國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

具備多重學習分類器的智慧型軟體測試

Intelligent Software Testing with Multiple Learning Classifiers

詹哲彰

Jhe-Jhang Jhan

指導教授：王凡 博士

Advisor: Farn Wang, Ph.D.

中華民國 100 年 6 月

June, 2011

國立臺灣大學碩士學位論文
口試委員會審定書

具備多種學習分類器的智慧型軟體測試

Intelligent Software Testing with Multiple Learning Classifiers

本論文係詹哲彰（學號 R98921077）在國立臺灣大學電機工程學系完成之碩士學位論文，於民國 100 年 6 月 29 日承下列考試委員審查通過及口試及格，特此證明。

口試委員：[illegible]（簽名）
（指導教授）

[illegible]　　陳郁方

戴顯權

顏嘉志

郁方

系主任 顏嗣鈞（簽名）

誌謝

本論文能順利完成，首先要感謝這兩年來辛苦提攜與指導我們的王凡教授，於求學期間提供了良好的研究環境，以及不厭其煩的指導與教誨，令學生在探索研究之路受益良多。感謝口試委員戴顯權教授、郁方教授、陳郁方博士、張茂榮組長以及顏嘉志主任提供寶貴的意見，使得論文能更加完善。

接下來，要感謝在電機所兩年裡一起奮鬥的好夥伴們：胡揚、哲榮、力瑋、雅蘭以及淑芬，無論是在課業上還是生活上，都給予了我很大的幫助。同時也謝謝實驗室的助理、學弟們以及研究所期間認識的好朋友們，謝謝你們帶給了我豐富的兩年碩士生活。

謝謝國中同學們、高中同學們、大學同學們，以及所有幫助過我、關懷過我的人，致上由衷感謝。

謝謝婉齡這段日子以來包容我的壞脾氣，無時無刻的給我關懷與鼓勵，能夠體諒我在忙碌的時候無法陪妳。妳的貼心與支持，是我進步的原動力。

最後，特將本文獻給我最敬愛的父母，感謝您們無怨無悔的養育與無時無刻的關懷照顧，還有弟弟順安平日在家時，順便幫我準備飯菜，替我能省下不少瑣碎時間，讓我能無後顧之憂的專注於課業研究中，順利完成學業，願以此與家人共享。

中文摘要

測試在開發軟體的過程中佔了非常重要的一環，但是隨著時代的進步，硬體發展快速，軟體也隨之成長，傳統的純粹依賴人力來做軟體測試也越來越花時間，取而代之的是，結合機器學習來做軟體測試。

在本篇文章中，我們提供了一個結合支持向量機、類神經網路與 L*等機器學習演算法的多重分類器結構，這個結構主要分成三個部分：第一部分是提供使用者一些函式，藉由將這些函式安插在程式中，讓使用者可以收集程式執行時的資訊，當然這些資訊也可以由使用者自己提供；第二部分是將第一部分收集來的資訊，分成訓練資料以及學習資料。其中，訓練資料用來建立模型而測試資料用來測試模型的準確性。最後，我們把這些模型產生的結果利用一些組合函式產生出更好的結果。

我們主要研究如何在沒有規格的情況下，使用這些機器學習技術來正確產生程式的測試準則，並在文章最後以兩個網路上的共享資源程式來做實驗，比較不同機器學習演算法以及使用組合函式之後的準確性。

關鍵字：軟體測試，機器學習，多重分類器，支援向量機，類神經網路，L*，測試準則

ABSTRACT

Testing is an essential process of software development. Along with the progress of scientific and technological development of hardware, software systems become larger and larger. It is time consuming to do software testing with manpower traditionally. Using machine learning in place of labor efforts is getting more and more attractive.

In this thesis, we present a multiple-classifier structure of software testing with machine learning algorithms including support vector machine (SVM), neural network (NN) and L*. The structure is composed of three phases. In the first phase, the structure uses some functions which can insert into user programs to collect useful information as our input data during program executions. Those input data, of course, could also be provided by users. In the second phase, we separate the input data into training data and testing data. The training data is used for building models and the testing data is for testing the models we build. The last phase is that we combine the results of models to generate a better result by using combinatorial functions.

We investigate how to use machine learning techniques to automatically construct test oracles for programs without reliance on explicit specifications, and experiment with two open-source benchmarks to compare the accuracy of different learning algorithms and the accuracy after using combinatorial functions in the end.

Keyword: software testing, machine learning, multiple classifier, SVM, NN, L*, test oracle

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

Machine learning, a subfield of artificial intelligence, is an active research field in Computer Science. It is concerned with design and development of algorithms and statistical data. The algorithms allow computers evolving these behaviors based on statistical data; that is, extracting the part we are interested in from the past data or experience as training data (or feature selections [1]), building a model and defining different parameters. Then let model execute as programs, and use a useful algorithm to tune the parameters based on those training data. After learning is done, we could use this model to do some predication.

The training data are composed by input and output values. We could divide the learning process into regression analyses and classification according to the output value. The output of regression analyses is a continuous value. For example, predicting the number of stock index in the future is a kind of regression analyses. The output of classification is a lot of numbers of integers such as *{-1, 1}*. For example, predicting the score is pass or fail.

1.1 Motivation

Since the cost of software testing has escalated because of the increasing complexity of current systems, how to reduce the cost is an important issue. There are a lot of methods proposed to solve this problem. One of them is machine learning. There have been several researches to solve the software testing by using machine learning algorithms. [2] uses support vector machine to generate a test oracle in black-box testing. [3] uses the technique bootstrapping in machine learning to promote the prediction accuracy of the model they build. [4] provides a better input to software testing by using machine learning. [5] uses EXIST (Exploration eXploitation Inference for Software Testing) which is developed by EDA (Estimation of Distribution Algorithm) and online learning to maximize the number of distinct feasible paths of the process. There are many machine learning algorithms which could be used in different kind of software testing. However we don't know what is the best choice for a certain kind of software testing. We build a framework which contains several machine learning algorithms and compare their performance in software testing in order to find the most suitable machine learning algorithm for a certain kind of software testing.

1.2 Research Goal

We want to find an efficient way to reduce the cost of software testing by applying efficient machine learning algorithms, and combine them to generate a better result. To do this we propose a multiple-classifier structure of software testing with machine learning algorithms, such as support vector machine (SVM) [6], neural network (NN) [7], and L* algorithm [8]. We separate our input data into two sets: a training set and a testing set. The training set is used by the machine learning algorithms to generate the models as our classifiers to predict unknown input data. The testing set is used to test the performance of the classifiers. The training data and testing data are gathered from public resources.

1.3 Contribution

The main contribution is constructing a simple framework which contains three machine learning algorithms as classifiers. We make it easy to use, and compare performance of each machine learning algorithm. Then we combine the result of each classifier to generate a new result by using some combinatorial functions. We could find out that the prediction accuracy of combinatorial function is better than the result of single classifier.

Without specifications, SVM and NN could use input and output values to do software testing. However, L* would ask some problems that need a teacher to tell the answers. In the past, the answer is got through the execution of SUT or the specification of SUT. Since there is an overhead in execution of SUTs, and no specification of SUT, we propose a flowchart instead. The flowchart uses SVM/NN as the teacher of L* algorithm to get fault models. Since we do not have the visualization of SVM/NN model, we use the visualization of fault models as the visualization of SVM/NN models. Therefore, the other contribution is that we design the flowchart of L* training to build the fault model and do software testing.

1.4 Thesis Framework

The remainder of this thesis is organized as follows: Chapter 2 reviews the related work on multiple-classifier systems. Chapter 3 we give the background knowledge of support vector machine (SVM), neural network (NN), L* algorithm and unified modeling language. Chapter 4 describes the structure of software testing with multiple learning classifiers. Chapter 5 presents the details of the structure implementation. Chapter 6 shows the experimental result with two open-source programs. Chapter 7 gives conclusion and some directions of future work.

Chapter 2 Related Work

Multiple-classifier system is getting more and more popular due to their ability to combine the output of each classifier into a better result [9]. At present, there are many researches about how to design the multiple-classifier system. A successful design of multiple-classifier system is related to structure and combinatorial methodology.

[10] classifies current different structures of multi-classifier systems into three types. They are cascading, parallel, and hierarchical. In the cascading classifier, the result of a classifier is used as the input of the next classifier. In the parallel classifier, all of the classifiers are executed in parallel, and then the results of classifiers are used to obtain a new result by using combinatorial function. Hierarchical classifiers are a combination of cascading classifier and parallel classifier.

[11] classifies different structures of multi-classifier systems in more detail. They are conditional, hierarchical, hybrid, and multiple (parallel) topologies. Conditional topology first selects the result from any classifier as the final result. If the result is not correct, another classifier is selected. All the selection is random. Hierarchical topology is similar to conditional one. The only difference is the selection has priority. Hybrid topology selects the best result of classifiers as the final result. Parallel topology is as same as the parallel classifier in [10].

[12] classifies current combinatorial methodologies into four types. The first type is linear combination method. It uses linear functions such as summation and product to do combination. The second type is non-linear combination method. It uses rank based classifiers to generate final result such as majority voting. Majority voting selects the classifier which gets the highest vote to generate final result. The third type is statistical.

It uses some probabilistic functions such as Bayes rule. The last one is computationally intelligent. It uses machine learning algorithm such as genetic algorithms for combination.

All the outputs of the multiple classifier mechanisms we described above are continuous numbers, but we use discrete numbers in our experiment. Because we create the test oracle about the pass and fail problem, the output number we define here is only -1 and 1. The mechanisms we described above may not suitable for our experiment, so we do a little modification to fit our experiment.

Chapter 3 Background

3.1 Support Vector Machine

Support vector machine (SVM) proposed by Vapnik and Corinna Cortes in 1995 [6], is a kind of learning method which is widely used in solving the problem of classification or regression. The basic idea is given a group of data in a high- or infinite-dimensional space R^d, where $d \in N$. finding a hyper-plane to separate those data into two parts. After that, if there is a new data, SVM could map it into the same space and predict which category it belongs to based on which side of the hyper-plain it falls on.

For example, there are people's profiles, and we want to know which party he belongs to. We pick the place as our training data and transfer it into longitude and latitude such as the graph show in Fig. 3.1. Then we use SVM to find a line which divides the map into two sides like the graph shown in Fig. 3.2. Suppose left hand side is the Democratic Party and the other side is the Republican Party. After training, we can use the model generated by SVM to predicate result. Suppose there is a new profile, then we could locate the place in the map. If it locates at left hand side, then we predict that he belongs to the Democratic Party. Otherwise he belongs to the Republican Party. The graphs in Fig. 3.1 and Fig. 3.2 are drawn by the tool provided by C.C. Chang [14].

Because SVM finds the hyper-plane which has the maximum margin of input data in different categories, it builds only one solution and is a global optimal solution. The margin is defined by the sum of the shortest distances from the closest input data in different category to the hyper-plane.

If you want to know more about SVM we refer you to [6], [13] for more details.

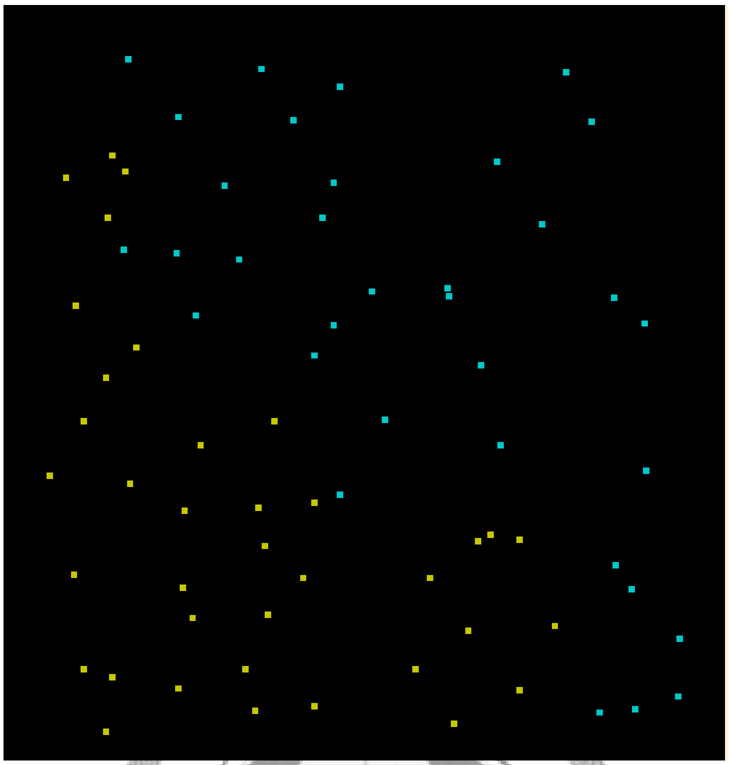


Fig. 3.1　Before Training

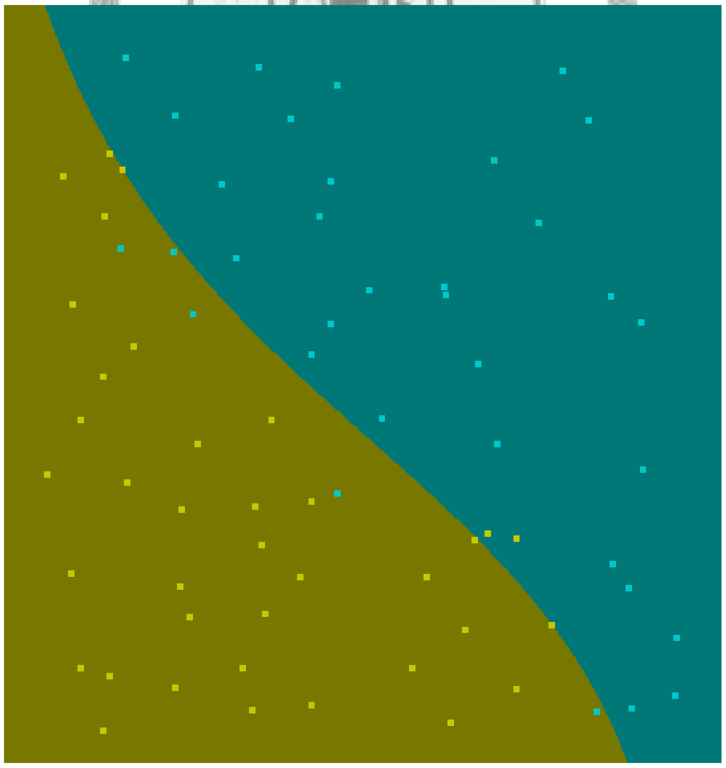


Fig. 3.2　After Training

3.2 Artificial Neural Network

Artificial neural network [7] is a kind of statistical learning method which is connected by artificial neurons. The model imitates the behavior of neuron network of organism in Nature. Each neuron represents a specific function called activation function and each transition between two neurons has a weight on it. The output of node depends on the activity function and weight. Fig. 3.3 is an example of graphical neuron. It has a set of input data *I* and a combine function *c* to merge those input data with a set of weights *W* corresponded to input *I* and bias *b* as the input of activity function *F*. Then it generates a specific output *O*.

Most neuron network algorithms constitute their learning structure as three layers. They are an input layer, one or more hidden layers and an output layer. Fig. 3.4 is the layer structure composed by neurons. Note that the node number of each layer may not be equivalence.

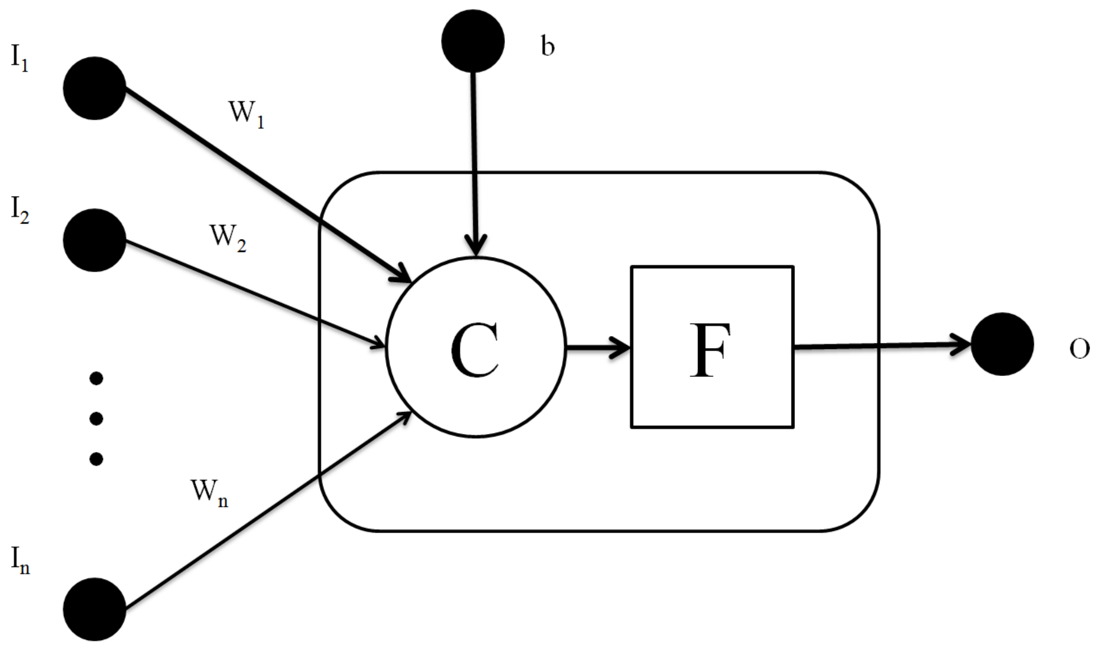


Fig. 3.3 Neuron

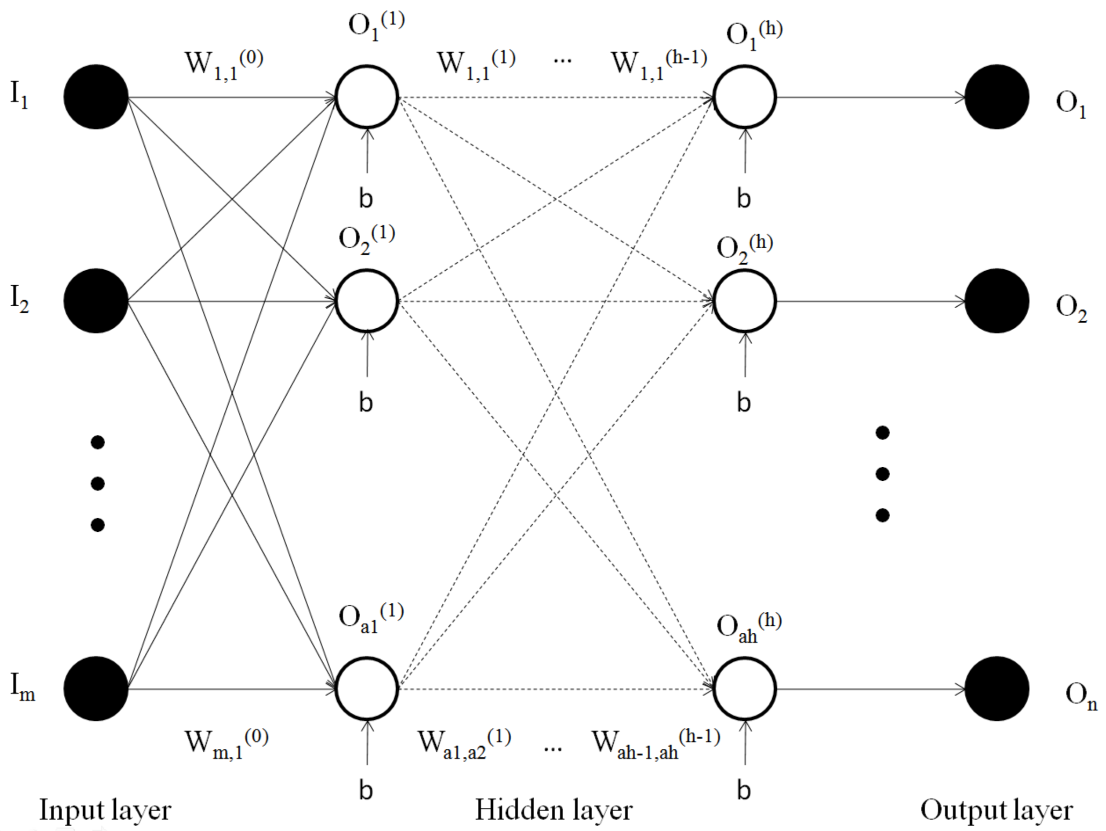


Fig. 3.4 Neuron Network of Multiple Layers

3.3 L* Algorithm

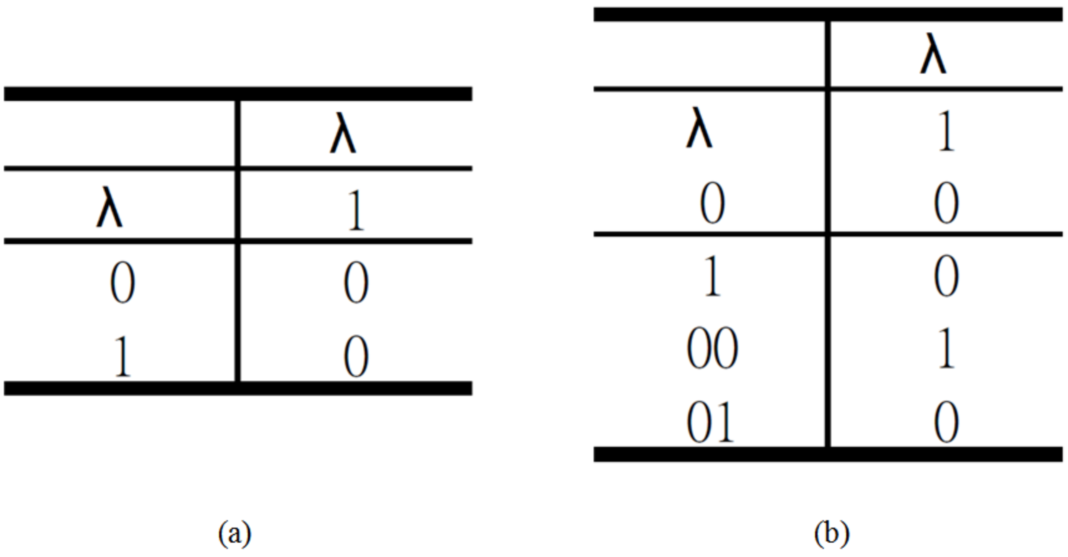
In 1987, Dana Angluin proposed an algorithm, L* [8], to learn from queries and counterexample. There are two different kinds of queries called membership query and equivalence query, and there is a teacher to answer these two queries. On a membership query, the algorithm would ask the teacher whether the string *s* is accepted or not. On an equivalence query, the algorithm would ask the teacher if the hypothesis model $\hat{M}$ which we construct is equivalence to the model M which we want to learn. If the answer is no, the teacher would return a counterexample. The learning structure is stored by observation table.

The row of observation table can be divided into two sections. One of the sections is the state section *S* and this part locates at the upper part of table, and every unique row of the upper part of the table represents a state from the DFA. The other part is transitions' section $S \bullet A$, and this part locates at lower part of the table which defines the transitions of the DFA. A state machine can be constructed according to an observation table. The table's columns are labeled by experiments *E* for distinguishing states-rows. The observation table must satisfy two conditions. They are closed and consistent.

- The observation table is closed means for all transition *t* in bottom part of the table there exists an state *s* in top part of the table such that $row(t) = row(s)$.
- The observation table is consistent means for all state s_1 and s_2 in top part of the table such that $row(s_1) = row(s_2)$, for all *a* in alphabet *A*, $row(s_1 \bullet a) = row(s_2 \bullet a)$.

The basic steps to build up the correct observation table are:

- If the machine accepts the row label concatenated by the column label, a field in the observation table is true. On the other hand, if the machine rejects the row label concatenated by the column label, a field in the observation table is false. A membership query with the concatenated string decides if a field is true or false.
- The observation table is not closed, this means the transition contains a row at the lower part of table is different from the state contains all rows at the top part of the table. When this situation occurs, the row from the transitions' part which is different from the state contains all rows at the top part of the table added as a state row and corresponding transition rows are added. For example, in Fig. 3.5 table (a) is not closed since $row(0)$ is different from $row(\lambda)$. So L* chooses to move the string *0* to the top part of the table and then queries the strings *00* and *01* to build new table.



	λ
λ	1
0	0
1	0

(a)

	λ
λ	1
0	0
1	0
00	1
01	0

(b)

Fig. 3.5 Example of Non-closed Table

- The observation table is not consistent, this means two state rows are equal but their corresponding transition rows are different. When this situation occurs, in order to separate these two states, i.e. let the two inconsistent states have different results, Add an additional column to the observation table and the label of the new column is the label of an already existing column label concatenate one letter from the alphabet. For example, in Fig. 3.7, table T_2 is inconsistent since the *row(λ)* = *row(0),* but *row(λ0)* $\neq$ *row (00).* In order to make the table consistent a new column is added to the table, the label of the new column is *A* which is *λ* concatenate *A* from the alphabet, after that all free fields are queried.
- If a counterexample is found, add the counterexample and its prefixes into the top part of the observation table. Then add the corresponding transitions in the lower part of table.
- After the observation table is closed and consistent, we could use this table to build a DFA by creating a state for every unique state row. Every state has a transition for every possible letter from the alphabet. The transitions row, labeled equal to the

origins states label concatenated the transition’s letter, determines the destination state. The destination state is the state with the same state row as the mentioned transitions row.

Here runs a simple example for L* algorithm in order to show how the L* works Initially, L* asks membership queries for the strings *λ*, *0*, and *1*. The initial observation table T_1 is shown in Fig. 3.6. 0 means fail and 1 means pass. This observation table is closed and consistent, so L* builds the hypothesis model $\hat{M}_1$ shown in Fig. 3.6. We could easily find the counterexample *00* labeled as false.

	λ
λ	1
0	1
1	1

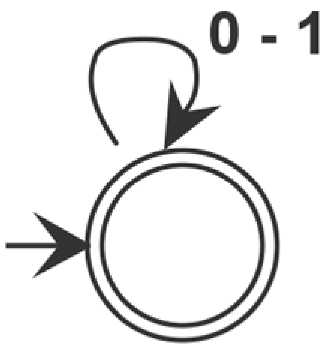


Fig. 3.6 Table T_1 and Hypothesis Model $\hat{M}_1$

To process the counterexample *00*, L* adds the strings *0* and *00* to top part of the table, and queries the strings *000*, *01*, and *001* to construct the observation table T_2 shown in Fig. 3.7. This observation table is closed but not consistent since $row(\lambda) = row(0)$ but $row(\lambda 0) \neq row\ (00)$.

Thus L* adds the string *0* to *E*, and queries the strings *0000*, *010*, *0010* and *10* to construct observation table T_3 shown in Fig. 3.8. This observation table is closed and consistent, so L* builds the hypothesis model $\hat{M}_2$ shown in Fig. 3.8.

	λ
λ	1
0	1
00	0
000	0
01	1
001	0
1	1

Fig. 3.7 Table T_2

	λ	0
λ	1	1
0	1	0
00	0	0
000	0	0
01	1	1
001	0	0
1	1	1

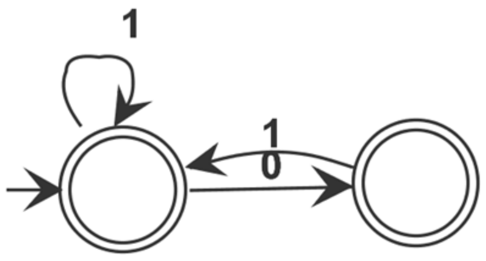


Fig. 3.8 Table T_3 and Hypothesis Model $\hat{M}_2$

We could easily find the counterexample *11* labeled as false. L* responds to this counterexample by adding the strings *1* and *11* to top part of the table then queries the strings *100*, *110*, *111*, *1100*, and *1110* to construct the observation table T_4 in Fig. 3.9. This table is found to be closed but not consistent, since $row(\lambda) = row(1)$ but $row(\lambda 1) \neq row\ (11)$.

	λ	0
λ	1	1
0	1	0
00	0	0
1	1	1
11	0	0
000	0	0
01	1	1
001	0	0
10	1	0
110	0	0
111	0	0

Fig. 3.9 Table T_4

	λ	0	1
λ	1	1	1
0	1	0	1
00	0	0	0
1	1	1	0
11	0	0	0
000	0	0	0
01	1	1	0
001	0	0	0
10	1	0	1
110	0	0	0
111	0	0	0

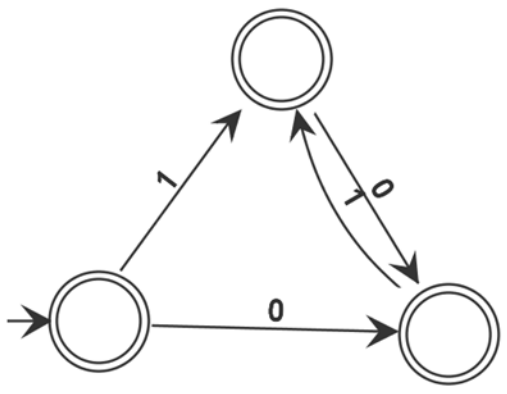


Fig. 3.10 Table T_5 and Hypothesis Model $\hat{M}_3$

Thus L* adds the string *1* to *E* and queries the strings *0001*, *011*, *0011*, *101*, *1101*, and *1111* to construct the observation table T_5 in Fig. 3.10. This table is closed and consistent, so L* builds the hypothesis model $\hat{M}_3$ shown in Fig. 3.10. And L* terminates with hypothesis model $\hat{M}_3$ as its output.

3.4 Unified Modeling Language

Unified Modeling Language (UML) [15] proposed by Grady Booch, Ivar Jacobson, and Jim Rumbaugh in 1996, is a specification language that is used in the software engineering field to describe the behavior of the target system from an external point of view. Now, the standard is managed, and was created by, the Object Management Group (OMG). And the latest version is UML 2.2 which has 14 types of diagrams. We will make use of use case diagram in this thesis.

Use case diagrams overview graphically the usage requirements (use cases) for a system, actors, and any relationship between those use cases. This use case diagram can only give the most basic view of a use case or a collection of use cases. So it can’t be used to define the function of use cases. Here are some the components of use case diagram.

- **Actor** in Fig. 3.11 is drawn as a human shape graph and it means a participant in the system, which maybe a people, a system or a virtual thing such as time etc. Note that there is no interaction among actors in the use case diagram.

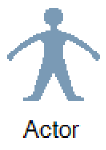


Fig. 3.11 Actor

- **Use case** in Fig. 3.12 is drawn a horizontal oval and a sequence of actions performed by system. It provides something of measurable value for actors.

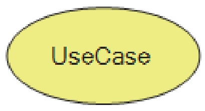


Fig. 3.12 Use Case

- **System box boundary** in Fig. 3.13 is displayed as rectangle around the use cases to indicate the scope of system.

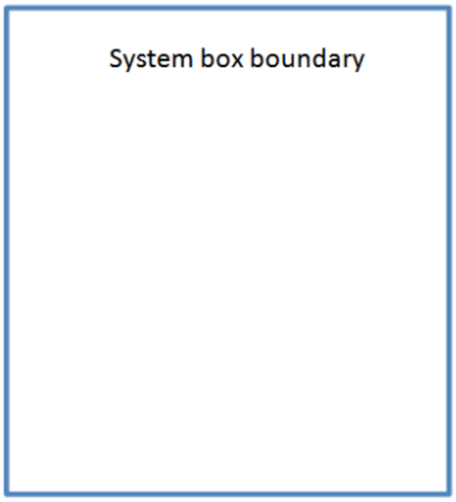


Fig. 3.13 System Box Boundary

- **Include** relationship in Fig. 3.14 between two use cases implies the behavior of the included use case will be inserted into the behavior of the including one. It is drawn as a dash arrow from including use case to included one, with label «include». The function is like a macro expansion in program. Note that the included use case is **always** required for the including use case. It means the included use case is not optional and must be executed. For example, you may not need to login when browsing websites of the Youtube.

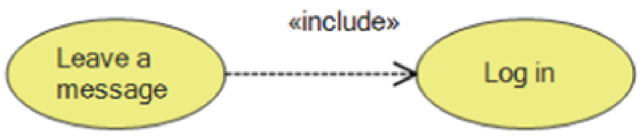


Fig. 3.14 Include Relationship

- **Extend** relationship in Fig. 3.15 between two use cases implies the behavior of the extending use case may be insert into the behavior of the extended one. It is drawn as a dash arrow from extending use case to extended one, with label «extend». Note that the extended use case **may** be required for the including use case. It means the extended use case is optional and may be executed. For example, you must need to login when you want to upload a video in the Youtube.

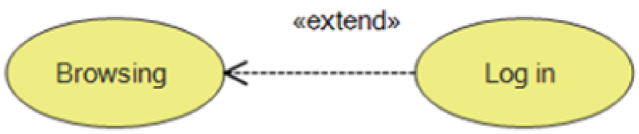


Fig. 3.15 Extend Relationship

- **Association** relationship in Fig. 3.16 between actor and use case is represented as solid line with an optional arrowhead which implies the direction of control flow. It exists whenever an actor is participated in a behavior in the use case.

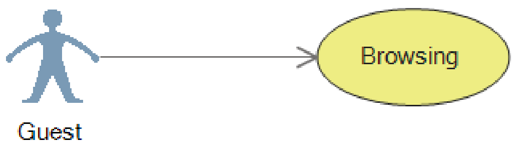


Fig. 3.16 Association Relationship

- If two use cases or two actors have common behaviors, we just need to describe the common once and describe any difference in another case or actor. For example, In website of the Youtube, both Guest and Member can browse videos, but only member can upload videos. So the meaning of **generalization** relationship in Fig. 3.17 between two use cases or between two actors is to present the situation describe above. It is a solid line ending in a hollow triangle drawn from the common to the customized use case.

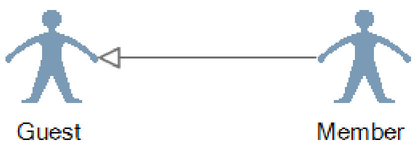


Fig. 3.17 Generalization Relationship

In Fig. 3.18, we use the components mentioned above to describe a simple message board. There are two actors, guest and member, in simple message board. Both of them can browse the message, but only member can leave a message. You must login when leave a message.

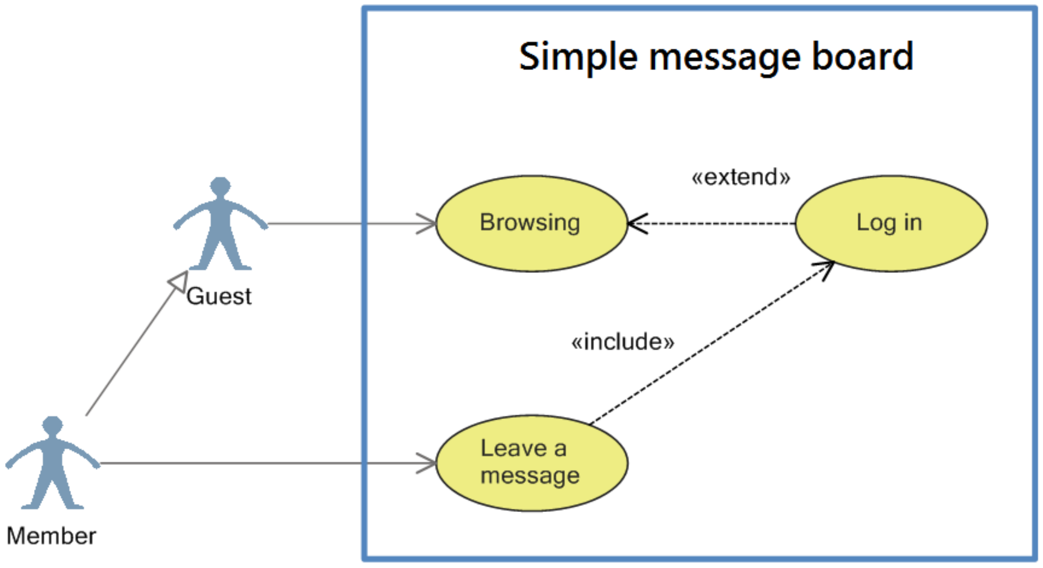


Fig. 3.18 Example of Use Case Diagram

Chapter 4 Software Testing with Multiple Learning Classifiers

Since software testing is getting more time-consuming, how to reduce the cost is becoming an important issue. Many solutions has be proposed, one of them is machine learning. Here we use three machine learning algorithms to solve software testing problems.

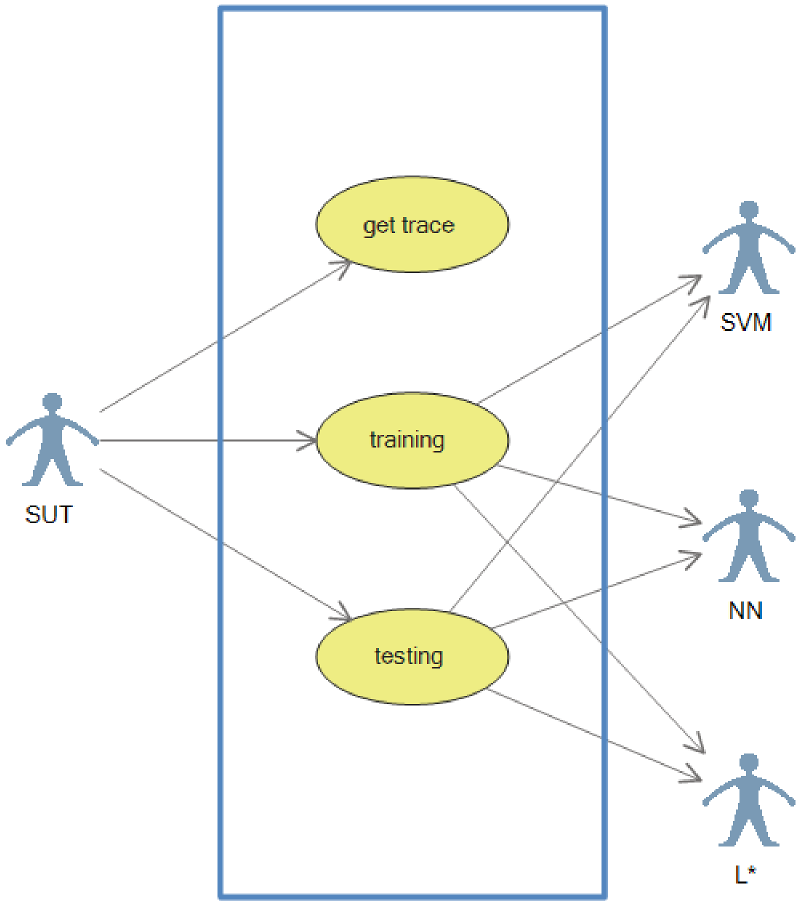


Fig. 4.1 System Structure

4.1 The Usage of System Structure

Fig. 4.1 is the whole system structure we build. We make use of use case diagram to overview the usage requirements. There are three main use cases in our system structure. They are get traces, training and testing respectively. In the SUT, if we want to do software testing, first we need to collect training traces and testing traces. This is the usage of get traces. After we get traces, we need to select algorithm to train the model, this is the usage of training and there are three algorithms for training. After training, we need to test the accuracy of the model, this is the usage of testing and there are three algorithms to chose as same as training.

4.2 Workflow of System Structure

Our research workflow can be divided into three parts according to the usages. The first part is collecting the traces from SUT which is shown in Fig. 4.2. After that, there should generate a trace file and a dictionary which is corresponding to trace file. Then we translate the trace file and corresponding dictionary to different input of machine language. Due to different properties of machine learning algorithms, there are different ways to do translation. The input of L* algorithm is similar to the trace we collect, so we can translate it without feature extraction. The input of SVM and NN algorithms is a vector which represents different features. So we need to do feature extraction to get the input files of SVM and NN algorithms. After translation, we can get input files of different machine learning algorithms. The input file is composed of training file and testing file. And we generate a model by using training file to do training which is shown in Fig. 4.3. Then use the model and testing file to do testing which is shown in Fig. 4.4 to get result.

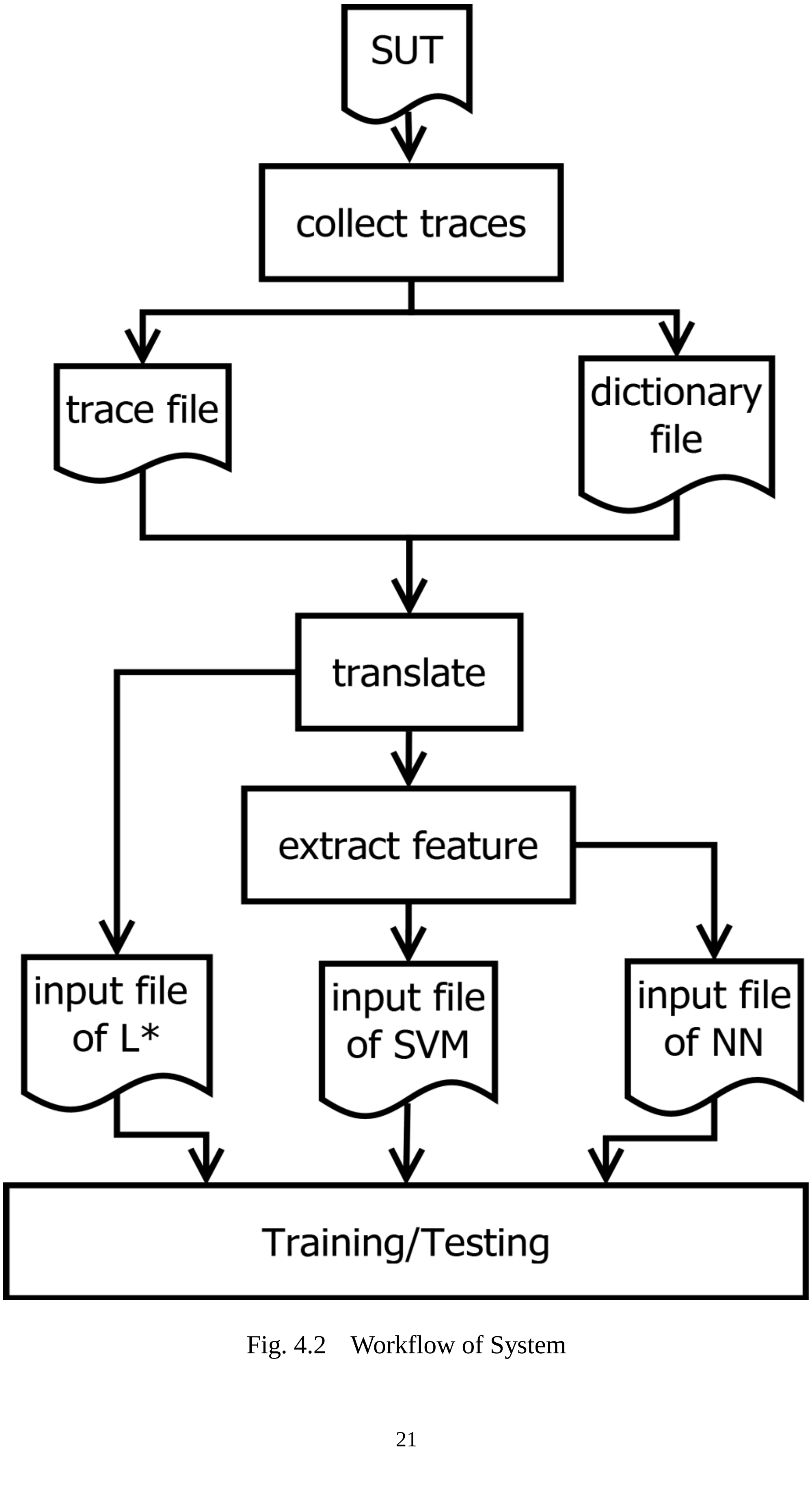


Fig. 4.2 Workflow of System

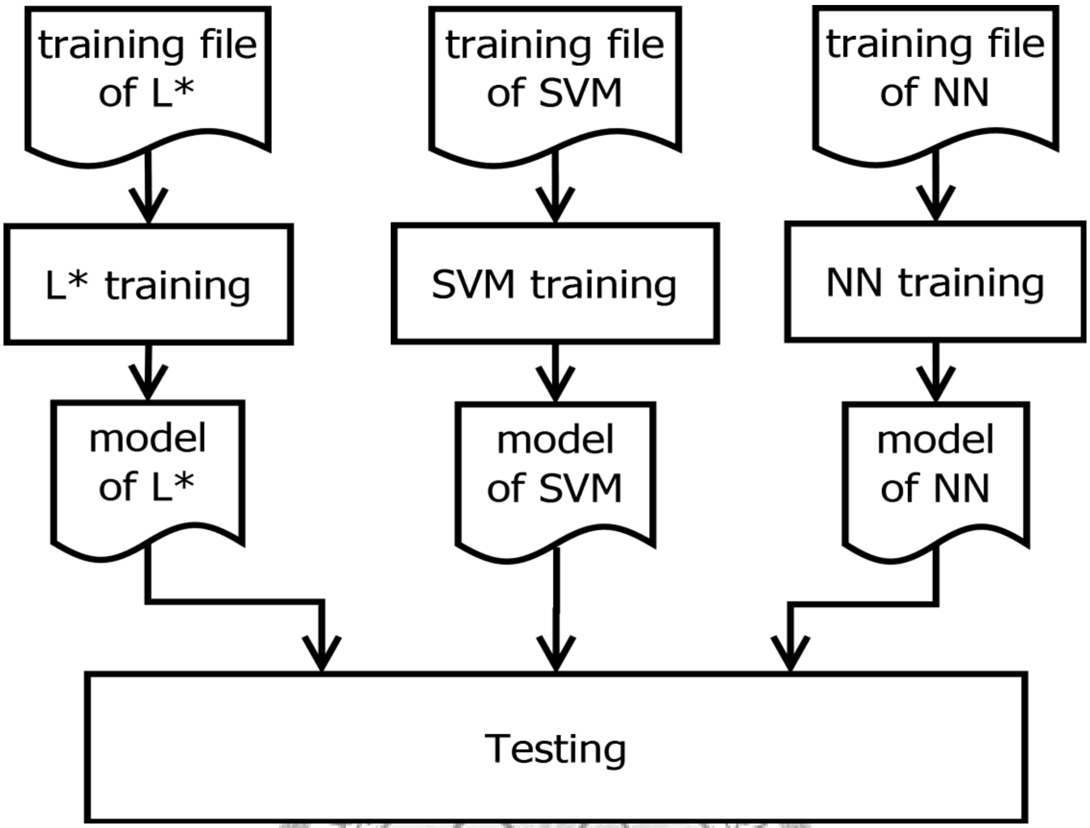


Fig. 4.3 Workflow of Training

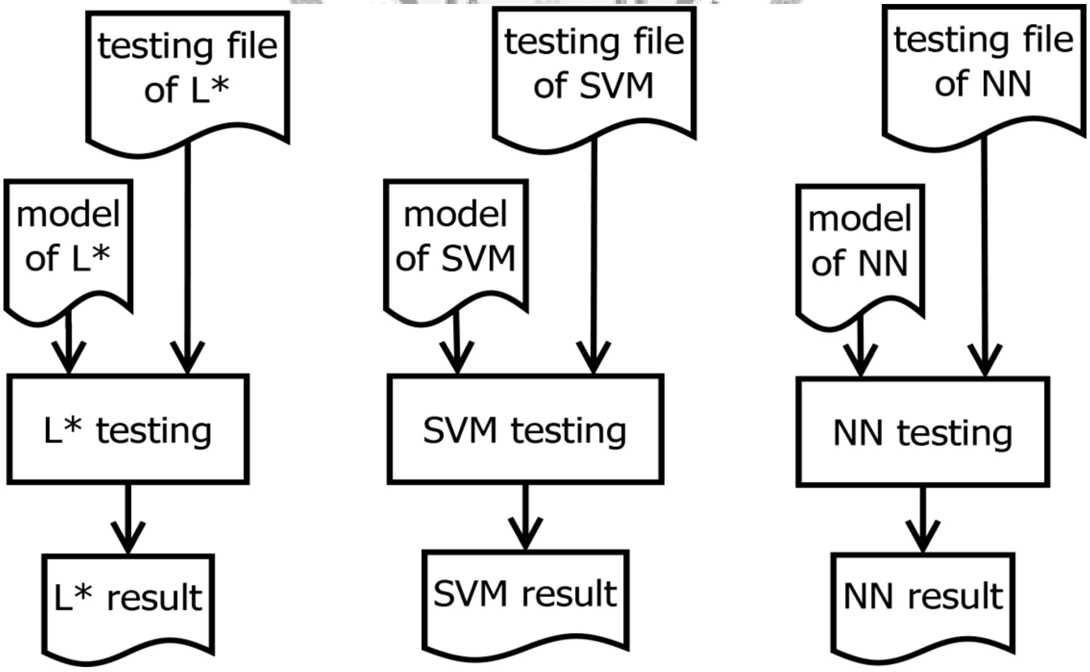
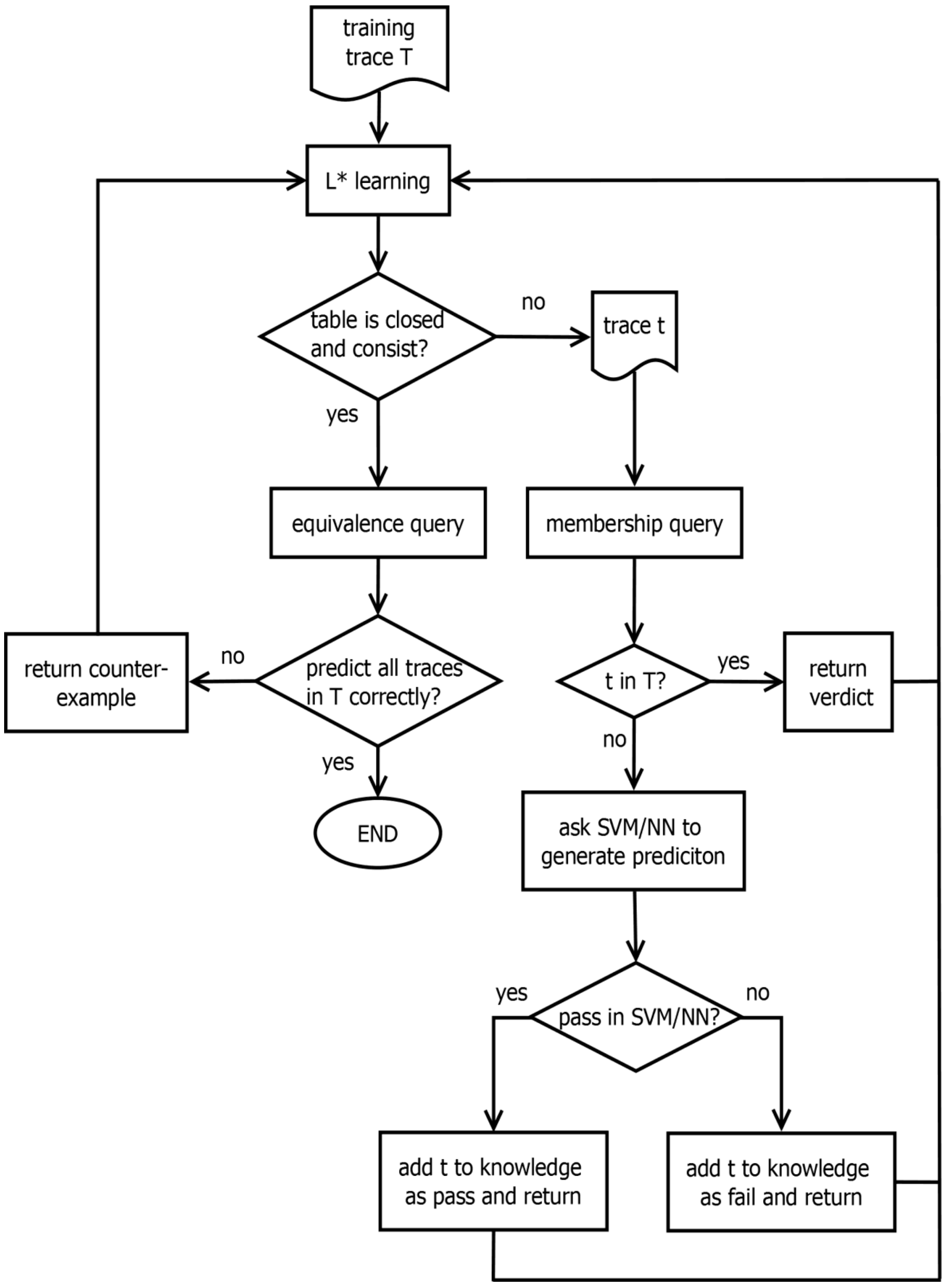


Fig. 4.4 Workflow of Testing in Different Machine Learning Algorithms

4.3 Workflow of L* training

Remember that L* needs a teacher to answer membership query and equivalence query we mentioned in Chapter 3.3. The teacher must be reliable, and usually there will be something as a teacher. For example, the specification or result from actual system execution. It is possible that there is an overhead while system execution or it is possible for lack of specifications, so we choose SVM/NN which is already learned as teacher of L*.

Fig. 4.5 is our workflow of L* training. First, L* algorithm will check whether the observation table is closed and consistent or not. If the observation table is not closed and consistent, in order to make the observation table closed and consistent, there is a trace set which the table want to know whether they are members or not, then L* will set membership query to ask whether each trace *t* in trace set is a member of the model. We let L* algorithm to check whether trace *t* is in the training trace *T* collected while system execution. If trace *t* is in training trace *T*, L* will return the verdict which is recorded in training trace *T*. Then we end up this membership query. If trace *t* is not in trace *T*, we make L* algorithm ask SVM/NN whether it is pass or not. If it is pass, L* will return pass and add trace *t* to its knowledge as pass, if it is fail, L* algorithm will return fail and add trace *t* to its knowledge as fail. If the observation table is closed and consistent, then we will set equivalence query to check whether the hypothesis model $\hat{M}$ is equal to system model M. Our method which is used for deciding the equivalence between M and $\hat{M}$ is checking whether the hypothesis model $\hat{M}$ could predict all traces in *T* correctly. If it could not predict all traces in *T* correctly, then we pick the trace which is predicted wrongly by $\hat{M}$ as counterexample and continues the L* learning. Otherwise, we terminate the L* training and return the hypothesis model $\hat{\mathrm{M}}$.



training trace T
L* learning
table is closed and consist?
no
trace t
yes
equivalence query
membership query
return counter-example
no
predict all traces in T correctly?
t in T?
yes
return verdict
yes
no
END
ask SVM/NN to generate prediciton
yes
pass in SVM/NN?
no
add t to knowledge as pass and return
add t to knowledge as fail and return

Fig. 4.5 Flowchart of L* algorithm

4.4 Multiple Learning Classifiers

[10] categorizes current different multi-classifier systems into three categories: Cascading, Parallel and Hierarchical. We use parallel classifiers in this thesis. The structure for parallel classifiers is shown in Fig. 4.6. This is the most common methodology used in multi-classifier systems. First, we operate all of the classifiers in parallel with the input data to get the corresponding prediction data. Then the results are obtained by using combinatorial function with those prediction data. Because all of the classifiers need to be executed to obtain the corresponding prediction data, this methodology incurs an overhead as it is time-consuming.

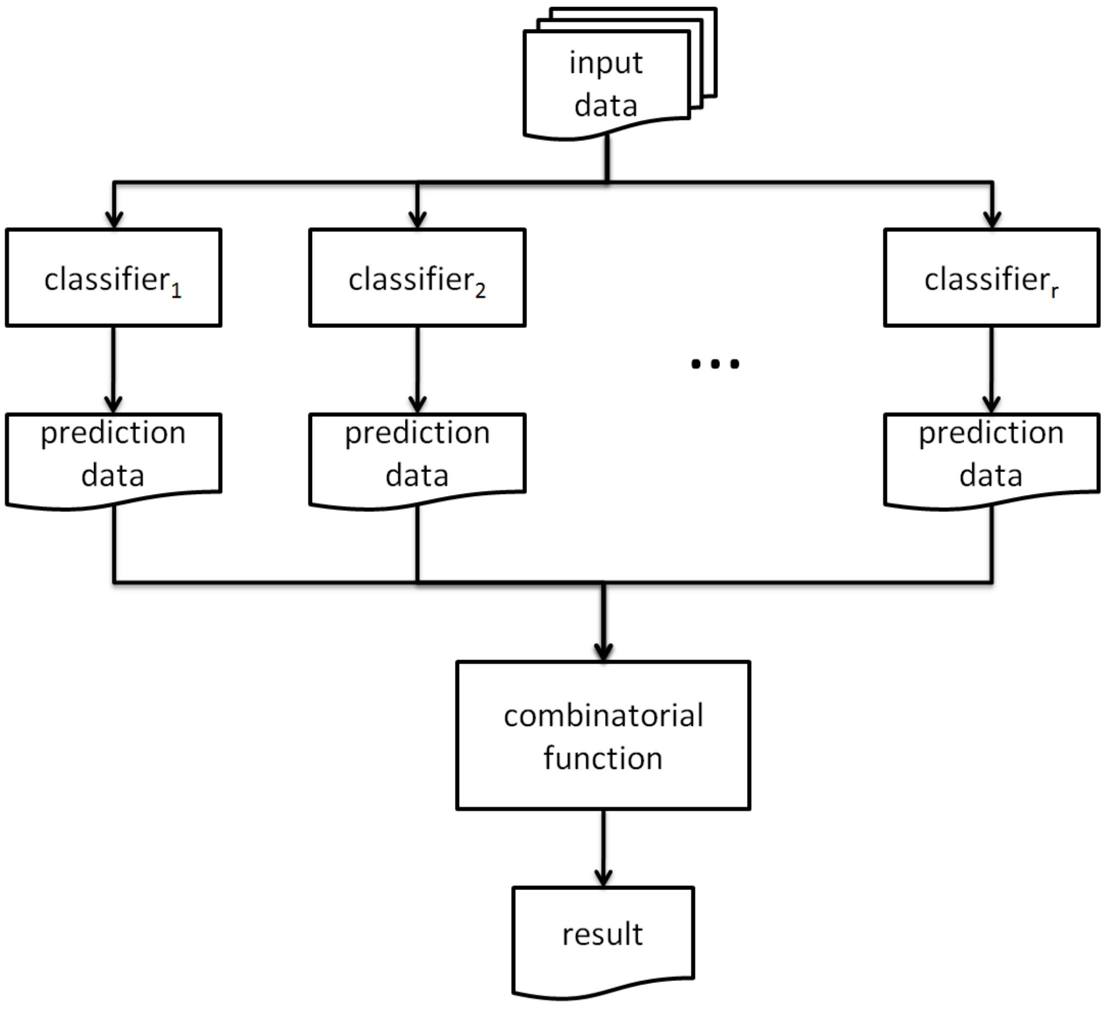


Fig. 4.6 Structure of Parallel Classifiers

The combinatorial function affects the result heavily. If we design a good combinatorial function, the system can reach better performance. However, an unsuitable combinatorial function may lead the system to generate poor performance. We present four combinatorial functions in the thesis. They are maximum, summation, weighted summation, and product.

Maximum is the simplest implementation of combinatorial functions. the prediction data of classifier with the highest accuracy is chosen as the output result of the system. If the prediction data of the classifiers are denoted by $C_i(x)$, where $i = 1, \dots, r$, then the output of the maximum combinatorial function is provided in formula (4.1).

$$f_{MAX}(x) = \max\{C_1(x), \dots, C_r(x)\} \tag{4.1}$$

Summation uses the summation operation to generate the result by adding the prediction data of all classifier in the system. The output of the summation function is defined in formula (4.2).

$$f_{SUM}(x) = \sum_{i=1}^{r} C_i(x) \tag{4.2}$$

Weighted summation is an advanced version of the summation. For each classifier C_i, it is assigned a weight w_i and the weight value can be varied. First, the weighted summation let the prediction data of each classifier C_i multiply its assigned weight w_i and then sum up all the obtained value. The function is described as formula (4.3).

$$f_{W-SUM}(x) = \sum_{i=1}^{r} C_i(x) w_i \tag{4.3}$$

We can see the formula (4.1) and (4.2) are special cases in formula (4.3). If we have the weight $w_i = 1$, for classifier C_i which has the highest accuracy in formula (4.3), then we could get the same result as formula (4.1). If we have the weight $w_i = 1$, for all i

= 1,…,r, then we get the same result as formula (4.2).

Product is similar to the summation, it multiples the values instead of summing up the prediction data of each classifiers. The product function is presented in formula (4.4).

$$f_{PROD}(x) = \prod_{i=1}^{r} C_i(x) \tag{4.4}$$

Chapter 5 Implementation

In this chapter, we describe the details about the implementation of our software testing with multiple learning classifiers. Our program is written in C and C++, so in this chapter, programs start in C or C++ fashion.

This work is implemented with off-the-shelf SVM library libsvm [14], NN library FANN [17], and L* library libalf [18]. Basically, these libraries provide complete functions about training and testing. All we need to do is modifying the function to fit our system framework.

5.1 How to Collect Trace

Because we are interested in the structure of system, we want to know the relationship of procedures in system. To get this goal, we use two functions from InTOL and insert them into the program. These two functions will help us to collect trace when system runs. They are:

- *InTOL_set_event(char event_type, const char* event);*
- *InTOL_assert(bool cond);*

The fuction *InTOL_set_event* has two parameters. Their types are *char* and *const char**. First parameter is event type, and there are two kinds of event type. They are input event and output event respectively. The other one is event which labels what event occurs, so the function *InTOL_set_event* records what kind of event happened. The function *InTOL_assert* has one parameter, and it is a *bool* type parameter which represents the rule the system must to obey. Once the program goes against the rule, the output trace must be failure. The test trace we collect is a sequence of events. In fact,

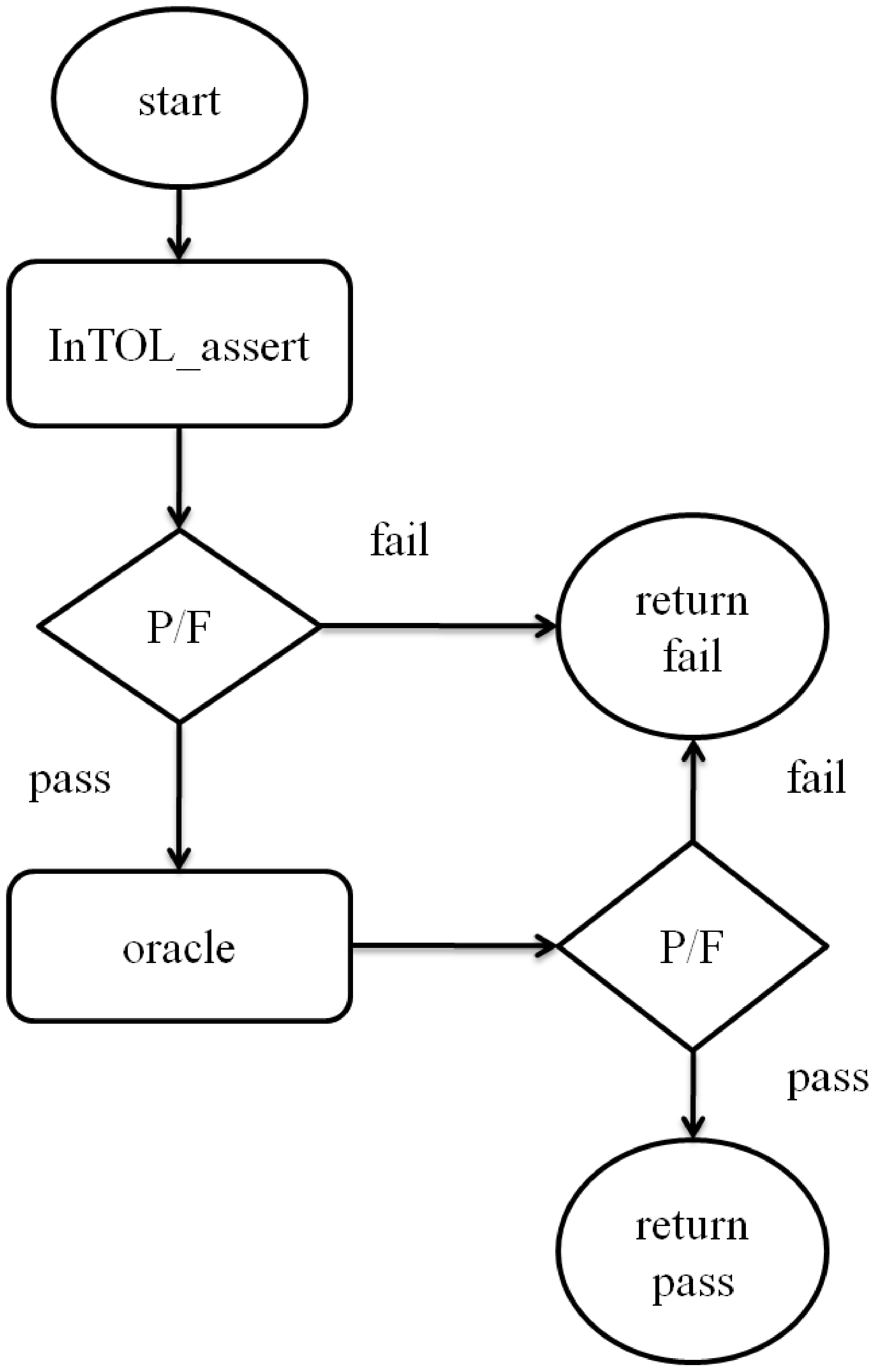


Fig. 5.1 Flowchart of Deciding Failure

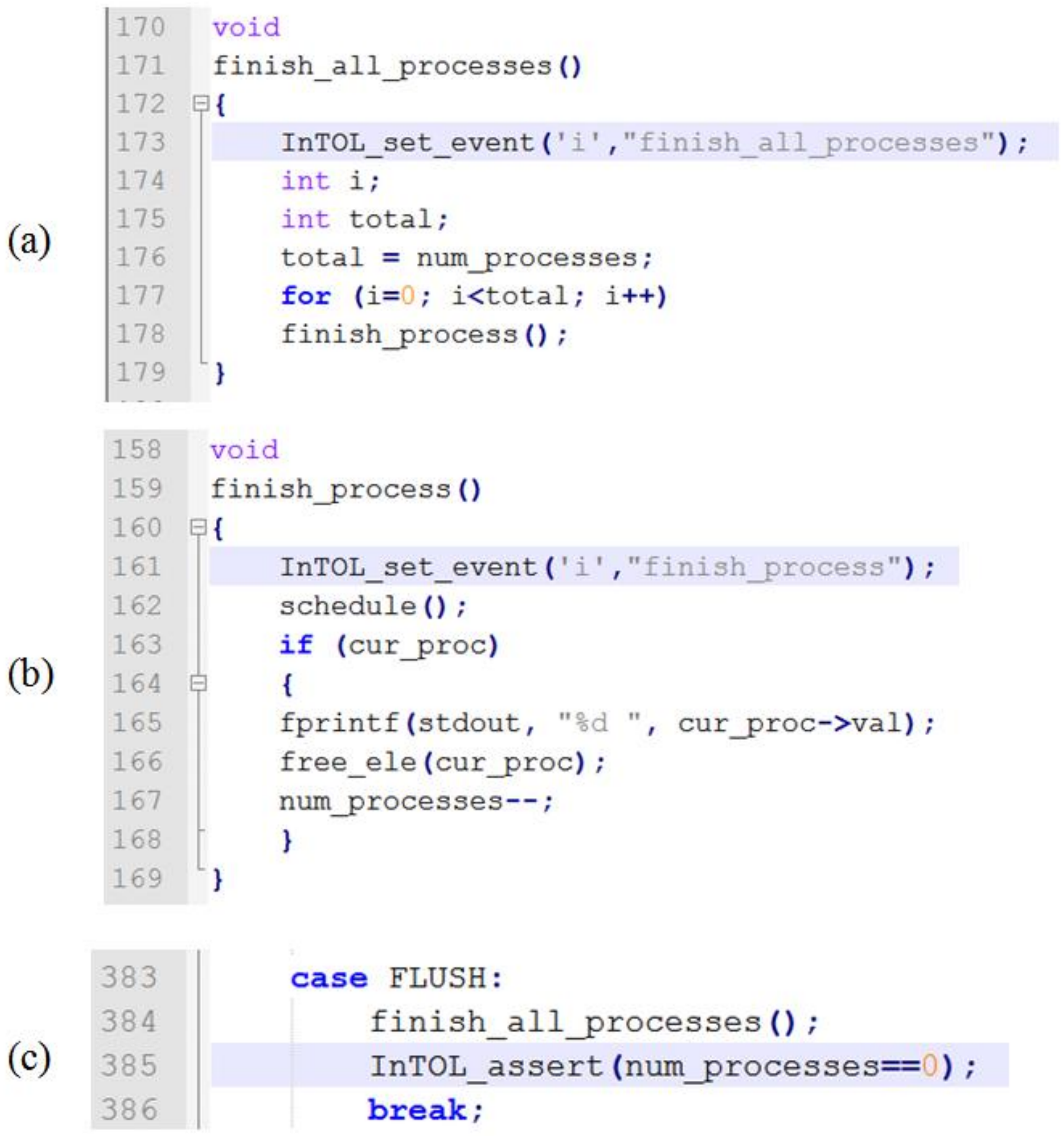
there is no bound to the lengths of the test traces. However L* learning time is related to the number of event and the length of test trace. The more number of event you use, the more time of L* training you cost.

5.2 How to Decide Failure

Fig. 5.1 shows the collected trace will be fail in two conditions. First, when function *InTOL_assert* returns fail message, the trace must be fail. For example, when system bumps into the function *InTOL_assert*, the function *InTOL_assert* says variable *a* must equals to 1, but the variable *a* is 2 in practice. Second, the function *InTOL_assert* returns pass, but output of oracle is fail, and then it will be failure. The trace will be pass except these two conditions.

5.3 How to Insert Function

How to insert function? It depends on what purpose you need. For example, if we want to know the whole structure of system and the relationship of functions, we could insert code into every procedure. If we want to know the most important procedure or the procedure from revision recently, we could insert more functions to the procedure which we focus on than others. There is an example of insert function shown in Fig. 5.2. The three fragments of example are from same program. There are some processes working in the program. The total number of processes is recorded in global variable *num_process*. When system executes case *FLUSH* in line 383. The function *finish_all_processes* will be called, and then it will call fuction *finish_process* to do their job in line162 for every process. After job done, the process must be free, and the global variable *num_process* will be decreased by 1. After these steps, the global variable *num_process* sould be 0, and we insert function *InTOL_assert* with the condition *num_process* equals 0 to check the relationship is correct.


(a)

```
void
finish_all_processes()
{
    InTOL_set_event('i',"finish_all_processes");
    int i;
    int total;
    total = num_processes;
    for (i=0; i<total; i++)
    finish_process();
}
```

(b)

```
void
finish_process()
{
    InTOL_set_event('i',"finish_process");
    schedule();
    if (cur_proc)
    {
    fprintf(stdout, "%d ", cur_proc->val);
    free_ele(cur_proc);
    num_processes--;
    }
}
```

(c)

```
    case FLUSH:
        finish_all_processes();
        InTOL_assert(num_processes==0);
        break;
```

Fig. 5.2 Fragment of Code

5.4 Feature Selection

After collect the trace from the method we mentioned above, we need to transfer these traces into the input type of SVM and NN. How to transfer is also an important issue. This issue is called feature selection. The machine learning algorithm could do better performance with a good feature selection. Here we use five feature sets X_0, X_1, X_2, X_3, and X_4. Then we concatenate them as an input vector. These five feature sets are explained in the following.

5.4.1 X_0

X_0 is a structure which records the last W events of trace, i.e. the X_0 records events from the tail of trace to the head of trace up to W. The W is a window size which means the biggest capacity that X_0 could record. For example, given $X_0 = \{x_0, x_1, .. , x_w\}$, and the event index of *a*, *b*, *c*, *d*, and *e* in dictionary are *index(a)* = 1, *index(b)* = 2, *index(c)* = 3, *index(d)* = 4, and *index(e)* = 5. If window size $W = 4$ and there is an trace $t = abcde$, then x_0 = *index(e)* = 5, x_1 = *index(d)* = 4, x_3 = *index(c)* = 3, and x_4 = *index(b)* = 2.

5.4.2 X_1

Given the event set E, and there is a trace t which is composed of a sequence events, each event belongs to E. For every event e_1 and e_2 in E, there exists a function $F(e_1, e_2)$ which means the number of e_1 occurrences without any e_2 event from the end of a trace t. i.e. the number of event e_1 since the last event e_2. For example, given a trace $t = abbcccaabc$, $F(c, a) = 1$, $F(b, a) = 1$, and $F(a, c) = 0$.

5.4.3 X_2

Given the event set E, and there is a trace t which is composed of a sequence events, each event belongs to E. For every event e_1 and e_2 in E, there exists a function $F(e_1, e_2)$ which means the max number of e_1 occurrences without any e_2 event in a trace t. i.e. the max number of event e_1 before the event e_2. For example, given a trace t =

abbcccaabc, *F(c,a)* = 3, *F(b,a)* = 2, and *F(a,c)* = 2.

5.4.4 X_3

Given the event set *E*, and there is a trace *t* which is composed of a sequence events, each event belongs to *E*. If there is an event *e* in *t*, then |*e*| means the number of event *e* in trace *t*. For every event e_1 and e_2 in *E*, there exists a function $F(e_1, e_2)$ which means the number of e_1 occurrences minus the number of e_2 occurrences in a trace *t*. i.e. $|e_1| - |e_2|$. For example, given a trace *t* = *abbcccaabc*, *F(c,a)* = 1, *F(b,a)* = 0, and *F(a,c)* = -1.

5.4.5 X_4

Given the event set *E*, and there is a trace *t* which is composed of a sequence events, each event belongs to *E*. For every event e_1 and e_2 in *E,* there exists a function $F(e_1, e_2)$ to check whether it is satisfied with all of the following three conditions or not:

- For every e_1 event occurrence in trace *t*, there must (transitively) follow an e_2 event occurrence.
- For every e_2 event occurrence in trace *t*, there must be (transitively) followed by an e_1 event occurrence.
- There must be an e_2 event occurrence between two e_1 event occurrences.

For example, if trace *t = abcaabc, F(b, c)* is *true* and *F(a, b)* is *false*.

Chapter 6 Experiment

We use the two benchmarks (SUT) which are *TCAS* with bug version 10 and 28 and benchmark *schedule* with bug version 3 in SIR [16] to demonstrate our technique. We present the experimental studies of different training data sizes with five different classifiers which are L* with teacher SVM, L* with teacher NN, SVM, SVM trained with optimal parameters and NN. The performance of constructed test oracle is measured with prediction accuracy and time cost. The prediction accuracy is the percentage of correctly labeled test cases of a testing data set with the constructed test oracle. The time cost is the used time to train the test oracle and testing data. The experimental data are collected on Intel(R) Core(TM) i7 CPU 860@ 2.8GHz with 2G RAM, running on Ubuntu 9.10.

To objectively demonstrate the effect of different training data size, we have the prediction accuracy to test with a testing data set of 200 test cases. Each experiment is run for 10 times and the average performance data is recorded. And has time limit which is 30 minutes. Because the way to build model of L* with teacher NN or SVM is different from the way to build model of NN and SVM. Their prediction accuracy is totally different when it is time out. That is, L* would generate hypothesis model during training. If it is time out, we could still use the hypothesis model to do prediction. So the prediction accuracy of L* would not be 0. However the model of NN and SVM is generated after training, if it is time out, there is no model for testing. The prediction accuracy will be 0.

6.1 Experiment Setting

Following are the settings of each algorithm mechanism we use in this experiment:

1. NN: Here we use multiple layer structure. The number of neutron in input layer is as same as the element number in vector. The number of hidden layer is 3 and each hidden layer has 3 neurons. Because we just need to know the output is pass or fail, the number of neutron in output layer is 1.
2. SVM: the kernel we use is radial basis function (RBF) with no parameters.
3. SVM with optimal parameter: the kernel we use is radial basis function (RBF). And use the procedure which is provided by *libsvm* to find suitable parameters.
4. L* with teacher NN: Here we use the NN model from 1 to be the teacher of L*.
5. L* with teacher SVM: Here we use the SVM model from 3 to be the teacher of L*.

6.2 Results of Each Classifier

Following we list the result of experiments in table type. The first column of table is the size of training data set, the second one is prediction accuracy, and the last one is average execution time.

The experiments of benchmark *TCAS* is from Table 6.1 to Table 6.5. We could find several things.

- When the size of training data set increases, the prediction accuracy will also increase. However there is an exception in Table 6.4, we could find the time cost in experiment of size 200 and 1000 is time out. This means the leaning is not complete, so it could not promote accuracy to 100%. We believe it may be 100% while increase time limit.
- It will cost much time with increment of size, except for L* algorithm. We think it may be because L* builds hypothesis model $\hat{M}$ according to the teacher. Here we use NN or SVM as the teacher of L*. The model construction of SVM and NN is

depends on the training data. If data size is too small, the information would be not enough. If data size is too big, there would be redundant information in the data set. It would build different size of model, although the benchmark is same and the time is relative to the size and complexity of model.

- The experiment time of SVM with optimal parameter is much higher than that without optimal parameter, because it costs the most part of time to find parameter.
- SVM seems more suitable to be the teacher of L* than NN, because the accuracy of SVM is higher than accuracy of NN.

size of training data set	**prediction accuracy**	**Time**
200	98.8%	0.168s
400	98.8%	0.262s
600	99.2%	0.310s
800	99.4%	0.389s
1000	100%	0.413s

Table 6.1 *TCAS* Experiment of NN

size of training data set	**prediction accuracy**	**Time**
200	94%	0.109s
400	94%	0.156s
600	100%	0.284s
800	100%	0.293s
1000	100%	0.301s

Table 6.2 *TCAS* Experiment of SVM

size of training data set	**prediction accuracy**	**Time**
200	100%	6.102s
400	100%	6.327s
600	100%	17.831s
800	100%	26.350s
1000	100%	35.218s

Table 6.3 *TCAS* Experiment of SVM with Optimal Parameter

size of training data set	**prediction accuracy**	**Time**
200	94%	Time out
400	100%	63.776s
600	100%	18.891s
800	100%	1239.611s
1000	92%	Time out

Table 6.4 *TCAS* Experiment of L* with Teacher NN

size of training data set	**prediction accuracy**	**Time**
200	100%	113.651s
400	100%	77.231s
600	100%	1392.952s
800	100%	110.710s
1000	100%	261.330

Table 6.5 *TCAS* Experiment of L* with Teacher SVM

The experiments of benchmark *schedule* are from Table 6.6 to Table 6.10. We could find several things.

- When the size of training data set increases, the prediction accuracy will almost increase in Table 6.6, Table 6.7 and Table 6.8 except for the size 1000. The data in size 1000 may be more discrete than the other and does not have strong connection with testing data. Moreover it will cost much time with increment of size as same as experiments of benchmark *TCAS*.
- The accuracy in Table 6.6, Table 6.7 and Table 6.8 is lower than same strategy used in *TCAS*. It may be because the *schedule* is more complex than *TCAS* and do not collect enough information.
- In Table 6.9 and Table 6.10, the cost time is time out in every experiment. For the higher prediction accuracy, we think it may be because the *schedule* program is so complex that we do not give enough time to let it to learn. For the lower prediction accuracy, we think it may be because the teacher NN and SVM do not get enough prediction accuracy such that it could not give a correct answer.
- The experiment time of SVM with optimal parameter is much higher than that without optimal parameter, and the reason is as same as experiment in *TCAS*.
- Here we can't figure out which one is suitable to be the teacher of L* because of the low prediction accuracy. Due to the result of experiments in *TCAS*, we still believe SVM is more suitable the NN if there are high prediction accuracy and enough time to learn.

We could find out that L* costs much time than the other learning algorithm from Table 6.4, Table 6.5, Table 6.9, and Table 6.10. It may be because it needs a lot queries to build the model and wait for teacher's respond. Because of many reasons we mentioned above, we think L* is not suitable for such a complex system.

size of training data set	**prediction accuracy**	**Time**
200	88.6%	1.137s
400	89.6%	2.541s
600	90.95%	3.862s
800	91.05%	4.839s
1000	90.4%	5.694s

Table 6.6 *Schedule* Experiment of NN

size of training data set	**prediction accuracy**	**Time**
200	89.6%	0.207s
400	88.5%	0.440s
600	89.5%	0.772s
800	89%	1.063s
1000	90%	1.517s

Table 6.7 *Schedule* Experiment of SVM

size of training data set	**prediction accuracy**	**Time**
200	90.5%	17.082s
400	92%	48.567s
600	94%	98.898s
800	94%	164.815s
1000	91.5%	248.747s

Table 6.8 *Schedule* Experiment of SVM with Optimal Parameter

size of training data set	prediction accuracy	Time
200	82%	Time out
400	88.8%	Time out
600	23%	Time out
800	80.5%	Time out
1000	61.5%	Time out

Table 6.9 *Schedule* Experiment of L* with Teacher NN

size of training data set	prediction accuracy	Time
200	41%	Time out
400	89%	Time out
600	33.5%	Time out
800	88.5%	Time out
1000	43.5%	Time out

Table 6.10 *Schedule* Experiment of L* with Teacher SVM

6.3 Results of Multiple Classifiers

Following we list the result of experiments in table type. The first column of table is the size of training data set, and columns which are from the second one to the fifth one are prediction accuracy of combinatorial functions mentioned in Chapter 4.4. Table 6.11 is the *TCAS* experiment which uses multiple classifiers. The weight values are obtained according to the accuracy of each classifier. We could easily find out that multi-classifier methodologies reach a better performance except the product. Obviously, the product function is an unsuitable combinatorial function, and it affects the performance very much.

Table 6.12 is the *schedule* experiment which uses multiple classifiers. We could find out that the weighted summation function performs better than other combinatorial functions and the SVM with optimal parameter.

Although, we could get a better result by using weighted summation function, it is time-consuming. The total cost time of combinatorial function is approximately equal to sum up the time of each classifier.

Combinatorial function / **size of training data set**	**Maximum**	**Summation**	**Weighted summation**	**product**
200	100%	100%	100%	66.5%
400	100%	100%	100%	91%
600	100%	100%	100%	28.5%
800	100%	100%	100%	28.5%
1000	100%	100%	100%	28.5%

Table 6.11 *TCAS* Experiment of Multiple Classifiers

Combinatorial function / **size of training data set**	**Maximum**	**summation**	**Weighted summation**	**product**
200	90.5%	90%	91.5%	39%
400	92%	89%	92%	88.5%
600	94%	91%	94%	63%
800	94%	93.5%	94%	77.5%
1000	91.5%	92%	92%	56%

Table 6.12 *Schedule* Experiment of Multiple Classifiers

Chapter 7 Conclusion and Future Work

In this thesis, we present a comparison among different machine learning algorithms and find out a better result by using combinatorial function. We also find out that L* may not suitable for big program and we think it is more suitable in unit testing. The performance is good when doing software testing in small program. We successfully combine different algorithms such as L* with teacher SVM and L* with teacher NN to build fault model and do prediction. We build the framework of using different machine algorithms to the software testing at the same time.

In the future, we may improve the performance of our work by finding a better way to decide the weight of weighted summation or by collecting more machine learning algorithm to do software testing or by tuning the parameter such as the number of neuron and the number of layer in hidden layer in NN or by selecting more useful feature selection. It is possible to use the model which is constructed by L* to do some model checking, and it is also possible to use L* in unit testing to get a great performance.

Bibliography

[1] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection", J. Mach. Learn. Res. 3, pp. 1157–1182.

[2] F. Wang, J.H. Wu, C.H. Huang, and K.H. Chang, "Evolving a test oracle in black-box testing", in *Fundamental Approaches to Software Engineering*, pp. 310-325, 2011.

[3] J.F. Bowring, J.M. Rehg, and M.J. Harrold, "Active learning for automatic classification of software behavior", in *International Symposium on Software Testing and Analysis*, pp 195-205, 2004.

[4] M.D. Ernst, J Cockrell, W.G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution", in IEEE Transactions of Software Engineering, Vol. 27, No. 2, 2001.

[5] N. Baskiotis, M. Sebag, M.C. Gaudel, and S. Gouraud, "A machine learning approach for statistical software testing", in *Proc. International Joint Conference on Artificial Intelligence*, 2007.

[6] C. Cortes and V. Vapnik. "Support-vector networks", in *Machine Learning*, Vol. 20, pp. 273-297, 1995.

[7] M. T. Hagan, H. B. Demuth, and M. Beale, "Neural Network Design", Boston: PWS Publishing Co., 1996.

[8] D. Angluin, "Queries and concept learning", in *Machine Learning*, 2, pp. 319-342, 1988.

[9] T.G. Dieterich, "Machine-learning research: Four current direction", *The AI Magazine* 18(4), pp. 97-136, 1998.

[10] Y. Lu, "Knowledge integration in a multiple classifier system", *Appl. Intell.* 6(2), pp. 75-86, 1996.

[11] L. Lam, "Classifier combinations: Implementations and theoretical issues", *In MCS 2000: Proceedings of the First International Workshop on Multiple Classifier Systems*, Springer-Verlag, London, UK, pp. 77-86, 2000

[12] A.M. de P Canuto, "Combining neural networks and fuzzy logic for applications in character recognization", *PHD thesis*, University of Kent, 2001.

[13] Y.-C. F. Wang and D. Casasent, "A hierarchical classifier using new support vector machine", in *ICDAR*, IEEE Computer Society, pp. 851–855, 2005.

[14] C.C. Chang and C.J. Lin, "LIBSVM : a library for support vector machines". In *ACM Transactions on Intelligent Systems and Technology*, 2:27:1--27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[15] M. Fowler, and K. Scott, "UML distilled: Applying the standard object modeling language", New York: AddisonWesley Longman, 1997.

[16] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do, "Software-artifact infrastructure repository", at http://sir.unl.edu/portal, 2006.

[17] S. Nissen, "Implementation of a fast artificial neural network library (fann)", Department of Computer Science University of Copenhagen (DIKU), Tech. Rep. 2003. Note http://fann.sf.net.

[18] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, David R. Piegdon, "Libalf: the automata learning framework", in *Proceedings of CAV'2010*. pp.360~364.