CAAFE: Combining Large Language Models with Tabular Predictors for Semi-Automated Data Science

Noah Hollmann [1 2 3] **Samuel Müller** [1 2] **Frank Hutter** [1 2]

Abstract

As the field of automated machine learning (AutoML) advances, it becomes increasingly important to incorporate domain knowledge into these systems. Our approach combines the advantages of classical ML classifiers (robustness, predictability and a level of interpretability) and LLMs (domain-knowledge and creativity). We introduce Context-Aware Automated Feature Engineering (CAAFE), a feature engineering method for tabular datasets that utilizes an LLM to iteratively generate additional semantically meaningful features for tabular datasets based on the description of the dataset. The method produces both Python code for creating new features and explanations for the utility of the generated features.

Despite being methodologically simple, CAAFE improves performance on 11 out of 14 datasets - boosting mean ROC AUC performance from 0.798 to 0.822 across all dataset - similar to the improvement achieved by using a random forest instead of logistic regression on our datasets.

Furthermore, CAAFE is interpretable by providing a textual explanation for each generated feature. CAAFE paves the way for more extensive semi-automation in data science tasks and emphasizes the significance of context-aware solutions that can extend the scope of AutoML systems to semantic AutoML. We release our code and a a simple demo.

1. Introduction

Automated machine learning (AutoML; e.g., (Hutter et al., 2019)) is very effective at optimizing the machine learning (ML) part of the data science workflow, but existing systems leave tasks such as data engineering and integration of domain knowledge largely to human practitioners. However, model selection, training, and scoring only account for a small percentage of the time spent by data scientists (roughly 23% according to the "State of Data Science"(Anaconda, 2020)). Thus, the most time-consuming tasks, namely data engineering and data cleaning, are only supported to a very limited degree by AutoML tools, if at all.

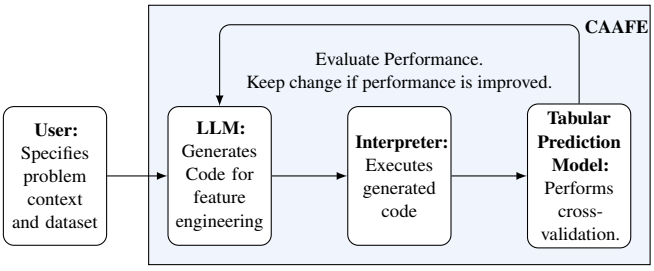

Figure 1. CAAFE accepts a dataset as well as user-specified context information and operates by iteratively proposing and evaluating feature engineering operations.

While the traditional AutoML approach has been fruitful and appropriate given the technical capabilities of ML tools at the time, large language models (LLMs) may extend the reach of AutoML to cover more of data science and allow it to evolve towards *automated data science* (De Bie et al., 2022). LLMs encapsulate extensive domain knowledge that can be used to automate various data science tasks, including those that require contextual information. They are, however, not interpretable, or verifiable, and behave less consistently than classical ML algorithms. E.g., even the best LLMs still fail to count or perform simple calculations that are easily solved by classical methods (Hendrycks et al., 2021; OpenAI Community, 2021).

In this work, we propose an approach that combines the scalability and robustness of classical ML classifiers (e.g. random forests (Breiman, 2001)) with the vast domain knowledge embedded in LLMs. We bridge the gap between LLMs and classical algorithms by using code as an interface that allows LLMs to interact with classical algorithms and provides an interpretable interface to users. Our proposed

[1]University of Freiburg [2]Prior Labs [3]Charité Hospital Berlin. Correspondence to: Noah Hollmann <noah.homa@gmail.com>.

Accepted after peer-review at the 1st workshop on Synergy of Scientific and Machine Learning Modeling, SynS & ML ICML, Honolulu, Hawaii, USA. July, 2023.

method, CAAFE, generates Python code that creates semantically meaningful features that improve the performance of downstream prediction tasks in an iterative fashion and with algorithmic feedback as shown in Figure 1. Furthermore, it provides explanations for the utility of generated features. This allows for human-in-the-loop, interpretable AutoML (Lee & Macke, 2020), making it easier for the user to understand a solution, but also to modify and improve on it. Our approach combines the advantages of classical ML (robustness, predictability and a level of interpretability) and LLMs (domain-knowledge and creativity).

Automating the integration of domain-knowledge into the AutoML process has clear advantages that extend the scope of existing AutoML methods. These benefits include: i) Reducing the latency from data to trained models; ii) Reducing the cost of creating ML models; iii) Evaluating a more informed space of solutions than previously possible with AutoML, but a larger space than previously possible with manual approaches for integrating domain knowledge; and iv) Enhancing the robustness and reproducibility of solutions, as computer-generated solutions are more easily reproduced. CAAFE demonstrates the potential of LLMs for automating a broader range of data science tasks and highlights the emerging potential for creating more robust and context-aware AutoML tools.

2. Background

2.1. Large Language Models (LLMs)

LLMs are neural networks that are pre-trained on large quantities of raw text data to predict the next word in text documents. Recently, GPT-4 has been released as a powerful and publicly available LLM (OpenAI, 2023a). It achieves state-of-the-art performance on various tasks, such as text generation, summarization, question answering and coding.

LLMs as Tabular Prediction Models Hegselmann et al. (2023) recently showed how to use LLMs for tabular data prediction by applying them to a textual representation of these datasets. A prediction on an unseen sample then involves continuing the textual description of that sample on the target column. However, this method requires encoding the entire training dataset as a string and processing it using a transformer-based architecture, where the computational cost increases quadratically with respect to $N \cdot M$, where N denotes the number of samples and M the number of features. Furthermore, the predictions generated by LLMs are not easily interpretable, and there is no assurance that the LLMs will produce consistent predictions, as these predictions depend directly on the complex and heterogeneous data used to train the models. So far, Hegselmann et al. (2023) found that their method yielded the best performance on tiny datasets with up to 8 data points, but was outperformed for larger data sets.

LLMs for Data Wrangling Narayan et al. (2022) demonstrated state-of-the-art results using LLMs for entity matching, error detection, and data imputation using prompting and manually tuning the LLMs. Vos et al. (2022) extended this technique by employing an improved prefix tuning technique. Both approaches generate and utilize the LLMs output for each individual data sample, executing a prompt for each row. This is in contrast to CAAFE, which uses code as an interface, making our work much more scalable and faster to execute, since one LLM query can be applied to all samples.

2.2. Feature Engineering

Feature engineering refers to the process of constructing suitable features from raw input data, which can lead to improved predictive performance. Given a dataset $D = (x_i, y_i)_{i=1}^n$, the goal is to find a function $\phi : \mathcal{X} \to \mathcal{X}'$ which maximizes the performance of $A(\phi(x_i), y_i)$ for some learning algorithm A. Common methods include numerical transformations, categorical encoding, clustering, group aggregation, and dimensionality reduction techniques, such as principal component analysis (Wold et al., 1987).

Various strategies for automated feature engineering have been explored in prior studies. However, none of the existing methods can harness semantic information in an automated manner. The potential feature space, when considering the combinatorial number of transformations and combinations, is vast. Therefore, semantic information is useful, to serve as a prior for identifying useful features. By incorporating semantic and contextual information, feature engineering techniques can be limited to semantically meaningful features enhancing the performance by mitigating issues with multiple testing and computational complexity and boosting the interpretability of machine learning models. This strategy is naturally applied by human experts who leverage their domain-specific knowledge and insights.

3. Method

We present CAAFE, an approach that leverages large language models to incorporate domain knowledge into the feature engineering process, offering a promising direction for automating data science tasks while maintaining interpretability and performance.

Our method takes the training and validation datasets, D_{train} and D_{valid}, as well as a description of the context of the training dataset and features as input. From this information CAAFE constructs a prompt, i.e. instructions to the LLM containing specifics of the dataset and the feature engineering task. Our method performs multi-

```
Dataset description: Breast Cancer Wisconsin (Original)
     Data Set. Features are computed from a digitized
     image of a fine needle aspirate (FNA) of a breast
     mass. They describe characteristics of the cell
     nuclei present in the image. The target feature
     records the prognosis (malignant or benign).
```

```
# Iteration 1

# Total_Cell_Size: Useful for identifying larger cells
     that may be more indicative of malignancy
# Input samples: ('Cell_Size_Uniformity': [1.0, 2.0,
     6.0], 'Single_Epi_Cell_Size': [2.0, 2.0, 10.0])
df['Total_Cell_Size'] = df['Cell_Size_Uniformity'] * df
     ['Single_Epi_Cell_Size']
```

```
Performance before adding features ROC 0.923, ACC
     0.985. Performance after adding features ROC
     0.932, ACC 0.991. Improvement ROC 0.008, ACC
     0.007. The code was executed and changes to df
     were kept.
```

Figure 2. Exemplary run of CAAFE on the Wisoconsin Breast Cancer. User-provided input is shown in blue, ML-classifier generated data shown in red and LLM generated code is shown with syntax highlighting. The generated code contains a comment per generated feature that follows a template provided in our prompt (Feature name, description of usefulness, features used in the generated code and sample values of these features). In this run, CAAFE improves the ROC AUC on the validation dataset from 0.0.923 to 0.932 in one feature engineering iteration. More iterations would be applied afterwards.

ple iterations of feature alterations and evaluations on the validation dataset, as outlined in Figure 1. In each iteration, the LLM generates code, which is then executed on the current D_{train} and D_{valid} resulting in the transformed datasets D'_{train} and D'_{valid}. We then use D'_{train} to fit an ML-classifier and evaluate its performance P' on D'_{valid}. If P' exceeds the performance P achieved by training on D_{train} and evaluating on D_{valid}, the feature is kept and we set $D_{train} := D'_{train}$ and $D_{valid} := D'_{valid}$. Otherwise, the feature is rejected and D_{train} and D_{valid} remain unchanged. Figure 2 shows a shortened version of one such run on the Tic-Tac-Toe Endgame dataset.

Prompting LLMs for Feature Engineering Code Here, we describe how CAAFE builds the prompt that is used to perform feature engineering. In this prompt, the LLM is instructed to create valuable features for a subsequent prediction task and to provide justifications for the added feature's utility. It is also instructed to drop unnecessary features, e.g. when their information is captured by other created features.

The prompt contains semantic and descriptive information about the dataset. Descriptive information, i.e. summary statistics, such as the percentage of missing values is based solely on the train split of the dataset. The prompt consists of the following data points:

A A user-generated dataset description, that contains contextual information about the dataset (see Section 7 for details on dataset descriptions for our experiments)

B Feature names adding contextual information and allowing the LLM to generate code to index features by their names

C Data types (e.g. float, int, category, string) - this adds information on how to handle a feature in the generated code

D Percentage of missing values - missing values are an additional challenge for code generation

E 10 rows of sample data from the dataset - this provides information on the feature scale, encoding, etc.

Additionally, the prompt provides a template for the expected form of the generated code and explanations. Adding a template when prompting is a common technique to improve the quality of responses (OpenAI, 2023b). We use Chain-of-thought instructions – instructing a series of intermediate reasoning steps –, another effective technique for prompting (Wei et al., 2023). The prompt includes an example of one such Chain-of-thought for the code generation of one feature: first providing the high-level meaning and usefulness of the generated feature, providing the names of features used to generate it, retrieving sample values it would need to accept and finally writing a line of code. We provide the complete prompt in Figure 3 in the appendix.

If the execution of a code block raises an error, this error is passed to the LLM for the next code generation iteration. We observe that using this technique CAAFE recovered from all errors in our experiments. One such example can be found in Table 2.

4. Results

In this section we showcase the results of our method in three different ways. For the detailed experimental setup please see Appendix 7. First, we show that CAAFE can improve the performance of a state-of-the-art classifier. Next, we show how CAAFE interacts with traditional automatic feature engineering methods and conclude with examples of the features that CAAFE creates.

Performance of CAAFE CAAFE can improve our strongest classifier, TabPFN, substantially. If it is used with GPT-4, we improve average ROC AUC performance from 0.798 to 0.822, and enhance the performance for 11/14

Table 1. ROC AUC OVO results using TabPFN. ± indicates the standard deviation across 5 splits. [R] indicates datasets where reduced data was used because TabPFN had 100% accuracy by default, see Appendix 11.1.

	TabPFN		
	No Feat. Eng.	CAAFE (GPT-3.5)	CAAFE (GPT-4)
airlines	**0.6211** ±.04	0.619 ±.04	0.6203 ±.04
balance-scale [R]	0.8444 ±.29	0.844 ±.31	**0.882** ±.26
breast-w [R]	0.9783 ±.02	**0.9809** ±.02	**0.9809** ±.02
cmc	0.7375 ±.02	0.7383 ±.02	**0.7393** ±.02
credit-g	0.7824 ±.03	0.7824 ±.03	**0.7832** ±.03
diabetes	0.8427 ±.03	**0.8434** ±.03	0.8425 ±.03
eucalyptus	**0.9319** ±.01	0.9317 ±.01	**0.9319** ±.00
jungle_chess..	0.9334 ±.01	0.9361 ±.01	**0.9453** ±.01
pc1	0.9035 ±.01	0.9087 ±.02	**0.9093** ±.01
tic-tac-toe [R]	0.6989 ±.08	0.6989 ±.08	**0.9536** ±.06
⟨Kaggle⟩ health-insurance	0.5745 ±.02	0.5745 ±.02	**0.5748** ±.02
⟨Kaggle⟩ pharyngitis	0.6976 ±.03	0.6976 ±.03	**0.7078** ±.04
⟨Kaggle⟩ kidney-stone	0.7883 ±.04	0.7873 ±.04	**0.7903** ±.04
⟨Kaggle⟩ spaceship-titanic	0.838 ±.02	0.8383 ±.02	**0.8405** ±.02
Mean ROC AUC	0.798 ±.05	0.7987 ±.05	**0.8215** ±.04
Mean ROC AUC Rank	2.43	2.32	**1.25**

datasets.On the evaluated datasets, this improvement is similar (71%) to the average improvement achieved by using a random forest (AUC 0.783) instead of logistic regression (AUC 0.749). We can see that CAAFE even improves performance for all of the new datasets from Kaggle. If we use CAAFE with GPT-3.5 only, we can see that it performs clearly worse than with GPT-4, and only improves performance on 6/14.

There is great variability in the improvement size depending on whether (1) a problem is amenable to feature engineering, i.e. is there a mapping of features that explains the data better and that can be expressed through simple code; and (2) the quality of the dataset description (e.g., the balance-scale dataset contains an accurate description of how the dataset was constructed) Per dataset performance can be found in Table 3. CAAFE takes 4:43 minutes to run on each dataset, 90% of the time is spent on the LLM's code generation and 10% on the evaluation of the generated features. Running CAAFE with 10 iterations costs 0.71$ per dataset on average. For the 14 datasets, 5 splits and 10 CAAFE iterations, CAAFE generates 52 faulty features (7.4%) in the generation stage, from which it recovers (see Figure 2).

Feature Engineering Strategies CAAFE shows a diverse set of examples of feature engineering strategies applied by our method. We show examples in the Appendix in Table2 where CAAFE combines features, creates ordinal versions of numerical features through binning, performs string transformations, removes superfluous features, and even recovers from errors when generating invalid code.

5. Conclusion

Our study presents a novel approach to integrating domain knowledge into the AutoML process through Context-Aware Automated Feature Engineering (CAAFE). By leveraging the power of large language models, CAAFE automates feature engineering for tabular datasets, generating semantically meaningful features and explanations of their utility. Our evaluation demonstrates the effectiveness of this approach, which complements existing automated feature engineering and AutoML methods.

This work emphasizes the importance of context-aware solutions in achieving robust outcomes. We expect that LLMs will also be useful for automating other aspects of the data science pipeline, such as data collection, processing, model building, and deployment. As large language models continue to improve, it is expected that the effectiveness of CAAFE will also increase.

Dataset descriptions play a critical role in our method; however, in our study, they were derived solely from web-crawled text associated with public datasets. If users were to provide more accurate and detailed descriptions, the effectiveness of our approach could be significantly improved.

However, our current approach has some limitations. Handling datasets with a large number of features can lead to very large prompts, which can be challenging for LLMs to process effectively. The testing procedure for adding features is not based on statistical tests, and could be improved using techniques of previous feature engineering works. Finally, the usage of LLMs in automated data analysis comes with a set of societal and ethical challenges. Please see Section 6 for a discussion on safeguarding code execution, biases in LLMs and societal implications of automation.

Future research may explore prompt tuning, fine-tuning language models, and automatically incorporating domain-knowledge into models in other ways. Also, there may lie greater value in the interaction of human users with such automated methods, also termed human-in-the-loop AutoML (Lee & Macke, 2020), where human and algorithm interact continuously. This would be particularly easy with a setup similar to CAAFE, as the input and output of the LLM are interpretable and easily modified by experts.

6. Broader Impact Statement

Social Impact of Automation The broader implications of our research may contribute to the automation of data science tasks, potentially displacing workers in the field. However, CAAFE crucially depends on the users inputs for feature generation and processing and provides an example of human-in-the-loop AutoML. The automation of routine tasks could free up data scientists to focus on higher-level problem-solving and decision-making activities. It is essential for stakeholders to be aware of these potential consequences, and to consider strategies for workforce education and adaptation to ensure a smooth transition as AI

technologies continue to evolve.

Replication of Biases AI algorithms have been observed to replicate and perpetuate biases observed in their training data distribution. CAAFE leverages GPT-4, which has been trained on web crawled data that contains existing social biases and generated features may be biased on these biases. When data that contains demographic information or other data that can potentially be used to discriminate against groups, we advise not to use CAAFE or to proceed with great caution, double checking the generated features.

Execution of AI-generated Code The automatic execution of AI-generated code carries inherent risks, such as misuse by malicious actors or unintended consequences from AI systems operating outside of controlled environments. Our approach is informed by previous studies on AI code generation and cybersecurity (Rohlf, 2023; Crockett, 2023). We parse the syntax of the generated python code and use a whitelist of operations that are allowed for execution. Thus operations such as imports, arbitrary function calls and others are excluded. This does not provide full security, however, e.g. does not exclude operations that can lead to infinite loops and excessive resource usage such as loops and list comprehensions.

AI Model Interpretability As the adoption of advanced AI methods grows, it becomes increasingly important to comprehend and interpret their results. Our approach aims to enhance interpretability by providing clear explanations of model outputs and generating simple code, thus making the automated feature engineering process more transparent.

Risk of increasing AI capabilities We do not believe this research affects the general capabilities of LLMs but rather demonstrates their application. As such we estimate our work does not contribute to the risk of increasing AI capabilities.

References

Anaconda. The state of data science 2020. Website, 2020. URL https://www.anaconda.com/state-of-data-science-2020.

Bouthillier, X., Delaunay, P., Bronzi, M., Trofimov, A., Nichyporuk, B., Szeto, J., Mohammadi Sepahvand, N., Raff, E., Madan, K., Voleti, V., Ebrahimi Kahou, S., Michalski, V., Arbel, T., Pal, C., Varoquaux, G., and Vincent, P. Accounting for variance in machine learning benchmarks. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 747–769, 2021. URL https://proceedings.mlsys.org/paper_files/paper/2021/file/cfecdb276f634854f3ef915e2e980c31-Paper.pdf.

Breiman, L. Random forests. *Machine learning*, 45:5–32, 2001.

Crockett, A. Ai generated code creates a new security attack vector, April 2023. URL https://dev.to/adam_cyclones/ai-generated-code-creates-a-new-security-attack-v

De Bie, T., De Raedt, L., Hernández-Orallo, J., Hoos, H. H., Smyth, P., and Williams, C. K. Automating data science. *Communications of the ACM*, 65(3):76–87, 2022.

Feurer, M., van Rijn, J. N., Kadra, A., Gijsbers, P., Mallik, N., Ravi, S., Mueller, A., Vanschoren, J., and Hutter, F. Openml-python: an extensible python api for openml. *arXiv*, 1911.02490. URL https://arxiv.org/pdf/1911.02490.pdf.

Hegselmann, S., Buendia, A., Lang, H., Agrawal, M., Jiang, X., and Sontag, D. Tabllm: Few-shot classification of tabular data with large language models, 2023.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. *CoRR*, abs/2103.03874, 2021. URL https://arxiv.org/abs/2103.03874.

Hollmann, N., Müller, S., Eggensperger, K., and Hutter, F. Tabpfn: A transformer that solves small tabular classification problems in a second. *arXiv preprint arXiv:2207.01848*, 2022.

Hutter, F., Kotthoff, L., and Vanschoren, J. (eds.). *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2019. Available for free at http://automl.org/book.

Lee, D. J.-L. and Macke, S. A human-in-the-loop perspective on automl: Milestones and the road ahead. *IEEE Data Engineering Bulletin*, 2020.

Narayan, A., Chami, I., Orr, L., Arora, S., and Ré, C. Can foundation models wrangle your data?, 2022.

OpenAI. Gpt-4 technical report, 2023a.

OpenAI. openai/openai-cookbook: Examples and guides for using the openai api. https://github.com/openai/openai-cookbook, 2023b. (Accessed on 04/20/2023).

OpenAI Community. GPT-3 can't count syllables - or doesn't "get" haiku. https://community.openai.com/t/gpt-3-cant-count-syllables-or-doesnt-get-haiku/18733, 2021. Accessed on: 2023-03-21.

Rohlf, C. Ai code generation and cybersecurity, April 2023. URL https://www.cfr.org/blog/ai-code-generation-and-cybersecurity.

Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. doi: 10.1145/2641190.2641198. URL http://doi.acm.org/10.1145/2641190.2641198.

Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. Openml: networked science in machine learning. *CoRR*, abs/1407.7722, 2014. URL http://arxiv.org/abs/1407.7722.

Vos, D., Döhmen, T., and Schelter, S. Towards parameter-efficient automation of data wrangling tasks with prefix-tuning. In *NeurIPS 2022 First Table Representation Workshop*, 2022. URL https://openreview.net/forum?id=8kyYJs2YkFH.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023.

Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

Wold, S., Esbensen, K., and Geladi, P. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

7. Experimental Setup

Technical Setup The data is stored in a Pandas dataframe (Wes McKinney, 2010), which is preloaded into memory for code execution. The generated Python code is executed in an environment where the training and validation data frame is preloaded. The performance is measured on the current dataset with ten random validation splits D_{valid} and the respective transformed datasets D'_{valid} with the mean change of accuracy and ROC AUC used to determine if the changes of a code block are kept, i.e. when the average of both is greater than 0. We use OpenAI's GPT-4 and GPT-3.5 as LLMs (OpenAI, 2023a) in CAAFE. We perform ten feature engineering iterations and TabPFN in the iterative evaluation of code blocks.

Setup of Downstream-Classifiers We evaluate our method with Logistic Regression, Random Forests (Breiman, 2001) and TabPFN (Hollmann et al., 2022) for the final evaluation while using TabPFN to evaluate the performance of added features. We impute missing values with the mean, one-hot or ordinal encoded categorical inputs, normalized features and passed categorical feature indicators, where necessary, using the setup of Hollmann et al. (2022) [1].

Evaluating LLMs on Tabular Data The LLM's training data originates from the web, potentially including datasets and related notebooks. GPT-4 and GPT-3.5 have a knowledge cutoff in September 2021, i.e., almost all of its training data originated from before this date. Thus, an evaluation on established benchmarks can be biased since a textual description of these benchmarks might have been used in the training of the LLM.

We use two categories of datasets for our evaluation: (1) widely recognized datasets from OpenML released before September 2021, that could potentially be part of the LLMs training corpus and (2) lesser known datasets from Kaggle released after September 2021 and only accessible after accepting an agreement and thus harder to access by web crawlers.

From OpenML (Vanschoren et al., 2013; Feurer et al.), we use small datasets that have descriptive feature names (i.e. we do not include any datasets with numbered feature names). Datasets on OpenML contain a task description that we provide as user context to our method. When datasets are perfectly solvable with TabPFN alone (i.e. reaches ROC AUC of 1.0) we reduce the training set size for that dataset. We focus on small datasets with up to 2 000 samples in total, because feature engineering is most important and significant for smaller datasets.

We describe the collection and preprocessing of datasets in detail in Appendix 11.1.

Evaluation Protocol For each dataset, we evaluate 5 repetitions, each with a different random seed and train- and test split to reduce the variance stemming from these splits (Bouthillier et al., 2021). We split into 50% train and 50% test samples and all methods used the same splits.

8. Reproducibility

Code release In an effort to ensure reproducibility, we release code to reproduce our experiments at `https://github.com/cafeautomatedfeatures/CAFE`

Availability of datasets All datasets used in our experiments are freely available at `OpenML.org` (Vanschoren et al., 2014) or at `kaggle.com`, with downloading procedures included in the submission.

9. Full LLM Prompt

Figure 3 shows the full prompt for one examplary dataset. The generated prompts are in our repository: `https://github.com/cafeautomatedfeatures/CAFE/tree/main/data/generated_code`

[1] `https://github.com/automl/TabPFN/blob/main/tabpfn/scripts/tabular_baselines.py`

```
The dataframe ‘df‘ is loaded and in memory. Columns are also named attributes.
Description of the dataset in ‘df‘ (column dtypes might be inaccurate):
"**Tic-Tac-Toe Endgame database**
This database encodes the complete set of possible board configurations at the end of tic-tac-toe games, where "x"
    is assumed to have played first.  The target concept is "win for x" (i.e., true when "x" has one of 8 possible
    ways to create a "three-in-a-row").  "

Columns in ‘df‘ (true feature dtypes listed here, categoricals encoded as int):
top-left-square (int32): NaN-freq [0.0%], Samples [2, 2, 2, 2, 2, 2, 0, 1, 1, 2]
top-middle-square (int32): NaN-freq [0.0%], Samples [0, 0, 1, 1, 1, 2, 0, 2, 2, 2]
top-right-square (int32): NaN-freq [0.0%], Samples [1, 0, 1, 2, 1, 1, 1, 0, 2, 1]
middle-left-square (int32): NaN-freq [0.0%], Samples [1, 0, 2, 1, 2, 0, 0, 2, 1, 2]
middle-middle-square (int32): NaN-freq [0.0%], Samples [0, 2, 2, 1, 2, 1, 1, 1, 2, 1]
middle-right-square (int32): NaN-freq [0.0%], Samples [1, 1, 2, 2, 2, 2, 0, 0, 0, 0]
bottom-left-square (int32): NaN-freq [0.0%], Samples [2, 1, 1, 0, 0, 1, 2, 0, 1, 1]
bottom-middle-square (int32): NaN-freq [0.0%], Samples [2, 0, 0, 0, 1, 2, 2, 2, 1, 0]
bottom-right-square (int32): NaN-freq [0.0%], Samples [2, 2, 0, 2, 0, 1, 2, 1, 2, 0]
Class (category): NaN-freq [0.0%], Samples [1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0]


This code was written by an expert datascientist working to improve predictions. It is a snippet of code that adds
    new columns to the dataset.
Number of samples (rows) in training dataset: 71

This code generates additional columns that are useful for a downstream classification algorithm (such as XGBoost)
    predicting "Class".
Additional columns add new semantic information, that is they use real world knowledge on the dataset. They can e.g.
     be feature combinations, transformations, aggregations where the new column is a function of the existing
    columns.
The scale of columns and offset does not matter. Make sure all used columns exist. Follow the above description of
    columns closely and consider the datatypes and meanings of classes.
This code also drops columns, if these may be redundant and hurt the predictive performance of the downstream
    classifier (Feature selection). Dropping columns may help as the chance of overfitting is lower, especially if
    the dataset is small.
The classifier will be trained on the dataset with the generated columns and evaluated on a holdout set. The
    evaluation metric is accuracy. The best performing code will be selected.
Added columns can be used in other codeblocks, dropped columns are not available anymore.

Code formatting for each added column:
```python
# (Feature name and description)
# Usefulness: (Description why this adds useful real world knowledge to classify "Class" according to dataset
    description and attributes.)
# Input samples: (Three samples of the columns used in the following code, e.g. 'top-left-square': [2, 2, 2], 'top-
    middle-square': [0, 0, 1], ...)
(Some pandas code using top-left-square', 'top-middle-square', ... to add a new column for each row in df)
```end

Code formatting for dropping columns:
```python
# Explanation why the column XX is dropped
df.drop(columns=['XX'], inplace=True)
```end

Each codeblock generates exactly one useful column and can drop unused columns (Feature selection).
Each codeblock ends with ```end and starts with "```python"
Codeblock:
```

Figure 3. Full LLM Prompt for the CMC dataset. The generated code will be the reply to this prompt.

10. Additional Results

Table 2. Examples of common strategies employed by CAAFE for feature extension. The full code and comments are automatically generated based on the user-provided dataset descriptions.

Description	Generated code

Combination
Example from the Kaggle Kidney Stone dataset.

```
# Usefulness: Fever and rhinorrhea are two of the most common symptoms of
    respiratory infections, including GAS pharyngitis. This feature captures their
    co-occurrence.
# Input samples: 'temperature': [38.0, 39.0, 39.5], 'rhinorrhea': [0.0, 0.0, 0.0]
df['fever_and_rhinorrhea'] = ((df['temperature'] >= 38.0) & (df['rhinorrhea'] > 0))
    .astype(int)
```

Binning
Example from the Kaggle Spaceship Titanic dataset.

```
# Feature: AgeGroup (categorizes passengers into age groups)
# Usefulness: Different age groups might have different likelihoods of being
    transported.
# Input samples: 'Age': [30.0, 0.0, 37.0]
bins = [0, 12, 18, 35, 60, 100]
labels = ['Child', 'Teen', 'YoungAdult', 'Adult', 'Senior']
df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels)
df['AgeGroup'] = df['AgeGroup'].astype('category')
```

String transformation
Example from the Kaggle Spaceship Titanic dataset.

```
# Feature: Deck
# Usefulness: The deck information can help identify patterns in the location of
    cabins associated with transported passengers.
# Input samples: 'Cabin': ['F/356/S', 'G/86/P', 'C/37/P']
df['Deck'] = df['Cabin'].apply(lambda x: x[0] if isinstance(x, str) else 'Unknown')

# Feature: CabinSide
# Usefulness: The side of the cabin can help identify patterns in the location of
    cabins associated with transported passengers.
# Input samples: 'Cabin': ['F/356/S', 'G/86/P', 'C/37/P']
df['CabinSide'] = df['Cabin'].apply(lambda x: x.split('/')[-1] if isinstance(x, str
    ) else 'Unknown')
```

Removing features
Example from the Balance Scale dataset.

```
# Drop original columns
# Explanation: The original columns 'left-weight', 'left-distance', 'right-weight',
    and 'right-distance' can be dropped as the newly created columns 'left_moment
    ', 'right_moment', and 'moment_difference' capture the relevant information
    for predicting the 'class'.
df.drop(columns=['left-weight', 'left-distance', 'right-weight', 'right-distance'],
    inplace=True)
# No more additional columns needed
```

Error Recovery
Example from the Kaggle Health Insurance dataset. The code execution failed because missing values were not considered in the dataframe. In the next iteration code is suggested that can handle missing values.

```
# Feature: Age_difference
# Usefulness: Age difference between upper and lower age can be useful in
    determining the likelihood of a person applying for a policy.
# Input samples: 'Upper_Age': ['C33', 'C2', 'C3'], 'Lower_Age': ['Owned', 'Rented',
    'Rented']
df['Age_difference'] = df['Upper_Age'].str.extract('(\d+)').astype(int) - df['
    Lower_Age'].str.extract('(\d+)').astype(int)

Feedback: failed with error: cannot convert float NaN to integer.

# Feature: Age_difference
# Usefulness: Age difference between upper and lower age can be useful in
    determining the likelihood of a person applying for a policy.
# Input samples: 'Upper_Age': ['C33', 'C2', 'C3'], 'Lower_Age': ['Owned', 'Rented',
    'Rented']
df['Age_difference'] = df['Upper_Age'].str.extract('(\d+)').astype(float).fillna(0)
    - df['Lower_Age'].str.extract('(\d+)').astype(float).fillna(0)
```

11. Datasets

Name	# Features	# Samples	# Classes	OpenML ID / Kaggle Name
balance-scale	4	125	3	11
breast-w	9	69	2	15
cmc	9	1473	3	23
credit-g	20	1000	2	31
diabetes	8	768	2	37
tic-tac-toe	9	95	2	50
eucalyptus	19	736	5	188
pc1	21	1109	2	1068
airlines	7	2000	2	1169
jungle_chess_2pcs_raw_endgame_complete	6	2000	3	41027
pharyngitis	19	512	2	*pharyngitis*
health-insurance	13	2000	2	*health-insurance-lead-prediction-raw-data*
spaceship-titanic	13	2000	2	*spaceship-titanic*
kidney-stone	7	414	2	*playground-series-s3e12*

Table 3. Test datasets used for the evaluation. See Section 7 for a description of the datasets used.

11.1. Dataset Collection and Preprocessing

OpenML datasets We use small datasets from OpenML (Vanschoren et al., 2013; Feurer et al.) that have descriptive feature names (i.e. we do not include any datasets with numbered feature names). Datasets on OpenML contain a task description that we provide as user context to our method and that we clean from redundant information for feature engineering, such as author names or release history. While some descriptions are very informative, other descriptions contain much less information. We remove datasets with more than 20 features, since the prompt length rises linearly with the number of features and exceeds the permissible 8,192 tokens that standard GPT-4 can accept. We show all datasets we used in Table 3 in Appendix 11. When datasets are perfectly solvable with TabPFN alone (i.e. reaches ROC AUC of 1.0) we reduce the training set size for that dataset to 10% or 20% of the original dataset size. This is the case for the datasets "balance-scale" (20%), "breast-w" (10%) and "tic-tac-toe" (10%). We focus on small datasets with up to 2 000 samples in total, because feature engineering is most important and significant for smaller datasets.

Kaggle datasets We additionally evaluate CAAFE on 4 datasets from Kaggle that were released after the knowledge cutoff of our LLM Model. These datasets contain string features as well. String features allow for more complex feature transformations, such as separating Names into First and Last Names, which allows grouping families. We drop rows that contain missing values for our evaluations. Details of these datasets can also be found in Table 3 in Appendix 11.

11.2. Dataset Descriptions

The dataset descriptions used were crawled from the respective datasource. For OpenML prompts uninformative information such as the source or reference papers were removed.