Qualitative Clustering of Software Repositories Based on Software Metrics

***Abstract*—Software repositories contain a wealth of information about the aspects related to software development process. For this reason, many studies analyze software repositories using methods of data analytics with a focus on clustering. Software repository clustering has been applied in studying software ecosystems such as GitHub, defect and technical debt prediction, software remodularization. Although some interesting insights have been reported, the considered studies exhibited some limitations. The limitations are associated with the use of individual clustering methods and manifesting in the shortcomings of the obtained results. In this study, to alleviate the existing limitations we engage multiple cluster validity indices applied to multiple clustering methods and carry out consensus clustering. To our knowledge, this study is the first to apply the consensus clustering approach to analyze software repositories and one of the few to apply the consensus clustering to software metrics. Intensive experimental studies are reported for software repository metrics data consisting of a number of software repositories each described by software metrics. We revealed seven clusters of software repositories and relate them to developers' activity. It is advocated that the proposed clustering environment could be useful for facilitating the decision making process for business investors and open-source community with the help of the Gartner's hype cycle.**

***Index Terms*—empirical software engineering, clustering, analysis of software repositories**

I. Introduction

Software repositories contain a wealth of information about the aspects related to software development process. Therefore, a retrospective analysis of such software repositories can provide valuable insights into the evolution, growth, qualitative characteristics, and problems of the corresponding software development projects. The insights gained through such retrospective analysis can affect the decision-making process in a project, and improve the quality of the software system being developed.

To conduct such retrospective analysis, Munaiah et al. (2017) [1] proposed to classify software repositories as "engineered" or "not engineered", i.e., they proposed to group the repositories based on the similarity of their attributes. However, such software ecosystems as GitHub contain more than 290 million repositories and more than 87 million users, and manually labeling them is a time-consuming process as shown by Borges and Valente (2018) [2] when they focus only on 5000 repositories due to the fact that they had to label the repositories manually. For this reason, many studies use unsupervised learning techniques such as clustering to determine the similarity between software repositories and to provide ground for repository analysis.

Determining software repository similarity is an essential building phase in studying the dynamics and the evolution of such software ecosystems as GitHub [3]. For instance, Borges and Valente (2018) [2] determined similar repositories with clustering methods to understand the growth patterns of GitHub repositories' number of stars, which leads to better understanding of how people star the repositories and what this starring is attributed to.

Considering ensemble methods in machine learning, it is of interest to consider here a suite of different clustering methods providing different partitions of data and aggregate their results into a single consolidated clustering without accessing the features or methods that determined these partitions, i.e., create a consensus clustering. The solution to this problem was introduced by Strehl and Ghosh (2002) [4]. However, to our knowledge, there is no prior study applying the consensus clustering approach to analyze software repositories. We also noted that the application of consensus clustering had received comparatively little attention in the case of software metrics.

In this work, we aim to overcome the limitations (such as limited validity) of using a single clustering method with our main objectives along with key aspects of originality being the following:

- Thoughtful use of several clustering methods and building consensus results following consensus clustering as proposed by Strehl and Ghosh (2002) [4]. We discuss it as one of the key points in this study.
- A careful analysis of clustering results with the use of multiple cluster validity indices.

Our hypothesis is that it is possible to qualify a software repository by using its quantitative metrics.

To clearly present the purpose of our work, we reviewed the motivation that lead researchers to both classify and cluster software repositories. The motivation of related studies is attributed to one of the following:

- Facilitating projects' experience reuse, groups' collaboration and shared work;
- Facilitating defect and technical debt prediction;
- Recovering a semantic representation of the software design for different software systems with diverse domains, structure, and behavior;
- Removing noise from large quantities of software projects in a resource-efficient way;
- Studying the dynamics and the evolution of such software ecosystems as GitHub and reducing the amount of

information that developers need to parse in order to stay up to date with development activity in their projects.

The relevance of this study can be mostly attributed to studying the software ecosystems and identifying important indicators about development activity in software projects. In addition, we advocate that the proposed clustering could be useful to help in decision making to software investors and the open-source community.

At the experimental end, the contributions of this study are three-fold:

- We demonstrate that in our experiment repositories are always attributed to one of the developers' activity clusters;
- We describe these clusters;
- We find "prototype" vectors describing the clusters, which may be compared to software metrics describing an arbitrary repository of similar size to classify it, i.e., we can calculate the distance between the arbitrary repository's metrics vector and the prototype vectors and relate the repository to the cluster, whose prototype is the closest to the repository.

Across the study, we adhere to the standard notation. The N software repository metrics data are represented as n-dimensional vectors, say $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$. The number of clusters is denoted by c. The data were normalized using min-max normalization, and unless stated otherwise, by the "software metrics data" we mean the normalized software repository metrics data.

The work is organized as follows: Section II describes related studies. Section III first provides a brief overview of the proposed methodology to cluster software repositories and analyze the results and then details significant parts of the methodology such as clustering with application of validity indices, consensus clustering, and aggregation of the results. Section IV provides the description of the empirical study we conducted to validate our approach. Section V provide discussion and Section VI draws the conclusion.

II. Related studies

Even though our work is aimed at clustering of software repositories based on the corresponding metrics data, we briefly review the proposals to classify repositories manually and point at their shortcomings to see if the clustering used in related works corresponds to the proposed manual classifications and whether it addresses the shortcomings of manual approaches. We also note which of the manual classifications our study is related to the most. Only then we review the work on clustering software repositories and describe their limitations.

A. Existing Proposals to Manually Classify Software Repositories

Munaiah et al. (2017) [1] classified repositories as "engineered" or "not-engineered" with the intention of identifying high-quality GitHub repositories and removing noise (e.g., repositories for homework). "Engineered" is a repository that provides general-purpose utilities to users other than the owners and is similar to those of Amazon, Apache, Microsoft, and Mozilla. They manually classified 450 GitHub repositories based on this definition and two of its aspects: "organization" and "utility". The main limitation of this approach is a bias towards industry giants and their repositories under the assumption that they use the "sound software engineering practices".

Borges and Valente (2018) [2] performed the large-scale classification of application domains on GitHub based on the top 5000 public repositories by the number of stars. The top-3 domains by the number of projects were web libraries and frameworks, non-web libraries and frameworks, and software tools – they can be viewed as meta-projects used to implement other projects. Borges and Valente (2018) [2] hypothesized that the application domain is useful for understanding the clusters of GitHub repositories based on number of stars history over time. However, they noticed that there is no statistical difference between the number of stars of systems software, applications, web libraries and frameworks, and documentation, so under the perspective of clustering using number of stars evolution, these domains can be grouped into one, resulting into three-domain repository classification.

Altogether, the limitation of the domain-based classification is that several domains can be grouped together into one or divided into a number of smaller domains, therefore, such grouping is subjective and it is hard to precisely determine the number of separate domains in advance.

Treude et al. (2018) [5] aimed at manually identifying "unusual events" in 200 randomly sampled GitHub projects with at least 500 commits and at least 100 pull requests or 100 issues based on projects' repository metrics. Such unusual events and the association between events and metrics were determined via a survey of 140 software developers responsible for or affected by these unusual events. The repositories were classified as having unusually large or small values (based on the "extreme outlier" definition) in these types of events:

- Commit-related events – the most useful metrics to detect these events according to the survey with developers are "the number of lines of code modified/deleted/added in a commit";
- Issue-related events – with useful metrics for detection being "days between open and closed" and "number of comments for label";
- Pull request-related events – detected by "number of comments".

Treude et al. (2018) [5] claimed that this classification can "help prevent potential problems early on, encourage discussion where it is needed, and give important pointers to events in a project's history to be reviewed". According to the findings of Treude et al. (2018) [5] , the limitation is that awareness tools based on commit or source code activity alone are not sufficient to communicate all the information developers care about in a project, which means that additional data related to issues and pull-requests have to be collected and analyzed.

Out of the proposed approaches to classify software repositories, our study corresponds more to the one by Treude et al. (2018) [5] , as we try to identify the metrics that would characterize software repositories qualitatively and signal about potential anomalies. To determine a set of metrics that are useful for anomaly detection, Treude et al. (2018) [5] rely on the opinion of software developers who caused or deal with these anomalies, however, they found out that developers "value simple and easily understandable metrics over complex ones", though simple and easily understandable metrics might not be enough to uncover the underlying data patterns. To address this, we conduct an automatic clustering of the repositories based on a set of GitHub metrics instead of relying on the simple "extreme outlier" definition of anomaly, and then try to identify useful metrics that indicate something unusual in the resulting clusters of repositories.

B. Clustering Software Repositories

There has been active research on clustering software repositories and corresponding software projects even if such clustering has not been always connected to the classifications mentioned in Section 2.1. We review the most relevant work in the field and present our review in the following paragraphs.

Kawaguchi et al. (2006) [6] proposed a tool that automatically categorizes software systems for identifying similar projects to facilitate reuse and sharing knowledge among software projects. As metrics for clustering, they used identifiers uncovered by latent semantic analysis from source code, and then they applied cosine similarity and unifiable cluster map to automatically categorize software repositories. Applied on 41 programs in C in five categories coming from SourceForge, their method generated 40 clusters with mostly 2-3 software systems per cluster. The limitation of such clustering is that it produces too element-specific, not descriptive, and small clusters.

Jureczko and Madeyski (2010) [7] used such clustering algorithms as hierarchical, K-means, and Kohonen's neural network to identify groups of software projects with similar characteristic from the defect prediction point of view. They measured Chidamber and Kemerer, Bansiy and Davis metrics suites, lack of cohesion in methods (LCOM3), and McCabe's cyclomatic complexity on 92 versions of 38 proprietary, open-source and academic projects, and used these metrics for clustering. The existence of two clusters was proven with statistical testing: custom-built solutions and text processing projects by medium sized international team. The limitation of the proposed clustering method is that it requires the information about defects to be predicted, therefore, it is hard to externally validate it.

Borges and Valente (2018) [2] studied GitHub repository starring practices and meaning, characteristics, and dynamic growth of GitHub stars. They used K-Spectral Centroid algorithm on top 5000 public GitHub repositories by the number of stars in order to cluster the repositories and analyze the stars growth patterns. They identified three clusters that suggest a linear growth at different speeds and one cluster with a sudden growth in the number of stars. This classification study has a limited external validity of the method as authors focus only on the characteristics of the most starred GitHub projects.

Saied et al. (2018) [8] used DBSCAN-based hierarchical clustering on 6638 libraries and 38000 client systems hosted in repositories from GitHub to group the libraries that are most frequently co-used together by clients in order to relieve developers from manual analysis. To cluster libraries, they applied usage vector for each library indicating which client systems use the library. The obtained usage patterns exhibited high usage cohesion with an average of 77%. The study has the limitation in the method's external validation – due to the data collection used, there could be duplicates and missed dependencies, which might mislead the clustering algorithm and provide biased results.

Pickerill et al. (2020) [9] clustered 1,786,601 GitHub repositories into "engineered" and "not engineered", following the classification by Munaiah et al. (2017) [1]. They extracted information about repositories' Git commit, integration, committer, integrator, and merge frequency, and, using these metrics, applied K-means clustering algorithm, which resulted in 38% of the repositories being labeled as "well-engineered". The main limitation of such clustering is limited external validity, since the authors validated the clustering only on a small manually labeled subset of the 1,786,601 repositories, and, as they noted, there is a possibility that both the ground truth and clustering are wrong, and their agreement is coincidental.

Rokon et al. (2021) [3] claimed that determining repository similarity is an essential building block in studying the dynamics and the evolution of such software ecosystems as GitHub, and for this purpose they proposed an embedding of repository metadata, code, and structure, which they then used to cluster 1013 GitHub repositories with hierarchical clustering. This resulted into three clusters of benign, malware, and REST API related repositories, or 26 sub clusters of the three. This clustering has limited internal validity as authors used only one cluster validity index.

Tsoukalas et al. (2021) [10] divided 27 software projects from the technical debt dataset [11] into six clusters of similar projects with respect to their technical debt aspects using K-means algorithm and built specific technical debt forecasting models for each cluster using regression algorithms. As metrics for clustering, they used effort in minutes to fix code smells, bugs, vulnerability issues and number of lines of code, bugs, smells, as well as cyclomatic complexity. The results showed that the prediction errors tend to be statistically significantly lower in within-cluster technical debt forecasting than in cross-cluster forecasting. The limitation of this study is that the authors did not provide the interpretation for the resulting six clusters, which limits the external validity of the clusters.

Xu et al. (2021) [12] performed a large-scale empirical study with 40 clustering algorithms on 27 versions of 14 open-source projects to explore the impacts of clustering-based models on defect prediction performance. They used code complexity, process, and network metrics and compared

the clustering-based models to supervised defect prediction models. The results showed that not all clustering models are worse than supervised models, however, authors did not provide an analysis or interpretation of the resulting clusters, which limits the external validity of such clustering.

Tan et al. (2022) [13] focused on hierarchical clustering and Bunch clustering algorithms for remodularization and provided information about their suitability according to the features of the software repositories such as bugs, code smells, duplications, number of lines of code, size, and number of stars. The resulting clusters were described in terms of how well the proposed clustering algorithms performed according to the MoJoFM metrics, however no cluster interpretation was provided. For validation, authors took top 30 GitHub repositories by the number of stars written in Java with at least 10 commits, which limited external validity of the resulting clusters.

The conclusions from the literature review are as follows. We did not identify a prior study that would analyse open-source software repositories from the perspective of developers' community interest, however we believe that this is an important aspect since the knowledge about this interest would help to better understand not only the state of the repository itself but also of the correspond software development project. In addition, the proposed studies have limitations in terms of validity of the clustering results since most of them use a single clustering technique.

In this work, we try to overcome the limitation of cluster validity by using multiple clustering techniques, cluster validity indices, and consensus clustering. It is also worth noting that the application of consensus clustering to the software metrics case had received rather little attention. We were able to identify only few of the studies considering the application of consensus clustering such as those by Coelho et al. (2014) [14] and Puchala et al. (2022) [15]. In addition, we were unable to identify a study that would apply consensus clustering to analyze software repository metrics. In this study, we also address that.

III. Methodology

In order to avoid size or popularity based clustering, we narrow down the set of considered software repositories to approximately the same size and popularity repositories. Then, we divide this set into subsets and conduct clustering in each of the subsets, aggregate clustering results across the subsets, compare them, and make a conclusion about how consistent our clustering was across the subsets of software repositories. The division to subsets allows us to validate our clustering in the same way as if we had multiple datasets and clustered each of them.

Our methodology of identifying software repository clusters consists of three steps. The first step is data collection and preparation. We consider N repositories on GitHub. We separate these N repositories into $P < N$ sets of $\frac{N}{P}$ repositories. In addition, we predefine the fixed number of consensus clusters c based on the relevant literature and consensus clustering performance index.

The next step is processing of the repository sets. We pick the first $\frac{N}{P}$ repositories, collect n metrics from them and apply the following algorithm:

1) We normalize each metric to the [0,1] range.

2) We group repositories into c clusters based on the n metrics using m clustering algorithms with different values of parameters that we tune.

3) For each grouping resulting from the previous step, we calculate three cluster validity indices and obtain at most three best groupings (in the sense that each groping optimizes a certain validity index) for each algorithm. Refer to Section III-A for details.

4) We use the consensus clustering approach in order to build a new "winner" grouping from the best ones (refer to Section III-C for details).

5) For each group in the winner grouping, we calculate prototype vector of n metrics that is an average vector in the group.

We repeat the entire algorithm for the next $\frac{N}{P}$ repositories, and so on. As a result, we get a collection of c sets of prototype vectors.

The final step of our methodology is aggregation of the results. We match the calculated cluster prototypes to compare them and the clusters they represent across P sets of clustered repositories. For that, we consider the minimum-weight matching problem in P-partite graph across all $c \cdot P$ prototypes (for details refer to Section III-D), and, as a result, get c sets of P matched prototypes. We calculate the discrepancy in every such set of prototype vectors and, using a similarity measure, calculate the discrepancy value d for each of the c sets, which shows us how different the cluster prototypes are across the P sets of data for each of the c clusters.

A. Clustering Data

To get the initial groupings of repositories, from which we will take the "best" ones, we apply multiple algorithms to avoid being subject to specifics of the math of a single clustering algorithm. We use partitioning-based K-means as one of the generic clustering algorithms (it is applied in many related studies for its simplicity), density-based DBSCAN as more advanced algorithm that allows to identify clusters of arbitrary shapes and handle "noise" in the data and is widely used, and graph-based spectral clustering that can be sought as an intermediate "link" between the two, since it is connected to K-means and to DBSCAN as noted by Dhillon et al. (2004) [16] and Schubert et al. (2018) [17]. The algorithms, their parameters that we use to tune them and their output are presented in Table I.

To internally validate the clustering, we measure three cluster validity indices on the clustering results produced by m clustering algorithms run on the metrics data. We use the predefined range of values for such parameters as number of clusters for K-means and Spectral, and ε for DBSCAN. For measuring the indices, we use the Euclidean distance. Since

TABLE I
CLUSTERING ALGORITHMS USED IN THE STUDY.

Clustering algorithm	Parameter for tuning	Algorithm's output
K-Means	Number of clusters k	Partition matrix $\boldsymbol{W}$, where each element w_{ik} indicates whether a repository metrics vector $\boldsymbol{x}_i$ belongs to cluster k, and the cluster prototypes (centroids) $\boldsymbol{\mu}_k$
Spectral clustering	Number of clusters k	Partition matrix $\boldsymbol{W}$ and affinity matrix $\boldsymbol{A}$ constructed on software repository metrics vectors $\boldsymbol{x}_i$ [18]
DBSCAN	Neighborhood radius ε	Partition matrix $\boldsymbol{W}$, where noise vectors are given label '-1', and indices j of the core vectors $\boldsymbol{x}_j$ [19]

multiple validity indices can show different optimal solutions in terms of the parameter that we tune (e.g., number of clusters for K-means) for the same algorithm, we consider one unique solution per validity index, i.e., if two out of three indices showed the same solution, and the third one showed a new solution, we would consider the two unique solutions; if all three showed the same solution or different solutions, we would consider only one or three unique solutions correspondingly. Due to this reason, on the output of the initial clustering stage, we have $r \geq m$ cluster labelings – one labeling per each unique optimal solution demonstrated by the validity indices.

B. Cluster Validity Indices

We apply Silhouette, Calinski-Harabasz, and Davies-Bouldin indices as they are popular in the clustering literature. The following paragraphs briefly describe them.

For each clustered sample $\boldsymbol{x}_i$ of the software repository metrics data, the silhouette coefficient is calculated using the mean intra-cluster distance and the mean nearest-cluster distance (between a sample and the nearest cluster that the sample is not a part of). Kaufman and Rousseeuw (2009) [20] introduced the term silhouette coefficient for the maximum value of the mean silhouette over the entire dataset for a specific number of clusters c. To find the optimal clustering result, we look for the maximum value of the Silhouette index across the range of considered parameter values and mark a value of a parameter optimal if it corresponds to the maximum value of Silhouette.

Calinski-Harabasz index [21] measures the similarity of a software repository to other repositories in its cluster (cohesion) as compared to other clusters (separation). Cohesion is estimated based on the distances from the repositories in a cluster to its cluster centroid and separation is based on the distance of the cluster centroids from the global centroid. To identify the optimal clustering result, we look for the maximum value of Calinski-Harabasz index across the range of considered parameter values.

Davies-Bouldin index [22] estimates the cohesion based on the distance from the software repositories in a cluster to its centroid and the separation based on the distance between cluster centroids. To find the optimal clustering result, we look for the minimum value of Davies-Bouldin index across the range of considered parameter values and mark a value of a parameter optimal if it corresponds to the minimum value of Davies-Bouldin.

The values of parameters corresponding to the optimal values of validity indices are saved and then used on the clustering algorithms to cluster the software repository metrics data $\boldsymbol{x}_i$.This clustering then produces the initial optimal cluster labelings λ_q coming from the partition matrices produced by the algorithms, where $q = 1, \ldots, r$, and r is the number of the observed optimal values of parameters across all algorithms (one algorithm can produce up to three optimal cluster labelings, as we use three validity indices).

C. Consensus Clustering

Consensus clustering (also called cluster ensembles) provides improved quality of solution and robust clustering as compared to using a single clustering method [23]. The main idea of the consensus clustering approach is to transform the set of individual cluster labelings $\boldsymbol{\Lambda} = \{\lambda_q | q \in 1, \ldots, r\}$ (refer to Section III-A) into a single consensus labeling Λ that separates the software repository metrics data into clusters of software repositories using the consensus function:

$$\Gamma : \boldsymbol{\Lambda} \rightarrow \lambda. \tag{1}$$

In this study, we focus on the graph and hypergraph based as well as non-negative matrix factorization based consensus clustering methods as they are the most popular among the studies related to consensus clustering and are easy to understand and implement [24]. We use the consensus clustering methods (each provides a different consensus function) proposed by Strehl and Ghosh (2002) [4], Fern and Brodley (2004) [25], and Li et al. (2007) [26], namely Cluster-based similarity partitioning algorithm, Hypergraph-partitioning algorithm, Meta-clustering algorithm, Hybrid bipartite graph formulation, and Nonnegative matrix factorization, to avoid being subject to the specifics of one consensus function's math, and choose the one that maximizes the average mutual information:

$$\lambda = arg \max_{\hat{\lambda}} \frac{1}{r} \sum_{q=1}^{r} \phi(\hat{\lambda}, \lambda_q), \tag{2}$$

where $\phi(\hat{\lambda}, \lambda_q)$ is the normalized mutual information between labelings $\hat{\lambda}, \lambda_q$ and is defined as proposed by Strehl and Ghosh (2002) [4]. The following paragraphs describe the consensus clustering algorithms used.

The idea of Cluster-based similarity partitioning algorithm is to create a square matrix $\boldsymbol{S}$, where each element s_{ij} denotes the fraction of clusterings in which two software repository

metrics vectors $\boldsymbol{x}_i, \boldsymbol{x}_j$ are in the same cluster. Next, the similarity matrix is used to redo the clustering of the repository metrics vectors using a similarity-based clustering algorithm, e.g., partitioning of the induced similarity graph (vertex = repository, edge weight = similarity).

In Hypergraph-partitioning algorithm, the consensus clustering problem is formulated as partitioning the hypergraph by cutting a minimal number of hyperedges. All hyperedges are considered to have the same weight, and all vertices are equally weighted. The algorithm looks for a hyperedge separator that partitions the hypergraph into c unconnected components of approximately the same size.

Meta-clustering algorithm is based on clustering clusters of software repositories. Each cluster is represented by a hyperedge. The idea is to group and collapse related hyperedges and assign each repository metrics vector $\boldsymbol{x}_i$ to the collapsed hyperedge in which it participates most strongly. The hyperedges that are considered related for the purpose of collapsing are determined by a graph-based clustering of hyperedges. Each cluster of hyperedges is referred to as a meta-cluster. Collapsing reduces the number of hyperedges from $\sum_{q=1}^{r} c_q$ to c.

Hybrid bipartite graph formulation constructs a graph, where vertices represent software repository clusters or the repository metrics vectors $\boldsymbol{x}_i$. In this graph, cluster vertices are only connected to repository vertices and vice versa, forming a bipartite graph. The algorithm partitions the cluster vertices and the repository vertices of the bipartite graph simultaneously. The partition of the repositories is then outputted as the final clustering.

Nonnegative matrix factorization defines the consensus clustering as the median partition problem, fixing the following distance as a measure of likeness between partitions:

$$l(\lambda, \lambda') = \sum_{i,j=1}^{N} l_{ij}(\lambda, \lambda'), \tag{3}$$

where $l_{ij}(\lambda, \lambda') = 1$ if $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ belong to the same cluster in one partition λ and belong to different clusters in the other partition λ', otherwise $l_{ij}(\lambda, \lambda') = 0$. The consensus partition is defined by the median partition problem using l as a dissimilarity measure between partitions.

We use the average normalized mutual information to assess the quality of obtained consensus clustering – we take the consensus that has the largest value of the mutual information and, therefore, the highest quality of obtained clustering.

D. Matching Cluster Prototypes and Discrepancy Calculation

At the final step of methodology described in the beginning of Section III, we match the prototypes solving the minimum-weight matching problem in P-partite graph $G = (V, E)$, where vertices $V = \{v_1, v_2, \dots\}$ are prototypes produced for P sets of clustered repositories, edges $E = \{e_1, e_2, \dots\}$ represent and are weighted by the Euclidean distance between prototypes of different repository sets, and each p-th partition represents the set of c prototypes for the p-th repository set; $p = 1, \dots, P$.

We need to find a set of P vertices for each of the c clusters. To solve this problem, we consider all possible combinations of vertices (prototype vectors) from different partitions (P repository sets) and calculate the cost for each combination as the sum of all pairwise distances between prototypes in that combination. The formula for the cost of one combination of i-th prototype of the first repository set, k-th prototype of the second set, …, h-th prototype of the $(P-1)$-th set, and j-th prototype of the P-th set is the following:

$$C_{\underbrace{ik\dots hj}_{P}} = (\Delta_{ik} + \dots + \Delta_{ih} + \Delta_{ij}) + (\dots + \Delta_{kh} + \Delta_{kj}) + \dots + (\Delta_{hj}), \tag{4}$$

where Δ_{ik} is the Euclidean distance between i-th prototype of the corresponding first set and k-th prototype of the corresponding second set. So, we reformulate the problem as the P-index linear assignment problem [27], where $X_{ik\dots j}$ is the solution to the problem and equals 1 if a combination of prototypes (vertices) $i, k, \dots, j$ is matching with the minimum weight, otherwise the value is 0. Therefore, we have the following problem:

$$\min \underbrace{\sum_{i=1}^{c}\sum_{k=1}^{c}\cdots\sum_{j=1}^{c}}_{P} C_{ik\dots j} X_{ik\dots j} \tag{5}$$

subject to:

$$\sum_{i=1}^{c}\sum_{k=1}^{c}\dots X_{ik\dots j} = 1 \text{ for all } j \in \{1, \dots, c\},$$

$$\dots$$

$$\sum_{i=1}^{c}\cdots\sum_{j=1}^{c} X_{ik\dots j} = 1 \text{ for all } k \in \{1, \dots, c\},$$

$$\sum_{k=1}^{c}\cdots\sum_{j=1}^{c} X_{ik\dots j} = 1 \text{ for all } i \in \{1, \dots, c\},$$

$X_{ik\dots j} \in \{0, 1\}$ for all $\{i, k, \dots, j\} \in \{1, \dots, c\}^P$, where $\{1, \dots, c\}^P$ are disjoint sets of corresponding prototypes of P repository sets. To solve this problem, we use mixed integer linear programming [28].

To make a conclusion about how similar the clusters are across P sets of repositories, we calculate the pairwise discrepancy value d for every set $c_i; i = 1, \dots, c$; consisting of P matched prototypes vectors. For that, we use cosine distance, since it is based on cosine similarity and we want to measure how similar (or dissimilar) the prototype vectors are. However, to avoid the cosine's distance sensitivity to the mean, we use the adjusted cosine distance:

$$d_{kt} = 1 - \frac{\sum_{i=1}^{n}(\mu_{ik} - \bar{\boldsymbol{\mu}_k})(\mu_{it} - \bar{\boldsymbol{\mu}_t})}{\sqrt{\sum_{i=1}^{n}(\mu_{ik} - \bar{\boldsymbol{\mu}_k})^2}\sqrt{\sum_{i=1}^{n}(\mu_{it} - \bar{\boldsymbol{\mu}_t})^2}}, \tag{6}$$

where μ_{ik} are the elements and $\bar{\boldsymbol{\mu}_k}$ is the mean of the prototype vector $\boldsymbol{\mu}_k$, and d_{kt} measures the adjusted cosine distance between a pair of matched prototypes $\boldsymbol{\mu}_k$ and $\boldsymbol{\mu}_t$

and is bounded in range $[0, 2]$. After measuring the pairwise discrepancy d_{kt} for a set c_i of prototype vectors, we get a single discrepancy value for this set:

$$d_{c_i} = \max_{\boldsymbol{\mu}_k, \boldsymbol{\mu}_t \in c_i} d_{kt}. \quad (7)$$

In the end, we will have c discrepancy values d_{c_i}, each corresponding to one of the c clusters. To make sure that we have a meaningful discrepancy values, we repeat the whole algorithm described in Section III on the random data of the same size $N \times n$ and compare the results with the values achieved in the real data.

IV. Experimental Studies

A. Data Collection and Preparation

The experimental study was conducted on $N = 1659$ GitHub project repositories with $n = 28$ metrics. This number of metrics was aggregated from the time series of metrics related to commits (lines added, lines deleted, lines changed, files changed, number of commits) and issues (number of opened and closed), taking their values on the latest date of time series, and additionally aggregating (summing) them across the three time periods: one month prior to the latest date, two week prior, and the whole history recorded in the time series. Time series were collected for each of the 1659 repositories. The latest date in the time series was the fifth of May 2022.

The criteria for repository search were the following: number of stars being in range $(100, 200)$, number of forks – in range $(50, 150)$, size being less than 2100 kilobytes. The number of repository subsets is $P = 7$, which gives us the same number of 237 repositories in each subset and provides us with an acceptable ratio of the number of repositories to the number of software metrics.

Following the example of Munaiah et al. (2017) [1] and Pickerill et al. (2020) [9], we initially took the number of clusters $c = 2$.

B. Processing of the Repository Sets

First, we normalized values of metrics in each repository set using min-max normalization. In each set of repositories we have identified optimal groupings using the three clustering algorithms and the three validity indices. As a metric for all of the algorithms and validity indices we used Euclidean distance. For DBSCAN, we took the minimum number of samples 5 [19].

The consensus clustering algorithms, and their achieved values of the objective function for each of the P repository sets are presented in Table II. The highest value of the performance index is underlined for each of the repository sets. The algorithms that correspond to the underlined values were used to get a single consensus clustering in each corresponding set of repositories. After calculating a single clustering for each of the repository sets, we calculated the prototypes according to our methodology. They were normalized for better visualization and are presented in Figure 2.

C. Aggregation of the Results

We matched the prototypes from different sets of P clustered repositories using mixed integer programming solver "Coin-or branch and cut" [29]. We received the following discrepancy values for the two sets of prototypes c_1 and c_2:

$$d_{c_1} \approx 0.461; d_{c_2} \approx 1.414.$$

We repeated the whole process for randomly generated data and got the following discrepancy values:

$$d^{rand}_{c_1} \approx 1.237; d^{rand}_{c_2} \approx 1.194.$$

We have aggregated the matched cluster prototypes from different repository sets by taking the mean of the matched prototypes for each cluster – the result is presented in Figure 1 (where the prototypes are normalized between each others for better visualization) – the metrics on the radar plots are numbered following the next order: issues, then commits metrics – full history (1-7 on the radar plots), past month (8-14), past two weeks (15-21), the latest date (22-28). Compared to the results generated on random data, the discrepancy for c_1 shows relatively consistent result in terms of cosine distance between the cluster prototypes. After inspecting the resulting matched cluster prototypes, we conclude that cluster c_1 represents repositories with low activity, probably abandoned by the developers. However, for the c_2, the discrepancy value shows low consistency across prototypes as shown by high maximum value of cosine distance between matching prototypes d_{c_2}. We believe this indicates the presence of more than one subcluster inside the cluster c_2, and for this reason we repeat the clustering, however, this time without the division to P subsets as in c_2 we have only 133 repositories.

D. Division of the second cluster to subclusters

We take the number of clusters and consensus function that maximize the average normalized mutual information (ANMI). We have achieved the best consensus function with the meta-clustering algorithm and the number of clusters $c = 6$ according to the value of ANMI of approximately 1.223. The normalized (for ease of analysis and better visualization) consensus cluster prototypes are presented in Figure 2 – the metrics are numbered following the same order as in Figure 1. The majority of the 133 "active" repositories were assigned to the cluster c_{21}, and the rest five clusters contained either one repository (clusters c_{26}, c_{25}, c_{23}), two (cluster c_{24}), or three (cluster c_{22}).

For the sake of repository analysis, we inspect the cluster prototypes and mark the metrics as "high" or "low" if they correspond to the value of 1 or 0 on Figure 2 correspondingly for each cluster. However, we cannot objectively make a conclusion on a value of a metric in the range $(0, 1)$, therefore, here we only mark the metrics having values of either 0 or 1. Cluster c_{21} prototype represents a repository with low values of total created and total (aggregated across the whole history in the dataset) closed issues, total added and removed commits per day, total files changed with commits per day, and all

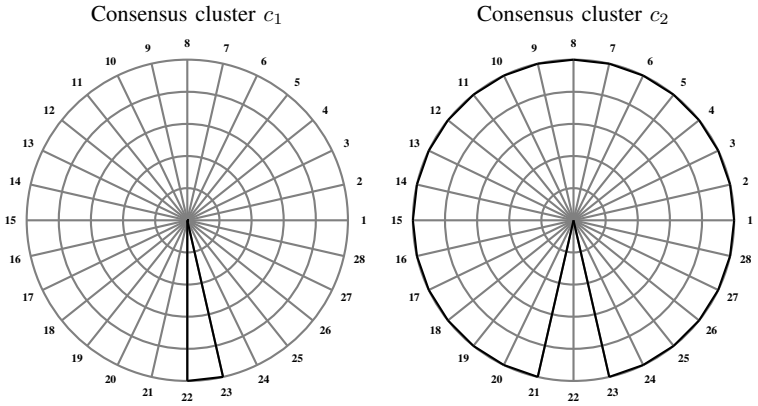

Fig. 1. Preliminary cluster prototypes.

TABLE II
VALUES OF OBJECTIVE FUNCTION ON TWO CLUSTERS FOR DIFFERENT CONSENSUS CLUSTERING ALGORITHMS. THE HIGHEST VALUE OF THE OBJECTIVE FUNCTION IS UNDERLINED FOR EACH OF THE REPOSITORY SETS.

Consensus clustering algorithm	Objective function: Average normalized mutual information						
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7
Hypergraph-partitioning algorithm	0.093	0.091	0.306	0.441	0.802	0.424	1.073
Meta-clustering algorithm	<u>0.96</u>	<u>0.996</u>	<u>1.07</u>	<u>0.522</u>	0.561	<u>0.544</u>	<u>1.143</u>
Hybrid bipartite graph formulation	0.353	0.098	0.807	0.116	0.198	0.097	0.415
Cluster-based similarity partitioning algorithm	0.095	0.093	0.058	0.111	<u>0.847</u>	0.092	0.784
Nonnegative matrix factorization	0.179	0.811	0.243	0.271	0.451	0.042	1.038

metrics aggregated for the rest of the time periods (month, two weeks, and the latest date). Cluster c_{22} prototype has low values of all metrics at the latest date. Cluster c_{23} prototype has high values of open and closed issues for the past month and a low value of closed issues on the latest date. Cluster c_{24} prototype has high values of total created and closed issues, overall number of commits and files changed, number of closed issues for past month, closed issues for the past two weeks and low values of metrics on the latest date. Cluster c_{25} has high values of the metrics on the latest date, and low values of overall number of commits and removed lines. Cluster c_{26} has high values of lines changed for all periods and commits for past two weeks, and low values of the metrics on the latest date.

To validate our clustering, we manually labeled the "active" 133 repositories according to our identified clusters using expert judgement. Then we took the manually labeled (according to the subclusters $c_{21} - c_{26}$) 133 repositories (from the cluster c_2) and automatically labeled 1526 repositories (from the cluster c_1) and reconstructed the initial dataset with 1659 repositories, however now with labels. We used these labels to conduct the cross-validation using random forest classifier (since it lowers risk of overfitting) [30] and to estimate the accuracy of the classification. The achieved mean accuracy with shuffling and splitting 10 times and train to test ratio being 70/30 is 0.92 with standard deviation of 0.011.

V. DISCUSSION

Judging by the prototypes, we conclude that the cluster c_{21} represents repositories with relatively small activity, while other clusters represent different types of activity in repositories, such as sudden peak of developing activity lately (cluster c_{22}, where the metrics for the past two weeks have noticeably larger values than for the past month); fresh repository, where developers' activity is decreasing (cluster c_{23}, where the number of commits and closed issues for the latest date and the past two weeks is decreased as compared to the past month); repository on the maintenance stage (cluster c_{24}, where the number of closed issues is the highest among the prototypes across all time periods except for the latest date); fresh, steadily developed repository (cluster c_{25}, where the activity for the past two weeks is higher than for the past month, and all metrics are the highest among the prototypes on the latest date); repository at a peak of developers' activity (cluster c_{26}, where the number of lines of code added and deleted is the highest among the prototypes for the past two weeks, month, and the whole history, and there is a relatively high number of commits for the past month and two weeks). We were able to infer this interpretation by inspecting not only the prototypes, but also the repositories inside the clusters. For the different activity types clusters, we inspected the following repositories:

Cluster c_{22} https://github.com/Vonage/vonage-python-sdk, https://github.com/elastic/ecs-logging-java, https://github.com/eclipse/tahu;

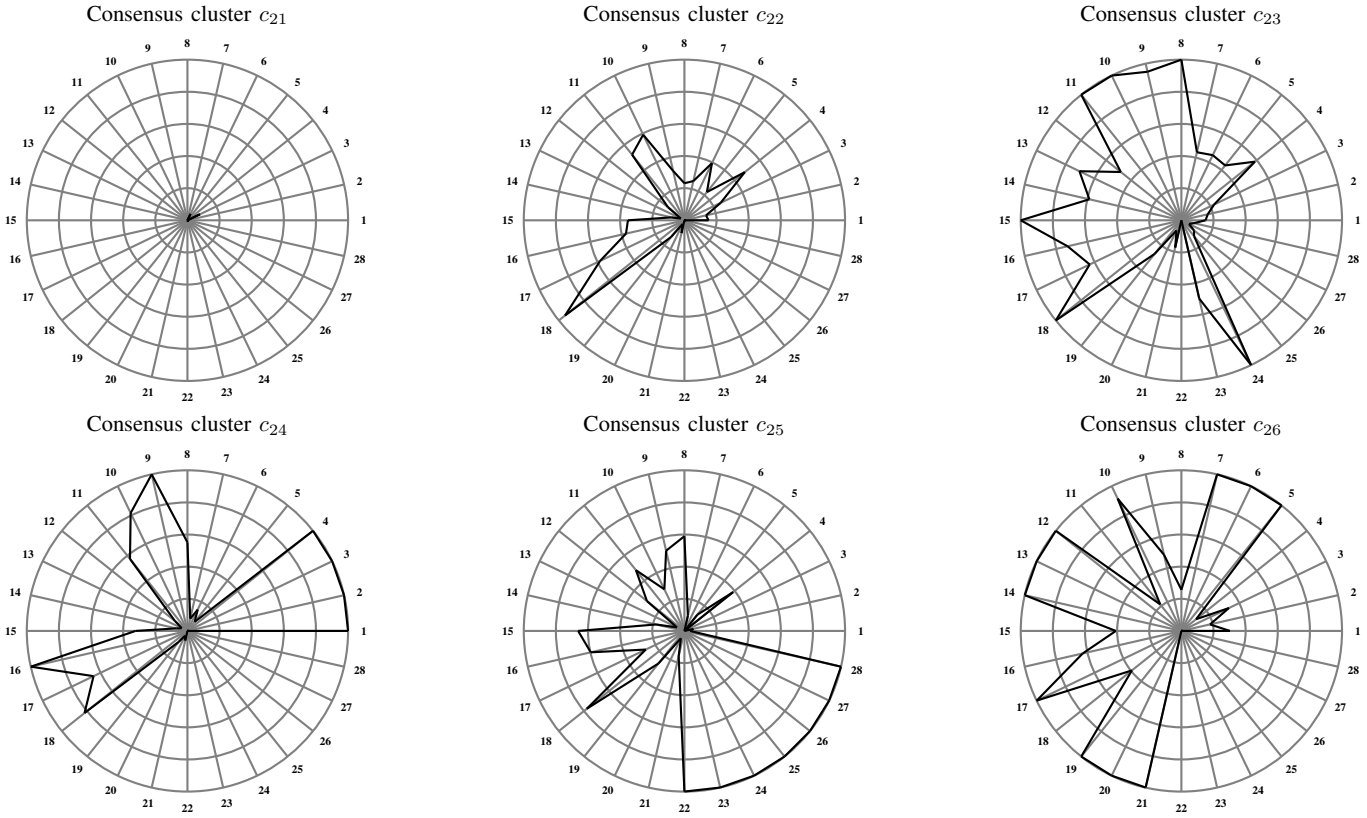

Fig. 2. Prototypes for subclusters of c_2.

Cluster c_{23} https://github.com/Tencent/Firestorm;
Cluster c_{24} https://github.com/kernitus/BukkitOldCombatMechanics, https://github.com/GoogleCloudPlatform/cloud-sql-jdbc-socket-factory/graphs/contributors;
Cluster c_{25} https://github.com/apache/flink-kubernetes-operator;
Cluster c_{26} https://github.com/RS117/RLHD.

To summarize, we have the following stages of developers activity according to our interpretation: initial steady development, where the repository has been launched only recently and there is an intense development activity going on; active development, where the repository is past its initial development stage, and now the developers change more lines of code and do more commits; the disillusionment stage, where the developers' activity is decreasing or already dropped dramatically as compared to the other stages, which might be caused by lack of necessary technologies to support the development and the corresponding project, or the team has decided to leave the project temporary or permanently; sudden peak of activity, where the developers contribute more with each week and potentially are coming back to the left earlier project (due to the advancements in connected technologies) or they are coming from a new team that was assigned to bring the project back on track; the maintenance stage, where developers are more focused on closing the issues, which means that the project has reached its maturity and now is extensively used in the community.

We believe that these stages of development of the open-source projects can be put on a graph in a similar fashion as the Gartner hype cycle [31]. However, the Gartner use their hype cycle to describe the development of emerging technologies, while we use a similar hype cycle to describe the developers' community interest in open-source software projects and the corresponding software repositories as shown in Figure 3. On the hype cycle, we mark the position of the active repository clusters on the hype cycle. We do not put the "expectation" on the y axis as Gartner do, but we put the "community interest" there, which corresponds to how actively developers are participating in the developing of an open-source repository. On the x axis we have a representation of what we believe are the stages of development of an open-source software repository. It is worth noting that the hype cycle in Figure 3 does not present precise values of certain metrics, but is rather a visual representation of how we connect our data analysis results with business and community perspective. Such connection between data analysis results in the form of activity clusters and business perspective in the form of the hype cycle is useful for understanding the stage of maturity and interest of developers in a software repository as well as the corresponding technology development, which in turn will help the business representatives and the community to decide which projects are more profitable to invest their resources in

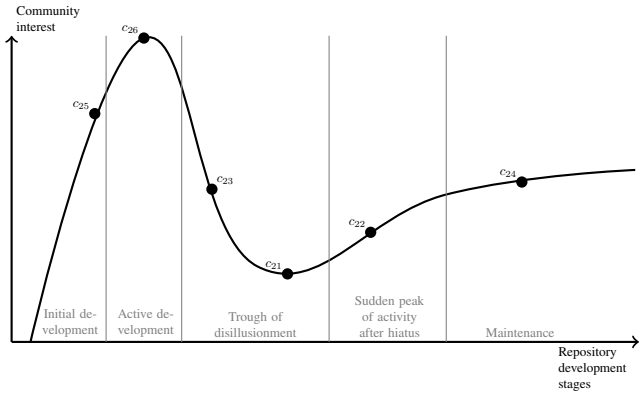

Fig. 3. Hype cycle for active software repository clusters.

and which ones should be left out of consideration.

Our results show that there are a vast number of abandoned or low developers activity repositories among the repositories with relatively high popularity/size ratio on GitHub. Our methodology allowed us to identify "outliers" among inactive repositories, which turned out to represent different types of developers' activity and connect them to the hype cycle similar to the Gartner's. We believe this demonstrates that if an open-source software repository have reached a relatively high popularity to size ratio (as shown by the number of stars and forks vs. the size in kilobytes), then in most cases this repository is already in a such maturity state, where there is relatively small amount of developers activity going on. The active repositories that we have identified are presented in such small quantity that they look more like an anomaly as compared to the majority of other maturity repositories.

In this work we used consensus clustering as related studies reported it provides improved quality of solution and robust clustering, as well as more stability with respect to random initialization compared to a single clustering method [23], [32]. In addition, we believe that using cluster ensembles is beneficial to a user, who has no proficiency in clustering-based data analysis and does not have a clear criteria on how to use a specific clustering technique, such as software managers. Consensus clustering will allow them to worry less about the specifics of math of a single clustering method and focus more on the data analysis.

A. Threats to Validity

In order to avoid threats to internal validity, we used cluster validity indices, several clustering techniques, and interpreted the resulting clusters to make sure that our clustering is caused by the meaningful effects of the chosen metrics.

In terms of external validity (i.e., the extent to which the results of a study can be generalized to other situations), we cannot generalize our findings to the development platforms other than GitHub. However, GitHub now hosts more than 200 million public repositories [33], making it a good starting point for this research.

In terms of construct validity (i.e., the degree to which a test measures what it claims, or purports, to be measuring), while our initial list of software repository metrics was designed to be useful from the perspective of changes analysis (e.g., commits added per day) and based on used and provided by GitHub API, there could be other important metrics that we did not include in the initial set of repository metrics. However, our proposed approach can be used with different set of metrics including code metrics, which could provide a deeper insight into repositories' code structure.

VI. Conclusions

The goal of our work was a) to understand if the software repositories can be grouped into clusters based on their metrics to gain dedicated practical insights b) while using consensus clustering to avoid being subject to the specifics of math of a single clustering technique. We divided the metrics data into multiple subsets and clustered each subset into two clusters, then measured the discrepancy value between the resulting clusters. By inspecting the discrepancy values, we were able to identify that there are more than two clusters inside our repository metrics dataset. We revealed seven clusters of different types of repositories in terms of developers' activity, of which the majority turned out to be either abandoned or having relatively low activity, while the rest corresponded to different types of developers activity. We mapped the results of the data analysis to the Gartner hype cycle to facilitate the link between the analysis and business perspective part.

For the future work we are considering expanding our set of metrics to include metrics related to pull requests, comments, releases, and workflows to see if our clustering based on both code and process metrics will be able to predict whether a repository is on-demand among users and popular among developers or not. This would allow us to make inference about the connection of code and process metrics of GitHub software repositories. In addition, we also plan to study the clustered repositories in detail by inspecting their history of development in terms of the software metrics and potentially modify our hype cycle into a more precise graph that would show community's interest to open-source software projects and repositories.

References

[1] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[2] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[3] M. O. F. Rokon, P. Yan, R. Islam, and M. Faloutsos, "Repo2vec: A comprehensive embedding approach for determining repository similarity," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 355–365.

[4] A. Strehl and J. Ghosh, "Cluster ensembles—a knowledge reuse framework for combining multiple partitions," *Journal of machine learning research*, vol. 3, no. Dec, pp. 583–617, 2002.

[5] C. Treude, L. Leite, and M. Aniche, "Unusual events in github repositories," *Journal of Systems and Software*, vol. 142, pp. 237–247, 2018.

[6] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.

[7] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th international conference on predictive models in software engineering*, 2010, pp. 1–10.

[8] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.

[9] P. Pickerill, H. J. Jungen, M. Ochodek, M. Maćkowiak, and M. Staron, "Phantom: Curating github for engineered software projects using time-series clustering," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2897–2929, 2020.

[10] D. Tsoukalas, M. Mathioudaki, M. Siavvas, D. Kehagias, and A. Chatzigeorgiou, "A clustering approach towards cross-project technical debt forecasting," *SN Computer Science*, vol. 2, no. 1, pp. 1–30, 2021.

[11] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019, pp. 2–11.

[12] Z. Xu, L. Li, M. Yan, J. Liu, X. Luo, J. Grundy, Y. Zhang, and X. Zhang, "A comprehensive comparative study of clustering-based unsupervised defect prediction models," *Journal of Systems and Software*, vol. 172, p. 110862, 2021.

[13] A. J. J. Tan, C. Y. Chong, and A. Aleti, "E-sc4r: Explaining software clustering for remodularisation," *Journal of Systems and Software*, vol. 186, p. 111162, 2022.

[14] R. A. Coelho, F. dos RN Guimaraes, and A. A. Esmin, "Applying swarm ensemble clustering technique for fault prediction using software metrics," in *2014 13th International Conference on Machine Learning and Applications*. IEEE, 2014, pp. 356–361.

[15] S. P. R. Puchala, J. K. Chhabra, and A. Rathee, "Ensemble clustering based approach for software architecture recovery," *International Journal of Information Technology*, pp. 1–7, 2022.

[16] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: spectral clustering and normalized cuts," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 551–556.

[17] E. Schubert, S. Hess, and K. Morik, "The relationship of dbscan to matrix factorization and spectral clustering," in *LWDA*, 2018.

[18] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp. 849–856.

[19] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[20] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[21] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[22] D. L. Davies and D. W. Bouldin, "A cluster separation measure," *IEEE transactions on pattern analysis and machine intelligence*, no. 2, pp. 224–227, 1979.

[23] J. Ghosh and A. Acharya, "Cluster ensembles," *Wiley interdisciplinary reviews: Data mining and knowledge discovery*, vol. 1, no. 4, pp. 305–315, 2011.

[24] S. Vega-Pons and J. Ruiz-Shulcloper, "A survey of clustering ensemble algorithms," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 25, no. 03, pp. 337–372, 2011.

[25] X. Z. Fern and C. E. Brodley, "Solving cluster ensemble problems by bipartite graph partitioning," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 36.

[26] T. Li, C. Ding, and M. I. Jordan, "Solving consensus and semi-supervised clustering problems using nonnegative matrix factorization," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 2007, pp. 577–582.

[27] E. Balas and M. J. Saltzman, "An algorithm for the three-index assignment problem," *Operations Research*, vol. 39, no. 1, pp. 150–161, 1991.

[28] H. P. Williams, "Integer programming," in *Logic and Integer Programming*. Springer, 2009, pp. 25–70.

[29] M. J. Saltzman, "Coin-or: an open-source library for optimization," in *Programming languages and systems in computational economics and finance*. Springer, 2002, pp. 3–32.

[30] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[31] J. Fenn and H. LeHong, "Hype cycle for emerging technologies, 2011," *Gartner, July*, vol. 28, 2011.

[32] L. I. Kuncheva and D. P. Vetrov, "Evaluation of stability of k-means cluster ensembles with respect to random initialization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 11, pp. 1798–1808, 2006.

[33] "The 2021 state of Octoverse," https://octoverse.github.com/, accessed: 2022-06-19.