UNIVERSITY OF TRIESTE
DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

Project Exam December 18, 2023

Global and Multiobjective Optimization Project Report: Integration of Genetic Algorithms and Language Models

by

Luca Pernice

Summary

When utilizing our large language model for specific tasks, its performance isn't solely dependent on the model itself but also on the provided prompt. Consequently, multiple attempts might be necessary to discover a prompt yielding a satisfactory solution. Alternatively, implementing a genetic algorithm could facilitate this process by evolving initial prompts. Through automated iterations, this algorithm can eventually yield the most effective prompt discovered. The first segment of this report focuses on a fundamental implementation of this concept. The latter part delves into a more advanced approach wherein the language model is leveraged to evolve Python code for the creation of new neural network architectures.

1 EvoPrompt

Reference paper [1]: Connecting Large Language Models with Evolutionary Algorithms Yields Powerful Prompt Optimizers

Colab: `https://colab.research.google.com/drive/1kWnaktaJOKvtNjuH8dUkaZkbXOAXDDYV?usp=sharing`

When working with a large language model for specific tasks, fine-tuning the model becomes imperative for improved results. However, this process can be notably costly. An alternative approach involves retaining the parameters of Large Language Models (LLMs) while employing continuous prompt tuning methods. Yet, in certain cases—such as when utilizing GPT-4—accessing model parameters is impossible.

Considering this limitation, employing Genetic Algorithms (GAs) to evolve discrete prompts emerges as a viable solution. This approach avoids involvement with the model's parameters, focusing solely on crafting sentences for prompts. Despite its apparent suitability, evolving natural language expressions presents challenges. For instance, ensuring human-comprehensibility and coherence after a random crossover between two expressions is not straightforward.

Evolutionary Algorithms (EAs) operators commonly struggle to grasp token relations as they are predominantly designed for sequences. However, the solution lies within the LLMs themselves, as they are specifically engineered to comprehend and generate natural language. The proposed strategy involves a synergy between LLMs and EAs, where language models generate new prompts and execute crossover and mutation operations, while EAs guide the optimization process. This collaboration aims to harness the strengths of both methodologies for enhanced efficiency in prompt evolution.

To conduct all the computations required for the evolutionary cycle without the need for a powerful GPU, the code is executed in a Colab Notebook, leveraging the Stable Beluga 2 [2] language model hosted by Petals [3] (`https://github.com/bigscience-workshop/petals`), a community-based system that facilitates GPU sharing. For more information, refer to the provided links.

Let's delve into the Colab code. In the initial portion of the notebook, the Stable Beluga 2 model is employed to perform crossover on two sentences. The first sentence is ”**Joe wants to eat a pizza.**” while the second is ”**Mark loves to play football.**” Performing the crossover necessitates crafting an appropriate input for the model. This presents a challenge that varies depending on the model's configuration: some models generate completions for given prompt sentences, while others are designed to receive instructions as prompts and execute the requested task. Additionally, some models are specialized for engaging in conversations (chatbots), and so on.

While not essential for this specific test (it will be addressed in the implementation section), an effective prompt should enable the model to generate offspring in a format that enables the automation of the evolutionary process. The model's output, the offspring, then becomes the input for the model to be evaluated. Remember that the primary objective is to evolve prompts, and prompts should be composed of meaningful strings of text.

The initial prompt tested is: ”**Prompt 1: Joe wants to eat a pizza. Prompt 2: Mark loves to play football. Crossover:** ”. The model's completion is: ”**1. Joe wants to eat a pizza while watching Mark play football. 2. Mark loves to play football with Joe while they eat pizza together. Crossover Prompt** ”.

The format of the output string is not satisfactory. Additionally, the offsprings are significantly longer than the input prompts. To address these issues, the model is provided with an example of how to perform crossover (One Shot Prompting). This approach can be extended to multiple examples (Few Shot Prompting), which can facilitate the model's ability to learn new tasks without requiring additional training data. For instance, the prompt is: ”**Prompt 1: Lisa is having fun. Prompt 2: Jean is bored. Crossover: 1. Lisa is bored. 2. Jean is having fun. Prompt 1: Joe wants to eat a pizza. Prompt 2: Mark loves to play football. Crossover:** ”. While the output is not explicitly provided here (it can be found in the Colab notebook), the two offsprings can be readily identified: ”**Joe wants to play football**” and ”**Mark wants to eat a pizza.**” Even though the offsprings now have the same length as the input sentences, the example in the prompt introduces a significant bias into the crossover operation.

Substituting certain words with others can significantly impact the output (which is why prompt evolution is employed). By incorporating the term ”offsprings” into the crossover prompt, shorter sentences are produced without the need for any examples. The Colab notebook includes numerous additional tests involving the Stable Beluga 2 model performing both crossover and mutation operations. These tests demonstrate the model's ability to perform these operations, albeit with some refinements required before automation.

For these improvements, the paper "Connecting Large Language Models with Evolutionary Algorithms Yields Powerful Prompt Optimizers" ([1]) serves as a guideline.

Figure 2 displays a crafted prompt intended for executing both crossover and mutation. This prompt directs the language model to "*follow the instructions step-by-step*" to produce the final prompt within brackets, achieving the desired outcome. In fact, one can consider any algorithm that LLMs can execute by providing detailed step-by-step instructions.

The identical prompt proposed in the paper functions well with Stable Beluga 2, enabling us to proceed with a functional GA implementation. Firstly, it's important to note that this project aims to demonstrate a functional implementation within constraints of limited hardware resources and time availability. Computational work involves not just generating tokens with LLMs but also evaluating them. While running the Stable Beluga 2 model isn't an issue thanks to Petal (despite the need to establish a connection each time the model's generating function is called, significantly impacting the overall algorithm execution time), the focus remains on minimizing the evaluation component. The simple evaluation suggested in the Colab notebook involves tallying the occurrences of 'a' and 'A' characters in the output string derived from the provided prompt. Although this might seem like a less sophisticated metric, it's incredibly easy to implement and lightweight in computational terms, particularly as only a fixed maximum number of tokens are generated.

In summary, the current objective is to demonstrate that the GA, executed by Stable Beluga, can guide the optimization process from an initial population towards retaining an optimal prompt. In this case, the aim is a prompt that generates an output string with a high count of 'a' and 'A' characters.

The initial population consists of 10 randomly generated questions created by Google Bard. While not listing all of them here, examples of these questions include queries such as "**In the symphony of life, what instrument plays the melody of resilience?**" or "**In the relentless pursuit of dreams, what ingredient holds the key to unwavering determination?**"

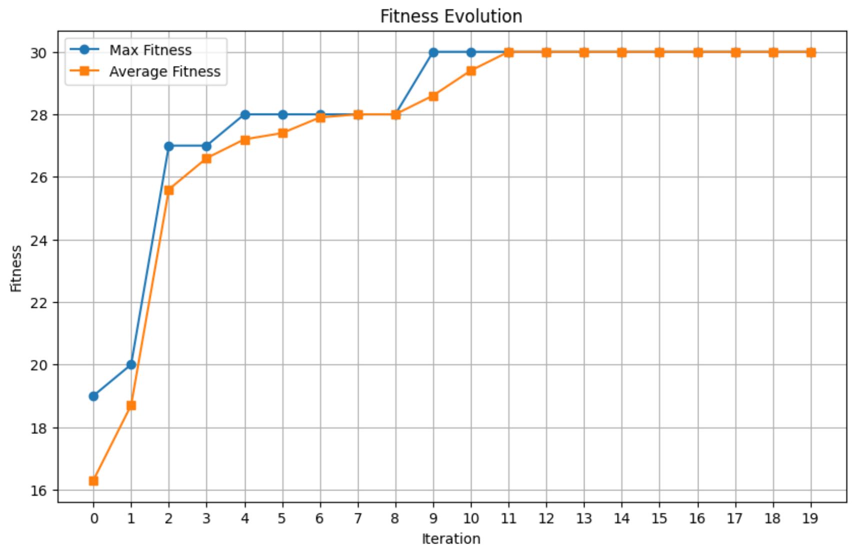


Figure 1: Fitness evolution from colab code.

Subsequently, each prompt undergoes the previously discussed evaluation process: the prompt generates an answer string, and a function tallies the occurrences of 'a' and 'A' characters within the string.

The selection phase in the Colab implementation diverges from the approach outlined in the paper. Specifically, a tournament selection method is employed. During the evolution phase, each individual generates offspring by pairing with another random individual from the population. In contrast, the paper utilizes a roulette wheel selection method to choose the two parents before executing crossover. There are various potential implementations for this aspect, considering the diverse settings within genetic algorithms, and determining the most effective methods can be a challenging task.

Finally, the evolution can be performed. Given that the program operates on a Colab virtual machine, both the population and fitness for each iteration are stored in an external file. This file can be downloaded to a local machine, ensuring persistence. Consequently, the evolution process can be interrupted at any point and resumed from the last iteration on another runtime. The existence of a file containing the evolution's data also enables a comprehensive review of the entire history and facilitates the creation of a graph(Figure 1).

After 10 iterations, it appears that the search has become trapped in a local optimum. This is evidenced by the final populations comprising repeated individuals. This situation arises because the model is performing crossovers with identical sentences, relying solely on mutations for variation. Adjusting the temperature of the Stable Beluga 2 generating function allows for the generation of more diverse outputs and consequently, diverse mutations. Employing a higher temperature, possibly an adaptive one, along with a selection process exerting less pressure, could aid in escaping such local optima with reasonable number of iterations. Nevertheless, it's evident that despite this setback, the fitness has shown significant improvement within a few iterations, underscoring the algorithm's potential effectiveness.

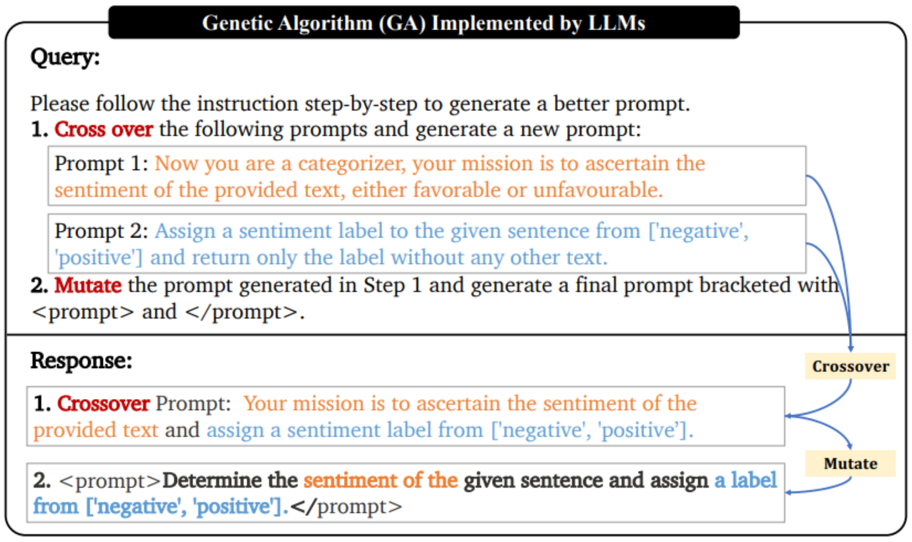
Previously, it was mentioned that one could attempt any algorithm using LLMs by crafting effective step-by-step instruction prompts. Indeed, the paper also details how to implement differential evolution (DE). The creators of EvoPrompt demonstrated that, at times, the DE variant yields superior results compared to the GA variant. However, in the final segment of the Colab notebook, Stable Beluga fails to generate the expected output within brackets, as it did previously. According to the paper, the authors executed the evolutionary operators using GPT-3.5, indicating that they designed the prompt specifically for that model. Nevertheless, it's uncertain whether this prompt would function effectively in other language models.

2 EvoPrompting

Reference paper [4]: EvoPrompting: Language Models for Code-Level Neural Architecture Search

Note: I haven't included any code in this section as running it on my laptop would be both costly and time-consuming. This serves as a brief overview of a more sophisticated implementation involving Genetic Algorithms (GA) and Language Models (LLMs). It also presents an alternative to the Neural Architecture Search (NAS) method covered in our course lectures.

In the preceding section of the report, the evolved prompts consisted of natural language sentences. Given the proficiency of language models in working with programming code, the current aim is to utilize these models to generate new code. Genetic Programming typically requires a representation of the program,



Genetic Algorithm (GA) Implemented by LLMs

Query:

Please follow the instruction step-by-step to generate a better prompt.
1. Cross over the following prompts and generate a new prompt:

Prompt 1: Now you are a categorizer, your mission is to ascertain the sentiment of the provided text, either favorable or unfavourable.

Prompt 2: Assign a sentiment label to the given sentence from ['negative', 'positive'] and return only the label without any other text.

2. Mutate the prompt generated in Step 1 and generate a final prompt bracketed with <prompt> and </prompt>.

Response:

1. Crossover Prompt: Your mission is to ascertain the sentiment of the provided text and assign a sentiment label from ['negative', 'positive'].

2. <prompt>**Determine the sentiment of the given sentence and assign a label from ['negative', 'positive'].**</prompt>

Figure 2: GA process implemented by LLMs (pag. 5 paper)

often in the form of a tree or a stream of instructions. By capitalizing on the language model's capacity to handle code strings, the program can essentially be represented by its own code. This eliminates the necessity of manually designing a search space, encompassing any syntactically valid piece of code within the search domain.

The referenced paper introduces a method named EvoPrompting for *Code-Level Neural Architecture Search*. Although EvoPrompt and EvoPrompting share a similar approach (and similar names), a closer examination reveals several differences.

The primary concept behind the EvoPrompting (Figure 4) method involves initializing a population of codes, specifically neural network architectures. However, unlike random code generation, these codes are well-known and carefully designed. Let's consider the example of the Crossover and Mutation prompts outlined in the paper (Figure 3).

An usual requirement for effective crossover or mutation operators is 'unbiasedness'. These operators must be random and independent of the fitness value. This randomness is crucial to explore the search space adequately and prevent

```
"""Metrics:
{'num_params': '4800', 'val_accuracy': '0.865'}
"""
class Model(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Dense(features=10)(x)
    return x

 """Metrics:
{'num_params': '4300', 'val_accuracy': '0.880'}
"""
class Model(nn.Module):
```

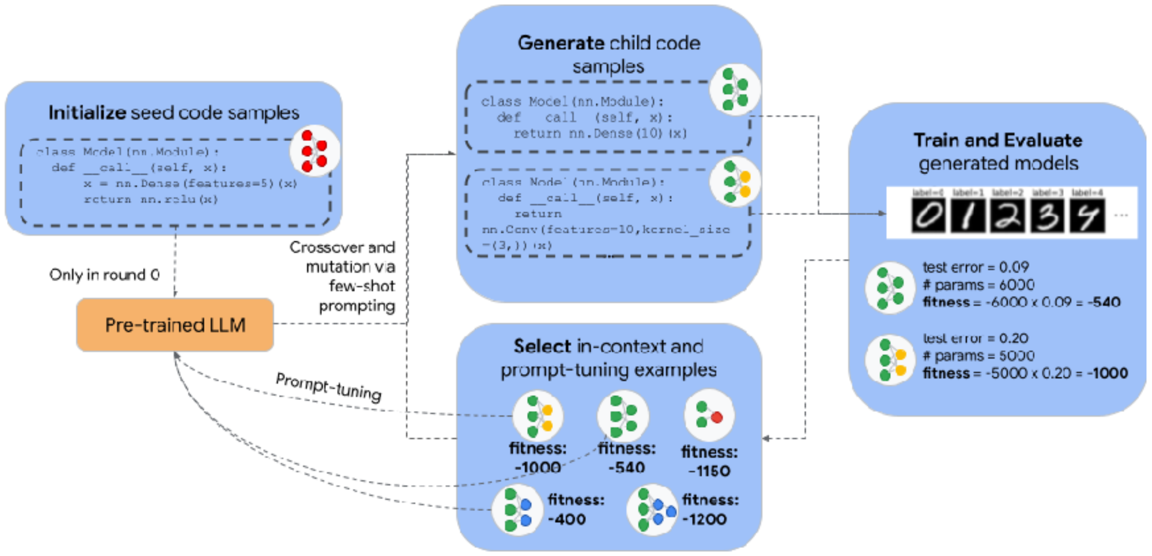
Figure 3: An example of a few-shot prompt (pag. 5 of the paper) . The second in-context example here is omitted for brevity.

rapid convergence to a local optimum.

The paper introduces a crucial aspect: '*In the last line of the prompt, we create a target set of metrics to condition π_θ's generations, indicating the desired validation accuracy and model size of the proposed architecture.*' This suggests that starting with a population of already high-quality code reduces the need for extensive exploration in the search space, minimizing bias-related issues.

A notable distinction between this few-shot prompt example and the prompts offered by the EvoPrompt framework, as seen in the previous section, is the absence of instructions here. The prompt is constructed solely using a fixed number of shots, with each shot comprising a line detailing a target set of metrics and then the program code lines associated with a parent from the population. It would be interesting to observe how a prompt designed with specific instructions would appear and function in this context.

It's important to clarify: in the previous section of this report, while experimenting with different crossover prompts, there was a discussion about avoiding bias by not using few-shot prompting. However, in that context, the 'shots' were intended as examples of performing crossover, resulting in hand-designed offsprings. In this scenario, when the paper mentions 'shots', it refers to a single parent code and not as an example of hand-made crossover.


Initialize seed code samples
Only in round 0
Pre-trained LLM
Crossover and mutation via few-shot prompting
Prompt-tuning
Generate child code samples
Train and Evaluate generated models
label=0
label=1
label=2
label=3
label=4
test error = 0.09
# params = 6000
fitness = -6000 x 0.09 = -540
test error = 0.20
# params = 5000
fitness = -5000 x 0.20 = -1000
Select in-context and prompt-tuning examples
fitness: -1000
fitness: -540
fitness: -1150
fitness: -400
fitness: -1200

```
class Model(nn.Module):
  def __call__(self, x):
    x = nn.Dense(features=5)(x)
    return nn.relu(x)
```

```
class Model(nn.Module):
  def __call__(self, x):
    return nn.Dense(10)(x)
```

```
class Model(nn.Module):
  def __call__(self, x):
    return
nn.Conv(features=10,kernel_size
=(3,))(x)
```

Figure 4: An overview of EVOPROMPTING (taken from page 2 of the paper).

Apart from these distinctions, once the core concept is grasped and a broader framework is established in the preceding section of this report, the implementation becomes relatively straightforward. As of now, the paper hasn't released any code (although it's expected to be made public soon), but adapting the Colab notebook provided earlier should be sufficent. A reference for guidance on this can be found in this GitHub repository:`https://github.com/algopapi/EvoPrompting_Reinforcement_learning`. This repository employs a version of EvoPrompting to enhance optimization in reinforcement learning algorithms. Unfortunately, the author of this code hasn't yet shared any result. To sum up, at the time of writing this, the paper's authors haven't made any code available, and the developer behind the referenced code, which draws inspiration from the paper, hasn't presented any results either.

Reviewing the results and conclusions presented in the paper, ”*EVOPROMPTING can uncover novel, competitive, and even state-of-the-art architectures that optimize for both accuracy and model size.*” Additionally, the paper includes the Python source code for the newly discovered GNNs. The primary advantage of this approach over NAS methods is the elimination of the necessity for a manually designed search space. This potential could pave the way for a new AutoML approach.

3 Conclusion

Both the EvoPrompt and EvoPrompting papers feature tables and graphs displaying their results, which are not duplicated in this report. Interested readers can refer to the papers via the provided links for details.

Instead, concluding this report necessitates a final observation regarding the Colab code, an addition that doesn't exist in the referenced papers. First, it's worth highlighting the ease of implementing the GA algorithm. There's no need to manually design and code crossover and mutation functions as the language model autonomously manages these operations. Admittedly, a well-designed prompt is essential for the language model's effectiveness. Nonetheless, the synergy between GA and LLMs is fascinating, leveraging the language model's natural language capabilities to optimize its own inputs.

Another noteworthy point is that while Stable Beluga 2 is utilized here instead of GPT as featured in the papers, at least for GA, it demonstrates commendable performance. It would be intriguing to explore how different language models execute evolutionary operators, including assessing their scalability.

Moreover, the speed at which the implemented GA converges is remarkable. As depicted in Figure 1, the fitness swiftly converges within a few iterations to reach a local optimum. In this instance, constrained computational time necessitated a swift convergence, although being stuck in a local optimum isn't typically favorable. The challenge arises as the language model conducts mutations, providing limited control. The only controllable parameters are the selection pressure, independent of the language model, and the model's temperature. Although crafting specific prompts for mutation like '*perform a smaller/larger mutation*' might seem a plausible approach, it didn't worked for me with Stable Beluga 2.

Regarding the temperature of the model, it's crucial to note that higher values may sacrifice precision, potentially causing the output to deviate from the input prompt's instructions.

References

[1] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with

evolutionary algorithms yields powerful prompt optimizers, 2023.

[2] Dakota Mahan, Ryan Carlow, Louis Castricato, Nathan Cooper, and Christian Laforte. Stable beluga models.

[3] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[4] Angelica Chen, David M. Dohan, and David R. So. Evoprompting: Language models for code-level neural architecture search, 2023.