INCREMENTAL BOUNDED MODEL CHECKING USING MACHINE LEARNING TECHNIQUES

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Student id: 10580396
Author: Mengze Li
Supervisor: Lucas Cordeiro

Department of Computer Science

Contents

List of Tables

List of Figures

List of Codes

Abstract

This paper presents and evaluates the flag predictor for ESBMC based on five state-of-the-art machine learning models, which can automatically predict the optimal flag combination for ESBMC in term of the verification task. The five candidate machine learning models are Decision Tree, K-nearest Neighbours (KNN), support vector machine (SVM), random forest and neural network, respectively. In this project, training dataset and testing dataset are generated from a set of verification tasks in SV-COMP. Taking various criteria into consideration, experimental results demonstrates that decision tree is the most suitable machine learning algorithm for implementing the flag predictor for ESBMC. If compared to the other machine learning models, DT has achieved the highest accuracy and F1-score with the smallest time consumption. In addition, decision tree has been successfully applied to implement the flag predictor for ESBMC, which substantially improves the efficiency and effectiveness of ESBMC. In particular, if compared to the verification results generated by ESBMC with its default flags, the flag predictor assists ESBMC to increase valid verification results by 21% and reduce verification time consumption by 34%.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Re- productions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions de- scribed in it may take place is available in the University IP Policy (see http://documents. manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http: //www.library.manchester.ac.uk/about/regulations/) and in The University’s policy on presentation of Theses

Acknowledgements

Firstly of all, I would like to express my heartfelt gratitude to Dr Lucas Cordeiro, my supervisor, for his constant encouragement, patience and guidance on my project. He has provided me with helpful materials and encouraged me to solve challenges. Besides, I would like to thank all my friends, especially my girlfriend, who has helped me and encouraged me to get through the most tough time.

Chapter 1

Introduction

1.1 Motivation

ESBMC [1] is a context-bounded model checker based on satisfiability modulo theories (SMT) [2], which can be applied to verify both single- and multi-threaded C/C++ programs. With the aim of verifying C/C++ programs and generating the verification results, ESBMC takes the C/C++ programs as input and extracts verification conditions which are then passed to one of the SMT solvers to generate the final verification results. As ESBMC is consisted of different components such as Clang [3], GOTO converter, AST converter and SMT solvers [4], there are numerous flags such as --z3, --mathsat and--falsification. Each flag represents a verification option. For example, --z3 and --mathsat indicate that verifying the programs using the SMT solver Z3 [5] and MathSAT [6], respectively. Different flags can be combined to specify the verification option, and this results in considerable execution options.

An ideal flag combination can make ESBMC verify the aimed program effectively and efficiently. On the contrary, an inappropriate flag combination can lead to huge execution time or unsatisfactory verification results. Besides, the optimal flag combination could differ with respect to different input C/C++ programs and flags for different tasks are used to be selected based on expert experience which has an uncertain possibility of failure in selecting the optimal flag combination. As for these specific reasons, a flag predictor, which can provide the optimal flag combination in term of the aimed C/C++ programs, is in demand.

Selecting the optimal flag combination could make ESBMC generate the expected verification results with less time consumption which saves both labour and time cost. In addition, SV-COMP [7] is a competition on software verification which provides a large set of benchmarks categorized by different explicit properties for comparing performances of different software verifiers. ESBMC has achieved the third place in Overall in SV-COMP 2018, the third place in Falsification in SV-COMP 2019 and SV-COMP 2020. A flag predictor for ESBMC could assist ESBMC to shorten the verification time and generate satisfactory results aiming at achieving better results in SV-COMP.

1.2 Description

This project is to implement a flag predictor for ESBMC, which can automatically generate an optimal flag combination with respect to the input C/C++ program, by employing a machine learning model. This project can be divided into three main sections. The first section is feature engineering which extracts a set of feature vectors based on a set of input C/C++ programs. Each complete feature vector is consisted of three parts: feature extracted from the input file, flags chosen for verification and the label coded from the verification result.

The second section is model training and evaluation which uses these generated feature vectors as training and testing datasets to train and evaluate various machine learning models.

Once the model is trained and selected, the final section is to implement and test the flag predictor. The model with best overall performance is selected to implement the flag predictor for ESBMC. Then, generate a list of testing verification tasks and compare the overall verification results and time consumption for using ESBMC with and without the flag predictor.

The flag predictor can greatly reduce the labour and time cost of selecting appropriate verification flags and reduce the risks of employing ESBMC with inappropriate flags. Therefore, users can generate satisfactory verification results with less time consumption especially having a considerable number of tasks that need to be verified.

1.3 Aims and Objectives

The main goal of this project is to implement a flag predictor for ESBMC that can automatically provide optimal flags in term of different input programs. It contains three main sections: feature engineering, model training and evaluation, implementation and evaluation of flag predictor.

As a complete feature vector is consisted of three parts which are features extracted from the input file, flags chosen for verification process and the label coded from the verification result, steps need to be completed are as follows, in order to achieve the feature engineering (generating a set of complete feature vectors).

1. Select raw data.
   Verification tasks of different categories in C/C++ language from SV-COMP are ideal for this project. Tasks in three different categories (Array, Floats and Loop) are chosen. Each category is consisted of a large number of verification tasks.
2. Extract features from the selected raw data
   Each task (C source code) can be converted into AST (Abstract Syntax Tree) [8]. We count the number of different types of representative child nodes to form the first part of feature vector.
3. Construct complete feature vectors
   We need to list all possible flag combinations. Each task needs to be verified by ESBMC with all different flag combinations, and we need to label all the verification results. For each flag combination, we encode flags and the corresponding verification results as the second part and the third part of a feature vector. Then, combine the first part generated in Step 2, which is extracted from the corresponding task, to form a complete feature vector. Repeat this process for all tasks and all possible flag combinations.

The first section feature engineering is completed by following the above steps. Then, we can use these feature vectors for model training and evaluation.

4. Model training and evaluation

Split these generated feature vectors from Step 3 into training and testing datasets to train and test the performances of different machine learning models. Since we model the problem as a multi-class classification problem, five different state-of-the-art machine learning models, KNN (K-nearest neighbors) [9], SVM (support vector machine) [10], Decision Tree [11], Random Forest [12], Neural Network [13] are selected. Different criterion is applied to evaluate their model performances in different data categories.

The last section is implementation and evaluation of the flag predictor which is stated as follows.

5. Implement and test the flag predictor
   First is to generate a list of verification tasks for testing this predictor. For each verification task, use the model to predict all verification results. Then, choose the flag combination, which is predicted to lead to the best verification result, to verify the task. Next, record the verification result and execution time. Repeat the above process for each testing task. Compare the overall verification results and time consumption of verifying these tasks using ESBMC with and without the flag predictor.

1.4. Contributions

This paper describes and evaluates a decision-tree-based flag predictor for ESBMC, which can employ the trained decision tree classifier to predict and select the optimal flag combination for a new verification task. Therefore, the main contributions of this paper are as follows. Firstly, train and evaluate five machine learning models for predicting the verification result in term of the input verification task and the flag combination, which demonstrates that decision tree classifier has the highest accuracy with least training time. Secondly, implement and evaluate the proposed flag predictor base on decision tree classifier, which identifies that the proposed flag predictor can potentially improve the efficiency and effectiveness of ESBMC.

1.5 The Structure and outline of the dissertation

This article contains five chapters. It begins with an abstract that briefly describes this project, key results and achievements. Next, Chapter 1: Introduction is presented that introduces the reasons of the demand the flag predictor for ESBMC and detailed motivation. Besides, project description, main contribution, aims and objectives are also provided in this chapter. Chapter 2 starts with a description and understanding of problems and awareness of the solutions to the technical challenges. Then it describes the technical background that are involved in this project, including model checking, bounded model checking, ESBMC and various machine learning techniques. Chapter 3 thoroughly presents the methodology which can be divided into three sections. It contains the detailed description of each step in the three sections. Chapter 4 is experimental evaluation and discussion, which begins with the experimental objectives and description and is followed by a description of experimental setup. Next, experimental results and threats to validity are presented and discussed. Last chapter of this dissertation is conclusion and future work. This contains the conclusions and a summary of achievements, reflection, identification of improvements and related future work.

Chapter 2

Background

2.1 Description and understanding of problems

The main aim of this project is to implement a flag predictor for ESBMC which can automatically predict the optimal flag combination while using ESBMC to verify different input C files. The flag predictor can largely reduce the risks of huge execution time and unexpected results due to selecting inappropriate flags by users, especially facing a large scale of verification tasks. Besides, it can assist ESBMC to approach its performance limit with less user participation and less time cost.

In order to achieve this aim, various technical knowledge from the field of machine learning and verification are required. According to the steps that mentioned in Section aim and objectives in Chapter 1, this chapter will present the relevant technical background knowledge including Model Checking and Bounded Model Checking, ESBMC, AST, classification, various state-of-the-art machine learning methods and various criteria.

2.2 Awareness of the solutions to the technical challenges

1. The second step is to convert selected experimental data into AST form which is used to obtain the first part of feature vectors. Thus, the definition, description and application of AST are introduced.
2. With the purpose of constructing a set of complete feature vectors, ESBMC are employed. For this specific reason, basic conceptions of model checking and bounded model checking are presented. Furthermore, the verification tool ESBMC and its relevant knowledge, covering the architecture and usage, are provided in this chapter.
3. As for model training and evaluation, various state-of-the-art machine learning methods are employed. This chapter introduces different categories of classification and different machine learning approaches to solve classification problems. Scikit-learn library is also introduced in this chapter. Lastly, criteria for evaluation and relevant machine learning knowledge is covered.

2.3 Model checking and Bounded Model checking

Model checking [14] is an automatic verification technique that is able to verify if a finite-state system satisfies the specifications by systematically exploring and checking all reachable states. Given a finite-state system and a set of properties, using model checking technique to verify the system regarding to the properties can be processed in three steps [15]: modelling, specification, and verification.

Modelling is to generate a state-transition model that is consisted of finite states. Next, specification expresses the system behaviour and the given properties in temporal logic. It is not necessary to

specify all properties. Instead, partial properties are acceptable in this step, which allows model checking technique to perform partial verification in next step. Verification is the last step that explores all reachable states of the system and checks if each of them satisfies the properties specified in Step specification. If verification fails, it indicates that there is a state that violates the properties. In this case, a counterexample, that contains diagnostic information from initial system state to the property violation state, is generated. On the contrary, if the verification succeeds, it demonstrates that all relevant systems states are checked and satisfies the specifications.

The difference between Model Checking and Bounded Model Checking [16] is whether a bound is set to limit reachable states of the given system. Model checking checks if all reachable states satisfy the given properties, while bounded model checking checks if a subset of the system states within the given bound satisfy the properties. In brief, BMC checks the negation of a given property within a provided bound over a given transition system [17].

The principle of BMC is as follows. Given a transition system, a property to be checked and a bound K, BMC firstly unrolls the system K times and converts the unrolled system and the negation of the property into a verification condition. Next, SAT [18] or SMT solver is employed to check the satisfiability of this verification condition. If it is satisfiable, there is a reachable state within depth K that violates the given property. A satisfiable assignment of the verification condition, which is a counterexample that contains the execution trace and diagnostic information, is generated. On the contrary, if verification condition is unsatisfiable, the given property is satisfied over all reachable states at depth K or less.

2.4 ESBMC

ESBMC is a context-bounded model checker for automated verification of single- and multi-threaded C/C++ programs. It not only verifies the predefined safety properties (e.g., array bound violations, pointer safety, division by zero), but also provides assert-statements which allows users to declare additional properties. In addition, ESBMC offers both C and Python API which is simple for users to get started.

The composition of ESBMC and workflow of ESBMC are stated as follows. Figure 2.1 presents the workflow of ESBMC. Firstly, ESBMC takes a C program as input and passes it to Clang. Clang, which is a widely used state-of-the-art complier suite for C/C++/Objective-C/Objective-C++, is the front-end of ESBMC. ESBMC employs Clang to generate AST with respect to the input C program, and then sends it to its next component: control-flow graph (CFG) generator. CFG generator is applied to simplify the AST and convert it into a GOTO program. Next, ESBMC uses symbolic execution engine to execute the GOTO program. In this process, ESBMC unrolls loops k times and converts the unrolled program into static single assignment (SSA) form. SSA is then passed to the SMT back-end, in which SSA is translated into SMT formula containing all properties to be checked [19].

The back-end of ESBMC employs a SMT solver to check the satisfiability of the generated verification condition in SMT formula. One of five SMT solvers (Boolector, Z3, MathSAT, CVC4,

Yices) can be chosen for verification. If it is satisfiable, verification fails. In this case, a counterexample, which points out the property violation, is provided by ESBMC. On the contrary, if the formula is unsatisfiable, verification succeeds [20]. Besides, verification unknown and timeout are two possible results generated by ESBMC. Verification unknown indicates that ESBMC fails to compute the verification results, and timeout indicates verification is not completed in the set time limit.

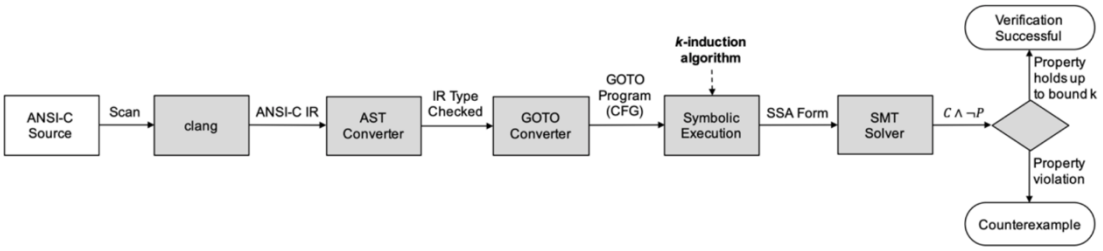

Figure 2.1: workflow of ESBMC [21]

There are different flags for verification with ESBMC. Users can configure their own execution flags, which could determine the execution times and verification results, based on different verification tasks and verification aims. For example, users can choose one of different SMT solvers which are Boolector, Z3, MathSAT, CVC4, Yices. Furthermore, ESBMC provides three flags corresponding to three different encoding methods: --ir, --floatbv and –fixedbv. Also, there are three verification methods of ESBMC: --k-induction, --incremental-bmc, and --falsification.

An example of an execution command of verifying a c program by ESBMC is as follows.
esbmc inputfile.c --z3 --floatbv --falsification
In this command, inputfile.c is the verification task which is the input C program. Z3 is chosen as the SMT solver and floatbv is selected as the encoding method. Also, verification method is specified with flag --falsification.

2.5 Abstract Syntax Tree

Abstract syntax tree is a tree representation of the abstract syntactic structure of source code written in a formal language. AST is more abstract comparing to the source code. This is due to the fact that it disregards many grammatical elements including blank lines, punctuation, and delimiters, while it contains the syntactic details such as if-else-condition and while-loop [22]. The nodes of AST consist of the syntactic constructs of the source code. In this project, AST is used as the representation of input source code in first section Feature Engineering. Input C program (verification task) is firstly converted into an AST, which contains the representative information of the input file. Then, the generated AST is applied to extract partial feature vectors from these various attributes and syntactic information by counting the numbers of different types of nodes in the AST. Besides, the generation of AST from source code is fast, and tree-shaped representation is easily retrieved and traversed.

2.6 Classification

In machine learning, classification problem [23] is a supervised leaning problem in which models are built by learning from provided data and are used to make new predictions. To be more specific, classification is to predict class labels for unlabelled instances from the problem domain by learning from the existing dataset such as instance-label pairs. The classification model can be described as a map function from input variables to discrete output variables, which is applied to assign class labels to new unlabelled instances.

Classification problems can be classified into three categories which are binary classification problems, multi-class classification problems and multi-label classification problems, respectively [24]. Binary classification represents the tasks that classify the unlabelled instances into one of two classes. For example, a classification task requires us to classify a set of pet pictures (containing dogs, cats, parrots, etc.) into two categories which are “Cat” and “Not cat”. Multi-class classification refers to the classification tasks that have more than two classes. For instance, we need to classify the set of pet pictures into three categories. They are “Cat”, “Dog”, and “Other”. Multi-label classification stands for the classification tasks (containing two or more classes) in which one or more class labels can be assigned to each instance. Taking the pet pictures as an example, if the class labels are “Cannot fly”, “Can fly” and “Can walk”, then two class labels can be assigned to pictures of parrots, which are “Can fly” and “Can walk”.

In order to solve classification problems by computer programs, we need to make them learn knowledge from existing labelled data. First of all, it is necessary to generate training dataset in which models learn knowledge from. Testing dataset is also significant since it is used to evaluate the trained models. As for classification problems, an instance in training or testing dataset should consist of a feature vector and its corresponding label. Thus, extract features from labelled raw data (such as text, tables, figures) to form a set of feature vectors, and combine feature vectors with their corresponding labels to form a set of instances as the whole dataset. Then, split the dataset into two sub-datasets, namely training dataset and testing dataset. Classification problems are now converted into mapping problems which requires us to find mapping functions from a set of vector variables to a set of discrete variables.

With the purpose of build the described mapping functions, previous study [25] presents numerous machine learning models based on different machine learning algorithms. State-of-the-art machine learning algorithms such as k-nearest neighbours, decision tree, support vector machine, naïve bayes, can be applied to solve both binary classification problems and multi-class classification problems [26]. As for multi-label classification problems [27], machine learning algorithms, including multi-label random forest [28], multi-label gradient boosting [29], can be employed. Selecting an appropriate algorithm based on the nature of classification problems (including the number of instances, the number of features in an instance, the number of class labels) is crucial, as it will have great influences on the classification results.

2.7 KNN (K-nearest neighbours)

K-nearest neighbours is a simple machine learning algorithm which can be used in classification. It predicts the class label for unlabelled data based on the majority votes of top K nearest labelled data.

The main steps of using k-nearest neighbours algorithm to predict the class label for an unlabelled instance are as follows. Firstly, it takes the training dataset as input which is consisted of a set of labelled instances. Then, it calculates the distances between the unlabelled instance and each labelled instance, in which different metrics for distance computation (such as Euclidean distance, Manhattan distance) can be applied. Next, it sorts the training instances based on calculated distances in ascending order and selects top K training instances, in which K is configured by users. Lastly, it returns the majority class label among the k instances as the class for the unlabelled instance.

Previous study [30] has listed the advantages and shortcomings of KNN algorithm for classification. As for advantages, it is easy to implement and simple to understand the working principles. It requires a few parameters which can easily be tuned by using cross-validation. However, it suffers from several pain points. Firstly, since it makes prediction based on the computed distances, it requires considerable computation if the dimension of data is high, or the number of instances is large. In this case, the algorithm becomes inefficient, which means that the classification speed slows down significantly. Besides, previous study [31] points out another disadvantage that is caused by high-dimensional data, which is the curse of dimensionality in KNN. This indicates Euclidean distance is not effective since the distances from the unlabelled instance to all labelled instances are almost the same.

In order to overcome the shortcomings caused by high-dimensional data, feature extraction and dimension reduction are two effective ideas. Several methods, which combines these two ideas in one step as a pre-processing step, are proposed by previous research [32]. For example, principal component analysis (PCA) [33], linear discriminant analysis (LDA) [34], and canonical correlation analysis (CCA) [35] are all feasible techniques in reducing dimension.

2.8 SVM (support vector machine)

Support vector machine is a fast supervised learning algorithm that can be used for classification. It is widely applied in the field of spam detection, text classification and sentiment analysis. Different from KNN, SVM is still effective even though dimension of data is high. As for binary classification problem, the objective of support vector machine algorithm is to find a hyperplane in N-dimensional space (where N is the dimension of data) that maximizes the separation of the data points to their potential classes.

Taking data points in two-dimensional space as an example, figure 2.2 shows possible hyperplanes that distinctly classifies data points into two categories. SVM is aimed at choosing the hyperplane that has the maximum margin, because unlabelled data is classified with more confidence if the margin is larger. The data points with minimum distance to the selected hyperplane are named as Support Vectors. In figure 2.2, two red data points and one blue data points on the dotted lines are support vectors. The hyperplane in two-dimensional space is a line, and it becomes a plane if it is in three-dimensional space. With the dimension of data (number of features) increases, it is more difficult to imagine the optimal hyperplane.

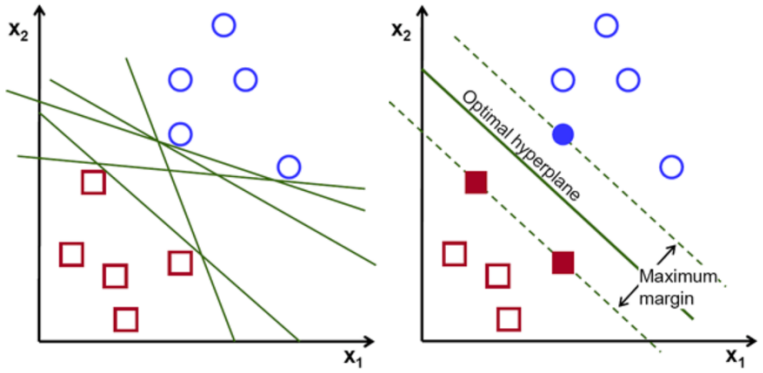

Figure 2.2: possible hyperplanes

If the data points cannot be easily separated by a linear SVM, non-linear SVM is applied to classify these data points. In this case, kernel functions are employed in order to make the data points separable by transforming them into higher dimension space. Figure 2.3 shows an example of plotting non-separable data points into separable data points. According to the distribution of all data points in two-dimensional space, it cannot be separated by a line (hyperplane in two-dimensional space). Therefore, a kernel function is applied to plot these data points into three-dimensional space. Then, it becomes easy to find a hyperplane that clearly classifies the data points into their own class. Different kernel functions, such as Gaussian Kernel Radial Basis Function (RBF), Sigmoid Kernel and Polynomial Kernel, can be selected based on the nature of the classification problem.

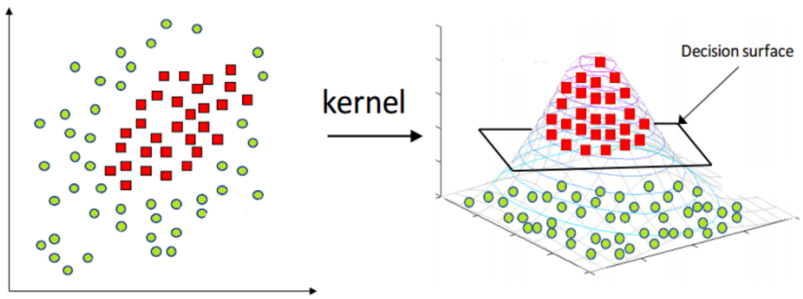

Figure 2.3: transformation two-dimensional data points into three-dimensional space

There are two proposed approaches, namely one-vs-one and one-vs-rest, which make SVM can also be applied to multi-class classification [36]. The common objective of the two approaches is to split a multi-class classification problem into multiple binary classification problems. In one-vs-one approach, since hyperplanes between every two classes are built, it requires us to find m(m-1)/2 classifiers, where m is the number of classes. As for one-vs-rest approach, we pick one class and treat the rest points as another class. Then, we need to find a hyperplane between the selected class and all others. Therefore, m hyperplanes are required to be built. Both are valid approaches, but one-vs-rest is preferred in practice due to less runtime.

2.9 Decision Tree

Decision tree is an easy-to-implement supervised learning algorithm for classification and regression [37]. It is simple to be visualized as a tree structure, in which the leaves are class labels and internal nodes represents the features [38]. Decision tree classifier is constructed based on labelled training data. If it is applied to classify an unlabelled instance, it starts from the root node, in which a decision is made based on the corresponding feature value. Then, it goes to one of the child nodes and decides the next movement. Repeat this process until it reaches a leaf node, in which the corresponding class label is assigned to the unlabelled instance. Therefore, the branches in decision tree refers to the conjunctions of features that lead to class labels.

Figure 2.4 shows a simple example of a decision tree. It classifies the pictures of animals including dogs, tigers, and parrots. Thus, there are three leaves that represent "dogs", "tigers" and "parrots", respectively. If we need to classify a new unlabelled picture, it firstly starts with the feature of the root node, which is "have wings". If the animal in the picture has wings, then it goes to the right child node which is a leaf. In this case, it is assigned to the class "parrots". Otherwise, if it doesn't have wings, it goes to the left child node whose feature is "pet". If the animal in the picture is widely kept as a pet, it goes to the left leaf node which is class "dogs". Otherwise, it is assigned to label "tigers".

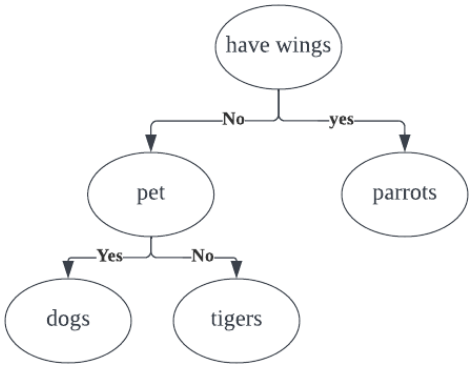

Figure 2.4: a simple example of a decision tree

The construction of a decision tree classifier is a recursive process. It starts with selecting a feature for root node based on information gain. The feature leading to the largest information gain is selected to build the root node. This is because information gain represents the reduction of class uncertainty if a feature is known. Then, different decisions based on the feature values leads to different child nodes. Next, consider unused feature and calculate the information gain of each feature. Pick the largest information gain of feature to establish the child node. Recursively repeat the process until all features are used or the limitation (such as the depth of tree, the minimum information gain) is reached.

Information gain in ID3 algorithm is calculated by the difference of entropy before and after split. Entropy can be calculated by the following equation 2.1

$$E = -\sum_i^C \ p_i \log_2 \ p_i \tag{2.1}$$

Where p_i is the probability of randomly choosing an element of class i, and C is the number of classes. Thus, the information gain can be calculated by the following equation 2.2:

$$Gain = E_{\text{before}} - E_{\text{split}} \tag{2.2}$$

2.10 Random Forest

A random forest classifier is stable and powerful machine learning model which is consisted of multiple decision tree classifiers on various sub-samples of dataset. Thus, the construction of a random forest classifier repeats the process of build a decision tree [39]. Firstly, choose the number of decision tree classifiers that are used to consist of a random forest classifier. Then, select a subset of the dataset and train a decision tree classifier based on the selected data. Repeat this process until the number of decision trees is reached. Random forest classifies a unlabelled instance by choosing the majority votes of decision trees.

Since a random forest classifier is formed by a number of decision tree classifiers, random forest algorithm benefits from its structure. The first advantage of the random forest model is stability which means a new data or missed data rarely impacts the model. If a new data point is added to the whole dataset, it would not have considerable influence on the random forest model. This is because it is significantly difficult to impact all decision trees in the random forest model. Besides, a number of decision trees that are trained on various sub-samples can reduce the overall biasness of the random forest algorithm.

However, the structure of random forest model also brings several disadvantages. The principal disadvantage is the complexity of building and applying a random forest model. Since it contains a larger number of decision trees, this indicates that more computation and training time are required. In addition, it takes more time for generating predictions.

2.11 Neural Network

Neural network is a state-of-the-art machine learning model which is widely applied in the industrial field of artificial intelligence such as image recognition and medical diagnosis. A neural network can be applied to both classification and regression. Neural network is constructed by connecting layers of perceptron which receives inputs and generates outputs by multiplying inputs by weights and passing them into an activation function [40]. The widely used activation functions are logistic sigmoid function, hyperbolic tan function, and the rectified linear unit function. Thus, it is named as multi-layer perceptron (MLP), which is consisted of an input layer, an output layer and one or more hidden layers. Each neuron of the input layer represents a feature of input data, whereas the output of the output layer in classification problem refers to a class label.

Multi-layer perceptron (MLP) is capable to model with a large number of non-linear data. MLP allows users to customize various hyperparameters including the number of hidden layers, the number of neurons on each hidden layers, and the weights. We can config the MLP based on the nature and demand of the classification problem. However, it requires a large computation and time

consumption to tune these hyperparameters. In addition, MLP is sensitive to weight initializations because of non-convex functions in hidden layers, which indicates that different weight initializations can result in greatly different validation accuracy.

2.12 Confusion matrix

Confusion matrix is a table visualizing and summarizing the performance of a classification algorithm [41]. Table 2.1 shows the template of a confusion matrix. It contains four basic classification results as follows.

1. Ture Positive (TP): TP represents the number of successful predictions of positive samples.
2. True Negative (TN): TN represents the number of successful predictions of negative samples.
3. False Positive (FP): FP is the number of negative samples which are falsely predicted as positive.
4. False Negative (FN): FN is the number of positive samples which are falsely predicted as negative.

Confusion matrix		Prediction	
		Positive	Negative
Actual values	Positive	TP	FN
	Negative	FP	TN

Table 2.1: a template of confusion matrix

Based on confusion matrix, performance metrics including accuracy, precision, recall and F1-score can be calculated via the following equations.

1. Accuracy represents the ratio of samples that are successfully predicted into their actual classes.

$$\text{Accuracy} = \frac{(TP+TN)}{(TP+FP+FN+TN)} \tag{2.3}$$

2. Precision refers to the ratio of correctly predicted positive samples to all predicted positive samples.

$$\text{Precision} = \frac{TP}{TP+FP} \tag{2.4}$$

3. Recall refers to the ratio of correctly predicted positive samples to all actual positive samples.

$$\text{Recall} = \frac{TP}{TP+FN} \tag{2.5}$$

4. F1-score combines both precision and recall and can be used to compare two classifiers.

$$F1Score = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{2.6}$$

As for the confusion matrix for multi-class classification, it is a m*m table, where m is the number of classes. We can transform the table into m 2x2 confusion matrix tables. For example, table 2.2 is a confusion matrix for three-class classification. It can be converted into three 2*2 confusion matrix. One of three confusion matrix is shown in table 2.3. According to table 2.3, class 2 and class 3 are combined into one class (class 2 or 3). Accuracy, precision, recall and F1-score of class 1 can be calculated based on table 3. These performance metrics of class 2 and class 3 can also be computed by following the similar process.

		Predicted classes		
		Class 1	Class2	Class 3
Actual classes	Class 1	15	5	10
	Class 2	2	20	9
	Class 3	3	8	17

Table 2.2: an example of confusion matrix for three-class classification

Confusion matrix		Prediction	
		Class 1	Class 2 or 3
Actual values	Class 1	15	5+10 =15
	Class 2 or 3	2+3 = 5	20+9+8+17 = 54

Table 2.3: transformed confusion matrix

In order to evaluate the overall performance of a multi-class classifier, there are three methods that can be applied, namely Macro-average, weighted-average and micro-average.

2.13 Scikit-learn

Scikit-learn (sklearn) [42] is an open-source machine learning library for Python, which is built on NumPy, SciPy and matplotlib. It provides a large number of machine learning techniques including various classification algorithms, regression models, clustering algorithms, and considerable criteria for measuring performance of different machine learning models. These machine learning techniques are programmed in Class that allows users to import these ready-to-use classes. The classes of machine learning approaches have different attributes and implement different methods. This enables users to get access to the attributes and call the methods via several simple lines of python codes.

Chapter 3

Research Methodology

This chapter firstly presents the overview of the project which contains brief description of experimental steps and experimental tools. Next, it thoroughly describes each step of experiments including implementation, comparison, testing and evaluation

3.1 Overview

In order to implement a flag predictor for ESBMC using machine learning techniques, this project can be divided into three experimental sections. The main steps and tools of each experimental section are described as follows.

The first experimental section is feature engineering which is aimed at generating feature vectors from the selected experimental raw data as training data and testing data. Each feature vector is consisted of three parts. Thus, this experimental section can be processed by completing steps as follows.

1. Select verification tasks (C source program) of different sub-categories with suffix .c from SV-COMP 2017-2022 as experimental raw data. As shown in Figure 3.1, the whole set of verification tasks is consisted of different sub-category of verification tasks. Split the set of verification into training raw data and testing raw data, in which training raw data takes 80% and testing raw data takes 20%.
2. For each selected verification task, convert it into AST by Clang and construct the first part of a feature vector by counting the numbers of different types of nodes in AST.
3. Select main flags of ESBMC and list all combinations of the chosen flags.
4. Categorize and encode the flags. Use different feature codes to represent different flags.
5. For each selected verification task, employ ESBMC to verify it with all flag combinations as shown in Figure 3.2. For each flag combination, it is represented by a unique list of feature codes, which is the second part of a feature vector.
6. Categorize and encode the verification results. Use different labels to represent different verification results.
7. For each flag combination, ESBMC generates a verification result which is translated into a label. This is the third part of a feature vector.
8. As shown in Figure 3.2, concatenate these three parts of a feature vector to form a complete feature vector.
9. Repeat step 2-8 on the whole set of selected verification tasks.

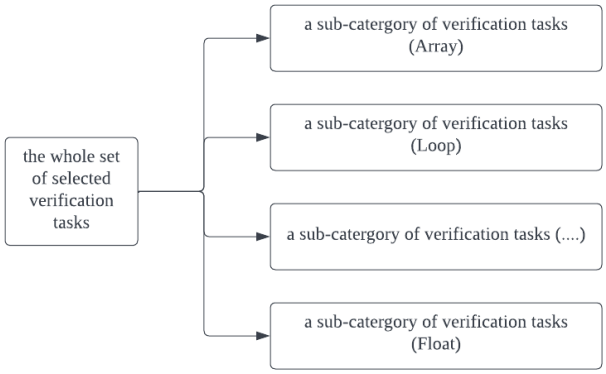

Figure 3.1: Composition of experimental raw data

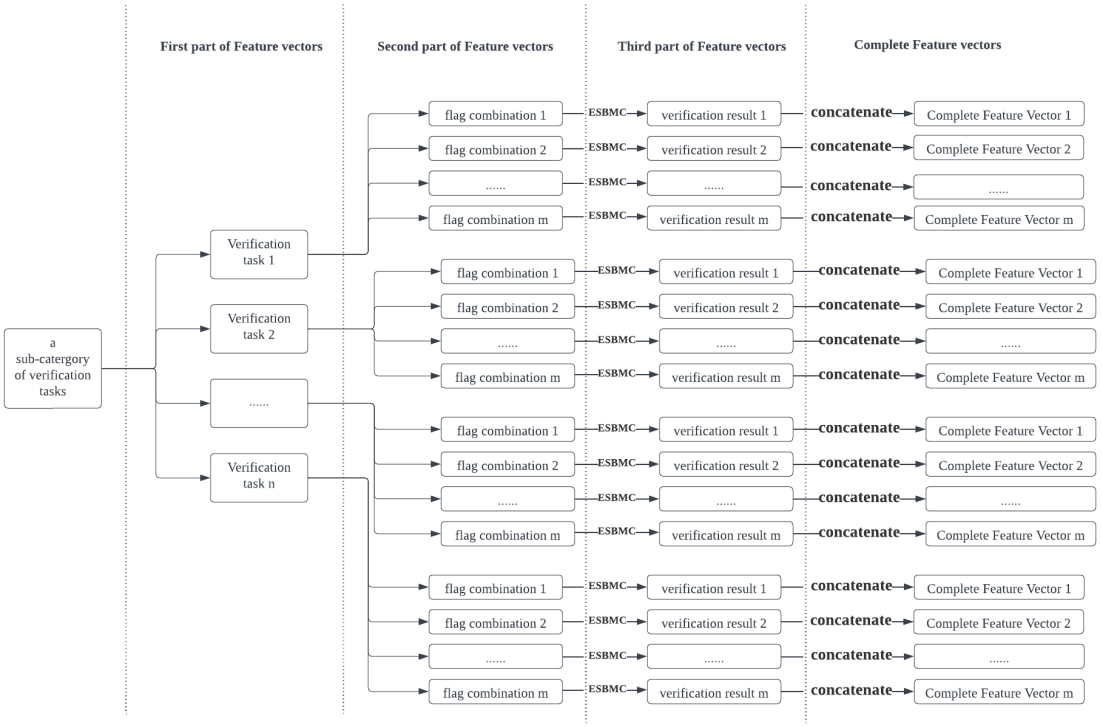

Figure 3.2: brief workflow of generating complete feature vectors

The second experimental section is model training and evaluation. It takes generated feature vectors as training and testing datasets to train and evaluate different machine learning models.

1. Choose different machine learning models which are suitable for multi-class classification. Since the verification results are encoded into different labels (more than two), this problem is converted as a multi-class classification problem. KNN (K-nearest neighbours), SVM (support vector machine), Decision Tree, Random Forest, Neural Network are selected.
2. Import the chosen machine learning models from Python Scikit-learn library. GridSearchCV is also imported which allows us to tune hyperparameters automictically in order to derive the best performance of the chosen machine learning models.

3. Compare the performances of different machine learning models by confusion matrix. In addition, accuracy, precision, recall and F1-score are applied to evaluate the trained models.
4. Repeat Step 2-4 on all sub-categories of feature vectors.
5. After evaluating the results based on different criterion, choose a machine learning model with the best overall performance to implement the flag predictor.

The third experimental section is implementation and evaluation of flag predictor. The workflow of the flag predictor is shown in Figure 3.3.

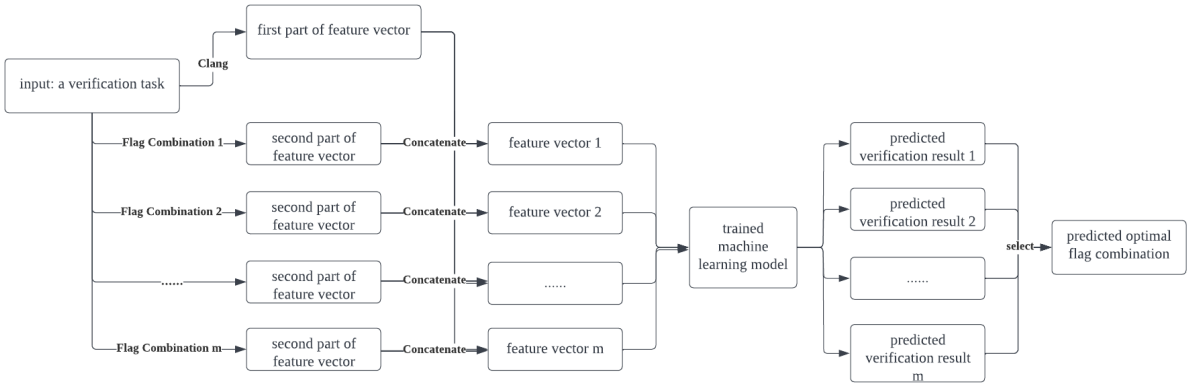

Figure 3.3: workflow of the implemented flag predictor

In order to predict the optimal flag combination for a verification task, the flag predictor takes the task as input and employs Clang to convert it into AST. Next, it generates the first part of feature vector based on the AST. Meanwhile, it encodes different flag combinations into different lists of feature codes and concatenates them with the first part of feature vector to form a set of feature vectors. The flag predictor passes the set of feature vectors to the trained machine learning model which is chosen at the end of previous section. Then, the trained machine learning model predicts verification results. Finally, the flag predictor selects the flag combination, which leads to the best predicted verification result, as the optimal flag combination.

The flag predictor can be implemented and evaluated by steps presented as follows.

1. Employ the selected machine learning model.
2. Split the whole set of feature vectors into training dataset and testing dataset.
3. Train the model with the generated training dataset and use GridsearchCV to tune the hyperparameters in order to achieve better performance.
4. Assemble different components shown in the workflow to complete the implementation.
5. Use the testing data to evaluate. For each testing C source file, use the implemented flag predictor to predict the optimal flag combination. Employ ESBMC to verify the testing files with their corresponding predicted flag combinations. Then, employ ESBMC to verify them with the default flag combinations.
6. Compare the overall time-consumption and the overall verification results.

In summary, the main aim of feature engineering is to generate feature vectors for training and testing machine learning models. The second section is to use the achieved set of feature vectors to

train and evaluate various machine learning models. Then, select the “best” machine learning model. Given an input C file and a flag combination, the selected machine learning model could predict the verification result. Next is to employ the model to implement the flag predictor. Given a C file, flag predictor traverses all flag combinations and employs the trained machine learning model to predict all verification results. It outputs the flag combination, which leads to the best verification result, as the optimal flag combination. Finally, use a set of verification tasks to test and evaluate the implemented flag predictor.

3.2 Feature Engineering

3.2.1 Selecting raw data

SV-COMP is an annual competition on software verification aiming at comparing the performance of different verifiers regarding to different safety properties. It provides various verification tasks with different explicit properties to check such as memory safety, array index out of bound, division by zero. These verification tasks are classified into different categories. Taking time consumption of training several machine learning models into consideration, we select three sub-categories (Array, Float, Loop) from 6th SV-COMP to 11th SV-COMP as the whole set of verification tasks in this project.

3.2.2 Generating first part of feature vectors

Once the raw data (a set of C source files) is ready to be processed, we need to abstract features which could have substantial influence on time consumption and performance on ESBMC. These features are used to form the first part of feature vectors, while the rest parts of feature vectors will be discussed later.

ESBMC firstly employs Clang to convert the input C program into AST, and the time consumption of parsing the AST node objects by ESBMC could greatly differ. Therefore, AST is selected as the representation where the feature vector is extracted. In addition, since the scale of the raw data is large, we could benefit from the fast conversion speed and easy analysis of AST, which allows us to generate a set of feature vectors with low time consumption.

Firstly, given the set of raw data (C source files), Clang is applied to convert them into ASTs. Clang provides a built-in AST-dump mode for converting the input C file into an AST, which is enabled with flag -ast-dump. As shown in Code 3.1, it is a simple example of C source program without C headers, and it includes a for-loop, multiple variable declarations, variable assignments and a return statement.

The model for the Clang’s AST nodes is a class hierarchy without a common ancestor. There is a numerous number of node types in Clang’s AST. Statements (Stmt) and declarations (Decl) are two basic nodes, which have a large number of childnodes such as ForStmt, DoStmt, GotoStmt, IfStmt, VarDecl, EmptyDecl. Also, expressions (Expr) are statements in Clang’s AST. The code 3.2 shows the generated Clang’s AST form of code 3.1, which does not include the internal declaration of

Clang. The first line of code 3.1 (Main function declaration) is converted into the node FunctionDecl, which is the root node of this generated AST. According to the code 3.2, node FunctionDecl has one childnode CompoundStmt which represents a group of statements, namely ForStmt, DeclStmt and ReturnStmt. ForStmt represents for loop statement in line 3 of code 3.1, and ReturnStmt corresponds the statement return 0. In addition, node DeclStmt and its childnode VarDecl are used to represent the statement of declaring variables.

```
int main(){
int a;
for(int k = 0; k < 10; k++)
{
   a = k;
}

   return 0;
}
```

Code 3.1: Example of C source program

```
`-FunctionDecl 0x7fa3f009ce28 <test.c:1:1, line:9:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x7fa3f009d208 <col:11, line:9:1>
    |-DeclStmt 0x7fa3f009cf90 <line:2:1, col:6>
    | `-VarDecl 0x7fa3f009cf28 <col:1, col:5> col:5 used a 'int'
    |-ForStmt 0x7fa3f009d1a0 <line:3:1, line:6:1>
    | |-DeclStmt 0x7fa3f009d048 <line:3:5, col:14>
    | | `-VarDecl 0x7fa3f009cfc0 <col:5, col:13> col:9 used k 'int' cinit
    | |   `-IntegerLiteral 0x7fa3f009d028 <col:13> 'int' 0
    | |-<<<NULL>>>
    | |-BinaryOperator 0x7fa3f009d0b8 <col:16, col:20> 'int' '<'
    | | |-ImplicitCastExpr 0x7fa3f009d0a0 <col:16> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x7fa3f009d060 <col:16> 'int' lvalue Var 0x7fa3f009cfc0 'k' 'int'
    | | `-IntegerLiteral 0x7fa3f009d080 <col:20> 'int' 10
    | |-UnaryOperator 0x7fa3f009d0f8 <col:24, col:25> 'int' postfix '++'
    | | `-DeclRefExpr 0x7fa3f009d0d8 <col:24> 'int' lvalue Var 0x7fa3f009cfc0 'k' 'int'
    | `-CompoundStmt 0x7fa3f009d188 <line:4:1, line:6:1>
    |   `-BinaryOperator 0x7fa3f009d168 <line:5:3, col:7> 'int' '='
    |     |-DeclRefExpr 0x7fa3f009d110 <col:3> 'int' lvalue Var 0x7fa3f009cf28 'a' 'int'
    |     `-ImplicitCastExpr 0x7fa3f009d150 <col:7> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x7fa3f009d130 <col:7> 'int' lvalue Var 0x7fa3f009cfc0 'k' 'int'
    `-ReturnStmt 0x7fa3f009d1f8 <line:8:3, col:10>
      `-IntegerLiteral 0x7fa3f009d1d8 <col:10> 'int' 0
```

Code 3.2: AST form of code 3.1

With the length of code increases, the amount and types of AST nodes will considerably increase. Therefore, in order to generate practical feature vectors, we select 12 significant and common nodes as features, which are ForStmt, WhileStmt, IfStmt, SwitchStmt, DoStmt, LabelStmt, DeclStmt, VarDecl, FunctionDecl, CallExpr, ImplicitCastExpr and DeclRefExpr. Thus, given a verification task, the first part of feature vector is generated by counting the numbers of the selected AST nodes in the converted AST. Taking code 3.1 and its generated AST (code 3.2) as an example, the first part of feature vector is 1, 0, 0, 0, 0, 0, 2, 2, 1, 0, 2, 4. This is because the AST contains 1 ForStmt, 2 DeclStmt, 2 VarStmt, 1 FunctionDecl, 2 ImplicitCastExpr and 4 DeclRefExpr, while it doesn not have nodes in types WhileStmt, IfStmt, SwitchStmt, DoStmt, LabelStmt or CallExpr.

3.2.3 Generating Second part of feature vectors

The first part of feature vectors only contains features from the input verification tasks. With the

aim of predicting the optimal flag combination for ESBMC, we need to add flags of ESBMC as the second part of feature vectors. Thus, we select the main flags of ESBMC and list all combinations of the chosen flags.

ESBMC provides a flag --help for users to view additional options (BMC options, solver configuration, property checking, k induction, etc.) of ESBMC and their corresponding execution flags. We select flags from three categories: SMT solvers, encoding methods and verification methods, which could have great influence on performance of ESBMC. Five SMT solvers are selected which are Boolector, Z3, MathSAT, CVC4 and Yices, respectively. Three flags of encoding methods --ir, --floatbv and –fixedbv are also used as one of the features in second part.

There are also three verification methods: k-induction, falsification and incremental BMC. As for k-induction and incremental BMC, ESBMC provides a flag --max-k-step to set the maximum number of iterations. Besides, users can use the flag --k-step to set incremental limitation. In this project, 7 pairs of max-k-step and k-step are selected which are (25, 1), (50, 1), (100, 2), (200, 4), (400, 16), (800, 32) and (1600, 64). However, if the verification method is chosen as falsification, users do not need to specify the maximum number of iterations or incremental limitation.

Since integer encoding mode is not supported by Boolector in ESBMC, verification method can only be falsification if the SMT solver and encoding method are specified as Boolector and ir, respectively. Therefore, this experiment has 211 flag combinations in total. For example, “--z3 --fixedbv --k-induction --max-k-step 50 --k-step 1” is one of the flag combinations.

After listing all flag combinations, in order to convert them into the second part of feature vectors, we need to use digital codes to represent these features. Table 3.1 shows all experimental flags and their feature codes. For example, translate flag combination “--z3 --fixedbv --k-induction --max-k-step 50 --k-step 1” into the second part of feature vector is 1, 2, 2, 1.

SMT solver	Feature code	Encoding method	Feature code	Verification method	Feature code	(Max-k-step, k-step)	Feature code
Boolector	0	Ir	0	Falsification	0	(25, 1)	0
Z3	1	Floatbv	1	Incremental BMC	1	(50, 1)	1
MathSAT	2	Fixedbv	2	K-induction	2	(100, 2)	2
CVC4	3					(200, 4)	3
Yices	4					(400, 16)	4
						(800, 32)	5
						(1600, 64)	6

Table 3.1: flags and corresponding feature codes.

3.2.4 Generating the third part of feature vectors

The aim of generating feature vectors is to train machine learning models which can predict the

verification results in term of the input verification tasks and selected flags. Thus, the third part of feature vectors are extracted from the verification results.

we firstly need to classify verification results into different categories and model this problem as a multi-class classification problem. There are five kinds of verification results shown as follows. VERIFICATION FAILED and VERIFICATION SUCCESSFUL are two common verification results, which indicate ESBMC successfully complete the verification process against the input verification task. VERIFICATION FAILED shows that ESBMC fails to verify the given task due to property violation. On the contrary, VERIFICATION SUCCESSFUL means that ESBMC successfully verify the task. Both are valid verification results. Timed out is a verification result when ESBMC cannot complete the verification process within the time limit. Besides, if the ESBMC concludes with result VERIFICATION UNKNOWN, it means that ESBMC cannot complete the verification process within the specified number of iterations. The last possible verification result is ERROR which demonstrates that an error is occurred during the verification process.

Table 3.2 shows the classification labels of different verification results. Since VERIFICATION FAILED and VERIFICATION SUCCESSFUL are both valid verification results, they can be classified into the same category, and execution time becomes the criteria of efficiency. As shown in Table 3.2, if ESBMC generates VERIFICATION FAILED and VERIFICATION SUCCESSFUL as the verification result within 1 second, it is labelled with code 0. If the verification time consumption is in range (1, 60], the label is 1. If it takes more than 60 seconds and no more than 300 seconds, the label is 2. Besides, Timed out, VERIFICATION UNKNOWN and ERROR are respectively labelled with 3, 4 and 5.

	VERIFICATION FAILED or VERIFICATION SUCCESSFUL			Timed out	VERIFICATION UNKNOWN	ERROR
Time consumption range / seconds	[0, 1]	(1, 60]	(60, 300]			
label	0	1	2	3	4	5

Table 3.2: class labels of different verification results

After defining the conversion rules between verification results and labels, we need to use the selected 211 flag combinations to generate the third part of feature vectors. For each verification task, we traverse the 211 flag combinations, which means that we need to employ ESBMC verify the task 211 times, and each verification process is executed with different flag combinations. Repeat this process for all selected verification tasks. For example, if a verification result is VERIFICATION FAILED and its execution time is 0.65, the third part of this feature vector is 0.

3.2.5 Generating complete feature vectors

After defining the methods for generating each part of feature vectors, we need to concatenate them

to form complete feature vectors.

The whole process for generating complete feature vectors of a given verification task is described as follows. Taking code 3.1 as an example, this verification task (code 3.1) is passed to Clang to generate AST, where the first part of feature vector is extracted. Use the method describe in Section 3.2.2 to generate the first part of feature vector which is 1, 0, 0, 0, 0, 0, 2, 2, 1, 0, 2, 4. Next, verify code 3.1 with 211 flag combinations and obtains the verification results. Employ table 3.1 and table 3.2 to generate the second part and third part of feature vectors. For example, if flag combination "–z3 –fixedbv –k-induction –max-k-step 50 –k-step 1" is applied and ESBMC provides VERIFICATION SUCCESSFUL as the verification result within 1 second, the second part and third part of this feature vector is 1, 2, 2, 1, and 0, respectively. Thus, this complete feature vector is 1, 0, 0, 0, 0, 0, 2, 2, 1, 0, 2, 4, 1, 2, 2, 1, 0. As a result, 211 feature vectors are generated against one verification task.

Repeat the above process for all selected verification tasks to build a whole set of feature vectors.

3.3 Machine learning model training and evaluation

This section firstly introduces different machine learning models which are suitable for implementing ESBMC's flag predictor. In addition, it describes the methods and tools of using these machine learning models and tuning their hyperparameters.

After generating a set of complete feature vectors (instances), we need to train different machine learning models and use different criterion to select the machine learning model with the best overall performance. This problem is modelled as a multi-class classification problem as follows. Given a verification task and a flag combination, classify the instance (generated from provided verification task and the flag combination) into one of six verification result classes. To be more specific, this problem is to predict verification result in term of input verification task and selected flag combination. In order to solve this multi-class classification problem, KNN (K-nearest neighbours), SVM (support vector machine), Decision Tree, Random Forest and Neural Network are selected. Python scikit-learn library, which provides various machine learning models and auxiliary methods, is employed in this experimental section.

First of all, we need to split the whole set of feature vectors into training dataset and testing dataset. Training dataset takes 80% of the whole set of instances and testing dataset is consisted of the rest instances. We use the training dataset to train different machine learning models and use testing dataset to evaluate their performances.

3.3.1 K-nearest neighbours

Sklearn.neighbors.KneighborsClassifier is imported as KNN model. There are five main hyperparameters that need to be tuned while training a KNN classifier. The first hyperparameter is named as "n_neighbors" which means the number of neighbours to use. The default value of "n_neighbors" is 5. "weights" stands for the weight function used in prediction. If "weights" is set

to “uniform”, it represents that all points in each neighbourhood are weighted equally. While, if it is set to “distance”, it means that closer neighbours have a greater influence.

Next, hyperparameter “algorithm” stands for the algorithm used to compute the nearest neighbours. “auto”, “ball_tree”, “kd_tree”, and “brute” are four possible values. “ball_tree” will use BallTree and “kd_tree” will use KDTree. Brute-force search will be used if it is set to “brute”. The default value is “auto”, which will automatically select the most appropriate algorithm. If BallTree or KDTree is selected, we can adjust the fourth hyperparameter “leaf_size” which could greatly influence the speed of construction and query. The default value is 30, and the optimal value is determined by the nature of classification problem. Lastly, “p” is also a significant hyperparameter which decides the method of calculating distances. If “p” equals to 1, Manhattan distance will be applied. If “p” is set to its default value 2, it will use Euclidean distance. Minkowski distance is used if “p” is an arbitrary value other than 1 or 2.

KneighborsClassifier provides methods of training and using a KNN model. Firstly, we declare a KNN classifier via KneighborsClassifier with selected hyperparameters. Then, train the KNN model by using method fit() which is to fit the KNN classifier from the training dataset. Lastly, test or employ the trained model by method predict() which is used to predict the class label for the provided data.

3.3.2 Support Vector Machine

As for training a support vector machine model, we use SVM model in sklearn.svm.SVC library. There are three main hyperparameters which are “C”, “kernel”, “gamma”. “C” is regularization parameter which needs to be set as a positive float. The strength of the regularization is inversely proportional to C. Next, kernel type is specified via “kernel”. We can choose one of five kernel types which are “linear”, “rbf”, “poly”, “sigmoid” and “precomputed”, respectively. If “rbf”, “poly” or “sigmoid” is chosen, “gamma” is applied to specify the kernel coefficient by setting it as “auto” or “scale”.

Similar to KneighborsClassifier, SVC also provides methods fit() and predict() to train and employ the SVM model. However, if the scale of dataset is too large, it could be impractical due to large training time consumption.

3.3.3 Decision Tree

Decision tree is built in Class sklearn.tree.DecisionTreeClassifier. Since a decision tree model is trained by learning simple decision rules inferred from data features, hyperparameters of DecisionTreeClassifier are mainly about customizing the splitting rules.

“Criterion” is the first hyperparameter which determines the function to measure the quality of a spilt. It can be set to one of three functions (“gini”, “log_loss”, “entropy”). Besides, hyperparameter “splitter” is used to select the strategy of choosing the split at each node. It is set to either “best” or “random”. Strategy “best” is to choose the best split and “random” is to choose the best random

split. “Max_feature” is also an important hyperparameter when looking for the best split. Max_feature represents the number of features to consider when looking for the best split. In addition, “max_depth” is the max depth of the tree, which can be used to control the termination of split. If it is set to its default value None, the split will not end until all leaves are pure or until all leaves contain less than “min_samples_split” samples. “Min_samples_split” is the minimum number of samples required to split an internal node.

Class DecisionTreeClassifier provides fit() function and predict() function which can be used to train and test a decision tree classifier. Besides, it also contains other methods which can check the status of the decision tree. For example, method get_depth() returns the depth of the decision tree, and decision_path() enables users to check the decision path in the tree.

3.3.4 Random Forest

Class sklearn.ensemble.RandomForestClassifier is employed to build the random forest classifier in this project. Random forest classifier fits a number of decision tree classifiers on various sub-datasets, which could improve the accuracy of prediction and control over-fitting. The sub-dataset size is determined by hyperparameter “max_samples” if “bootstrap” is set to True. Otherwise, if “bootstrap” is False, the whole set of data is applied to build each tree in the forest. Furthermore, “n_estimators” defines the number of trees in the forest. Since random forest classifier involves a number of decision tree classifier, RandomForestClassifier has some hyperparameters which are identical to the hyperparameters of DecisionTreeClassifier, such as “Criterion”, “max_depth”, “min_samples_split”, “max_features”.

After setting the hyperparameters of RandomForestClassifier, use fit() method to build a random forest of decision trees from the training dataset. Then, predict() is applied to predict the class in term of given instances. Besides, method score() can be used to evaluate the trained random, since it returns the mean accuracy on the given test data and their corresponding labels.

3.3.5 Neural Network

Sklearn.neural_network has two sub-classes MLPClassifier and MLPRegressor which are respectively used to solve classification problems and regression problems. Multi-layer perceptron is a supervised learning algorithm containing one input layer, one output layer and one or more hidden layers. Class MLPClassifier implements a MLP algorithm using backpropagation. As we need to build neural network for solving this multi-class classification problem, Softmax is applied as the output function while using MLPClassifier.

“Hidden_layer_sizes”, “solver”, “alpha”, “learning_rate_init” and “max_iter” are five main hyperparameters. As for “hidden_layer_sizes”, the ith element represents the number of neurons in the ith hidden layer. For example, hidden_layer_sizes = (6,) means one hidden layer with 6 neurons. Next, we can use “solver” to declare the solver for weighted optimization. There are three choices: “lbfgs”, “sgd” and “adam”. If solver is selected as “sgd” or “adam”, hyperparameter “learning_rate_init’ controls the step-size in updating the weights. Furthermore, “Max_iter”

represents the maximum number of iterations, and “Alpha” refers to Strength of the L2 regularization term, whose default value is 0.0001.

In this project, we use Class MLPClassifier and config the hyperparameters of MLPClassifier to initialize a neural network. Next, we use the provided method fit() and predict() to train the model and predict the class label using the trained multi-layer perceptron classifier.

3.3.6 GridsearchCV

In this project, GridsearchCV is applied to tune hyperparameters of imported machine learning models from Python Scikit-learn classes. It traverses all different hyperparameters that users feed into parameter grid and produces the best hyperparameter combination based on scoring metric which is chosen by users.

Class sklearn.model_selection.GridSearchCV contains four parameters: “estimator”, “param_grid”, “scoring”, “cv”. “Estimator” should be an estimator object, such as sklearn.svm.SVC() and sklearn.tree.DecisionTreeClassifier(). “Param_grid” is a dictionary that takes parameter names as keys and lists of parameter settings as values. For example, if we are tuning hyperparameters of KNN, “param_grid” could be {‘n_neighbors’: [5, 6, 7, 8, 9, 10], ‘algorithm’: [“auto”, “ball_tree”, “kd_tree”, “brute”]}. Besides, “cv” is specified the cross-validation splitting strategy. If an integer k is assigned to “cv”, it indicates that k-fold cross validation is applied. The parameter “scoring” determines the strategy to evaluate the performance of cross-validated models on the testing dataset. Scoring metric is flexible and allows users to evaluate models via one single score or multiple scores, which can be selected from a list of scoring parameter values such as “accuracy”, “f1”, “roc_auc”, etc.

Class sklearn.model_selection.GridSearchCV implements fit() which is run on all set of parameters. It has an attribute best_param which enables users to get the best hyperparameter combination of the machine learning model based on user-defined scoring metric.

3.3.7 Confusion matrix

Confusion matrix is a matrix that is applied to define and visualize the performance of a classification technique. It contains four basic characteristics: True Positive, False Positive, True Negative, False Negative. With these four characteristics, different metrics, such as accuracy, precision, recall, F1-score, can be calculated and used to evaluate the classification algorithm. In this project, we use confusion matrix that is implemented in Class sklearn.metrics.confusion_matrix to evaluate the performances of different machine learning models. It takes two main parameters which are “y_true” and “y_pred”, respectively. “y_ture” represents the ground truth of target values and “y_pred” is the predicted target values. The output is a confusion matrix, which is further used to calculate more metrics to evaluate these classification algorithms.

3.4 Implementation and evaluation of flag predictor

The flag predictor for ESBMC is consisted of three parts: feature vector generator, trained machine

learning model and result selector. This section firstly describes the implementation and workflow of each part in details. Then, it presents the implementation and evaluation methods of the flag predictor.

3.4.1 Trained Machine learning model

After evaluating the performances of different machine learning models, we select the one with the best overall performance as the classification algorithm to implement the flag predictor. Firstly, we need to split the whole set of verification tasks into two sub-datasets: training dataset and testing dataset. 80% of verification tasks are split into training dataset and the rest are used as testing data. Next, use methods stated in Section 3.2 to generate feature vectors based on training dataset and train the selected machine learning model with these feature vectors. In this process, GridSearchCV is employed to tune the hyperparameters in order to achieve the best performance.

3.4.2 Feature vector generator

Once the model is trained, the core of the flag predictor is implemented, which could predict the verification result class if a verification task and a flag combination are provided. Therefore, we need to implement a feature vector generator. Given a verification task, the feature vector traverses all flag combinations and generate a set of feature vectors (without class label) as mentioned in Section 3.2. The feature vector generator can be implemented based on the methods presented in Section 3.2.2 and Section 3.2.3.

3.4.3 Result Selector

Next, we combine the feature vector generator and the trained machine learning model. It takes a verification task as input and predicts all verification results by traversing all flag combinations. In order to select the optimal verification result, a result selector is necessary. Given a set of verification result labels, a result selector selects the best verification result by picking the smallest result label, and it outputs the corresponding flag combination. For example, if one predicted verification result is “VERIFICATION SUCCESSFUL 0.1s” which is labelled with 0 and another predicted verification result is “VERIFICATION SUCCESSFUL 10s” which is labelled with 1, the result selector will select the first result as the best verification result since 0 is smaller than 1. Then, the corresponding flag combination is the output of the result selector.

3.4.4 Flag predictor

We combine the feature vector generator, the trained model and the result selector to form the flag predictor for ESBMC. The flag predictor works as follows. It takes a verification task as input and passes it to the feature vector generator which can traverse all flag combinations and generate a set of feature vectors. Then, the set of feature vectors are sent to the trained machine learning model to predict their verification result labels. Flag predictor passes the predicted labels to the result selector, which selects the best verification result and outputs the corresponding flag combination as the optimal flag combination.

3.4.5 Evaluation

In order to evaluate the implemented flag predictor for ESBMC, we use testing dataset (a set of verification tasks) as the input of flag predictor. As for each verification task, the flag predictor generates the optimal flag combination. Then, we use ESBMC to verify these verification tasks with their predicted optimal flag combinations. In comparison, we use ESBMC to verify these verification tasks with the default flags. Total verification time and the number of valid verification results (VERIFICATION SUCCESSFUL and VERIFICATION FAILED) are two criteria to evaluate the flag predictor.

Chapter 4

Experimental Evaluation and Discussion

4.1 Objectives and Description

Two main experimental objectives are presented.

1. Train and evaluate five machine learning models (KNN, SVM, decision tree, random forest and neural network). Compare these state-of-the-art machine learning models via three criteria which are accuracy, F1-score, training time.
2. Evaluate the flag predictor based on the "best" machine learning approach. Identify if the flag predictor can assist ESBMC to improve its efficiency and effectiveness.

To train the five machine learning models, the whole set of feature vectors are split into training dataset and testing dataset. Training dataset takes 80% of the whole set and testing dataset takes 20%. Machine learning models in scikit-learn are employed and GridSearchCV is applied to tune hyperparameters. To evaluate these machine learning models, testing dataset is applied. We build confusion matrix, in which the accuracy and F1-score are calculated. Besides, training time and testing time are both recorded. Taking accuracy, F1-score, training time and testing time into consideration, five candidate models are compared via these criteria, and the model with the best overall performance is selected as the core of the proposed flag predictor.

Table 4.1 shows the numbers of different datasets that are applied in experiments.

	Amount
Total raw data	753
Whole set of feature vectors	753*211= 158883
Training raw data	602
Training raw data	602*211 = 127022
Testing raw data	151
Testing dataset	151*211=31861

Table 4.1: Amount of different datasets

The flag predictor can be developed by following the proposed method in Section 3.4. In order to evaluate the flag predictor. The experiments are designed to demonstrate if the flag predictor can help ESBMC to improve its efficiency and effectiveness as follows. Firstly, get the set of verification tasks which are used to generate testing dataset. Then, verify these verification tasks by ESBMC with its default flags and record the number of valid verification results as VR1 and the total verification time as VT1. Verify the same set of verification tasks by ESBMC with the predicted flags and record the number of valid verification results as VR2 and the total verification time VT2. Effectiveness is identified by comparing if VR2 is larger than VR1, which indicates that the flag predictor can assist ESBMC to increase the number of valid verification results. Efficiency is

identified by comparing of VT2 is smaller than VT1, which indicates that the flag predictor can assist ESBMC to reduce the verification time consumption.

4.2 Experimental setup

All experiments in this project were conducted on a 2.3 GHz OCTA Intel Core i9 processor with 16GB of RAM, running macOS Catalina 10.15.7 64-bits. As for feature engineering, Clang 9.0.1 is applied to generate ASTs. ESBMC 6.4.0 and five built-in SMT solvers (including Boolector 3.0, Z3 4.8.8, MathSAT, Yices 2.2.0, CVC4) are used to generate the labels of feature vectors. Verification time is measured in seconds based on CPU time. The timeout for each verification process is 5 minutes. Once the feature construction is completed, Python 3.7.6 is employed as the programming language to program the rest experimental sections. Classes from Scikit-learn library are imported to provide different machine learning models, hyperparameter tuning approach and measurement criteria.

4.3 Experimental results

Five tables that contain the confusion matrixes are presented as follows. Table 4.2 is the confusion matrix of KNN. Table 4.3 is the confusion matrix of Neural network. Confusion matrix of decision tree and confusion matrix of random forest are presented in Table 4.4 and Table 4.5, respectively. Table 4.6 contains the confusion matrix of SVM.

		Prediction class					
		0	1	2	3	4	5
Actual class	0	8921	417	192	8	225	128
	1	128	1821	4	7	175	84
	2	34	16	178	22	0	73
	3	167	131	57	1542	6	92
	4	381	12	0	0	761	95
	5	859	1072	71	187	1158	12837

Table 4.2: confusion matrix of KNN

		Prediction class					
		0	1	2	3	4	5
Actual class	0	7518	86	18	961	785	523
	1	591	1463	17	89	57	2
	2	78	9	165	47	16	8
	3	350	127	59	1278	148	33
	4	87	0	4	61	982	115
	5	2518	157	21	794	778	11916

Table 4.3: confusion matrix of neural network

		Prediction class					
		0	1	2	3	4	5
Actual class	0	8619	62	19	8	133	1050
	1	294	1816	3	0	41	65
	2	7	33	219	28	4	32
	3	155	147	82	1222	119	270
	4	322	16	43	3	847	18
	5	1185	414	55	0	490	14040

Table 4.4: confusion matrix of decision tree

		Prediction class					
		0	1	2	3	4	5
Actual class	0	8531	49	51	13	74	1173
	1	531	1317	88	5	111	167
	2	92	6	128	2	29	66
	3	580	3	134	1185	6	87
	4	38	29	22	10	1077	73
	5	863	655	187	236	101	14142

Table 4.5: confusion matrix of random forest

		Prediction class					
		0	1	2	3	4	5
Actual class	0	5718	5	19	27	833	3289
	1	310	1741	0	3	18	147
	2	207	1	91	13	2	9
	3	263	37	21	1659	9	6
	4	382	177	45	69	483	93
	5	4281	2009	11	220	312	9351

Table 4.6: confusion matrix of SVM

Confusion matrixes are shown as above. Accuracy and F1-score are calculated based on these confusion matrixes. Figure 4.1 is a histogram that shows the accuracy and F1-score of the five candidate machine learning models. Decision tree has the highest accuracy and the highest F1-score, which are 84.0% and 84.11%, respectively. The accuracy and F1-score of random forest classifier and KNN classifier are both slightly lower than decision tree. The accuracy and F1-score of neural network are 73.2% and 73.38%, respectively. However, the performance of SVM is unsatisfactory, whose accuracy is 59.7% and F1-score is 58.91%.

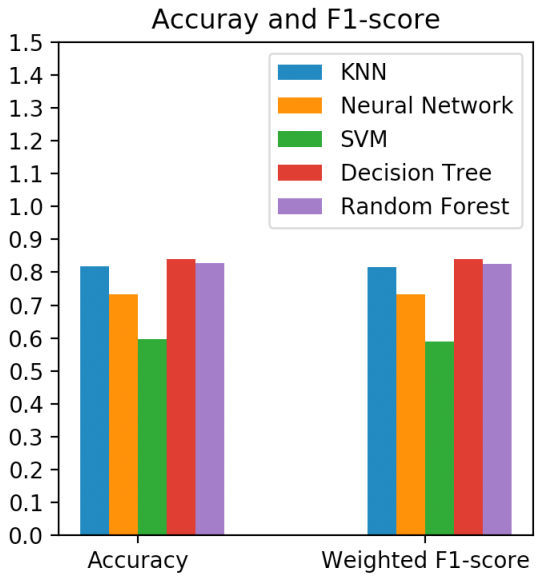

Figure 4.1: a histogram showing accuracy and F1-score of five algorithms

By comparing the accuracy and F1-score, we can remove the neural network and SVM from the model candidate list. As for comparing the rest three machine learning models, training time need to be considered. Table 4.7 lists the training time for KNN, decision tree and random forest. It shows that KNN is the most time-consuming in the process of training and decision tree has completed the model training with the least time consumption. Therefore, decision tree is selected as the machine learning model for implementing the flag predictor, since it has achieved the highest accuracy and F1-score with the least training time among the five candidate machine learning models. The first experimental objective is achieved.

	Training time (seconds)
KNN	2871
Decision Tree	17
Random Forest	265

Table 4.7: training time of three machine learning models

The experiments of evaluating the flag predictor are processed by two steps. The first step is to use ESBMC to verify the set of testing raw data (verification tasks) with its default flags. Record the total verification time and the number of valid verification results which is either VERIFICATION SUCCESSFUL or VERIFICATION FAILED. Label the total verification time as VT1 and the number of valid verification results as VR1. The second step is to employ ESBMC to verify the same set of verification tasks with the predicted flags. Record the total verification time as VT2 and the number of valid verification results as VR2.

Table 4.8 records VT1, VT2, VR1 and VR2. The total verification time decreases from 43816 seconds to 28918 seconds. For the same verification tasks, the flag predictor assists ESBMC to reduce the total verification time consumption by 34%. It demonstrates that the flag predictor can help ESBMC to improve its efficiency. In addition, as shown in table 10, the number of valid

verification results grows from 57 to 69, if the predicted flags are used. 21% increase in the number of valid verification results indicates that the flag predictor can help ESBMC to enhance its effectiveness. Therefore, the second experimental objective is achieved.

	ESBMC with default flags	ESBMC with predicted flags
Total verification time (seconds)	43816	28918
Valid verification results	57	69

Table 4.8: VT1, VT2, VR1 and VR2

4.4 Threats to Validity

All experimental objectives are achieved, and experimental results identifies that the proposed flag predictor can substantially improve the efficiency and effectiveness of ESBMC. Nonetheless, two threats to the validity of the results are discovered.

First is the imbalanced class distribution, which could result in low accuracy in classifying instances of the minority class. There are six class labels, but the training data is not distributed in average. Instances with class label 0, class label 4, class label 5 are the majority in the training data, whereas instances in other classes only take a small proportion. Although ignoring imbalanced class distribution could not impact the overall accuracy, a lack of considering class distribution may reduce the conclusion.

Second is the scale of testing data for the flag predictor. Taking time consumption into consideration, 151 verification tasks are applied as testing data for the flag predictor. Though the experimental objective of evaluating the flag predictor is achieved, testing data containing more verification tasks can make this study more convincing.

Chapter 5

Conclusion and future work

As a powerful bounded model checker for C/C++ programs, ESBMC allows users to config verification settings for the verification process. Various options can be combined and selected, which is difficult for inexperienced users to choose the suitable flag combination. Inappropriate flag combination can result in considerable time consumption and invalid verification results. In order to overcome this shortcoming, we firstly trained and evaluated five machine learning algorithms (KNN, SVM, decision tree, random forest, neural network) based on feature vectors extracted from three subsets (Array, Floats, Loop) of verification tasks in SV-COMP. Next, after evaluation based on three criteria (accuracy, F1-score, training time), we developed and evaluated a flag predictor for ESBMC, which relies on decision tree classifier to predict the optimal flag combination with regard to the input verification task.

5.1 Summary of achievements

Evaluation results among the five candidate machine learning algorithms demonstrate that the decision tree reaches the highest accuracy and F1-score with the least training time. Experimental results on the implemented flag predictor shows that the flag predictor helps ESBMC to increase the number of valid verification results by 21% and decrease the total verification time by 34%. This indicates that the flag predictor can substantially assist ESBMC to improve both efficiency and effectiveness.

5.2 Reflection, identification of improvements

As for the experiments, there are three aspects that can be improved. Firstly, the raw data is selected from three subsets of verification tasks in SV-COMP. More data in other categories are not considered for generating the feature vectors. Verification tasks from all categories can be applied for further experiments. Secondly, more machine learning models can be added to the candidate list. Five machine learning models are applied, but there are more machine learning models with different versions for different types of classification. More machine learning models can be evaluated for further experiments. Third is the criteria of evaluating the machine learning models, we can use more criteria to evaluate candidate machine learning models, in order to select the most suitable machine learning model for implementing the flag predictor.

5.3 Future work

The future work follows the three aspects of further improvements. First is to use data from all categories to generate the feature vectors. Second is to employ and evaluate more machine learning models. Third is to use more criteria to evaluate these candidate machine learning models in order to find the most suitable algorithm as the core of the flag predictor.

Bibliography

[1] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B. and Nicole, D.A., 2018, September. ESBMC 5.0: an industrial-strength C model checker. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 888-891).

[2] Barrett, C. and Tinelli, C., 2018. Satisfiability modulo theories. In Handbook of model checking (pp. 305-343). Springer, Cham.

[3] Lattner, C., 2008, May. LLVM and Clang: Next generation compiler technology. In The BSD conference (Vol. 5, pp. 1-20).

[4] Reynolds, A. and Kuncak, V., 2015, January. Induction for SMT solvers. In International Workshop on Verification, Model Checking, and Abstract Interpretation (pp. 80-98). Springer, Berlin, Heidelberg.

[5] Moura, L.D. and Bjørner, N., 2008, March. Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 337-340). Springer, Berlin, Heidelberg.

[6] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A. and Sebastiani, R., 2008, July. The mathsat 4 smt solver. In International Conference on Computer Aided Verification (pp. 299-303). Springer, Berlin, Heidelberg.

[7] Beyer, D., 2022. Progress on software verification: SV-COMP 2022. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 375-402). Springer, Cham.

[8] Neamtiu, I., Foster, J.S. and Hicks, M., 2005, May. Understanding source code evolution using abstract syntax tree matching. In Proceedings of the 2005 international workshop on Mining software repositories (pp. 1-5).

[9] Zhang, Z., 2016. Introduction to machine learning: k-nearest neighbors. Annals of translational medicine, 4(11).

[10] Noble, W.S., 2006. What is a support vector machine?. Nature biotechnology, 24(12), pp.1565-1567.

[11] Myles, A.J., Feudale, R.N., Liu, Y., Woody, N.A. and Brown, S.D., 2004. An introduction to decision tree modeling. Journal of Chemometrics: A Journal of the Chemometrics Society, 18(6), pp.275-285.

[12] Biau, G. and Scornet, E., 2016. A random forest guided tour. Test, 25(2), pp.197-227.

[13] Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Mohamed, N.A. and Arshad, H., 2018. State-of-the-art in artificial neural network applications: A survey. Heliyon, 4(11), p.e00938.

[14] Clarke, E.M., 1997, December. Model checking. In International Conference on Foundations of Software Technology and Theoretical Computer Science (pp. 54-56). Springer, Berlin, Heidelberg.

[15] Baier, C. and Katoen, J.P., 2008. Principles of model checking. MIT press.

[16] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O. and Zhu, Y., 2009. Bounded model checking. Handbook of satisfiability, 185(99), pp.457-481.

[17] Li, M. and Cordeiro, L., 2021, November. Assisted Counterexample-Guided Inductive Optimization for Robot Path Planning. In 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-8). IEEE.

[18] Gong, W. and Zhou, X., 2017, June. A survey of SAT solver. In AIP Conference Proceedings (Vol. 1836, No. 1, p. 020059). AIP Publishing LLC.

[19] Gadelha, M.R., Monteiro, F., Cordeiro, L. and Nicole, D., 2019, April. ESBMC v6. 0: verifying C programs using k-induction and invariant inference. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 209-213). Springer, Cham.

[20] Araujo, R.F., Ribeiro, A., Bessa, I.V., Cordeiro, L.C. and Joao Filho, E.C., 2017, November. Counterexample guided inductive optimization applied to mobile robots path planning. In 2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR) (pp. 1-6). IEEE.

[21] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B. and Nicole, D.A., 2018, September. ESBMC 5.0: an industrial-strength C model checker. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 888-891).

[22] Cui, B., Li, J., Guo, T., Wang, J. and Ma, D., 2010, October. Code comparison system based on abstract syntax tree. In 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT) (pp. 668-673). IEEE.

[23] Kotsiantis, S.B., Zaharakis, I.D. and Pintelas, P.E., 2006. Machine learning: a review of classification and combining techniques. Artificial Intelligence Review, 26(3), pp.159-190.

[24] Kotsiantis, S.B., Zaharakis, I. and Pintelas, P., 2007. Supervised machine learning: A review of classification techniques. Emerging artificial intelligence applications in computer engineering, 160(1), pp.3-24.

[25] Osisanwo, F.Y., Akinsola, J.E.T., Awodele, O., Hinmikaiye, J.O., Olakanmi, O. and Akinjobi, J., 2017. Supervised machine learning algorithms: classification and comparison. International Journal of Computer Trends and Technology (IJCTT), 48(3), pp.128-138.

[26] Grandini, M., Bagli, E. and Visani, G., 2020. Metrics for multi-class classification: an overview. arXiv preprint arXiv:2008.05756.

[27] Tsoumakas, G. and Katakis, I., 2007. Multi-label classification: An overview. International Journal of Data Warehousing and Mining (IJDWM), 3(3), pp.1-13.

[28] Jain, V., Phophalia, A. and Mitra, S.K., 2022. HML-RF: Hybrid Multi-Label Random Forest. IEEE Access, 10, pp.22902-22914.

[29] Rapp, M., Mencía, E.L., Fürnkranz, J., Nguyen, V.L. and Hüllermeier, E., 2020, September. Learning gradient boosted multi-label classification rules. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases (pp. 124-140). Springer, Cham.

[30] Kramer, O., 2013. K-nearest neighbors. In Dimensionality reduction with unsupervised nearest neighbors (pp. 13-23). Springer, Berlin, Heidelberg

[31] Laaksonen, J. and Oja, E., 1996, June. Classification with learning k-nearest neighbors. In Proceedings of international conference on neural networks (ICNN'96) (Vol. 3, pp. 1480-1483). IEEE.

[32] Zebari, R., Abdulazeez, A., Zeebaree, D., Zebari, D. and Saeed, J., 2020. A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. Journal of Applied Science and Technology Trends, 1(2), pp.56-70.

[33] Abdi, H. and Williams, L.J., 2010. Principal component analysis. Wiley interdisciplinary reviews: computational statistics, 2(4), pp.433-459.

[34] Izenman, A.J., 2013. Linear discriminant analysis. In Modern multivariate statistical techniques (pp. 237-280). Springer, New York, NY.

[35] Hardoon, D.R., Szedmak, S. and Shawe-Taylor, J., 2004. Canonical correlation analysis: An overview with application to learning methods. Neural computation, 16(12), pp.2639-2664.

[36] Franc, V. and Hlavác, V., 2002, August. Multi-class support vector machine. In 2002 International Conference on Pattern Recognition (Vol. 2, pp. 236-239). IEEE.

[37] Song, Y.Y. and Ying, L.U., 2015. Decision tree methods: applications for classification and prediction. Shanghai archives of psychiatry, 27(2), p.130.

[38] Safavian, S.R. and Landgrebe, D., 1991. A survey of decision tree classifier methodology. IEEE transactions on systems, man, and cybernetics, 21(3), pp.660-674.

[39] Oshiro, T.M., Perez, P.S. and Baranauskas, J.A., 2012, July. How many trees in a random forest?. In International workshop on machine learning and data mining in pattern recognition (pp. 154-168). Springer, Berlin, Heidelberg.

[40] Ramchoun, H., Ghanou, Y., Ettaouil, M. and Janati Idrissi, M.A., 2016. Multilayer perceptron: Architecture optimization and training.

[41] Visa, S., Ramsay, B., Ralescu, A.L. and Van Der Knaap, E., 2011. Confusion matrix-based feature selection. MAICS, 710(1), pp.120-127.

[42] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J., 2011. Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 12, pp.2825-2830.