SMARTFEAT: Efficient Feature Construction through Feature-Level Foundation Model Interactions

Yin Lin
University of Michigan
irenelin@umich.edu
USA

Bolin Ding
Alibaba Group
bolin.ding@alibaba-inc.com
USA

H. V. Jagadish
University of Michigan
jag@umich.edu
USA

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com
USA

ABSTRACT

Before applying data analytics or machine learning to a data set, a vital step is usually the construction of an informative set of features from the data. In this paper, we present SMARTFEAT, an efficient automated feature engineering tool to assist data users, even non-experts, in constructing useful features. Leveraging the power of Foundation Models (FMs), our approach enables the creation of new features from the data, based on contextual information and open-world knowledge. Our method incorporates an intelligent *operator selector* that discerns a subset of operators, effectively avoiding exhaustive combinations of original features, as is typically observed in traditional automated feature engineering tools. Moreover, we address the limitations of performing data tasks through row-level interactions with FMs, which could lead to significant delays and costs due to excessive API calls. We introduce a *function generator* that facilitates the acquisition of efficient data transformations, such as dataframe built-in methods or lambda functions, ensuring the applicability of SMARTFEAT to generate new features for large datasets. Code repo with prompt details and datasets: (https://github.com/niceIrene/SMARTFEAT).

1 INTRODUCTION

Machine learning (ML) plays a crucial role in myriad decision-making processes, ranging from automated policing [23] to medical diagnosis [14] and pricing plan development[5]. Raw data collected through data integration is seldom suitable for direct use for such machine learning (or any other data analytics): there is typically a need for appropriate data wrangling to construct high-quality features. This process is highly dependent on domain expertise and requires considerable manual effort by data scientists.

To mitigate manual labor, automated approaches have been developed as surveyed in [27]. However, traditional automatic feature engineering (AFE) methods typically rely on a predefined set of operators (e.g. scaling a single column, adding two columns), applied directly to the original dataset for generating new features [11]. Unfortunately, many of the features generated using these methods lack meaningful information and require significant effort
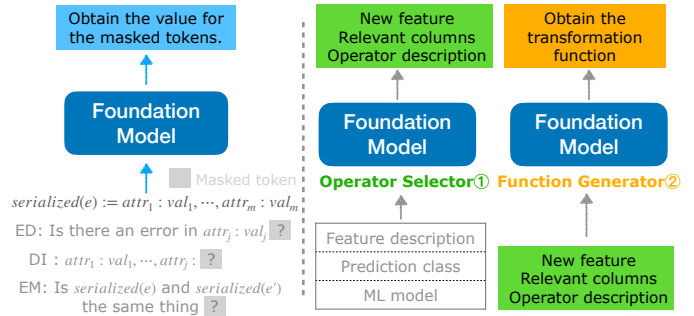
Figure 1: Comparison of row-level interactions to feature-level interactions.

for high-quality feature selection [27]. The limited flexibility in choosing appropriate operators and the inability to capture domain-specific information effectively has led researchers to explore new approaches that can leverage contextual information to enhance feature generation and selection. Foundation models, as we discuss next, are a natural choice for this purpose.

Recently, the emergence of foundation models (FMs) such as BERT [6], PaLM [4], and GPT [3] has brought significant advancements in many applications. These models, trained on extensive web-crawled data across diverse tasks, offer the unique capability of adaptation to new tasks without necessitating task-specific fine-tuning [3]. Consequently, data scientists have been exploring the potential of FMs in handling crucial data-related tasks, including data imputation (DI), error detection (ED), and entity matching (EM), demonstrating SoTA performance [19].

Incorporating FMs, designed to process natural language input and generate corresponding natural language output, into tasks of data manipulation is challenging. The prevailing strategy involves serializing and tokenizing each entry in the dataset and using FMs to predict the masked tokens, leveraging their ability to predict the next words as depicted in Figure 1. However, conducting row-level completions for large datasets can become impractical due to the time and financial costs associated with excessive interactions with FMs, hindering efficient and scalable applications.

In this paper, we seek to overcome this obstacle and improve the efficiency of integrating FMs into data tasks. Specifically, we explore performing feature-level interactions with FMs, aiming to transform

Table 1: Example data set (Prediction class: Safe 1=yes, 0=no).

Sex	Age	Age of car	Make, Model	Claim in last 6 month	City	Safe
M	21	6	Honda, Civic	1	SF	0
F	35	2	Toyota, Corolla	0	LA	1
M	42	8	Ford, Mustang	0	SEA	1
F	22	14	Chevrolet, Cruze	1	SF	0
M	45	3	BMW, X5	0	SEA	1
F	56	5	Volkswagen, Golf	0	LA	1

the natural language output of FMs into executable functions that can be applied to construct new features. We demonstrate our idea using the following example.

EXAMPLE 1.1. *Consider an insurance company that aims to determine the insurance rate for each customer based on the probability of them filing a new insurance claim within the next 6 months. The dataset in Table 1 includes information such as policyholder demographics, car details, location, and historical insurance records. The prediction class "safe" indicates whether a policyholder is considered safe and less likely to file an insurance claim within the next 6 months.*

To enhance prediction performance, we present SMARTFEAT, a practical AFE tool that leverages FMs to predict suitable transformations based on contextual information. SMARTFEAT's new features for the example may include:

(F1) *Bucketized Age: Groups individuals' ages into predefined bins using a bucketization function.*

(F2) *Manufacturing Year of the Car: Computes the difference between the car's age and the current year.*

(F3) *Claim Probability per Car Model: Provides the historical claim probability for each car model by grouping the data accordingly and calculating the average of 'Claim in last 6 months'.*

(F4) *City Population Density: Extracts population density information from the city feature.*

In comparison with traditional AFE tools, SMARTFEAT offers several compelling advantages:

- **Broader coverage of operators.** SMARTFEAT supports an extensive range of operators, surpassing machine learning-based recommendations [20], and pre-defined operators [10, 11]. The *operator-guided* prompt templates in SMARTFEAT enable consideration of an extensive set of candidate features. Moreover, leveraging encoded knowledge [19] from FMs enables the generation of diverse transformations, such as get_dummies, split, and user-defined functions. For instance, constructing the bucketization feature (F1) with user-defined boundaries can incorporate practical thresholds observed in real-world insurance quotes, like the frequently used threshold of 21 years old.

- **Better explainability and efficiency.** SMARTFEAT leverages contextual information within the dataset for feature generation. Given descriptions of the current feature set, SMARTFEAT efficiently interacts with FMs using both *"proposal"* and *"sampling"* strategies [26] to generate candidate features. Unlike traditional AFE tools that disregard contextual information and may generate numerous non-meaningful features, SMARTFEAT selectively considers relevant operators. For example, in the construction of feature (F2), SMARTFEAT exclusively considers the subtraction operator, discarding other binary operators.

- **Ability to generate highly correlated features.** By providing prompts that specify the prediction class and downstream ML model, SMARTFEAT tends to generate features that exhibit a notable correlation with the prediction class and are appropriate for downstream ML models. For instance, the creation of the *GroupbyThenAvg* feature (F3) illustrates the claim history of each car model, demonstrating a noteworthy correlation with the "Safe" attribute. Moreover, certain models like k-nearest-neighbors (KNN) tend to perform better when the data is normalized or has similar ranges [22].

- **Ability to access external data and resources.** SMARTFEAT can leverage external knowledge without explicitly incorporating external sources, as demonstrated in the construction of feature F4. Although FMs may have limitations on accessing real-time data, they can provide potential sources and APIs for users to utilize, as further demonstrated in Section 3.3.

In sum, this work makes the following contributions. Firstly, we present an AFE tool that integrates FMs, enhancing the effectiveness of feature engineering. By leveraging contextual information and open-world knowledge, SMARTFEAT generates a diverse set of relevant features for downstream prediction tasks. Additionally, our approach enables feature-level interactions with FMs and derives values of new features through the generated transformation functions, empowering SMARTFEAT to handle large datasets.

2 BACKGROUND AND RELATED WORK

Foundation models for data wrangling. Pre-trained language models, also known as foundation models, such as BERT [6], RoBERTa [18], GPT-3 [3], and ChatGPT, are neural networks trained on large corpora of text data encompassing various tasks. FMs learn the semantics of natural language by predicting the probability of masked words during pre-training and generate text based on the log probability during inference. Typically, FMs consist of billions of parameters and can be applied to a wide range of tasks through fine-tuning or few-shot prompting. Researchers have recently explored the potential of applying FMs in data management. Narayan et al. [19] propose cast data tasks including entity matching (EM), error detection (ED), and data imputation (DI) as prompting tasks to explore FMs' ability to perform classical data wrangling. DITTO [17] formulates EM as sequence-pair classification, utilizing transformer-based foundation models for enhanced language understanding. RPT [24] investigates pre-trained transformers for data preparation, including EM, DI, and ED. Retclean [2] addresses limitations in handling model errors, unseen datasets, and users' private data for ED and DI tasks. Usually, in the context of these works, FMs are integrated into the data management processes by serializing data entries and predicting masked tokens.

Automated feature engineering. A data processing workflow for machine learning or data analytics typically includes critical steps: data acquisition, integration [21], cleaning [1], feature engineering [27], and machine learning training. Our focus lies specifically on the feature engineering step [1]. In this context, automated feature engineering (AFE) tools play a vital role in assisting non-experts in constructing high-quality features from raw input data. AFE often

---

[1]We concentrate on the single table scenario, as table joins are typically part of the integration step.

employs data transformations to generate new features, thereby enhancing the performance of machine learning predictions. Various approaches, such as DSM [10] and OneBM [15] integrate multiple relations by enumerating potential transformations, while ExploreKit [11] generates new features using pre-defined operators. However, these methods, often referred to as *expansion-selection* methods, tend to create non-meaningful features and require extensive feature selection efforts as the number of generated features is unbounded [13]. Other approaches, like Cognito [13] based on tree-like exploration and AutoFEAT [9] based on iterative search, address some limitations but have low search efficiency. Learning-based methods, such as TransGraph [12] based on Q-learning and LFE [20] based on MLPs, have also been explored. FeSTE [7] proposes using external sources like Wikipedia to enhance classification accuracy but relies on entity matching and may not be applicable to all datasets. The most relevant work to SMARTFEAT is CAAFE [8], which also leverages FMs to create and validate new feature selections. However, our approach differs from CAAFE in that we define an operator-based search space for generating new features, as opposed to employing an interactive Chain-of-Thoughts [25] methodology.

3 PROPOSED METHOD

Given a dataset $\mathcal{D}$ consisting of instances $\mathcal{I}$ with original feature set $\mathcal{A} = \{A_1, A_2, \cdots, A_n\}$ and a classification attribute $\mathcal{Y}$, such that $\mathcal{D}_x = \mathcal{I} \times \mathcal{A}$ and $\mathcal{D}_y = \mathcal{Y}$. Our primary objective is to identify a set of suitable operators $Op = \{Op_1, \cdots, Op_m\}$ and their corresponding transformation functions $\mathcal{F} = \{f_1, \cdots, f_m\}$. By applying each transformation function to $\mathcal{D}$, we can obtain a set of candidate new features $\{A_1^{cand}, \cdots, A_m^{cand}\}$ for all $f_i \in \mathcal{F}$.

3.1 Overview

The feature generation process of SMARTFEAT is similar to other AFE tools [10, 11, 13] that utilize a search process to iteratively create new features and incrementally enhance the existing feature set. These newly generated features are subsequently taken into consideration for the ongoing generation of additional features.

Figure 1 illustrates, in each iteration, the process by which SMART-FEAT searches for and generates each new feature, employing two core components: the *operator selector* and the *function generator*, both supported by foundation models (FMs).

The input to SMARTFEAT comprises three elements: *(1) dataset feature description, (2) prediction class, and (3) downstream classification model*. The *dataset feature description* typically contains a belief description of the feature content, data type, and potentially the data domain of categorical features. This information can often be found in the data card of each open-source dataset, such as those available on platforms like Kaggle. We assume that, in most cases, the feature names are descriptive and this information is accessible or can be user-generated. We will also explore SMARTFEAT's performance with minimal input, consisting only of the feature names in Section 4.

The *prediction class* indicates the target variable that the downstream model aims to predict. Lastly, we specify the *classification model* used downstream. While our discussion mainly focuses on the downstream task of binary classification, our model can adapt
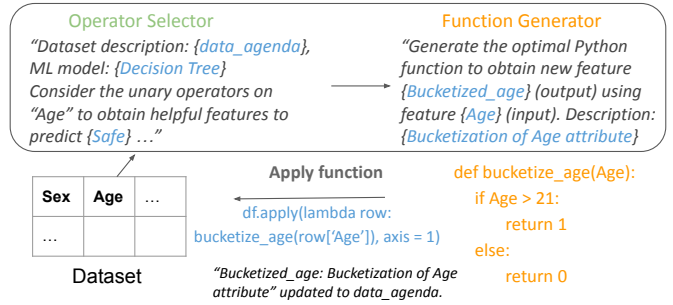

Figure 2: Illustrative example: constructing *Bucketized Age*.

to constructing features for other downstream applications with minor adjustments to the prompts.

The *operator selector* (①) chooses suitable operators and generates the descriptions for new features. It encodes a set of prompt templates corresponding to the operators considered for generating new features. Given the input, the operator selector uses the prompt for the current operator to interact with the FM model. It selects the appropriate operators and provides the (i) *name of the new feature*, the (ii) *relevant columns* to compute the new feature, and the (iii) *new feature description*. These three outputs serve as input for the function generator to generate the transformation function.

The *function generator* (②) seeks to obtain the optimal transformation function or provide necessary information, such as data sources, by interacting with an FM model. SMARTFEAT utilizes the state-of-the-art FM interaction toolkit, LangChain[2], to parse the output and then automatically applies the transformation function to the dataset. Once the new feature is successfully generated, both the feature name and its description (generated by the operator selector) are included in the *dataset feature description* and used to generate additional features in the next iteration.

Consider the example in Figure 2. The current operator being explored by the operator selector is a unary operator. Interacting with its FM model, the operator selector generates the output, including a feature name (Bucketized_age), a feature description, and the relevant column(s). The function generator then uses this output to obtain an executable function, applies it to the original dataset, and updates the data agenda. We next delve into the details of the two components in Section 3.2 and Section 3.3, respectively.

3.2 Operator-guided feature generation

The goal of this component is to efficiently generate candidate features without exhaustively enumerating all potential combinations. In this subsection, we first discuss a set of operators used to guide the generation of new features. Then, we explain how we prompt the FM and generate the candidate feature set.

Operator types. We consider four types of operators for generating candidate features: unary, binary, high-order, and extractor. For each operator, we use a prompt template to interact with the FM.

Unary operators encompass normalization, bucketization, and a set of unary operations such as getting dummies and date splitting.

Table 2: Example of prompt templates and outputs for operator selector.

Strategy	Operator	Prompt template	FM output
Proposal	Unary	"⋯ *Consider the unary operators on the attribute {org_attr} that can generate helpful features to predict {y_attr}.* *List all possible appropriate operators, and your confidence levels (certain/high/medium/low)* ⋯ "	Op_1 ((certain/high/...): desc. 1 Op_2 ((certain/high/...): desc. 2 ⋯
Sampling	High-order	"⋯ *Generate a groupby feature for predicting {y_attr} by applying 'df.groupby(groupby_col)[agg_col].transform(function)'.* *Specify the groupby_col, agg_col, and the aggregation function.*"	{groupby_col: [cols] , agg_col: col, function: mean/max/ ⋯}

The operator selector does not determine the specific transformation function to be selected, for example, whether to use min-max scaling or standardization for normalization, or what bucket boundaries to set for the bucketization. The focus is solely on assessing the types of unary operations that are beneficial, leaving the function selection to the second phase.

Binary operators include four basic arithmetic operations: $+, -, \times, \div$. We leave the more complex combinations of two original features to extractor operators.

High-order operators, where we consider the *GroupbyThenAgg* operation. We consider original features that are capable of categorizing data into distinct subsets as valid candidates for *Groupby* columns and features containing numerical information that can be aggregated as valid aggregate columns. The aggregate functions include mean, max, average, and others, and we allow the FM to choose the optimal function.

Extractors extract information that cannot be obtained from the previous operators. They handle more complex transformations and feature extractions, such as computing an index as a weighted combination of several features. Additionally, they can leverage library functions and assist in extracting information from external sources, such as obtaining the population density for each city in the motivating example.

Prompting FM for operator selection. As discussed in Section 3.1, the operator selector contains a set of prompt templates for each type of operator. Based on the characteristics of different operators, we employ two prompting strategies in our search for new features: the *proposal strategy* and the *sampling strategy* [26], which are encoded into the prompt templates of the operators.

In the proposal strategy, the FM is prompted to propose all potential candidates for Op. The candidates are drawn from FM's output by $[Op^{(1)}, \cdots, Op^{(i)}] \sim p^{\text{propose}}(Op^{(1, \cdots, i)}|z_{\text{descr}}, z_{\text{y}}, z_{\text{model}})$. From these proposals $Op^{(1, \cdots, i)}$, SMARTFEAT selects the most probable options, generating descriptive details, relevant columns, and a feature name. This strategy is more effective when dealing with relatively smaller search spaces because the time for searching is limited, and it also avoids duplication. For instance, when exploring the unary operators, we can apply the proposal prompting strategy to each original feature to propose potential unary operators that can be applied to the feature.

Table 2 presents the prompt template and the FM output for the unary operator. Based on the FM output, SMARTFEAT then selects the operators with *certain* or *high* confidence to generate new features. SMARTFEAT parses the output to obtain the new

feature name as "OpName_OrgAttr", the feature description as the operator description, and the relevant columns as the [OrgAttr].

In the sampling strategy, the FM is prompted to provide one candidate operator at a time, i.i.d. sampled from a Chain of Thought, i.e., $Op^{(i)} \sim p^{\text{sample}}(Op|(z_{\text{descr}}, z_{\text{y}}, z_{\text{model}})$. The sampling method works better when the generation space is rich. For instance, for high-order operators, the selection of *Groupby* columns can grow exponentially with the number of categorical features. In such cases, the sampling strategy leads to a more efficient generation and a more diverse set of candidates.

In SMARTFEAT, users can set a sampling budget for feature generation and a threshold for generation errors. The sampling process continues until the budget for sampling or the threshold for generation errors (invalid/repeated features) is reached.

Table 2 presents the prompt template and the FM output for the high-order operator. The FM returns an operator with the selected *Groupby* columns, the aggregate column, and the aggregate function. Subsequently, SMARTFEAT parses the output and returns the transformation function as the feature description and the feature name as "GroupBy_Gcol_func_Acol". The *Groupby* columns and the aggregate column are included as relevant columns.

Generating the candidate feature set. We discuss how operator-guided feature generation works, which aims to maximize the coverage and efficiency of the generation process. We begin by exploring unary operators for each original feature using the proposal strategy. Based on the original and unary features, we apply binary and high-order operators using the sampling strategy. Lastly, we consider extractors that can operate on multiple inputs using the sampling strategy, further enriching the current feature set.

We note that prediction performance improvement can also benefit from removing features. In SMARTFEAT, we employ a heuristic for dropping features: if an original feature undergoes a unary transformation and is not used by any other operators, we consider the original feature less important and, therefore, remove it from the feature set. The exploration of utilizing FMs for feature removal is left as future work.

3.3 Transformation function generation

After identifying candidate operators, the next step involves generating the transformation functions that compute the values of the new features. For each candidate, the function generator initiates an interaction with the FM to decide whether a transformation function can be derived, leading to three possible scenarios.

Table 3: Dataset statistics.

	# of cat. attr	# of num. attr	# of rows	field
Diabetes	0	9	769	Health
Heart	7	7	3,657	Health
Bank	8	10	41,189	Finance
Adult	8	6	30,163	Society
Housing	1	8	20,641	Society
Lawschool	5	7	4,591	Education
West Nile Virus	3	8	10,507	Disease
Tennis	0	12	944	Sports

Firstly, if a transformation function can be derived, SMARTFEAT generates it using the relevant columns as input and the new feature as the output. In most cases, interaction with FM is needed to obtain the most appropriate and efficient transformation function. For instance, employing FMs can assist in selecting suitable buckets for bucketization and importing necessary library functions as required. In the case of the high-order operator, the function generator can construct the transformation function directly from the output of the operator selector without the need to interact with the FM.

In certain situations, explicit functions may not be obtainable, requiring row-level text completion to calculate the new feature value, such as extracting the approximate population density for each city. To address this, we serialize the data entries and append the new feature name and a masked token as $A_1 : v_1, \cdots, A_k : v_k, A_{\text{new}} :?$, and then query the FM to generate the new feature values. However, for large datasets, the cost of obtaining these values through API calls could be prohibitive. Hence, we provide users with several examples and allow them to decide if the new features are valuable enough considering the associated cost.

Lastly, in situations where the function generator cannot produce a suitable transformation function or text completion is not applicable, the function generator suggests potential data sources to assist data users in constructing the new feature.

Evaluating generated features. We implemented a basic verification mechanism to ensure the quality of the features derived from FM-generated code. After obtaining the feature values, we perform feature selection to remove features that are highly null, single-valued, or dummy variables derived from high-cardinality original features. This feature selection process enhances the reliability and effectiveness of the generated features.

4 EVALUATION

We explore the performance of various downstream classification models when using features generated by SMARTFEAT with datasets from diverse fields, including health, finance, society, education, disease, and sports. We compare these against the state-of-the-art AFE methods.

4.1 Experimental setup

Datasets. We conducted the evaluation on eight supervised binary classification datasets, publicly available on Kaggle[3]. The summary of dataset statistics is shown in Table 3. We assessed the effectiveness of the FM in effectively handling column contexts across various domains. We randomly partitioned each dataset into 75% for training and 25% for testing and used 10-fold cross-validation. Prior to conducting the feature engineering process, we

executed standard data cleaning procedures on the datasets, including the removal of missing values (dropna) and the factorization of categorical features.

Metrics. To assess the effectiveness of new features, we employed five downstream machine learning classification algorithms from sklearn[4], which include Linear Regression (LR), GaussianNB (NB), Random Forest (RF), and Extra Tree (ET). Additionally, we incorporated a deep neural network (DNN) to demonstrate the potential improvement brought about by the constructed features, even though DNNs inherently possess the capability to learn deep features [16]. For all models, we utilized default parameter settings. The neural network architecture comprised two hidden layers, each consisting of 100 units and employing the ReLU activation function. In our evaluation, we considered the Area Under the ROC Curve (AUC) as the primary performance metric.

Baselines. We compared SMARTFEAT with the SoTA AFE tools as discussed in Section 2. However, only DSM [10], AutoFEAT [9], and CAAFE [8] offer publicly accessible implementations. Consequently, our experiments for comparison are based on these works. For DSM, we utilized its community-supported tool called **Featuretools** [5], which exhaustively generates new features using predefined operators and incorporates feature selection to eliminate highly correlated, highly null, and single-value features. AutoFEAT (referred to as **AutoFEAT** [6]) adopts a different approach by constructing a large set of non-linear features and subsequently performing a search algorithm to select an effective subset. **CAAFE**[7] is an AFE tool that utilizes FMs to generate Python code for data transformations. It includes a validation step: newly generated transformations are retained only if they demonstrate performance improvement on the validation set. For the implementation of Featuretools, we specifically utilized the primitives "add_numeric," "multiply_numeric," and "agg_primitive," while retaining default settings for other parameters. For AutoFeat, we used all default parameters. For CAAFE, we used its implementation with GPT-4 and 10 feature generation iterations as in [8].

In SMARTFEAT, we leveraged OpenAI's GPT-4 as the FM in the operator selector. For the function generator, we used the default OpenAI model in LangChain (GPT-3.5-turbo) due to its comparable performance and better efficiency. We employed the zero-shot prompting method in all FMs. The sampling budget for operators using the sampling prompting strategy was set to 10.

All experiments were conducted on a macOS system with a 1.4 GHz Quad-Core Intel Core i5 processor and 16GB of memory. We established a time limit of one hour for all experiments.

4.2 Evaluation results

Classification result. Table 4 and 5 present the results of the evaluation in terms of AUC values. The AUC improvement in percentage compared with the initial AUC is shown in parentheses. We highlighted the best-performing approaches in bold and underlined the baselines that do not support all ML models.

---

[3]https://www.kaggle.com/competitions/

[4]https://scikit-learn.org/stable/supervised_learning.html

[5]https://featuretools.alteryx.com/en/stable/

[6]https://github.com/cod3licious/autofeat/tree/master

[7]https://github.com/automl/CAAFE

Table 4: Comparison of the average AUC values (↑) of different ML models between SMARTFEAT and baseline methods.

Methods	Diabetes	Heart	Bank	Adult	Housing	Lawschool	West Nile Virus	Tennis
Initial AUC	82.20	67.38	91.46	76.81	86.72	**84.00**	78.96	77.93
SMARTFEAT	**86.76 (+4.3%)**	**72.15 (+7.0%)**	91.47 (≈)	**87.00 (+13.3%)**	**92.19 (+6.3%)**	83.68 (-0.4%)	**82.12 (+4.0%)**	87.39 (+9.5%)
CAAFE	-	69.67 (+3.4%)	91.73 (+0.3%)	83.10 (+8.2%)	92.15 (+6.3%)	83.86 (-0.2%)	80.11 (+1.8%)	**88.50 (+13.6%)**
Featuretools	82.24 (≈)	66.78 (-0.9%)	91.04 (-0.5%)	73.85 (-3.9%)	79.47 (-8.1%)	83.82 (-0.2%)	73.12 (-7.4%)	81.29 (+4.3%)
AutoFeat	75.24 (-8.4%)	64.92 (-3.7%)	-	-	77.63 (-10.5%)	-	70.90 (-10.2%)	71.73 (-8.0%)

Table 5: Comparison of the median AUC values (↑) of different ML models between SMARTFEAT and baseline methods.

Methods	Diabetes	Heart	Bank	Adult	Housing	Lawschool	West Nile Virus	Tennis
Initial AUC	83.18	69.19	92.77	80.63	91.28	83.73	77.66	80.41
SMARTFEAT	**87.78 (+5.5%)**	**71.70 +(3.6%)**	92.86 (≈)	**86.97(+7.9%)**	90.97 (-0.3%)	83.32 (-0.5%)	**82.06 (+5.7%)**	88.06 (+9.5%)
CAAFE	-	70.87 (+2.4%)	**93.06 +(0.3%)**	**87.00 (+7.9%)**	**92.84 +(1.7%)**	83.77 (≈)	80.90 (+4.2%)	**89.51 (+10.9%)**
Featuretools	82.78 (-0.5%)	69.37(-0.3%)	91.06 (-1.8%)	68.91 (-14.5%)	73.39(-19.0%)	83.74(≈)	75.71 (-2.5%)	83.03 (+3.3%)
AutoFeat	84.20 (+0.1%)	70.42 (+1.7%)	-	-	75.65(-17.1%)	-	76.53 (-1.4%)	67.83 (-15.6%)

We reported the average and median AUC values across all classification models. The results indicate that the FM-assisted methods, SMARTFEAT, and CAAFE consistently achieve better performance compared with other baselines. SMARTFEAT achieves a performance improvement of the average AUC for up to 13.3%, outperforming the other baselines in 5 out of 8 datasets (Table 4). However, we observed that the performance improvement on the *Bank* dataset and the *Lawschool* dataset is not evident, as also observed in the other baselines. This is because, in these two datasets, the original features are well-constructed, making feature engineering less impactful for performance enhancement.

CAAFE improves the performance of the initial AUC on all datasets. This is because it uses the validation set to evaluate the effect of each newly generated transformation and only retains the ones that improve the model performance, effectively preserving the helpful transformations. However, this step would be impractical when working with large datasets. CAAFE experienced a timeout on the DNN model on three large datasets, *Bank*, *Adult*, and *Bank*. CAAFE outperforms SMARTFEAT mainly on datasets that consist of more numerical features, such as the *Tennis* dataset. This is because, without the operator selector, we observed that the proposed transformations mainly perform combinations of numerical attributes. In addition, CAAFE samples the feature values, which also helps enhance the effectiveness of the transformations. However, for datasets where diverse types of new features would be beneficial, such as the *West Nile Virus* dataset, SMARTFEAT can generate more helpful features. CAAFE failed on the *Diabetes* dataset in that it suggested divide-by-zero transformations without handling the NAN values and caused the ML models to fail.

On the other hand, the transformations performed by Featuretools and AutoFeat are agnostic to the dataset context and the prediction task. In some cases, this results in the new features being less suitable for improving prediction performance, sometimes leading to a decrease in the AUC values for these baselines.

Efficiency. Feature generation is a task performed in concordance with human designers. As such, we need systems to take no more than a few minutes. On all datasets, both SMARTFEAT and Featuretools finished in well under 10 minutes. This was not the case for Autofeat, as it did not finish within our 60-minute time-out for

the *Bank* and *Adult* datasets. As discussed earlier, CAAFE also experienced a time-out on large datasets and complex models due to the validation step. It also incurs a relatively higher time compared to SMARTFEAT and Featuretools in general.

Feature importance. We now delve into the features generated by different approaches to assess their usefulness. We focus on the *Tennis* dataset where most of the approaches demonstrate effectiveness. We assessed the feature importance of all features, including both the new features and the original feature set. We evaluated three feature selection metrics provided by sklearn: the Information Gain (also known as mutual information), Recursive Feature Elimination, and Feature Importance metric based on tree-based selection (indicator for Gini index). We excluded statistics like χ^2-test or F-value as they are only applicable to either categorical or numerical attributes. Specifically, we examined the percentage of new features in the top-10 important features identified by each metric. For example, if SMARTFEAT has IG@10 = 80%, it means that 8 out of the top 10 ranked features under the information gain metric are new features generated by SMARTFEAT. A higher percentage indicates the generation of more useful features.

As shown in Table 6, CAAFE generates a small number of new features, mainly due to the removal of features in the validation step. However, all new features generated by CAAFE are considered important under all metrics. In comparison with the other two baselines, SMARTFEAT generates a smaller number of new features. This is attributed to the operator selector, which intelligently selects a subset of operators based on column context, avoiding an exhaustive enumeration of all possible features. SMARTFEAT demonstrates effectiveness in obtaining helpful features. Featuretools also successfully obtains important features but with a relatively large number of generated features. AutoFeat initially acquires a vast set of new features, but the feature selection steps only retain 5 features, which are shown less helpful under these metrics.

Ablation study. We conducted an ablation study to show how different operators in SMARTFEAT contribute to improving prediction performance. Using the *Tennis* dataset as an example, we assessed the AUC values obtained by adding the new features generated by each type of operator.

As shown in Table 7, the binary and extractor operators bring performance improvement in most cases, particularly for NB, RF,

Table 6: Percentage of top-10 important features (↑) generated by different methods under varying feature selection metrics, *Tennis*.

	SMARTFEAT	CAAFE	Featuretools	AutoFeat
# generated features	25	5	89 (sel-35)	1978 (sel-5)
IG@10	90%	50% (all)	90%	10%
RFE@10	80%	50% (all)	90%	30%
FI@10	80%	50% (all)	90%	30%

Table 7: Ablation study on operators in SMARTFEAT across different downstream ML models, *Tennis*.

	Initial	+Unary	+Binary	+High-order	+Extractor	all
LR	88.17	88.27	88.51	88.22	88.53	88.06
NB	66.85	65.16	79.68	66.49	90.00	84.05
RF	80.41	81.17	87.38	80.15	89.88	89.56
ET	79.14	75.14	88.02	77.56	90.04	88.86
DNN	84.50	87.31	87.57	86.08	86.92	86.46
Avg	79.81	79.41	86.31	79.70	89.07	87.39

and ET. In this dataset, the features introduced by the extractor are mainly index-like attributes computed from the combination of a set of attributes. LR's performance remains almost unaffected, indicating that feature engineering does not provide substantial benefits in this case. For DNN, we observed that almost all types of features contribute to enhancing the prediction results. This is because *Tennis* is a relatively small dataset, where simple models with well-constructed features usually perform better.

However, it's important to note that this trend may not be consistent across other datasets. For example, in the *West Nile Virus* dataset, the most beneficial features are those generated by the high-order operators.

Impact of Feature Descriptions. We also investigate the significance of having informative feature descriptions as input. To assess this, we conducted a comparative experiment on the *Tennis* dataset using only feature names without descriptions in SMARTFEAT. The feature names in the Tennis dataset are less descriptive, featuring abbreviations like `FSW.1` to represent "First Serve Percentage for player 1." In this experiment, the AUC score dropped to 77.86 (-1.4%) for the average and 79.39 (+2.2%) for the median. The experimental results emphasize the importance of including meaningful feature descriptions, particularly when clear and informative feature names are not available.

5 FINAL REMARKS

Foundation models hold the promise of leveraging contextual information and open-world knowledge for many data management and analysis tasks, such as feature generation. However, it is infeasible to feed these models with simple linearizations of relational tables for large databases, for reasons of efficiency and cost, even if the FMs were able to accept such large inputs. In this paper, we proposed operator-guided feature generation, coupled with feature-level interactions with FMs, to achieve both high efficiency and comprehensive coverage.

Moreover, FMs are susceptible to unpredicted errors, arising from limited access to data context or the inherent generative nature of FMs. To mitigate these errors, we employ feature selection to reduce the risk of generating low-quality features. While our experiments demonstrate the advantages of feature generation, further

improvements in error correction and detection are areas for future research. Integrating Foundation Models into data systems also poses challenges, especially when FMs rely on natural language input. Our paper primarily focuses on feature generation tasks, exploring one potential solution by discovering potential operators and their corresponding transformations. However, for more complex data-wrangling tasks that involve exploring data content and enabling feature-level interactions, discussions on obtaining informative data descriptions become essential.

ACKNOWLEDGEMENTS

REFERENCES

[1] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.

[2] M. S. Ahmad, Z. A. Naeem, M. Eltabakh, M. Ouzzani, and N. Tang. Retclean: Retrieval-based data cleaning using foundation models and data lakes. *arXiv:2303.16909*, 2023.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.

[4] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv:2204.02311*, 2022.

[5] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, 28(3):653–664, 2016.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[7] A. Harari and G. Katz. Few-shot tabular data enrichment using fine-tuned transformer architectures. In *ACL*, pages 1577–1591, 2022.

[8] N. Hollmann, S. Müller, and F. Hutter. Llms for semi-automated data science: Introducing caafe for context-aware automated feature engineering, 2023.

[9] F. Horn, R. Pack, and M. Rieger. The autofeat python library for automated feature engineering and selection. In *PKDD*, pages 111–120. Springer, 2020.

[10] J. M. Kanter and K. Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *DSAA*, pages 1–10. IEEE, 2015.

[11] G. Katz, E. C. R. Shin, and D. Song. Explorekit: Automatic feature generation and selection. In *ICDM*, pages 979–984. IEEE, 2016.

[12] U. Khurana, H. Samulowitz, and D. Turaga. Feature engineering for predictive modeling using reinforcement learning. In *AAAI*, volume 32, 2018.

[13] U. Khurana, D. Turaga, H. Samulowitz, and S. Parthasrathy. Cognito: Automated feature engineering for supervised learning. In *ICDMW*, pages 1304–1307, 2016.

[14] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17, 2015.

[15] H. T. Lam, J.-M. Thiebaut, M. Sinn, B. Chen, T. Mai, and O. Alkan. One button machine for automating feature engineering in relational databases. *arXiv:1706.00327*, 2017.

[16] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[17] Y. Li, J. Li, Y. Suhara, A. Doan, and W.-C. Tan. Deep entity matching with pre-trained language models. *arXiv:2004.00584*, 2020.

[18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv:1907.11692*, 2019.

[19] A. Narayan, I. Chami, L. Orr, and C. Ré. Can foundation models wrangle your data? *PVLDB*, 16(4):738–746, 2022.

[20] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. S. Turaga. Learning feature engineering for classification. In *IJCAI*, volume 17, pages 2529–2535, 2017.

[21] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas. Blocking and filtering techniques for entity resolution: A survey. *CSUR*, 53(2):1–42, 2020.

[22] L. E. Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[23] R. Richardson, J. M. Schultz, and K. Crawford. Dirty data, bad predictions: How civil rights violations impact police data, predictive policing systems, and justice. *NYUL Rev. Online*, 94:15, 2019.

[24] N. Tang, J. Fan, F. Li, J. Tu, X. Du, G. Li, S. Madden, and M. Ouzzani. Rpt: relational pre-trained transformer is almost all you need towards democratizing data preparation. *arXiv:2012.02469*, 2020.

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[26] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv:2305.10601*, 2023.

[27] M.-A. Zöller and M. F. Huber. Benchmark and survey of automated machine learning frameworks. *JAIR*, 70:409–472, 2021.