A review of the risk of the ecological fallacy in empirical software engineering

Bennett Narron
North Carolina State University
Raleigh, North Carolina
bynarron@ncsu.edu

Akond Rahman
North Carolina State University
Raleigh, North Carolina
aarahman@ncsu.edu

Manish Singh
North Carolina State University
Raleigh, North Carolina
mrsingh@ncsu.edu

ABSTRACT

A software system is a hierarchical structure composed of several layers, including modules, packages, and files. When designing an experiment, it is important to identify the appropriate layer or layers for gathering data in order to ensure validity in hypothesis testing. Otherwise, the study is at risk of the ecological fallacy–a logical fallacy that follows from ecological inference. These ideas were introduced into the field of Empirical Software Engineering (ESE) through a publication by Posnett *et al.* , titled*Ecological inference in empirical software engineering*, and remain relevant today. The following review provides a high-level overview ecological inference and the ecological fallacy, explores research methods in ESE before and after the publication by Posnett *et al.* , and discusses how the field has changed as a result of the publication.

Keywords

Ecological Inference, Ecological Fallacy, Aggregation, Empirical Software Engineering

1. INTRODUCTION

Posnett *et al.* (2011) challenged previous works regarding the empirical study of software systems by positing that methodologies for collecting and examining data in software projects did not take into account the risk of the *ecological fallacy*–a logical fallacy that arises from inferring conclusions across different levels of aggregation [16]. The hierarchical structure of a software system presents many levels of aggregation for examination. A project may be decomposed into modules, packages, or files, each of which may be empirically evaluated for quality, distribution, collaboration, and productivity. As the study of a population may provide some insight about its individuals, studying the various levels of aggregation of a software system is may be useful for understanding, for instance, where software defects exists, when and why they surface, and how best to minimize them. Using one level of aggregation (e.g., a population) to infer information about some dis-aggregated entity (e.g., an individual) is called *ecological inference.* Ecological inference implies the risk of the ecological fallacy–an idea that had been overlooked prior to the publication of *Understanding ecological inference in empirical software engineering.*

Choosing the correct level of aggregation (e.g., file-level) is critical when testing hypotheses on one or more of these variables [16]. But where should one start? Top-to-bottom? At file-level? The authors set forth the understand the intricacies of ecological inference and how well hypotheses hold up across differing levels of aggregation.

This paper is intended to provide an overview of the effect of the risk of the ecological fallacy on empirical software engineering, before and after the publication by Posnett *et al.* . We will begin by providing a brief background on ecological inference and the risk of ecological fallacy, and followed by a high-level overview of the study of empirical software engineering. We will then begin discussing a selection of papers published prior to the acknowledgement of the risk of the ecological fallacy in empirical software engineering. We will follow the brief survey by further discussion of the paper by Posnett *et al.* , and then we will continue with publications that followed and how they attempt address the elephant in the room. We will conclude with some discussion about the effectiveness of this newfound awareness and how effectively it is being addressed.

2. BACKGROUND AND CONCEPTS

Before we begin with the survey, it will be beneficial to have a proper introduction to the origins of ecological inference and the risk of ecological fallacy and a greater understanding of the aims of empirical software engineering. An overview of these topics will provide a modest foundation for understanding the quandary that the risk of the ecological fallacy imposes on empirical software engineering.

2.1 Ecological Inference and the Risk of Ecological Fallacy

In criticism of medical statistical methods, *The Economist*, a publication famous for its editorials on everything zeitgeist, printed an article titled, *Signs of the Times*, that derived statistically significant correlations between emergency room patients, their ailments, and their astrological signs [19]. They reported that those born under the zodiac sign of Leo are 15% more likely to be admitted to the hospital for "gastric bleeding" than those born under any other sign, while Sagittarians are 38% more likely to be laid up with a broken arm. Both of the aforementioned statistics meet the typical 95% certainty threshold. So what in Hades is going on here?

Piantadosi *et al.* , in a publication in the *American Journal of Epidemiology* in 1988, would argue that ecological analyses suffer from the same confounding biases as individual-level analyses, though perhaps more severely [15]. Ecological inference introduces a vast amount of moving parts, as the factors that affect the targeted study group are compounded with those of the aggregation of that group, leading to an abundance of confounding factors and threats to validity to consider. Piantadosi *et al.* offer no solution beyond an urge to researchers to analyze data sensibly and perform *ad hoc* measures to minimize confounding bias. Thus, as statistical analyses of the most granular subject comes with its own set of potential biases, ecological inference comes with the risk of the ecological fallacy. So, Sagittarians need not worry about their brittle arms or Leos about their bloody guts, as "statistically-significant" correlations are easy to spot when due diligence has not been performed to eliminate as much bias as possible.

2.2 Empirical Software Engineering

Empirical Software Engineering (ESE) aims for observable outcomes in the study of software systems, including quality and productivity [16]. Ideally, these studies are conducted using large sample sizes (e.g. numerous software projects) in order to leverage statistical methods for hypothesis testing and to gather large quantities of data for mining methods and machine learning for building software tools to support programming tasks [16]. Inevitably, the need to decompose data, such as software projects, into analyzable data creates the opportunity for ecological inference to occur and, thus, an increased risk in the ecological fallacy.

3. SURVEY OVERVIEW

Note that the organization of the following sections occurs chronologically, beginning with papers published prior to 2011, follow by a summary of the paper by Posnett *et al.* on ecological inference, and concluding with papers published in 2012 or later. Each subsection may be viewed as a singleton summary of a publication. The reader may read them all serially or at whim (like watching episodes of *Seinfeld*), as more in-depth discussion will occur after all papers have been presented.

4. SURVEY - APPROACHING 2011

The following papers were composed before Posnett *et al.* published on the prevalence of the ecological inference in Empirical Software Engineering. For each paper, we will provide a bullet point format, defining keywords described by each author, summarizing select ideas presented in each paper, and finally, describing how the paper used ecological inference to test hypotheses or perform analysis.

4.1 Mining metrics to predict component failures [12]

4.1.1 Keywords:

- *Software metrics*: Measurement tools derived from software repositories to perform quantitative analysis.
- *Fault prediction*: A methodology to predict failures by mining and analyzing software repositories.

4.1.2 Key Points:

- *Related Work*:
  The authors of have paper cited multiple papers that are related to defects and failures, complexity metrics, and historical data. For example, that authors cited the work of Hudepohl et al. [2] that predicted software failures from software design metrics, and reuse information from software repositories.
- *Study Instruments*:
  - Five project repositories namely, Internet Explorer 6, IIS W3 Server Core, Process Messaging Component, DirectX, and NetMeeting
  - File-based metrics: module metrics, per-function metrics, per-class metrics
  - logistic regression
  - principal component analysis (PCA)
  - correlation measures: Pearson, Spearman

4.1.3 Use of Ecological Inference

Nagappan *et al.* took a holistic approach where analysis was performed on one level (did not differentiate between aggregated or dis-aggregated level).

4.2 Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings [9]

4.2.1 Keywords:

- *Classification*: This term refers to the categorization of data values into discrete classes. For example, a binary classification will have two categories or classes - "YES" and "NO".
- *Software Defect Prediction*: This term refers to the process of identifying error prone software modules by means of data mining techniques. This helps in efficient allocation of resources to high-risk software segments.
- *Data Mining*: It is a process of analyzing data from various sources and analyzing and deriving useful insights and patterns from the data , that can be used to make better decision in future.
- *Statistical Methods*: Statistics deals with the collection, interpretation, presentation and analysis of data. It offers various methods to estimate and correct for any bias within a sample and data collection procedures.

4.2.2 Key Points:

- *Related Work*:
  The paper discusses about the previous works which have used various models for software defect prediction. Classification has been a popular approach and various types of classifiers have been applied to predict defects, including statistical procedures, tree-based methods, neural networks and analogy-based approaches.
- *Motivational Statements*:
  Although a lot of previous work exists where numerous classification models have been used, the results about the superiority of one model or method over another or the usefulness of metric-based classification are not always consistent across different studies. There are

various potential sources of bias : comparing classifiers over one or a small number of proprietary data sets, depending on accuracy indicators that are inappropriate for software defect prediction and cross-study comparisons and limited use of statistical testing procedures to secure empirical findings. Thus, there is a need for more reliable research procedures before we conclude the results of the comparative studies of software prediction models.

- *Study Instruments*:
  The paper proposes a framework for organizing comparative classification experiments in software defect prediction. The authors conduct a large-scale benchmark of 22 different classification models over 10 public-domain data sets from NASA Metrics Data (MDP) repository and the PROMISE repository. Comparisons are based on the area under the receiver operating characteristic curve (AUC). The authors argue that AUC represents the most informative and objective indicator of predictive accuracy within a benchmarking context. The authors then use statistical methods to measure statistical significance of the difference between the performance of the various classification models.

- *New Results*:
  The authors have established that AUC is the primary accuracy indicator for comparative studies in software defect prediction as it separates predictive performance from class and cost distributions, which are project-specific characteristics that may be unknown or subject to change. Another contribution was the usage of statistical testing procedures for comparing and contrasting classification models. Further, as per the results obtained, the predictive accuracy of the models did not differ significantly. Thus, the assessment and selection of a classification model should not be based on the predictive accuracy. Instead it should be based on other factors such as computation costs, efficiency, ease of use and comprehensibility.

4.2.3 Ecological Inference in ESE

Posnett *et al.* provided a conceptual framework on how prediction models can vary if metrics that are collected at aggregated with that of metrics collected at non-aggregated level. The "Ecological Inference in Empirical Software Engineering" paper makes use of the conclusion from "Benchmarking Classification Models.." paper - that simple statistic measures like TPR, FPR do not work well in a software defect prediction context as it is possible for two groups to use the same model on same data set and yet come with different results just because they had different threshold values. So the authors use ROC and statistical testing methods in their experiment to model defects.

4.3 Predicting failures with developer networks and social network analysis [10]

4.3.1 Keywords:

- *Developer network*: A network that infers the developer dependencies for the developed software artifacts.

- *Social network analysis (SNA)*: Analyzing the collaborative structure that can be inferred directly, and indirectly from software module dependencies and developer dependencies.

- *Failure prediction*: A methodology to predict failures by mining and analyzing software repositories.

4.3.2 Key Points:

- *Related Work*:
  The authors cited multiple papers that are related to failure prediction, and network analysis. For example, that authors cited the work of Arisholm et al. [1] that examined several data mining techniques for fault prediction for a telecommunications project. Arisholm et al. used multiple techniques such as decision trees, neural networks, and logistic regression.

- *Study Instruments*:
  - Project repository from a telecommunications company called Nortel Networks.
  - File-based metrics: updates, code churn, developers, number of hub developers

- *Model selection*:
  system testing model, post-release model

- *Model validation*:
  cross-fold validation

4.3.3 Use of Ecological Inference

Meneely *et al.* took a holistic approach where social network analysis was performed on one level.

4.4 The Influence of Organizational Structure on Software Quality: An Empirical Case Study [13]

4.4.1 Keywords:

- *Organizational Structure*: The hypothetical structure that is used to categorize the responsibilities and assigner-assignee relationship inside that organization.

- *Software Quality Metric*: Empirical concepts to evaluate the quality of a software

- *Traditional Software Quality Metrics*: Empirical concepts that are derived from software artifacts to quantify/or predict software quality

- *Organizational Software Quality Metrics*: Empirical concepts that are derived from organizational aspects of a software organization

4.4.2 Key Points:

- *Related Work*:
  The paper provides a brief, yet comprehensive overview of empirical studies that discuss the implications of software organization, traditional software metrics such as code churn, code coverage, code complexity, and code dependencies.

- *Study Instruments*:
  - Version control of system used during the release point of Windows Vista, a Microsoft product
  - Step-wise regression, principal component analysis, precision, recall etc.
- *Baseline Results*:
  The prediction accuracy can be used as a baseline for empirical studies that addresses the impact of organizational structure for software quality of software binaries

4.4.3 Use of Ecological Inference

Nagappan *et al.* take a holistic organizational approach, disregarding the difference of aggregated and dis-aggregated levels.

4.5 Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista [5]

4.5.1 Keywords:

- *Distributed Development*: This term refers development of software that is spread across various levels - building, campus, locality, continent, world. Different teams working at different locations work together to build a complex software system.
- *Collocated Development*: This refers to development of software systems in which the teams are working in the same location - say same building or floor. In collocated development, teams can reach out to each other and communication and collaboration becomes relatively easy.
- *Software Quality*: This refers to the quality of software that is being produced. Quality is typically measured in terms of failure/bugs found in the software. A poor quality software will have more bugs and encounter failures.
- *Outsourcing*: It is a special case of distributed development model in which different companies work on building a complex software system.

4.5.2 Key Points:

- *Related Work*:
  The paper provides a brief survey of the important work done in the area of globally distributed software development. The authors discuss the effects of distributed development on bug resolution. Some papers conclude that it indirectly introduces delay while some say it does not have a strong effect. Some conclude only feasible decisions should be made in a project. Previous works have also focused on the effects on quality and productivity. It has been found that there is little, if any, correlation between geographic distance of developers and productivity and defect density. Offshoring can lead to several drawbacks such as more documentation needs. Some previous papers also focused on risk factors and categorized them. They found that globally distributed model did not increase defect density and they come up with key actions that make a successful global development model.
- *Motivational Statements*:
  The term "Distributed Software Development" is a general concept and could be operationalized in various ways. The paper stresses that it is important to consider the way that developers and other entities are distributed. The authors are interested in studying the effect of global distributed software development within the same company. Through this study, they want to confirm or refute whether global distributed software development leads to more failures within the context of this setting. Further, their study is unique since there is no prior work which has done such a large scale study within an organization at multiple levels of separation (building, campus, continent,etc). Further, the authors also want to find out if the binaries that are distributed differ from their collocated counterparts in any significant ways.
- *Hypotheses*:
  The paper formulates two hypotheses: H1: Binaries that are developed by teams of engineers that are distributed will have more post-release failures than those developed by collocated engineers. H2: Binaries that are distributed will be less complex, experience less code churn, and have fewer dependencies than collocated binaries. The first hypothesis is related to code quality or failures that are detected in post-release of software whereas the second hypothesis focuses on the properties of software binaries that result out of the two systems - distributed and collocated development.
- *Future Work*:
  The paper concludes that distributed development has little to no effect. The authors confirm that there are scenarios in which distributed development would work well. They observe that an organizationally compact but geographically distributed project would do well as compared to geographically local and organizationally distributed project. The authors observe a lot of factors and practices that Microsoft followed which played a big role in almost nullifying the negative effects of distributed development as opposed to conventional wisdom and widely held beliefs. The authors mention that as part of future work, they would want to discuss the effects of these practices in detail on the globally distributed software development.

4.5.3 Use of Ecological Inference

Bird *et al.* take a holistic approach, and discuss the global distributed model of software development, at different levels of separation and its effect on software quality.

4.6 Cross-project defect prediction: a large scale experiment on data vs. domain vs. process [22]

4.6.1 Keywords:

- *Post-release Defects*: Defects of a project that are discovered after the release of the software.
- *Cross-project Defect Prediction*: A methodology to use one defect prediction model, to predict defects of another project.

- *Similarity between Projects*: Metrics that measure the similarity between two projects.

4.6.2 Key Points:

- *Related Work*:
  The authors have multiple papers that are related to defect prediction, and cross-project predictions. For example, Turhan et al. [20] analyzed 12 NASA projects which were considered to be cross company. Turhan et al. identified cross-project metrics to be helpful in cross-project defect prediction, along with the increase of false positives.
- *Study Instruments*:
  - Open-source and in-house projects source code repositories.
  - Code measures: added line of code (LOC), total LOC, churn, pre-release bugs, and cyclomatic complexity

4.6.3 Use of Ecological Inference

Zimmerman *et al.* do not address nor discuss the findings achieved at different levels of aggregation of the studied software projects. Without care to mention their methods, it is likely that the authors used ecological inference in the analysis of their hypotheses and are at risk of the ecological fallacy.

4.7 A systematic and comprehensive investigation of methods to build and evaluate fault prediction models [1]

4.7.1 Keywords:

- *Fault Prediction Models*: "binary classifiers typically developed using one of the supervised learning techniques from either a subset of the fault data from the current project or from a similar past project." [8]
- *Fault Proneness*: the number of defects detected in a software component (e.g., class)
- *Cost-Effectiveness*: refers to the return on investment of time and resources expended to achieve a desired outcome. In the context of this paper, cost-effectiveness is measured per model by the ratio of the percentage of the number of lines of code examined (% NOS) to the percentage of totals faults discovered (% faults). Cost effectiveness is used as a variable for success criteria.
- *Verification*: in testing, the act of ensuring that a particular software system meets specifications and meets its intended purpose.

4.7.2 Key Points:

- *Motivational Statements*:
  After performing a literature review of select publications within the field of fault-proneness prediction models, the authors found that none addressed the impact of selecting any particular modeling technique for fault prediction. Therefore, the authors focus the paper on the systematic assessment of three aspects on how to build and evaluate fault-proneness models in context of a Java legacy system development project in an industrial setting. These three aspects are:
  - "compare many data mining and machine learning techniques to build fault-proneness models"
  - "assess the impact of using different metric sets" (e.g., source code structural measures, change history)
  - "compare several alternative ways of assessing the performance of the models", including:
    - ∗ confusion matrix criteria
    - ∗ ranking ability
    - ∗ author-proposed cost-effectiveness measure
- *Related Works*:
  Gathered in the aforementioned literature review, the authors present accounts of multiple fault prediction models in previous works, which they intend to use in their experiment. The authors note in response to related works that existing studies only considered "code structural metrics", and only a subset of studies included any further measures. The authors note that there exist a few studies which compare a comprehensive set of data mining techniques for building fault prediction models; though, none were performed systematically nor did they attempt to evaluate the benefits of including particular structural measures, such as code churn and process measures. Models gleaned from related works include:
  - Neural Network
  - Decorate C4.5
  - SVM
  - Logistic Regression
  - Boost C4.5
  - PART
  - C4.5 + Part
  - C4.5
- *New Results*:
  The authors' defend their research procedures as a way to move forward in fault prediction modeling. Having empirically evaluated "all combinations of three distinct sets of candidate measures (OO structural measures, code churn measures, and process change and fault measures), and eight, carefully selected modeling techniques, using a number of evaluation criteria", they proved (to the community and to themselves) that a gold standard in modeling is unattainable. The confounding effects of the many variables results in the need for ad hoc methods for developing predictive fault models.
- *Future Work*:
  The authors' cost-effectiveness measure (CE) for the purpose of the paper was a surrogate measure. Since publication, the authors have performed a pilot study to asses real CE and return on investment. The C4.5 prediction model was applied in a new release of the Java legacy software that was studied in this paper.

Developers spent one additional week of unit testing with the most fault-prone classes (determined by the authors' fault prediction model), yielding an ROI of about 100% by preventing these faults from continuing into later phases of the the PLC where they would have been more expensive to fix. In actual future work, the authors intend to extend these positive results to a larger-scale for further testing.

4.7.3 Use of Ecological Inference

While it seems that preceding related works focused on performance metrics and evaluation of various singular or combinatorial data-mining and fault prediction models, this paper had the intension of systematically exploring the space, and the authors were eager for results. The outcome was modest, suggesting that what is best for any given system is highly dependent upon the evaluation criteria applied. The authors conclude that it is important that predictive models are justified in context by any evaluation criteria. These results paved the way for the systematic approach to predictive modeling employed by the authors of the original paper, where evaluation criteria of such models was questioned at different levels of aggregation/disaggregation–a shortcoming of the procedures documented in this paper.

4.8 Studying the impact of social structures on software quality [2]

4.8.1 Keywords:

- *Measure of Discussion Contents*: Metrics related to software project development discussion that can be extracted from bug reports, stack traces, source code, and software repositories to measure software quality and software defects.
- *Measure of Social Structures*: Metrics related to developer dependencies during software development, and maintenance that can be extracted from source code, and software repositories to measure software quality and software defects.
- *Measure of Communication Dynamics*: Metrics related to developer communications during software development, and maintenance namely number of messages, length of messages, reply time, and interestingness, and workflow measures. These measures can be extracted from source code, and software repositories to measure software quality and software defects.
- *Post-Release Defects*: Software defects and failures discovered after deployment or release of software.

4.8.2 Key Points:

- *Related Work*:
  The authors of the paper have cited multiple papers that are related to defects prediction, and social analysis. For example, that authors cited the work of Ahlberg et al. [2] that predicted failure-prone software modules using object-oriented code metrics.
- *Study Instruments*:
  - Six months of data obtained from the release of Ecplise 6.0
  - Social interaction measures such as centrality, interestingness, and workflow measures
  - pairwise correlation of social interaction measures
  - hierarchical analysis of logistic regression model

4.8.3 Use of Ecological Inference

Bettenburg *et al.* took a holistic approach where analysis was performed on one level (did not differentiate between aggregated or dis-aggregated level).

4.9 Studying the impact of dependency network measures on software quality [14]

4.9.1 Keywords:

- *Dependency network*: An illustration where software modules are presented as nodes, and edges are represented for dependency between two modules.
- *Ego network*: A network that consists of the modules itself and the other modules it is dependent on.
- *Global network*: A network that consists of all the modules and all the other modules each other is dependent on.

4.9.2 Key Points:

- *Related Work*:
  The authors cited multiple papers that are related to software metrics, code complexity metrics, and dependency network analysis. For example, that authors cited the work of Wolf et al. [21] that analyzed the effect of social developer networks, developer networks on predicting build failures.
- *Study Instruments*:
  - Eclipse project repository.
  - Network measures: density, broker, egoBetween, nEgoBetween, hierarchy, power, closeness, information, dwReach
- *Analysis tools*:
  principal component analysis (PCA), Pearson, Spearman, and Recall

4.9.3 Use of Ecological Inference

Nguyen *et al.* analyze social network analysis on two levels namely local, and global which are labeled as ego, and global, respectively.

5. ECOLOGICAL INFERENCE IN ESE

5.1 Ecological Inference in Empirical Software Engineering [16]

5.1.1 Keywords

We identify the following keywords to be the most significant ones. Each of the keywords are accompanied with definitions.

- *Varying aggregation Levels*: A software system that has a hierarchical organization has different layers, where each higher layer is comprised of sub-layer components. The paper defines each of these higher layers as varying aggregation levels. The paper uses Eclipse as a real-life example: Eclipse consists of modules, which is a collection of packages. Each package is a collection of files. Here, each package can be labelled as an *aggregated level*, and file can be labelled as *dis-aggregated level.*

- *Ecological Inference*: Ecological inference is the empirical finding that is evident at aggregated level of software, as well as, dis-aggregated level of software. For example, in case of ecological inference, if an empirical finding that is evident at package level, which is collection of files, will also be evident at dis-aggregated levels such as files.

- *Ecological Fallacy*: Ecological fallacy is that particular empirical finding that is evident at aggregated level of software, is *not evident* at dis-aggregated level of software. In case of ecological fallacy, if an empirical finding that is evident at package level, will not be evident at dis-aggregated levels such as files.

- *Ecological Inference Risk*: The paper defines empirical inference risk as generalizing an empirical inference that is evident at aggregated level, to a dis-aggregated level without running the same model with the factors existent at the dis-aggregated level.

5.1.2 Key Points

- *Motivational Statement*:
Researchers have mined large scale software repositories at different levels of the software hierarchy and presented their findings. However, what remains unknown is, to what level, a hypotheses that was achieved at aggregated level, is applicable to a dis-aggregated level of the software. This paper presents a conceptual framework that investigates the *ideal* level to look for empirical findings, and whether those findings hold for both: aggregated and dis-aggregated levels of the software.

- *Study Instruments*:
The paper used data extracted from the JIRA tracking system and Github repositories which contained 18 different Apache Software Foundation projects including Cassandra, Lucene, and OpenEJB.

- *Statistical Tests*:
The paper uses hypotheses testing to determine if a certain hypotheses holds at aggregated level and dis-aggregated level. The authors state that they found a number of cases where the null hypotheses is rejected for aggregated level, which however, was not rejected at dis-aggregated level. The opposite observation was also observed and reported in the paper. z-test statistics were used in the paper to determine predictor performance.

- *Future Work*:
One possible future direction that can work on top of this paper is replicating the study for other software repositories which follow different hierarchical architectures, and are implemented in different languages. The authors have reported two levels of the hierarchy: files and packages. Another potential extension of this study can look at the effects of looking at modules and observe if the empirical findings hold.

6. SURVEY - FORWARD FROM 2011

The following papers were composed after Posnett *et al.* published on the prevalence of the ecological inference in Empirical Software Engineering. For each paper, we will provide a bullet point format, defining keywords described by each author, summarizing select ideas presented in each paper, and finally, describing how the paper addressed ecological inference to perform more informed hypothesis testing and analysis.

6.1 Think Locally, Act Globally: Improving Defect and Effort Prediction Models [3]

6.1.1 Keywords:

- *Global Model*: A prediction model that is created by using all the features available in the dataset.

- *Local Model*: A prediction model that is created by using a subset of the features available in the dataset. The other subsets are used in testing the predictive performance of the prediction model.

- *Goodness of Fit*: A statistical testing mechanism that is used to test the predictive performance of any prediction model. Example of some popular goodness of fit tests include Pearson's test, chi-squared test, R-square measure (used in this paper) etc.

- *Connecting Global and Local Models*: The concept of using results from a local model and use it to build a better predictor at the Global level. Posnett et al. in their work [16] stated that the same prediction model can behave differently at different levels of the same software. Menzies et al. [11] stated that smaller subsets of a typical software engineering dataset can provide better insight with respect to prediction. In this paper, Bettenburg et al. uses these findings to build better global models.

6.1.2 Key Points:

- *Motivational Statements*:
Empirical software engineering research has shown evidence that prediction models that work at aggregated and dis-aggregated levels differ in prediction performance. Similar studies have also shown the benefits of using prediction models applied at dis-aggregated levels. Te goal of replicate these findings and use them to create better prediction models that will help software engineering practitioners.

- *Study Instruments*:
  - PROMISE repository datasets:

    * Xalan 2.6
    * Luecene 2.4
    * CHINA
    * NasaCoc
  - Correlational Analysis, VIF Analysis, Cross Validation

- *Scripts*:
  The complete set of tools, code are available at http://sailhome.cs.queensu.ca/replication/local-vs-global/

6.1.3 Acknowledgement of Ecological Inference

Bettenburg *et al.* use the findings Posnett *et al.* as a motivational aspect and try to discover if the prediction results obtained at a dis-aggregated level can be applied to get better prediction results at the aggregated level.

6.2 Recalling the imprecision of cross-project defect prediction [18]

6.2.1 Keywords:

- *Empirical Software Engineering*: Focuses on experiments involving software systems (software products, processes, and resources). The purpose of these experiments is to collect data that can be used to validate theories about the processes involved in software engineering.
- *Fault Prediction*: To make "use of a plethora of structural measures in order to predict faults, even in the absence of a fault history... Example structural measures include lines of code, operator counts, nesting depth, message passing coupling, information flow-based cohesion, depth of inheritance tree, number of parents, number of previous releases the module occurred in, and number of faults detected in the module during the previous release [4]." Accurate fault prediction at an early stage of development, despite the lifecycle phase, allow for code to be fixed at a lower cost [4].
- *Code Inspection*: The practice of reviewing code for any defects or process improvements. May be performed by person, team of people, and/or a model built for fault detection.
- *Cross-Project Prediction*: "using data from one project to predict defects in another [18]." As the title of the paper suggests, this is an imprecise practice.

6.2.2 Key Points:

- *Motivation Statements*:
  Because new projects are slight on historical data, there is growing interest in cross-project prediction–using data from existing projects to predict defects in another. However, to date the results of these studies have been underwhelming. Past projects have used the standard IR-based measures of precision, recall, and f-score, with specific threshold settings determined by methods such as logistic regression. Rahman et al, argue that these are not well suited for cross-project prediction. The authors propose that a variety of trade-offs (viz, 5%, 10%, or 20% of files tested or inspected) would be more suitable.
- *Sampling Procedures*:
  The authors "collected defect data and predictive metrics for several projects". They chose data sets and process metrics considering the following features:
  - Commits - "Number of commits made to this file during release"
  - Active Devs - "Number of developers who made changes during this release"
  - Added - "LOC during this release normalized by file size"
  - Deleted - "Deleted LOC during this release normalized by file size"
  - Changed - "Changed LOC during this release normalized by file size"
  - Features - "Number of new features in this file during this release"
  - Improvements - "Number of improvements in this file during this release"
  - Log - "SLOC Log source lines of code"

  Project List: Axis2, CXF, Camel, Cayenne, Derby, Lucene, OpenEJB, Wicket, XercesJ
- *Statistical Tests*:
  The authors used Logistic Regression, and extension of Linear Regression, to discretely catorgorreplicateize defects into a binary determination of whether a project is default-prone or not. Logistic Regression was chosen because it simple, widely-applicable, and peer reviewed. The used R to program the model, with standard generalized linear model and the binomial link function. Simple approaches were chosen because:
  - "to the extend possible, replicate existing work"
  - "ensure that our findings do not leverage any exotic, powerful techniques"

  Because they were measuring performance and not testing any hypotheses, the authors simply used all available variables.
- *Results*:
  Suspicions were confirmed by the results of the analysis: cross-project fault prediction is significantly worse than prediction based on historical defect data within a project. However, the authors found "that cost-sensitive cross-project prediction is a) as good as within-project prediction, and b) substantially better than what we would expect under a random model." The conclude with the following block statement:

  > „“In terms of aucec, cross-project defect prediction performs surprisingly well and may have a comparable performance to that of the within-project models.”“

6.2.3 Acknowledgement of Ecological Inference

Rahman *et al.* demonstrate the consideration of the risk of ecological inference before proceeding, explicitly stating that their choice of file-level analysis was directly motivated by the publication by Posnett *et al.* .

6.3 Bug prediction based on fine-grained module histories [7]

6.3.1 Keywords:

- *Bug Prediction*: This term refers to the ability to predict a future bug by building a model using the historical metrics, which are mined from version histories of software modules.
- *Fine-grained Prediction*: This term refers to the use of method-level details while using historical metrics, mined from version histories of software modules, to build a model for predicting bugs.
- *Fine-grained Histories*: It refers to the use of fine-grained metrics, by creating metadata or a structure which will store the history or change information at very fine grained level , for example, capturing details of a method level change instead of package level or file level change alone.
- *Historical Metrics*: They are metrics coming from various categories such as code-related metrics, process-related metrics, organizational metrics and geographical metrics for capturing version history in different categories.

6.3.2 Key Points:

- *Related Work*:
  The paper says that bug prediction has been widely studied and recent findings show the usefulness of collecting historical metrics from software repositories for bug prediction models. Prediction models have been build using bug-fix information. Bug prediction has been done using file-level and package-level historical metrics and it has been observed that results from file-level metrics are more interesting. Fine-grained prediction has been a challenge because obtaining method histories from existing version control systems is a difficult problem.
- *Motivational Statements*:
  Although a lot of previous work has been done on bug prediction using file-level and package-level historical metrics, there is a need to do bug prediction at a fine-grained level such as method level. In ESEC/FSE 2011 conference, fine-grained prediction was selected as one of the future directions. Studies of fine-grained prediction are necessary because desirable results may be obtained when compared to coarse-grained prediction.
- *Study Instruments*:
  In order to collect detailed histories, the authors have proposed a fine grained version control system - Historage. Historage was constructed on top of Git and can control method histories of Java. The authors then empirically evaluate the prediction models with eight open source projects written in Java.
- *New Results*:
  The authors have come up with a new framework to capture fine-grained method-level historical metrics. Using eight open source projects, package level , file-level and method-level prediction models were compared based on effort-based evaluation. They found that method-level prediction is more effective than package-level and file-level prediction when considering efforts. They also observed that past bug information on methods does not correlate with post bugs in methods, and organizational metrics may not contribute to method-level prediction. Code-related metrics have positive correlations and interval-related metrics have negative correlations.

6.3.3 Acknowledgement of Ecological Inference

Ecological Inference in Empirical Software Engineering makes use of the conclusion from *Benchmarking Classification Models for software defect prediction* - that simple statistic measures like TPR, FPR do not work well in a software defect prediction context as it is possible for two groups to use the same model on same data set and yet come with different results just because they had different threshold values. So the authors use ROC and statistical testing methods in their experiment to model defects. The authors of this paper use this information and argue that method-level fine-grained historical metrics give interesting bug-prediction models which yield better results as compared to file-level or package-level historical metics.

6.4 Method-level bug prediction [6]

6.4.1 Keywords:

- *Bug Prediction*: This term refers to the ability to predict a future bug by building a model using the historical metrics, which are mined from version histories of software modules.
- *Fine-grained Prediction*: This term refers to the use of method-level details. Here, the author computes source code metrics at the method level along with change metrics to do fine-grained prediction
- *Fine-grained source code changes*: It refers to the code changes happening in the source code at the fine-grained level of methods as compared to changes observed at the file-level or the package-level.
- *Code Metrics*: They are metrics which are directly computed from the source code itself. There are two traditional suites of code metrics : CK metrics and a SCM, which is a set of metrics directly computed at the method level. They are used by author to generate bug prediction models.

6.4.2 Key Points:

- *Related Work*:
  The paper says that bug prediction has been widely studied and bug prediction models have been built using bug-fix information. Bug prediction has been done using file-level and package-level and researchers have been successful in locating files containing bugs. However, there has been no work on predicting bugs at the method level.

- *Motivational Statements*:
  Although a lot of previous work has been done on bug prediction using file-level and package-level historical metrics, there is a need to do bug prediction at a fine-grained level such as method level. The authors observe that although it is helpful to locate the file containing the bugs, most of the times such files are huge and complex in size and hence are easily classified as bug-containing file. However, it takes a lot of manual effort for the developer to identify the exact area containing the bug and thus becomes time consuming and cumbersome. Also, there is a risk of inferential fallacy when transferring empirical findings from an aggregated level. This paper hopes to save manual inspection steps and significantly improve testing effort allocation.

- *Study Instruments*:
  The authors collected the dataset consisting of code, change and bug metrics for 21 software sub-systems. They used source code metrics and change metrics to build bug prediction models.The bug data was obtained from bug tracking systems such as Bugzilla and the code change is linked to bugs through the bug ids found in the commit message of each bug fix.

- *New Results*:
  The experiments performed on 21 Java open-source systems show that the prediction models reach a precision and recall of 84% and 88%, respectively. Furthermore, results indicate that change metrics significantly outperform source code metrics.

6.4.3 Acknowledgement of Ecological Inference

Ecological Inference in Empirical Software Engineering makes use of the conclusion from *Benchmarking Classification Models for software defect prediction* - that simple statistic measures like TPR, FPR do not work well in a software defect prediction context as it is possible for two groups to use the same model on same data set and yet come with different results just because they had different threshold values. So the authors use ROC and statistical testing methods in their experiment to model defects. The authors of this paper use this information and argue that method-level fine-grained changed metrics give interesting bug-prediction models which yield better results as compared to file-level or package-level metrics or method-level source code metrics.

6.5 How, and why, process metrics are better [17]

6.5.1 Keywords:

- *performance*: The authors "compare the performance of different models in terms of both traditional measures such as AUC and F-score, and the newer cost-effectiveness measures"

- *stability*: The authors "compare the stability of prediction performance of the models across time and over multiple releases"

- *portability*: The authors "compare the portability of prediction models: how do they perform when trained and evaluated on completely different projects?"

- *stasis*: The authors "study stasis, viz.., the degree of change (or lack thereof) in the different metrics, and the corresponding models over time. [They] then relate these changes with their ability to prevent defects."

6.5.2 Key Points:

- *Motivational Statement*:
  The literature was divisive on the issues of the quality of process metrics and code metrics for use in defect prediction. The authors cited papers that preferred process metrics over code metrics, papers that preferred code metrics over process metrics, and papers that qualified the use of both. The goal of their research is to understand why and how each metric class (process or code) is effective, and under what conditions.

- *Sampling Procedures*:
  The authors selected 12 Java-based, Apache Software Foundations (ASF) projects from diverse domains. For each project, they extracted the commit history from each project's GIT repository, while also using GIT BLAME on every file at each release to get detailed contributor information. All 12 projects used JIRA, from which the authors extracted defect information and the defect-correcting commits. The JIRA commits were diffed with the GIT commits to label the modified files as defective.

- *Statistical Tests*:
  The output of all prediction models was the probability of defect proneness of files. Minimum thresholds for defect classification varied, while the authors recorded the accuracy, precision, recall, f-measure, receiver operating characteristic (ROC), and cost-effectiveness, stating that different thresholds could alter these values. The authors began by "comparing the performance of process and code metrics in release based prediction settings using AUC, AUCEC and F50". And in their own jargon:

  > „"We compared the AUC performance for all types of metrics for a given learning technique using Wilcoxon tests and corrected the p-values using Benjamini-Hochberg (BH) correction. We also do the same to compare the performance of different learning techniques for a given set of metrics."“

- *Anti-Patterns*:
  The authors acknowledge the work of Posnett et al. in Threats to Validity for avoiding generalization. They note that in order to avoid risk of the ecological fallacy, they have taken measures to compare their findings in a per project setting, reporting similar results. In a previous paper by Rahman [2], it was noted that the file-level was chosen for analysis in response to the risk of ecological fallacy, while the same decision was made for this work without explicit declaration.

6.5.3 Acknowledgement of Ecological Inference

It appears that every paper that follows (chronologically) Posnett *et al.* 's recognition of the risk of ecological fallacy has to, in some way at least once, describe some measure to taken to avoid the risk of it occurring. This paper is not an exception. In threats to validity, as mentioned previously, the authors describe an extra measure taken to cross-compare their results with analyses run per project. It appears that, even today, researchers in defect prediction are still determining what the ecological fallacy means in terms of the organization of their research and their analysis and results.

7. DISCUSSION AND CONCLUSION

Posnett *et al.* raised an issue that the ESE community has not yet fully realized how to sort out. Prior to the publication of *Ecological Inference in Empirical Software Engineering*, statistical methods performed on software project data paid little attention to the levels of aggregations used to test hypothesis and provided little insight on why particular decisions were made. Researchers were not privy to the consequences of the confounding biases spread among the software ecosystem, nor were they attune to their complexity. Thus, the careless use of ecological inference were not an oversight but a result of ignorance of its potential harm.

After the risk of the ecological fallacy was made apparent, researchers were dumbfounded. The incumbent processes for performing an empirical study on software systems were no longer valid, and a new threat to validity had to be addressed in every successive publication. The same is still true. Papers leading up to 2015 still don't quite seem to fully address the problem of ecological inference, or at least don't know quite how to deal with it yet. Current methods tend to stay local to the file level, to possibly avoid inferring across different levels of aggregation altogether. If any inference is made, then the authors perform their due diligence to address it while citing Posnett *et al.* . Today, ecological inference and the risk of ecological fallacy remain the elephant in the empirical software engineering room. No one seems to know quite yet how to acquaint it with the discipline.

8. REFERENCES

[1] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[2] N. Bettenburg and A. E. Hassan. Studying the impact of social structures on software quality. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 124–133. IEEE, 2010.

[3] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 60–69. IEEE Press, 2012.

[4] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Software fault prediction using language processing. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 99–110. IEEE, 2007.

[5] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality?: an empirical case study of windows vista. *Communications of the ACM*, 52(8):85–93, 2009.

[6] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[7] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.

[8] Y. Jiang, J. Lin, B. Cukic, and T. Menzies. Variance analysis in software fault prediction models. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 99–108. IEEE, 2009.

[9] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[10] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[11] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 343–351. IEEE Computer Society, 2011.

[12] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[13] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, pages 521–530. ACM, 2008.

[14] T. H. Nguyen, B. Adams, and A. E. Hassan. Studying the impact of dependency network measures on software quality. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[15] S. Piantadosi, D. P. Byar, and S. B. Green. The ecological fallacy. *American Journal of Epidemiology*, 127(5):893–904, 1988.

[16] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, 2011.

[17] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[18] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.

[19] *The Economist*. Signs of the times, February 22, 2007.

[20] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[21] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, pages 1–11. IEEE Computer Society, 2009.

[22] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.