EM.2023.Project.Team09

Artur Zagitov Ildar Zalialiev Giancarlo Succi

September 2023

Abstract

This study focuses on identifying a minimal subset of metrics that effectively represent the geometric structure of software repositories. Using metric collection tools like SourceMeter, it analyzes C# repositories to collect various metrics, including procedural and OO metrics. The research involves classifying repositories based on specific criteria, such as stars, forks, and C# code proportion. A total of 54 metrics (40 class-level, 14 method-level) are collected. The study employs dimensionality reduction techniques, using Sammon's error function and Kruskal's stress function, optimized through Particle Swarm Optimization (PSO) and Genetic Algorithm (GA). The results demonstrate that a subset of 7-10 class-level and 5-6 method-level metrics can effectively reduce error levels, representing the structural properties of the repositories accurately. This finding is validated using hypothesis testing on a separate validation set.

Introduction

In recent years, the field of software engineering has increasingly turned to metric-based analyses to understand and improve software development processes and products. This report presents a comprehensive study aimed at identifying a minimal set of software repository metrics that effectively capture the structural characteristics of software systems. Our approach uses a variety of metric collection tools to analyze open-source C# repositories from GitHub, followed by advanced statistical and optimization techniques to reduce these metrics into a meaningful, manageable subset.

The project's primary objective is to understand the "structure" of software repositories in a geometrical sense, using metrics to represent classes and methods as points in a multidimensional space. This structure refers to the relationships and properties shared among these elements, such as the number of variables and lines of code. Our methodology involves a thorough selection of repositories based on specific criteria, such as the number of stars and forks, open access status, and a high proportion of C# code. We then employ various Source Meter[1], which offers a unique set of metrics, to gather comprehensive data from these repositories.

The paper systematically discusses the tools used for metric collection, highlighting their strengths, limitations, and compatibility with different C# versions. We also elaborate on our data collection process, which includes automated repository cloning and version analysis modules to ensure compliance with our selection criteria. A significant portion of this study is dedicated to dimensionality reduction techniques. We employ methods like Sammon's error function and Kruskal's stress function, alongside optimization algorithms such as Particle Swarm Optimization (PSO) and Genetic Algorithm (GA), to identify the most representative subset of metrics. Our results not only demonstrate the effectiveness of these techniques in capturing the essence of software structure but also provide practical insights into the number of metrics required for a comprehensive yet concise analysis.

Contents

1 Contribution

Task	Teammate	
	Artur Zagitov	Ildar Zalialiev
Stage 1, 2: Choose a language and search for tools		
Search for tools for different languages	50%	50%
Test tools for C# and make a list of tools	20%	80%
Stage 3: Specify data sources and metrics		
Create an automated tool for searching and filtering repositories	0%	100%
Create a table of data sources	20%	80%
Create a table of metrics and associated tools	20%	80%
Stage 4: Collect data		
Create an automated tool for cloning repositories	0%	100%
Create an automated tool for gathering metrics	0%	100%
Collect metrics for all repositories	20%	80%
Stage 5: Perform the main task		
Research different dimensionality reduction methods	100%	0%
Decrease metrics dimension	100%	0%
Stage 6: State and test hypotheses		
State hypotheses	100%	0%
Test hypotheses	100%	0%
Stage 7, 8: Write report and prepare presentation		
Write a report	90%	10%
Prepare presentation	90%	10%

Table 1: Contribution table

2 Methodology

2.1 Metrics collection tools overview

We have selected C#, because it has a lot of repositories on github that meet the conditions ($\geq$ 30 000), and has a large amount of metric collection tools, that, in total, can provide more then 50 metrics.

To calculate code metrics we have applied only Source Meter [1], due to its large amount of metrics and the ability to automate the workflow through scripts, however, there are other tools, which can be used to collect metrics in C# projects, such as NDepend [2], C# Source Code Metrics [3], Designite [4] and others. Below detailed descriptions of this tools are given.

- **Source Meter** is a tool designed for comprehensive static source code analysis. One of the primary advantages of Source Meter is its capacity to gather a large amount of metrics. For C# projects, it can collect 50 class metrics and 17 method metrics. Furthermore, the tool is user-friendly, requiring only a single command to initiate the metrics collection process, making it easy to integrate into automated workflow through scripts. These are the two main reasons we chose Source Meter.

  However, Source Meter has a limitation in its compatibility with C# versions. Currently, it supports up to C# 8, whereas the latest version of C# is 11. To address this constraint, we plan to implement a mechanism that parses the .csproj (C# project definition) files to determine the language version used in each repository. By doing so, we can filter and process repositories accordingly.

- **NDepend** is a paid program with a free trial for gathering code metrics for .NET projects. Like Source Meter, NDepend allows to create project reports with large number of metrics at both class and method levels. It supports 24 class metrics and 19 method metrics for C# projects.

  However, it's important to mention that NDepend is a commercial tool and not freely available. Nevertheless, it does offer a 14-day trial period.

- **C# Source Code Metrics** is another free tool for code metrics collection for C# code. Its main distinguishing feature and the reason we chose this tool is its ability to calculate Halstead metrics, which include Halstead volume, Halstead difficulty, Halstead effort, Halstead operators, and Halstead operands for C# code snippets.

  The main drawback of this tool is its limited compatibility, as it can only analyze code written in C# versions up to 6.

- **Designite** is a freely available C# static analysis tool. it can identify code smells, perform trend analysis, and, most importantly for our project, extract object-oriented code metrics. This tool is able to collect 13 different metrics (3 at method level, 11 at class level).

We have also looked into some other metrics collection tools for C#, such as Microsoft.CodeAnalysis.Metric, SonarQube and Metrix++, however, these tools did not provide the ability to gather metrics at class and method levels separately.

2.2 Data sources

This section outlines our repository selection methodology and presents a table containing links to the selected repositories. We established specific criteria for repository selection, as follows:

1. The repository must have a minimum of 10 stars.
2. The repository must have at least 10 forks.
3. The repository must not be archived.
4. The repository must be open access and public.
5. The repository should contain a minimum of 10 commits.
6. Repositories should be sorted in descending order of size.
7. At least 60% of the code in the repository must be written in C#.

8. The repository must consist of at least 60,000 lines of C# code.

However, it's important to note that some of our tools, primarily Source Meter, support only up to C# 8. Moreover, determining the version of the C# language used by a repository without compilation is not feasible. Nevertheless, an official table [5] maps the .NET Runtime version to its default C# language version, as illustrated in Table 2.

Target	**Version**	**C# Language Version Default**
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

Table 2: C# Language Versions by .NET Target and Version

Thus, to address this limitation, we introduced an additional criterion for repository selection based on this table:

9. Target .NET Runtime version used by any project inside the solution must not exceed .NET Core 3.

To automate the process of searching, filtering, and cloning repositories based on these criteria, we developed a Python-based tool with several modules or steps:

- **Repository Discovery Module**: This module utilizes the GitHub API to collect and filter repositories based on the first seven criteria. It compiles a table with links to the selected repositories and provides additional information, such as star ratings and the percentage of code in C#. However, the last two criteria cannot be determined using the API alone. To ensure compliance, two additional modules were added.

- **Automated Cloning Module**: This module automates the cloning of all repositories collected by the first step. Due to its time-intensive nature, which may take up to 10 hours to complete, we implemented fault-tolerant techniques and the ability to resume progress from the last checkpoint in case of an unexpected interruption. Additionally, this module includes a timeout feature for cloning individual repositories to skip those that are excessively large or contain many heavy files. The module also updates the repository table with the location of each repository and an error code in the event of unsuccessful cloning.

- **C# Language Version Analysis Module**: This module scans each repository for files with the ".csproj" extension and parses them using XML to gather information about the .NET Runtime versions used by each project. This allows us to ensure compliance with the last criterion regarding the C# language version. Furthermore, it calculates the lines of code in C# in each repository, ensuring compliance with the eighth criterion.

In the initial step, we collected a total of 790 repositories from GitHub that met the first seven criteria. However, after filtering based on the last two criteria, only 118 repositories remained, as listed in Table A1 in the appendix, which we selected as our data sources.

2.3 Metrics

In this section, we outline the metrics utilized, all of which were collected using the SourceMeter Tool.

In total, we have collected 54 metrics, comprising 40 class-level metrics and 14 method-level metrics. We included only those metrics that produced at least one non-zero result for either a class or method during testing of SourceMeter on one of the repositories. Table A2 in the appendixpresents the abbreviations, full names, and brief descriptions of these metrics, as documented in the SourceMeter documentation [1]. Note, that accessing the SourceMeter documentation requires downloading the tool.

2.4 Data collection

For collection of code metrics for each project the python script was implemented and used with SourceMeter tool. For each repository two files were created with metrics for each class in the solution and for each method in the solution respectively. Both script and the collected data can be found in repository of this project https://github.com/threeteck/EM.Project.Team9.

2.4.1 Results of the Descriptive Analysis

After data collection a descriptive analysis was performed. The short description of the results of this analysis are outlined in the following subsections, the full code of the analysis and its results can also be found in the project repository.

2.4.1.1 Missing Values and Zero Columns

The analysis reveals that there are no missing values in the class metrics, while the method metrics exhibit a small percentage of missing values, particularly in the metric of Number of Outgoing Invocations (NOI). Moreover, a subset of the class and method metrics were found to have zero columns in certain projects, notably including Clone Coverage (CC), Depth of Inheritance Tree (DIT), Logical Lines of Duplicated Code (LLDC), Number of Ancestors (NOA), Number of Children (NOC), Number of Descendants (NOD), and Number of Parents (NOP), as illustrated in Figure 1.

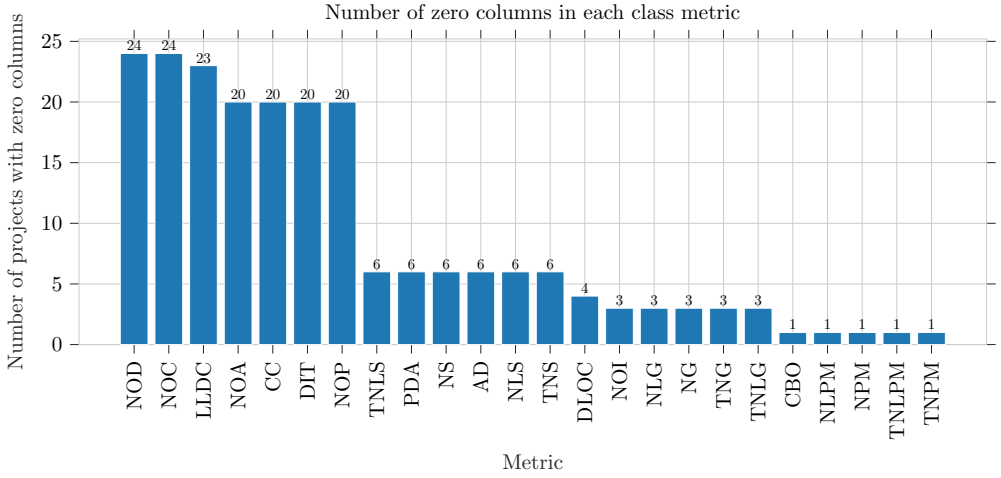

Figure 1: Number of projects with zero columns for each class metric

2.4.1.2 Statistics of Each Metric

The statistical exploration of the metrics points to significant variance in several metrics across projects, particularly evident in metrics related to the number of methods in a class and lines of code. Furthermore, an observation suggests that the majority of repositories contain a limited number of duplicated lines of code. Finally, it was found that Death of Inheritance Tree (DIT) metric has very low mean and standard deviation below 1, meaning that most classes in C# repositories inherits only from one parent or do not inherit at all.

Figures 2 and 3 show box plots that represent the distribution of mean values for each metric, calculated within individual repositories. These plots give an overview of how these mean values vary across different repositories.

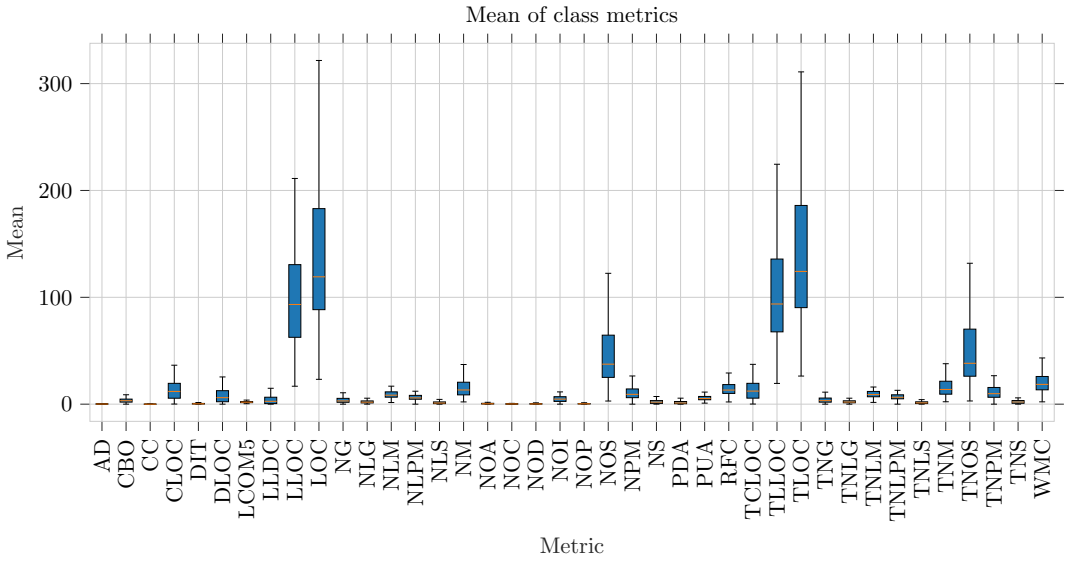

Figure 2: Box Plot of the Mean Values for Each Metric within Individual Repositories - Class-Level Analysis

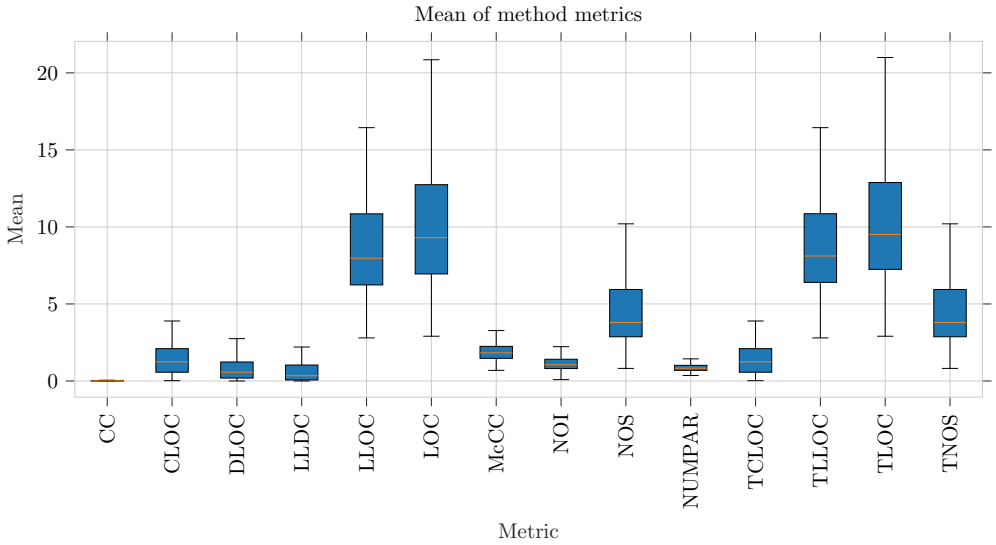

Figure 3: Box Plot of the Mean Values for Each Metric within Individual Repositories - Method-Level Analysis

2.4.1.3 Euclidean Distance Within Repositories

The computation of the mean Euclidean distance between classes and methods within each project provides insights into the structural differences and similarities across projects. The resulting distributions demonstrate a pattern resembling a normal distribution with extended tails, indicating variations in the project structures. Additionally, the hypothesis testing concerning the mean Euclidean distance does not support the assumption of a lesser mean distance for class metrics within the same folder.

2.4.1.4 Correlation Analysis Results

The correlation analysis of the metrics highlights a substantial number of metric pairs exhibiting high correlations, some surpassing 0.9. Notably, specific metrics related to the number of lines and statements have strong correlations, along with metrics such as Depth of Inheritance Tree (DIT), Number of Ancestors (NOA), Number of Statements (NOS), and Total Number of Statements (TNOS) also displaying significant correlations.

An effort was made to calculate the maximum set of metrics, such that no two metrics have correlation greater then 0.75. To do this, the problem was reformulated as a graph problem: find the maximum independent set of the graph, where the nodes are the metrics and the edges connect metrics that have a correlation ≥ 0.75. The maximum independent set was computed with networkx python package, which uses an approximate algorithm based on work [6]. Additionally, examining the constructed correlation graph can provide further insights. Figure 4 depicts this graph, highlighting in red the nodes that are part of the maximum independent set.

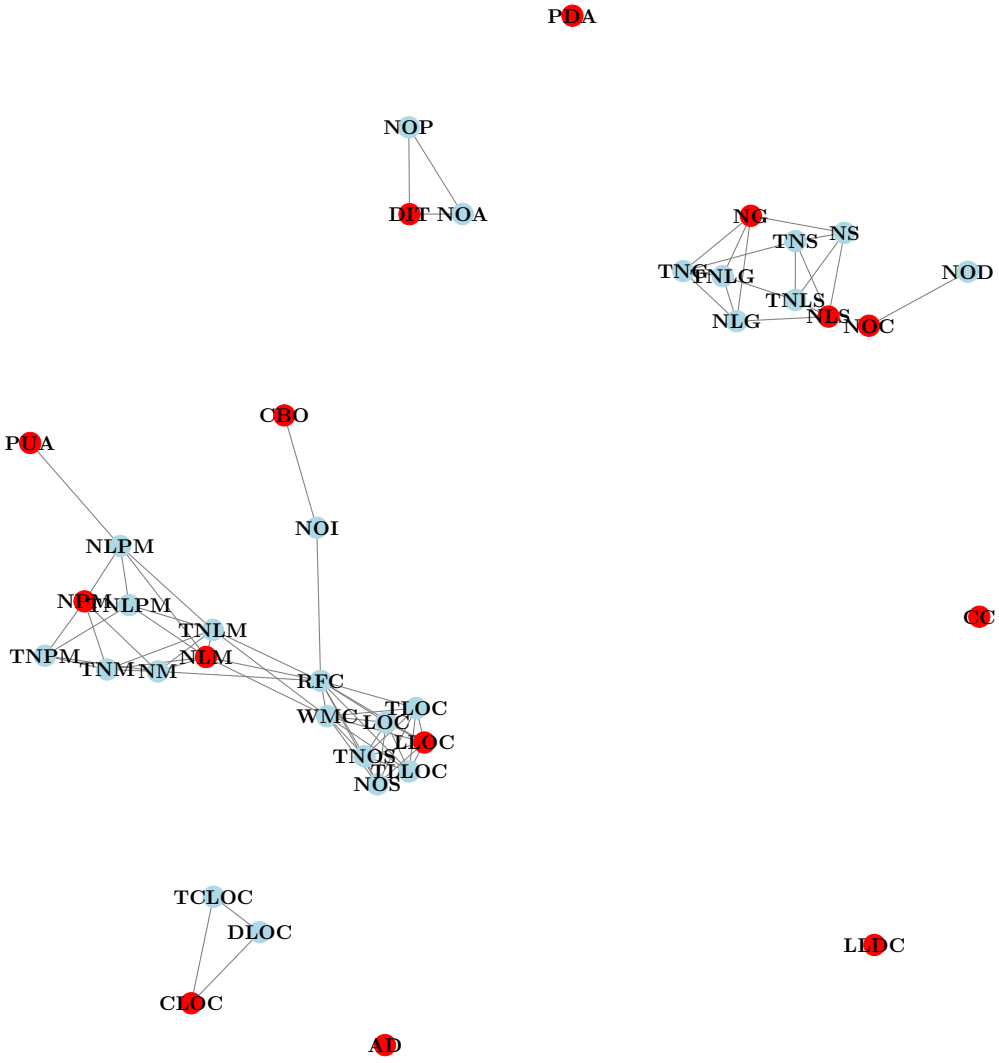

Figure 4: Correlation graph of metrics

After calculating the maximum independence sets for class metrics and method metrics it was found, that only 21 metrics in total, 14 for class level metrics and 7 for method level metrics. These metrics are listed below:

Maximum independent set of class metrics:

- Comment Lines of Code (CLOC)
- Number of Children (NOC)
- Public Undocumented API (PUA)
- Logical Lines of Code (LLOC)
- API Documentation (AD)
- Number of Local Methods (NLM)
- Number of Public Methods (NPM)
- Logical Lines of Duplicated Code (LLDC)
- Coupling Between Object classes (CBO)
- Depth of Inheritance Tree (DIT)
- Public Documented API (PDA)
- Clone Coverage (CC)
- Number of Getters (NG)
- Number of Local Setters (NLS)

Maximum independent set of method metrics:

- Comment Lines of Code (CLOC)
- Logical Lines of Duplicated Code (LLDC)
- Number of Outgoing Invocations (NOI)
- Documentation Lines of Code (DLOC)
- Logical Lines of Code (LLOC)
- Clone Coverage (CC)
- McCabe's Cyclomatic Complexity (McCC)

3 Dimensionality reduction

3.1 Data Representation

In our study, we analyze software repositories to identify the best subset of code metrics. Each repository is represented by two matrices, X_c and X_m, corresponding to class-level and method-level metrics, respectively. The rows in these matrices represent individual classes or methods, while the columns correspond to different code metrics.

3.2 Cost Functions and Distance Measures

The objective is to find a subset of code metrics that preserves the structure of the repository. We employ non-linear cost functions for this purpose: Sammon's error function[7] and Kruskal's stress function[8]. Both functions rely on the concept of distance in metric space, with the Euclidean distance serving as our chosen distance measure.

3.2.1 Sammon's Error Function

Sammon's error function aims to minimize the discrepancy between distances in the original metric space and the reduced space [7]. The function is defined as:

$$E = \frac{1}{\sum_{i<j} d_{ij}} \sum_{i<j} \frac{(d_{ij} - \hat{d}_{ij})^2}{d_{ij}} \tag{1}$$

where E is the Sammon's error, d_{ij} is the Euclidean distance between points i and j in the original space, and $\hat{d}_{ij}$ is the distance in the reduced space.

3.2.2 Kruskal's Stress Function

Kruskal's stress function [8] measures the discrepancy between high-dimensional and reduced space distances, aiming to faithfully represent the original structure in a lower-dimensional space. It is defined as:

$$E = \sqrt{\frac{\sum_{i<j}(d_{ij} - \hat{d}_{ij})^2}{\sum_{i<j} d_{ij}^2}} \tag{2}$$

where E represents Kruskal's stress value, d_{ij} is the distance between points i and j in the original space, and $\hat{d}_{ij}$ is the distance in the reduced space.

It is important to note, that usually Kruskal's stress function is evaluated using geodesic distance, based on the shortest path in the graph, which connects points through kNN algorithm. However, due to resource and time limitations, it was chosen to instead compute Kruskal's stress using Euclidean distance.

3.3 Fitness Function

The fitness function, utilized by optimization algorithms, accepts a parameter vector $p \in R^n$, where n is number of code metrics, that will be optimized by optimization algorithms, which consists of values p_i between 0 and 1. The Fitness function first selects the top k values of the parameter vectors and corresponding code metrics and calculates the cost function using the original matrix X_c or X_m and the reduced matrix with selected subset of metrics.

3.4 Optimization Algorithms

Two optimization algorithms are employed: Genetic Algorithm [9] and Particle Swarm Optimization (PSO) [10].

3.4.1 Particle Swarm Optimization (PSO)

Particle Swarm Optimization algorithm[10] is inspired by the social behavior of birds and fish. The algorithm involves a swarm of particles moving through the solution space, each adjusting its trajectory based on its own experience and the experience of neighboring particles. The key steps in PSO include:

1. **Initialization**: Generate random solutions.
2. **Velocity and Position Update**: Update each particle's velocity and position.
3. **Evaluation**: Assess the fitness of each particle.
4. **Update Bests**: Update the personal and global bests.
5. **Termination**: Repeat until the stopping criterion is met.

The velocity of particles are updated using the following equation:

$$\mathbf{v}_i^{(t+1)} = w\mathbf{v}_i^{(t)} + c_1 r_1(\mathbf{p}_{\text{best},i} - \mathbf{x}_i^{(t)}) + c_2 r_2(\mathbf{g}_{\text{best}} - \mathbf{x}_i^{(t)}) \tag{3}$$

Where:

- $\mathbf{v}_i^{(t+1)}$ is the new velocity of particle i at iteration $t+1$
- $\mathbf{v}_i^{(t)}$ is the current velocity of particle i at iteration t
- $\mathbf{x}_i^{(t)}$ is the current position of particle i at iteration t
- $\mathbf{p}_{\text{best},i}$ is the personal best position of particle i
- $\mathbf{g}_{\text{best}}$ is the global best position among all particles
- w is the inertia weight
- c_1 and c_2 are the cognitive and social parameters, respectively
- r_1 and r_2 are random numbers between 0 and 1

The position of particles are updated using the following equation

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)} \tag{4}$$

Where

- $\mathbf{x}_i^{(t+1)}$ is the new position of particle i at iteration $t+1$
- $\mathbf{x}_i^{(t)}$ is the current position of particle i at iteration t
- $\mathbf{v}_i^{(t+1)}$ is the new velocity of particle i at iteration $t+1$

For hyperparameters we have used the default following values:

- Cognitive parameter $c_1 = 0.8$
- Social parameter $c_2 = 0.9$
- Inertia weight $w = 0.58$
- Number of particles = 30
- Number of iterations = 30

The values for cognitive parameter, social parameter and inertia weight are default common parameters used for PSO, however, values for number of particles and number of iterations were chosen based on resource and time limitations.

3.4.2 Genetic Algorithm (GA)

The GA is an evolutionary algorithm inspired by natural selection, used for solving both constrained and unconstrained optimization problems [9]. The basic steps of GA are:

1. **Initialization**: Generate a random population of N individuals (solutions). Each individual is represented by a chromosome (a string of genes).
2. **Evaluation**: Calculate the fitness of each individual in the population. The fitness function depends on the problem being solved.
3. **Selection**: Select individuals for reproduction. Better individuals have a higher chance of being selected. This can be implemented through methods like roulette wheel selection or tournament selection.
4. **Crossover**: Pair selected individuals and exchange parts of their chromosomes to create new individuals (offspring). The crossover probability p_c determines the likelihood of crossover occurring between pairs. A common crossover technique is single-point crossover, where a point on the parent organisms' chromosome strings is selected and the genetic material is exchanged after this point.
5. **Mutation**: Apply mutation to the offspring at a mutation rate p_m. Mutation involves flipping some bits in the chromosome, which introduces new genetic material into the population and helps maintain genetic diversity.
6. **Accepting**: Replace some or all of the population with the new offspring, depending on the replacement strategy.
7. **Termination**: Repeat the process for a set number of generations or until a stopping criterion is met (e.g., a solution of satisfactory fitness is found).

Mathematically, the crossover and mutation operations can be represented as:

- **Crossover**: Given two parent chromosomes C_1 and C_2, the crossover operation produces two offspring O_1 and O_2. For a single-point crossover at point k, this can be represented as:

$$O_1 = \{c_{11}, c_{12}, ..., c_{1k}, c_{2(k+1)}, ..., c_{2n}\} \tag{5}$$
$$O_2 = \{c_{21}, c_{22}, ..., c_{2k}, c_{1(k+1)}, ..., c_{1n}\} \tag{6}$$

  where n is the length of the chromosome.

- **Mutation**: For a chromosome $C = \{c_1, c_2, ..., c_n\}$, the mutation operation changes some genes c_i with a probability p_m. For continuous chromosomes, this can be represented as:

$$c_i' = \begin{cases} c_i + \theta & \text{with probability } p_m \\ c_i & \text{otherwise} \end{cases} \tag{7}$$

  where θ is the random number from $N(0, \sigma^2)$ and σ is the hyperparameter.

These operations ensure exploration and exploitation in the search space, leading to the evolution of solutions that are adapted to the fitness function defined for the problem.

For hyperparameters we have used the following values:

- Mutation rate $p_m = 0.1$
- Crossover probability $p_c = 0.7$
- Mutation standard deviation $\sigma = 0.25$
- Population size = 30
- Number of generations = 30

The values for mutation rate, crossover probability and mutation standard deviation were selected after some experimentation, whlie values for population size and number of generations were chosen based on resource and time limitations.

3.5 Performance Optimization Techniques

To enhance the computational efficiency of algorithms, we implemented the following optimization techniques:

- **Vectorization**: Utilizing vectorized operations in place of for-loops for calculations.
- **Caching**: Storing fitness values for subsets of code metrics to avoid redundant computations.
- **Multithreading**: Employing multiple threads to parallelize computation-intensive tasks.

4 Results

To identify the minimal optimal subset of metrics that accurately preserves and represents the structure of the repositories, Sammon's error function and Kruskal's stress function were minimized for each k ranging from 1 to the total number of metrics. Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) were employed as the optimization algorithms for this minimization process. The minimization was performed separately for class and method metrics and for all repositories. Figure 5 illustrates the results of this minimization for class level metrics. For each value of k, the 10th and 90th quantiles, along with the average of the error function, are depicted for each optimization algorithm. The decision to utilize the 10th and 90th quantiles, as opposed to the minimum and maximum values, was based on their greater stability and more robust representation of the error function's margins.

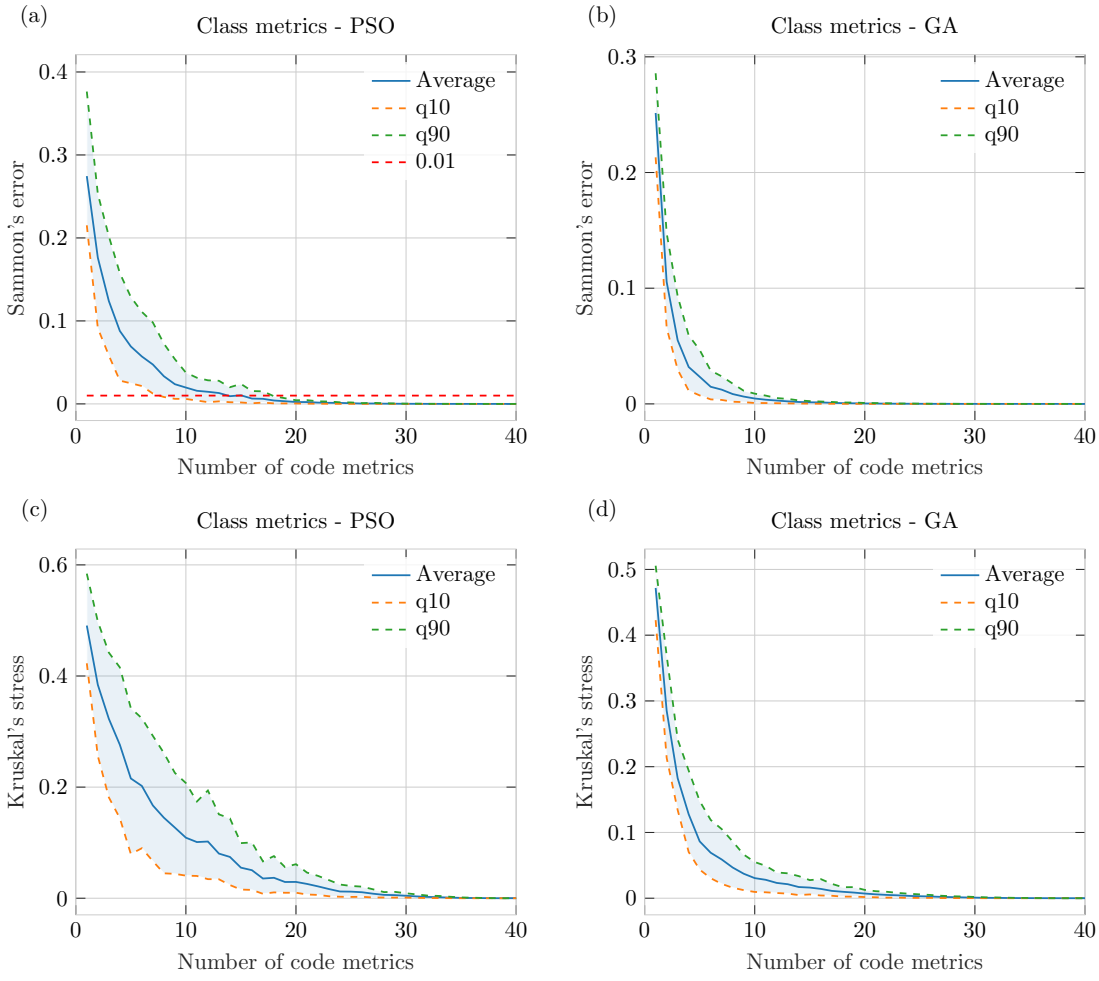

Figure 5: Error function values at different k for class level metrics: (a) Sammon's error obtained using PSO; (b) Sammon's error obtained using GA; (c) Kruskal's stress obtained using PSO; (d) Kruskal's stress obtained using GA

The data was initially separated into training and validation sets, comprising 82 and 36 repositories respectively. Subsequent analysis focused on identifying optimal subsets of metrics within the training set. Analysis of Sammon's error suggests that a subset of 7-10 metrics suffices to reduce the error to near zero, with the inclusion of additional metrics resulting in no significant change. The Kruskal's stress function indicated that the error decreased to less than 0.05 with a subset of approximately 10-15 metrics. These findings support the hypothesis that a set of 10 metrics is enough for representing the class level metrics of repositories.

For the optimization of the error function corresponding to method level metrics, not all methods were considered for certain repositories due to computational constraints, particularly when repositories contained more then 10,000 methods. For repositories exceeding 4,000 methods, a random undersampling technique was applied to reduce the method count to 4,000 prior to optimization. Figure 6 presents the outcomes of these optimization procedures for method level metrics.

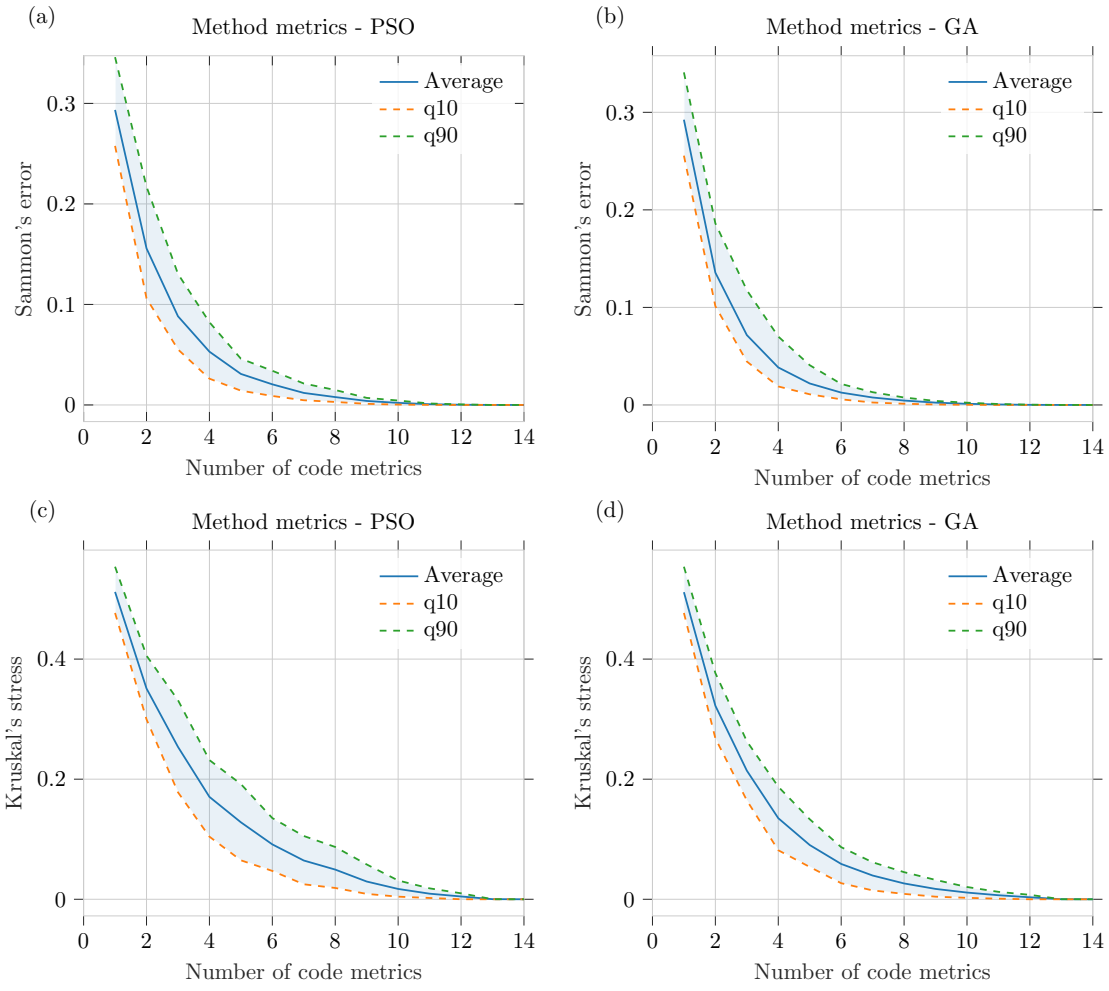

Figure 6: Error function values at different k for method level metrics: (a) Sammon's error obtained using PSO; (b) Sammon's error obtained using GA; (c) Kruskal's stress obtained using PSO; (d) Kruskal's stress obtained using GA

For method level metrics, the minimization of error functions yielded the following results. Sammon's error reached a level below 0.05 with approximately 5 metrics, while Kruskal's stress achieved a similar error threshold with around 6 metrics. Consequently, a decision was made to select 6 method level metrics for further analysis. The full reference table with mean values of Sammon's and Kruskal's errors at each number of metrics (k) for both optimization algorithms is given in the next section in Tables 3 and 4.

Given that each repository's optimization was conducted independently, the optimal subset of metrics for each k varied among the repositories. To define a universal optimal subset of 10 class level metrics and 6 method level metrics, a vote-counting aggregation method was implemented. This technique serves

to aggregate the metrics based on their frequency of occurrence across all optimizations. Figures 7 and 8 illustrate the number of votes for each class and method level metric across all repositories for subsets of 10 and 6 metrics at the class and method levels, averaged among PSO and GA respectively.

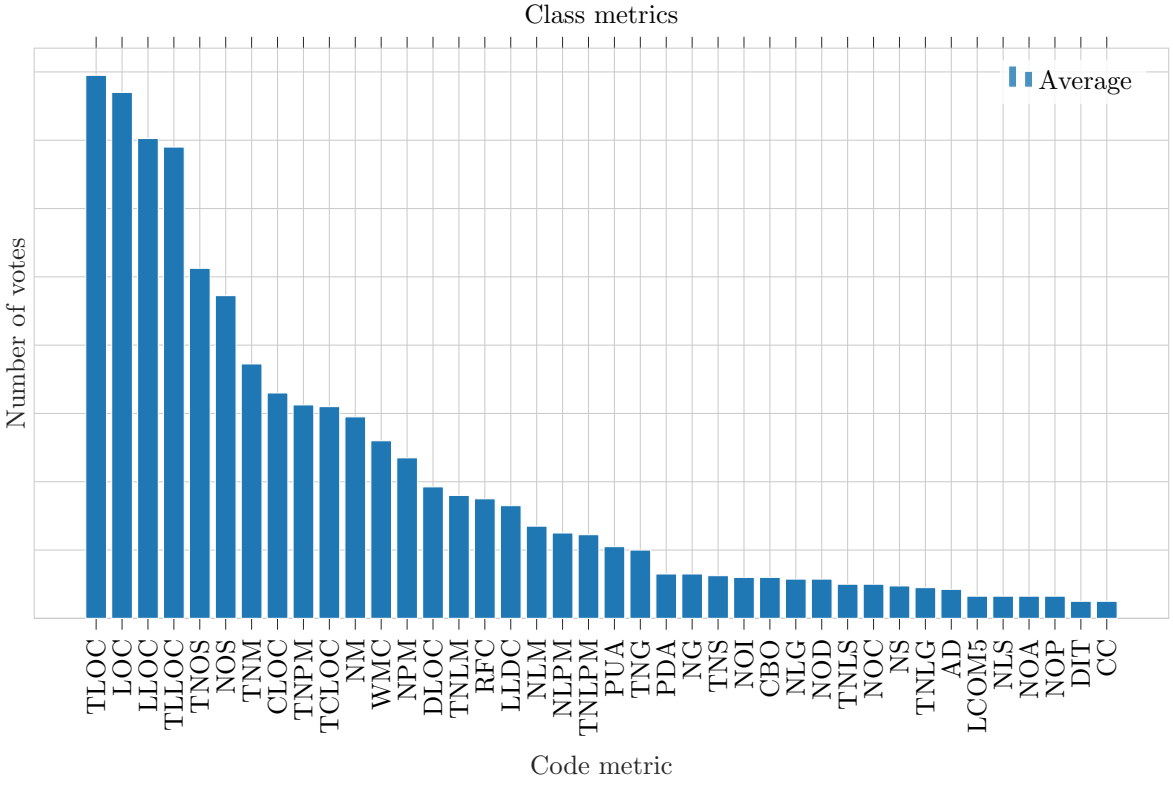

Figure 7: Average vote count for class level code metrics among PSO and GA

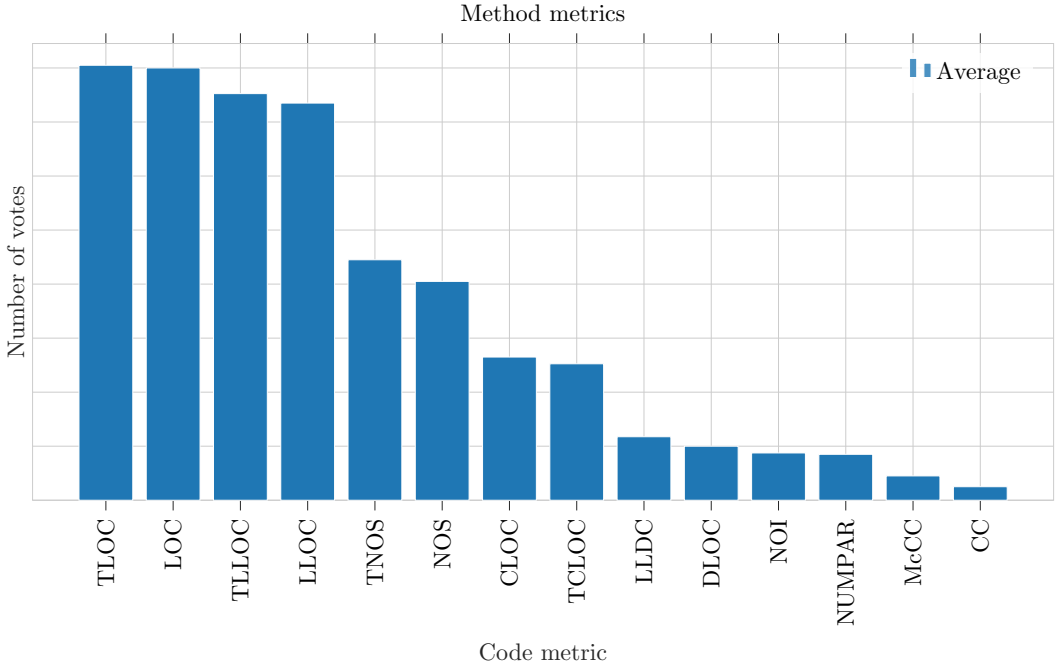

Figure 8: Average vote count for method level code metrics among PSO and GA

Thus, the following metrics for class level metrics were selected for optimal subset:

- TLOC
- LOC
- LLOC
- TLLOC
- TNOS
- NOS
- TNM
- CLOC
- TNPM
- TCLOC

The following metrics for method level metrics were selected for optimal subset:

- TLOC
- LOC
- TLLOC
- LLOC
- TNOS
- NOS

5 Discussion and validation of results

The results obtained from the optimization processes using PSO and GA provide insightful observations on the nature of dimensionality reduction in the context of software repository metrics. Both algorithms successfully identified subsets of metrics that significantly reduce Sammon's error and Kruskal's stress, thereby confirming the effectiveness of these algorithms in dimensionality reduction tasks for software analytics. However, a detailed comparative analysis is required to statistically validate whether the distributions of error functions at each number of metrics in the subset (k) and at both the class and method level metrics provided by PSO and GA are similar. This comparison is essential to understand if one algorithm outperforms the other significantly or if their performance is statistically indistinguishable.

5.1 Hypothesis Testing Results

5.1.1 Hypothesis 1 - Comparison of optimization algorithms

The first hypothesis tests whether the distributions of error functions for class and method level metrics are similar when optimized using Particle Swarm Optimization (PSO) and Genetic Algorithm (GA). We employed the Mann-Whitney U test to compare the distributions of Sammon's error and Kruskal's stress at each k value for both class and method level metrics. This non-parametric test is suitable for comparing two independent samples to determine if they come from the same distribution. The results of these tests are presented in the tables 3 and 4, showing the p-values obtained from comparing the distributions of Sammon's error and Kruskal's stress at each number of metrics in the subset (k), as well as the mean error.

Table 3: Mann-Whitney U test results for class level metrics

k	Sammon p-value	Kruskal p-value	PSO Mean Sammon	GA Mean Sammon	PSO Mean Kruskal	GA Mean Kruskal
1	0.018	0.039	0.277	0.250	0.491	0.465
2	< 0.001	< 0.001	0.174	0.107	0.385	0.282
3	< 0.001	< 0.001	0.125	0.056	0.317	0.179
4	< 0.001	< 0.001	0.086	0.032	0.270	0.126
5	< 0.001	< 0.001	0.067	0.023	0.214	0.086
6	< 0.001	< 0.001	0.057	0.015	0.192	0.068
7	< 0.001	< 0.001	0.046	0.012	0.167	0.057
8	< 0.001	< 0.001	0.035	0.008	0.144	0.046
9	< 0.001	< 0.001	0.027	0.006	0.123	0.036
10	< 0.001	< 0.001	0.022	0.005	0.110	0.030
11	< 0.001	< 0.001	0.017	0.004	0.097	0.028
12	< 0.001	< 0.001	0.016	0.003	0.094	0.023
13	< 0.001	< 0.001	0.013	0.002	0.078	0.021
14	< 0.001	< 0.001	0.009	0.002	0.073	0.017
15	< 0.001	< 0.001	0.010	0.001	0.055	0.016
16	< 0.001	< 0.001	0.007	0.001	0.047	0.013
17	< 0.001	< 0.001	0.006	0.001	0.036	0.011
18	< 0.001	< 0.001	0.004	0.001	0.035	0.009
19	< 0.001	< 0.001	0.003	0.001	0.030	0.008
20	< 0.001	< 0.001	0.002	0.000	0.028	0.007
21	< 0.001	< 0.001	0.002	0.000	0.024	0.006
22	< 0.001	< 0.001	0.002	0.000	0.020	0.005
23	< 0.001	< 0.001	0.001	0.000	0.016	0.004
24	< 0.001	< 0.001	0.001	0.000	0.012	0.004
25	< 0.001	< 0.001	0.001	0.000	0.011	0.003
26	< 0.001	< 0.001	0.000	0.000	0.010	0.003
27	< 0.001	< 0.001	0.000	0.000	0.008	0.002
28	< 0.001	< 0.001	0.000	0.000	0.006	0.002
29	< 0.001	< 0.001	0.000	0.000	0.005	0.001
30	< 0.001	< 0.001	0.000	0.000	0.004	0.001
31	< 0.001	< 0.001	0.000	0.000	0.003	0.001
32	< 0.001	< 0.001	0.000	0.000	0.002	0.001
33	< 0.001	< 0.001	0.000	0.000	0.002	0.000
34	< 0.001	< 0.001	0.000	0.000	0.001	0.000
35	< 0.001	< 0.001	0.000	0.000	0.001	0.000
36	< 0.001	< 0.001	0.000	0.000	0.000	0.000
37	< 0.001	< 0.001	0.000	0.000	0.000	0.000
38	< 0.001	< 0.001	0.000	0.000	0.000	0.000
39	0.063	0.021	0.000	0.000	0.000	0.000
40	1.000	1.000	0.000	0.000	0.000	0.000

Table 4: Mann-Whitney U test results for method level metrics

k	Sammon p-value	Kruskal p-value	PSO Mean Sammon	GA Mean Sammon	PSO Mean Kruskal	GA Mean Kruskal
1	0.874	0.834	0.292	0.291	0.512	0.510
2	< 0.001	< 0.001	0.153	0.135	0.350	0.322
3	< 0.001	< 0.001	0.087	0.071	0.250	0.215
4	< 0.001	< 0.001	0.054	0.038	0.170	0.136
5	< 0.001	< 0.001	0.032	0.022	0.130	0.089
6	< 0.001	< 0.001	0.020	0.013	0.093	0.059
7	< 0.001	< 0.001	0.012	0.008	0.064	0.039
8	< 0.001	< 0.001	0.008	0.005	0.048	0.026
9	< 0.001	< 0.001	0.004	0.002	0.030	0.018
10	< 0.001	< 0.001	0.002	0.001	0.018	0.011
11	< 0.001	0.003	0.001	0.000	0.010	0.007
12	0.003	0.006	0.000	0.000	0.005	0.003
13	0.467	0.689	0.000	0.000	0.000	0.000
14	1.000	1.000	0.000	0.000	0.000	0.000

The Mann-Whitney U test results indicate significant differences in the distribution of error functions for different values of k across both class and method level metrics. Furthermore, an examination of the mean values of error functions for different algorithms revealed that the Genetic Algorithm (GA) consistently outperforms the Particle Swarm Optimization (PSO).

5.1.2 Hypothesis 2 - Validation of Optimal Subsets

The reduction in Sammon's error and Kruskal's stress with the increasing number of metrics in the subset is noteworthy. For class level metrics, a subset of around 7-10 metrics was sufficient to reduce the error to near-zero levels, suggesting that beyond this point, additional metrics do not contribute significantly to preserving the structure of the repository. Similarly, for method level metrics, a subset of approximately 5-6 metrics achieved a similar threshold in error reduction. However, it is necessary to verify the generalizability of these results.

The second hypothesis of our study aimed to validate the effectiveness and the generalizability of the identified optimal subsets of metrics in minimizing Sammon's error on a validation set of software repositories. According to the hypothesis, when applying the optimal subsets derived from the training set, Sammon's error on the validation set should be lower than 0.05. To test this hypothesis, we employed a one-tailed t-test, where the null hypothesis (H0) was that the mean Sammon's error using the optimal subsets is greater than or equal to 0.05, while the alternative hypothesis (H1) was that the mean error is less than 0.05. The results of the test are presented in the Table 5.

Metric level	P-value	Sammon's mean error
Class level	9.14×10^{-24}	0.009
Method level	2.92×10^{-8}	0.027

Table 5: P-values and Sammon's mean error for class and method level metrics on validation set using optimal subset

A p-value less than the chosen significance level (commonly 0.05) leads to the rejection of the null hypothesis, indicating that the mean Sammon's error using the optimal subsets is significantly lower than 0.1. This outcome would validate our hypothesis, confirming the efficacy of the identified metric subsets in achieving low-dimensional representations with minimal loss of information.

Conclusion

In conclusion, this project has successfully demonstrated the feasibility of identifying a minimal subset of metrics to represent the geometric structure of software repositories effectively. Using advanced metric collection tools like SourceMeter and statistical techniques, the study has narrowed down the key metrics that accurately capture the structural characteristics of software systems. The results highlight that a subset of 7-10 class-level and 5-6 method-level metrics can significantly reduce the error levels, thus preserving the structural properties of the repositories. This finding has been validated on a separate validation set using hypothesis testing, confirming the generalizability of the results.

For further details and access to the data and scripts used in this project, please refer to the project repository at https://github.com/threeteck/EM.Project.Team9. This repository includes documentation, source code, and data sets, allowing others to replicate the study or extend it with new metrics or repositories.

References

[1] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "SourceMeter SonarQube plug-in," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, (Victoria, British Columbia, Canada), pp. 77–82, IEEE Computer Society, Sept. 2014.

[2] "Ndepend: code metrics collector tool." https://www.ndepend.com/docs/getting-started-with-ndepend. [Online; accessed 21-Sep-2023].

[3] "C# source code metrics." http://www.semdesigns.com/products/metrics/CSharpMetrics.html. [Online; accessed 21-Sep-2023].

[4] "Designite: code metrics collector tool." https://www.designite-tools.com/. [Online; accessed 21-Sep-2023].

[5] "Microsoft: C# language versioning." https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version. [Online; accessed 21-Sep-2023].

[6] R. Boppana and M. Halldórsson, "Approximating maximum independent sets by excluding subgraphs," *BIT*, vol. 32, pp. 180–196, 1992.

[7] J. W. Sammon, "A nonlinear mapping for data structure analysis," *IEEE Transactions on computers*, no. 5, pp. 401–409, 1969.

[8] J. B. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, vol. 29, no. 1, pp. 1–27, 1964.

[9] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[10] J. Kennedy and R. Eberhart, "Particle swarm optimization," *Proceedings of ICNN'95-International Conference on Neural Networks*, vol. 4, pp. 1942–1948, 1995.

Appendix

Table A1: GitHub Repository List

#	Url	Size (Kb)	Stars	Forks
1	https://github.com/egordorichev/BurningKnight	2334	308	32
2	https://github.com/GridProtectionAlliance/openPDC	2277	112	59
3	https://github.com/CobaltWolf/Bluedog-Design-Bureau	2159	115	135
4	https://github.com/MediaBrowser/Emby	1568	3631	778
5	https://github.com/bitwarden/mobile	1397	4988	687
6	https://github.com/etodd/Lemma	1278	566	71
7	https://github.com/microsoft/powerapps-tools	1250	975	415
8	https://github.com/Koziev/NLP_Datasets	1183	319	50
9	https://github.com/Blecki/dwarfcorp	911	554	67
10	https://github.com/cabbagecreek/Marlin3DprinterTool	845	245	65
11	https://github.com/KillzXGaming/Switch-Toolbox	767	795	124
12	https://github.com/SaviorXTanren/mixer-mixitup	702	174	55
13	https://github.com/antonpup/Aurora	684	1806	373
14	https://github.com/microsoftgraph/msgraph-sdk-powershell	673	593	141
15	https://github.com/cxun/WeChatHistory	642	197	87
16	https://github.com/sswelm/KSP-Interstellar-Extended	626	94	79
17	https://github.com/lofcz/SimplexRpgEngine	591	151	19
18	https://github.com/ispysoftware/iSpy	590	1178	524
19	https://github.com/hcmlab/nova	582	161	31
20	https://github.com/hermitdave/FrequencyWords	560	1003	535
21	https://github.com/RickStrahl/Westwind.AspnetCore.LiveReload	451	457	42
22	https://github.com/bcgit/bc-csharp	440	1353	494
23	https://github.com/SubtitleEdit/subtitleedit	438	6053	769
24	https://github.com/svenhb/GRBL-Plotter	420	553	164
25	https://github.com/figormartins/pokemon	411	111	10
26	https://github.com/simon-knuth/scanner	406	358	20
27	https://github.com/lunadream/XWall	405	132	35
28	https://github.com/PokemonUnity/PokemonUnity	403	1559	449
29	https://github.com/BlueMystical/EDHM_UI	398	138	13
30	https://github.com/asheigithub/apple-juice-actionscript	386	141	22
31	https://github.com/NotCoffee418/TrinityCreator	355	147	88
32	https://github.com/AvaloniaUI/Avalonia.Xaml.Behaviors	355	285	37
33	https://github.com/WorldWideTelescope/wwt-windows-client	351	179	57
34	https://github.com/AlbertMN/AssistantComputerControl	351	836	68
35	https://github.com/LavaGang/MelonLoader	349	1751	414
36	https://github.com/miyconst/Mi899	341	183	40
37	https://github.com/microsoft/OAT	340	89	15
38	https://github.com/kopaka1822/ImageViewer	324	256	31
39	https://github.com/dmustanger/7dtd-ServerTools	324	182	59
40	https://github.com/UnigramDev/Unigram	322	3084	406
41	https://github.com/PowerPointLabs/PowerPointLabs	312	167	96
42	https://github.com/rjamesnw/v8dotnet	309	151	39
43	https://github.com/d2phap/ImageGlass	308	5795	430
44	https://github.com/subhra74/xdm	304	5699	1037
45	https://github.com/microsoft/sqltoolsservice	283	384	142
46	https://github.com/octgn/OCTGN	282	350	130
47	https://github.com/Root-MtX/Nro2Nsp	282	257	43
48	https://github.com/rocksdanister/lively	274	12244	952
49	https://github.com/wieslawsoltes/PanAndZoom	267	260	35
50	https://github.com/saucepleez/taskt	266	882	321

Continued on next page

Table A1 – Continued from previous page

#	Url	Size (Kb)	Stars	Forks
51	https://github.com/brminnick/FaceOff	245	90	43
52	https://github.com/digital-standard/ThreeDPoseUnityBarracuda	244	1355	254
53	https://github.com/fcscript/CSharpHotUpdate	234	128	26
54	https://github.com/veldrid/veldrid	231	2294	253
55	https://github.com/DCourtel/Wsus_Package_Publisher	231	201	33
56	https://github.com/soopercool101/BrawlCrate	231	118	31
57	https://github.com/ModuleArt/quick-picture-viewer	226	573	57
58	https://github.com/lllyasviel/YGOProUnity_V2	216	275	117
59	https://github.com/WFCD/WFinfo	215	187	43
60	https://github.com/SharpRepository/SharpRepository	214	649	163
61	https://github.com/Nethereum/Unity3dSampleTemplate	211	83	46
62	https://github.com/PistonDevelopers/VisualRust	208	700	79
63	https://github.com/OndrejNepozitek/Edgar-DotNet	201	290	34
64	https://github.com/microsoft/RecursiveExtractor	201	164	22
65	https://github.com/GrapeshotGames/ServerGridEditor	200	82	140
66	https://github.com/xamarin/KimonoDesigner	198	256	63
67	https://github.com/mattjohnsonpint/GeoTimeZone	190	202	34
68	https://github.com/dotnet/TorchSharp	188	974	140
69	https://github.com/laochiangx/ABP-ASP.NET-Boilerplate-Project-CMS	180	750	308
70	https://github.com/Colanderp/UnityTutorials	179	299	121
71	https://github.com/ShokoAnime/ShokoDesktop	178	96	26
72	https://github.com/planetarium/libplanet	176	490	140
73	https://github.com/charlesw/tesseract	176	2062	726
74	https://github.com/compomics/ThermoRawFileParser	173	140	45
75	https://github.com/cadon/ARKStatsExtractor	172	433	238
76	https://github.com/machine/machine.specifications	169	872	184
77	https://github.com/punker76/simple-music-player	158	338	122
78	https://github.com/visose/Robots	156	239	110
79	https://github.com/Kerbalism/Kerbalism	155	157	96
80	https://github.com/SuRGeoNix/Flyleaf	152	528	63
81	https://github.com/dotnetcore/Magicodes.IE	151	1942	459
82	https://github.com/architecture-building-systems/revitpythonshell	149	437	106
83	https://github.com/dotnetprojects/DotNetSiemensPLCToolBoxLibrary	144	333	184
84	https://github.com/filoe/cscore	143	2030	457
85	https://github.com/microsoft/Office365APIEditor	141	126	55
86	https://github.com/jaskie/PlayoutAutomation	141	161	38
87	https://github.com/OpenWrap/openwrap	140	172	39
88	https://github.com/Tichau/FileConverter	139	3374	334
89	https://github.com/WindowsGSM/WindowsGSM	139	360	78
90	https://github.com/wieslawsoltes/ColorPicker	138	176	17
91	https://github.com/Doublevil/Houhou-SRS	138	142	28
92	https://github.com/monkog/3D-Virtual-Fitting-Room	136	87	37
93	https://github.com/dram55/MarioMaker2OCR	135	88	10
94	https://github.com/Raicuparta/nomai-vr	133	310	19
95	https://github.com/stefsietz/nn-visualizer	126	311	39
96	https://github.com/SolrNet/SolrNet	125	921	802
97	https://github.com/TylerTimoJ/LMCSHD	125	81	15
98	https://github.com/AlturosDestinations/Alturos.Yolo	125	402	128
99	https://github.com/Visual-Stylecop/Visual-StyleCop	124	82	25
100	https://github.com/stratisproject/StratisFullNode	123	84	61

Continued on next page

Table A1 – Continued from previous page

#	Url	Size (Kb)	Stars	Forks
101	https://github.com/aloneguid/parquet-dotnet	123	415	107
102	https://github.com/zlynn1990/SpaceSim	122	160	20
103	https://github.com/pdone/FreeControl	121	466	93
104	https://github.com/dib0/NHapiTools	120	82	59
105	https://github.com/DeadlyCrush/DeadlyTrade	120	148	17
106	https://github.com/thesupersonic16/HedgeModManager	119	189	58
107	https://github.com/ClassicUO/ClassicUO	119	481	298
108	https://github.com/Squalr/Squalr	107	1298	156
109	https://github.com/zgynhqf/Rafy	106	86	71
110	https://github.com/AutoFixture/AutoFixture	105	3085	339
111	https://github.com/NightlyRevenger/TataruHelper	105	277	48
112	https://github.com/GregBahm/PlaceViewer	104	390	42
113	https://github.com/mono/taglib-sharp	103	1169	304
114	https://github.com/JasonXuDeveloper/JEngine	102	1711	311
115	https://github.com/n00mkrad/cupscale	102	1957	107
116	https://github.com/bitbeans/SimpleDnsCrypt	101	2187	239
117	https://github.com/hudec117/Mpv.NET-lib-	101	127	36
118	https://github.com/mperdeck/jsnlog	100	188	37

Table A2: Metric Definitions

Abbr.	Name	Description
	Class Metrics	
AD	API Documentation	Ratio of the number of documented public methods in the class +1 if the class itself is documented to the number of all public methods in the class + 1 (the class itself); however, the nested and anonymous classes are not included.
CBO	Coupling Between Object classes	Number of directly used other classes (e.g. by inheritance, function call, type reference, attribute reference). Classes using many other classes highly depend on their environment, so it is difficult to test or reuse them; furthermore, they are very sensitive to the changes in the system.
CC	Clone Coverage	Ratio of code covered by code duplications in the source code element to the size of the source code element, expressed in terms of the number of syntactic entities (statements, expressions, etc.).
CLOC	Comment Lines of Code	Number of comment and documentation code lines of the class, including its local methods and attributes; however, its nested and anonymous classes are not included.
DIT	Depth of Inheritance Tree	Length of the path that leads from the class to its farthest ancestor in the inheritance tree.
DLOC	Documentation Lines of Code	Number of documentation code lines of the class, including its local methods and attributes; however, its nested and anonymous classes are not included.
LCOM5	Lack of Cohesion in Methods 5	Number of functionalities of the class. One of the basic principles of object-oriented programming is encapsulation, meaning that attributes belonging together and the operations that use them should be organized into one class, and one class shall implement only one functionality, i.e. its attributes and methods should be coherent. This metric measures the lack of cohesion and computes into how many coherent classes the class could be split. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between the two nodes if and only if a common (local or inherited) attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph except the ones that contain only constructors, destructors, getters or setters – as they are integral parts of the class.
LLDC	Logical Lines of Duplicated Code	Number of logical code lines (non-empty, non-comment lines) covered by code duplications in the source code element.
LLOC	Logical Lines of Code	Number of non-empty and non-comment code lines of the class, including the non-empty and non-comment lines of its local methods; however, its nested and anonymous classes are not included.
LOC	Lines of Code	Number of code lines of the class, including empty and comment lines, as well as its local methods; however, its nested and anonymous classes are not included.
NG	Number of Getters	Number of getter methods in the class, including the inherited ones; however, the getter methods of its nested and anonymous classes are not included. Methods that override abstract methods are not counted.

Continued on next page

Table A2 – Continued from previous page

Abbr.	Name	Description
NLG	Number of Local Getters	Number of local (i.e. not inherited) getter methods in the class; however, the getter methods of its nested and anonymous classes are not included. Methods that override abstract methods are not counted.
NLM	Number of Local Methods	Number of local (i.e. not inherited) methods in the class; however, the methods of nested and anonymous classes are not included.
NLPM	Number of Local Public Methods	Number of local (i.e. not inherited) public methods in the class; however, the methods of nested and anonymous classes are not included.
NLS	Number of Local Setters	Number of local (i.e. not inherited) setter methods in the class; however, the setter methods of its nested and anonymous classes are not included. Methods that override abstract methods are not counted.
NM	Number of Methods	Number of methods in the class, including the inherited ones; however, the methods of its nested and anonymous classes are not included. Methods that override abstract methods are not counted.
NOA	Number of Ancestors	Number of classes and interfaces from which the class is directly or indirectly inherited.
NOC	Number of Children	Number of classes and interfaces which are directly derived from the class.
NOD	Number of Descendants	Number of classes and interfaces which are directly or indirectly derived from the class.
NOI	Number of Outgoing Invocations	Number of directly called methods of other classes, including method invocations from attribute initializations. If a method is invoked several times, it is counted only once.
NOP	Number of Parents	Number of classes and interfaces from which the class is directly inherited.
NOS	Number of Statements	Number of statements in the class; however, the statements of its nested and anonymous classes are not included.
NPM	Number of Public Methods	Number of public methods in the class, including the inherited ones; however, the public methods of nested and anonymous classes are not included. Methods that override abstract methods are not counted.
NS	Number of Setters	Number of setter methods in the class, including the inherited ones; however, the setter methods of its nested and anonymous classes are not included. Methods that override abstract methods are not counted.
PDA	Public Documented API	The number of the documented public methods of the class (+1 if the class itself is documented). When calculating the metrics the nested, anonymous or local classes found in the class and their methods are not calculated.
PUA	Public Undocumented API	The number of the undocumented public methods of the class (+1 if the class itself is undocumented). When calculating the metrics the nested, anonymous or local classes to be found in the class and their methods are not calculated.
RFC	Response set For Class	Number of local (i.e. not inherited) methods in the class (NLM) plus the number of directly invoked other methods by its methods or attribute initializations (NOI).
TCLOC	Total Comment Lines of Code	Number of comment and documentation code lines of the class, including its local methods and attributes, as well as its nested and anonymous classes.

Continued on next page

Table A2 – Continued from previous page

Abbr.	Name	Description
TLLOC	Total Logical Lines of Code	Number of non-empty and non-comment code lines of the class, including the non-empty and non-comment code lines of its local methods, anonymous and nested classes.
TLOC	Total Lines of Code	Number of code lines of the class, including empty and comment lines, as well as its local methods, anonymous and nested classes.
TNG	Total Number of Getters	Number of getter methods in the class, including the inherited ones, as well as the inherited and local getter methods of its nested and anonymous classes.
TNLG	Total Number of Local Getters	Number of local (i.e. not inherited) getter methods in the class, including the local getter methods of its nested and anonymous classes.
TNLM	Total Number of Local Methods	Number of local (i.e. not inherited) methods in the class, including the local methods of its nested and anonymous classes.
TNLPM	Total Number of Local Public Methods	Number of local (i.e. not inherited) public methods in the class, including the local methods of its nested and anonymous classes.
TNLS	Total Number of Local Setters	Number of local (i.e. not inherited) setter methods in the class, including the local setter methods of its nested and anonymous classes.
TNM	Total Number of Methods	Number of methods in the class, including the inherited ones, as well as the inherited and local methods of its nested and anonymous classes. Methods that override abstract methods are not counted.
TNOS	Total Number of Statements	Number of statements in the class, including the statements of its nested and anonymous classes.
TNPM	Total Number of Public Methods	Number of public methods in the class, including the inherited ones, as well as the inherited and local public methods of its nested and anonymous classes.
TNS	Total Number of Setters	Number of setter methods in the class, including the inherited ones, as well as the inherited and local setter methods of its nested and anonymous classes.
WMC	Weighted Methods per Class	Complexity of the class expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe's Cyclomatic Complexity (McCC) values of its local methods.

Method Metrics

CC	Clone Coverage	Ratio of code covered by code duplications in the source code element to the size of the source code element, expressed in terms of the number of syntactic entities (statements, expressions, etc.).
CLOC	Comment Lines of Code	Number of comment and documentation code lines of the method; however, its anonymous classes are not included.
DLOC	Documentation Lines of Code	Number of documentation code lines of the method.
LLDC	Logical Lines of Duplicated Code	Number of logical code lines (non-empty, non-comment lines) covered by code duplications in the source code element.
LLOC	Logical Lines of Code	Number of non-empty and non-comment code lines of the method; however, its anonymous classes are not included.
LOC	Lines of Code	Number of code lines of the method, including empty and comment lines; however, its anonymous classes are not included.

Continued on next page

Table A2 – Continued from previous page

Abbr.	Name	Description
McCC	McCabe's Cyclomatic Complexity	Complexity of the method expressed as the number of independent control flow paths in it. It represents a lower bound for the number of possible execution paths in the source code and at the same time it is an upper bound for the minimum number of test cases needed for achieving full branch test coverage. The value of the metric is initially 1 which increases by 1 for each occurence of the following instructions: if, for, foreach, while, do-while, case label (label that belongs to a switch instruction), catch (handler that belongs to a try block), conditional statement (?:) and conditional access operators (?. and ?[]). Moreover, logical and (&&), logical or (||) and null coalescing (??) expressions also add to the final value because their short-circuit evalutaion can cause branching depending on the first operand. The following language elements do not increase the value: else, try, switch, default label (default label that belongs to a switch instruction), finally.
NOI	Number of Outgoing Invocations	Number of directly called methods. If a method is invoked several times, it is counted only once.
NOS	Number of Statements	Number of statements in the method; however, the statements of its anonymous classes are not included.
NUMPAR	Number of Parameters	Number of the parameters of the method. The varargs parameter counts as one.
TCLOC	Total Comment Lines of Code	Number of comment and documentation code lines of the method, including its anonymous classes.
TLLOC	Total Logical Lines of Code	Number of non-empty and non-comment code lines of the method, including the non-empty and non-comment lines of its anonymous classes.
TLOC	Total Lines of Code	Number of code lines of the method, including empty and comment lines, as well as its anonymous classes.
TNOS	Total Number of Statements	Number of statements in the method, including the statements of its anonymous classes.