Extraction of archetype from near duplicates in software documentation

Dmitry Koznov
Software Engineering Dept.
Saint Petersburg State University
St Petersburg, Russia
d.koznov@spbu.ru
ORCID: 0000-0003-2632-3193

Dmitry Luciv
Software Engineering Dept.
Saint Petersburg State University
St Petersburg, Russia
d.lutsiv@spbu.ru
ORCID: 0000-0002-6332-2360

George Chernishev
Analytical Information Systems Dept.
Saint Petersburg State University
St Petersburg, Russia
g.chernyshev@spbu.ru
ORCID: 0000-0002-4265-9642

Dmitry Grigoryev
Computer Science Dept.
Saint Petersburg State University
St Petersburg, Russia
d.a.grigoriev@spbu.ru
ORCID: 0000-0001-7855-0254

***Abstract*—Software documentation contains a large amount of duplicate text, which is often comprised of near duplicates — repetitions of the same text with slight differences. They emerge due to numerous copy-pastes that have been slightly modified. Uncontrolled near duplicates complicate documentation support to a significant degree. There are some research papers on detection and management of duplicates in software documentation, but only the Duplicate Finder approach addresses the problem of near duplicates. Nevertheless, Duplicate Finder's search algorithms do not provide extraction of archetype (common text) for detected groups of near duplicates (a set of near duplicates belong to one group if they have a lot of commonalities). Archetype of group can be used in visualization of the common text and differences of duplicates for manual analysis, as well as for reuse of documentation. In this paper, we present an algorithm for archetype extraction and results of experiments on documentation of several well-known open source Java projects JUnit, Mockito, SLF4J.**

***Index Terms*—Software Documentation, JavaDoc, Duplicates, Near Duplicates**

I. Introduction

Contemporary software documentation, just like the software, is becoming increasingly more complicated with every passing year. During the last 50 years, the importance of software documentation has been noted unfailingly. However, the problem of its quality has been recognized as well.

One of the obstacles for efficient documentation maintenance is the presence of textual duplicates [1–4]. A textual duplicate is a fragment of text which is repeated multiple times throughout the document, possibly with some degree of variation. Software documents accumulate a large number of textual duplicates during development and especially maintenance. Usually, textual duplicates emerge as the result of a copy-paste pattern where a textual fragment is copied and pasted into a different part of the document, possibly in a modified form. A collection of near duplicates which have a meaningful common part (an archetype) and small differences (deltas) can be combined in a single group [5].Thus, the term "the archetype of a near duplicate group" can be utilized.

The Duplicate Finder approach has been proposed in [4]. It implements interactive detection of near duplicates in software documentation [6]. Here, the meaningfulness of a found duplicate is provided via user participation. However, this approach does not explicitly extract the archetype of a near duplicate group. Meanwhile, archetypes allow to apply reuse techniques to documentation [2, 7–9], as well as improve the visualization of near duplicates, highlighting both the common part and differences of near duplicate groups.

The contributions of this paper are as follows:

- an algorithm for automatic extraction of the archetype of a near duplicate group, an estimate of its complexity, as well as new definitions of a near duplicate group and an archetype.
- an evaluation of the archetype extraction algorithm on documentation of open source projects JUnit, Mockito, SLF4J.

The rest of this paper is organized as follows: Background (Section II) describes our previous work which provides the context for the current paper. Related work (Section III) describes the longest common subsequence problem because it is the closest to archetype extraction, as well as the existing approaches in this area. Definitions (Section IV) provides basic definitions used in this work, including new definitions of a near duplicate group and an archetype. Archetype extraction algorithm (Section V) describes the proposed algorithm, provides an estimate of its complexity, and discusses its limitations. Evaluation (Section VI) describes the evaluation of the algorithm.

II. Background

Our previous research was dedicated to adaptive reuse of software documentation [7, 8] based on extending adaptive reuse of software by Bassett-Jarzabek [5, 10] to the documentation domain. In this context, we have faced the problem of detection of near duplicates. In most cases, real software documentation is developed ad hoc, and it needs improvement: in particular, by means of searching for near duplicates and facilitating documentation reuse. Methods and algorithms for near duplicate detection were suggested [6, 8, 11, 12], a method to improve software documentation based on duplicate detection was presented [13], the Duplicate Finder tool to support duplicate detection was developed [4].

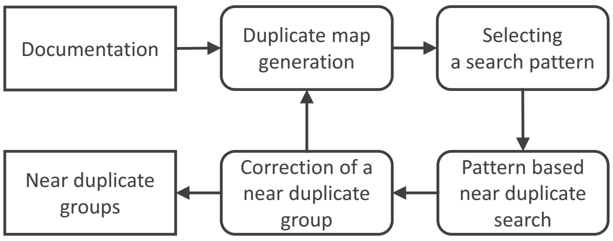
Documentation
Duplicate map generation
Selecting a search pattern
Pattern based near duplicate search
Correction of a near duplicate group
Near duplicate groups

Fig. 1. Software documentation improvement workflow

The context of this paper is a method for pattern based near duplicate detection presented in [6]. This method provides meaningfulness of duplicate detection via user participation. A diagram which describes the workflow of the technique is presented in Fig. 1. Let us describe the method in detail.

During *Duplicate map generation*, a reuse map of the document is created. As preparation, the automated near duplicate detection algorithm is run. It provides every token of the document with a weight that equals the number of near duplicates that contain this token. These weights are normalized, and next, represented by colors of different saturation from red to white. Thus, the user sees the most repeated text fragments in the document as the reddest areas on the map. In one of these areas, the user selects a search pattern (*Selecting a search pattern*), and then starts the *Pattern based near duplicate search*. Next, the user performs *Correction of a near duplicate group* that was detected, removing false positives and modifying boundaries of the text fragments.

Pattern-based near duplicate search uses Levenshtein distance [14] to detect similar text fragments, but does not extract common parts of these text fragments explicitly. In turn, it does not have visualization of common parts and differences of near duplicates in one group. Moreover, it does not allow to perform documentation refactoring and reuse the way we have suggested in [8, 11]. We close the gap in this paper.

III. Related work

The problem of finding the commonalities in some data (and the related problem of finding differences) is well known. For example, there is a large number of diff algorithms which compare text, trees, etc. [15, 16]. However, these algorithms usually compare only two information asserts. In our case, the number of text fragments in a near duplicate group is generally greater than two.

Sequence Alignment is a known bioinformatics problem [17], which is aimed at finding similar genome regions. These algorithms are used for text analysis as well [18], and they can be applied to two and more data elements. Despite them being suitable for our task, we should note that these algorithms are intended for large data volumes, and the tools that implement them are complicated and heavyweight. Meanwhile, in our case it would be more appropriate to use simpler solutions.

Token-based clone detectors can be employed for solving our task [19]. However, these tools are intended for source code analysis. Therefore, they have to be additionally configured and adapted to be used for documentation. We have used Clone Miner [20], but it emerged that its algorithms are quite complex and somewhat generally oriented. Besides, they do not work quite as required for solving our task. Furthermore, these tools contain bugs that are not easy to fix. All of these reasons make using this kind of tools not appropriate for our task.

The Longest Common Subsequence problem is widely known [21]. It is aimed at finding commonalities in different kinds of data sets. Its contexts are natural language processing [22, 23], time series analysis [24], etc. However, the context of our task includes not only finding an LCS, but also calculating all of its occurrences (i.e. coordinates) in the input data. Nevertheless, there are algorithms for this problem that have open source implementations and are not intended for tasks that are more general. Thus, they are reliable. We have decided to use one of these algorithms [25] and its open source implementation in Python [26].

IV. Definitions

Let us provide several definitions that will be used further on for describing the proposed archetype extraction algorithm.

Definition 1: **A document** D is an ordered sequence of tokens (words).

It should be noted that, compared to [6], we use tokens rather than symbols. Neither the definitions nor the algorithm and its properties are affected by this.

Definition 2: **A text fragment** of document D is $f = (b, e)$, where b denotes the number of the first token of the fragment from the start of document D, and e — the last. The length of fragment f is denoted as $|f| = (e - b + 1)$, and its tokens as $t_1, \ldots, t_{|f|}$.

Definition 3: **An exact duplicate group** is a collection of document fragments whose text is identical.

Definition 4: Let $F = \{f^1, \ldots, f^N\}$ be a set of text fragments of D, and $A = \langle A_1, \ldots, A_M \rangle$ be a sequence of exact duplicate groups. The following is satisfied:

1) For each f^j and $\forall i\ a_i^j \subset f^j$, and in each f^j elements a_i^j occur in increasing order of i.
2) $A' = \langle A'_1, \ldots, A'_{M'} \rangle$, for which the previous property is satisfied, and $\sum_{j=1}^{M'} \sum_{i=1}^{N} |{a'}_j^i| > \sum_{j=1}^{M} \sum_{i=1}^{N} |a_j^i|$, does not exist.

Then A is the **archetype** of F, the remaining text of F is **deltas**.

Definition 5: $F = \{f^1, \ldots, f^N\}$ is a **near duplicate group** if:

1) $\exists A = \langle A_1, \ldots, A_M \rangle$ which is the archetype of F, and
2) $\exists\, 0 < k \leq 1$:

$$\forall j \in \{1, \ldots, N\} : \frac{\sum_{i=1}^{M} |a_i^j|}{|f^j|} \geq k.$$

Let us consider an example of a near duplicate group from the JUnit documentation $F = \{f^1, f^2\}$. The f^1 is as follows:

```
A matcher that delegates to
throwableMatcher and in addition appends
the stacktrace of the actual Throwable in
case of a mismatch.
```

The f^2 looks like this:

```
A matcher that delegates to
exceptionMatcher and in addition appends
the stacktrace of the actual Exception in
case of a mismatch.
```

We have **bolded** the deltas here. As can be seen, the fragments of the archetype belong to three exact groups $A = \langle A_1, A_2, A_3 \rangle$: A_1 consists of three occurrence of the following string "`A matcher that delegates to`", A_2 — "`and in addition appends the stacktrace of the actual`", and A_3 —"`in case of a mismatch`".

Definition 6: $\text{LCS}(s_1, s_2)$ is an operation that detects the longest common subsequence of tokens for strings s_1 and s_2.

Definition 7: $\text{next}(s, t, n) = \inf\{j | s_j = t \wedge j \geq n\}$ is a function looks up the first token t in string s, starting with index $n \in 1, \ldots, |s|$.

V. Archetype extraction algorithm

A. Algorithm

The proposed algorithm consists of two parts: (i) detecting repeating text in the fragments of a near duplicate group and (ii) calculating the occurrences of this text. Every occurrence of the archetype $\langle a_1^j, \ldots, a_M^j \rangle$ in text fragment f^j is represented as a collection of integer intervals, where each interval comprises a pair of coordinates (beginning and end) of an archetype fragment a_i^j in document D. The specification of the algorithm is presented in Listing 1. The input of the algorithm is a group of near duplicates $F = f^1, \ldots f^N$ found in document D. The output of the algorithm is the group's archetype $A = \langle A_1, \ldots, A_N \rangle$, which is a sequence of exact duplicate groups of document D, where $\langle a_1^j, \ldots, a_M^j \rangle$ are contained in f^j.

Algorithm 1: Extraction of near duplicate group archetype

```
Input: $F = \{f^1, \ldots, f^N\}$ — near duplicate group
1  $N \leftarrow \#F$
2  $T \leftarrow f^1$
3  for $j = 2, \ldots, N$ do  $T \leftarrow \text{LCS}(T, f^j)$
   // $T = t_1, \ldots, t_{|T|}$ contains archetype
      tokens
4  $A \leftarrow$ empty sequence, $B = 0, E' = 0, E = 0$
   // $B, E', E$ are integer vectors
5  for $j = 1, \ldots, N$ do  $b_j \leftarrow \text{next}(f^j, t_1, 1)$
6  $E \leftarrow B$
7  for $t = t_2, \ldots, t_{|T|}$ do
8      for $j = 1, \ldots, N$ do  $e'_j \leftarrow \text{next}(f^j, t, e_j + 1)$
9      if $\exists j:\ e'_j > e_j + 1$ then
10         append $\langle (b_1, e_1), \ldots, (b_N, e_N) \rangle$ to $A$
11         $B \leftarrow E'$
12     $E \leftarrow E'$
13 append $\langle (b_1, e_1), \ldots, (b_N, e_N) \rangle$ to $A$
Output: $A$ — archetype of $F$
```

On lines 2 and 3, the archetype is detected as a sequence of tokens. Next, on lines 4–13, the occurrences of this sequence in every text fragment f^j are calculated, i.e., exact duplicate groups are constructed. The loop on line 5 calculates the first token of the archetype in every text fragment f^j (i.e., the first token for all elements of the first exact duplicate group A_1). The loop on line 7 iterates over all tokens of the archetype, starting with the second one. On line 8, all of the occurrences of archetype token t_i in text fragments of F are found (more precisely, the closest occurrences to the previously found occurrences of t_{i-1}). The coordinates of these occurrences are placed into vector E'. Lines 9–11 check whether if it is true that at least one occurrence of t_i (vector E') does not come after the corresponding occurrence of t_{i-1} in D (vector). If this is not true, then the construction of the current exact duplicate group continues. However, if it is true, then the group that was constructed on this step is added to the archetype (line 10), and coordinates contained in E' are considered as the start of a new group (line 11). On line 13, the construction of the last exact duplicate group added to A is finished. Note that the construction of the first and the last groups of A is removed from the main loop (lines 7–12), because the loop detects the extension points of the archetype, whereas the beginning and the end cannot be such.

B. Complexity

Theorem 1: The computational complexity of the proposed algorithm is $\mathcal{O}(N * L_F^2)$, where N is the number of near duplicates in F, and $L_F = \max_{f^j \in F} |f^j|$ is the size of the largest (by the number of tokens) text fragment in F.

We will prove that the complexity can be estimated as $\mathcal{O}((N-1) * L_F^2) + \mathcal{O}(N * L_F)$. The assertion of the theorem will follow from this, since, in our case, $L_F \geq 1$. During archetype extraction, we search for the longest common subsequence of strings that consist of no more than $L_F = \max_{f^i \in F} |f^i|$ tokens $N-1$ times. The algorithm of $\text{LCS}(s_1, s_2)$ calculation has the complexity of $\mathcal{O}(|s_1| * |s_2|)$ [25]. Hence, we can estimate the computational complexity of archetype extraction as $\mathcal{O}((N-1) * L_F^2)$. During the detection of archetype occurrences in near duplicates, all N near duplicates (that contain no more than L_F tokens each) are iterated over with the next operation once. Hence, the complexity of the loops on lines 6 and 7 is estimated as $\mathcal{O}(N * L_F)$, although they are nested. Furthermore, occasionally, component-wise operations on vectors B, E, E' and a collection of exact groups V are performed inside the loop on lines 6–11 and outside of it. The complexity of each such operation is estimated as $\mathcal{O}(N)$. They are performed no more than $|T|$ times, but since it is obvious that $|A| \leq L_F$, the complexity of the entire second part of the algorithm can be estimated as $\mathcal{O}(N * L_F)$. Thus, the overall computational complexity of the algorithm can be estimated as $\mathcal{O}((N-1) * L_F^2) + \mathcal{O}(N * L_F)$.

C. Constraints

In some cases, the found occurrences of an archetype in near duplicates and even its composition can depend on the order the duplicates are being considered. However, these cases are extremely rare in practice, and taking them into account increases the complexity of the algorithm. Therefore, we have decided to disregard them.

VI. Evaluation

The proposed algorithm has been implemented in the Duplicate Finder toolkit [4]. We have used the longest common subsequence algorithm (LCS) from [25, 26].

For our experiments, we have used three well-known open source Java projects: JUnit, Mockito, and SLF4J. Using Duplicate Finder, we have found 190 near duplicates in 50 groups in their JavaDoc documentation. In the process, we have limited the search to the comments to similar methods (e.g., `assertEquals` for different parameter types).

We have obtained the following performance results for this corpus: the archetype extraction algorithm run for less than 0.01 second for any group, which is negligible in comparison to pattern search (from 5 seconds to 2 minutes [6]). The archetypes of all groups were detected correctly.

We have conducted more experiments for additional quality control of archetype extraction. We have found duplicate groups which contained the largest text fragments (from 80 to 500 tokens) and whose archetype was split into many parts (with a maximum of 7). These kind of data are quite complex for our algorithm. In one case, the algorithm's output was incorrect. Moreover, it emerged that when archetype extraction is conducted on large text fragments that do not have any common text, the algorithm places random words into the archetype. For example, consider a string "`for Mockito class`". The algorithm will find its duplicate in a different text fragment as several disconnected words "`for...Mockito...Class`", adding all of them to the archetype. However, we should note that both cases arised on quite complex text fragments that are rarely encountered in practice.

A lot of the near duplicates (copy-pastes) we have found were actually exact duplicates except for the occurrences of a single term, e.g., `bool/double/char`. Large new text fragments were inserted into copy-pastes locally. At the same time, detecting the near duplicate groups in our experiments, we were trying to make each of them comprise a full description of some source code functionality.

According our experiments copy-pasted fragments rarely changed a lot on a small text interval, which is proven by the small number of splits. That is, the average number of elements in archetype $A = \langle A_1, \ldots, A_M \rangle$ in our corpus was $|A| = 3$. Therefore, on average archetypes were split into three parts. However, we have found an example that contained seven.

Furthermore, in 60 groups we detected it was found 9 exact duplicate groups. In our previous studies [6, 8, 12], the percentage of exact duplicate groups was significantly higher. Furthermore, considering near duplicate groups, most of their archetypes were split into two parts, i.e. $|A| = 2$. Therefore, we can hypothesize that the majority of duplicates in documentation are near duplicates, and there is only a problem to detect them. To do that we need to have the available tools, as well as the efficiency of the grouping technique used. We have confirmed the necessity for the latter in this study: often, ambiguities arise during classification. Consider an example. String S is quite short and is often encountered in text (we have had cases where the number of occurrences has been greater than 100). However, some of its occurrences are contained in larger fragments that are also duplicates, but there is a considerably smaller number of them. It is unclear in which group S should be placed under condition that we would like to have duplicate groups that do not intersect. Moreover, even more complicated cases can arise.

The technique should also contain rules for detecting boundaries of copy-pastes that are included into near duplicate groups. As mentioned above, we believe that one group should correspond to different descriptions of the same functionality in source code, used in different parts of software. For example, several methods for creating a rectangle can exist in a graphical library: using specified points, etc. The documentation for all of these methods can be identical, only differing in the parameter descriptions. It may be reasonable to have a single near duplicate group that contains the documentation for all such methods.

VII. Conclusion

The proposed archetype extraction algorithm has significantly improved practical usability of the interactive duplicated search in software documentation. First, an explicitly detected archetype allows to see the copy-pasted text clearly, as well as what has been added to it in every fragment. This eases finding discrepancies in a text fragments, and improves the understanding of situation with duplicate text as a whole. Second, having an archetype facilitates the reuse of documentation text, replacing the archetype in every group with an occurrence of the same text, and represent the differences of the fragments as parameters of these occurrences [7, 8]. With this, it becomes possible to automatically apply all changes of the archetype to all fragments of the group.

Our further research will be focused on implementing documentation reuse in different documentation formats, as well as integration of both pattern-based search and archetype extraction with duplicate visualization tools. The obtained results can be used for studying near duplicates in software documentation. Furthermore, it would be reasonable to develop an experimental tool for duplicate detection specifically in JavaDoc comments, providing users with both a technique for grouping similar information and additional duplicate attributes (for example, names for program entities that the duplicates are referring to). Finally, an interesting research direction is integration of task of finding and using near duplicates with ontology engineering [27, 28].

References

[1] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?" in *Proceedings of ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 79–88.

[2] M. Nosál' and J. Porubän, "Reusable software documentation with phrase annotations," *Central European Journal of Computer Science*, vol. 4, no. 4, pp. 242–258, 2014.

[3] A. Wingkvist, W. Lowe, M. Ericsson, and R. Lincke, "Analysis and visualization of information quality of technical documentation," in *Proceedings of the 4th European Conference on Information Management and Evaluation*, 2010, pp. 388–396.

[4] D. Luciv, D. Koznov, G. Chernishev, H. A. Basit, K. Romanovsky, and A. N. Terekhov, "Duplicate finder toolkit," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 171–172.

[5] P. G. Bassett, *Framing Software Reuse: Lessons from the Real World*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[6] D. Luciv, D. Koznov, A. Shelikhovskii, G. A. C. K. Romanovsky, A. Terekhov, D. Grigoriev, and A. Smirnova, "Interactive near duplicate search in software documentation," *To appear in Programming and Computer Software*, no. 6, 2019.

[7] D. Koznov and K. Romanovsky, "Docline: A method for software product lines documentation development," *Programming and Computer Software*, vol. 34, no. 4, pp. 216–224, 2008.

[8] D. Koznov, D. Luciv, H. A. Basit, O. E. Lieh, and M. Smirnov, "Clone Detection in Reuse of Software Technical Documentation," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics (2015)*, ser. Lecture Notes in Computer Science, M. Mazzara and A. Voronkov, Eds. Springer Nature, 2016, vol. 9609, pp. 170–185.

[9] M. A. Oumaziz, A. Charpentier, J.-R. Falleri, and X. Blanc, “Documentation Reuse: Hot or Not? An Empirical Study,” in *Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, May 29-31, 2017, Proceedings*, G. Botterweck and C. Werner, Eds. Cham: Springer International Publishing, 2017, pp. 12–27.

[10] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, “XVCL: XML-based variant configuration language,” in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 810–811.

[11] D. Luciv, D. Koznov, H. Basit, and A. Terekhov, “On fuzzy repetitions detection in documentation reuse,” *Programming and Computer Software*, vol. 42, no. 4, pp. 216–224, 2016.

[12] D. Luciv, D. Koznov, G. Chernishev, A. N. Terekhov, K. Romanovsky, and D. Grigoriev, “Detecting Near Duplicates in Software Documentation,” *Programming and Computer Software*, vol. 44, no. 5, pp. 335–343, Sep. 2018.

[13] D. Koznov, D. Luciv, and G. Chernishev, “Duplicate management in software documentation maintenance,” in *Proceedings of V International conference Actual problems of system and software engineering (APSSE 2017)*, vol. 1989. CEUR Workshop Proceedings, 2017, pp. 195–201.

[14] V. Levenshtein, “Binary codes capable of correcting spurious insertions and deletions of ones,” *Problems of Information Transmission*, vol. 1, pp. 8–17, 1965.

[15] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[16] T. Lindholm, “A three-way merge for XML documents,” in *Proceedings of the 2004 ACM symposium on Document engineering*. ACM, 2004, pp. 1–10.

[17] X. Liu, L. J. Dekker, S. Wu, M. M. Vanduijn, T. M. Luider, N. Tolić, Q. Kou, M. Dvorkin, S. Alexandrova, K. Vyatkina *et al.*, “De novo protein sequencing by combining top-down and bottom-up tandem mass spectra,” *Journal of proteome research*, vol. 13, no. 7, pp. 3241–3248, 2014.

[18] A. N. Pankratov, R. K. Tetuev, M. I. Pyatkov, V. P. Toigildin, and N. N. Popova, “Spectral analytical method of recognition of inexact repeats in character sequences,” *Proceedings of the Institute for System Programming of the RAS*, vol. 27, no. 6, pp. 335–344, 2015.

[19] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[20] H. A. Basit and S. Jarzabek, “Efficient token based clone detection with flexible tokenization,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 513–516.

[21] L. Bergroth, H. Hakonen, and T. Raita, “A survey of longest common subsequence algorithms,” in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000.* IEEE, 2000, pp. 39–48.

[22] M. Elhadi and A. Al-Tobi, “Duplicate detection in documents and webpages using improved longest common subsequence and documents syntactical structures,” in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*. IEEE, 2009, pp. 679–684.

[23] T. Gottron, “Clustering template based web documents,” in *European Conference on Information Retrieval*. Springer, 2008, pp. 40–51.

[24] A. Al-Maruf, H.-H. Huang, and K. Kawagoe, “Time series classification method based on longest common subsequence and textual approximation,” in *Seventh International Conference on Digital Information Management (ICDIM 2012)*. IEEE, Aug. 2012.

[25] J. W. Ratcliff and D. E. Metzener, “Pattern Matching: The Gestalt Approach,” *Dr. Dobb’s Journal*, vol. 13, no. 7, pp. 46–72, 1988.

[26] “Python difflib module,” https://docs.python.org/3/library/difflib.html.

[27] E. Bolotnikova, T. Gavrilova, and V. Gorovoy, “To a method of evaluating ontologies,” *Journal of Computer and Systems Sciences International*, vol. 50, no. 3, pp. 448–461, 2011.

[28] D. Kudryavtsev and T. Gavrilova, “From anarchy to system: A novel classification of visual knowledge codification techniques,” *Knowledge and Process Management*, vol. 24, no. 1, pp. 3–13, 2017.