cASpER: ECR strategy based on Game Theory

Marco Calenda

ACM Reference Format:
Marco Calenda. 2022. cASpER: ECR strategy based on Game Theory. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

cASpER (Automated code Smell dEtection and Refactoring) is an IntelliJ IDEA plugin that (i) integrates two code smell detection approaches such as Decor [8] and Taco [10] to support the identification of four types of code smells (*i.e.*, Feature Envy, Misplaced Class, Blob and Promiscuous Package), (ii) offers refactoring recommendations implementing approaches previously proposed in literature [1, 2], (iii) automatically modifies the source code according to the desired refactoring operations and (iv) visualizes, in a single view, source code metrics, concepts and attributes.

In this paper, I focus on the previously point (ii) concerning the refactoring that, when facing Blob (or Promiscuous Package), involves an *Extract Class Refactoring* (ECR) approach to split a large low cohesive class (or package) into smaller classes with an higher cohesion. By applying ECR it is also likely that the overall coupling of the system will increase because of the new dependencies between the extracted classes. The following project is based on the studies of *Bavota et al.* [3]. They showed that game theory provides a suitable solution to the ECR problem by modelling it as a game between n player where each player represents a class to be extracted from a given class and seeks to maximize its cohesion while maintaining its coupling as low as possible.

The following sections describe the activities performed on the tool, in order to implement the game theory splitting, with emphasis on the results obtained compared to the default splitting approach performed by cASpER based on graph theory. Section 2 provides background information about game theory and the structural and semantic measures used in the approach. Section 3 briefly describes the ECR algorithm while Section 4 illustrates its implementation in cASpER. Finally, Section 5 presents the final outcomes and gives concluding remarks and thoughts.

Github. https://github.com/MarcoCalenda14/cASpER

2 BACKGROUND

In this section I provide information about game theory and the structural and semantic measures used to take into account the impact of refactoring on the overall cohesion and coupling.

Game Theory. Game theory is a branch of applied mathematics widely applied in the social sciences, especially in economics. It attempts to provides tools for analyzing strategic situations by capturing the behavior of individuals, called players, that make interdependent decisions. A solution to a game describes the optimal decisions of the players, who may have similar, opposed, or mixed interests, and the outcomes that may result from these decisions.

To formalize the problem I consider finite, n-player, normal-form games $G = \langle N, (A_i), (u_i) \rangle$:

- $N = \{1, \ldots, n\}$ is the set of players.
- $A_i = \{a_{i1}, \ldots, a_{im_i}\}$ is the set of actions available to player i, where m_i is the number of available actions for that player. I use $a = (a_1, \ldots, a_n)$ to denote a profile of actions, one for each player, and a_{-i} denote the same profile excluding the action of player i.
- $u_i : A_1 \times \ldots \times A_n \rightarrow \Re$ is a function that maps, for each player, a profile of actions to a value (payoff).

Each player i select a mixed strategy from the set P_i that specifies the probability distribution used to select the action that the player will play in the game.

Definition 1 *A strategy profile $p^* \in P$ is a Nash Equilibrium if:*

$$\forall i \in N, a_i \in A_i : u_i(a_i, p^*_{-i}) \leq u_i(p^*_i, p^*_{-i}) \tag{1}$$

In other words, Nash Equilibrium is a solution to any game, involving two or more players, in which each player is assumed to know the equilibrium strategies of the other players and no player has anything to gain by changing only his own strategy unilaterally. In 1950 John Nash demonstrated that each finite game have at least one Nash Equilibrium in mixed strategies [9].

Similarity measures for refactoring. In the approach proposed are used a combination of three structural and semantic measures, namely Structural Similarity between Methods (*SSM*), Call-based Dependency between Methods (*CDM*) and Conceptual Similarity between Methods (*CSM*).

Let I_i be the set of instance variables referenced by method m_i. The *SSM* of m_i and m_j is calculated as:

$$SSM\left(m_i, m_j\right) = \begin{cases} \frac{|I_i \cap I_j|}{|I_i \cup I_j|} & \text{if } \left|I_i \cup I_j\right| \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Thus, the higher the number of instance variables the two methods share, the higher the two methods should be in the same class.

Let $calls(m_i, m_j)$ be the number of calls performed by method m_i to m_j and $calls_{in}(m_j)$ be the total number of incoming calls to m_j. $CDM_{i \rightarrow j}$ is defined as:

$$CDM_{i \rightarrow j} = \begin{cases} \frac{\text{calls}(m_i, m_j)}{calls_{in}(m_j)} & \text{if } calls_{in}\left(m_j\right) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

So, if $CDM_{i \rightarrow j} = 1$, then m_j is only called by m_i while if $CDM_{i \rightarrow j} = 0$ then m_i never calls m_j. To ensure a commutative measure, just

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

like the other ones, the overall *CDM* is:

$$CDM_{i,j} = max\{CDM_{i\to j}, CDM_{j\to i}\} \tag{4}$$

Finally, two methods are conceptually related if their domain semantics are similar *i.e.* they perform similar actions. To measure *CSM*, Latent Semantic Indexing (LSI) is used to represent each method as a vector that spans a space defined by the vocabulary extracted from the methods. The conceptual similarity between two methods is then the cosine of the angle between their corresponding vectors. The *CSM* of m_i and m_j is calculated as:

$$CSM\left(m_i, m_j\right) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{\left\|\overrightarrow{m_i}\right\| \cdot \left\|\overrightarrow{m_j}\right\|} \tag{5}$$

where $\overrightarrow{m_i}$ and $\overrightarrow{m_j}$ are the vectors corresponding to the methods and $||\overrightarrow{x}||$ represents the Euclidean norm for the vector x. The three measures (*i.e. SSM*, *CDM*, *CSM*) have values in $[0, 1]$ each.

3 THE ECR ALGORITHM

This project is based on a proposed approach [3] that models the ECR problem as an iterative algorithm that incrementally assigns the methods of a smelly class to n players $P_1, P_2, \ldots, P_n$ representing the classes to be extracted. It takes as input a class C to be refactored, the number n of classes that should be extracted from C, and n seed methods (one for each player).

Identifying the players and the seed methods. The number of players, and a seed method for each player is not a trivial information as it's not possible to extract with an analysis of a Blob class due to its size and-or complexity. Thus, it is proposed an heuristic based on topic analysis technique using Latent Dirichlet Allocation (LDA) firstly applied in machine learning by Blei *et al.*[4] that is a topic model widely applied in natural language processing. It mimics the way a developer would read the code and assess the main responsibilities of the methods of a class based on the words composing the methods without a deeper structural analysis. By the textual information in the source code (identifiers and comments) the model extracts a set of topics each one composed by a list of recurring words. To obtain a better estimation the similar topics are merged, the similarity between two topics T_i and T_j is computed using the Jaccard similarity coefficient:

$$sim\left(T_i, T_j\right) = \frac{\left|W_i \cap W_j\right|}{\left|W_i \cup W_j\right|} \tag{6}$$

where W_k represents the set of words describing the topic T_k. Two topics are merged if their similarity is higher than a threshold s. The threshold is fixed at 0.5, *i.e.*, topics that share more than half of the words describing them are merged together. The final set of topics represents the responsibilities of the class to be refactored, which defines the number of players that will take part in the game. In addition, the seed method assigned to each player is the method with the highest relevance score with the topic corresponding to the player.

Starting the game. Each player initially holds only its seed method. The n players will then contend for the $l-n$ methods (where l is the number of initial methods of the Blob). At each iteration, each player takes at most one of the unassigned methods and, if a player does not take any method, then it plays the *null* move (the purpose is that a player takes a method only if there is a clear advantage in taking it). It is also prevented that every player play the null move at the same time, as this would stop the iterative algorithm. The combination of moves to be performed by the n players during an iteration is chosen by finding the Nash equilibrium in the payoff matrix. Each entry of the matrix corresponds to a combination of possible moves for the n players $P_1, P_2, \ldots, P_n$ and contains a tuple of payoffs $p_1, p_2, \ldots, p_n$ for this combination (a payoff for each player). At the first iteration, the size of the matrix is $(l - n + 1)^n$, as each player can take one of the unassigned $l - n$ methods or play the null move. For an entry in the matrix, the tuple of payoffs measures the effect of the corresponding tuple of moves on the cohesion and coupling of the classes under construction. Once the tuple of moves to be performed has been identified, the methods taken by the players are added to the corresponding classes and removed from the payoff matrix that is then recomputed without the chosen methods. Finally, the instance variables of the original class are distributed among the extracted classes according to how they are used by the methods in the new classes, each instance variable is assigned to the new class having the higher number of methods using it.

The payoff matrix. To compute the payoff matrix it is captured the effect of the moves on cohesion and coupling using a combination of the similarity measures presented in Section 2, *i.e.*, *SSM*, *CDM* and *CSM*.

First of all I compute the overall similarity between two methods m_i and m_J as:

$$\begin{aligned} \text{sim}\left(m_i, m_j\right) = w_{SSM} \cdot SSM\left(m_i, m_j\right) + \\ w_{CDM} \cdot CDM\left(m_i, m_j\right) + w_{CSM} \cdot CSM\left(m_i, m_j\right) \end{aligned} \tag{7}$$

where $w_{CDM} + w_{CDM} + w_{CDM} = 1$ and their values express the confidence in each measure.

The similarity between two methods is used to compute the similarity between a class under construction and a single method m_h that a generic player P_k can take during an iteration:

$$\text{Sim}\left(C_k, m_h\right) = \frac{1}{|C_k|} \sum_{m_l \in C_k} \text{sim}\left(m_l, m_h\right) \tag{8}$$

where C_k represents the set of methods already assigned to player P_k.

A payoff is then computed making the following assumption: during an iteration of the game the goal of each player is to take a method having a high similarity with the class it is building and to leave methods having a low similarity with its class to the other players. In this way, suppose that at a given iteration player P_k takes method m_i and leaves a non empty set of methods M_k to the other players. Then, the higher the similarity between C_k (*i.e.* the class P_k is building) and m_i, the higher the expected increase of cohesion for C_k, the higher the payoff for the player P_k. On the contrary, the higher the similarity between C_k and the methods in M_k, the higher the expected increase of coupling between C_k and the other classes, the lower the payoff for the player P_k. Thus, we define the payoff p_k for player P_k as:

$$p_{k,i} = \text{Sim}\left(C_k, m_i\right) - \frac{\sum_{m_j \in M_k} \text{Sim}\left(C_k, m_j\right)}{|M_k|} \tag{9}$$

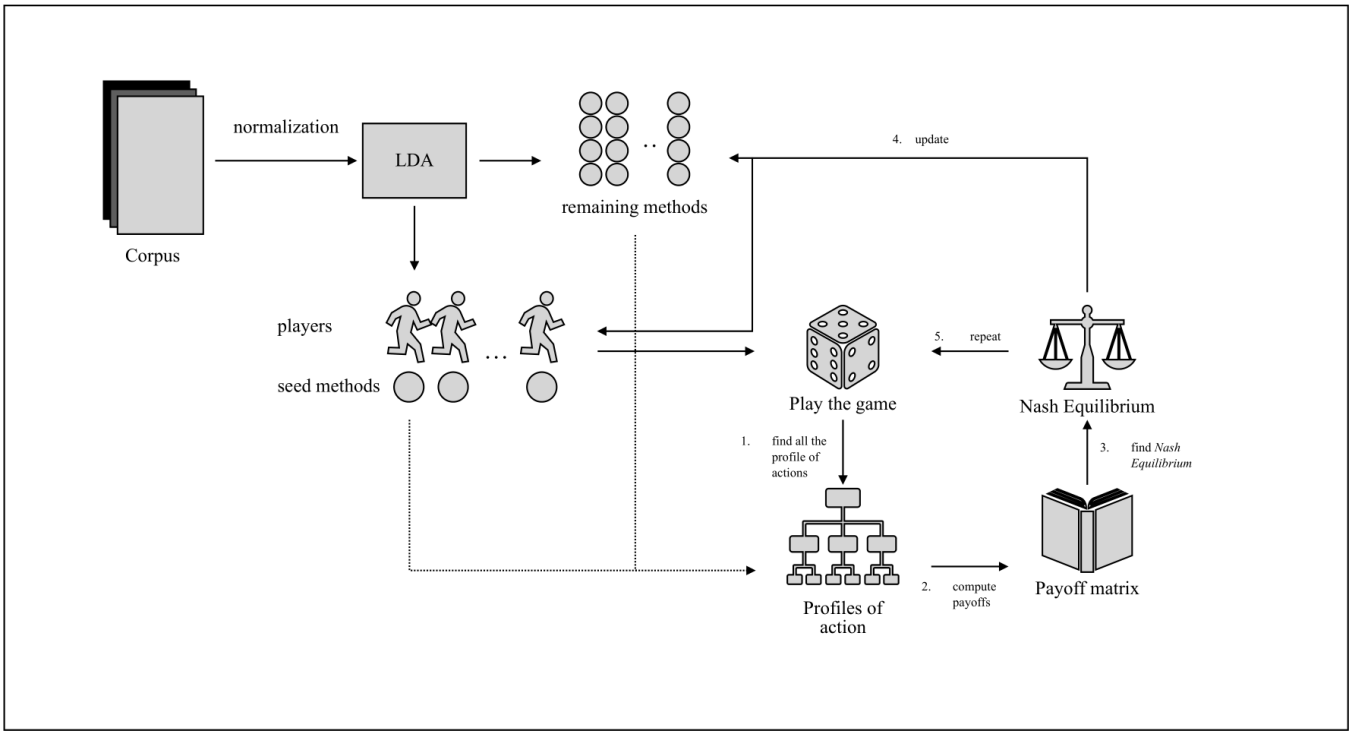


Figure 1: THE ENTIRE PROCESS OF THE ECR WITH GAME THEORY

A P_k is also allowed to play the *null* move. This move is still desirable for P_k if the methods in M_k taken by the other players have low similarity with the class C_k. Therefore, we define the payoff p_k as:

$$p_{k,null} = \epsilon_1 - \frac{\sum_{m_j \in M_k} \text{Sim}\left(C_k, m_j\right)}{|M_k|} \tag{10}$$

The value of ϵ_1 balances the negative effect of the similarity of class C_k with methods in M_k left to the other players: when the similarity between the class C_k and the methods in M_k is low, the payoff p_k encourages P_k to take the *null* move rather than taking a method with low similarity with its class.

Finally, a player P_k could be the only one taking a method m_i during an iteration, *i.e.*, all the other players play the *null* move. In this case, it is not possible to compute the average similarity between the class C_k and the methods left to the other players during this move, as the set M_k is empty and then the payoff p_k is defined as:

$$p_{k,i} = \text{Sim}\left(C_k, m_i\right) - \epsilon_2 \tag{11}$$

The value of ϵ_2 balances the positive effect of the similarity of class C_k with method m_i and it is used to discourage P_k to take a method with low similarity with C_k when the other players play the *null* move. The values of w_{CDM}, w_{CDM}, w_{CDM}, ϵ_1, ϵ_2 are empirically established [3].

4 IMPLEMENTATION IN CASPER

As described in SECTION 3 the first step concerned the topic extraction. First of all I performed a normalization process of the corpus (the collection of text documents, one for each method of the smelly class) by removing special chars, programming keywords (Java), and other common English/Italian terms; cASpER provide a useful stop-word list that is already used during the code smell detection phase. Finally I performed tokenization, in order to produce meaningful tokens suitable for the topic extraction, by splitting camel case and stemming by reducing inflected words to their root form. Regarding the LDA-based heuristic to identify the initial game configuration, I applied LDA using the MALLET tool [7] which is an implementation in Java of Gibbs' sampling algorithm. I also used the hankcs's implementation [5] but it behaves slightly worse in terms of accuracy of the number of topics detected. I run it for 1,000 sampling iterations with a burn-in of 200. I set the number of topics to extract to 10 and each topic is described by 20 words. As last step of the topic extraction, I merge the topics that have a Jaccard Similarity coefficient higher than 0.5 and I assign to each topic (*i.e.* an actual player) the method with the highest Jaccard Similarity coefficient as seed method of the player. The result is the input of the ECR algorithm: n players and for each a seed method.

The similarity between the methods, taking into account the measures described in SECTION 3, are computed in a matrix $l \times l$ (with l number of initial methods). To compute the payoffs, I find, using a recursive algorithm, all the possible n-element variation of $l - n + 1$ elements (the remaining methods to assign and the *null* move) with repetition allowed. Every variation denotes a profile of actions, one for each player. I use variation because I have to arrange $l - n + 1$ elements (the order matter) and pick n elements. Taking into account that $a_i = (a_{i1}, \ldots, a_{in})$ represents a generic

profile of actions and −1 is the value assigned to the *null* move, I discard the following profiles:

$$\exists a_{ij}, a_{ik} \in a_i : a_{ij} = a_{ik} \neq -1 \tag{12}$$

$$\forall a_{ij} \in a_i : a_{ij} = -1 \tag{13}$$

By this, I remove all the entries, in the original proposed approach payoff matrix, that would have −1 as value of payoff (in order to never pick this as Nash Equilibrium) *i.e.* the variations with two or more same method pick and the variation where each player plays the *null* move. When a valid variation is computed I calculate its payoff tuple:

- if a_i does not contain a *null* move then for each player the payoff is calculated with equation (9).
- if a_i contain at least one *null* move, the payoff is calculated with equation (10) for the player playing the *null* move.
- if a_i contain all *null* move except for one player, the payoff is calculated with equation (11) for the player not playing the *null* move.

The weights of the similarity measures (*i.e.* w_{SSM} , w_{CDM} , and w_{CSM}) and the balancing parameters (*i.e.* ϵ_1 and ϵ_2) used in the implementation are $w_{CSM} = 0.5$, $w_{SSM} = 0.1$, $w_{CDM} = 0.4$, $\epsilon_1 = 0.5$, and $\epsilon_2 = 0.2$.

Algorithm 1 Recursive-Variations

```
Input
    a = (): profile of actions to build
    A = (a_1, a_2, ·a_m): set of actions available to each player
    n: number of players
    pos = 0: current player position
Output
    P: set of payoffs

if (pos == n) then
    P ← P+Compute-Payoff(a)
else
    for k ∈ A do
        if (k == −1) or (k ∉ a) then          ▷ -1: null move
            a ← a + {k}
            Recursive-Variations(a, A, n, pos + 1)
        end if
    end for
end if
```

Once the payoffs are computed I look for the Nash Equilibrium: for each variation I check if it meets the equation (2) by comparing that each player is playing its best move knowing the other players strategies. In order to reduce the computational effort of the research I store, for each player, the maximum payoffs reachable from all the possible variation, *i.e.* no compare is needed if it's already the best payoff obtainable. Of course, a best payoff coming from a move already in the variation isn't accepted and neither the *null* move when all the other players are playing the *null* move as well. By filtering all the variation I produce a not null [9] set of Nash Equilibrium and, eventually (if more than one Nash Equilibrium is found), I choose the tuple with the highest total payoff.

After an iteration, the methods specified in the Nash Equilibrium are assigned to the respective players and removed from the list of remaining methods. The variations and the payoffs are recomputed. The entire process is described in Figure 1. At the end of the game I build the classes by assigning them the resulting methods choice and the instance variables with higher reference in the respective methods.

5 EVALUATION

I formulate the following research question:

- **RQ**$_1$: Do the extracted classes have a lower cohesion than the original class?
- **RQ**$_2$: Do the extracted classes have a higher cohesion and lower coupling than the classes extracted with the default splitting method?

To answer research questions, I compare the cohesion of the Blob with the refactored classes using the following metrics:

- **Lack of Choesion in Methods (LCOM)**: it counts the sets of methods in a class that are not related through the sharing of some of the class's fields. The original definition of this metric [6] considers all pairs of a class's methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods to not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that don't share a field access the number of method pairs that do.
- **Lack of Choesion in Methods 3 (LCOM3)**: it is a variation of LCOM and it counts the number of disjoint components in the graph that represents each method as a node and the sharing of at least one field of the class as an edge.
- **Cohesion Among Method of Class (CAM)**: it computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods. A metric value close to 1.0 is preferred.

In order to evaluate the trade off I also computed coupling to verify that the refactoring does not increase class coupling excessively. For this purpose I use the following metric:

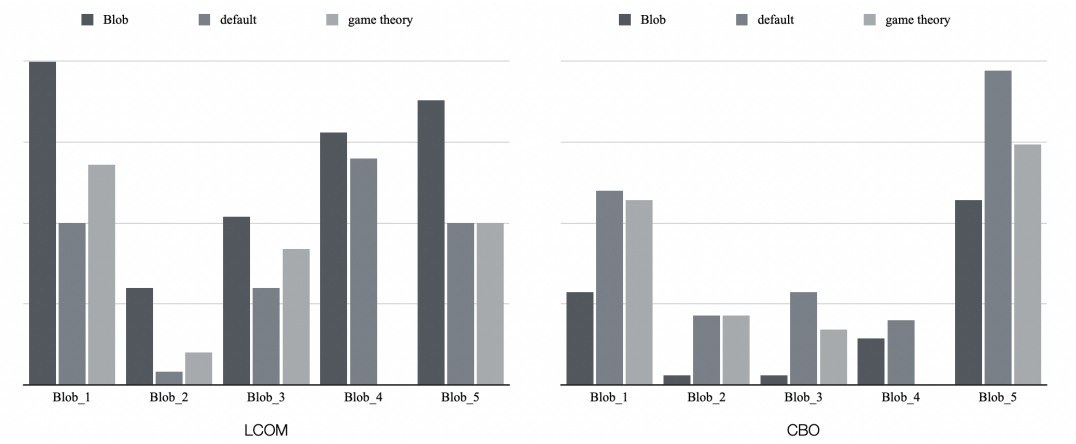
- **Coupling Between Object (CBO)**: it represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

As a first evaluation I took into account an artificial Blob created by myself merging four classes from my "Software Engineering" project. Table 1 compares the original Blob and the classes refactored using the two different approaches of cASpER. The table reports statistics concerning the cohesion and the coupling for each Blob and for each extracted class. The results show how the cohesion of the extracted classes with game theory is higher than the original Blob but the approach still produces a low cohesive

Table 1: refactoring an artificial Blob

Pre-refactoring				Post-refactoring							
				default approach				game theory approach			
LCOM	LCOM3	CAM	CBO	LCOM	LCOM3	CAM	CBO	LCOM	LCOM3	CAM	CBO
38	0.5625	0.3235	4	0	0.7308	0.3714	8	0	0.6667	0.5556	2
				1	1	0.75	2	0	0.50	0.05	2
				1	1	0.75	2	0	0	0.75	1
				1	2	0.75	1	0	0	0.5833	1
								10	0.7619	0.3125	4

Table 2: refactoring five Blob from topic "java-project" on Github



class compared to the classes extracted by the default approach. Despite of this, game theory approach maintains a slightly lower coupling between the objects. This confirm that game theory approach mainly aim to find the best trade-off between cohesion and coupling while the default approach try to increase as much as possible the extracted classes cohesion not worrying much about the side effect of the increasing coupling.

In a second experiment I refactored five Blobs from five different java project under the topic "java-project" on GitHub. They have been chosen Blob with an LCOM greater than 200. Table 2 shows the increasing cohesion and coupling (LCOM and CBO), the default approach obtains better results in terms of cohesion but it confirms an overall increasing of coupling unlike the game theory approach. A particular case is represented by the class Blob_4 where the approach could not split the original class since the LDA produced only one topic after merging. Due to limited resources, I can't evaluate the results of this implementation on larger open-source projects.

ECR with game theory produces good results in terms of trade off between cohesion and coupling. In spite of the results obtained, this implementation must be evaluated more, with tuning of the the balancing parameters (*i.e.* ϵ_1 and ϵ_2) and Jaccard Similarity threshold that is crucial to extract the right number of players. The performance are not good enough to evaluate large classes in a real-life scenario because it involves too many resources in terms of memory load and time elapsed. It is also crucial to remark that the time spent for the LDA training and the game iterations is up to ten times slower than the default approach and, considering the results obtained, in my opinion this is not a valid alternative at least for the actual implementation. The next study could focus on improve the overall project from an engineering perspective, trying to find better implementation choice in order to achieve better results in terms of performance.

REFERENCES

[1] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.

[2] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2013. Using structural and semantic measures to improve software modularization. *Empir. Softw. Eng.* 18, 5 (2013), 901–932. https://doi.org/10.1007/s10664-012-9226-8

[3] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Andrian Marcus, Yann-Gaël Guehéneuc, and Giuliano Antoniol. 2014. In medio stat virtus: Extract class refactoring through nash equilibria. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE).* 214–223. https://doi.org/10.1109/CSMR-WCRE.2014.6747173

[4] David M. Blei, A. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (2003), 993–1022.

[5] hankcs. 2017. *LDA implementation.* https://github.com/hankcs/LDA4j Accessed: June 15, 2022.

[6] M. Hitz and B. Montazeri. 1996. Chidamber and Kemerer's metrics suite: a measurement theory perspective. *IEEE Transactions on Software Engineering* 22, 4 (1996), 267–271. https://doi.org/10.1109/32.491650

[7] Andrew Kachites McCallum. 2002. MALLET: A Machine Learning for Language Toolkit. (2002). http://www.cs.umass.edu/ mccallum/mallet.

[8] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, X-Country = US, X-Editorial-Board = yes, X-International-Audience = yes, X-Language = EN, X-Pays, 1 (2010), 20–36.

[9] John Nash. 1951. Non-Cooperative Games. *Annals of Mathematics* 54, 2 (1951), 286–295.

[10] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A Textual-based Technique for Smell Detection. In *Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10. https://doi.org/10.1109/ICPC.2016.7503704