Fine-Tuning LLMs for Automated Feature Engineering

Yoichi Hirose[1] **Kento Uchida**[1] **Shinichi Shirakawa**[1]

[1]Yokohama National University

Abstract Automated machine learning (AutoML) significantly reduces human effort in developing machine learning systems. However, automated feature engineering (AutoFE) for tabular datasets, an important topic in AutoML, is still challenging because it requires exploiting contextual knowledge, such as dataset descriptions and domain expertise. To address this issue, previous work has introduced a framework that utilizes large language models (LLMs) to generate code for feature engineering, taking contextual knowledge as input. Upon evaluating this framework, we observed that LLMs often generate code that is non-executable. This paper provides a novel dataset for fine-tuning LLMs to improve the stability of code generation for feature engineering. We created candidate features by iteratively applying predefined operations to input features in publicly available tabular datasets. Subsequently, we evaluated the effectiveness of each candidate feature by training machine learning models with the feature-appended datasets. The top features that improve predictive performance for each dataset were selected and paired with metadata from the corresponding dataset. In the experiment, we demonstrate that the fine-tuned LLMs using the proposed dataset succeed in stably generating valid code for feature engineering. The experimental result shows that smaller LLMs with fine-tuning exhibit better stability to their larger counterparts without fine-tuning.

1 Introduction

Training prediction models from a tabular dataset appears in various real-world applications (Borisov et al., 2022) and is a typical problem in machine learning. Feature engineering (Zheng and Casari, 2018) is a promising approach to improve the predictive performance of machine learning models in tabular datasets. Feature engineering creates new features by composing and transforming existing features in a given tabular dataset. Exploring new features by leveraging meta-information and domain knowledge of a target tabular dataset, such as the description of each feature, is an important process in feature engineering (Khurana, 2018).

Because the manual feature engineering process is laborious and difficult for non-experts, feature engineering automation techniques have been investigated (Chen et al., 2021). Several existing methods (Horn et al., 2020; Kanter and Veeramachaneni, 2015) exhaustively apply pre-defined operations to raw features and then select promising aggregated features based on dimensionally reduction techniques. However, the computational cost of these approaches increases as the number of raw features increases because the number of candidate aggregated features depends on the number of raw features. Another promising approach is to construct new effective features without exhaustive search by leveraging domain knowledge. For instance, constructing a model that generates new features from the meta-information of a target dataset can be considered.

As a method following above-mentioned approach, Hollmann, Müller, and Hutter (2023) proposed an optimization framework named Context-Aware Automated Feature Engineering (CAAFE). This framework uses large language models (LLMs) to iteratively generate features from meta-information of the tabular dataset.

However, we have found the features generated by LLMs in the CAAFE framework are often non-executable. Figure 1 shows the ratio of non-executable codes generated by CAAFE (and the proposed model denoted with "SFT". See Section 3 for detail). We observed that more than 75% and

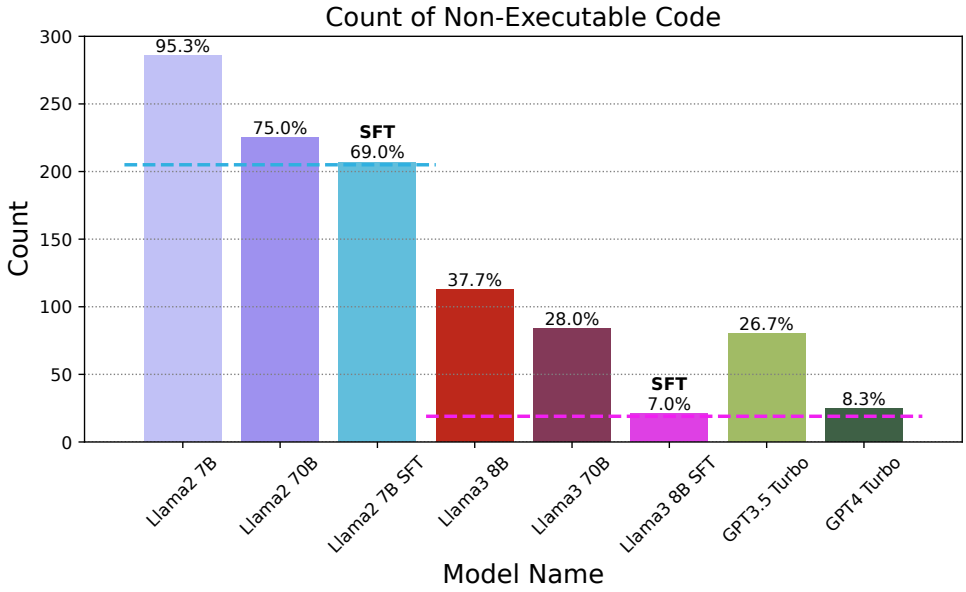

Figure 1: The count of non-executable code that each LLM model generates in evaluation.

28% of the feature engineering codes generated by Llama 2 (7B and 70B) and Llama 3 (8B and 70B) were not executable, respectively. This phenomenon increases both the run time of the CAAFE framework and the input text length for LLMs, which leads to higher costs.

To mitigate the instability in feature generation, we fine-tuned LLMs with a custom dataset that contains meta-information of publicly available datasets and corresponding code for generating effective additional features. These codes in our fine-tuning dataset were obtained by repeating exhaustive search with pre-defined operations. We investigated the effects of supervised fine-tuning (SFT) of CAAFE framework using our fine-tuning dataset and confirmed that our fine-tuning dataset improved the stability of feature generation. Additionally, our fine-tuning dataset improved the prediction performance with the Llama 2 series and maintained the performance with the Llama 3 series. Finally, the comparison with CAAFE applied to GPT models, which requires an API fee, showed that our fine-tuned open models were a practical alternative.

2 Fine-Tuning Dataset for Feature Engineering

For improved stability in automated feature engineering with LLMs, we construct a fine-tuning dataset containing metadata of existing tabular datasets and code for generating effective additional features. Our search strategy for identifying effective additional features involves an iterative process of exhaustive search. In each iteration, we generate the candidate features and evaluate them with four machine learning models $M_j (j = 1, \ldots, 4)$. After three iterations, we determine the top performing feature $F_{i,j}$ for each tabular dataset T_i and model type M_j. Since we used four models and 672 datasets, our fine-tuning datasets $\mathcal{D} = \cup_{i,j}\{(I_i, M_j, F_{i,j})\}$ consist of 2,688 tuples of metadata I_i, model types M_j, and effective additional features $F_{i,j}$.

2.1 Datasets and Metadata used to Construct Fine-tuning Dataset

We found almost 5,000 tabular datasets and their metadata in OpenML (Vanschoren et al., 2013) and collected tabular datasets satisfying all of the following conditions from those publicly available datasets. First, we collected tabular datasets with more than 100 samples to stabilize the search performance for effective additional features. Then, we reduced the number of samples to 5,000

when the total amount of samples exceeded 5,000. Next, We selected the datasets with fewer than 15 features to reduce the training cost of machine learning models. After that, we checked the data type of features and selected the datasets that contain at least one feature with the data type appeared in Table 2 (shown in the Appendix). Finally, we obtained the target datasets by selecting the datasets that contain a "description" of the dataset in the corresponding metadata provided by OpenML. We also manually excluded the datasets whose contents are duplicated with 10 evaluation datasets shown in Table 1. The total number of datasets satisfying all the conditions was 672. The metadata corresponding to the obtained datasets contains an overview of the dataset, names of features, and names of target variables.

2.2 Search Strategy for Effective Additional Features

We first evaluate the performance of the machine learning models without additional features as a baseline. Next, we construct candidate features by applying the operations in Table 2 (shown in the Appendix) to all features and all pairs of features that have compatible data types. Then, we evaluate each candidate feature based on the performance of the machine learning model trained with it in addition to the original features. We select the 20 features that lead to the highest performance and iteratively construct the candidate features using both original and previously selected features. We repeat this procedure three times, resulting in a total of 60 selected features. Finally, we select the best candidate feature with the highest performance achieved through this search. By choosing only the best feature, we ensure that our fine-tuning dataset contains only well-performing features.

We used the following four models to construct the fine-tuning dataset: linear model, decision tree, K-nearest neighbor, and multi-layer perceptron. We selected these models to reduce the time for creating the fine-tuning dataset while maintaining the diversity of the selected features. Note that, as the linear model, we used a logistic regression model and a linear regression model for classification and regression tasks, respectively. We implemented these models using `scikit-learn 1.2.1` (Pedregosa et al., 2011) and used the default training setting for each model. We evaluated the performance of the model using five-fold cross-validation, which was measured by RMSE for regression tasks and accuracy for classification tasks.

In a previous study (Hollmann, Müller, and Hutter, 2023), LLMs were used to generate both code for calculating additional feature values and comments explaining the properties of the generated features. These comments included the name of the generated feature, a description of the feature, its usefulness, and the names and sample values of existing features used to generate additional features. To use the same content in fine-tuning, we make GPT-3.5 to generate the name of the obtained feature and a description of usefulness. We provide the examples of generated content in Figure 3 (shown in Appendix).

3 Experiment and Result

We ran the CAAFE framework with eight LLMs, including those fine-tuned with our dataset. First, we compared the count of non-executable generated codes for feature engineering and confirm that fine-tuning LLMs with our dataset increases the stability of feature generation. Next, we compared the accuracy of tabular prediction tasks among features generated with different LLMs. We found that supervised fine-tuning improves accuracy in the Llama 2 series and achieves comparable results in the Llama 3 series. Finally, our comparison with GPT models suggests that our fine-tuned models are a viable alternative to GPT models when considering the cost-accuracy trade-off.

3.1 Experimental Setting

We used TabPFN (Hollmann, Müller, Eggensperger, et al., 2023) as a machine learning model for tabular prediction tasks in the evaluation of generated features, because it performs best in Hollmann, Müller, and Hutter (2023). We evaluated each of 10 tabular datasets in three trials with

different seeds, generating 10 features in each trial, following the CAAFE settings. These evaluation datasets were excluded from the fine-tuning process. The seed only affects the LLM inference, while the rest of stochastic components, such as splitting of tabular datasets and TabPFN inference, are fixed with seed 0.

We chose Llama series (Touvron et al., 2023) as LLMs to measure the effect of supervised fine-tuning (SFT). We ran the CAAFE framework with 7B, 70B, and 7B with SFT for Llama 2 Chat, and 8B, 70B, and 8B with SFT for Llama 3 Instruct. For comparison, we also ran the framework with OpenAI's GPT models: GPT-3.5 Turbo (`gpt-3.5-turbo-0125`) and GPT-4 Turbo (`gpt-4-1106-preview`).

We fine-tuned the LLMs using low-rank adaptation (LoRA) (Hu et al., 2022) with the settings shown in Table 3 (shown in the Appendix), and used the last checkpoint as the SFT model.

In the Llama series, we performed inference with top-p sampling, setting p at 0.92 and the temperature at 1.0, while limiting the maximum token generation to 500. We manually determined these settings using non-evaluation tabular datasets by checking whether the generated code appears natural to humans, based on typical hyperparameters. Additionally, we revised the original CAAFE prompt format to a more structured one to enhance the understanding of prompts for LLMs without changing the content from the original prompts in CAAFE as in Figure 2 (shown in Appendix). All other settings followed those used in the previous study.

3.2 Result

Figure 1 shows the count of non-executable code in a total of 300 generated code for each model in this experiment. It indicates that LLMs with supervised fine-tuning (SFT) are less likely to generate non-executable code for feature engineering. Specifically, the error rate for the Llama 2 7B model has decreased from 95.3 % to 69.0 %. This rate is lower than the 75.0 % observed in the more capable 70B model. Similarly, the rate for the Llama 3 8B model has fallen from 37.7 % to 7.0 %. This rate is not only below 28.0 % in the 70B model but also 8.3 % in GPT-4 Turbo, one of the cutting-edge LLMs.

Table 1 shows the mean test accuracies using the features generated by each LLM. Table 1 reveals that Llama 2 models with SFT outperform those without SFT. The result suggests that SFT using our dataset effectively improves the accuracy of LLMs in certain cases.

For Llama 3 models, they exhibit similar performance levels, while there is a case where they have achieved the highest performance among all models evaluated. Furthermore, when comparing the GPT models to the Llama 3 8B SFT model, the GPT models generally show better accuracy. As the GPT models require API fees to generate new features, open models, such as Llama 3, are advantageous in terms of monetary cost.

4 Conclusion

To address the instability in feature generation, we fine-tuned LLM using a custom dataset including meta-information of publicly available datasets and the corresponding codes for generating effective additional features. Our investigation with the CAAFE framework demonstrated that supervised fine-tuning using our fine-tuning dataset significantly enhanced the stability of feature generation. Moreover, we observed an increase in accuracy within the Llama 2 series, whereas the fine-tuned models achieved comparable performance in the Llama 3 series. Comparison with GPT models revealed that our fine-tuned open models became a practical alternative.

5 Broader Impact Statement and Limitations

To fine-tune LLMs, considerable computational resources are required, which poses a significant limitation in environments with limited computational resources. Additionally, creating a dataset for training LLMs requires significant computational resources. Despite efforts to remove duplicate tabular datasets during the preparation of our fine-tuning dataset, the possibility of contamination

Table 1: The mean test accuracy across three trials for each dataset and LLM (standard deviation in parentheses). The underline represents the higher mean accuracy between the SFT model and the non-SFT model of the same size for each dataset. The bold indicates the highest mean accuracy in each row.

Model Type	Llama 2 (Chat)			Llama 3 (Instruct)			GPT (Turbo)	
Model Name	7B	70B	7B SFT	8B	70B	8B SFT	3.5	4
Dateset Name								
Airlines	0.6460 (0.0000)	0.6507 (0.0050)	0.6503 (0.0075)	0.6460 (0.0000)	0.6460 (0.0000)	0.6460 (0.0000)	**0.6610** (0.0128)	0.6463 (0.0006)
Balance Scale	0.8340 (0.0000)	0.8340 (0.0000)	**0.8847** (0.0558)	0.8340 (0.0000)	0.8513 (0.0012)	0.8367 (0.0023)	0.8800 (0.0745)	0.8613 (0.0067)
Breast W	0.9910 (0.0000)	0.9910 (0.0000)	0.9910 (0.0000)	0.9910 (0.0000)	0.9910 (0.0000)	**1.0000** (0.0000)	0.9940 (0.0052)	0.9910 (0.0000)
Cmc	0.7300 (0.0000)	0.7307 (0.0006)	0.7300 (0.0000)	0.7333 (0.0000)	0.7320 (0.0000)	0.7300 (0.0000)	0.7323 (0.0021)	**0.7327** (0.0015)
Credit G	0.8070 (0.0000)	0.8073 (0.0006)	0.8070 (0.0000)	0.8083 (0.0000)	0.8080 (0.0000)	0.8090 (0.0000)	**0.8103** (0.0021)	0.8080 (0.0000)
Diabetes	0.8170 (0.0000)	0.8193 (0.0025)	0.8170 (0.0000)	0.8190 (0.0000)	0.8170 (0.0000)	0.8170 (0.0000)	**0.8247** (0.0035)	0.8220 (0.0010)
Eucalyptus	0.9180 (0.0000)	0.9190 (0.0010)	0.9187 (0.0012)	0.9190 (0.0000)	0.9190 (0.0000)	0.9220 (0.0000)	0.9213 (0.0006)	**0.9227** (0.0025)
Jungle Chess	0.9240 (0.0000)	0.9307 (0.0059)	0.9260 (0.0017)	0.9340 (0.0000)	**0.9380** (0.0000)	0.9240 (0.0000)	0.9317 (0.0015)	0.9373 (0.0042)
Pc1	0.8810 (0.0000)	0.8827 (0.0029)	0.8817 (0.0012)	**0.8930** (0.0000)	0.8880 (0.0000)	0.8850 (0.0000)	0.8853 (0.0025)	0.8887 (0.0029)
Tic Tac Toe	0.5970 (0.0000)	0.7250 (0.1119)	0.5970 (0.0000)	0.6123 (0.0000)	**0.9310** (0.0000)	0.5970 (0.0000)	0.8050 (0.1079)	0.8707 (0.1290)

remains. This issue is not unique to our dataset; pre-training datasets used for LLMs, such as those in the Llama series, may also be subject to contamination. For improved generalization, it would be beneficial to expand the experimental results beyond the current 10 datasets. Including evaluation datasets with more than 15 columns could offer valuable insights into extrapolation capabilities. Moreover, it would be worthwhile to investigate whether error-free feature generation could result in the meaningless easy-to-create features.

After careful reflection, the authors have determined that this work presents no notable negative impacts to society or the environment.

Acknowledgements. This work was partially supported by JSPS KAKENHI (JP23H00491, JP23H03466), JST PRESTO (JPMJPR2133), NEDO (JPNP18002, JPNP20006).

References

Borisov, V. et al. (2022). "Deep Neural Networks and Tabular Data: A Survey". In: *IEEE Transactions on Neural Networks and Learning Systems*.

Chen, Y., Q. Song, and X. Hu (2021). "Techniques for Automated Machine Learning". In: *ACM SIGKDD Explorations Newsletter* 22.2, pp. 35–50.

Hollmann, N., S. Müller, K. Eggensperger, et al. (2023). "TabPFN: A Transformer That Solves Small Tabular Classification Problems in a Second". In: *The Eleventh International Conference on Learning Representations (ICLR)*. URL: https://openreview.net/forum?id=cp5PvcI6w8_.

Hollmann, N., S. Müller, and F. Hutter (2023). "Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering". In: *Neural Information Processing Systems (NeurIPS)*. URL: https://openreview.net/forum?id=9WSxQZ9mG7.

Horn, F., R. Pack, and M. Rieger (2020). "The Autofeat Python Library for Automated Feature Engineering and Selection". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pp. 111–120.

Hu, E. J. et al. (2022). "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=nZeVKeeFYf9.

Kanter, J. M. and K. Veeramachaneni (2015). "Deep Feature Synthesis: Towards Automating Data Science Endeavors". In: *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pp. 1–10.

Khurana, U. (2018). "Automating Feature Engineering in Supervised Learning". In: *Feature Engineering for Machine Learning and Data Analytics*. CRC Press. Chap. 9.

McKinney, W. (2010). "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*, pp. 56–61.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85, pp. 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

Touvron, H. et al. (2023). "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971*.

Vanschoren, J. et al. (2013). "OpenML: Networked Science in Machine Learning". In: *SIGKDD Explorations* 15.2, pp. 49–60. URL: http://doi.acm.org/10.1145/2641190.2641198.

Zheng, A. and A. Casari (2018). *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. 1st. O'Reilly Media, Inc.

Submission Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]

   (b) Did you describe the limitations of your work? [Yes]

   (c) Did you discuss any potential negative societal impacts of your work? [N/A]

   (d) Did you read the ethics review guidelines and ensure that your paper conforms to them? `https://2022.automl.cc/ethics-accessibility/` [Yes]

2. If you ran experiments...

   (a) Did you use the same evaluation protocol for all methods being compared (e.g., same benchmarks, data (sub)sets, available resources)? [Yes]

   (b) Did you specify all the necessary details of your evaluation (e.g., data splits, pre-processing, search spaces, hyperparameter tuning)? [Yes]

   (c) Did you repeat your experiments (e.g., across multiple random seeds or splits) to account for the impact of randomness in your methods or data? [Yes]

   (d) Did you report the uncertainty of your results (e.g., the variance across random seeds or splits)? [No]

   (e) Did you report the statistical significance of your results? [No]

   (f) Did you use tabular or surrogate benchmarks for in-depth evaluations? [No] To our knowledge, Surrogate benchmarks do not exist for our problem setting.

   (g) Did you compare performance over time and describe how you selected the maximum duration? [No]

   (h) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No]

   (i) Did you run ablation studies to assess the impact of different components of your approach? [No]

3. With respect to the code used to obtain your results...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit versions), random seeds, an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [No]

   (b) Did you include a minimal example to replicate results on a small subset of the experiments or on toy data? [No]

   (c) Did you ensure sufficient code quality and documentation so that someone else can execute and understand your code? [No]

   (d) Did you include the raw results of running your experiments with the given code, data, and instructions? [No]

   (e) Did you include the code, additional data, and instructions needed to generate the figures and tables in your paper based on the raw results? [No]

4. If you used existing assets (e.g., code, data, models)…

   (a) Did you cite the creators of used assets? [N/A]

   (b) Did you discuss whether and how consent was obtained from people whose data you're using/curating if the license requires it? [N/A]

   (c) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you created/released new assets (e.g., code, data, models)…

   (a) Did you mention the license of the new assets (e.g., as part of your code submission)? [N/A]

   (b) Did you include the new assets either in the supplemental material or as a URL (to, e.g., GitHub or Hugging Face)? [N/A]

6. If you used crowdsourcing or conducted research with human subjects…

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

7. If you included theoretical results…

   (a) Did you state the full set of assumptions of all theoretical results? [N/A]

   (b) Did you include complete proofs of all theoretical results? [N/A]

A Details of Search Strategy for Effective Additional Features

Table 2 shows the list of operations to generate candidate features in exhaustive search. The input features `x` and `y` are assumed to be Series class in `pandas` (McKinney, 2010). The symbol `const.` is given a value randomly selected in the range between the maximum and minimum values in the corresponding column. The operations with single input are applied to all features with the consistent data type to "Feature1". The operations with two inputs are applied to all pairs of features with the consistent data type to "Feature1" and "Feature2", where one of the orders for input features is applied when the target operation is symmetric (represented with the mark "✓" in "Symm.").

Table 3 shows the settings of the low-rank adaptation (LoRA) (Hu et al., 2022) in our SFT.

Table 2: List of operations to generate candidate features in exhaustive search.

Feature1 (pandas Series x)	Feature2 (pandas Series y)	Symm.	Candidate Features (const. is replaced with a randomly selected value)

numeric	–	–	`x+const.` `x*const.` `x*x` `x/const.`
			`-x` `1/x` `x%const.` `x==const.`
			`x!=const.` `x!=const.` `x>const.` `x>x.mean()`
			`(x.mean()+x.std()>x)&(x>x.mean()-x.std())`
			`x.abs()` `np.cos(x)` `np.sin(x)` `np.sqrt(x)`
			`np.log(x)`

datetime	–	–	`x.dt.year` `x.dt.month` `x.dt.day`
			`x.dt.hour` `x.dt.minute` `x.dt.second`
			`x.dt.dayofyear` `x.dt.quarter` `x.dt.weekday`
			`x.dt.is_year_start` `x.dt.is_month_start`
			`x.dt.is_quarter_start` `x.dt.is_year_end`
			`x.dt.is_month_end` `x.dt.is_quarter_end`

category	–	–	`x.map(x.value_counts())`
bool	–	–	`~x`

category	numeric	–	`x.map(y.groupby(x).mean())` `x.map(y.groupby(x).min())`
			`x.map(y.groupby(x).max())` `x.map(y.groupby(x).median())`
			`x.map(y.groupby(x).sum())` `x.map(y.groupby(x).std())`
			`x.map((y>const.).groupby(x).sum())`
			`y>x.map(y.groupby(x).mean())`

category	datetime	–	`x.map(y.groupby(x).mean())`
numeric	numeric	✓	`x+y` `x*y` `x==y` `x>y`
bool	bool	✓	`x|y` `x&y`
category	category	✓	`x.astype(str)+'-'+y.astype(str)`
datetime	datetime	✓	`x-y` `x>y`
timedelta	timedelta	✓	`x-y` `x>y`
numeric	numeric	–	`x/y`
category	category	–	`x.map( y.groupby(x).apply(` `lambda z: stats.entropy(z.value_counts(normalize=True))`

Table 3: LoRA Settings.

Name	Value
Rank	1
Alpha	1
Dropout	0.1
Target Modules	$\mathbf{W}_{\text{key}}, \mathbf{W}_{\text{query}}, \mathbf{W}_{\text{value}}, \mathbf{W}_{\text{out}}$
Learning Rate	0.001
Weight Decay	0.01
Epoch	1
Batch Size	4
Machine	four A100 80GB PCIe GPU

```
You have a dataframe, df, loaded in memory. Your task is to generate a code snippet that adds new columns to this
dataframe. These columns should be informative for a downstream classification algorithm, such as XGBoost, which is
predicting "${predicting}". Below is the metadata for the dataset, a description of the existing columns, and other
relevant information.

# Metadata
## Dataset Description
The description of the dataset is provided within triple quotes. Note that the column data types mentioned might be
inaccurate.

"""
${data_description_unparsed}
"""

## Columns in `df`
Here is a list of the actual data types of the features, with categorical data encoded as integers.

"""
${samples}
"""

## Additional Information
* The number of samples (rows) in the training dataset is ${len_df}.
* This code snippet is developed by an expert data scientist aiming to enhance prediction accuracy.

# Code Format
Each code snippet should start with "python" and end with "end". You are to add only ${how_many} column per snippet.
Each addition should follow this format:

```python
# (Feature name and description)
# Usefulness: Description of how this feature adds valuable insights for classifying "${predicting}" based on the
dataset description and attributes.
# Input samples: Examples of the columns used in the code (e.g., '${col_name_1}': ${col_value_1}, '${col_name_2}': ${
col_value_2}, ...)
(The pandas code using columns like '${col_name_1}', '${col_name_2}', etc., to create a new column for each row in df)
```end

Use the following format for removing redundant columns:
```python
# Reason for dropping column XX
df.drop(columns=['XX'], inplace=True)
```end

## Guidelines
* Introduce additional columns that add new semantic information, leveraging real-world knowledge about the dataset.
* These new columns may include feature combinations, transformations, or aggregations, derived from existing columns in
 the dataset.
* The scale and offset of columns are irrelevant.
* Ensure that all columns you use are present in the dataset. Pay close attention to the descriptions of the columns,
including data types and the significance of various classes.
* This process may also involve dropping columns that could be redundant and potentially detrimental to the predictive
performance of the subsequent classifier (a practice known as feature selection).
* Removing columns can reduce the risk of overfitting, which is particularly beneficial for smaller datasets.
* The classifier will be trained on the modified dataset, including any newly generated columns, and evaluated on a
separate holdout set.
* The primary evaluation metric is accuracy. The most effective modification will be adopted.
* While you can use added columns in subsequent code snippets, dropped columns will no longer be available.
```

Figure 2: Revised prompt format containing equivalent information as the original.

```
Code:
df["tmp0"] = df["Age_of_patient_at_time_of_operation"] * df["Age_of_patient_at_time_of_operation"]
df["tmp1"] = df["Number_of_positive_axillary_nodes_detected"] / df["tmp0"]
df["new"] = -df["tmp1"]

Feature Name:
influence-of-age-on-survival

Description:
Measure of the impact of patient's age on their survival status

Usefulness:
By calculating the ratio of the number of positive axillary nodes detected to the square of the patient's age at the
time of operation and assigning it a negative sign, this feature captures a potential relationship between age and
survival status. The negative sign suggests that as this ratio increases, the likelihood of survival may decrease.
Consequently, it provides valuable insights into how age influences the patient's survival status, which can be crucial
for accurately predicting the likelihood of a patient surviving 5 years or longer after undergoing breast cancer surgery
.
```

```
Code:
df["new"] = df["Address_bit_1"].astype(str) + '-' + df["Address_bit_2"].astype(str)

Feature Name:
Address-Bit-Combination

Description:
Concatenation of two address bits

Usefulness:
This new feature, "Address-Bit-Combination," is created by concatenating the values of "Address_bit_1" and "
Address_bit_2" columns. Based on the dataset description, it seems that these address bits may represent certain
configurations or states. By combining these two address bits into a single feature, it could potentially capture unique
 patterns or interactions between the two. This could be valuable for predicting the target variable "class" as it
introduces a new aspect to the data that may not be as evident when analyzing individual bits separately. The expert
data scientist likely believes that this feature could improve the model's predictive accuracy by providing more nuanced
 information for classification.
```

```
Code:
df["tmp0"] = df["INCOME"] / df["POP"]
df["tmp1"] = df["YCOORD"] / df["XCOORD"]
df["tmp2"] = np.tan(df["tmp1"])
df["new"] = df["tmp0"] * df["tmp2"]

Feature Name:
adjusted_income_density

Description:
Calculated adjusted income density based on population and income

Usefulness:
The new feature "adjusted_income_density" captures the relationship between income and population density within each
county. By multiplying the income-to-population ratio with the tangent of Y spatial coordinate to X spatial coordinate
ratio, the feature reflects the spatial distribution of income across counties. This can provide valuable insights into
how income levels are geographically spread and may help in predicting the proportion of votes cast in the 1980
presidential election based on income and population factors.
```

```
Code:
df["tmp0"] = df["bmi"] + df["s4"]
df["tmp1"] = df["s5"] + df["tmp0"]
df["new"] = df["bmi"] + df["tmp1"]

Feature Name:
disease_progression_indicator

Description:
An indicator reflecting the combined effect of blood serum measurements on disease progression

Usefulness:
By combining the impact of serum measurements with BMI, the new feature "disease_progression_indicator" provides a more
comprehensive view of how these factors collectively contribute to disease progression. This can aid in better
understanding the relationship between patient characteristics and disease severity, potentially improving the accuracy
of predicting disease progression one year after baseline.
```

Figure 3: Examples of code explanations generated by GPT-3.5, including feature name, description, and usefulness from the code and metadata.