Lecture 2 Machine learning in Software Engineering

- Existing approaches
- Reformulate software engineering problems
- Evaluation criteria for search based software engineering

Lecture 1 Search-based Software Engineering

- Search-based Software Engineering
- Machine learning
  - techniques
- Software Engineering
  - subdisciplines
  - activities
  - artifacts (UML, source code, source repository, test data, runtime data,..)
Machine learning in Software Engineering

ML algorithms can be used in tackling software engineering problems.

ML algorithms not only can be used to build tools for software development and maintenance tasks, but also can be incorporated into software products to make them adaptive and self-configuring.

A maturing software engineering discipline will definitely be able to benefit from the utility of ML techniques.

Search Based Software Engineering

Reformulating software engineering problems as search problems
Machine learning approaches for software engineering

- Requirement engineering - AL, BBN, LL, DT, ILP
- Rapid prototyping - GP
- Component reuse - IBL (CBR4)
- Cost/effort prediction - IBL (CBR), DT, BBN, ANN
- Defect prediction – BBN, AR
- Design defect detection - AR
- Test oracle generation - AL (EBL5)
- Test data adequacy - CL
- Validation – AL
- Restructuring/Refactoring CU
- Design patterns – CU
- Data structures – ANN, SVM
- Reverse engineering – CL

Major types of learning:
- concept learning (CL)
- decision trees (DT)
- artificial neural networks (ANN)
- Bayesian belief networks (BBN)
- reinforcement learning (RL)
- genetic algorithms (GA) and genetic programming (GP)
- instance-based learning (IBL)
- inductive logic programming (ILP)
- analytical learning (AL)
- support vector machines (SVM)
- association rules (AR)
- clustering (CU)
Component reuse

Component retrieval from a software repository is an important issue in supporting software reuse.

Formulated into an instance-based learning problem:
1. Components in a software repository are represented as points in the n-dimensional Euclidean space (or cases in a case base).

2. Information in a component can be divided into indexed and unindexed information (attributes). Indexed information is used for retrieval purpose and unindexed information is used for contextual purpose.

3. Queries to the repository for desirable components can be represented as constraints on indexable attributes.

4. Similarity measures for the nearest neighbors of the desirable component can be based on the standard Euclidean distance, distance-weighted measure, or symbolic measure.

5. The possible retrieval methods include: K-Nearest Neighbor, inductive retrieval, Locally Weighted Regression.

6. The adaptation of the retrieved component for the task at hand can be structural (applying adaptation rules directly to the retrieved component), or derivational (reusing adaptation rules that generated the original solution to produce a new solution).
Rapid prototyping/Generating programs
Rapid prototyping - tool for understanding and validating software requirements.

In genetic programming (GP), a computer program is often represented as a tree:
- nodes are functions
- leaf nodes are input to functions
Start with a random generated tree (program), GP generates the final computer program.

The framework of a GP-based rapid prototyping process can be described as follows:

1. Define sets of functions and terminals to be used in the developed (prototype) systems.

2. Define a fitness function to be used in evaluating the worthiness of a generated program. Test data (input values and expected output) may be needed in assisting the evaluation.

3. Generate the initial program population.

4. Determine selection strategies for programs in the current generation to be included in the next generation population.

5. Decide how the genetic operations (crossover and mutation) are carried out during each generation and how often these operations are performed.

6. Specify the terminating criteria for the evolution process and the way of checking for termination.

7. Translate the returned program into a desired programming language format.
**Requirement engineering**

Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate.

A requirement may be functional or non-functional. A functional requirement describes a system service or function, whereas a non-functional requirement represents a constraint imposed on the system.

Functional requirements can be “learned” from the data if there are empirical data from the problem domain that describe how the system should react to certain inputs.

1. Let X and C be the domain and the co-domain of a system function f to be learned. The data set D is defined as: D = {<xi, ck>| xi in X and ck in C}.

2. The target functions f to be learned is such that any xi in X and any ck in C, f(xi) = ck.

3. The learning methods applicable here have to be of supervised type. Depending on the nature of the data set D, different learning algorithms (in AL, BBN, CL, DT, ILP) can be utilized to capture (learn) a system’s functional requirements.
Reverse engineering (program comprehension and understanding)

Recover the design or specification of a legacy system from its source or executable code

Legacy systems are old systems that are critical to the operation of an organization which uses them and that must still be maintained. They may be poorly structured and their documentation may be either out-of-date or non-existent.

Deriving functional specification of a legacy software system from its executable code.

1. Given the executable code $p$ and its input data set $X$, and output set $C$, the training data set $D$ is defined as: $D = \{ <x_i, p(x_i)> | x_i \text{ in } X \text{ and } p(x_i) \text{ in } C \}$.

2. The process of deriving the functional specification $f$ for $p$ can be described as a learning problem in which $f$ is learned through some ML algorithm such that any $x_i$ in $X \ [ f(x_i) = p(x_i) ]$.

3. Many supervised learning methods can be used here (e.g., CL).
**Validation**

Check a software implementation against its specification

If the specification and the executable code is available validation can be performed as an analytic learning task as follows:

1. Let $X$ and $C$ be the domain and co-domain of the implementation (executable code) $p$, which is defined as: $p: X \rightarrow C$.

2. The training set $D$ is defined as: $D = \{<x_i, p(x_i)> | x_i \in X \}$.

3. The specification for $p$ is denoted as $B$, which corresponds to the domain theory in the analytic learning.

4. The validation checking is defined to be: $p$ is valid if any $<x_i, p(x_i)>$ in $D [B \text{ and } x_i \text{ imply } p(x_i)]$.

5. Explanation-based learning algorithms can be utilized to carry out the checking process.
Software defect prediction

predict the likely delivered quality and maintenance effort before software system is deployed

size, complexity, testing metrics, proces quality must be taken into consideration in order for the defect prediction to be successful.

compute the probability distribution for any subset of variables given the values or distributions for any subset of the remaining variables - Bayesian Belief Networks (BBN)

specifies the probability that the variable will take on each of its possible values (e.g., “very low”, “low”, “medium”, “high”, or “very high” for the variable “Defects Detected”) given the observed values of the other variables (complexity, metrics, etc)

When using a BBN for a decision support system such as software defect prediction, the steps below should be followed.

1. Identify variables in the BBN. Variables can be:
   - hypothesis variables for which the user would like to find out their probability distributions (hypothesis variable are either unobservable or too costly to observe),
   - information variables that can be observed
   - mediating variables that are introduced for certain purpose (help reflect independence properties, facilitate acquisition of conditional probabilities, and so forth).

2. Define the proper causal relationships among variables. These relationships also should capture and reflect the causality exhibited in the software life-cycle processes.

3. Acquire a probability distribution for each variable in the BBN.
**Software defect prediction**

use any supervised learning technique

- Collect data (measurement, metric, resource allocation) about defective systems, components, classes, modules. The measurements will be used as attribute values

- train the model using well known defective/un-defective system/component/class

- The model can be used to predict defect rate for new systems (other than the ones used in the training phase)
Project effort (cost) prediction

approach to the project effort prediction using instance-based learning.

1. Introduce a set of features or attributes (e.g., number of interfaces, size of functional requirements, development tools and methods, and so forth) to characterize projects.

2. Collect data on completed projects and store them as instances in the case base.

3. Define similarity or distance between instances in the case base according to the symbolic representations of instances (e.g., Euclidean distance in an n-dimensional space where n is the number of features used). To overcome the potential curse of dimensionality problem, features may be weighed differently when calculating the distance (or similarity) between two instances.

4. Given a query for predicting the effort of a new project, use an algorithm such as Knearest Neighbor, or, Locally Weighted Regression to retrieve similar projects and use them as the basis for returning the prediction result.
Software restructuring/refactoring/transformation

Model the system as a set of objects.  
Object can be a class, a module, a function/method, a package.

Use a clustering algorithm to group the objects.

Using the obtained partition one can:

- identify the new/better structure of the system
- derive transformations for the analyzed system
- identify components, concepts or other higher level abstraction in the source code (i.e. design patterns)

Object in the software system can be described by a set of features (i.e. software metric) -> we can use Euclidian or other distances
We can define custom distances between the objects. We can encapsulate domain knowledge in the definition of the distance used in the clustering algorithm.
Some Existing Work

Scenario-based requirement engineering
Formal method for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders.

The method is based on a learning algorithm that takes scenarios as examples and counter-examples (positive and negative scenarios) and generates goal specifications as temporal rules.
Software project effort estimation

Instance-based learning techniques are used in for predicting the software project effort for new projects. The empirical results obtained (from nine different industrial data sets totaling 275 projects) indicate that cased-based reasoning offers a viable complement to the existing prediction and estimations techniques.

Decision trees (DT) and artificial neural networks (ANN) are used to help predict software development effort. The results were competitive with conventional methods such as COCOMO and function points. The main advantage of DT and ANN based estimation systems is that they are adaptable and nonparametric.
**Software defect prediction**

Bayesian belief networks are used to predict software defects, incorporating multiple perspectives on defect prediction into a single, unified model.

Variables are chosen to represent the life-cycle processes of
- specification
- design
- implementation
- testing
Evaluation criteria for search based software engineering

Criterias that capture essential aspects/goals for any search based approach

**Base line validity**
The solution should be able to find better solutions or find the in less computational effort than random search
In some case maybe we just have to change the choice of search technique

**Discovery of known solutions**
There are situations where there is no known general algoritm for solving a problem.
We can validate the search-based approach by showing that the metaheuristic search is able to find solutions that compare well with known individual solutions.

**Discovery of desirable solutions**
Gather empirical data to provide evidence of the kind of solution which may be obtained.

**Efficiency**
In many cases a search-based approach is slower than existing analytical approaches (search involve repeated trails)
If the search-based approach produce better solutions then it can be a valuable tool where quality overrides speed.

**Validation with respect to existing analytical techniques (no search-based)**
Even if there are non search-based approaches the search-based variant may still be useful if existent approaches:
- only work on subset of the problem space
- solution is not consistent, we can use the search-based variant as a "second guess"
- only produce sub-optimal solutions, we can use search-based variant to improve on the known solution or to produce better solutions

**Psychological consideration**
aesthetic application of metaheuristic search
(semi) automatization of common software engineering task
We can use search-based to better understand solutions