# Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering

**Noah Hollmann**
University of Freiburg
Charité Hospital Berlin
Prior Labs
noah.homa@gmail.com

**Samuel Müller**
University of Freiburg
Prior Labs
muellesa@cs.uni-freiburg.de

**Frank Hutter**
University of Freiburg
Prior Labs
fh@cs.uni-freiburg.de

## Abstract

As the field of automated machine learning (AutoML) advances, it becomes increasingly important to incorporate domain knowledge into these systems. We present an approach for doing so by harnessing the power of large language models (LLMs). Specifically, 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, a simple demo and a python package.

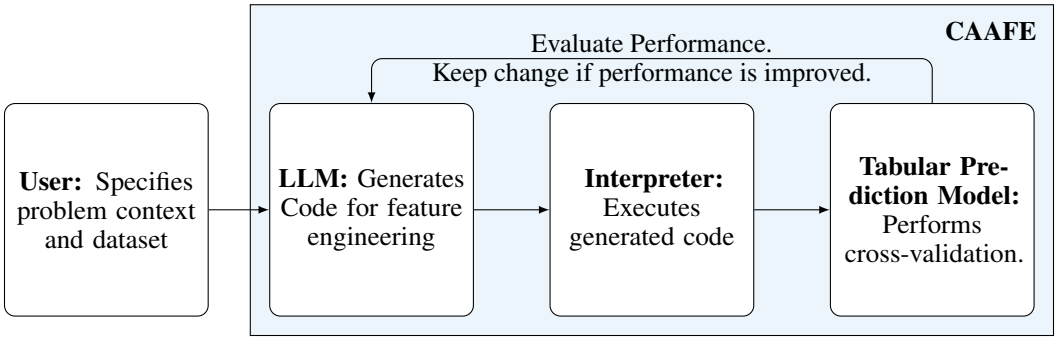


Figure 1: CAAFE accepts a dataset as well as user-specified context information and operates by iteratively proposing and evaluating feature engineering operations.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

# 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.

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, as visualized in Figure 2. We bridge the gap between LLMs and classical algorithms by using code as an interface between them: LLMs generate code that modifies input datasets, these modified datasets can then be processed by classical algorithms. Our proposed 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, CAAFE generates a comment for each feature which explains the utility of generated feature. This allows interpretable AutoML, 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). The architecture of GPT-4 is not oublished, it is likely based on a deep neural network that uses a transformer architecture (Vaswani et al., 2017), large-scale pre-training on a diverse corpus of text and fine-tuning using reinforcement learning from human feedback (RLHF) (Ziegler et al., 2019). It achieves state-of-the-art performance on various tasks, such as text generation, summarization, question answering and coding. One can adapt LLMs to a specific task without retraining by writing a prompt (Brown et al., 2020; Wei et al., 2021); the model parameters are frozen and the model performs in-context inference tasks based on a textual input that formulates the task and potentially contains examples.

**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

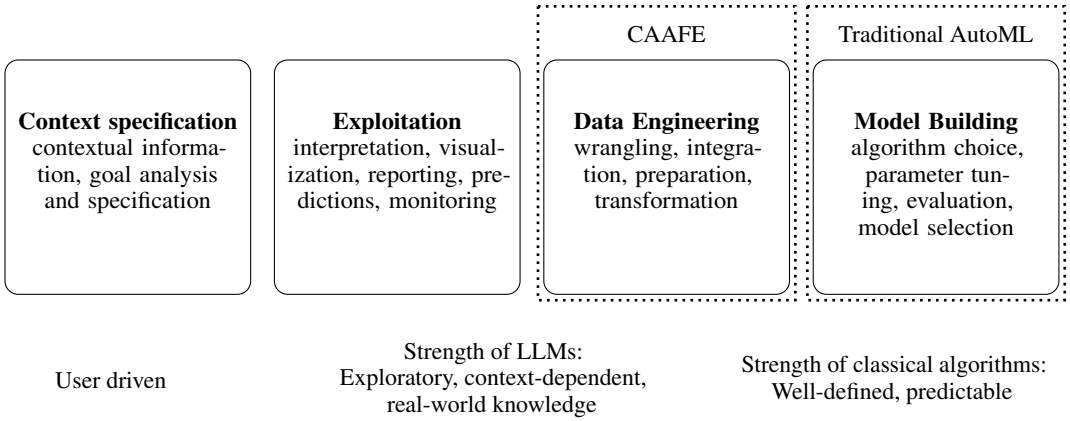


Figure 2: Data Science pipeline, inspired by De Bie et al. (2022). CAAFE allows for automation of semantic data engineering, while LLMs could provide even further automation: (1) Context specification is user driven (2) exploitation and data engineering can be automated through LLMs (3) model building can be automated by classical AutoML approaches.

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 samples, 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).

Deep learning methods are capable of learning suitable transformations from the raw input data making them more data-driven and making explicit feature engineering less critical, but only given a lot of data. Thus, appropriate feature engineering still improves the performance of classical and deep learning models, particularly for limited data, complex patterns, or model interpretability.

Various strategies for automated feature engineering have been explored in prior studies. Deep Feature Synthesis (DFS; Kanter & Veeramachaneni (2015)) integrates multiple tables for feature engineering by enumerating potential transformations on features and performing feature selection based on model performance. Cognito (Khurana et al., 2016) proposes a tree-like exploration of the feature space using handcrafted heuristic traversal strategies. AutoFeat (Horn et al., 2019) employs an iterative subsampling of features using beam search. Learning-based methods, such as LFE (Nargesian et al., 2017), utilize machine learning models to recommend beneficial transformations while other methods use reinforcement learning-based strategies (Khurana et al., 2018; Zhang et al., 2019). Despite these advancements, none of the existing methods can harness semantic information in an automated manner.

### 2.2.1 Incorporating Semantic Information

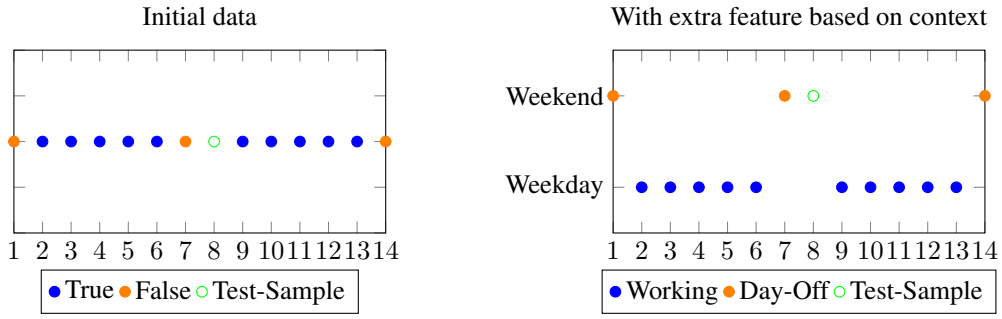


Figure 3: Contextual information can simplify a task immensely. On the left-hand side no contextual information is added to the plot, and it is hard to predict the label for the green query point. On the right-hand side contextual information is added and a useful additional feature (weekend or weekday) is derived from which a mapping from features to targets can be found.

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. Figure 3 exemplifies the usefulness of contextual information.

## 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 multiple 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 4 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 4 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

```
Dataset description: 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").
```

```
# ('number-of-x-wins', 'Number of ways x can win on the board')
# Usefulness: Knowing the number of ways x can win on the board can be useful in
    predicting whether x has won the game or not.
# Input samples: 'top-left-square': [2, 2, 1], 'top-middle-square': [1, 2, 0], ...
df['number-of-x-wins'] = ((df['top-left-square']==1) & (df['top-middle-square']==1) & (df
    ['top-right-square']==1)).astype(int) + ((df['middle-left-square']==1) & (df['middle
    -middle-square']==1) & (df['middle-right-square']==1)).astype(int) [...]
```

```
Iteration 1
Performance before adding features ROC 0.888, ACC 0.700.
Performance after adding features ROC 0.987, ACC 0.980.
Improvement ROC 0.099, ACC 0.280. Code was executed and changes to df
    retained.
```

```
# ('number-of-o-wins', 'Number of ways o can win on the board')
# Usefulness: Knowing the number of ways o can win on the board can be useful in
    predicting whether o has won the game or not.
# Input samples: 'top-left-square': [2, 2, 1], 'top-middle-square': [1, 2, 0], ...
df['number-of-o-wins'] = ((df['top-left-square']==2) & (df['top-middle-square']==2) & (df
    ['top-right-square']==2)).astype(int) + ((df['middle-left-square']==2) & (df['middle
    -middle-square']==2) & (df['middle-right-square']==2)).astype(int) [...]
```

```
Iteration 2
Performance before adding features ROC 0.987, ACC 0.980.
Performance after adding features ROC 1.000, ACC 1.000.
Improvement ROC 0.013, ACC 0.020. Code was executed and changes to df
    retained.
```

Figure 4: Exemplary run of CAAFE on the Tic-Tac-Toe Endgame dataset. User generated 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 0.888 to 1.0 in two feature engineering iterations.

**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 random rows 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 5 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 3.

**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 (Hollmann et al., 2022) in the iterative evaluation of code blocks.

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.

## 4 Experimental Setup

**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].

**Setup of Automated Feature Engineering Methods** We also evaluate popular context-agnostic feature engineering libraries Deep Feature Synthesis (DFS; Kanter & Veeramachaneni, 2015) and AutoFeat (Horn et al., 2019)[2]. We evaluate DFS and AutoFeat alone and in combination with CAAFE. When combined, CAAFE is applied first and the context-agnostic AutoFE method subsequently. For DFS we use the primitives "add_numeric" and "multiply_numeric", and default settings otherwise. For TabPFN, DFS generates more features than TabPFN accepts (the maximum number of features is 100) in some cases. In these cases, we set the performance to the performance without feature engineering. For AutoFeat, we use one feature engineering step and default settings otherwise.

**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, marked in Table 1. 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 G.1.

[1] https://github.com/automl/TabPFN/blob/main/tabpfn/scripts/tabular_baselines.py
[2] https://github.com/alteryx/featuretools, https://github.com/cod3licious/autofeat

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

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

**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.

## 5 Results

In this section we showcase the results of our method in three different ways. 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.

Table 2: Mean ROC AUC and average rank (ROC AUC) per downstream classification method and feature extension method. Best AutoFE method per base classifer is shown in bold. The features generated by CAAFE are chosen with TabPFN as classifier. Rank are calculated across all classifiers and feature engineering methods. FETCH was too computationally expensive to compute for all base classifiers in the rebuttal. Each seed and dataset takes up to 24 hours and has to be evaluated for each base classifer independently. Thus, we use features computed for logistic regression for all other classifiers.

| | | | Baselines | | | | CAAFE | |
|---|---|---|---|---|---|---|---|---|
| | | No FE | DFS | AutoFeat | FETCH | OpenFE | GPT-3.5 | GPT-4 |
| Log. Reg. | Mean | 0.749 | 0.764 | 0.754 | 0.76 | 0.757 | 0.763 | **0.769** |
| | Mean Rank | 27.4 | **23.6** | 26.2 | 25.2 | 25 | 24.8 | 24.3 |
| Random Forest | Mean | 0.782 | 0.783 | 0.783 | 0.785 | 0.785 | 0.79 | **0.803** |
| | Mean Rank | 23.4 | 22.1 | 21.8 | 23.5 | 22.3 | 23.1 | **19.9** |
| ASKL2 | Mean | 0.807 | 0.801 | 0.808 | 0.807 | 0.806 | 0.815 | **0.818** |
| | Mean Rank | 12.2 | 12.9 | 12.6 | 13.4 | 13.5 | **10.9** | 11.6 |
| Autogluon | Mean | 0.796 | 0.799 | 0.797 | 0.787 | 0.798 | 0.803 | **0.812** |
| | Mean Rank | 17.6 | 15.4 | 16.4 | 17.6 | 16.6 | 15.8 | **14.1** |
| TabPFN | Mean | 0.798 | 0.791 | 0.796 | 0.796 | 0.798 | 0.806 | **0.822** |
| | Mean Rank | 13.9 | 15 | 14.8 | 16.5 | 13.9 | 12.9 | **9.78** |

**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, as shown in Table 2, and enhance the performance for 11/14 datasets.On the evaluated datasets, this improvement

Table 3: 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. 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.

| Description | Generated code |
|---|---|
| **Combination**<br>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.`<br>`# Input samples: 'temperature': [38.0, 39.0, 39.5], 'rhinorrhea': [0.0, 0.0, 0.0]`<br>`df['fever_and_rhinorrhea'] = ((df['temperature'] >= 38.0) & (df['rhinorrhea'] > 0)).astype(int)` |
| **Binning**<br>Example from the Kaggle Spaceship Titanic dataset. | `# Feature: AgeGroup (categorizes passengers into age groups)`<br>`# Usefulness: Different age groups might have different likelihoods of being transported.`<br>`# Input samples: 'Age': [30.0, 0.0, 37.0]`<br>`bins = [0, 12, 18, 35, 60, 100]`<br>`labels = ['Child', 'Teen', 'YoungAdult', 'Adult', 'Senior']`<br>`df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels)`<br>`df['AgeGroup'] = df['AgeGroup'].astype('category')` |
| **String transformation**<br>Example from the Kaggle Spaceship Titanic dataset. | `# Feature: Deck`<br>`# Usefulness: The deck information can help identify patterns in the location of cabins associated with transported passengers.`<br>`# Input samples: 'Cabin': ['F/356/S', 'G/86/P', 'C/37/P']`<br>`df['Deck'] = df['Cabin'].apply(lambda x: x[0] if isinstance(x, str) else 'Unknown')`<br><br>`# Feature: CabinSide`<br>`# Usefulness: The side of the cabin can help identify patterns in the location of cabins associated with transported passengers.`<br>`# Input samples: 'Cabin': ['F/356/S', 'G/86/P', 'C/37/P']`<br>`df['CabinSide'] = df['Cabin'].apply(lambda x: x.split('/')[-1] if isinstance(x, str) else 'Unknown')` |
| **Removing features**<br>Example from the Balance Scale dataset. | `# Drop original columns`<br>`# 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'.`<br>`df.drop(columns=['left-weight', 'left-distance', 'right-weight', 'right-distance'], inplace=True)`<br>`# No more additional columns needed` |
| **Error Recovery**<br>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`<br>`# Usefulness: Age difference between upper and lower age can be useful in determining the likelihood of a person applying for a policy.`<br>`# Input samples: 'Upper_Age': ['C33', 'C2', 'C3'], 'Lower_Age': ['Owned', 'Rented', 'Rented']`<br>`df['Age_difference'] = df['Upper_Age'].str.extract('(\d+)').astype(int) - df['Lower_Age'].str.extract('(\d+)').astype(int)`<br><br>`Feedback: failed with error: cannot convert float NaN to integer.`<br><br>`# Feature: Age_difference`<br>`# Usefulness: Age difference between upper and lower age can be useful in determining the likelihood of a person applying for a policy.`<br>`# Input samples: 'Upper_Age': ['C33', 'C2', 'C3'], 'Lower_Age': ['Owned', 'Rented', 'Rented']`<br>`df['Age_difference'] = df['Upper_Age'].str.extract('(\d+)').astype(float).fillna(0) - df['Lower_Age'].str.extract('(\d+)').astype(float).fillna(0)` |

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 datasets.

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 1. 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. In Appendix F we plot the performance, time and cost of CAAFE across feature engineering iterations, showing the tradeoff between these parameters. 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 3).

**Incorporating Classical AutoFE Methods** Classical AutoFE methods can readily be combined with our method, one simply runs CAAFE first and then lets a classical AutoFE method find further feature extensions, as we did in Table 2. For less powerful downstream classifiers, namely Logistic Regression and Random Forests, we observe that applying AutoFE additionally to CAAFE improves performance further. The AutoML method TabPFN on the other hand is not improved by applying the evaluated AutoFE methods. This discrepancy might stem from the larger hypothesis space (complexity) of TabPFN, it can get all necessary information from the data directly. For all combinations of classifiers and additional AutoFE methods, we can see that CAAFE improves performance on average.

## 6 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. LLMs, at times, exhibit a phenomenon known as "hallucinations.", where models produce inaccurate or invented information. Within CAAFE, this might result in the generation of features and associated explanations that appear significant and are logically presented, even though they may not be grounded in reality. Such behavior can be problematic, especially when individuals place trust in these systems for essential decision-making or research tasks. Finally, the usage of LLMs in automated data analysis comes with a set of societal and ethical challenges. Please see Section B for a discussion on the broader impact and ethical considerations.

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.