Next Word Generation Using Genetic Programming

Tzu-Chien Lin
Institute of Information Systems and Applications
NTHU
112065517

Yu-Ching Chen
Department of Computer Science
NTHU
112062527

***Abstract*—In this research endeavor, we delve into the utilization of evolutionary-based methodologies, with a particular focus on Genetic Programming (GP), to tackle the "next word prediction" challenge within the realm of Natural Language Processing (NLP). This challenge revolves around generating coherent and contextually relevant continuations in text generation tasks, a pivotal aspect of NLP systems. Through the application of GP, our objective is to enhance the effectiveness and efficiency of text generation by evolving ensemble strategies. First, we use three embedding methods—Word2Vec, GloVe, and fastText—to train GP models. We discover that the GP model with fastText embeddings demonstrates the most robust performance. Subsequently, we employ random forest and AdaBoost in combination with GP to improve performance. However, due to the relatively small population size and limited number of evolutionary iterations, the results of these ensemble methods are weaker than those of GP models using non-ensemble methods, though they undergo a more comprehensive evaluation.**

***Index Terms*—Genetic programming, Natural language processing, Next word prediction**

I. Motivation, Goal, and Challenges

Recently, Large Language Models (LLMs), such as ChatGPT, have gained immense popularity for its ability to provide responses of consistently high quality to a wide range of natural language processing (NLP) queries. This underscores the efficacy of LLMs and neural networks. It prompts us to consider whether evolutionary computation, a subfield of artificial intelligence, can achieve similar prominence. Within evolutionary computation, Genetic Programming (GP) stands out as one of the prominent algorithms utilized for solving machine learning problems.

Given the limited exploration of NLP problems using GP and our current capabilities, this paper does not aim to replicate the performance of ChatGPT. Instead, our primary objective is to next-word prediction, predicting the subsequent word following a given sequence. We follow the approach of Manzoni et al. [4] to design the architecture of our GP system. Additionally, we compare the effects of different dataset production methods and employ ensemble techniques, specifically random forest and Adaboost, to enhance GP performance.

During the experiment, we found that combining GPs with different word embeddings generated from various embedding models using ensemble methods is challenging. Since the data varies with the embedding model, it is difficult to crossover two GP trees with different data and calculate fitness values. Therefore, the base learners of the ensemble are all trained with data transformed from the same embedding models. Beyond data differences, implementing random forest and AdaBoost combined with GP also presents difficulties. The specifics of random forest and AdaBoost require thorough consideration since the training methods for decision trees and GP trees differ. Finally, implementing the system in Python resulted in longer-than-expected training times for the GP, preventing us from testing the effect of different GP operator values on performance.

The source code of our implementation is publicly available at https://github.com/atwolin/EC-term-paper. Note that you need to train embedding models first.

II. Related Work

A. Next-word prediction

Our objective, the next-word prediction problem, a.k.a. language modeling, falls within the domain of Natural Language Generation (NLG). NLG is a field focused on devising methodologies to generate natural language from computerized representations of information [1].

Kim et al. [2] developed a GP-based system to respond to user queries. Initially, the system extracts keywords from the query and matches them with predefined answer scripts. Subsequently, using GP, the system generates responses. Kim et al. employed grammars for constructing GP trees, whereas Lin and Cho [3] introduced a GP conversational system utilizing binary trees, termed sentence plan trees, to streamline processing time. Internal nodes of sentence plan trees are annotated with joint operators, while leaves are labeled with predefined templates of simple sentences. In a subsequent study, Manzoni et al. combined GP with word embeddings generated by word2vec. The set of functionals for internal nodes comprises mathematical symbols like summation, subtraction, etc., while the set of terminals consists of initial words.

B. Word Representation

Word representations are vectors that enable machines to understand and interpret natural language words by describing their meanings through their distributions. These representations are also known as word embeddings. There are three types of word representations: static-word embeddings, sense-aware embeddings, and contextualized embeddings. [6] Static-word embeddings, generated by training models such as Word2Vec, GloVe, and fastText, produce static word types.

Sense-aware embeddings also generate static word representations but address the meaning conflation problems faced by static-word embeddings. Contextualized embeddings, on the other hand, dynamically generate representations based on the context in which a new word appears. We use three static word embeddings—Word2Vec, GloVe, and fastText—as our embedding methods.

C. Ensemble

Ensemble learning is an approach that combines multiple weak learners to create a strong learning model with low bias and high variance. This technique produces more robust and powerful predictions. The three main ensemble methods are bootstrap aggregating (bagging), boosting, and stacking. Among numerous ensemble methods, random forest and AdaBoost are particularly powerful. Random forest consists of trees created by sampling from the training dataset, while AdaBoost builds trees by updating the weights of the training data. Sašo Karakatič and Vili Podgorelec [14] designed a structure combining GP and AdaBoost, known as Genetic Programming with AdaBoost (GPAB), for classification problems, demonstrating improved classification accuracy. In this paper, we follow the GPAB structure and modify it to apply GP trees to regression problems. In addition, we also select random forest to combine with GP as an ensemble method. We name these ensemble methods as ensemble GPs.

III. Problem Formulation

Our objective is to leverage genetic programming (GP) within the field of evolutionary computation to address the "next-word prediction" issue in natural language processing (NLP). This problem can be divided into four key components:

$InitialState$ Represented by vectors corresponding to the first words of a sentence within the chosen embedding.

$SuccessorFunction$ Determines the most probable words based on the current context of the text.

$GoalTesting$ Evaluates the consistency and relevance between the generated prediction vocabulary and the ground truth text.

$PathCost$ Measures the expenditure required to transition from the initial state to the goal state, encompassing operational costs or other relevant metrics. Path cost can be quantified by comparing the disparity between the generated prediction vocabulary and the actual text.

Together, these components form a framework for addressing the next-word prediction problem.

During the search phase, we define the prediction model's search space, which includes the vocabulary list and potential word sequence combinations. We employ GP-based algorithms to identify optimal solutions.

In the execution phase, we train the model using arithmetic operators combined with word embeddings to determine the most effective GP structure for predicting the next word. Subsequently, we evaluate the model's accuracy by using the cosine similarity between the output vector and the target vector as our model's successor function. Based on the analysis outcomes, under-performing GP trees will be replaced to enhance the model's overall performance and efficiency. Finally, we select the GP trees with the best successor function as our final models.

IV. Evolutionary Algorithm

We adopt the framework of the GP system proposed by Manzoni et al. as our baseline, modifying their settings to implement our GP system (details shown in Table 1). While Manzoni et al. use Word2Vec embeddings, we add two other embedding methods: GloVe and fastText. We then adjust different crossover methods to investigate their effects, naming these models as simple GP.

For ensemble GPs, we train a random forest GP by sampling 80% of the original dataset into 20 groups. For each group, we select the best 10 individuals to be candidates for the ensemble, and finally, we pick the best 5 individuals from those candidates. For AdaBoost GP, we combine GPAB and the AdaBoostRegressor from scikit-learn. In each generation, AdaBoost GP checks if it encounters the boosting interval; if so, it selects the best individual and updates its data weights. After evolution is complete, AdaBoost GP also selects the best 5 individuals as predictors. Both the random forest GP and AdaBoost GP output the final prediction vector by averaging the results of the best 5 individuals. Pseudo code for these two algorithms is listed as Algorithm 1 and Algorithm 2. Following this, we explain the implementation of simple GP, random forest GP, and Adaboost GP, respectively.

In the following sentences, we will refer to the GP models with Word2Vec embeddings as GP-Word2Vec, GP models with GloVe embeddings as GP-GloVe, and GP models with fastText embeddings as GP-fastText.

A. Simple GP

Firstly, we attempted to integrate two Python libraries related to GP, namely DEAP and Geppy. The Geppy library, built on the foundation of DEAP, offers several advantages in handling GP expression trees and evolving the whole process. With straightforward built-in functions, Geppy includes tools for creating elements such as chromosomes or genes, facilitating selection and crossover operations more efficiently.

However, after a short period, we noticed that the convenience came at the cost of flexibility and extensiveness. Implementing specific operations for expression trees, such as limiting tree height, became challenging using Geppy's type-creation methods. In contrast, the DEAP library, which serves as the foundational layer for Geppy, provides a simpler and more flexible way of handling such operations. Additionally, since Geppy is a variation of GP, it is not a suitable API for our needs.

We subsequently turned to DEAP, and since Geppy's structure is built on DEAP, the transition did not require significant time or effort. The official documentation and source code of DEAP provided a comprehensive understanding of how functions are constructed throughout the evolutionary process. This clarity allowed us to effectively implement our main

TABLE I: Implementation for simple GP and ensemble GPs

Representation	Tree structures; Function set = {sum, subtraction, multiplication, protected division, squaring, square root}; Terminal set = first 5 word embeddings of sentences
Population	sampling one-hundredth of a random shuffle of the 266,920 headlines with six words, i.e. 2,669 sentences ** for simple GP; 10, 15, 20, 25, 50, 100 for random forest GP; 100 for Adaboost GP (all dimensions are 10)
Crossover	Simple subtree crossover for simple GP; Simple subtree, uniform, size fair, one-point crossover, randomly selected at each generation for ensemble GPs
Probability of crossover	** for simple GP; 1 for random forest GP; 1 for Adaboost GP
Mutation	Simple subtree mutation
Probability of mutation	** for simple GP; 0.3 for random forest GP; 0.1 for Adaboost GP
Parent selection	Tournament selection, size = 3, two candidates with the highest fitness are the parents
Survivor selection	Steady-state breeding strategy; Resulting child tree replaces the worst individual in the tournament
Termination	Maximum evaluation = 20000 fitness evaluations for simple GP; Maximum evaluation = 10,000 and 100 for random forest GP; Maximum generation = 100 for Adaboost GP
Number of runs	10 for simple GP; 7 for random forest GP; 1 for Adaboost GP
Specialty	Maximum depth of trees = 5

genetic programming evolution function with the help of DEAP's built-in functions.

When it came to crossover and evaluation, the provided functions did not meet our expectations or needs. Our plan included four crossover methods; however, DEAP only offered two built-in crossover functions, and only one of them had the operational effect we desired. This was one of the first major issues we encountered: how the data was structured within the expression tree and the challenges of exchanging tree nodes with conflicting attributes.

B. Random Forest GP

Random forest with GP has been extensively researched, with some studies using GP to produce decision trees. Initially, we referenced this type of approach [7] and invested considerable time modifying and implementing the code to align with our objectives. However, the differing logic made it difficult to achieve our goal of replacing decision trees with GP and executing evolutions. Consequently, we opted for a simpler design to build the random forest GP.

C. Adaboost GP

Our AdaBoost GP is a combination of GPAB and the AdaBoostRegressor from scikit-learn, as GPAB primarily addresses classification problems. To obtain more reliable models, we referred to scikit-learn's algorithm and made some modifications. Consequently, the boosting component in our source code closely resembles that of scikit-learn's implementation.

Algorithm 1 Pseudo Code for Random Forest GP

Require: Evaluation limits as *evalLimit*
Require: Number of groups with the same random sampling data as *numPick*
Ensure: Trees for ensemble as *ensemble*

```
evaluation ← 0
candidate ← empty list
while evaluation ≤ evalLimit do
    for i = 1 to numPick do
        population ← randomly generated with subdata
        population.evolution
    end for
    candidate ← candidate + population.topTen
end while
ensemble ← candidate.topFive
return ensemble =0
```

Algorithm 2 Pseudo Code for Adaboost GP

Require: Generation limits as *generationLimit*
Require: Boosting interval as *boostingInterval*
Ensure: Trees for ensemble as *ensemble*

```
population ← randomly generated
candidate ← None
generation ← 0
while generation ≤ generationLimit do
    population.evolution
    if generation % boostingInterval == 0 then
        sample_weight ← boosting_sklearn
        if sample_weight is None then
            sample_weight ← original sample weight
            continue
        end if
        candidate ← candidate + population.topTen
        population.update_data_weight
    end if
    generation ← generation + 1
end while
ensemble ← candidate.topFive
return ensemble =0
```

V. Results

A. Training Phase

1) Simple GP: In Figures 1, 2, and 3, we compare our GP models using five crossover methods: simple subtree crossover, uniform crossover, fair crossover, and one-point crossover. The results show that GP-fastText models exhibit the highest fitness values and robust convergence compared to GP-Word2Vec and GP-GloVe models. GP-fastText models

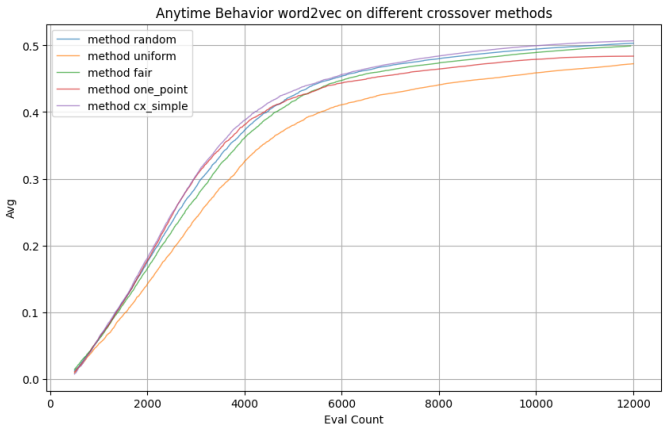

Fig. 1: Anytime behavior (average over 3 trials) of baseline models with Word2Vec embeddings for evaluation = 12,000

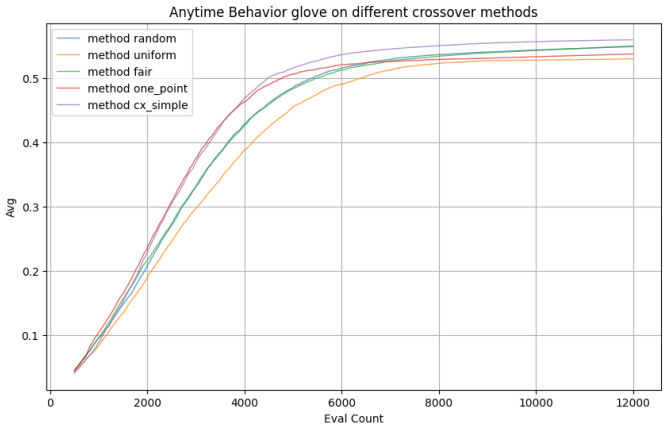

Fig. 2: Anytime behavior (average over 3 trails) of baseline models with GloVe embeddings for evaluation = 12,000

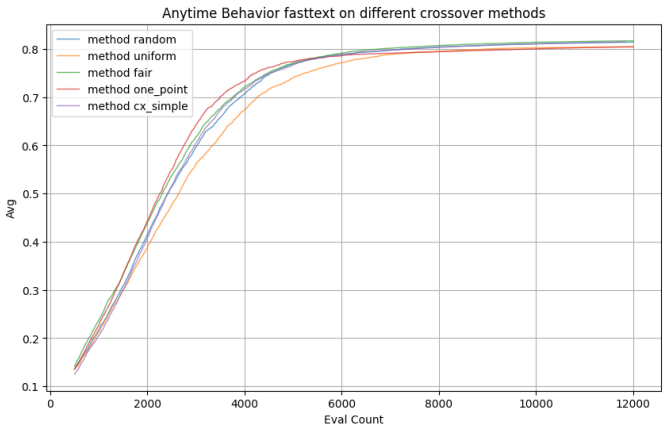

Fig. 3: Anytime behavior (average over 3 trails) of baseline models with fastText embeddings for evaluation = 12,000

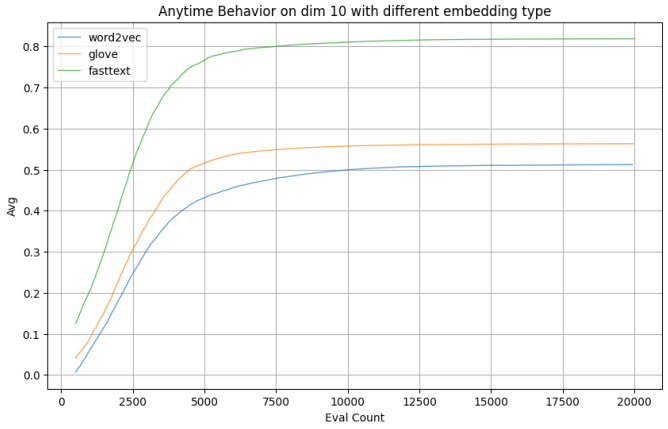

Fig. 4: Anytime behavior (average over 2 trails) of baseline models embedding for dimension = 10

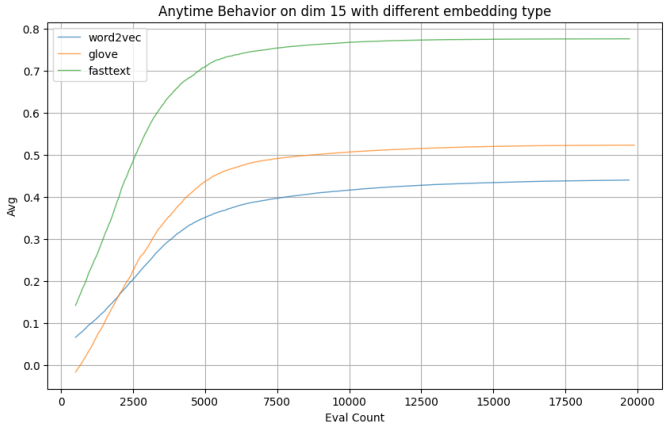

Fig. 5: Anytime behavior (average over 2 trails) of baseline models embedding for dimension = 15

can achieve fitness values up to 0.8, whereas GP-Word2Vec and GP-GloVe models remain below 0.6, with GP-Word2Vec models performing the poorest.

Among the five crossover methods, uniform crossover has the worst performance, characterized by relatively low solution quality and slow convergence speed. This may be due to the infrequent overlap of common regions, which require the same arity between two trees, resulting in fewer crossover executions. In contrast, one-point crossover demonstrates the greatest convergence speed across all three types of GP models. It provides stable performance because the common region requirements are less stringent than those of uniform crossover. Thus, modifying the definition of the common region for uniform crossover could potentially yield better results. Figures 4, 5, and 6 demonstrate the differing behaviors of three embedding models—fastText, GloVe, and Word2Vec—across embedding dimensions of 10, 15, and 20.

We can clearly observe that the evolving performance using the fastText model shows a significant improvement compared to both the GloVe and Word2Vec models, regardless

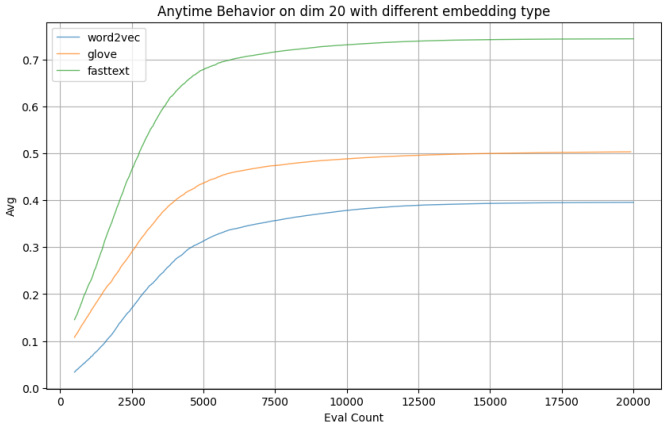

Fig. 6: Anytime behavior (average over 2 trails) of baseline models embedding for dimension = 20

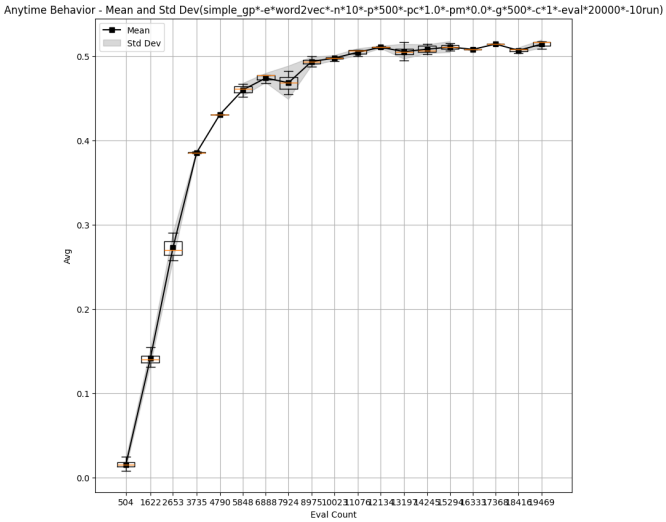

Fig. 7: Anytime behavior (average over 10 trails) of simple GP with 10-dimension Word2Vec embeddings for evaluation = 20,000

of the embedding dimension. This observation aligns with our main experiment results. Across the three different embedding dimensions, fastText achieved performance levels around 0.75 to 0.82, GloVe ranged from 0.5 to 0.56, and Word2Vec ranged from 0.4 to 0.5.

Throughout the comparison of the three dimension sizes, we noticed a slight performance drop each time we increased the dimension size, regardless of the model used. In a relevant study, Manzoni et al. combined genetic programming with word embeddings generated by Word2Vec, and they also observed the same behavior: lower embedding dimensions lead to better performance.

Given the large number of 100,000 evaluation counts, this experiment became time-consuming. Considering the limited time we had, we ran 100,000 evaluations a few times and observed that the growth rate did not change significantly after 15,000 evaluations. Therefore, we decided to set the termi-

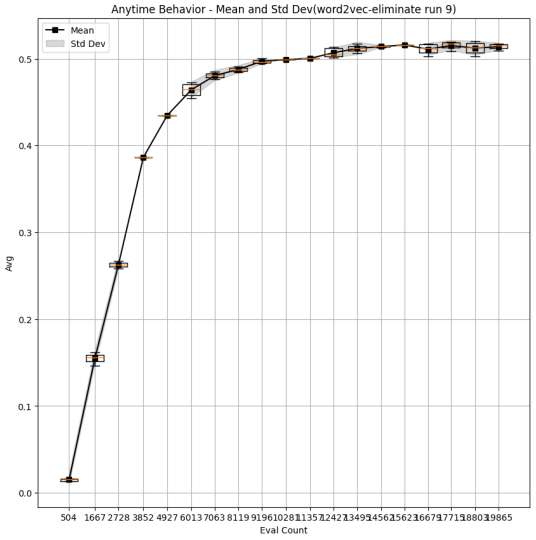

Fig. 8: Anytime behavior (average over 9 trails, excluding one outlier) of simple GP with 10-dimension Word2Vec embeddings for evaluation = 20,000

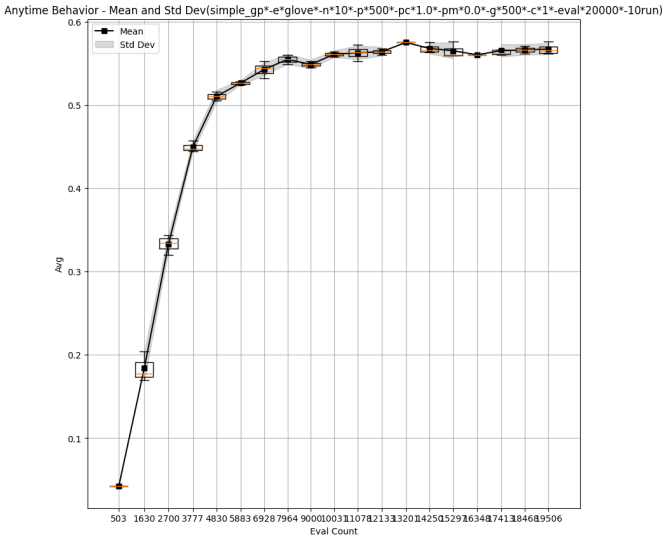

Fig. 9: Anytime behavior (average over 10 trails) of simple GP with 10-dimension GloVe embeddings for evaluation = 20,000

nation condition at 20,000 evaluations to better understand the behavior of different embedding models and parameter settings.

Figures 7 (Word2Vec), 9 (GloVe), and 10 (fastText) demonstrate the anytime behavior of Simple GP over 20,000 evaluation counts. The growth slope becomes smoother after 14,000 evaluation counts, especially with the fastText embedding model. However, we observe that with Word2Vec and GloVe, the growth behavior is more fluctuating, as indicated by the changing box plots and the up-and-down patterns shown on the line.

This observation provides more convincing evidence that the fastText embedding model outperformed both GloVe and

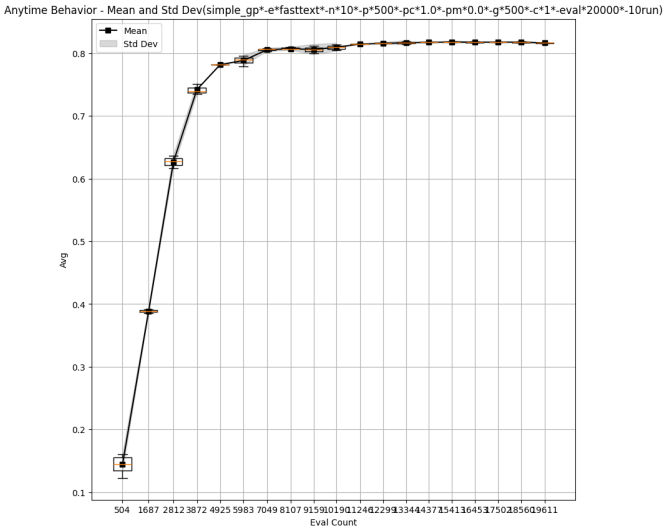

Fig. 10: Anytime behavior (average over 10 trails) of simple GP with 10-dimension fastText embeddings for evaluation = 20,000

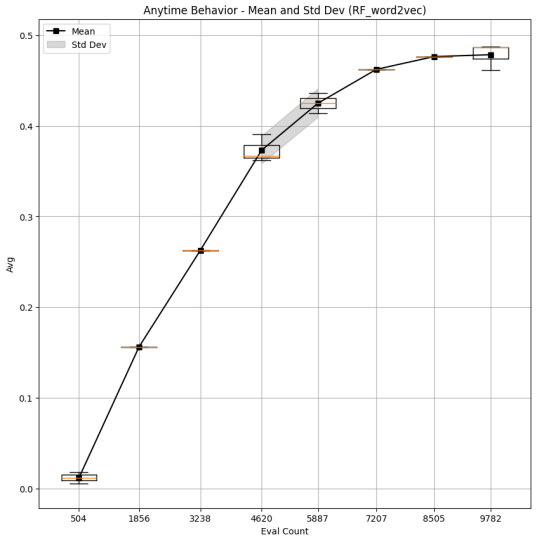

Fig. 11: Anytime behavior (average over 7 trails) of random forest GP with Word2Vec embedding for evaluation = 10,000

Word2Vec during this Simple GP experiment. This conclusion is supported not only by performance metrics but also by the speed of convergence.

In Figure 7, the Word2Vec model shows more unstable behavior around the 9,000th evaluation count. Upon investigating the data, we discovered that this instability was caused by the 9th run. Considering that genetic programming algorithms are based on randomness, we excluded the data from the 9th run and plotted another graph, shown in Figure 8. As we can see, the growth line of Word2Vec becomes much calmer and more stable.

2) Random Forest GP: Figure 11 shows the anytime behavior of random forest GP with 10-dimension Word2Vec

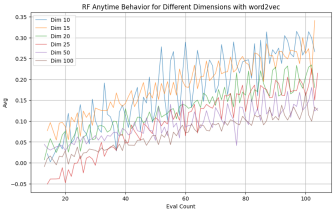

Fig. 12: Anytime behavior (average over 30 trails) of random forest GP with different Word2Vec-embedding dimension for evaluation = 100

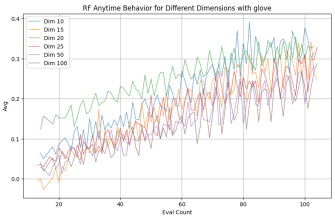

Fig. 13: Anytime behavior (average over 30 trails) of random forest GP with different GloVe-embedding dimension for evaluation = 100

embeddings for 10,000 evaluation times. Figures 11, 12, and 13 demonstrate the differing behaviors of three embedding models—fastText, GloVe, and Word2Vec—across embedding dimensions of 10, 15, 20, 25, 50, and 100. These figures indicate that the random forest has not yet converged; however, due to time constraints, we had to halt the experiments. In Figures 12, 13, and 14, we can observe consistent results: lower embedding dimensions lead to better performance, and GP models with fastText embeddings achieve the best average fitness values.

3) Adaboost GP: From Figure 15, we can observe that from the beginning, the Adaboost GP model with fastText embeddings consistently exhibits significantly higher fitness values compared to Adaboost GP models with Word2Vec and GloVe embeddings. This performance trend aligns with the results obtained from the simple GP experiments. However, it's notable that the Adaboost GP models have not yet converged. Similar to the random forest GP, Adaboost GP can potentially achieve convergence by increasing the number of generations.

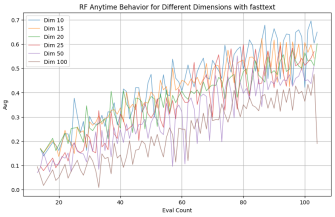

Fig. 14: Anytime behavior (average over 30 trails) of random forest GP with different fastText-embedding dimension for evaluation = 100

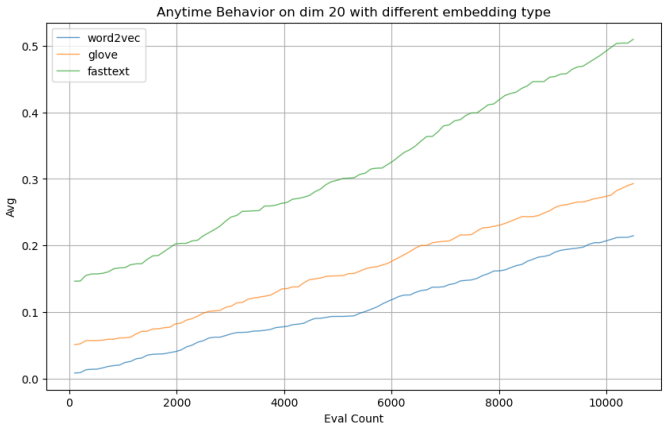

Fig. 15: Anytime behavior (average over 1 trails) of Adaboost GP with different embedding type for generation = 100

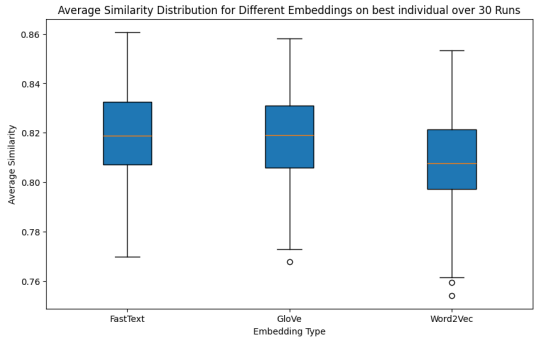

Fig. 16: Average similarity distribution for different embeddings on simple GP over 30 runs

B. Testing Phase

1) Simple GP: During the testing phase, we selected the best individual from all the runs (as shown in the table below) and applied it to the testing data. We collected all the generated results over 30 trials and recorded the average distribution, as depicted in Figure 16. Although the performance varied slightly across trials, all three embedding models achieved a similarity score of around 0.8. This result was surprisingly good, especially considering that the best individuals collected from GloVe and Word2Vec had fitness values of around 0.55 on the training data.

The detailed results are as follows:

TABLE II: Best individual from simple GP records selected from experiment shown on Fig 7 ,9 ,and 10

Embedding Type	Fitness Value	best Individual Expression
fastText	0.819969677554395	(e + (d + (c + ((e + a) + b))))
GloVe	0.574898279866091	(((e + (a + a)) + e) + (((d - a) + (e + b)) + c))
Word2Vec	0.510411650871976	((((b + a) + (e + c)) + a) + d)

2) Random Forest: For random forest GP with Word2Vec embeddings, we selected the best five individuals from all the runs and applied them to the testing data. We collected all

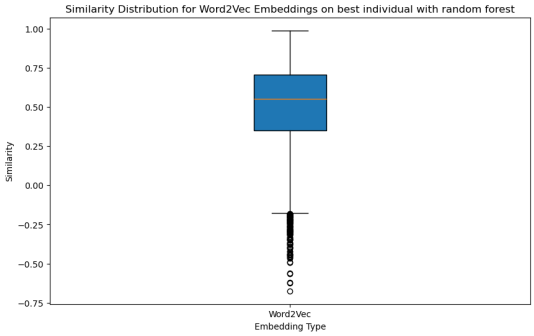

Fig. 17: Average similarity distribution for GP-Word2Vec over 7 runs

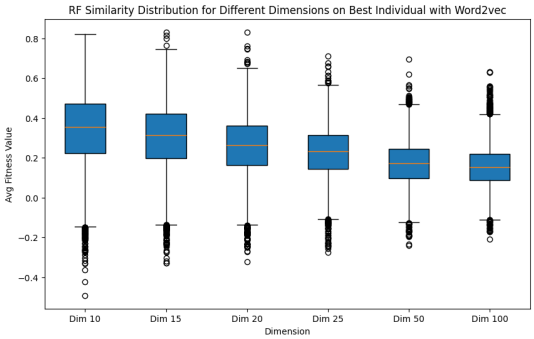

Fig. 18: Similarity distribution for different GP-Word2Vec dimensions

the generated results over 7 trials and recorded the average distribution, as shown in Figure 17. The performance was worse than simple GP with the same embedding model. This may be due to the random forest GP's evolution not fully converging, or there may be an issue with the step of selecting candidates.

Figures 18, 19, and 20 display distributions for three embedding types—Word2Vec, GloVe, and fastText, respectively—with different embedding dimensions. GP models with Word2Vec embeddings and GP models with fastText embeddings exhibit similar results to those observed by Manzoni

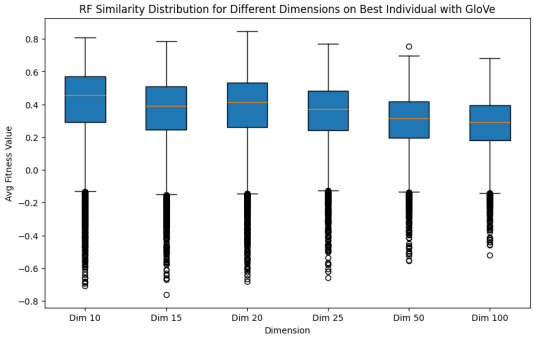

Fig. 19: Similarity distribution for different GP-GloVe dimensions

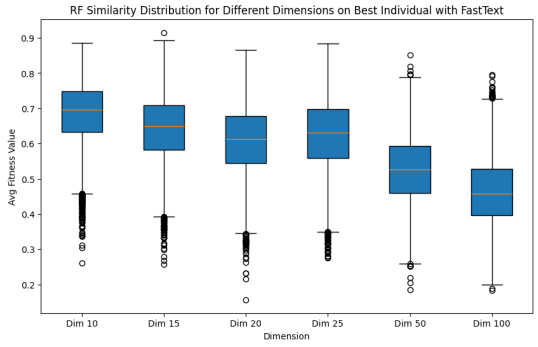

Fig. 20: Similarity distribution for different GP-fastText dimensions

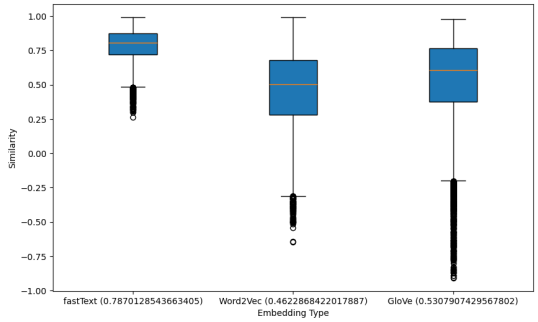

Fig. 21: Similarity Distribution for different embeddings on Adaboost GP

et al., wherein lower dimensions lead to better performance and outliers are arranged almost symmetrically. In contrast, GP models with different embedding dimensions of GloVe embeddings perform similarly, with their outliers distributed below the mean of fitness values.

3) Adaboost GP: Figure 21 demonstrates distributions for Adaboost GP models with three types of embeddings for 100 generations. While the results of these models do not surpass the results of simple GP, Adaboost GP models with fastText embeddings have the highest average fitness value, followed by the models with GloVe embeddings, with the lowest being the models with Word2Vec embeddings.

VI. Conclusion

During the experiments, the simple GP with baseline parameters did generate the expected result, with lower embedding dimensions resulting in higher performance. However, in the training results provided earlier, both Word2Vec and GloVe showed a significant improvement in fitness values for the best individual testing results, while fastText remained the best but with not much improvement from the average fitness.

For ensemble GPs, since the number of evolutions and runs were too small to compare with simple GP, the models with three different types of embeddings showed the same trend: models with fastText embeddings had the best performance, followed by models with GloVe embeddings, with the least being models with Word2Vec embeddings.

Throughout this paper, the experiment faced some implementation issues. Firstly, the restriction on tree height was not followed during the evolution process, possibly due to the self-implemented crossover. This aspect should be re-implemented in the future. Additionally, the selection mechanism of candidates and archive for ensemble needs to be reviewed and revised. Furthermore, understanding why fastText outperforms both Word2Vec and GloVe on all types of GPs—simple GP, random forest GP, and Adaboost GP—would be an interesting topic for further investigation. The potential reason might be the higher quality of data in fastText embeddings, but this requires further research. With more sufficient evolution times, training, and testing data over longer evaluation periods, we might obtain a more comprehensive understanding of the results.

References

[1] Araujo, L. Genetic programming for natural language processing. Genet Program Evolvable Mach 21, 11–32 (2020). https://doi.org/10.1007/s10710-019-09361-5

[2] Kim, KM., Lim, SS., Cho, SB. (2004). User Adaptive Answers Generation for Conversational Agent Using Genetic Programming. In: Yang, Z.R., Yin, H., Everson, R.M. (eds) Intelligent Data Engineering and Automated Learning – IDEAL 2004. IDEAL 2004. Lecture Notes in Computer Science, vol 3177. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-28651-6_121.

[3] Lim, S., Cho, SB. (2005). Language Generation for Conversational Agent by Evolution of Plan Trees with Genetic Programming. In: Torra, V., Narukawa, Y., Miyamoto, S. (eds) Modeling Decisions for Artificial Intelligence. MDAI 2005. Lecture Notes in Computer Science(), vol 3558. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11526018_30.

[4] Luca Manzoni, Domagoj Jakobovic, Luca Mariot, Stjepan Picek, and Mauro Castelli. 2020. "Towards an evolutionary-based approach for natural language processing. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20). Association for Computing Machinery, New York, NY, USA, 985–993. https://doi.org/10.1145/3377930.3390248.

[5] Sašo Karakatič and Vili Podgorelec. 2018. Building boosted classification tree ensemble with genetic programming. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '18). Association for Computing Machinery, New York, NY, USA, 165–166. https://doi.org/10.1145/3205651.3205774.

[6] Marianna Apidianaki; From Word Types to Tokens and Back: A Survey of Approaches to Word Meaning Representation and Interpretation. Computational Linguistics 2023; 49 (2): 465–523. https://doi.org/10.1162/coli_a_00474.

[7] H. Zhang, A. Zhou and H. Zhang, "An Evolutionary Forest for Regression," in IEEE Transactions on Evolutionary Computation, vol. 26, no. 4, pp. 735-749, Aug. 2022, doi: 10.1109/TEVC.2021.3136667.

[8] Zhou, Z. et al. (2023). A Boosting Approach to Constructing an Ensemble Stack. In: Pappa, G., Giacobini, M., Vasicek, Z. (eds) Genetic Programming. EuroGP 2023. Lecture Notes in Computer Science, vol 13986. Springer, Cham. https://doi.org/10.1007/978-3-031-29573-7_9.

[9] R. Poli, W. B. Langdon, and N. F. McPhee. A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. (With contributions by J. R. Koza). GPBiB

[10] 2002. Foundations of genetic programming. Springer-Verlag, Berlin, Heidelberg.

[11] Banzhaf, Wolfgang Nordin, Peter Keller, Robert Francone, Frank. (1998). Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications.

[12] R. Poli and W. B. Langdon, "Schema Theory for Genetic Programming with One-Point Crossover and Point Mutation," in Evolutionary Computation, vol. 6, no. 3, pp. 231-252, Sept. 1998, doi: 10.1162/evco.1998.6.3.231.

[13] P. D'haeseleer, "Context preserving crossover in genetic programming," Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, Orlando, FL, USA, 1994, pp. 256-261 vol.1, doi: 10.1109/ICEC.1994.350006.
[14] The sentences have been revised by ChatGPT.