Improving the Comprehensibility of Generated Test Suites Using Test Case Clustering

Olsthoorn, Mitchell

DOI
10.1109/ICST62969.2025.10988953

Publication date
2025

Document Version
Final published version

Published in
Proceedings of the 2025 IEEE Conference on Software Testing, Verification and Validation (ICST)

Improving the Comprehensibility of Generated Test Suites using Test Case Clustering

1st Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

2025 IEEE Conference on Software Testing, Verification and Validation (ICST) | 979-8-3315-0814-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICST62969.2025.10988953

***Abstract*—Software testing is critical for ensuring the quality of software systems. Manually writing test cases is time-intensive and costly, which has led to the development of automated test case generation techniques. However, the adoption of these techniques is limited due to the difficulties in comprehending the generated test cases. In this paper, we propose an approach to improve the comprehension of automatically generated test suites by clustering the test cases within the test suite. Our approach clusters the test cases based on the test objectives (e.g., lines and branches) they cover, grouping together those with similar attributes to enhance developer understanding. To evaluate our approach, we conducted an empirical study with 52 participants performing three software maintenance tasks based on related work. The results show developers agree with the proposed clusters and that clustered test suites facilitate faster software maintenance tasks.**

***Index Terms*—software testing, test case generation, test case clustering, software comprehension**

I. Introduction

Through the years, search-based test case generation approaches have gained significant attention in the software testing community [1], [2], [3]. These approaches use meta-heuristic search algorithms to automatically generate test cases that satisfy specific test objectives, such as covering lines, branches, or paths in the code. While these techniques have been shown to be effective in detecting faults in software systems [4], [5], [6], their adoption in the industry remains limited [7].

One contributing factor to this limited adoption is that automatically generated test cases often still require significant human intervention to understand the test cases and validate the assertions [8], [9]. Therefore, by enhancing the comprehensibility of the generated test cases we can significantly reduce the cost associated with software testing activities when using automatic test case generation tools.

Program comprehension is a well-studied field, with most research focusing on source code, exploring aspects like readability, complexity, and visualization [10]. In contrast, test code comprehension has received less attention, likely due to its simpler structure. Some researchers have proposed techniques to improve test code comprehension [11], [12], [13], including for automatically generated test cases [14], [15], [16], [17]. However, all these studies focus on individual test cases or methods without exploring comprehension at the test suite level.

To address this gap, this work proposes a post-processing approach for automatically generated test suites that aims to reduce the effort required for developer comprehension. Our approach clusters test cases based on the similarity of the search objectives (*i.e.,* function and branch coverage) covered by the test cases to enhance the overall suite's comprehensibility. We hypothesize that clustering test cases based on what they cover will group together test cases that are semantically similar, making it easier for developers to understand the test suite as a whole.

To evaluate our hypothesis and approach, we conducted an empirical study with 52 participants performing three software maintenance tasks. The tasks were designed based on the concept of learning activities related to program comprehension [18], assessing the impact of test case clustering on the comprehensibility of the test suite. We evaluate both whether the participants agreed with proposed clusters and if clustering improved the comprehensibility of the test suite. The participants completed these tasks via an online survey, and their performance was evaluated based on the quality of the task outcomes and the time taken to complete the tasks.

Our results show that, in most cases, developers agree with the clusters of test cases. Regarding comprehensibility, we found that test suites using the clustering approach allowed developers to complete software maintenance tasks more quickly than those without clustering. While clustering improved task completion speed, it did not significantly affect the quality of task outcomes compared to non-clustered test suites.

In summary, we make the following contributions: (i) a novel test case clustering approach targeting automatically generated test suites, (ii) an empirical study evaluating the impact of test case clustering on the comprehensibility of such test suites, and (iii) a replication package to facilitate the reproducibility of our study [19].

II. Approach

Our approach is a post-processing step for automatically generated unit-level test suites that aims to enhance the comprehensibility of a test suite by clustering test cases based on the search objectives (*i.e.,* functions and branches) they cover. We hypothesize that clustering test cases based on their covered objectives will group together semantically similar test cases, making it easier to identify groups of related test cases, improving the comprehensibility of the test suite.

Our approach first extracts the covered objectives from the generated test cases. The objectives are then encoded into a binary representation, where each digit indicates whether the objective is covered by the test case (1) or not (0). Next, we apply dimensionality reduction to the binary representation of the test coverage data to reduce the high dimensionality and sparsity of the data. Thirdly, we cluster the test cases using the K-Means algorithm [20]. Both of these steps are explained in detail in the following subsections. Finally, our approach generates a new test suite using the assigned cluster labels to improve the organization and presentation of the test cases. Several testing frameworks support hierarchical organization of test cases. Mocha [21], for example, uses a nested organization style similar to RSpec Behavior-Driven Development style [22], employing keywords to define hierarchical relationships. JUnit 5 [23] on the other hand supports nested test classes, which can be used to represent the clusters. Alternatively, developers can use the cluster labels to group test cases in the test suite documentation or naming conventions.

A. Dimensionality Reduction

Test coverage data is characterized by high dimensionality and sparsity. The dimensionality of the data corresponds to the total number of functions and branches in the class under test, which can vary significantly depending on the class's complexity [24]. Additionally, as individual test cases typically cover only a small subset of these objectives, most dataset values are zero. The sparsity becomes worse as the complexity of the class increases, leading to a large number of dimensions with little to no data.

High dimensionality and sparsity pose significant challenges for clustering. In high-dimensional spaces, distances between data points tend to converge, reducing the effectiveness of distance-based clustering methods like K-Means [20] or hierarchical clustering [25]. Sparse data adds an additional challenge, as many dimensions have minimal impact on clustering but significantly increase computational overhead. Furthermore, sparsity causes data points to become widely dispersed in high-dimensional space, making it difficult for density-based methods like DBSCAN [26] to identify dense regions for clusters. Noise and outliers also have a greater impact in high-dimensional data, further complicating clustering tasks.

To address these challenges, we use an Autoencoder model [27] to obtain a low-dimensional representation of the data as it provides flexibility regarding architecture and activation functions. An Autoencoder consists of two components: an encoder, which compresses the input data into a reduced representation, and a decoder, which reconstructs the original data from this compressed form [27]. By minimizing reconstruction error during training, the Autoencoder learns the data's significant features while ignoring noise and irrelevant details.

After training the Autoencoder on the test case data, we use only the encoder to transform the high-dimensional data into a compact, low-dimensional representation. This representation effectively preserves the essential features of the test cases, making the data more suitable for clustering.

B. Clustering of Test Cases

After extracting features from the test case data using the Autoencoder model, we apply the K-Means algorithm to perform clustering. Since K-Means requires the number of clusters to be predefined, we use the Elbow Method [28] and Silhouette Coefficient [29] to determine the optimal number.

The Elbow Method [28] analyzes the relationship between the number of clusters and the Within-Cluster Sum of Squares (WCSS) [30]. As the number of clusters increases, WCSS decreases, but at a certain point, the reduction rate slows significantly, forming an "elbow". This "elbow point" indicates the optimal cluster number. However, identifying a clear "elbow point" is not always straightforward.

To address this, we also use the Silhouette Coefficient [29], which measures clustering quality by evaluating both cohesion within clusters and separation between clusters. A value close to 1 indicates well-defined clusters, with points tightly grouped within their own cluster and well-separated from others.

By combining the insights from the Elbow Method and the Silhouette Coefficient, we can reliably determine the optimal number of clusters for the test case data.

III. Empirical Study

To evaluate our hypothesis and approach, we conducted an empirical study with 52 participants performing three software maintenance tasks to answer the following research questions:

RQ1 To what extent do developers agree with the proposed clusters?

RQ2 To what extent does test case clustering improve the comprehensibility of the generated test suites?

The first research question aims to assess the effectiveness of our clustering approach in grouping similar test cases as developers might have a different view on what constitutes similar test cases. The second research question aims to evaluate the impact of clustering test cases on the comprehensibility of the test suite by measuring the efficiency and effectiveness of developers at performing software maintenance tasks.

A. Tasks

Since comprehensibility can not be measured directly, we designed three tailored software maintenance tasks related to software testing, inspired by prior research [18], that aim to indirectly assess the comprehensibility of test suites.

1. Fix the Failing Test Cases: Participants are given the change history of the class under test and an automatically generated test suite for the unmodified version of the code. Using the change history, participants were asked to identify failing test cases and modify them to ensure the test suite remained valid for the updated version of the code. For this task, we monitored (i) which test cases participants identified as failing, (ii) the number of correctly fixed test cases, and (iii) the time spent.

2. Identify Potential Error Scenarios: Participants are provided with an automatically generated test suite without access to the class under test. They were asked to analyze the test cases to determine input boundaries and conditions under which a specified method would throw exceptions. Additionally, they were also asked to rate their understanding of the class functionality and their confidence on this rating. For this task, we monitored (i) the number of input conditions selected that participants thought might cause the method to throw an error and (ii) the time spent.

3. Evaluate and Cluster the Test Cases: Participants were asked to review the clusters produced by our test clustering approach and rate their agreement with each cluster. They were also asked to share the criteria they would use to group test case if they were in charge of the clustering process. For this task, we monitored (i) the level of agreement with the clusters.

B. Classes Under Test

To generate the test case, we used SynTest-JavaScript [31], [32], a search-based test case generation tool for JavaScript, on its default settings as it integrates with Mocha [21], a testing library that offers native support for hierarchical test case organization. We generated test cases for three JavaScript classes: `Polygon.js` (9 functions, 18 branches, 161 sloc), `Queue.js` (8 functions, 12 branches, 110 sloc), and `ShoppingCart.js` (11 functions, 22 branches, 98 sloc). These classes were chosen to represent varying levels of complexity and to minimize participants having prior familiarity with them.

For Task 1, participants were provided with a code change history to identify test cases likely to fail due to the changes. These code changes were created by making modifications to the classes that (i) are related to one specific branch of the code at a time and (ii) are independent from other chances.

More details about the classes and the changes made to them can be found in the replication package [19].

C. Participants

The 52 participants that participated in the study represent a diverse range of JavaScript programming experience: 4 participants had less than a year of experience, 15 had 1-2 years, 22 had 3-5 years, 9 had 6-10 years, and 2 had over 10 years of experience. Their professional backgrounds included 29 software developers, 18 students, and 5 researchers. To ensure the integrity of the responses, qualifying questions were included to identify and exclude participants who appeared to complete the survey hastily.

D. Procedure

This study was conducted through an online survey created using Alchemer [33], a platform designed for interactive surveys. Before starting the tasks, participants completed a pre-task questionnaire to indicate their experience with software testing and JavaScript development. After completing each task, participants answered post-task questionnaires, which included open-ended questions and 5-point Likert scale ratings [34].

Task 1 involved two classes under test, each with two test suite variants: clustered and non-clustered. Participants were split into two groups: one started with the clustered test suite of the first class followed by the non-clustered test suite of the second, while the other group followed the reverse order. This was done to ensure that the results were not influenced by the order in which the participants completed the tasks [35]. For task 2, the participants were randomly given either the clustered or non-clustered test suite of the third class under test. Lastly, for task 3, the participants were given only the clustered test suite of one of the classes under test.

For statistical analysis, we opted for the Wilcoxon rank-sum test [36], a non-parametric statistical test, as not all data followed a normal distribution and this test is less sensitive to outliers and distributional assumptions. We used a significance level of 0.05 to determine statistical significance. Additionally, we used Cliff's Delta [37] to measure the effect size of the differences between treatments.

IV. Results

A. RQ1: To what extent do developers agree with the proposed clusters?

The results from task 3 show that most participants agreed with our clustering outcomes, with $40.4\,\%$ selecting *Agree* and $27.2\,\%$ selecting *Strongly agree*. The next most common response at $21.7\,\%$ was *Neutral*, neither agreeing nor disagreeing with our clustering results. Only a small subset of participants ($9.6\,\%$) disagreed.

In five out of the seven clusters, the majority of participants agreed with the clustering results, suggesting that participants generally agree that the test cases in these clusters share common features, justifying their grouping. However, for the remaining clusters, more participants were neutral or disagreed with the clustering results, indicating that these clusters may not have been as effective in grouping similar test cases.

In our qualitative analysis, we asked participants to explain their reasoning for selecting a particular response. The participants that agreed with the clustering indicated that the test cases invoked the same methods or had similar inputs and outputs. On the other hand, the participants that disagreed with the clustering results often cited reasons such as testing different methods, using different inputs, or having test cases with a different purpose.

One example of a cluster that participants generally disagreed with contained two test cases that invoked two different methods that did not seem related to each other. When we analyzed the test cases in this cluster, we found that one of these methods invoked the other internally. As a result, our clustering method grouped these test cases together, as there is an overlap in the branches covered by the test cases.

Another example where participants questioned the clustering is related to the sequence of method calls. When two test cases invoke the same methods but in a different order, our clustering approach might group them together. The participants, however, often considered the primary method invocation for determining how to cluster the test cases.

When we asked the participants about their criteria for clustering test cases, the majority of them mentioned method invocations as the most critical factor, with input parameters and output results coming next.

In summary, most participants agree with our clustering approach for the majority of the test case clusters, but they disagree with a few of the clustering outcomes.

B. RQ2: To what extent does test case clustering improve the comprehensibility of the generated test suites?

In task 1, we evaluated the effectiveness and efficiency of participants in fixing failing test cases. Effectiveness is measured by the number of test cases correctly fixed, while efficiency is the number of correctly fixed test cases divided by the time spent on the task.

Our results show that for the test suite generated from the `Polygon.js` class, participants were able to identify and fix slightly more test cases on average when using test case clustering. However, when looking at the statistical analysis, we found no significant difference between the two approaches (p-value $= 0.125$, effect size $= 0.246$). Similarly, for test suites generated from the `Queue.js` class, no significant difference in effectiveness was observed (p-value $= 0.765$), suggesting that test case clustering does not help participants identify and fix more test cases.

When looking at efficiency, we found that for the test suite generated from the `Polygon.js` class, test case clustering significantly improved efficiency (p-value $= 0.0246$, effect size $= 0.380$), meaning that participants were able to fix the test cases quicker. However, no significant difference in efficiency was found for the `Queue.js` class (p-value $= 0.315$). One possible explanation for this might have to do with the complexity of the classes. The `Queue.js` class has fewer branches and a lower cyclomatic complexity than `Polygon.js`, resulting in fewer generated test cases (8 vs. 17). With less test cases to review, participants could potentially find it easier to identify differences and similarities in the test suite without relying on clustering for comprehension. Our clustering approach might therefore not be as beneficial for simpler classes and test suites.

When we asked which aspects of the test suite were most helpful for fixing the test cases, the participants highlighted different elements for clustered and non-clustered test suites. For clustered test suites, the majority of the participants found the test suite structure most beneficial, followed by the test case naming and test assertions. In contrast, for non-clustered test suites, the participants identified test case naming as the most helpful feature, followed by the overall test suite structure. The focus on test case naming for non-clustered test suites is consistent with previous research on source code readability and test code readability, which has highlighted the importance of test case names in helping participants understand the content of test cases [15], [16], [17].

For task 2, where participants were asked to identify potential error scenarios in the test suite, our results show that participants using test suites with clustering identified more exception-triggering conditions on average (6.25) compared to those without clustering (5.625). Notably, some participants in the clustering group identified all exception conditions, while none in the non-clustering group achieved this.

Regarding efficiency, the clustering group also showed a slightly higher average score (0.697) compared to the non-clustering group (0.634). However, these differences were not statistically significant for either effectiveness or efficiency.

In summary, using test case clustering can significantly improve developer efficiency for certain software maintenance tasks when dealing with test suites that contain a large number of test cases, without affecting the quality.

C. Threats to Validity

One threat to the external validity of our study is the generalizability of our results. We conducted our study with three classes under test due to time constraints and the complexity of the tasks. The classes were chosen to have diversity in terms of complexity and application domain. Further experiments on a larger set of classes under test would increase the confidence in the generalizability of our study and, therefore, is part of our future work.

A threat to the internal validity of our study is the practice effect, where participants improve their performance on later tasks due to familiarity gained from earlier tasks. To mitigate this risk, we have followed the best practices for experimental design [18], such as counterbalancing the order of tasks and randomizing the test suite variants.

V. Conclusion and Future Work

In this paper, we proposed a test case clustering approach to enhance the comprehensibility of automatically generated test suites. Our approach clusters test cases based on the covered objectives, such as functions and branches, to group semantically similar test cases together. We conducted an empirical study with 52 participants to evaluate the impact of test case clustering on the comprehensibility of test suites through three software maintenance tasks.

Our results showed that most participants agreed with our clustering approach for grouping similar test cases. Furthermore, by using the clustering approach, participants were able to complete the software maintenance tasks quicker than with the non-clustered test suites, reducing the effort required to understand the test suite.

In future work, we plan to evaluate our clustering approach with a broader set of classes under test to further validate the effectiveness of our approach. Additionally, we want to explore other clustering algorithms and dimensionality reduction techniques to improve the clustering quality and reduce the computational overhead.

VI. Acknowledgments

This work was conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology.

REFERENCES

[1] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2009.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of systems and software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[4] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 143–154.

[5] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[6] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, pp. 611–639, 2015.

[7] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 337–346.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–38, 2015.

[9] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. Van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 192–203.

[10] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension. in 2017 ieee/acm 25th international conference on program comprehension (icpc)," *IEEE, 308ś311*, 2017.

[11] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 2007, pp. 213–222.

[12] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring test case similarity to support test suite understanding," in *Objects, Models, Components, Patterns: 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings 50*. Springer, 2012, pp. 91–107.

[13] S. Zhang, "Practical semantic test simplification," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1173–1176.

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.

[15] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[16] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 287–298.

[17] G. Gay, "Improving the readability of generated tests using gpt-4 and chatgpt code interpreter," in *International Symposium on Search Based Software Engineering*. Springer, 2023, pp. 140–146.

[18] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 348–359.

[19] M. Olsthoorn, "mitchellolsthoorn/icst-short-2025-test-clustering-replication: v1.0.0," Feb. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14867808

[20] M. Clarke, "Pattern classification and scene analysis," 1974.

[21] "Mocha - the fun, simple, flexible javascript test framework." [Online]. Available: https://mochajs.org/

[22] "Rspec: Behaviour driven development for ruby." [Online]. Available: https://rspec.info/

[23] "Junit5." [Online]. Available: https://junit.org/junit5/

[24] A. Arcuri and G. Fraser, "On the effectiveness of whole test suite generation," in *Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings 6*. Springer, 2014, pp. 1–15.

[25] L. Rokach and O. Maimon, "Clustering methods data mining and knowledge discovery handbook (pp. 321–352)," 2005.

[26] 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.

[27] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[28] R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[29] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[30] A. W. Edwards and L. L. Cavalli-Sforza, "A method for cluster analysis," *Biometrics*, pp. 362–375, 1965.

[31] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Guess what: Test case generation for javascript with unsupervised probabilistic type inference," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 67–82.

[32] M. Olsthoorn, D. Stallenberg, and A. Panichella, "Syntest-javascript: Automated unit-level test case generation for javascript," in *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, 2024, pp. 21–24.

[33] "Enterprise online survey software tools - alchemer." [Online]. Available: https://www.alchemer.com/

[34] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, "Likert scale: Explored and explained," *British journal of applied science & technology*, vol. 7, no. 4, pp. 396–403, 2015.

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236.

[36] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[37] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.