Lecture 8 – Application of machine learning in software testing

- Software testing
- Role of machine learning in software testing
- Artificial neural network
- Use of ANN in software testing

Lecture 7 – Machine Learning based Software Defect Prediction Techniques

- Software defect prediction
- Software Metrics
- Machine Learning based Software Defect Prediction
- Detailed Example: Association rule based Software Defect Prediction
Software testing

Testing is observing the behavior of a program in many executions. Execute the program for some input data and observe if the results are correct for these inputs.

Testing does not prove program correctness (only give us some confidence). On the contrary, it may prove its incorrectness if one execution give wrong results.

Testing can never completely identify all the defects within software.

Testing methods

Exhaustive testing
Check the program for all possible inputs. Impractical so we need to choose a finite number of test cases.

Black box testing
The selection of input data for testing is decided by analyzing the specification. Distinct cases of the problem are decided and we use a test input data for each case

White box testing
Select the test data by analyzing the text of the program. We select test data such that all the execution paths are covered. We test a function such that each statement is executed.

Testing levels
Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing
Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. Testing unit of code in isolation (functions). Test small parts of the program independently

Integration testing
Considers the way program works as a whole. After all modules have been tested and corrected we need to verify the overall behavior of the program.
Software testing

Automated testing
Test automation is the process of writing a computer program to do testing that would otherwise need to be done manually. Use of software to control the execution of test, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions

A test case has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement. A test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors.

Code coverage

Is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.

Test coverage is a useful tool for finding untested parts of a code-base.

Coverage criteria:
- Function coverage- Has each function (or subroutine) in the program been called?
- Statement coverage- Has each statement in the program been executed?
- Branch coverage - Has each branch of each control structure (such as in if and case statements) been executed?
- Condition coverage (or predicate coverage) - Has each Boolean sub-expression evaluated both to true and false?
# White box vs Black Box testing

```python
def isPrime(nr):
    
    """
    Verify if a number is prime
    return True if nr is prime False if not
    raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:
        #1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

**Black Box**
- test case for a prime/not prime
- test case for 0
- test case for negative number

**White Box (cover all the paths)**
- test case for 0
- test case for negative
- test case for 1
- test case 3
- test case for prime (no divider)
- test case for not prime

```python
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```
Role of machine learning in software testing

Various types of data can be collected while testing software:

- Execution traces and coverage information for test cases can be captured at different levels of detail.
- Failure data that captures where and why a failure occurs is also of interest.

Testing problems:

- completeness of test suites,
- the automation of test oracles,
- the distribution of testing resources according to levels of risks (risk-driven testing),
- localization of faults causing failures.

Recurrent issues in applying machine learning techniques to software testing:

- whether, for a given application, we can possibly obtain adequate data, both in terms of form and content, to learn interesting, actionable patterns.
- patterns that are an inaccurate approximation of reality or cannot be easily used to drive beneficial actions are of no practical interest.
- the benefits must be significantly higher than the cost of additional data collection and analysis. The complexity of patterns to be learned may be such that the data needs complex preprocessing before a machine learning algorithm becomes capable of learning useful patterns.
Examples of machine learning for software testing

**Test specification and test suite refinement:** understand the limitations of test suites and their possible redundancies

MELBA (MachinE Learning based refinement of Black-box test specification) methodology requires as input both a test suite and a test specification in the form of Category-Partition (CP) categories and choices

- Categories are properties of parameters that can have an influence on the behavior of the software under test (e.g., size of an array in a sorting algorithm).
- Choices (e.g., whether an array is empty) are the potential values of a category
- Abstract test cases are generated: shows an output equivalence class and pairs (category, choice) that characterize its inputs and environment parameters (instead of raw inputs).
- MELBA can identify a number of potential problems (by analyzing decision tree):
  - Case 1—Instances (test cases) can be misclassified: the wrong output equivalence class is associated to a test case.
  - Case 2—Certain categories or choices are not used in the tree
  - Case 3—Certain combinations of choices, across categories, are not present on any path, from the root node to any leaf of the tree.
  - Case 4—A leaf of a tree contains a large number of instances (test cases).

**Debugging / Fault localization:** to identify suspicious statements, which are likely to contain a fault related to a failure observed during testing. This can be used to help fault localization during debugging.

Use of decision trees to learn various failure conditions based on information regarding the test cases' inputs and outputs. Failing test cases executing under similar conditions are then assumed to fail due to the same fault(s). Statements are then considered suspicious if they are covered by a large proportion of failing test cases that execute under similar conditions.

Uses categories and choices. Ex. Triangle: input values characterize the length of triangle sides. We can use CP to define categories on the relationships among the lengths of the triangle sides, e.g., whether they are equal or otherwise

Then the test cases can be expressed as tuples in terms of these properties and we refer to these tuples as abstract test cases, e.g., input data \((1, 1, 2)\) becomes \((\text{side1}=\text{side2}, \text{side2}<\text{side3}, \text{side3}>\text{side1})\)
Examples of machine learning for software testing

Risk-driven testing: prioritize testing efforts.

Usually this is done by analyzing the “risk” associated with a functionality or system components, depending on the testing level. Risk is usually defined as a combination of probability of failures and the damages they can potentially cause focused on the construction of models predicting the location of faults across files or classes.

Test oracles: A test oracle might specify correct output for all possible input or only for specific input. It might not specify actual output values but only constraints on them.

Using machine learning we can learn for a given function/module the relations between inputs and outputs. This can be used as a test oracle for further testing as usually modules/functions are changing during the lifetime of a project.

Especially useful in the context of iterative development and testing, when algorithms are constantly refined and re-tested
Examples of machine learning for software testing

**Test Data Generation**: automatic generation of test data (input / expected output)

Test data generation can have different objectives:

- the coverage of specific program structures, as part of a structural, or white-box testing strategy;
- the exercising of some specific program feature, as described by a specification;
- attempting to automatically disprove certain gray-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example the worst-case execution time of a segment of code.

Machine learning used in software test data generation: Hill Climbing, Simulated Annealing, Evolutionary Algorithms

**Hill Climbing** works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighborhood of this solution is investigated. If a better solution is found, then this replaces the current solution. Hill climbing is simple and gives fast results. However it is easy for the search to yield sub-optimal results when the hill climbed leads to a solution that is locally optimal, but not globally optimal.

**Simulated Annealing** is similar in principle to Hill Climbing. However, by probabilistically accepting poorer solutions, Simulated Annealing allows for less restricted movement around the search space.

**Evolutionary Algorithms** use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection. **Genetic Algorithms** are probably the most well known form of Evolutionary Algorithm. For Genetic Algorithms, the search is primarily driven by the use of **recombination** - a mechanism of exchange of information between solutions to "breed" new ones - whereas Evolution Strategies principally use **mutation** - a process of randomly modifying solutions.

The name "Genetic Algorithm" comes from the analogy between the encoding of candidate solutions as a sequence of simple components, and the genetic structure of a chromosome. Continuing with this analogy, solutions are often referred to as individuals or chromosomes. The components of the solution are sometimes referred to as genes, with the possible values for each component called alleles, and their position in the sequence the locus.
Test Data generation – White box testing – static approaches

Many approaches use the control flow graph (CFG) of a program.

```c
int tri_type(int a, int b, int c)
{
    int type;
    if (a > b)
    {
        int t = a; a = b; b = t;
    }
    if (a > c)
    {
        int t = a; a = c; c = t;
    }
    if (b > c)
    {
        int t = b; b = c; c = t;
    }
    if (a + b <= c)
    {
        type = NOT_A_TRIANGLE;
    }
    else
    {
        type = SCALENE;
        if (a == b && b == c)
        {
            type = EQUILATERAL;
        }
        else if (a == b || b == c)
        {
            type = ISOSCELES;
        }
    }
    return type;
}
```

Control dependency graph
Test Data generation – White box testing – static approaches

Symbolic Execution
Symbolic Execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure.

The technique can be used to derive a constraint system in terms of the input variables which describes the conditions necessary for the traversal of a given path.

Solutions to the constraint system are input data which will execute the path. Constraint satisfaction problems are in general NP-complete. Heuristic methods can be used in to attempt the finding of a solution.

Domain Reduction
Symbolic execution is used to develop the constraints in terms of the input variables. Domain reduction is then used to attempt a solution to the constraints.
Dynamic Structural Test Data Generation

Dynamic methods execute the program in question with some input, and then simply observe the results via some form of program instrumentation.

Goal is to generate test data to cover all the possible execution paths (white box testing).

Dynamic method overcome some limitation of the static approaches:

– presence of pointer arithmetic (*p = *q)
– arrays (ex a[i]!=a[j])

Random Testing: generate random input observe the output and the path covered.

Applying Local Search: The tester selects a path through the program, and then produces a straight-line version of it, containing only that path. The inputs are modified to obtain other paths (alter the input based on the constraint predicates).

Applying Simulated Annealing: a neighborhood structure has to be defined for the various different input variable types. For integer and real variables, the neighborhood is simply a defined range of values around each individual value. Since the ordering of values is not significant for boolean and enumerated types, all values for these variables are considered as neighbors. The objective function is simply the branch distance of the required branch when control ow diverges away from the intended path, or away from the target structure down a critical branch.

Applying Evolutionary algorithms: often referred to in the literature as Evolutionary Testing

• Coverage-Oriented Approaches: reward individuals on the basis of covered program structures.
• Structure-Oriented Approaches: A separate search is undertaken for each uncovered structure required by the coverage criterion.
Artificial neural network (ANN)

The feed-forward ANN can be used for *classification* or *prediction* scenarios. The network is trained to classify certain patterns into certain groups, and then is used to classify novel patterns which were never presented to the net before.

Artificial neural networks are generally presented as systems of interconnected "neurons" which can compute values from inputs, and are capable of machine learning as well as pattern recognition thanks to their adaptive nature.

An artificial neuron is a computational model inspired in the natural neurons. Natural neurons receive signals through synapses located on the dendrites or membrane of the neuron. When the signals received are strong enough (surpass a certain threshold), the neuron is activated and emits a signal though the axon. This signal might be sent to another synapse, and might activate other neurons.

These basically consist of inputs (like synapses), which are multiplied by weights (strength of the respective signals), and then computed by a mathematical function which determines the activation of the neuron.
Artificial neural network (ANN)

Artificial neurons are organized in layers, and send their signals “forward”, and then the errors are propagated backwards.

The network receives inputs by neurons in the input layer, and the output of the network is given by the neurons on an output layer. There may be one or more intermediate hidden layers.

The back-propagation algorithm uses supervised learning, which means that we provide the algorithm with examples of the inputs and outputs we want the network to compute, and then the error (difference between actual and expected results) is calculated.

The idea of the back-propagation algorithm is to reduce this error, until the ANN learns the training data. The training begins with random weights, and the goal is to adjust them so that the error will be minimal.
Artificial neural network (ANN) – sample implementation

Implementation propagate forward

```java
public PropagationResult forwardPropagation(float[] inValues) {
    if (inValues.length != nrInputNeurons) {
        throw new IllegalArgumentException("invalid number of input values");
    }
    PropagationResult result = new PropagationResult();
    // calculam iesirile din stratul hidden
    float[] hiddenOuts = propagateForwardOnLayer(inValues, weightInHi);
    result.setHiddenResults(hiddenOuts);
    // calculam iesirile din stratul out
    float outputs[] = propagateForwardOnLayer(hiddenOuts, weightHiOut);
    result.setOutputs(outputs);
    return result;
}

private float[] propagateForwardOnLayer(float[] inValues, float[][] weights) {
    float[] out = new float[weights[0].length];
    for (int i = 0; i < out.length; i++) {
        // initializam cu bias0ul
        float sum = weights[weights.length - 1][i];
        for (int j = 0; j < weights.length - 1; j++) {
            sum = sum + inValues[j] * weights[j][i];
        }
        out[i] = activationF.f(sum);
    }
    return out;
}
```
Artificial neural network (ANN) – sample implementation - training

```java
public float train(float[] input, float[] expectedOut) {
    PropagationResult result = netw.forwardPropagation(input);
    backPropagation(input, result, expectedOut);
    return computeError(result.getOutputs(), expectedOut);
}

private float computeError(float[] networkOut, float[] expectedOut) {
    float error = 0;
    for (int i = 0; i < netw.getNumberOfOutputNeurons(); i++) {
        error += (networkOut[i] - expectedOut[i])
                * (networkOut[i] - expectedOut[i]);
    }
    return error;
}

private void backPropagation(float[] input, PropagationResult result, 
                              float[] expectedOut) {
    float[] networkOut = result.getOutputs();
    ActivationFunction activationF = netw.getActivationF();
    // calculam eroarea pe stratul de iesire
    float[] outputDelta = new float[networkOut.length];
    for (int i = 0; i < networkOut.length; i++) {
        outputDelta[i] = (expectedOut[i] - networkOut[i])
                         * activationF.fDeriv(networkOut[i]);
    }
    // calculam eroarea pe stratul ascuns
    float[] networkHiddenOut = result.getHiddenResults();
    float[] hiddenDelta = new float[networkHiddenOut.length];
    for (int i = 0; i < netw.getNumberOfHiddenNeurons(); i++) {
        float sum = 0;
        for (int j = 0; j < networkOut.length; j++) {
            sum += netw.getWeightHiddenOut(i, j) * outputDelta[j];
        }
        hiddenDelta[i] = activationF.fDeriv(networkHiddenOut[i]) * sum;
    }
    // actualizam ponderile hiddenOutput
    for (int i = 0; i < networkOut.length; i++) {
        for (int j = 0; j < netw.getNumberOfHiddenNeurons(); j++) {
            momentumHiOut[j][i] = learningRate * outputDelta[i]
                                  * networkHiddenOut[j] + alfa * momentumHiOut[j][i];
            float aux = netw.getWeightHiddenOut(j, i) + momentumHiOut[j][i];
            netw.setWeightHiddenOut(j, i, aux);
        }
        // actualizez si pentru bias
        momentumBiasHiOut[i] = learningRate * outputDelta[i] + alfa
                               * momentumBiasHiOut[i];
        float aux = netw.getBiasInHidden(i) + momentumBiasHiOut[i];
        netw.setBiasInHidden(i, aux);
    }
}
```
Use of Artificial neural network (ANN) in software testing

ANN Test oracle
Artificial neural networks can be used to compute a function using example inputs and outputs

For an existing function one can use ANN to "learn" a function(method) and use the trained ANN as a test oracle.
In an iterative development scenario this can be useful to test new versions of the method
Triangle function example: given the length of each side the function decides if the triangle is isosceles, scalene, equilateral, or invalid
triangleType(int a, int b, int c)

ANN structure:
Input Layer: 3 input neurons (length side 1, length side 2, length side 3)
Output Layer: 4 output neurons (isosceles, scalene, equilateral, or invalid)
Hidden Layer (Optional)

Training data generation
We invoke the triangleType function for random values and obtain the result. So the train data will consist of several test data:
input value of a, value of b, value of c
expected output: [0/1, 0/1, 0/1, 0/1] we put 1 to the corresponding triangle type and 0 for the rest

Training
We scale the input values (range 0-1) and train the network.

Prediction:
for any input data (a, b, c) we can propagate forward the scaled values.
On the output layer we get real values (probabilities intuitively) for the triangle being isosceles, scalene, equilateral, or invalid.