USING DATA MINING FOR AUTOMATED SOFTWARE TESTING*

M. LAST†
Department of Information Systems Engineering,
Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel
mlast@bgu.ac.il
http://www.ise.bgu.ac.il/faculty/mlast/

M. FRIEDMAN‡
Department of Physics, Nuclear Research Center — Negev,
Beer-Sheva, Israel
mlfrid@netvision.net.il

A. KANDEL§
Department of Computer Science and Engineering,
University of South Florida, Tampa, FL, USA
kandel@csee.usf.edu
http://www.csee.usf.edu/faculty/kandel.htm

In today’s software industry, the design of test cases is mostly based on human expertise, while test automation tools are limited to execution of pre-planned tests only. Evaluation of test outcomes is also associated with a considerable effort by human testers who often have imperfect knowledge of the requirements specification. Not surprisingly, this manual approach to software testing results in heavy losses to the world’s economy. In this paper, we demonstrate the potential use of data mining algorithms for automated modeling of tested systems. The data mining models can be utilized for recovering system requirements, designing a minimal set of regression tests, and evaluating the correctness of software outputs. To study the feasibility of the proposed approach, we have applied a state-of-the-art data mining algorithm called Info-Fuzzy Network (IFN) to execution data of a complex mathematical package. The IFN method has shown a clear capability to identify faults in the tested program.

Keywords: Automated software testing; regression testing; input–output analysis; data mining; info-fuzzy networks; finite element solver.

*This is a modified and extended version of the paper “The Data Mining Approach to Automated Software Testing”, published in the Proceedings of SIGKDD ’03, August 24–27, 2003, Washington, DC, USA.
†Corresponding author.
‡Currently on sabbatical at the Department of Information Systems Engineering, Ben-Gurion University of the Negev, Israel.
§Currently on sabbatical at the College of Engineering, Tel-Aviv University, Israel.
1. Motivation and Background

A recent study by the National Institute of Standards & Technology [35] found that “the national annual costs of an inadequate infrastructure for software testing is estimated to range from \textdollar22.2 to \textdollar59.5 billion” (p. ES-3) or about 0.6 percent of the US gross domestic product. This number \textit{does not include} costs associated with catastrophic failures of mission-critical software (such as the \textdollar165 million Mars Polar Lander shutdown in 1999). According to another report, the U.S. Department of Defense alone loses over \textbf{four billion dollars} a year due to software failures.

A program \textit{fails} when it does not meet the requirements [36]. The purpose of testing a program is to discover faults that cause the system to fail rather than proving the program correctness (\textit{ibid.}). Some even argue that the program correctness can never be verified through software testing [7]. The reasons for that include such issues as the size of the input domain, the number of possible paths through the program, and wrong or incomplete specifications. A successful test is one that reveals a problem in software; tests that do not expose any faults are useless, since they hardly provide any indication that the program works properly [19]. In developing a large system, the test of the entire application (\textit{system testing}) is usually preceded by the stages of \textit{unit testing} and \textit{integration testing} [36]. The activities of system testing include \textit{function testing}, \textit{performance testing}, \textit{acceptance testing}, and \textit{installation testing}.

The ultimate goal of function testing is to verify that the system performs its functions as specified in the requirements and there are no undiscovered errors left. Since proving the code correctness is not feasible for large software systems, the practical testing is limited to a series of experiments showing the program behavior in certain situations. Each choice of input testing data is called a test case. If the structure of the tested program itself is exploited to build a test case, this is called a \textit{white-box} (or clear-box) approach. Several white-box methods for automated generation of test cases are described in the literature. For example, the technique of Ref. 7 uses mutation analysis to create test cases for unit and module testing. A test set is considered adequate if it causes all mutated (incorrect) versions of the program to fail. The idea of testing programs by injecting simulated faults into the code is further extended in Ref. 41. Another paper [42] presents a family of strategies for automated generation of test cases from Boolean specifications. However, as indicated by Ref. 41, modern software systems are too large to be tested by the white-box approach as a single entity. White-box testing techniques can work only at the subsystem level. In function tests that are aimed at checking that a complex software system meets its specification, \textit{black-box} (or closed box) test cases are much more common. The actual outputs of a black-box test case are compared to expected outputs based on the tester’s knowledge and understanding of the system requirements.

Since the testing resources are always limited, each executed test case should have a reasonable probability of detecting a fault along with being non-redundant,
effective, and of a proper complexity [19]. It should also make the program failure obvious to the tester who knows, or is supposed to know, the expected outputs of the system. Thus, selection of the tests and evaluation of their outputs are crucial for improving the quality of tested software with less cost. If the functional requirements are current, clear, and complete, they can be used as a basis for designing black-box test cases. Assuming that requirements can be re-stated as logical relationships between inputs and outputs, test cases can be generated automatically by such techniques as cause-effect graphs [36] and decision tables [2]. A method for automatic generation of test vectors for Software Cost Reduction (SCR) specifications is described in [3].

To stay useful, any software system, whether it is an open source code, an off-the-shelf package or a custom-built application, undergoes continual changes. Most common maintenance activities in software life-cycle include bug fixes, minor modifications, improvements of basic functionality, and addition of brand new features. The purpose of regression testing is to verify that the basic functionality of the software has not changed as a result of adding new features or correcting existing faults. According to [35], “as software evolves, regression testing becomes one of the most important and extensive forms of testing” (p. A-1).

A regression test library or regression suite is a set of test cases that are run automatically whenever a new version of software is submitted for testing. Such a library should include a minimal number of tests that cover all possible aspects of system functionality. The standard way to design a minimal regression suite is to identify equivalence classes of every input and then use only one value from each edge (boundary) of every class. Even with a limited number of equivalence classes, this approach leads to an amount of test cases, which is exponential in the number of inputs. Besides, the manual process of identifying equivalence classes in a piece of software is subjective and inaccurate [19] and the equivalence class boundaries may change over time.

Ideally, a minimal test suite can be generated from a complete and up-to-date specification of functional requirements. Unfortunately, frequent changes make the original requirements documentation, even if once complete and accurate, hardly relevant to the new versions of software [10]. To ensure effective design of new regression test cases, one has to recover (reverse engineer) the actual requirements of an existing system. In [39], several ways are proposed to determine input-output relationships in tested software. Thus, a tester can analyze system specifications, perform structural analysis of the system’s source code, or observe the results of system execution. While available system specifications may be incomplete or outdated, especially in the case of a “legacy” application, and the code may be poorly structured, the data provided by the system execution seems to be the most reliable source of information on the actual functionality of an evolving system.

In this paper, we extend the idea initially introduced by us in [23] that input-output analysis of execution data can be automated by the IFN (Info-Fuzzy
Network) methodology of data mining [23, 28]. In [23] the proposed concept of IFN-based testing has been demonstrated on individual discrete outputs of a small business program. The current study evaluates the effectiveness of the IFN methodology on a complex mathematical application having multiple continuous outputs. This is also the first time that we deal with the question of determining the minimal number of training cases required to construct the IFN-based model of a given software system.

The rest of this paper is organized as follows. Section 2 provides the background on the Info-Fuzzy Network (IFN) methodology of data mining and its most important features. Section 3 presents the IFN-based method of input-output analysis, especially for systems having multiple continuous outputs. Section 4 describes a detailed case study (Unstructured Mesh Finite Element Solver). Finally, Sec. 5 summarizes the paper with initial conclusions and directions for future research in this area as well as practical applications of the proposed methodology.

2. The Info-Fuzzy Method of Data Mining

2.1. Overview

Classification and prediction are the two primary tasks of data mining [17] aimed at predicting the values of some target (output) attributes based on the values of some predicting (input) attributes. When a target attribute is characterized by a finite set of unordered values or labels ("classes"), this is usually called as a classification problem. Whenever we are interested in predicting a continuous value, the problem is referred to as prediction. Regression models that assume some pre-defined (e.g., linear) relationship between inputs and outputs are the most common technique for predicting continuous or ordered values. However, any classification method can be applied to continuous data as well by converting the task of value prediction into the task of range prediction. Both classification and prediction methods require a set of training examples with known values of target attributes. The training phase of inducing a predictive model from labeled data is known as supervised learning.

Decision tree learning is one of the most common supervised methods for inducing discrete-valued functions. Each internal node in a decision tree structure denotes a test on an input attribute, each branch represents an input value, and leaf (terminal) nodes are associated with labels, values or value ranges in the domain of the target attribute. Thus, decision trees represent a disjunction of conjunctions of input values [33]. The predicted value or class of a new instance is found by traversing the tree from the root node to one of its leaves. Decision-tree algorithms include CART [4], C4.5 [37], EODG [21], and many others. In this paper, we use the novel info-fuzzy network (IFN) methodology for data-driven induction of compact and accurate decision trees [28]. Info-fuzzy network is characterized by an “oblivious” structure, where the same input attribute is used across all nodes of a given layer (level). A detailed description of the IFN model is provided in the next sub-section.
2.2. Info-fuzzy network structure

The components of an info-fuzzy network (see Fig. 1) include the root node, a changeable number of hidden layers (one layer for each selected input), and the target (output) layer representing the possible output values. Each target node is associated with a value (class) in the domain of a target attribute. The target layer has no equivalent in decision trees, but a similar concept of category nodes is used in decision graphs [21]. In the case of a continuous target attribute, the target nodes represent disjoint intervals in the attribute range. The sample network in Fig. 1 has three output nodes denoted by numbers 1, 2, and 3.

A hidden layer No. 1 consists of nodes representing conjunctions of values of the first 1 input attributes, which is similar to the definition of an internal node in a standard decision tree. For continuous inputs, the values represent intervals identified automatically by the network construction algorithm. In Fig. 1, we have two hidden layers (No. 1 and No. 2). Like in OODG (Oblivious Read-Once Decision Graphs) [21], all nodes of a hidden IFN layer are labeled by the same feature. IFN extends the “read-once” restriction of OODG to continuous input attributes by performing multi-way splits of a continuous domain at the same network level.

The final (terminal) nodes of the network represent non-redundant conjunctions of input values that produce distinct outputs. The five terminal nodes of Fig. 1 include (1,1), (1,2), (2,3,1), and (3,2). If the network is induced from execution data of a software system, each interconnection between a terminal and a target node represents a possible output of a test case. For example, the connection (1,1) → 1 in Fig. 1 means that we expect the output value of 1 for a test case where both input variables are equal to 1. The connectionist nature of IFN resembles the structure of a multi-layer neural network. Accordingly, we characterize our model as a network and not as a tree.

![Fig. 1. Info-fuzzy network — an example.](image-url)
If the target attribute $T$ is continuous, the expected output of a terminal node $z$ is calculated as the mean target value of all training examples assigned to that node:

$$\text{Pred}_z(T) = \frac{\sum_{t_k \in z} V_k(T)}{N_z}$$

(1)

where $N_z$ is the number of training cases associated with the node $z$ and $V_k(T)$ is the output value of the training example $t_k$.

The idea of restricting the order of input attributes in graph-based representations of Boolean functions, called “function graphs”, has proven to be very useful for automatic verification of logic design in hardware and software testing. Bryant [5] has shown that each Boolean function has a unique (up to isomorphism) reduced function graph representation, while any other function graph denoting the same function contains more vertices. Moreover, all symmetric functions can be represented by graphs where the number of nodes grows at most as the square of the number of input attributes. As indicated by Kohavi [20], these properties of Boolean function graphs can be easily generalized to oblivious read-once decision graphs representing $k$-categorization functions. Consequently, the “read-once” structure of info-fuzzy networks makes them a natural modeling technique for testing complex software systems.

2.3. Network induction algorithm

In this paper, we represent each output variable by a separate info-fuzzy network. Thus, without loss of generality, we describe here an algorithm for constructing a network of a single output variable. The induction procedure starts with defining the target layer (one node for each target interval or class) and the “root” node representing an empty set of input attributes. The input attributes are selected incrementally to maximize a global decrease in the conditional entropy of the target attribute. Unlike CART$^{TM}$ [4], C4.5 [37], and EODG [21], the IFN algorithm is based on the pre-pruning approach: when no attribute causes a statistically significant decrease in the entropy, the network construction is stopped. In this paper, we focus on the selection of continuous input attributes, which present in our case study. The treatment of discrete input attributes by the IFN algorithm is covered elsewhere (e.g., see [23]).

The algorithm performs discretization of continuous input attributes “on-the-fly” by using an approach based on the information-theoretic heuristic of Fayyad and Irani [11]: recursively finding a binary partition of an input attribute that minimizes the total conditional entropy of the target attribute across all nodes of the front layer. This process is illustrated in Fig. 2. The stopping criterion of the IFN construction algorithm is different from [11]. Rather than searching for a minimum description length (minimum number of bits for encoding the training data), we make use of a standard statistical likelihood-ratio test [38]. The search for the best partition of a continuous attribute is dynamic: it is performed each time a candidate
input attribute is considered for selection. The resulting discretization intervals in the range of each input attribute can be considered as *equivalence classes* from the software testing standpoint.

We calculate the estimated conditional mutual information between the partition of the interval $S$ at the threshold $Th$ and the target attribute $T$ given the node $z$ by the following formula (based on \cite{6}):

$$MI(Th; T/S, z) = \sum_{t=0}^{M_t-1} \sum_{y=1}^{2} P(S_y; C_t; z) \cdot \log \frac{P(S_y; C_t; S, z)}{P(S_y/S, z) \cdot P(C_t/S, z)}$$

(2)

where

- $P(S_y/S, z)$ is an estimated conditional (a posteriori) probability of a sub-interval $S_y$, given the interval $S$ and the node $z$;
- $P(C_t/S, z)$ is an estimated conditional (a posteriori) probability of a value $C_t$ of the target attribute $T$ given the interval $S$ and the node $z$;
- $P(S_y; C_t/S, z)$ is an estimated joint probability of a value $C_t$ of the target attribute $T$ and a sub-interval $S_y$ given the interval $S$ and the node $z$; and
- $P(S_y; C_t, z)$ is an estimated joint probability of a value $C_t$ of the target attribute $T$, a sub-interval $S_y$, and the node $z$.

The statistical significance of splitting the interval $S$ by the threshold $Th$ at the
node $z$ is evaluated using the likelihood-ratio statistic (based on [38]):

$$G^2(T_h; T/S, z) = 2 \sum_{j=0}^{M_i-1} \sum_{y=1}^{2} N_{ij}(S_y, z) \cdot \ln \frac{N_t(S_y, z)}{P(C_t/S, z) \cdot E(S_y, z)}$$  (3)

where

- $N_t(S_y, z)$ is the number of occurrences of the target value $C_t$ in sub-interval $S_y$ and the node $z$;
- $E(S_y, z)$ is the number of records in sub-interval $S_y$ and the node $z$;
- $P(C_t/S, z)$ is an estimated conditional (a posteriori) probability of the target value $C_t$ given the interval $S$ and the node $z$; and
- $P(C_t/S, z) \cdot E(S_y, z)$ is an estimated number of occurrences of the target value $C_t$ in sub-interval $S_y$ and the node $z$ under the assumption that the conditional probabilities of the target attribute values are identically distributed over each sub-interval.

The Likelihood-Ratio Test is a general-purpose method for testing the null hypothesis $H_0$ that two random variables are statistically independent. If $H_0$ holds, then the likelihood-ratio test statistic $G^2(T_h; T/S, z)$ is distributed as chi-square with $NT(S, z) - 1$ degrees of freedom, where $NT(S, z)$ is the number of values of the target attribute in the interval $S$ at node $z$. The default significance level ($p$-value) used by the IFN algorithm is 0.1%.

A new input attribute is selected to maximize the total significant decrease in the conditional entropy, as a result of splitting the nodes of the last layer. The nodes of a new hidden layer are defined for a Cartesian product of split nodes of the previous hidden layer and discretized intervals of the new input variable. If there is no candidate input variable significantly decreasing the conditional entropy of the output variable the network construction stops. In Fig. 1, the first hidden layer has three nodes related to three intervals of the first input variable, but only nodes 1 and 3 are split, since the conditional mutual information as a result of splitting node 2 proves to be statistically insignificant. For each split node of the first layer, the algorithm has created two nodes in the second layer, which represent the two intervals of the second input variable. None of the four nodes of the second layer are split, because they do not provide a significant decrease in the conditional entropy of the output.

In Table 1, we show the main steps for constructing an info-fuzzy network from a set of continuous input attributes. Complete details are provided in [23] and [28].
Table 1. IFN construction procedure.

<table>
<thead>
<tr>
<th>Input:</th>
<th>The set of ( n ) training examples; the set ( C ) of candidate inputs; the target (output) variable ( T ); the minimum significance level sign for splitting a network node (default: ( \text{sign} = 0.1% )).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>A set ( I ) of selected inputs, discretized intervals for every input, and an info-fuzzy network. Each selected input has a corresponding hidden layer in the network.</td>
</tr>
<tr>
<td>Step 1</td>
<td>Initialize the info-fuzzy network (single root node representing all runs, no hidden layers, and a target layer for the values of the output variable). Initialize the set ( I ) of selected inputs as an empty set: ( I = \emptyset ).</td>
</tr>
<tr>
<td>Step 2</td>
<td>While the number of layers (</td>
</tr>
<tr>
<td>Step 2.1</td>
<td>For each candidate input ( A/A \in C; A \notin I ) do</td>
</tr>
<tr>
<td>Step 2.1.1</td>
<td>For each distinct value ( T_h ) included in the range of ( A ) (except for the last distinct value) Do:</td>
</tr>
<tr>
<td></td>
<td>For each node ( z ) of the final hidden layer Do:</td>
</tr>
<tr>
<td></td>
<td>Calculate the likelihood-ratio test for the partition of the interval ( S ) at the threshold ( T_h ) and the target attribute ( T ) given the node ( z ).</td>
</tr>
<tr>
<td></td>
<td>If the likelihood-ratio statistic is significant, mark the node as “split” by the threshold ( T_h ) and increment the conditional mutual information given the threshold ( T_h ).</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td>Step 2.1.2</td>
<td>Find the threshold ( T_{h_{\text{max}}} ) maximizing the conditional mutual information across all nodes</td>
</tr>
<tr>
<td>Step 2.1.3</td>
<td>If the maximum conditional mutual information is greater than zero, then Do:</td>
</tr>
<tr>
<td></td>
<td>For each node ( z ) of the final hidden layer Do:</td>
</tr>
<tr>
<td></td>
<td>Partition each sub-interval of ( S ) (go to Step 2.1.1)</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Define a new discretization interval of ( A )</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td>Step 2.2</td>
<td>Find the candidate input ( A^* ) maximizing ( \text{cond MI}(A) )</td>
</tr>
<tr>
<td>Step 2.3</td>
<td>If ( \text{cond MI}(A^*) = 0 ), then</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Expand the network by a new hidden layer associated with the input ( A ), and add ( A ) to the set ( I ) of selected inputs ( I = I \cap A ).</td>
</tr>
<tr>
<td></td>
<td>End Do</td>
</tr>
<tr>
<td>Step 3</td>
<td>Return the set of selected inputs ( I ), the associated discretization classes, and the network structure</td>
</tr>
</tbody>
</table>

sets though IFN models contained less input attributes. A reasonably high predictive accuracy of IFN models is important if we intend to use them as “automated oracles” in regression testing, but expecting them to become “perfect predictors” of all outputs in complex software systems is certainly unrealistic. On the other hand, as shown in [23], the inherent compactness of these models can help us to recover the most dominant requirements from execution data and, consequently,
The IFN algorithm is trained on inputs provided by RTG and outputs obtained from a legacy system by means of the Test Bed module. As indicated above, a separate IFN model is built for each output variable. The following information can be derived from each IFN model:

1. A set of input attributes relevant to the corresponding output.
2. Logical (if...then...) rules expressing the relationships between the selected input attributes and the corresponding output. The set of rules appearing at each terminal node represents the distribution of output values at that node (see [28]).

3. Input-Output Analysis with Info-Fuzzy Networks

The training phase of the IFN-based input-output analysis is shown in Fig. 3. Random Tests Generator (RTG) obtains the list of system inputs and outputs along with their types (discrete, continuous, etc.) from System Specification. No information about the functional requirements is needed, since the IFN algorithm automatically reveals input-output relationships from randomly generated training cases. In our case study (see the next section), we explore the effect of the number of randomly generated training cases on the predictive accuracy of the IFN model. Systematic, non-random approaches to training set generation may also be considered.

The IFN algorithm is trained on inputs provided by RTG and outputs obtained from a legacy system by means of the Test Bed module. As indicated above, a separate IFN model is built for each output variable. The following information can be derived from each IFN model:

1. A set of input attributes relevant to the corresponding output.
2. Logical (if...then...) rules expressing the relationships between the selected input attributes and the corresponding output. The set of rules appearing at each terminal node represents the distribution of output values at that node (see [28]).
(3) **Equivalence Classes.** In testing terms, each discretization interval represents an “equivalence class”, since for all values of a given interval the output values conform to the same distribution.

(4) **A set of non-redundant test cases.** The terminal nodes in the network are converted into test cases, each representing a non-redundant conjunction of input values/equivalence classes and the corresponding distribution of output values. Whenever the behavior of the tested application is characterized by some regular pattern, we expect the IFN-based number of test cases to be much smaller than the number of random cases used for training the network. This assumption is supported by the results of our case studies including the one presented in this paper.

(5) **Output prediction and verification.** The induced IFN model can be used to determine the predicted (most probable) value of the output in each test case. The information about the distribution of output values at the corresponding terminal nodes can also be used to evaluate the correctness of actual outputs produced by a new version of the tested system. In this paper, we implement the simplest, linear measure of test correctness for continuous outputs:

\[ d_l(T) = \text{abs}(V_l(T) - \text{Pred}_z(T)), \quad d_l \in z \]  

where \( \text{Pred}_z(T) \) is the predicted output value at the terminal node \( z \) and \( V_l(T) \) is the actual output value of the test case \( t_l \). The test fails if the above calculated difference \( d_l(T) \) exceeds a pre-defined threshold.

The usage of IFN models in the regression testing phase is depicted in Fig. 4.

---

**Fig. 4.** IFN-based IO analysis: the regression testing phase.
A brief description of each module in the IFN-based testing environment is provided below:

**Legacy System (LS).** This module represents a stable version of a program, a component or a system that produces execution data for training the IFN algorithm. We assume here that this module is data-driven, i.e. it should have a well defined interface in terms of obtained inputs and returned outputs. Examples of data-driven software range from real-time controllers to scientific applications.

**Specification of Application Inputs and Outputs (SAIO).** Basic data on each input and output variable in the Legacy System interface includes variable name, type (discrete, continuous, nominal, etc.), and a list or a range of possible values. Such information is generally available from requirements management and test management tools (e.g., Rational RequisitePro® or TestDirector®).

**Random Tests Generator (RTG).** This module generates random combinations of values in the range of each input variable. Variable ranges are obtained from the SAIO module (see above). The number of training cases to generate is determined by the user. The generated training cases are used by the Test Bed and the IFN modules.

**Test Bed (TB).** This module, sometimes called “test harness”, feeds training cases generated by the RTG module to the original Legacy System (LS). It also provides test cases stored in the Regression Test Library for the subsequent versions to be tested. Execution of training/test cases can be performed by commercial test automation tools. The TB module obtains the LS outputs for each executed case and forwards them to the IFN module.

**Info-Fuzzy Network Algorithm (IFN).** The input to the IFN algorithm includes the training cases randomly generated by the RTG module and the outputs produced by the Legacy System. IFN also uses the descriptions of variables stored by the SAIO module. The IFN algorithm is run repeatedly to find a subset of input variables relevant to each output and the corresponding set of non-redundant test cases. Actual test cases are generated from the automatically detected equivalence classes by using an existing testing policy (e.g., one test for each side of every equivalence class) and stored in the Regression Test Library.

**IFN-Based Automated Oracle.** As indicated above the induced IFN model can be used as an automated oracle for the new, tested versions of the program.

In [23], we have applied the IFN algorithm to execution data of a small business application (Credit Approval), where the algorithm has chosen 165 representative test cases from a total of 11 million combinatorial tests. The Credit Approval application had 8 mostly discrete inputs and two outputs (one of them binary). It was based on well-defined business rules implemented in less than 300 lines of code. In the next section, we evaluate the proposed approach on a much more complex program, which, unlike Credit Approval, is characterized by real-valued inputs and outputs only.
4. Case Study: Finite Element Solver

4.1. The finite element method

The finite element method was introduced in the late 1960’s for solving problems in mechanical engineering [31, 43] and quickly became a powerful tool for solving differential equations in mathematics, physics and engineering sciences [12, 13, 15]. The method consists of two stages. The first stage is finding a “functional”, which is usually an integral that contains the input of the problem, such as coefficients of the differential equation and boundary or initial conditions and an unknown function, and is minimized by the solution of the differential equation. The existence of such a functional is a necessary condition for implementing the finite element method. If the functional exists, the differential equation may be solved indirectly by minimizing the functional rather than by a direct approach such as the finite differences method. The second stage is partitioning (usually referred to as ‘triangulating’) the domain over which the equation is solved into “elements”, usually triangles. This process is called “triangulation” and provides the “finite elements”. Over each element the solution is approximated by a low degree polynomial (usually no more than fourth order).

The coefficients of each polynomial are unknown and are determined at the end of the minimization process. The union of the polynomials associated with all the finite elements provides an approximate solution to the problem. As in finite differences, a finer “finite element mesh” will provide a better approximation. Due to its simplicity the choice of ‘linear elements’, i.e. linear polynomial approximation over each element, is most commonly applied. However, the use of quadratic or cubic polynomials which generates a nonlinear approximation over each element, improves the accuracy of the approximate solution.

If the problem’s domain is a square or a rectangle, the finite differences method is simple and easily applied. However, if the domain’s boundary is an arbitrary curved line and the boundary conditions complicated, the finite element method is always superior, provided of course that an appropriate functional can be found.

4.2. The code

The case study investigated in this paper is the code Unstructured Mesh Finite Element Solver (UMFES) which is a general finite element program for solving 2D elliptic partial differential equations (e.g., Laplace’s equation) over an arbitrary bounded domain, with given general Boundary Conditions (B.C.). UMFES consists of about 3,000 lines of FORTRAN 77 code and can solve any well-posed boundary condition problem, deterministic and eigenvalue alike. The program is composed of two parts. The first obtains the domain over which the differential equation is solved as input, and triangulates it using fuzzy expert system’s technique [14]. The second part solves a system of linear equations whose unknowns are the approximate solution values at the triangular mesh vertices.
The input for the code’s first part is a sequence of points which defines the domain’s boundary counterclockwise. It is assumed that the domain is simply connected, or that at least has only a finite number of holes. In the case of holes the user must first divide the domain into several simply connected subdomains and provide their boundaries separately. The output of the code’s first part is a sequence of triangles whose union is the original domain. The triangulation is carried out in such a way that two arbitrary triangles whose intersection is not an empty set, have either one complete side in common or a single vertex. The rules for triangulation which are based on a vast accumulated knowledge and experience were chosen to minimize the process and to increase the approximate solution accuracy for a prefixed number of nodes. For example, given an initial domain, we first divide its boundary to small segments, thus extending the initial list of boundary points (nodes). We are now requested to find the best choice of a single secant that will divide the domain into a couple of subdomains. The ‘best’ choice which is a rule of thumb, is to draw a secant that will be completely inside the domain, will provide two subdomains with a ‘similar’ number of boundary nodes and at the same time will be ‘as short as possible’. After drawing this ‘optimal’ secant, we treat each subdomain separately and repeat the process. The triangulation terminates when each subdomain is a triangle. Other rules guarantee that each triangle will ‘hardly ever’ possess an angle close to 0° or to 180°. This enables us to use fewer triangles and increase the solution’s accuracy. Two types of triangulations are used: (a) the triangles are ‘close’ to right-angled isosceles, and (b) the triangles are ‘close’ to equilateral. The approximate solution is calculated at the triangles’ vertices which are called nodes. The nodes are numbered so that each two nodes whose numbers are close, are also geometrically close, ‘as much as possible’. The fulfillment of this request guarantees that the coefficient matrix of the unknowns whose inverse is calculated at the final stage of the numerical process, is such that its nonzero elements are located close to the main diagonal, which is advantageous for large matrices.

UMFES allows a variable density of the triangles. Such triangulation is shown in Fig. 5 where the density function is

\[ \rho(x, y) = 1 + 2x^2 + 3y^2. \]  

Consequently, the triangulation is such that the least number of triangles per area unit is at (0, 0) and the largest occurs at (3/4, 1). The density function in Fig. 9 (this is the particular problem for which UMFES was applied) is clearly constant.

The UMFES code can cope with any two-dimensional finite domain with a finite number of holes. Once the triangulation is completed, the code’s second part applies the finite element method to solve any 2-D elliptic partial differential equation with general boundary conditions. The user must now provide the boundary conditions, i.e. information about the solution over the domain’s boundary.

The code can treat 3 types of B.C. The first is of Dirichlet type, where the solution \( \phi \) is a specified function over some portion of the boundary. The second
The code can treat 3 types of B.C. The first is of Dirichlet type, where the solution is a specified function over some portion of the boundary. The second type is called homogeneous Neumann and requests $\partial \phi / \partial n = 0$ over a second portion of the boundary. The third is a mixed type B.C., represented by the relation $\partial \phi / \partial n + \sigma \phi = \eta$ where $\sigma$ and $\eta$ are functions, defined along a third portion of the boundary. The density of the finite elements is specified by the user and should be based on some preliminary knowledge of the solution’s features such as the sub-domains where it is expected to vary slow or fast.

4.3. Applications for real world problems

In its early stages UMFES was developed as a finite element solver for steady-state problems. The domain and the B.C. (Boundary Conditions) were fixed and a single triangulation was performed for solving an arbitrary 2D elliptic PDE (Partial Differential Equation). While the original code was designed for linear PDE’s, a second updated version could handle ‘simple’ nonlinear PDE such as the Thomas-Fermi equation, which in spherical coordinates and assuming axial symmetry is given by

$$\frac{\partial^2 \phi}{\partial r^2} + \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial \phi}{\partial \theta} \right) - \frac{\phi^{3/2}}{\sqrt{r}} = 0.$$  

Given a set of B.C. i.e. any combination of Dirichlet, homogeneous Neumann and mixed type, the solution to the Thomas–Fermi equation can be obtained by minimizing the functional

$$F(\phi) = \iint_D \left[ \frac{x}{y^2} (\phi_x^2 + \phi_y^2) + \frac{4x\phi^{5/2}}{5r^{5/2}} \right] dx dy$$

where $D$ is the solution domain and $\phi$ is an arbitrary function defined over $D$ that satisfies the boundary conditions. This problem was treated by UMFES over arbitrary ellipsoids, imposing Dirichlet and homogeneous Neumann B.C. [13].

More than five years ago (1998) UMFES was improved to treat dynamic problems as well, i.e. problems where the boundary and the B.C. are time-dependent.
Whether or not a water jet is created depends solely on the radius of the ring $D$. If the ring is large enough, a jet is not be created, as demonstrated by the velocity map in Fig. 7 (here $D = 300 \mu m$).

This latest version was fully tested and successfully applied to several disciplines including physics, engineering, and medicine. The following example demonstrates a medical application: prevention of tissue damage by water jet creation during an eye surgery.

The process of a vapor bubble (surrounded by water) expansion and collapse is simulated using UMFES for solving Laplace equation (see Sec. 4.4) with Dirichlet and homogeneous Neumann B.C. At each time step the bubble’s boundary which is the moving part of the domain’s boundary is updated along with its B.C. The expansion of the bubble is applied to destroy a blood clot a few microns away. However, during the later stage of collapse, there is a danger of water jet creation away from the bubble collapse center that may damage a tissue close by. Only a smart design of the fiber through which a short-pulse laser beam originally starts the bubble, guarantees that the bubble collapse will be limited on the fiber and eliminates the possibility of water jet creation. Figure 6 presents a cross-section of the cone-shaped fiber, an attached cylindrical ring and the initial bubble.

Whether or not a water jet is created depends solely on the radius of the ring $D$. If the ring is large enough, a jet is not created, as demonstrated by the velocity map in Fig. 7 (here $D = 300 \mu m$).
Using Data Mining for Automated Software Testing 385

A reduction in the ring’s size (for example $D = 150 \, \mu m$) is apparently technically desirable but may lead to a jet creation as seen by the velocity map in Fig. 8.

A successful design of the fiber-ring apparatus will yield a moderate ring with no jet creation. The particular problem chosen as a case study in this paper is given below.

4.4. The case study

We solved the Laplace equation

$$\nabla^2 \phi = \phi_{xx} + \phi_{yy} = 0$$

over the unit square

$$D = \{(x, y)|0 \leq x \leq 1, 0 \leq y \leq 1\}$$

where the solution equals some given $f(x, y)$ on the square’s boundary. This problem, i.e. finding $\phi(x, y)$ inside the square, has a unique solution. To find it we first define the functional

$$F = \int \int_D (\phi_x^2 + \phi_y^2)dx\,dy$$

where $\phi(x, y)$ is an arbitrary function which equals $f(x, y)$ on the square’s boundary. This functional attains its minimum at $\phi = \phi_0$ where $\phi_0$ is the exact solution. To get an approximate solution we triangulate the square as shown in Fig. 9. As there is no preliminary knowledge about the solution’s behavior in various locations, we perform homogeneous triangulation.

A low degree polynomial (say, linear) is chosen to approximate the solution over each triangle. The unknown coefficients of each polynomial are determined by minimizing the functional $F$ (Eq. (10)) subject to the boundary conditions $f$:

$$\phi = f, \text{ on } B.$$
Using Data Mining for Automated Software Testing

no preliminary knowledge about the solution’s behavior in various locations, we perform homogeneous triangulation. A low degree polynomial (say, linear) is chosen to approximate the solution over each triangle. The unknown coefficients of each polynomial are determined by minimizing the functional $F$ (Eq. (10)) subject to the boundary conditions $f: \mathbb{R} \rightarrow \mathbb{R}$, $f_0 \in \mathbb{B}$.

Fig. 9. Triangulation of the domain.

4.5. The experiments

We have solved the case study of Sec. 4.4 by replacing $f(x, y)$ with constants. Each problem was associated with Dirichlet B.C. defined by four constants (inputs), which specified the solution over the sides of the unit square. The four inputs were denoted as $Side_1, \ldots, Side_4$ respectively. The five outputs were taken as the solution’s values at the fixed points defined by the following coordinate pairs (see Fig. 9):

$$
$$

Accordingly, we referred to the outputs as $Out_1, \ldots, Out_5$. The minimal distance from the boundary affects the predictive accuracy of the numerical solution. Thus, we can expect that the prediction for the last point ($Out_5$) will be the least accurate one.

The B.C. conditions were taken randomly between $-10$ and $+10$, and we chose a sufficiently fine mesh of elements that guaranteed at least 1% accuracy at each output. The maximum and minimum values of the solution are known to occur on the boundary, and consequently the numerical values inside the domain must be between $-10$ and $10$ as well. Thus, this typical problem was represented by a vector of 9 real-valued components: 4 inputs and 5 outputs.

Our first series of experiments was aimed at finding the minimal number of cases required to train the info-fuzzy network up to a reasonable accuracy. In this and all subsequent experiments we have partitioned the values of all target attributes into 20 intervals of equal frequency each, assuming this would be a sufficient precision for identifying the basic input-output relationships of the program. The algorithm was repeatedly trained on the number of cases varying between 50 and 5,000, producing five networks representing the five outputs in each run. All training cases were generated randomly in the input space of the problem. The predictions of every
induced model have been compared to the actual output values in the training cases and in separate 500 validation cases using the Root Mean Square Error (RMSE). In Fig. 10, we present the average RMSE over all five outputs as a function of the number of training records, while Fig. 11 shows RMSE for each individual output.
Table 2. IFN models induced from 1,000 cases.

<table>
<thead>
<tr>
<th>Output</th>
<th>Selected Input Attributes</th>
<th>Total Nodes</th>
<th>Terminal Nodes</th>
<th>Total Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out1</td>
<td>Side4, Side1, Side2</td>
<td>79</td>
<td>67</td>
<td>264</td>
</tr>
<tr>
<td>Out2</td>
<td>Side2, Side1, Side4</td>
<td>86</td>
<td>74</td>
<td>276</td>
</tr>
<tr>
<td>Out3</td>
<td>Side3, Side2, Side4</td>
<td>90</td>
<td>77</td>
<td>275</td>
</tr>
<tr>
<td>Out4</td>
<td>Side3, Side4, Side1</td>
<td>102</td>
<td>91</td>
<td>279</td>
</tr>
<tr>
<td>Out5</td>
<td>Side1, Side3, Side4, Side2</td>
<td>102</td>
<td>77</td>
<td>318</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>459</td>
<td>386</td>
<td>1412</td>
</tr>
</tbody>
</table>

Table 3. Out1: Examples of extracted rules.

<table>
<thead>
<tr>
<th>Rule No.</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If Side4 is between $-9.982$ and $-7.895$ and Side1 is between $-9.981$ and $-6.298$ then Out1 is between $-9.325$ and $-6.135$</td>
</tr>
<tr>
<td>27</td>
<td>If Side4 is between $-7.895$ and $-6.075$ and Side1 is between $-9.981$ and $-6.298$ then Out1 is between $-9.325$ and $-6.135$</td>
</tr>
<tr>
<td>51</td>
<td>If Side4 is between $-6.075$ and $-3.758$ and Side1 is between $-9.981$ and $-6.298$ then Out1 is between $-6.135$ and $-4.652$</td>
</tr>
<tr>
<td>221</td>
<td>If Side4 is between $5.473$ and $7.872$ and Side1 is more than $7.904$ then Out1 is more than $5.892$</td>
</tr>
<tr>
<td>246</td>
<td>If Side4 is more than $7.872$ and Side1 is more than $7.904$ then Out1 is more than $5.892$</td>
</tr>
<tr>
<td>261</td>
<td>If Side4 is between $2.703$ and $5.473$ and Side1 is between $4.865$ and $7.904$ and Side2 is between $-9.992$ and $0.58$ then Out1 is between $4.103$ and $4.893$</td>
</tr>
</tbody>
</table>

Both figures demonstrate a steep decrease in the mean error between 50 and 1,000 cases. Beyond 1,000 cases, the training error continues to decrease very slowly, while the validation error remains relatively stable. These results suggest that 1,000 cases only should be sufficient for training the info-fuzzy network on execution data of this program. This confirms our earlier expectation that Out5 behaves differently from the other outputs, which are closer to the domain boundaries. In fact, its mean error is almost two times higher than the error of any other output.

Table 2 shows the input attributes included in the model of every output after training the algorithm on 1,000 cases. The selected attributes are listed in order of their appearance in the network. The same table also presents the size of each model (total number of nodes), the number of non-redundant test cases (number of terminal nodes), and the number of logical rules describing the induced input-output relationships. An immediate result is that the test library based on the induced models would have only 386 test cases (one per terminal node) versus the original training set of 1,000 random cases. One can also see that only three input attributes out of four are sufficient for predicting the values of all outputs except for Out5, which is the hardest point to predict (see above). Also, the models are quite similar to each other in terms of the number of nodes and number of rules. Sample rules predicting the value of the first output (Out1) are shown in Table 3.
Using Data Mining for Automated Software Testing

Validation Mean Square Error (Faulty Program)

Consolidating the networks of multiple outputs into a single network is a subject of ongoing research.

The next step was to evaluate the capability of the IFN model to detect an error in a new, possibly faulty version of this program. We have generated four faulty versions of the original code, where the outputs were mutated within the boundaries of the solution range as follows:

1. error = 0.25*rand [-10, 10], where rand [-10, 10] is a random number in the range of the solution
2. error = 0.50*rand [-10, 10]
3. error = 0.75*rand [-10, 10]
4. rand [-10, 10]

Each faulty version was used to generate 500 validation cases. The validation RMSE of each model induced from 1,000 training cases is shown in Fig. 12. We can see that even for the smallest injected error (1/4 of the solution range) there is a considerable difference of at least 30% between the RMSE of the outputs generated by the original program and the RMSE of the outputs generated by the faulty program. For all outputs, including the “hard to predict” output no. 5, the statistical significance level of this difference was found much higher than 0.001 (using the $F$ test). These results show the effectiveness of IFN in discriminating between the correct and the faulty versions of tested software.

We have also studied the IFN capability to identify individual cases that produce incorrect output. In other words, we are interested to maximize the probability of
catching an error (True Positive Rate), while minimizing the probability of mistaking correct outputs for erroneous (False Positive Rate). As a criterion for detecting an error, we have used the average of absolute differences between predicted and actual outputs. The ROC (Receiver-Operating-Characteristic) Curve for the four mutated versions of the original program is shown in Fig. 13. We can see that for the smallest error (0.25 of the output range), we can get TP = 90% with FP slightly over 25%. However, for larger errors, the values of TP become very close to 100%, while keeping FP close to 0%.

5. Summary and Conclusions

In this paper, we have introduced an emerging methodology for automated regression testing of data-driven software systems. As indicated in [39], such systems include embedded (real-time) applications, application program interfaces (API), and form-based web applications. The proposed methodology is based on the info-fuzzy method of data mining, which includes the following testing-related features:

- The method learns functional relationships automatically from execution data. Current methods of test design assume existence of detailed requirements, which may be unavailable, incomplete or unreliable in many legacy systems.
- The method is applicable to modeling complex software programs (such as UMFES), since it does not depend on the analysis of program code like the white-box methods of software testing.
- The method can automatically produce a set of non-redundant test cases covering the most common functional relationships existing in software.
Issues for ongoing research include applying other data-mining methods to regression testing, inducing a single multi-output predictive model, and application of the proposed methodology to large-scale software systems. Future experiments will also include evaluation of the method’s capability to detect various types of errors injected in the code of the tested program.

Acknowledgements

This work was partially supported by the National Institute for Systems Test and Productivity at the University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248 and by the Fulbright Foundation that has granted Prof. Kandel the Fulbright Research Award at Tel-Aviv University, College of Engineering during the academic year 2003-2004.

References

1. Astra QuickTest from Mercury Interactive, http://astratryandbuy.mercuryinteractive.com
15. M. Friedman, M. Strauss, P. Amendt, R. A. London and M. E. Glinsky, Two-

