NEW TRENDS IN LEARNING FOR SOFTWARE ENGINEERING

Alaa Hamouda
Department of Computer Engineering, Engineering Faculty,
Al-Azhar University, Egypt
alaa.hamouda@azhar.edu.eg

***Abstract*— Software is nowadays a critical component of our lives and everyday-work working activities. However, as the technological infrastructure of the modern world evolves a great challenge arises for developing high quality software systems with increasing size and complexity. Software engineers and researchers are striving to meet this challenge by developing and implementing software engineering methodologies able to deliver software products of high quality, within budget and time constraints. The field of machine learning in software engineering has recently emerged to provide means for addressing, studying, analyzing, and understanding critical software development issues and at the same time to offer mature machine learning techniques such as artificial neural network, Bayesian networks, decision trees, fuzzy logic, genetic algorithms, and rule induction. Machine learning algorithms have proven to be of great practical value to software engineering. Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development tasks could be formulated as learning problems and approached in terms of learning algorithms. In this paper, we first take a look at the characteristics and applicability of some frequently utilized machine learning algorithms. We then present the application of machine learning in the different phases of software engineering that include project planning, requirements analysis, design, implementation, testing and maintenance.**

Keywords*— machine learning, software engineering, learning algorithms

I. INTRODUCTION

The software intensive system become much more complex in terms of the number of functional and nonfunctional requirements they need to support. The impact of low quality can also have a catastrophic impact on the mission of these systems in many critical applications. Moreover, the cost of software development dominates the total cost of such systems. Machine learning is the study of how to build computer programs that improve their performance at some task through experience. Research in applying machine learning techniques to software Engineering have grown tremendously in the last two decades producing a large number of projects and publications [1].

The hallmark of machine learning is that it results in an improved ability to make better decisions. Machine learning algorithms have proven to be of great practical value in a variety of application domains. Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development and maintenance tasks could be formulated as learning problems and approached in terms of learning algorithms. To meet the challenge of developing and maintaining large and complex software systems in a dynamic and changing environment, machine learning methods have been playing an increasingly important role in many software development and maintenance tasks. The past two decades have witnessed an increasing interest, and some encouraging results and publications in machine learning application to software engineering. As a result, a crosscutting niche area emerges. Currently, there are efforts to raise the awareness and profile of this crosscutting, emerging area, and to systematically study various issues in it [2].

The machine learning techniques are proposed in order to reduce the time to market and enhance the quality of software systems. Yet many of these machine learning techniques remain largely used by the research community and with little impact on the processes and tools used by the practicing software engineer. The recent survey papers published in this field are mainly targeted to the research community. They are driven by the specific machine learning techniques used rather than the software engineering activities supported [3]. This survey paper attempts to cover the research and practice of applying machine learning techniques to the software engineering processes. It also highlights open practical problems to the research community in applying such techniques by surveying the recently proposed work in this area.

The paper is organized as follows. Section 2 focuses on the application of machine learning algorithms in software engineering phases; project planning, analysis, design, implementation, testing, and maintenance. Section 3 highlights the current software engineering open problems candidate for applying machine learning. Finally, section 4 comes to conclude the paper.

II. MACHINE LEARNING IN SOFTWARE ENGINEERING

There is an increased interest in the niche area of machine learning and software engineering [4]. In applying machine learning to solving any real-world problem, there is usually some course of actions. It normally follows the following steps [1]:

Problem formulation: The first step is to formulate a given problem such that it conforms to the framework of a particular learning method chosen for the task.

Problem representation: The next step is to select an appropriate representation for both the training data and the knowledge to be learned. Different learning methods have different representational formalisms.

Data collection: The third step is to collect data needed for the learning process. The quality and the quantity of the data needed are dependent on the selected learning method.

Domain theory preparation: Certain learning methods rely on the availability of a domain theory for the given problem. How to acquire and prepare a domain theory (or background knowledge) and what the quality of a domain theory (correctness, completeness) is therefore become an important issue that will affect the outcome of the learning process.

Performing the learning process: Once the data and a domain theory (if needed) are ready, the learning process can be carried out. The data will be divided into a training set and a test set.bKnowledge induced from the training set is validated on the test set.

Analyzing and evaluating learned knowledge: Analysis and evaluation of learned knowledge is an integral part of the learning process. The interestingness and the performance of the acquired knowledge will be scrutinized during this step, often with the help from human experts, which hopefully will lead to the knowledge refinement.

Fielding the knowledge base: what this step entails is that the learned knowledge is used. The knowledge could be embedded in a software development system or a software product, or used without embedding it in a computer system.
The power of machine learning methods does not come from a particular induction method, but instead from proper formulation of the problems and from crafting the representation to make learning tractable. The phases of software development have already witnessed the use of machine learning algorithms. In this section, we discuss the machine learning usages in these different phases.

A. Project Planning

Software project failures have been an important subject in the last decade. Software projects usually don't fail during the implementation and most project fails are related to the planning and estimation steps. Inaccurate estimation is a real problem in the software production's world which should be solved. Presenting the efficient techniques and reliable models seems required regarding the mentioned problem. The conditions of the software projects are not stable and the state is continuously changing so several methods should be presented for estimation that each method is appropriate for a special project [5]. Of the types of software engineering issues people are interested in applying machine learning techniques to, twenty-eight out of the collected sixty publications (almost 47%) deal with the issue of how to build models to predict or estimate certain property of software development process or artifacts [6].

1) Size Estimation

Software managers need to make important decisions in the early stages of a project. To help them in this difficult task, prediction models and the experience of passed projects are fundamental. Software size metrics play a significant role to the success of this task. The most known and used software size metric is lines of code (LOC). This metric is easy to understand and, after the development, is also easy to collect and be included in a historical database. However, during the software management, LOC estimation is necessary. There are different approaches to the LOC estimation problem. Machine learning can help estimate the LOC using 1) function point sizing: uses the number of function points (FP) that considers domain characteristics and the functionalities of the software and 2) component sizing: uses the number of components (NOC) of the application, such as subsystems, modules, screens, reports, files, etc. Neural network is used to estimate the predicted LOC using function points, project domain, and number of components types as input.

Statistical regression methods are traditionally used to obtain an equation for LOC estimation from FP and NOC. To apply these methods, the user must provide the format for the equation to be obtained. He has to decide to do a linear, a quadratic, a higher-order polynomial regression or whether to try to fit the data points to some other non-polynomial family of functions. To overcome this limitation, Machine Learning techniques (ML) have emerged as a suitable approach for the regression problem in the software size estimation [7].

The idea is to obtain an equation to estimate LOC for each mentioned approach (FP and NOC), using Genetic Programming (GP) and Neural Networks (NN). The main motivation to choose these techniques for this task is their capability of learning from historical data, discovering a solution with different variables and operators, being robust with respect to noisy data. The models are analyzed and the ML techniques are compared [8]. The input features include the number of menu components, number of input components, number of output components, function points, and Project domains.

2) Cost Estimation

Project planning is one of the most important activities in software projects. Poor planning often leads to project faults and dramatic outcomes for the project team. If cost and effort are determined pessimistic in software projects, suitable occasions can be missed; whereas optimistic predictions can be caused to some resource losing. The main reason for this problem is imprecision of the estimation [9]. Machine learning algorithms are good techniques for cost estimation depending on previous data. Neural networks are used for cost estimation.

Majority of researches on using the neural networks for software cost estimation, are focused on modeling the Cocomo method. Cocomo depends on some factors that affect

the effort and cost of the project. For example in [10] a neural network has been proposed for estimation of software cost. Scale Factors (SF) and effort multipliers (EM) as input of the neural network. Examples of effort multipliers include the degree of flexibility in the development process and the process maturity of the organization. Scale factors include several parameters like required percentage of reusable components and complexity of system modules

B. Requirements

Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate. The situation in which machine learning algorithms will be particularly useful is when there exist empirical data from the problem domain that describe how the system should react to certain inputs. One of the important usage of machine learning in requirements phase is to generate use cases from the description of the system. An automated software development was required to acquire the preference directly from user and generate corresponding Use Cases with improved output in minimum time consumed. The major emphasis of [11] was automatic identification of objects from a problem domain. User provides the input text in English language related to the business domain. After the lexical analysis of the text, syntax analysis is performed on word level to recognize the word category. The available lexicons are categorized into nouns, pronouns, prepositions, adverbs, articles, conjunctions, etc. The syntactic analysis of the programs would have to be in a position to isolate subject, verbs, objects, adverbs, adjectives and various other complements. It is little complex and multipart procedure. A procedure is defined that understands these sentences and discover the actor, finds interaction, goals and intended objects. The system has the following major modules:

- Tokenization

This module reads the natural language text and converts the text into lexical tokens.

- Part Of Speech Tagging

The major function of this module is to identify and extract the various parts of speech in the given text.

- Meaning Understanding

It understands the semantics of the given test and on the basis of extracted semantics, the object, subject and adverb parts of the sentences are identified. Table 1 shows the output as an example. The block diagram of the system is illustrated in figure 1.

Table 1: Example for identifying actor, action, and object

Lexicons	Phase-I	Phase –II
User	Noun	Actor
fills	Verb	Action
the	Article	-------
form	Noun	Object

About 50 different real-time scenarios were provided to the system to check its accuracy. 85% of the scenarios were accurately translated into use case diagram. Accuracy if further improves if the extracted information is filtered before generating the diagrams.

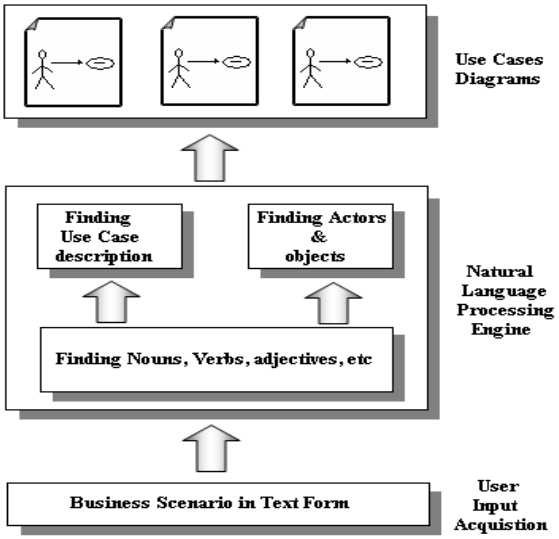

Figure 1: Block diagram of the use case generator system

C. Design

One of the most important problems facing the software engineer is to develop quality architecture from the requirements model. In this section we describe recent work on software design using machine learning techniques in components re-use and user interface design.

1) Components Re-use

There is usually a repository for components to be reused. Component retrieval from a software repository is an important issue in supporting software reuses [12]. This task can be formulated into an instance-based learning problem as follows:

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).
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 and inductive retrieval.
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).

Such Case Based Reasoning (CBR) can be augmented with additional mechanisms to help aid other issues in reuse library.

A technique called active browsing is incorporated into a tool that helps assist the browsing of reusable library for desired components [13]. An active browser infers its similarity measure from a designer's normal browsing actions without any special input. It then recommends to the designer components it estimates to be close to the target of the search, which is accomplished through a learning process.

2) User Interface Design

It is well-accepted that learnability is an important aspect of usability, yet there is little agreement as to how learnability should be defined, measured, and evaluated. Over the past three decades, various concepts, methodologies, and components of usability have matured into an abundant body of arguably well-accepted usability engineering methodologies. However, even when developed while following the well-established methodologies, interfaces still possess usability problems. In a recent study, Lazar et al. found that users lose up to 40% of their time due to "frustrating experiences" with computers, with one of the most common causes of these frustrations being missing, hard to find, and unusable features of the software [14].
Part of the difficulty is that interface usage requires learning. Indeed, there is a consistent agreement that learnability is an important component of usability.

This learnability is employed in the web applications design. A good design and organization of a website is essential in improving the website's attractiveness and popularity in Web applications. However, it is not an easy task for every Web designer to satisfy the aims initially designated. There are many reasons, which are associated with the above disappointments. First, different users have their own navigational tasks so following different access traces. Second, even the same user may have different information needs at different times. Third, the website is not logically organized and the individual web pages are aggregated and placed in inappropriate positions, resulting in the users uneasily locating the needed information; and furthermore, a site may be designed for a particular kind of use, but be used in many different ways in practice; the designer's original intent is not fully realized. All above mentioned reasons affects the satisfactory degree of a website use. By deeply looking into the causes of such reasons, we can intuitively see that it is mainly because the early website design and organization only reflects the intents of website designers or developers, instead, the user opinions or tastes are not sufficiently taken into account. Inspired by this observation, using Web Usage Mining techniques is intuitively proposed to address the improvement of website design and organization. Essentially the knowledge learned from Web Usage Mining is able to reveal the user navigational behavior and to benefit the site organization improvement by leveraging the knowledge [15, 16, 17, and 18].

In [19] the authors proposed an approach to address this challenge by creating adaptive websites. The approach is to allow web sites automatically improve their organization and presentation by learning from visitor access patterns. Different from other methods such as customized Web sites, which are to personalize the Web page presentation to individual users, the proposed approach is focused on the site optimization through the automatic synthesis of index pages. The basic idea of the proposed is originated from learning the user access patterns and implemented by synthesizing a number of new index pages to represent the user access interests.

D. Implementation

Implementation is a core process in the software engineering life cycle. One of the challenges in this phase is the modularization. Another one is refactoring. In this section, these two main issues are discussed.

1) Modularization

Mancoridis et al. were the first to address the problem of software modularization using genetic algorithms leading to the development of a tool called Bunch [20] for module clustering. The objective is to improve the module quality. All versions of module quality are combinations of cohesion and coupling into a single weighted fitness function, used to guide the search. Other authors explored using genetic programming to capture cohesion and coupling in different metrics [21 and 22]. As well as providing candidate module structures, the multi objective approach, as with all multiple objectives [23], allows the engineer to explore the tradeoffs and tensions between the objectives, something often advocated as an advantage.

2) Refactoring

Refactoring is a state-of-the-art practice in software development to improve the design of existing software systems without changing the external behavior [24]. Developers often use this technique to prepare object-oriented systems for further improvements and extensions of functionality [25]. For project managers it is interesting to know which locations are likely to demand refactoring. With limited time and resources project managers want to focus on the most important refactorings [26]. The right time for a general refactoring can be better determined with machine learning based approach.

[27] screen evolution data from versioning systems of two open source projects where refactoring is recognized as important engineering activity for software evolution. As a result they group the features into different categories:

- Size

This category contains size measures such as lines of code from an evolution perspective: linesAdded, linesModified, or linesDeleted relative to the total LOC (lines of code) of a file.

- Team

The number of authors of files influences the way software is developed. They expect that the more authors are working on the changes the higher the probability of rework and mistakes. They define a feature for the authorCount relative to the changeCount.

- Work habits

To get estimation for the work habits of the developers, they inspect the number of addingChanges, modifyingChanges, and deletingChanges per author and per file. This information provides input to the defect prediction of files.

- Complexity of existing solution

Changes are more difficult to add as the software is more difficult to understand and the contracts between existing parts have to retain. As a result they investigate the changeCount in relation to the number of changes during the entire history of each file.

- Difficulty of problem

They use the information whether a file was newly introduced during the prediction period as feature for machine learning: To measure how often a file was involved during development with the introduction of other new files they use coChangeNew as a second indicator.

- Relational Aspects

They use the co-change coupling between files to estimate their relationship. The first features of this category are couplings such as the number of changes/revisions where other files have been committed with. They use the number of co-changed files relative to the change count of the learning period as feature coChangedFiles.

- Time constraints

As software processes stress the necessity of certain activities and artifacts, they believe that the time constraints are important for software predictions. They measure the average number of days between revisions and the number of days per line of code added or changed.

Prediction Target: Refactoring Proneness

With the described features, they predict the number of refactorings. The prediction models are based on two class problems, where in each application they group files in one of the two classes: having refactoring vs. without refactoring, having one refactoring vs. having several refactorings, figure 2. In the field of machine learning this procedure is called classifier stacking.

The number of refactorings is obtained from the commit messages of the versioning system. They use evolution data not only for the computation of machine learning features, but also for the identification of change events as refactorings. Decision tree and neural network are used as classifiers. The F-measure was about 65%.

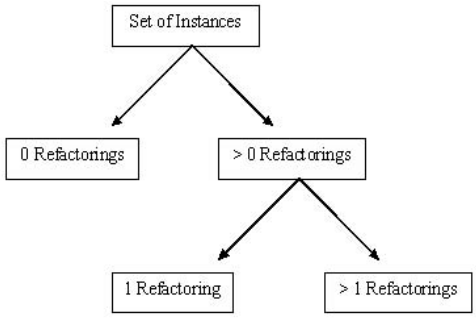

Figure 2: Refactoring Analysis Setup

E. Testing

A software quality model aims to find the underlying relationship between software measurements and software quality [28]. Early detection of fault-prone software components enables verification experts to concentrate their time and resources on the problem areas of the software system under development [29]. Software quality models, thus, help ensure the reliability of the delivered products. I

The basic hypothesis of software quality prediction is that a module currently under development is fault prone if a module with the similar product or process metrics in an earlier project (or release) developed in the same environment was fault prone [30]. Therefore, the information available early within the current project or from the previous project can be used in making predictions. This methodology is very useful for the large-scale projects or projects with multiple releases.

To meet the needs of testers, fault-prone prediction models must be efficient and accurate. Many modeling techniques have been developed and applied for software quality prediction. These include: logistic regression, artificial neural network, fuzzy classification, genetic algorithms, and classification trees [31].

A software quality model can be considered superior over its counterparts if it has both a higher defect detection rate, and a higher overall accuracy. About 65-75% of critical modules and non-fault-prone modules were correctly predicted in [31] using the overall accuracy criterion. Decision trees correctly predicted 79.3% of high development effort fault-prone modules (detection rate), while the trees generated from the best parameter combinations correctly identified 88.4% of those modules on the average. In one case study, among five common classification techniques: classification trees, factor-based discriminant analysis, fuzzy classification, and neural network, fuzzy classification appears to yield better results. Since most of these studies have been performed using different datasets, reflecting different software development environments and processes, the final judgment on “the best” fault-prone module prediction method is difficult to make.

Several used dataset contains 21 software metrics, which describe product's size, complexity, and some structural characteristics. Each module is measured in terms of the same software product metrics. A class label is associated with each module, indicating if the module is defect-free or fault-prone. Defects were detected during the development or in the deployment.

F. Maintenance

Software maintenance is widely recognized to be the most expensive and time-consuming aspect of the software process [32 and 33]. To maintain a system properly one needs to have at least a partial understanding of the system; this, in turn, requires knowing about the relationships among elements of the system and to accurately estimate the needed efforts and time. The following section will focus on these two important maintenance issues.

1) Relevance Discovery

The complexity of a software system is clearly related to the complexity of the network of relationships in the system. A key tenet of the research is to know what elements of the system are related to each other. An abstraction called the relevance relation is used to represent relationships that would be useful for a maintainer to know, figure 3. A relevance relation maps a tuple of system elements to a value indicating how related they are. Each tuple in a relevance relation therefore contains a set of related elements.

Well-managed software projects maintain repositories that keep track of software revisions and use mechanisms to record software problem reports as well as the changes applied to the software to address these problems. Number of Shared Directly Referred Types and Number of Shared Routines Directly Referred are examples of the main attributes to be input of the relevance discovery process. Conceptually, machine learning systems learn models from past experience that can be applied in future unseen situations or scenarios.

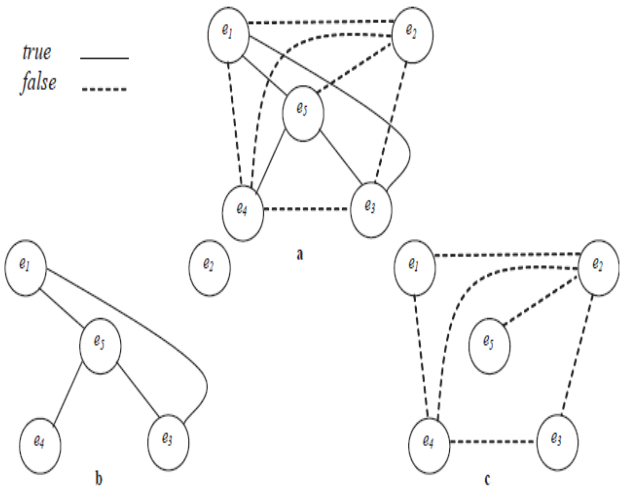

Figure 3: Relevance Discovery

1) Maintenance Effort Prediction

Predictions of effort needed to carry out software maintenance tasks may be an important input to maintenance managers when planning maintenance activities and performing cost benefit analyses. Some research papers have been published describing the development and evaluation of formal models to predict software maintenance task effort [l, 2] and [3].

In [34], the focus is on the development and the relative accuracy of different types of software maintenance task effort prediction models. [34] collected data on 109 randomly selected maintenance tasks in a large organization with about 20,000 employees. The 110 maintainers of the organization maintained more than 70 applications. The size of the applications varied from a few thousand lines of code (LOC) to about 500,000 LOC, and the age of the applications varied from less than a year to more than 20 years. Among others, the following data were collected for each maintenance task:

- Type of maintenance task, (corrective or perfective)
- Priority of task (high, medium or low priority)
- Maintainer's knowledge and confidence about how to solve the task immediately after having read or heard the task specification
- Years of experience as maintainer
- Education level of the maintainer.
- Work-hours (effort) spent on the task.
- Task size and the programming languages used.
- Types of change on code (deletion of module, change of interface, change of control flow, change of data declarations, change of data or assignment statement)
- Information sources used (communication with the users of the application, user documentation, language or tool documentation, communication with system personnel)
- Age and size of the changed application.
- Informal description of the task.

[34] chose a size measure based on LOC. Size of a maintenance task was defined as:

Size = LOC Inserted + LOC Updated + LOC Deleted.

Neural network and regression were used as approaches for effort prediction. The prediction accuracy was acceptable.

III. Open Problems

Most of presented work is immature and a lot of related issues are still open. However, there are other open problems in which machine learning can play a great role. For example, machine learning can help in the requirements engineering phase in developing knowledge based systems and ontologies to manage the requirements and model problem domains [3].

Ontologies are developed by many organizations to reuse, integrate, and merge data and knowledge and to achieve interoperability and communication among their software systems. Semantic web and ontological techniques can be used to elicit, represent, model, analyze and reason about knowledge and information involved in requirements

engineering processes. Although there are some researchers have developed the Ontology-based software Development Environment (ODE) based on software process ontology, it still needs a lot of work to be mature and practical.

One of the most difficult problems is the problem of transforming requirements into architectures. Much research is needed in this area to address the ever increasing complexity of functional and non-functional requirements. Recent important research problems are developing product line architectures and service-oriented architectures.

One area that has received some attention is the use of automated algorithms with machine learning to make repair assignments. Each bug report contains a substantial amount of information. The one-line summary and full text description can be used to characterize each report as they uniquely describe each report. In any case, more studies with respect to the appropriate criteria for selecting assignment policy, reward mechanisms and management goals need to be undertaken.

IV. Conclusion

In this paper, we have discussed issues and current status regarding machine learning applications to software engineering problems. The existing work certainly proves that the field of software engineering is a fertile ground for the application of machine learning methods. It is clear that there is an increased interest in the niche area of machine learning and software engineering.

Machine learning methods can be used to complement existing software engineering tools and methodologies to make headway in all aspects of essential difficulties. The strength of machine learning methods lies in the fact that they have sound mathematical and logical justifications and can be used to create and compile verifiable knowledge about the design and development of software artifacts. This has been demonstrated in the body of the discussed work.

Neural network, case based reasoning, and decision tree are the top three popular machine learning techniques people feel comfortable in using.

The power of machine learning methods does not come from a particular induction method, but instead from proper formulation of the problems and from crafting the representation to make learning tractable.

Analysis and evaluation of learned knowledge is an integral part of the learning process. There are known practical problems in many learning methods such as overfitting, local minima, or curse of dimensionality that are due to either data inadequacy, noise or irrelevant attributes in data, nature of a search strategy, or incorrect domain theory.

Machine learning plays a good role in the different phases of software engineering; project planning, requirements analysis, design, implementation, testing, and even maintenance. It is expected that this interest in applying machine learning in software engineering tasks will increase significantly especially with the increase interest in the empirical software engineering.

References

[1] Du Zhang, Jeffrey J.P. Tsai, Machine Learning and Software Engineering, Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02), 2002.

[2] Du Zhang, Jeffrey J.P. Tsai, Advances in Machine Learning Applications in Software Engineering, idea group publishing, 2007.

[3] Hany Ammar and et al, Software Engineering Using Artificial Intelligence Techniques: Current State and Open Problems, The Second International Conference on Communications and Information Technology, Feb 2012.

[4] Abdel Salam Sayad and Hany Ammar, Pareto-optimal search-based software engineering (POSBSE): A literature survey, In 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2013), San Francisco, CA, USA, 2013.

[5] Vahid Khatibi, Dayang N. A. Jawawi, Software Cost Estimation Methods: A Review, Journal of Emerging Trends in Computing and Information Sciences, Vol. 2, No. 1, 2011.

[6] K. Moløkken and M. Jørgensen, A review of surveys on software effort estimation, International Symposium on Empirical Software Engineering, 2003. pp. 223-231.

[7] José Javier Dolado, A Validation of the Component-Based Method for Software Size Estimation, Journal of IEEE Transactions on Software Engineering - special section on current trends in exception handling—part II, Vol. 26, No. 10, October 2000. pp 1006-1021.

[8] Evandro N. Regolin and et al, Exploring Machine Learning Techniques for Software Size Estimation, IEEE International Conference of the Chilean Computer Science Society, 2003.

[9] A. B. Nassif, D. Ho and L. F. Capretz, Towards an early software estimation using log-linear regression and a multilayer perceptron model, Journal of Systems and Software, 2012.

[10] Attarzadeh, I. Siew Hock Ow, Proposing a New Software Cost Estimation Model Based on Artificial Neural Networks, IEEE International Conference on Computer Engineering and Technology (ICCET), Vol. 3, 2010. pp 487-491.

[11] Imran Sarwar Bajwa, and S. Irfan Hyder, UCD-Generator – A LESSA Application for Use Case Design, IEEE International Conference on Information and Emerging Technologies ICIET 2007, Karachi, Pakistan, 2007, pp 182-187.

[12] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang, Search based software engineering: Trends, techniques and applications, ACM Computing Surveys, Vol. 45, No. 11, November 2012.

[13] C. Drummond, D. Ionescu and R. Holte, A learning agent that assists the browsing of software libraries, IEEE Transaction in Software Engineering, Vol. 26, No. 12, December 2000. pp. 1179-1196.
[14] J. Lazar, A. Jones, and B. Shneiderman, Workplace user frustration with computers: An exploratory investigation of the causes and severity, Behavior and Information Technology, Vol. 25, No. 3, 2006. pp 239-251.
[15] Bettina Berendt, Laura Hollink, Vera Hollink, Markus Luczak-R□osch, Knud M□oller, and David Vallet, Usage analysis and the web of data, Special Interest Group on Information Retrieval Forum, Vol. 45, No. 1, May 2011. pp 63 -69.
[16] Vera Hollink, Theodora Tsikrika, and Arjen P. de Vries, Semantic search log analysis: A method and a study on professional search, Journal of the American Society for Information Science and Technology, Vol. 62, No. 4, 2011. pp 691-713.
[17] Julia Hoxha, Martin Junghans, and Sudhir Agarwal, Enabling semantic analysis of user browsing patterns in the web of data, Computing Research Repository, 2012. pp 1204-2713.
[18] Jerey Pound, Peter Mika, and Hugo Zaragoza, Ad-hoc object retrieval in the web of data, In Proceedings of the 19th international conference on World Wide Web, New York, USA, 2010. pp 771-780.
[19] M. Perkowitz and O. Etzioni, Adaptive web sites: automatically synthesizing web pages, proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, Menlo, Park, CA, USA, 1998. pp 727–732. American Association for Artificial Intelligence.
[20] Brian S. Mitchell and Spiros Mancoridis, On the Automatic Modularization of Software Systems using the Bunch Tool, IEEE Transactions on Software Engineering, Vol. 32. No. 3, March 2006. pp 193–208.
[21] M´arcio de Oliveira Barros, An analysis of the effects of composite objectives in multi-objective software module clustering, In Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12), ACM, Philadelphia, USA, 7-11 July 2012. pp 1205–1212.
[22] Erik M. Fredericks and Betty H.C. Cheng, Exploring automated software composition with genetic programming, In Proceeding of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '13), ACM, Amsterdam, The Netherlands, 6-10 July 2013. pp 1733–1734.
[23] Abdel Salam Sayad and Hany Ammar, Pareto-optimal search-based software engineering (POSBSE): A literature survey, In 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2013), San Francisco, CA, USA, 2013.
[24] Raymond P. L. Buse and Westley Weimer, Learning a metric for code readability, IEEE Transaction in Software Engineering, Vol. 36, No. 4, 2010. pp 546–558.
[25] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini, Model refactoring using interactive genetic algorithm, In Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13), Vol. 8084, St. Petersburg, Russia, 24-26 August 2013. Springer. pages 96–110.
[26] Mel ´O Cinn´edie, Laurie Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam, Experimental assessment of software metrics using automated refactoring, In 6th IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012), Lund, Sweden, September 2012. pp 49–58.
[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In Proceedings of the International Conference on Software Engineering, volume, Edinburgh, Scotland, UK, May 2004. pp 563–572.
[28] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An open source tool for search based software testing of C programs. Journal of Information and Software Technology, Vol. 55, No. 1, January 2013. pp 112–125.
[29] Phil McMinn, Mark Harman, Youssef Hassoun, Kiran Lakhotia, and Joachim Wegener, Input domain reduction through irrelevant variable removal and its effect on local, global and hybrid search-based structural test data generation, IEEE Transactions on Software Engineering, Vol. 38, No. 2, March and April 2012. pp 453 – 477.
[30] T. M. Khoshgoftaar, E. B.Allen, F. D. Ross, R. Munikoti, N. Goel, and A. Nandi, Predicting fault-prone modules with case-based reasoning, ISSRE; the Eighth International Symposium on Software Engineering, Albuquerque, New Mexico, November, Los Alamitos, CA: IEEE Computer Society, 1997. pp. 27-35.
[31] T. M. Khoshgoftaar, and N. Seliya, Tree-based software quality estimation models for fault prediction, METRICS 2002, the Eighth IIIE Symposium on Software Metrics, Ottawa, Canada, June 4-7, Los Alamitos, CA: IEEE Computer Society, 2002. pp. 203-214.
[32] Claire Le Goues, Stephanie Forrest, and Westley Weimer, Current challenges in automatic software repair, Software Quality Journal, Vol. 21, No. 3, 2013. pp 421–443.
[33] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer, GenProg: A generic method for automatic software repair, IEEE Transactions on Software Engineering, Vol. 38, No. 1, 2012. . pp 54–72.
[34] Magne Jnrgensen, Experience with the accuracy of software maintenance task effort prediction models, IEEE Transactions on Software Engineering, Vol. 21, No. 8, August 1995.