Applying Machine Learning in Software Engineering Activities

Gurunath Ashok Hanamsagar
Computer Science Department
North Carolina State University
Raleigh, NC 27606, USA
ghanams@ncsu.edu

Saurabh Suresh Sakpal
Computer Science Department
North Carolina State University
Raleigh, NC 27606, USA
ssakpal@ncsu.edu

Shivani Sharma
Computer Science Department
North Carolina State University
Raleigh, NC, 27606, USA
ssharm21@ncsu.edu

ABSTRACT
This paper tracks how knowledge extraction and information retrieval techniques have been used to improve upon different areas of Software Engineering. There has been phenomenal development in Machine Learning and Data Analytics in past few years. These innovations and ideas have encouraged researchers to optimize and automate different processes in Software engineering. From bug localization, feature location, code clone detection to software fault prediction.

Keywords
Software Engineering, Machine Learning, Latent Dirichlet allocation

1. INTRODUCTION
Software Engineering has always been one of the most applied topics in the industry. It is followed by all industry players and compliance is often the yardstick to measure the quality of code. Developments in all other fields affect SE as it comprises all the components: data, compute, storage, applications. Advancements in Machine Learning has had significant impact on processes that take place. Compliance checking mechanisms have become more sophisticated, impact analysis has become much more visual and better planning is much easily possible.

Precise estimation of software development has major implications for the management of software development. If planning committee's estimate is too low, then the software development team will be under considerable pressure to finish the product prematurely, and hence the resulting software may not be fully functional or validated. Thus, the product may contain residual errors that need to be corrected during a later part of the software life cycle, in which the cost of corrective maintenance is higher. On the other hand, if a manager's estimate is too high, then too many resources will be committed to the project and hence get wasted. Furthermore, if the company is engaged in contract software development, then too high an estimate may fail to secure a contract and too low will mean breaking it, neither of which is favorable. The importance of software effort estimation has motivated considerable research in recent years.[19]

The challenge of modeling software system structures in a fastly moving scenario gives rise to a number of demanding situations. First situation is where software systems must dynamically adapt to changing conditions. The second one is where the domains involved may be poorly understood. And the last but not the least is one where there may be no knowledge (though there may be raw data available) to develop effective algorithmic solutions. Not surprisingly, machine learning methods can be (and some have already been) used in developing better tools or software products.[19] Our preliminary study identifies the software development and maintenance tasks in the following areas to be appropriate for machine learning applications: requirement engineering, traceability, feature location, bug fixing.

Machine learning deals with the issue of how to build programs that improve their performance at some task through experience. Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are particularly useful for poorly understood problem domains where little knowledge exists for the humans to develop effective algorithms and domains where there are large databases containing valuable implicit regularities to be discovered; or domains where programs must adapt to changing conditions. [19] Unsurprisingly, 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.

We have surveyed research papers where Machine Learning was used to improve Software Engineering mechanisms across the timeline of 2009-2016 and come up with our understanding of turn of events. We have summarized most of the work that has been done in the related work section. In this paper, we take a look at how machine learning (ML) algorithms can be used to build tools for software development and maintenance tasks. The summary is just to depict what field in SE the paper has focussed and what concepts have been utilised from the field of ML. In the next section, we have provided our take on what could be the plausible reasons for the focus being shifted. We have also

attempted to link the dots between how other changes in rest of the fields in Computer Science have affected the SE field by introducing newer applications workflows. The conclusion section at the end briefly describes what our learnings were with this paper. How we are able to get a better picture of the status and type of work going on in the Software Engineering field.

2. MOTIVATION

An informal survey has over the past decades found that various SE tasks are addressed through IR and NLP of software documents. [11] Common examples include traceability, link recovery, concern/concept/feature/bug location, software search, change impact analysis, requirements analysis, bug triage, refactoring, defect prediction, software redocumentation, etc. With advancements in NLP and Machine Learning resources, there was a steep increase in Text Retrieval (TR) in the field of SE. Machine learning is typically used when the data set is to be trained and used later with a different usage which is more sophisticated. Software engineering houses immense data which needs to understood smartly and made inference from in order to ensure better product delivery. Hence, we are excited to highlight the evolution of IR practices using Machine Learning techniques in the field of Software Engineering.

3. RELATED WORK

3.1 Software Requirements

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. How to obtain functional requirements of a system is the focus here. The situation in which ML 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. Under this circumstance, functional requirements can be "learned" from the data through some learning algorithm.

An initial attempt to applying requirements analysis using Machine Learning can be traced back to 2000. Tropos was a Software framework which performed Requirement Analysis in two phases. Early Requirement analysis was responsible for model selection, organizational setting. The resultant of this is description regarding various actors and their corresponding goals with relevant function and qualities. Subsequently, they also used late requirement analysis which was responsible for producing system to be, operational environment along with relevant functions and qualities. The late requirement model comprised a troubleshooting model which learned by experience aka, Machine Learning.

Requirements prioritization is an important task which helps in determining which features go into which release. This is a pivotal task as the development and testing work depends on the schedules set as per the releases. Application of machine learning techniques was explained in [12]. They used case based ranking to reduce the elicitation issue in requirements prioritization. This elicitation algorithm is a three stage process which comprises iterative ranking of requirement sets. This phase of the algorithm uses Machine learning techniques.

One common problem in Software engineering is tracing the product features and designs back to requirements to demonstrate compliance. These requirements may include functional requirement as demanded by a product or even non-functional requirements, for e.g., compliance to standards etc/ This task is pivotal and can be done on the go or later. Current methods for demonstrating compliance rely either on after-the-fact audits, which can lead to significant refactoring when regulations are not met, or else require analysts to construct and use traceability matrices to demonstrate compliance. Manual tracing can be prohibitively time-consuming; however automated trace retrieval methods are not very effective due to the vocabulary mismatches that often occur between regulatory codes and product level requirements. The 2010 paper [13] describes two machine learning approaches to counter these issues. In the first experiment, a probabilistic network model was used to generate traces from the ten targeted HIPAA regulations to requirements in each of the 10 patient healthcare systems. Trace queries were formulated directly from the text of each HIPAA security regulation. Metrics were computed individually for each of the HIPAA regulations by establishing threshold values that optimized certain measures. The second approach is based on the idea that when a training set is not available, a relevant set of indicator terms can be learned from domain specific documents mined from the Internet. This approach generally improved traces for the same set of regulations that were improved by the machine learning approach. The benefit of the web-mining approach is that it bypasses the time-consuming step of manually constructing a training set. The approach involves three steps. First a set of relevant domain specific documents are identified. Second, the documents are analyzed to extract a set of domain specific terms. Finally these terms are composed into a new query which is used to execute the trace.

In a commercial software, adherence to non functional requirements is equally pivotal as functional requirements as the system's success relies vastly on it. Given the need to analyze and implement NFRs from a wide variety of available sources, system analysts need to quickly identify and categorize NFRs. In [14], the authors have developed a tool-based approach, which they call NFR Locator, to classify and extract sentences in existing natural language texts into their appropriate NFR categories. The

classification is necessary to determine what role a sentence has. From sentences marked as non-functional, they would then extract critical information specific to each NFR category. They measured how NFRs can be identified in the documents and used for analysis. Machine learning algorithms performed better than traditional probabilistic ones for the same.

3.2 Software Traceability

Traceability is the feature of any software to be able to develop coherent relation between various modules. It represents the reason of being of any particular module. Documentation to Code traceability is a very important feature as it can also be linked to the requirements and initial design features from the same place. Traceability can also be to the testing case scenarios from the code or from the documentation. Traceability matrices are commonly used to identify these relations between modules. One naive approach would be to manually fill these matrices. This approach can get near to impossible when the scale of the software increases. Hence, Machine learning algorithms are used quite often to curb this issue. Traceability can provide important insights into system development and evolution assisting in both top-down and bottom-up program comprehension, impact analysis, and reuse of existing software, thus giving essential support in understanding the relationships existing within and across software requirements, design, and implementation.

Information Retrieval methods can be used for traceability issues. Recovering traceability links in Software Artifact Management System using IR has been described in [15]. According to them, the main drawback of existing software management systems is lack of automatic link generation and maintaining systems. They have used LSI (Latent Semantic Indexing) for the same which is a common Information Retrieval technique. The fact that keywords used in code and documentation are similar is exploited by the algorithms. NLP is thus a strong tool to go ahead with for such tasks and is used extensively, particularly in this domain of Software Engineering.

3.3 Software Maintainability

Any changes made to code after the release of a product is termed as software maintenance. It is the responsibility of the service provider to keep providing software support to the client. Thus, it is of utmost importance that a software product can be modified as easily as possible. In other words, it should be structured during the design phase keeping in mind the future possible changes. Thus, software maintainability becomes an important aspect of a product. This needs to be measured and quantified too and there has been work in applying Machine Learning concepts to this task.

Prediction software maintainability using ML algorithms was attempted in the paper [16]. Tracking the maintenance behaviour of the software product is very complex. This is precisely the reason that predicting the cost and risk associated with maintenance after delivery is extremely difficult which is widely acknowledged by the researchers and practitioners. In an attempt to address this issue quantitatively, the main purpose of the paper was to propose use of few machine learning algorithms with an objective to predict software maintainability and evaluate them. The proposed models are Group Method of Data Handling (GMDH), Genetic Algorithms (GA) and Probabilistic Neural Network (PNN) with Gaussian activation function. The prediction model was constructed using the above said machine learning techniques. In order to study and evaluate its performance, two commercial datasets UIMS (User Interface Management System) and QUES (Quality Evaluation System) were used. The code for these two systems was written in Classical Ada. Three different machine learning algorithms were used for the purpose of prediction of software maintainability. Even though many studies reported wide application of GMDH model and GA model in diverse fields for the purpose of prediction of high order input output relationship which is complex, non linear and unstructured but for the first time they were used for prediction of software maintainability.

Another such attempt restricted to Object Oriented software system was described in [17]. They developed an extreme learning machine (ELM) maintainability prediction model for object oriented software systems. The model is based on extreme learning machine algorithm for single-hidden layer feed-forward neural networks (SLFNs) which randomly chooses hidden nodes and analytically determines the output weights of SLFNs. Their maintainability prediction model based on type-2 fuzzy logic systems was developed for an object-oriented software system. The model is constructed using popular object-oriented metric datasets, collected from different object-oriented systems. Prediction accuracy of the model is evaluated and compared with commonly used regression-based models and also with Bayesian network based model which was earlier developed using the same datasets.

3.4 Feature Location

Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system [1]. As a software evolves these functionalities are added, removed and improved upon. Identifying an initial location in the source code that corresponds to a specific functionality is known as feature (or concept) location [2], [3]. Software developers often need to understand and modify an unfamiliar codebase. However, a manual search of a large amount of source code, even with the help of tools such as pattern matchers or an integrated development environments like Eclipse, Visual Studio etc is time consuming.

Dit et al. did a comprehensive study on the feature location techniques that have evolved over a decade [1]. This work has

helped us a lot to construct this case study for feature location. Techniques for feature location can be broadly divided into textual and dynamic search.

3.4.1. Textual Search

In textual search various Information retrieval (IR) techniques leverage the fact that identifiers and comments embed domain knowledge to locate source code that is textually similar to a query describing a feature. Marcus et al. in 2004 used Latent Semantic Embedding (LSI) to map concepts specified in natural language by the programmer to the relevant part of the source code. The corpus is preprocessed by splitting compound identifiers based on common naming conventions. The corpus is partitioned into documents representing all terms associated with a program element. Documents can be of different granularities, such as classes or methods. The corpus is then transformed into an LSI subspace through Singular Value Decomposition (SVD). After SVD, each document in the corpus has a corresponding vector. To search for code relevant to a feature, a programmer formulates a query consisting of terms which describe the feature. The query is also transformed into a vector, and a similarity measure between the query vector and all the document vectors is used to rank documents by their relevance to the query. The new approach was compared against grep and ASDGs, and several advantages were found. LSI is as easy to use as grep, yet it produces better results. Also, LSI was able to identify some relevant program elements missed by Abstract System Dependence Graphs (ASDGs) [9]

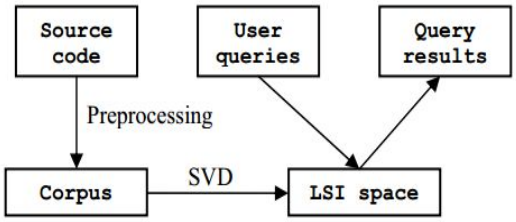


Figure 1: The concept location process using LSI

As seen in the work of Cleary and Exton [4][5], IR was still a common technique. Their approach, called cognitive assignment, considers indirect correspondences between query and document terms so that relevant source code can be retrieved even if it does not contain any of the query terms. Queries are expanded by analyzing term relationships from both source code and non-source code artifacts. A case study was conducted on Eclipse in which cognitive assignment was compared to other IR techniques, such as language modeling, dependency language model, vector space model, and LSI. The results show that cognitive assignment matches the performance of the other IR techniques and in some cases it outperforms them.

In 2009 we see focus being shifted to Natural Language Processing with the work of Hill et al [6]. Their analysis centers on three types of phrases: noun phrases, verb phrases, and prepositional phrases. The phrases are extracted from method and field names and additional phrases are generated by also looking at a method's parameters. Once the phrases are extracted, they are grouped into a hierarchy based on partial phrase matching. The phrases are linked to the source code from which they were extracted.

In 2010 Abebe et al. introduced an approach that extract concepts from source code by applying NLP techniques [7]. Their approach applies natural language parsing to sentences constructed from the terms that appear in program element identifiers. The result of parsing is represented as a dependency tree. An ontology is then extracted by mapping linguistic entities (nodes and relations between nodes in the dependency tree) to concepts and relations among concepts. In the same year Wursch et al leveraged static analysis and semantic web-based technologies to provide users with a natural language guided query interface to answering program comprehension questions. The Web Ontology Language (OWL) is used to model an ontology for source code. This allows developers to query source code with some natural language guided vocabulary within their IDE [8].

3.4.2 Dynamic Search

Dynamic feature location relies on collecting information from a system during runtime. These techniques collect and analyze execution traces to identify a feature source code based on set operations. In 2006 Anotonio et al. proposed a statistical analyses of static and dynamic data to accurately identify features in large multithreaded object-oriented programs [3]. First, they collect dynamic data from different scenarios as traces, lists of events composed of variable accesses, object instantiations, function and method calls. Then an epidemiological metaphor is used to classify the events in traces as relevant or not to a feature of interest. The relevant events are also ranked to associate top-ranked events with a feature. Finally, they use feature-relevant events to build microarchitectures highlighting the source code constructs.

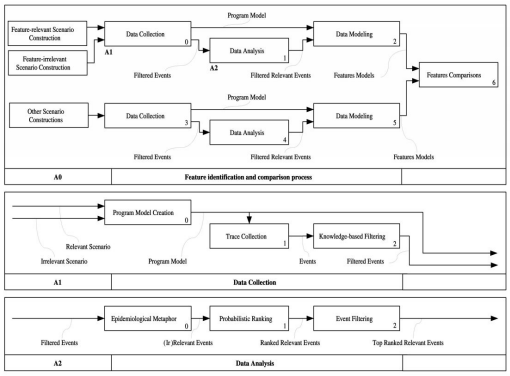


Fig: Feature identification and comparison process (detailed data collection and analysis).

3.4.3. *A Fusion Approach*

In 2012 Dit et al. attempted a data fusion approach to feature location [11]. Data fusion is the process of integrating multiple sources of information such that their combination yields better results than if the data sources are used individually. Their approach defines new feature location techniques based on combining information from textual, dynamic, and web mining or link analysis algorithms applied to software. A novel contribution of this work is the use of advanced web mining algorithms to analyze execution information during feature location. This was the first attempt to combine both static and dynamic approaches. The following figure describes this fusion in a nutshell.

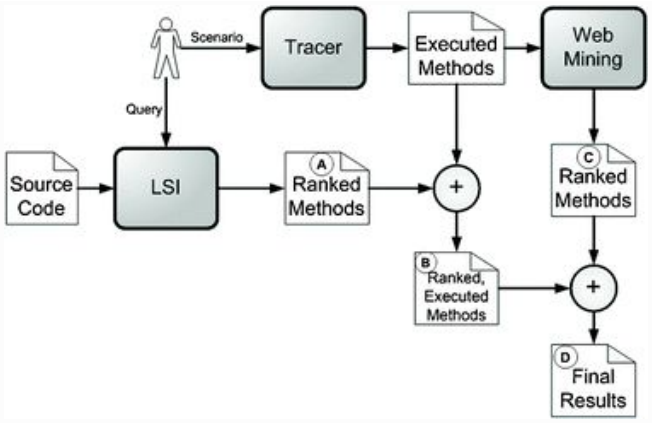


Fig: The ranked results of type A are: *IRLSI*; The ranked results of type B are: *IRLSIDynbin*; The ranked results of type C are: *WMHITS*(*h*,*bin*), *WMPR*(*bin*), etc.; The ranked results of type D are: *IRLSIDynbinWMHITS*(*h*,*bin*)*top*, *IRLSIDynbinWMPR*(*bin*)*top*, etc.

3.5 Bug Localization

Bug localization is a widely studied problem in program analysis. Debugging is a very painful and time consuming process for each programmer. With the growing of software size nowadays, it become harder and harder to find where is the bug located in the programs. Many approaches have been proposed to deal with computer aided debugging and testing, and some researches integrate machine technique to identify bug lines.

3.5.1 *Neural Networks for Bug Localization*

A paper was proposed by Wong et al [28]. The proposed approach used testing coverage information. Training of the backpropagation neural network used testing results and the statements covered by the test cases. After the training, the networks showed possible relationship between a statement in a program and a failure. Based on this result, suspicious statements were ranked according to its risk of being faulty. This information and the slices execution of failed tests are combined to improve the efficiency of the method. This approach proposed does not depend on a single model. However, the approach was validated by considering the testing criteria all-statements only in the context of procedural code and nowadays the Object Oriented (OO) paradigm is the most popular and used.

In Object Oriented Programming paradigm, the class methods are smaller, and the challenge is on the class testing that includes the integration testing of all methods in a class. A failure generally occurs when the methods are integrated and could be caused by a combination of several faults. Because of these different aspects and due to the importance of OO applications, another work by Ascari [27] tried to explore the use of Machine Learning (ML) techniques in the OO testing fault localization context. Using only coverage data, they successfully adapted the approach of Wong et al [28], based on NN, and separately implemented SVM based approach for locating faults in OO applications. The NN approach were found very hard to train because of the abundance of local minima and the high dimensionality of the weight space. On the other hand, the SVMs used a more efficient training algorithm and could represent complex, nonlinear functions too.

The chosen NN was a Multi-layer neural network with the backpropagation training algorithm. The training datasets were submitted to the Multi-layer neural algorithm. Sequential minimal optimization algorithm was used for training a support vector classifier. The following steps were followed for the NN based algorithm:

1. Instrumentation of each application (and correspondent versions) to produce the trace of each execution.
2. Implementation of a script to automatically capture and compare the outputs of the application and correspondent faulty versions
3. Automatic execution of the applications and faulty versions: the results were stored in files and a matrix was obtained. In such matrix the columns represent the covered methods.
4. Application of the Machine Learning algorithms by using the obtained matrices.
5. Creation of the virtual test sets: where each test cases covers only one method.
6. Estimation of the most suspicious methods by using the matrix of virtual test cases as input to the trained networks and results of Step 4.

3.5.2 *Bug Localization using LDA*

LDA stands for Latent Dirichlet Allocation. It was presented as graphical model for topic discovery by David Blie, Andre Ng, and Michael Jordan in 2003 paper[20]. The paper defines Latent

Dirichlet allocation (LDA) as a generative probabilistic model for collections of discrete data such as text corpora. LDA is a three-level hierarchical Bayesian model, in which each item of a collection is modeled as a finite mixture over an underlying set of topics. Each topic is, in turn, modeled as an infinite mixture over an underlying set of topic probabilities.

LDA represents documents as list of words. Each word belongs to a topic and each document belongs to one or more topics.

3.5.2.1 LDA Discovery

Fixed number of K topics are chosen to be discovered. Each document is broken down into words and each word is randomly assigned to one of the K topics. Topic representations of all the documents and word distribution of all the topics can be found out after random assignment. For each Word "w" in Document "d", following two things are computed:

1. *p(topic t | document d)* = the probability of words in document d that are currently assigned to topic t.
2. *p(word w | topic t)* = the probability that any document was assigned topic t over all documents because it contained word w.

Then each word "w" is reassigned to a new topic, such that *p(topic t | document d) * p(word w | topic t)* is highest. Once all the words are reassigned to new topic, topic representations of all the documents and word distribution of all the topics is updated. If the above steps are repeated large number of times, steady state is reached where all the topic assignments for each word are pretty good.

LDA can be explained with a simple example. If a document is assumed to belong to a particular topic then it is expected that lot of word from that particular topic will be seen in the document. If a word that belongs to this document has different topic from that of the document then we need to reassign topic for this word.

3.5.2.2 Naturalness of the Software

Originally, Code was treated like natural languages like English. Code was assumed to be as predictable and as repeatable as natural languages. Language models developed and tuned for natural language were applied to code artifacts. This assumption was proved wrong in the paper[22] published by Abram Hindle, Mark Gabel and Premkumar Devanbu in year 2012.

This paper instantiated a simple, widely used statistical language model, using modern estimation techniques over large software corpora. Standard cross-entropy and perplexity measures were used to capture high-level statistical regularity in software at the n-gram level.

Cross-entropy is a measure of how surprising a test document is to distribution model estimated from a corpus. A good model will predict with high confidence the contents of a new document drawn the training population. Thus a good model will be less perplexed when a new document is encountered. In the paper [22], n-gram model, usually used for natural language, was used on software corpus. Average cross entropy was calculated for n-gram model using 10 fold cross validation to figure out repetitiveness in software code. Based on these empirical studies, the paper [22] concluded that "corpus-based statistical language models capture a high level of local regularity in software, even more so than in English".

3.5.2.3 LDA Tuning for Software Artifact

Text in software artifacts has different properties, as compared to natural language text, thus, new solutions was needed for calibrating and configuring LDA and LSI to achieve better (acceptable) performance on software engineering tasks.

Applying LDA requires setting the number of topics and other parameters specific to the particular LDA implementation. For example, the fast collapsed Gibbs sampling generative model for LDA requires setting the number of iterations n and the Dirichlet distribution parameters α and β. Thus, in order to get accurate results by using LDA, correct value for n, α and β should be selected.

Finding an LDA configuration that provides the best performance is not a trivial task. Without a proper calibration, LDA's performance is suboptimal. Finding the best configuration of these parameters poses two problems. Firstly, A measure is required that can be used to assess the performances of LDA before applying it to a specific task. This measure should be independent from the supported SE task. Secondly, an efficient way is needed to find the best configuration of parameters without performing an exhaustive analysis for all possible combinations. Since exhaustive analysis is impractical due to the combinatorial nature of the problem.

Several heuristic based approaches were proposed for configuring the parameters of LDA. One such method was proposed by by Grant and Cordy for the feature location which used source locality heuristic [24]. Paper inferred appropriate number of LDA topics needed to optimize the topic distributions over a set of source code methods. Simple heuristic metric based on location of method in the source code and package structure that gives a quick and reasonable estimate about whether or not a pair of methods are conceptually related to one another was used in this paper. Grant and Cordy treated each document topic distribution as a vector, and used the cosine distance between two vectors as a measure of their conceptual similarity. Thus in this vector space representation, each document in a model with k topics is

represented by a vector of k real numbers ranging from 0 to 1. The first first heuristic was "A method is likely to be conceptually related to its nearest neighbours in the vector space representation of the source code methods."

Second heuristic the paper suggested was "Two methods are likely to be conceptually related if they are found in the same file or folder." This metric focused on co-location in the source code structure. Related methods are placed in the same class and in most languages, programmers follow one class per file concept. Also, related files are placed in same package.

Another method for selecting optimal number of LDA topics was proposed by Griffiths and Steyvers [25]. They proposed a method for choosing the best number of topics for LDA among a set of predefined topics. Their approach consists of (i) choosing a set of topics, (ii) computing a posterior distribution over the assignments of words to topics P(z|w, T), (iii) computing the harmonic mean of a set of values from the posterior distribution to estimate the likelihood of a word belonging to a topic (i.e., P(w|T)), and (iv) choosing the topic with the maximum likelihood. In their approach, the hyper-parameters α and β are fixed, and only the number of topics is varied, which in practice, is not enough to properly calibrate LDA. In our approach, we vary all the parameters (i.e., k, n, α and β), to find a (near) optimal configuration for LDA.

3.5.2.4 LDA Tuning using GA

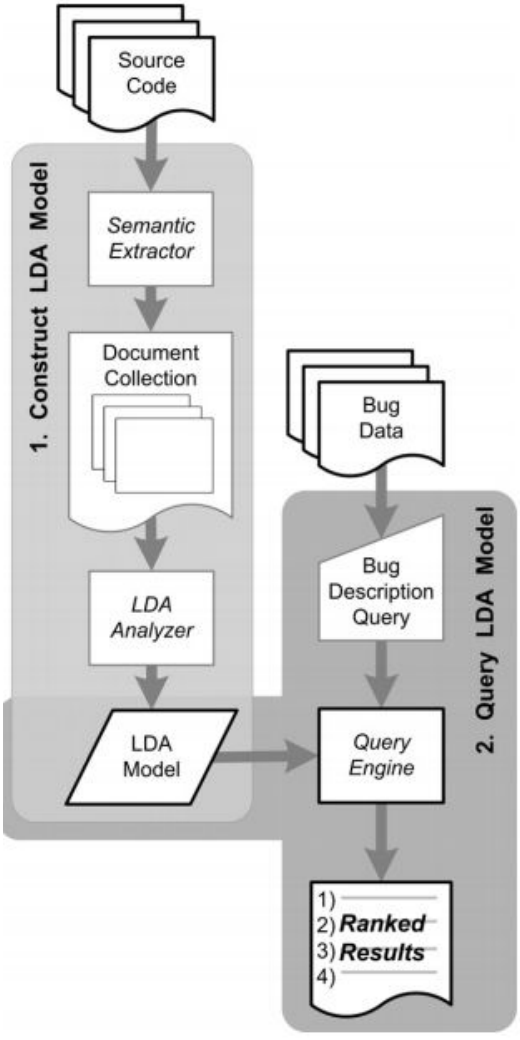
Heuristic based approaches described above focus only on identifying the number of topics that would result in the best performance of a task, while ignoring all the other parameters that are required to apply LDA in practice. Moreover, such approaches have not been evaluated on real SE applications or have been defined for natural language documents only, thus, they may not be applicable for software corpora.

In 2013 a paper[4] was published that introduced a novel solution called LDA-GA, which used Genetic Algorithms (GA) to determine a near-optimal configuration for LDA. LDA-GA was implemented on three different SE tasks: (1) traceability link recovery, (2) feature location, and (3) software artifact labeling. The results of the empirical studies demonstrated that LDA-GA is able to identify robust LDA configurations, which lead to a higher accuracy on all the datasets for three SE tasks as compared to previously published results, heuristics, and the results of a combinatorial search.

LDA configuration can be represented by P = [k, n, α, β] and we plan to maximizes the overall quality of the clustering produced by LDA. For solving such an optimization problem the paper applied Genetic Algorithms (GA) which is a stochastic search technique based on the mechanism of a natural selection and natural genetics. The problem that the paper wanted to solve had 4 decisions and 1 objective to maximise. The GA search starts with a random population of solutions, where each individual from the population represents a solution of the optimization problem. The population evolves through subsequent generations and, during each generation, the individuals are evaluated based on the fitness function that has to be optimized. Fitness was calculated using the mean Silhouette coefficient. For creating the next generation, new individuals are generated by (i) applying a selection operator, which is based on the fitness function, for the individuals to be reproduced, (ii) recombining, with a given probability, two individuals from the current generation using the crossover operator, and (iii) modifying, with a given probability, individuals using the mutation operator.

3.5.2.5 LDA for Bug Localization

Two steps are necessary to construct an LDA model of a software system: (1) build a document collection from the source code and (2) perform an LDA analysis on the document collection. These steps are detailed below and illustrated in following diagram:



Step 1: Build a document collection from the source code

Extract semantic information, such as comments and identifiers, from each source code element at the desired level of granularity (e.g., package, class, method). Preprocess the semantic information before writing it to the document collection. Store the preprocessed data extracted from each source element as a separate document in the collection. Generally, all documents are created using the method level of granularity, that is, each document represents one method from the source code. Also it is important to extract string-literals in addition to comments and identifiers as they are found in error messages and likely to be included in a bug description.

Step2: Generate an LDA model

LDA model generation is basically selecting a configuration for the model and run it over the document collection. Any method described above can be used to select the configuration and once selected the model can trained on the collection of document generated in step 1.

Step 3: Querying the Model

After step 2, the LDA model previously generated is ready for the query. Terms in the query are preprocessed in the same manner as the source code. Each query results in a list of source code elements ranked by similarity to the query. Query formulation involves using of Bug title and Bug description entered into the software's bug repository by the tester.

In 2011 Lukins, Kraft and Etzkorn published a paper on using LDA for Bug Localization [26]. They manually formulated the queries and made the queries more formal and specific by utilizing the Bug information. Following process was used to create queries:

1. Form an initial query by manually extracting keywords from the bug title. Words not related to the bug domain were ignored.
2. Form a second query by manually adding keywords from the summary of the initial bug report to the first query.
3. Form a third query by adding words related to the bug and/or removing words from query 1 or 2 that were less applicable to the bug domain (e.g., error). Words added were limited to the following:
    a. Common abbreviations or whole words when an abbreviation was used.
    b. Variants of words already in the query.
    c. Synonyms of words in the query.
    d. Sub-words of words in the query.

3.6 Verification & Validation

Verification and validation are important checking processes to make sure that implemented software systems conforms to its specification. To check a software implementation against its specification, we assume the availability of both a specification and an executable code. This process is pivotal to the process of software development and is often given equal importance as development. It incurs immense costs as the software product is worthless without the completion and authorization by the testing phase. Testing is closely related with almost all the phases of software development right from requirements engineering up to the release phase. Machine learning techniques have been quite useful in improving verification mechanisms.

Software testing costs estimation is a task quite commonly taken up before the start of the development process. Since resources are limited, it is critical to determine how they should be allocated throughout the software life-cycle. Tools are being developed to estimate software development costs. The purpose of this research was to demonstrate a technique using machine-learning to identify attributes that are important in predicting software testing costs and more specifically software testing time in a particular company. Understanding the factors that affect the cost of software testing allows management to plan for testing. Testing data on 25 software projects were collected. Using this database, a machine learning system identifies the factors that affect testing time by creating a classification tree in which nodes share similar values. By analyzing the classification tree they were able to determine the salient factors which placed a program into its group. The factors that they considered includes code complexity measures, measures of programmer and tester experience, measures of the use of software engineering principles such as structured programming techniques, and statistics collected during actual testing of the projects. The programs in a node had an average testing time that is reflective of any program whose attributes would place it into this group. Thus, the tree could be used, among other things, to predict testing time.

Integration testing is another domain where machine learning has found application. The authors considered the design of a complex software system, (e.g., telecom service), loosely couples in distributed architectures. The design of software systems, typically new services offered on the Internet, is more and more based on the integration of components from third party sources (COTS), loosely coupled in a distributed architecture.

The system integrator is in charge of providing a new service based on the integration of such components, with minimal development of interfacing software (often nicknamed as “glue”). The integrator faces the problem of providing a required system assembling COTS of which he / she has a limited knowledge. Since components come from third party, their internal structure is usually unknown and the documentation is not sufficient to work out all the details of their interactions with other components.[18] Therefore, the integrator would typically first test each component to learn its behaviour on typical requests it would have to serve in

the assembly, and then test the integrated system based on variations of system use cases. Due to the lack of formal models of components, in order to design test cases for the components and the integrated system, the integrator has to rely on his intuitions. Currently, there are not many tools that can help him in test generation. At the same time, use cases are often provided by domain experts (including system users). The use cases might be informal and obtained in an ad hoc way, and thus are hardly sufficient to check that the components will interact correctly in any combination of requests. This incremental approach is similar to the one presented in [18]. In this paper, they concentrate on the use of such an approach for component integration, and develop the necessary extensions to the learning algorithm.

In their approach, the test designer just has to concentrate on the abstraction of actual input and output events to consider in the models of components. It is a common observation that software designers can readily identify interfaces and data values, and provide some formal conceptualization or abstraction of them, whereas the real difficulty lies in producing a formal model for the behaviour. Their approach leaves the easiest part (event abstraction from actual PDUs or component interactions) to the test designer, and the model is built automatically based on this provided abstraction.

4. OUR COMMENTARY

As with any other fields in Computer Science, the developments in the field of Software Engineering were closely related to developments in other fields. As more complex algorithms became possible to be executed in lesser times, more complicated models were begun to be applied on common problems. Software Engineering deals with extensive data in terms of code, documentation, designs, patterns etc and hence there is no wonder that Data Analytics and Machine Learning has changed the dynamics of Software Engineering methods. A very common application in Software Engineering is the usage of Natural Language Processing. This assists on multiple levels right from the requirements phase till the verification and validation steps. Also, Machine learning has been extensively used in linking various modules of software engineering. We came across quite a lot of work where Machine Learning was used for traceability, fault detection, bug fixing, impact analysis etc., all of which deals majorly with linking code and documentation.

Requirements engineering has been present since the beginning when Software Engineering was introduced. Initial work comprising Machine Learning concepts was based more on the lines of how to organize the requirements and prioritize so as to set feasible release deadlines. It did not involve significant interaction with code and could be accredited to helping for the vision of the products and laying out release dates and deadlines. Focus then shifted to the task of linking the requirements to code and to the testing phases. This was quite significant since the machine learning concepts were being on a mixture of code and data. We feel that the scale of data analytics assisted in such developments as it was now possible to deal with humongous sized code base and documents. As we see in today's SE methodologies, each task in a development phase is closely knit to it's testing counterpart and also maps onto the requirement design. This is of utmost importance if we are to produce high quality industry standard software products. At later stages even non functional requirements started were also started to be mapped to the code and subsequently to the testing phase. measuring non functional requirements can be vague as compared to the Functional ones and therefore Hence, NLP and ML techniques were applied on those lines to meet the non functional demands.

Software maintainability came to light pretty early when the concept of support was highlighted after product release. Predicting how maintainable a software product has been is always challenging. There was great work on automated prediction of Software Maintainability during the course of development. Three algorithms were implemented and their results analysed which gave positive results. Another such paper used ML algorithms for Software maintainability in Object Oriented products. This was a significant breakthrough because object-oriented products were always considered to be hard to maintain and change as per future requirements.

In feature location, in the start there were attempts to apply IR techniques like LSI on natural language instances like code comments and documents. Later researchers shifted from IR techniques to Natural Language Processing (NLP) as seen in Hill et al. [9]. This might be contributed to the boom in machine learning and NLP with increasing computational power during this period. In 2010 Abebe et al did a bold move to use NLP on source code. This was a big shift since the programming language differs a lot from the natural language and the NLP techniques in this case are not straightforward to use. There were also attempts to make use of dynamic stack traces for feature location but such attempts diminished with the coming years. A fusion approach but Dit et al [11] introduced data fusion approach for the first time for feature location problem. This novel solution comes at the time when ML was booming with data fusion and multi layer graphs. With this attempt in 2012, dynamic analysis did a comeback but still not much of it was seen till date.

ML techniques have bettered the Software Testing mechanisms. Initially, the research was focussed more towards using ML to link the testing related documents with the requirements or the code that actually is being tested. That part of research falls mainly under traceability. It also involved how testing scenarios can be combined or divided so that we get rid of redundancy and at the same time we do not lose out on valuable scenarios. We

have described an interesting research where integration testing was made easy using ML concepts. Integration testing involved tedious work of understanding each module, and first validate the testing of each module. The paper provided a mechanism of decoupling the task of testing with the knowledge of the module using formal methods of development and test mechanisms. Such a profound method has made it quite easy for the actors involved in Testing.

5. CONCLUSION

Our primary learnings with this survey were the main advancements in the field of Software Engineering. How the definition, application and involvement of each of subcategories of SE has changed over the course of time. We possess a better understanding of the quality of a software product in today's terms. Since our focus of interest was Machine learning, we have learnt immensely so as to how modern ML techniques are clubbed with SE tasks. We feel it is an overlap of two vastly independent fields that can be overlapped to provide a mechanism for best practices in the field of software development. Since, the processes in SE were dominantly manually driven, automation in this field has been a welcome change. ML has enabled extremely large data sets to be considered for the betterment of the software engineering practices. The effects of this overlap can be seen across fields even outside the Computer Science. Accurate software engineering practices and predictions have led to more feasible business decisions, economic plans and legal implications. The resources are better organized and more transparency and accountability has creeped into the work of development.

6. ACKNOWLEDGMENTS

Our sincere thanks to Prof. Tim Menzies, Wikipedia, IEEE Xplore, Association of Computing Machinery.

7. REFERENCES

[1] Feature Location in Source Code: A Taxonomy and Survey Bogdan Dit, Meghan Revelle, Malcom Gethers, Denys Poshyvanyk The College of William and Mary

[2] The Concept Assignment Problem in Program Understanding Ted J. Biggerstaff Microsoft Research and Bharat G. Mitbander and Dallas Webster Microelectronics and Computer Technology Corporation (MCC)

[3] Feature Identification: An Epidemiological Metaphor Giuliano Antoniol and Yann-Gae Gue´he´neuc

[4] Assisting Concept Location in Software Comprehension Brendan Cleary and Chris Exton Lero, University of Limerick, Ireland, brendan.cleary@ul.ie

[5] An empirical analysis of information retrieval based concept location techniques in software comprehension. Brendan Cleary Chris Exton Jim Buckley Michael English

[6] Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse ∗ Emily Hill, Lori Pollock and K. Vijay-Shanker Department of Computer and Information Sciences University of Delaware Newark, DE 19716 USA {hill, pollock, vijay}@cis.udel.edu

[7] Natural Language Parsing of Program Element Names for Concept Extraction Surafel Lemma Abebe and Paolo Tonella Software Engineering research unit Fondazione Bruno Kessler, Trento, Italy {surafel,tonella}@fbk.eu

[8] Supporting Developers with Natural Language Queries Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall {wuersch,ghezzi,reif,gall}@ifi.uzh.ch s.e.a.l. – software architecture and evolution lab Department of Informatics University of Zurich, Switzerland

[9] Case Study of Feature Location Using Dependence Graph* Kunrong Chen, Václav Rajlich Department of Computer Science Wayne State University Detroit, MI 48202 USA rajlich@cs.wayne.edu

[10] Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Bogdan Dit Meghan Revelle Denys Poshyvanyk

[11] W. Zhao and L. Zhang. Sniafl: Towards a static non-interactive approach to feature location. ACM Transactions on Software Engineering and Methodology, 15(2), 2006.

[12] Facing scalability issues in requirements prioritization with machine learning techniques

[13] A Machine Learning Approach for Tracing Regulatory Codes to Product Specific Requirements

[14] Automated Extraction of Non-functional Requirements in Available Documentation

[15] Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods

[16] Software Maintainability Prediction using Machine Learning Algorithms

[17] Extreme Learning Machine as Maintainability Prediction model for Object-Oriented Software Systems

[18] Integration Testing of Components Guided by Incremental State Machine Learning Keqin Li CNRS LSR-IMAG Keqin.Li@imag.fr Roland Groz INPG LSR-IMAG Roland.Groz@imag.fr Muzammil Shahbaz France Télécom Muhammad.MuzammilShahbaz @orange-ft.com

[19] Applying Machine Learning Algorithms In Software Developement Du Zhang Department of Computer Science California State University Sacramento, CA 95819-6021 zhangd@ecs.csus.edu

[20] Latent Dirichlet Allocation. David M. Blei, Andrew Y. Ng,Michael I. Jordan.

[21] Introduction to Latent Dirichlet Allocation by Edwin Chen http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation/

[22] On the naturalness of software. Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, Premkumar Devanbu

[23] How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, Andrea De Lucia

[24] Estimating the Optimal Number of Latent Concepts in Source Code Analysis. Scott Grant, James R. Cordy

[25] Finding scientific topics. Thomas L. Griffiths and Mark Steyvers.

[26] Bug localization using latent Dirichlet allocation. Stacy K. Lukinsa, Nicholas A. Kraftb, Letha H. Etzkorna.

[27] Exploring machine learning techniques for fault localization. Luciano C. Ascari, Lucilia Y. Araki, Aurora R.T. Pozo, Silvia R. Vergilio.

[28] BP Neural Networ-Based Effective Fault Localization. W. Eric Wong