DS Agent based ToT

Gil Goren
Tel Aviv Univeristy
gil.goren@mail.tau.ac.il

Inbar Wald
Tel Aviv University
inbar.wald@mail.tau.ac.il

Abstract

Large language models were initially developed to predict the next word token, but their capabilities rapidly extended to tasks such as code generation. However, most current code generation benchmarks and tasks emphasize coding challenges with straightforward solutions, which do not effectively measure performance on tasks with ambiguous solutions, such as data science challenges. This work proposes a novel coding assistant approach that utilizes a Tree-of-Thoughts (ToT) framework. Like the Chain of Thought method, ToT explores and verifies various options at each step. Our research demonstrates that this approach significantly improves outcomes on data science challenges, although it is more prone to errors.

1 Introduction

Large Language Models (LLMs), such as GPT-2 (Radford et al., 2019), were initially designed for text generation but soon revolutionized tasks requiring mathematical, symbolic, commonsense, and knowledge reasoning. Code generation tasks have seen continuous improvement, with benchmarks like HumanEval (Chen et al., 2021) and Mostly Basic Programming Problems (MBPP) (Austin et al., 2021) featuring general coding challenges and solutions. These benchmarks offer easy evaluation by focusing on output rather than the generated code. However, while effective, they are tailored for problems with clear outcomes.

In contrast, data science challenges demand creativity and advanced problem-solving skills, often requiring a trial-and-error approach due to the possibility of multiple solutions to the same problem. Various studies have explored data science assistants, including fine-tuning existing LLMs for specific tasks within the data science domain (Colin and Neel, 2023), such as using the Python library Pandas for data cleaning, preprocessing, and training. Unfortunately, the fine-tuned models were not released.

Research on problem-solving skills has highlighted the efficiency of the Chain-of-Thought (CoT) approach (Jason Wei, 2022), significantly improving results. The critical idea in CoT is to introduce a sequence of intermediate steps, $z_1, ..., z_n$, to bridge the initial input x and the final output y. Each z_i is a coherent language sequence that is a meaningful step towards solving the problem, facilitating the model's thought process. (Olga Golovneva, 2023) proposed extending CoT by adding verification steps at each stage to catch errors early.

An additional extension, "Tree of Thoughts" (ToT), was introduced by (Yao et al., 2023). ToT generalizes CoT by enabling LLMs to make decisions through multiple reasoning paths and self-evaluations, allowing the model to backtrack or look ahead as needed to make global choices. This approach significantly enhances problem-solving abilities in tasks requiring complex planning or search, such as Game of 24, creative writing, and mini crosswords.

In this work, we investigate the application of existing LLMs as data science assistants to tackle data science challenges using the ToT framework. The tree of thoughts will encompass steps like data exploration, data pre-processing, and model training. Each step's output will be verified before proceeding to the next. We will evaluate these models in a zero-shot scenario to compare their problem-solving skills. Our code is available at https://github.com/Gilgo2/AutoKaggleNLP/tree/master

2 Datasets

We leverage Kaggle and train our models on two distinct Kaggle competitions.

2.1 Titanic

(Cukierski, 2012) introduced an entry-level dataset in 2012, widely used by beginners in data science,

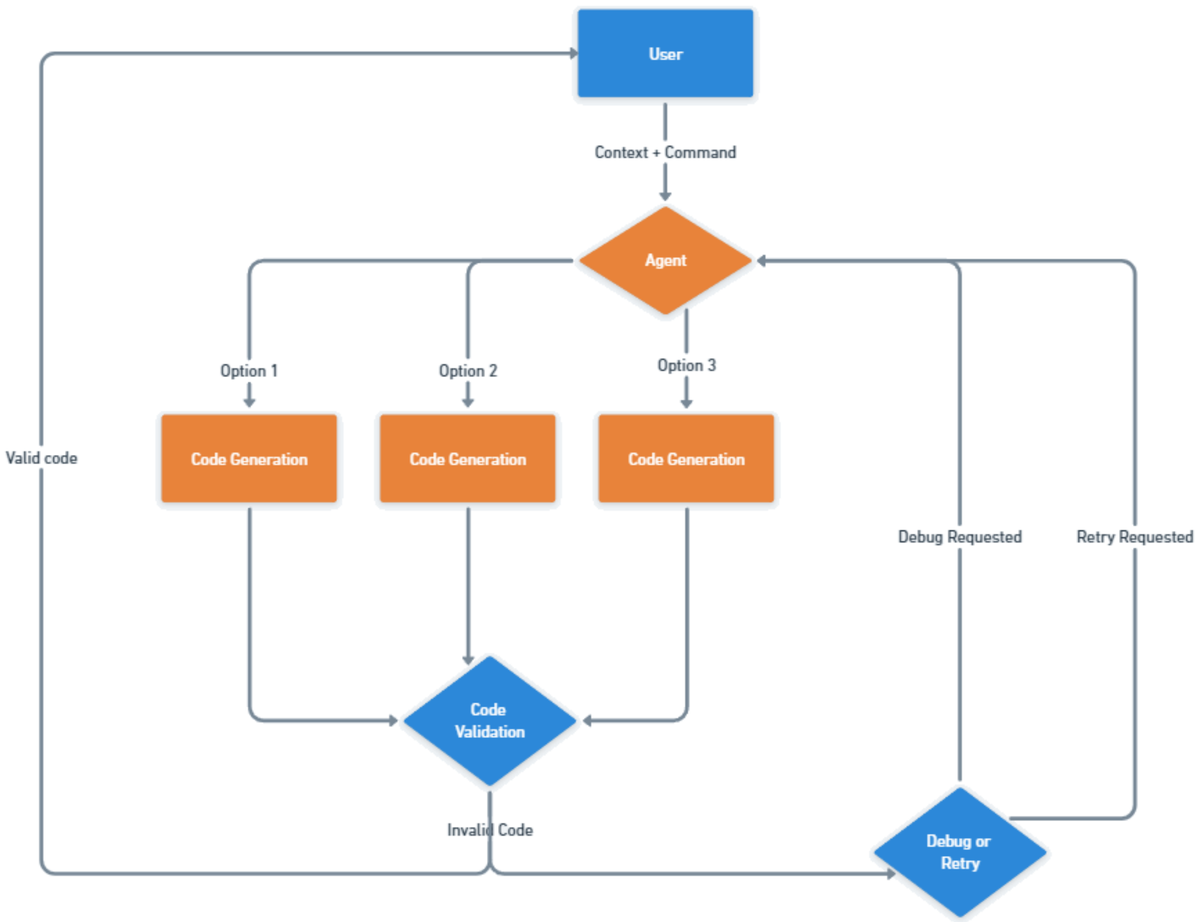

Figure 1: Flow chart of the ToT agent

where the objective is to predict a person's survival on the Titanic based on various characteristics such as name, title, cabin, and embarked port. This is a classification problem involving tabular data with ten columns.

2.2 Housing prices

(Anna Montoya, 2016) introduced another entry-level dataset in 2016, which is also widely used. The objective is to predict house prices based on size, location and other features. This is a regression problem involving tabular data with 80 columns.

3 Models

We utilized one open-source model along with several OpenAI models:

llama3:7b: Meta's latest open-source model.

We employed four different OpenAI models, including three from the GPT-4 series and one less powerful model, GPT-3.5-Turbo: **GPT-4o**, **GPT-4**, **GPT-4-Turbo**, and **GPT-3.5-Turbo**.

4 Tree of thought data science assistant

We utilize the Tree of Thought (ToT) framework to build a data science assistant. First, we provide instructions to the model, clarifying that it should function as a data science assistant using the ToT framework. We include details such as the number of child nodes to explore at each intermediate step, a description of the dataset, and the dataset's goal.

We divide the data science challenges into three steps: data exploration, data preprocessing, and model training. We provide specific instructions and formats for each step, including function signatures and expected output types. We verify the code by checking for errors and running some validation functions.

We expect the assistant to generate the entire code, self-debug if there are any errors, or retry the generation if it fails to self-debug. We instruct the model to return several different functions, formatted as function_1, function_2, etc., depending on the number of child nodes. All functions must pass all tests and run successfully; otherwise, we instruct the model to debug and regenerate the specific function.

4.1 Data Exploration

In this step, we aim for the model to gather information about the dataset to be utilized in the data preprocessing stage.

4.1.1 Instructions

Formatting

1. Receive as a parameter a single argument, which is a Pandas DataFrame named pdf
2. Return a string.

Exploration

1. Explore the dataset's structure, features, and target variable, understand their distribution.
2. Identify any missing or erroneous data.
3. Identify all categorial and numerical features.

4.1.2 Validations

1. The Returned object is a string.
2. function signature matches instructions.
3. Executing the code provides no errors.

4.2 Data preprocessing

This step is critical and responsible for feature engineering and data cleaning.

4.2.1 Instructions

Formatting

1. Receive as a parameter a single argument which is a Pandas DataFrame named pdf
2. Return a Pandas DataFrame.

Preprocessing

1. Utilize the exploration string from the last step.
2. Handle missing data
3. Encode all categorical variables into numerical format and remove the categorical variables
4. Scale or normalize numerical features
5. Conduct feature engineering.
6. Do not change the target feature.

4.2.2 Validations

1. The Returned object is a Pandas DataFrame.
2. function signature matches instructions.
3. All columns are numerical
4. Target feature is not destroyed
5. Executing the code provides no errors.

4.3 Model training

This step utilizes the preprocesed DataFrame to train ML models

4.3.1 Instructions

Formatting

1. Receive as a parameter two arguments named X_train and y_train (no access to test data)
2. Return a fitted ML model.

Prediction

1. Train several machine learning models.
2. Tune hyper-parameters to optimize model performance.
3. Employ techniques like cross-validations

4.3.2 Validations

1. Returned object is a fitted model.
2. function signature matches instructions.
3. Executing the code provides no errors.

4.4 Error Handling

The assistant handles errors using two configurable parameters: `self_debug_attempts` and `retry_attempts`. `self_debug_attempts` specifies the number of attempts the assistant has to fix its generated code upon encountering an error. If the assistant fails to correct the code within these attempts, it will retry and completely regenerate it. Upon reaching the `retry_attempts` limit, the assistant is considered to have failed to generate the code.

This work also examines and assesses how different LLMs manage exceptions and whether they can autonomously fix the code. For 2-child nodes, the debugging limit is set to 72 self-debug attempts and 8 generation retries, whereas for 1-child nodes, the limit is 26 self-debug attempts and 13 retries. The ToT configuration allows for more debugging attempts due to the increased amount of code it generates.

4.5 Comparisons

We compare the performance of the aforementioned models on various datasets. Notably, some models, such as llama-3:7b and GPT-3.5-turbo, have significantly fewer parameters and are considered less powerful than others.

In addition to evaluating the models, we also attempt to solve the challenges manually to provide a baseline for accuracy and RMSE results. This involves a naive approach, using no feature engineering and a simple random forest model, as well as taking the top 300 results from the respective Kaggle competitions.

4.6 Tree exploration

The assistant includes a configurable parameter, `child_count`, which determines the number of different paths to explore at each step. In the previously mentioned paper on the Tree of Thoughts (ToT) method, it was suggested that the tree could be explored using both breadth-first search (BFS) and depth-first search (DFS) methods. However, in our scenario, we aim to explore all possible paths, and therefore, we implemented only BFS to scan the entire tree.

5 Evaluation

5.1 Performance results

Table 1 shows the results for Titanic Survival and Housing Prices on accuracy and RMSE, respectively. the results indicate that not all models completed their runs successfully; specifically, llama3:7b and GPT-3.5-Turbo failed under the ToT configuration but succeeded with the CoT approach. Generally, the ToT approach with a 2-child configuration outperformed the 1-child configuration (i.e., the regular chain of thought method), except for GPT-4-Turbo, where the CoT approach outperformed the 2-child-node setting. This suggests that exploring multiple strategies tends to enhance performance, which aligns with the nature of data science challenges that require exploring diverse solutions.

Comparing the results across datasets, we observe that models struggled more with the Housing Prices dataset. This difficulty may be attributed to the dataset's 170 columns, compared to the 10 columns in the Titanic dataset, resulting in a significantly longer context.

5.2 Debug and retry analysis

5.3 Errors

Table 2 shows the number of times each model had to self-debug and retry. It also indicates whether a model successfully debugged itself and if it completed the task. A completion rate of 3/4 means that out of four nodes attempted, three provided code that passed all verification steps. The numbers vary based on the number of child nodes and the number of nodes completed in the final stage.

We observed that models encountered the most difficulty during the preprocessing stage, which aligns with our expectations. This stage involves the highest number of verifications and is the most delicate because it requires modifying the dataset without compromising it (e.g., altering the target variable).

Another key insight from these results is that self-debugging rarely works. When asked to fix its code, the model often repeats the same error and fails to resolve the issue.

We noticed that all models struggled more with the ToT approach. This was expected since the generated code needs to be more complex, comprising several functions, each of which must work correctly and pass all verifications. We also noted that weaker models frequently split the required code into two separate functions instead of completing all requirements within each function.

5.4 Error analysis

The collection of specific error messages is a crucial step in our analysis, as it provides valuable insights into the areas where the models struggle the most. figure 2 shows that the most common error is **KeyError**, which is usually due to the model trying to access field names that do not exist in the dataset. The models receive the feature names inside the context. We have observed cases where the model tried to access a column without a capitalized letter. The second most common error is **ColumnsIntegerError** - that is an error from a test we devised; training models expect to receive columns with only numbers; therefore, the pre-processing step must encode and then remove all categorical columns.

From this, we can understand that the errors are generally caused by the specific context (feature names) and the specific task (all columns must be integers); however, syntax errors, import errors and other code-only related errors are much lower in

Dataset	Model	1 child node	2-child node	
		best	best	average
Titanic Survival	Llama3:7b	**0.78**	Failed	Failed
	GPT-4o	0.83	**0.834**	0.821
	GPT-4	0.794	**0.847**	0.827
	GPT-4-turbo	0.839	**0.847**	0.831
	GPT-3.5-turbo	0.820	Failed	Failed
	Human-Naive	0.811	-	-
	Top-300-Kaggle	0.983	-	-
Housing Prices	Llama3:7b	Failed	Failed	Failed
	GPT-4o	196,689	**25,572**	27,750
	GPT-4	35,799	Failed	Failed
	GPT-4-turbo	**26,630**	28,723	29,324
	GPT-3.5-turbo	Failed	**28,121**	28,382
	Human-Naive	27,634	-	-
	Top-300-kaggle	12,547	-	-

Table 1: Best and average results observed. The results are the accuracy or RMSE on the test set of the titanic and housing prices datasets respectively.

Model	Task name	# self-debug		# retries		self-debugged		Completed	
		1-child	2-child	1-child	2-child	1-child	2-child	1-child	2-child
Llama3:7b	Exploration	0	0	0	0	-	-	1/1	2/2
Llama3:7b	Preprocessing	26	72	13	8	0/1	0/4	0/1	0/4
Llama3:7b	Training	-	-	-	-	-	-	-	-
GPT-4o	Exploration	0	0	0	0	-	-	1/1	2/2
GPT-4o	Preprocessing	0	32	0	4	-	0/4	1/1	3/4
GPT-4o	Training	0	0	0	0	-	-	1/1	6/6
GPT-4	Exploration	0	0	0	0	-	-	1/1	2/2
GPT-4	Preprocessing	0	0	0	0	-	-	1/1	4/4
GPT-4	Training	0	0	0	0	-	-	1/1	8/8
GPT-4-turbo	Exploration	0	0	0	0	-	-	1/1	2/2
GPT-4-turbo	Preprocessing	0	4	0	0	-	1/4	1/1	2/4
GPT-4-turbo	Training	0	0	0	0	-	-	1/1	4/4
GPT-3.5-turbo	Exploration	0	4	0	2	-	-	1/1	2/2
GPT-3.5-turbo	Preprocessing	0	72	0	8	-	0/4	1/1	0/4
GPT-3.5-turbo	Training	1	-	0	-	1/1	-	1/1	-

Table 2: Observed errors on the titanic dataset.

number. This result shows that models have learned much about code generation but had more difficulty adjusting to our specific tasks and features.

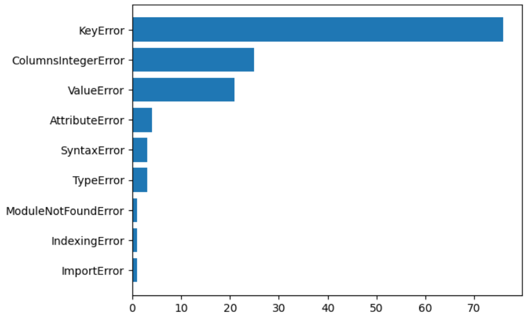

Figure 2: Bar chart of the frequency of different errors

6 Conclusions

We propose a novel approach for coding assistants by evaluating multiple options using a Tree of Thought (ToT) framework combined with various verification and test steps. Our results indicate a performance improvement, though accompanied by an increased error rate. We also observe that weaker models cannot produce solutions using the ToT approach, but we anticipate that this issue will be resolved as models continue to improve. Additionally, our analysis of errors reveals that models struggle with self-debugging, and regenerating the entire code tends to yield better results. Furthermore, we found that the longer the context, the more challenging it becomes for the model to generate accurate code.

References

DataCanary Anna Montoya. 2016. House prices - advanced regression techniques.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Guillermo Colin and Neel. 2023. *Training and Evaluating a Jupyter Notebook Data Science Assistant*.

Will Cukierski. 2012. Titanic - machine learning from disaster.

Dale Schuurmans Maarten Bosma Brian Ichter Fei Xia-Ed Chi Quoc Le Denny Zhou Jason Wei, Xuezhi Wang. 2022. Chain-of-thought prompting elicits reasoning in large language models.

Spencer Poff Martin Corredor Luke Zettlemoyer Maryam Fazel-Zarandi Asli Celikyilmaz Olga Golovneva, Moya Chen. 2023. Roscoe: A suite of metrics for scoring step-bystep reasoning.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate problem solving with large language models.