https://doi.org/10.31891/2307-5732-2025-353-3
УДК 004.051:004.912

YUSYN YAKIV
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
https://orcid.org/0000-0001-6971-3808
e-mail: yusyn@pzks.fpm.kpi.ua

OPTIMIZATION OF MEMORY USE BY IMPLEMENTATIONS OF THE BASIC CORDEGEN METHOD

This paper is devoted to the task of generating text corpora "on demand" as input data for solving software engineering problems during the development of information systems for their processing. One of the methods that solves such a task is the basic CorDeGen method, however, as the analysis showed, practically none of the existing studies consider the issue of optimizing practical metrics of software implementations of this method, such as memory usage. Only a few papers propose pre-allocating memory for generated texts "with excess" to simplify and speed up the generation process by removing unnecessary checks and constant memory allocation. However, this approach, implemented as a fast heuristic formula, leads to increased memory usage in most cases.

To solve this shortcoming, the paper proposes a formula for exactly estimating the length of each text by its ordinal index, depending on the input parameters of the basic CorDeGen method (the number of unique terms). This formula considers the set of terms that occur in a particular text, their length, the number of occurrences of each, as well as the length of separators between occurrences of the same term and between different terms.

The experimental verification showed the effectiveness of using the proposed formula for exact text length estimation in terms of reducing memory consumption by the reference implementation of the basic CorDeGen method and its parallel modifications. The efficiency increases with the corpus size – from 3% for small (2500 unique terms) to 10% for very large corpora (312500 unique terms) compared to using the existing fast heuristic formula. At the same time, the degree of slowdown in the generation process decreases with increasing corpus size – from 17 to 6 percent at the same size. In practice, reducing memory consumption by increasing "net" generation time can be especially useful for systems with little or limited available memory to avoid memory overuse.

Keywords: natural language processing; corpora generation; CorDeGen method; memory consumption optimization.

ЮСИН ЯКІВ
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

ОПТИМІЗАЦІЯ ВИКОРИСТАННЯ ПАМ'ЯТІ РЕАЛІЗАЦІЯМИ БАЗОВОГО МЕТОДУ CORDEGEN

Ця робота присвячена задачі генерування корпусів текстів «на вимогу» в якості вхідних даних для вирішення задач програмної інженерії під час розроблення інформаційних систем для їх обробки. Одним із методів, що вирішує таку задачу, є базовий метод CorDeGen, проте, як показав проведений аналіз, практично жодна із існуючих робіт не розглядає питання оптимізації практичних метрик програмних реалізацій цього методу, таких як використання пам'яті. Лише деякі роботи пропонують попередньо виділяти пам'ять для генерованих текстів «з надлишком», щоб спростити та пришвидшити процес генерування за рахунок видалення зайвих перевірок та постійного виділення пам'яті. Але такий підхід, реалізований у вигляді швидкої евристичної формули, призводить до збільшеного використання пам'яті у більшості випадків.

Для вирішення цього недоліку в роботі запропоновано формулу точної оцінки довжини кожного тексту за його порядковим індексом в залежності від вхідних параметрів базового методу CorDeGen (кількості унікальних термів). Ця формула враховує множину термів, що потрапляють до певного тексту, їх довжини, кількість входжень кожного, а також довжини роздільників між входженнями одного терму та між різними термами.

Проведена експериментальна перевірка показала ефективність використання запропонованої формули точної оцінки довжини тексту у частині зменшення споживання пам'яті еталонною реалізацією базового методу CorDeGen та його паралельних модифікацій. Ефективність збільшується із розміром корпусу – від 3% для маленьких до 10% для надвеликих корпусів у порівнянні із використанням існуючої швидкої евристичної формули. При цьому ступінь уповільнення процесу генерування зменшується із збільшенням корпусу – від 17 до 6 відсотків на тих самих розмірах. На практиці, зменшення споживання пам'яті за рахунок збільшення «чистого» часу генерування може бути особливо корисним для систем із малою чи обмеженою кількістю доступної пам'яті, для уникнення її перевикористання.

Ключові слова: оброблення природної мови; генерування корпусів; метод CorDeGen; оптимізація використання пам'яті.

Стаття надійшла до редакції / Received 07.04.2025
Прийнята до друку / Accepted 18.04.2025

Introduction

In many professional fields today, information systems play a crucial role in solving different tasks related to natural language processing, in particular text analysis. However, from a software engineering perspective, developing and testing these systems presents unique challenges that are rarely encountered in the context of other types of information systems. One such challenge is the volume of input data for development or verification testing, as such systems are typically designed to process text corpora. Manual corpora preparation may be insufficient in terms of time and human effort if dozens or hundreds of corpora are needed, so only corpora generation "on the fly" can cover such requirements.

The task of automatically generating text corpora, considering the peculiarities of their further use when solving software engineering problems, is still poorly researched, as are the methods of solving this task

in practice. Even fewer studies are devoted to the issue of software implementations of such methods and their practical optimizations, although due to them, different implementations of the same method can differ strikingly from each other in terms of metrics such as speed or memory usage. That is why this task is still relevant.

Analysis of recent research

Even though today's literature presents a large number of studies devoted to the construction and/or generation of text corpora, mostly all of them focus only on the one-time execution of this process. In conclusion, the properties of the methods and algorithms used or proposed in such papers for the construction and/or generation of corpus make them hardly (partially or completely) applicable to solve software engineering problems.

The majority of the studies are devoted to the construction/generation of corpora based on large natural data, which are transformed and processed in a certain way, e.g. [1–5]. This requirement significantly limits the possibility of their use in developing and/or testing information systems because the initial data must be stored, managed somewhere additionally, and should be retrieved on each usage.

Other methods, like the ones presented in [6–8], do not require a large amount of input natural data to obtain a corpus, relatively small volume is enough for them. This simplifies the data storage, management, and retrieval, however, these methods have a low speed, so their use will significantly slow down the process of solving software engineering problems.

Also, determining the structure and properties of the corpus generated by all these methods (presented in [1–8]) can be difficult.

There are also methods for generating text corpora, which are specialized for solving software engineering problems during the creation of information systems. These methods consider the specific requirements imposed on them by use in this context. The CorDeGen method [9] is one of such methods and consists of the following abstract steps [9]:

1. Input the parameter N_{terms} – the number of unique terms.
2. Calculate the parameter N_{docs} (the number of texts in the corpus) using the function $f(x)$.
3. For each term i:
   a. Receive the string representation of the term.
   b. Calculate the vector $\overline{tf_i}$, containing the number of occurrences of the term in texts, using the calculation of the function $g(x)$.
   c. Record to each text the string representation of the term based on the calculated number of occurrences from the vector $\overline{tf_i}$.

The description of the abstract CorDeGen method does not specify a specific method of receiving a string representation of a term and specific functions $f(x)$ or $g(x)$, only specific requirements that should be met by them. The basic CorDeGen is defined using the abstract steps and fixates the specific instances of these three components [9].

To date, several existing studies address or mention the issue of practical optimizations of the basic CorDeGen method and its implementations. The first such paper is [10], which considers the issue of accelerating the corpus generation process by parallelizing the main cycle of the method. This is possible because each iteration of the loop (calculating the representation and occurrences of the next term) is independent of each other, so they can be effectively executed in parallel. The only issue during parallelization is the synchronization of the recording of terms to texts, depending on the method of solving which, the naive parallel and parallel CorDeGen methods are proposed in [10].

The second is [9], which mentions the issue of memory allocation optimization for texts of the generated corpus. This paper shows that a direct practical implementation of the basic CorDeGen method requires constant manual or automatic memory allocation for texts as they are generated, which also slows down generation. Instead, [9] proposes pre-allocating memory for each text "with excess", the amount of which is calculated using a fast heuristic formula (1) for the length of the text:

$$\frac{N_{terms}{N_{docs}}^2}{\left\lfloor\frac{N_{docs}}{5}\right\rfloor+2}. \quad (1)$$

However, the heuristic formula (1), although fast to calculate, has its drawbacks. Firstly, as the name of this formula implies, it is derived heuristically and there is no formal proof that the amount of memory calculated using it will be sufficient for any corpus size. This means that when using it, checks for the sufficiency of allocated memory are still required during generation.

Secondly, this formula gives the same length estimate for all texts in the corpus, although in practice, in most cases, the lengths of the texts differ. For example, if N_{terms} is increased by 1 (but the value of N_{docs}, which depends on it, remains the same), the length of all texts according to this formula will increase by the same amount, although the new term will be recorded only in a certain subset of these texts. In addition, the power dependence of the text length according to formula (1) looks like it will cause a large overestimation of the text length for large N_{terms}.

These shortcomings of the proposed stage of preliminary estimation of text length for memory allocation can be eliminated by replacing formula (1) with another one that will exactly estimate the length of each text from the generated corpus.

Formulation of the goals of the article

The aim of this paper is to improve the efficiency of software implementations of the basic CorDeGen method according to the criterion of memory usage by adding to the generation process a preliminary stage of accurate estimation of the length of each text when allocating memory for them.

To achieve this goal, the following tasks were set and solved during this study:

- Derivation of the formula for the exact length of the text by its index for a certain value of N_{terms} when using the basic CorDeGen method.
- Adding the software implementation of preliminary estimation of text length during memory allocation based on the obtained formula to existing software implementing the basic CorDeGen method; verification of this implementation.
- Experimental evaluation of the effect of using an exact prior estimation of text length during memory allocation, compared to using a fast heuristic formula.

Presentation of the main material

Text length formula

It is obvious that the length of the text with index d is obtained as the sum of the lengths of all occurrences of terms that fall into this text, as well as the lengths of the separators between these occurrences. If we consider the general case when the separator of occurrences of one term and the separator of different terms do not coincide, then we obtain formula (2), where S_d is the set of term indices that fall into the text d, osl is the length of the separator of occurrences of one term, tsl is the length of the separator of different terms:

$$dl_d = \sum_{i \in S_d} count(i, d) \times (length(i) + osl) - osl + tsl. \quad (2)$$

Formula (2) assumes that the separator of different terms is always added after each term, that is, the text ends with this separator. If a specific implementation of the basic CorDeGen method does not add the separator after the last term (i.e., the text ends with the last occurrence of the last term, not the separator), then formula (2) transforms into formula (3) in also an obvious way:

$$dl_d = -tsl + \sum_{i \in S_d} count(i, d) \times (length(i) + osl) - osl + tsl. \quad (3)$$

Also, formula (2) or formula (3) changes in an obvious way if the lengths osl and tsl coincide or these separators are the same.

The basic CorDeGen method uses the index of the term i written in hexadecimal as its string representation. Accordingly, the length of such a representation can be calculated by the formula:

$$\text{length}(i) = \lfloor log_{16}(max(i, 1)) \rfloor + 1. \quad (4)$$

Considering that many programming languages may lack a built-in function for calculating a logarithm with an arbitrary base (in the case of formula (4) – with base 16), the formula (4) can be written using a logarithm with an available base, for example, decimal:

$$\text{length}(i) = \left\lfloor \frac{log_{10}(max(i,1))}{log_{10} 16} \right\rfloor + 1.$$

The basic CorDeGen method uses the concept of a "central" text for a term with index i to determine the indices of texts in which the term occurs, and the number of these occurrences. The term i is written to the "central" text (c_i) and texts lying in the interval from $c_i - r$ to $c_i + r$ (modulo N_{docs}), with c_i and r being calculated as

$$c_i = i \, mod \, N_{docs}, r = \left\lfloor \frac{N_{docs}}{5} \right\rfloor + 1.$$

This means, that a term i is assigned to text d if and only if d is within $\pm r$ steps of c_i on the circle $\{0,1,\dots,N_{docs} - 1\}$. In modular arithmetic, that condition is

$$(d - c_i) \, mod \, N_{docs} \in \{-r,\dots,r\},$$

or, equivalently,

$$c_i \in \{d - r,\dots,d + r\} \, mod \, N_{docs}.$$

Accounting the definition of c_i, this means

$$i \, mod \, N_{docs} \in \{d - r,\dots,d + r\} \, mod \, N_{docs}.$$

So, we can define S_d from formulas (2) and (3) as:

$$S_d = \{i \in \{0,1,\dots,N_{terms} - 1\} | i \, mod \, N_{docs} \in \{d - r,\dots,d + r\} \, mod \, N_{docs}\}. \quad (5)$$

In words, this means that text d is assigned all terms i whose remainder $mod \, N_{docs}$ lies in the circular interval from $d - r$ to $d + r$. This set of "eligible" remainders can be defined as

$$R_d = \{d - r,\dots,d + r\} \, mod \, N_{docs}.$$

If some $x \in R_d$, then all integers i of the form

$$i = kN_{docs} + x,$$

for some integer $k \geq 0$ and also $i < N_{terms}$, belong to S_d. Hence, the formula (5) also can be rewritten as

$$S_d = \bigcup_{x \in R_d} \{kN_{docs} + x | k \in \mathbb{Z}, 0 \leq kN_{docs} + x < N_{terms}\}.$$

The number of term i occurrences in the text d also depends on whether that text is "central" to that term or not. To calculate this, the total number of occurrences is calculated as

$$N_{docs}(i \, mod \, N_{docs} + 1),$$

and then this total number is evenly distributed between all $2r + 1$ texts, with the one exclusion – the "central" text receives the doubled count of term occurrences. This rule can be defined as

$$\mathrm{count}(i,d) = \frac{N_{docs} \times (i \bmod N_{docs} + 1) \times (2 - sgn|d - i \bmod N_{docs}|)}{2r+2}.$$

Reference implementation and its verification

The reference implementation of the CorDeGen method family (including the basic one and its parallel modifications) was developed by its authors with the .NET platform [11] and C# programming language [12]. Accordingly, this platform and programming language were used in this study to add the length estimation step to the reference implementation.

The .NET platform and its runtime (CoreCLR) use an automatic garbage collector instead of manual memory management [13]. In this case, all "live" objects are divided into several generations (in the standard implementation of the runtime – three), and garbage collections occur independently between generations [13].

The reference implementation uses the StringBuilder class to generate texts, which is designed in the .NET platform for memory-efficient string manipulation (because strings themselves are immutable). This class is actually an implementation of a linked list over an array of characters (buffer), to which strings are added during text generation [14]. When the buffer capacity of the current node (StringBuilder instance) is exhausted, a new node is created with sufficient capacity for the value that needs to be written, and characters continue to be written to the buffer of this instance [14]. When a StringBuilder instance is created, it receives an initial buffer capacity, which is actually the received estimate of the text length. An abstraction of text length estimation was added to the reference implementation with three different instances:

- A constant implementation that always returns a value of 16 characters – this value corresponds to the default StringBuilder buffer capacity; this implementation is not intended for practical use and will only be used within the framework of this study as a baseline to compare the other two implementations.
- Implementation of fast heuristic estimation.
- Implementation of the exact estimation formula obtained in this paper.

To verify the correctness of the last implementation, a property-based testing approach was used instead of human oracle-based example tests. In the case of exact text length estimation, the only and obvious property is that the estimated value must be equal to the length of the received text for each text from the generated corpus for any value of N_{terms}. The integration tests added to the reference implementation include this property for the basic, naive parallel, and parallel implementations of the CorDeGen method, using the xUnit [15] and FsCheck [16] libraries, with the following settings: N_{terms} from 1296 to 25000, the number of property tests per run is 200. The selected range of N_{terms} values is the compromise between maximum property coverage and its execution speed.

Experiments: divergence of heuristic estimation

During this study, the divergence between the values obtained by the fast heuristic formula and the exact formula obtained in this paper was experimentally measured.

The measurement was performed for all N_{terms} values in the range from 1296 to 62500 for each text from the generated corpus. The maximum divergence between the calculated values of text lengths for each value of N_{terms} is shown in **Fig. 1**.

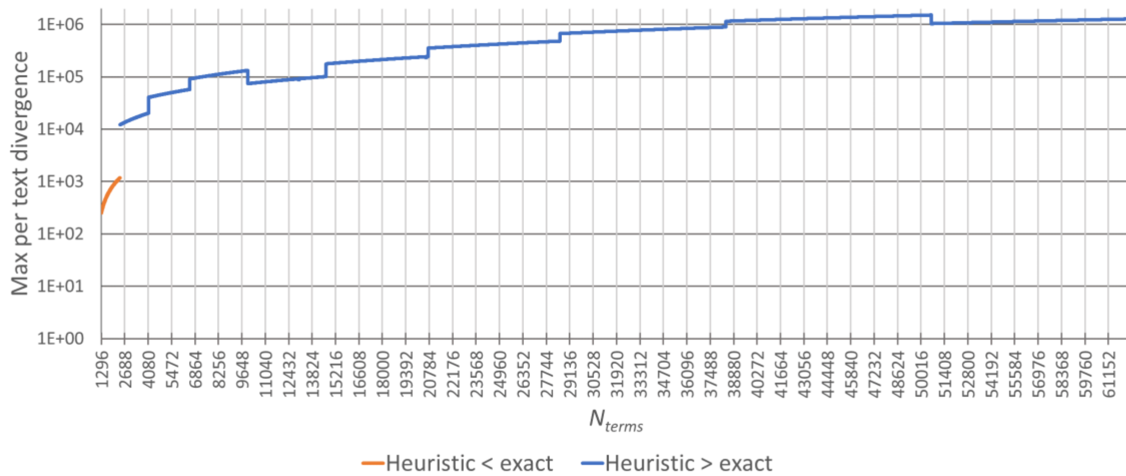

Fig. 1. Max divergence between heuristic and proposed exact estimation

As can be seen from **Fig. 1**, at all N_{terms} values from 1296 to 2400 (corresponding to six texts in the generated corpus), the values obtained by the fast heuristic formula are smaller than the exact values. In practice, this means that the initially allocated memory for the generated texts will not be enough, so additional memory will need to be allocated during the generation process.

As predicted by the main hypothesis of this study, starting with seven texts in the generated corpus (i.e. $N_{terms} > 2400$), the predictions of the fast heuristic formula begin to exceed the exact values, reaching 10^6 for a single text at large values of N_{terms}. This leads to the allocation of extra memory for each text, which will not be used, but only, for example, create unnecessary pressure on the garbage collector if it is present.

Experiments: benchmarking

In order to study the effect of usage of the formula of exact length estimation on the practical metrics (CPU, memory allocations, GC) of the reference basic CorDeGen implementation, benchmarking was performed within the framework of this study.

The BenchmarkDotNet library [17], which is a de-facto standard on the .NET platform and suggested by its developers, was used to write and run benchmarks of three developed instances of length estimation approaches, including the exact formula proposed in this paper. This library greatly simplifies the benchmarking process by automatically selecting the required number of methods call repetitions, automatically performing warm-up and jitting [18]. Also, this library automatically performs statistical processing of the obtained results, including the possibility of setting up a baseline [18]. As it was already mentioned above, the default StringBuilder capacity (16) is used as the baseline in this study.

The four N_{terms} values are used in the benchmarking process: 2500, 12500, 62500, 312500. These values correspond to the values used in other studies on the CorDeGen method topic but exclude the smallest ones (100, 500) that can be not representative.

The benchmarking was performed on a physical machine (laptop) with the following hardware: CPU with 6 physical/12 logical cores, 2.60 GHz; 16 Gb RAM, 2667 MHz. The results are shown in Table 1 and Table 2.

Table 1

Benchmarking: execution time (ms) and its ratio to the baseline

Length estimation	Mean	Median	Mean Ratio	Median Ratio
2500 unique terms				
Default capacity	1.059	1.060	–	–
Fast heuristic	1.074	1.087	1.014	1.025
Exact	1.257	1.261	1.187	1.190
12500 unique terms				
Default capacity	14.919	14.849	–	–
Fast heuristic	11.857	11.861	0.795	0.799
Exact	12.870	12.907	0.867	0.870
62500 unique terms				
Default capacity	134.632	135.754	–	–
Fast heuristic	97.305	97.970	0.723	0.722
Exact	105.983	106.005	0.787	0.781
312500 unique terms				
Default capacity	1462.355	1431.525	–	–
Fast heuristic	916.263	917.954	0.627	0.641
Exact	1008.783	1006.816	0.690	0.703

Table 2

Benchmarking: garbage collections per 1000 operations for each generation, allocated memory (MB), and its ratio to the baseline

Length estimation	Gen 0	Gen 1	Gen 2	Allocated Memory	Allocated Ratio
2500 unique terms					
Default capacity	390.6250	259.7656	85.9375	2.35	–
Fast heuristic	390.6250	216.7969	85.9375	2.38	1.013
Exact	347.6563	259.7656	173.8281	2.3	0.979
12500 unique terms					
Default capacity	3687.5000	1796.8750	890.6250	23.35	–
Fast heuristic	2468.7500	1000.0000	500.0000	23.83	1.021
Exact	2453.1250	578.1250	578.1250	23.24	0.995
62500 unique terms					
Default capacity	31500.0000	12250.0000	2750.0000	245.11	–
Fast heuristic	18666.6667	1666.6667	833.3333	257.91	1.052
Exact	18600.0000	1400.0000	1400.0000	244.65	0.998
312500 unique terms					
Default capacity	356000.0000	124000.0000	9000.0000	3023.64	–
Fast heuristic	199000.0000	11000.0000	5000.0000	3313.95	1.096
Exact	193000.0000	3000.0000	3000.0000	3019.23	0.999

As can be seen from Table 1 and Table 2, the implementation using the proposed exact length estimation formula is slower than the implementation using the fast heuristic estimation – from 6 to 17 percent of the "default" implementation, with the difference decreasing as N_{terms} increases. This is easily explained by the greater complexity of the exact formula compared to the heuristic estimate due to the need to calculate all valid term indices for the text. This introduces additional overhead into the implementation using this formula, which slows down the generation process, but it is still faster than the "default" implementation.

However, as expected, due to this overhead, the amount of memory allocated when running the implementation using the proposed exact length estimation formula is reduced up to 10 percent of the "default" implementation on large N_{terms} in the comparison with the fast heuristic estimation. At the same time, this implementation allocates less memory than even the "default" implementation (even if this difference is only a few megabytes on large N_{terms}), due to the fact that only one StringBuilder node is created, instead of many additional nodes created when the "default" implementation works. This also significantly reduces the number of garbage collections across all generations.

Similar results were also obtained when benchmarking the naive parallel and parallel versions of the basic CorDeGen method.

Conclusions

This study shows the feasibility of improving existing methods of generating text corpora intended for use in solving software engineering tasks in the context of natural language processing information systems. One of the points for such improvements is the practical metrics of implementations of such methods, in particular, memory usage. The basic CorDeGen method was chosen as the subject of this study, the analysis of which showed the possibility of optimizing memory usage by implementations of this method, due to the insufficient study of this issue in the literature to date.

To exactly estimate the length of a text by its index depending on the input value N_{terms} (and, accordingly, the amount of memory needed to be allocated for this text), this study derived a formula that takes into account: the set of terms included in this text (S_d); the lengths of these terms (length(i)); the number of their occurrences (count(i, d)); the lengths of the separators between terms (osl and tsl).

The existing reference implementation of the basic CorDeGen method and its modifications was modified in order to experimentally verify the effectiveness of using the proposed formula for exact text length estimation. To do this, a new abstraction was introduced in the implementation, which is responsible for the preliminary estimation of the length for allocating memory for each text, and several instances of this abstraction were added: by default, using the existing fast heuristic formula, and using the proposed exact formula. The implementation of the exact formula was further verified for correctness using property-based tests.

Experimental verification using a modified reference implementation confirmed the achievement of the main goal of this study – using the proposed exact estimation formula on large corpora allows reducing the amount of memory allocated by 10% compared to the fast heuristic formula. This result is achieved by increasing the generation time by 6% for the same corpus sizes. A small reduction in speed in exchange for reduced memory consumption can be especially critical for systems with small or limited memory.

The results presented in this paper can be further developed in the form of adapting the obtained exact estimation formula to other modifications of the CorDeGen method, for example, CorDeGen+ or DBCorDeGen.

References

1. Recski G., Iklódi E., Lellmann B., "BRISE-plandok: a German legal corpus of building regulations," Language Resources and Evaluation, (2024). https://doi.org/10.1007/s10579-024-09747-7.
2. Arhar Holdt Š., Kosem I., "Šolar, the developmental corpus of Slovene," Language Resources and Evaluation, (2024). https://doi.org/10.1007/s10579-024-09758-4.
3. Vitório D., Souza E., Martins L., "Building a relevance feedback corpus for legal information retrieval in the real-case scenario of the Brazilian Chamber of Deputies," Language Resources and Evaluation, (2024). https://doi.org/10.1007/s10579-024-09767-3.
4. Rakotomalala F., Hajalalaina A. R., Ravonimanantsoa Ndaohialy M. V., "FLICs (Facebook Language Informal Corpus): a novel dataset for informal language," International Journal of Data Science and Analytics, vol. 18, pp. 393-403, (2024). https://doi.org/10.1007/s41060-023-00460-2.
5. Lichtarge J., "Corpora generation for grammatical error correction," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (2019). https://doi.org/10.18653/v1/N19-1333.
6. Nazura J., Muralidhara B. L., "Automating Corpora Generation with Semantic Cleaning and Tagging of Tweets for Multi-dimensional Social Media Analytics," International Journal of Computer Applications, vol. 127, no. 12, pp. 11-16, (2015). https://doi.org/10.5120/ijca2015906548.

7. Alberti C., Andor D., Pitler E., Devlin J., Collins M., "Synthetic QA Corpora Generation with Roundtrip Consistency," in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, (2019). https://doi.org/10.18653/v1/P19-1620.

8. Boujelbane R., Ellouze Khemekhem M., Belguith L., "Mapping Rules for Building a Tunisian Dialect Lexicon and Generating Corpora," in Proceedings of the Sixth International Joint Conference on Natural Language Processing, (2013). https://aclanthology.org/I13-1048/

9. Yusyn Y., "Methods and software tools for metamorphic testing of software systems for automatic clustering of natural language text data," PhD thesis, Kyiv, (2022) [in Ukrainian].

10. Yusyn Y., Zabolotnia T., "Accelerating the process of text data corpora generation by the deterministic method," Eastern-European Journal of Enterprise Technologies, vol. 1, no. 2 (127), pp. 26-34, (2024). https://doi.org/10.15587/1729-4061.2024.298670.

11. Microsoft, "What's new in .NET 8," (2023). [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview.

12. Microsoft, "What's new in C# 12," (2023). [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-12.

13. Kokosa K., "Pro .NET Memory Management: For Better Code, Performance, and Scalability (1st Edition)", Apress, (2018).

14. Lock A., "Series: A deep dive on StringBuilder," (2021). [Online]. Available: https://andrewlock.net/series/a-deep-dive-on-stringbuilder/.

15. .NET Foundation and contributors, "Home > xUnit.net," (2019). [Online]. Available: https://xunit.net/.

16. Aichernig B., Schumi R., "Property-based Testing with FsCheck by Deriving Properties from Business Rule Models," in In 2016 IEEE Ninth International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 13th Workshop on Advances in Model Based Testing (A-MOST 2016), (2016).

17. NET Foundation and contributors, "Overview | BenchmarkDotNet," (2018). [Online]. Available: https://benchmarkdotnet.org/articles/overview.html.

18. Akinshin A., "Pro .NET Benchmarking: The Art of Performance Measurement", Apress, (2019).