop"* and *"if nesting_level is zero, then not a nested loop"*, which are essentially the encoded domain logic. This alignment provides confidence that the model's predictions are based on sensible patterns rather than spurious correlations.

In summary, the results on the synthetic test data were encouraging. The Random Forest model, in particular, was able to automatically predict loop iteration categories with high accuracy. This demonstrates that even a basic ML model can capture the fundamental factors determining loop execution counts and loop nesting, suggesting that the concept of learning loop behavior from data is feasible. The KNN model's moderate success also serves as a baseline; it performed the task better than chance, but its limitations highlighted the advantages of using more powerful models for this application.

Conclusion

This project has shown a successful application of machine learning to the problem of predicting loop iteration behavior, which is a critical component of WCET analysis in real-time systems. By creating a synthetic dataset of loops and training classification models, we were able to automatically determine whether a loop runs fewer than a threshold number of times and whether it is nested, using only simple features extracted from the loop definition. The approach effectively shifts some of the burden of loop analysis from manual estimation or rigid static analysis to a data-driven method. Our **Random Forest model** achieved high accuracy in classifying loops by iteration count and nesting status, indicating that patterns in loop behavior can indeed be learned from examples. This result is promising, as it suggests that with more data and refinement, ML models could assist developers and analysts in quickly obtaining loop bound information, thereby aiding the WCET estimation process and real-time system verification.

There are, however, important considerations and future directions to note. First, the current model was developed and validated on *synthetic data*. While this allowed us to verify the concept, real-world code may exhibit more complex loop patterns, dependencies, and behaviors (such as data-dependent loops, irregular increments, or early break conditions) that were not captured in our simulation. **Future work** should therefore focus on applying and testing the approach on loops extracted from actual codebases or benchmarks, to evaluate how well the models generalize to real software. This may involve creating a pipeline to parse code and generate features similar to those we used, or even employing machine learning models that can read code directly (for example, using techniques from program analysis or neural networks trained on code).

Additionally, the current study treated loop iteration prediction as a straightforward classification problem with a fixed threshold (10 iterations) for defining "small" loops. In a more general setting, one might want to predict the actual iteration count or a safe upper bound for each loop. Extending the model to perform regression (predicting a numeric bound) or multi-class classification (for various ranges of iterations) could increase its usefulness in WCET analysis. We also recognize that more advanced ML techniques could be explored: for instance, **transformer-based models or large language models** could potentially analyze loop code in a more granular way, and **reinforcement learning** might be used to simulate program execution steps to estimate loop bounds. These techniques were beyond the scope of this project but represent exciting avenues for research.

Another promising direction is to integrate machine learning predictions with traditional static analysis – a hybrid approach. ML models can quickly provide an estimate based on learned patterns, while static analysis can enforce safety (for example, ensuring the ML-predicted bound is not violated in any theoretical path). Such a combination could yield a robust system where machine learning provides a probabilistic guess and static methods provide guarantees or corrections, marrying efficiency with reliability.

In conclusion, **Automated Loop Iteration Prediction using ML** appears to be a viable approach for assisting WCET and real-time systems analysis. Our implementation demonstrates that even with a relatively simple feature set and ML model, we can achieve a high level of accuracy in classifying loop behavior. This work serves as a foundation that can be built upon with more data, more sophisticated models, and integration into tools. By bridging the gap between static code analysis and machine learning, developers and researchers can move towards more automated and intelligent analysis of program timing behavior, ultimately contributing to safer and more efficient real-time software systems.
