FeatAug: Automatic Feature Augmentation From One-to-Many Relationship Tables [Scalable Data Science]

Danrui Qi, Weiling Zheng, Jiannan Wang
Simon Fraser University
{dqi, weiling_zheng, jnwang}@sfu.ca

ABSTRACT

Feature augmentation from one-to-many relationship tables is a critical but challenging problem in ML model development. To augment good features, data scientists need to come up with SQL queries manually, which is time-consuming. *Featuretools* [22] is a widely used tool by the data science community to automatically augment the training data by extracting new features from relevant tables. It represents each feature as a group-by aggregation SQL query on relevant tables and can automatically generate these SQL queries. However, it does not include predicates in these queries, which significantly limits its application in many real-world scenarios. To overcome this limitation, we propose FeatAug, a new feature augmentation framework that automatically extracts predicate-aware SQL queries from one-to-many relationship tables. This extension is not trivial because considering predicates will exponentially increase the number of candidate queries. As a result, the original *Featuretools* framework, which materializes all candidate queries, will not work and needs to be redesigned. We formally define the problem and model it as a hyperparameter optimization problem. We discuss how the *Bayesian Optimization* can be applied here and propose a novel warm-up strategy to optimize it. To make our algorithm more practical, we also study how to identify promising attribute combinations for predicates. We show that how the beam search idea can partially solve the problem and propose several techniques to further optimize it. Our experiments on four real-world datasets demonstrate that FeatAug extracts more effective features compared to *Featuretools* and other baselines. The code is open-sourced at https://github.com/sfu-db/FeatAug.

1 INTRODUCTION

Machine learning (ML) can be applied to tackle a variety of important business problems in the industry, such as customer churn prediction [36], next purchase prediction [27], and loan repayment prediction [30]. While promising, the success of an ML project highly depends on the availability of good features [12]. When training data does not contain sufficient signals for a learning algorithm to train an accurate model, there is a strong need to investigate how to augment new features.

Due to the handcrafted feature augmentation being time-consuming, many automatic feature augmentation methods [10, 11, 14, 17, 22, 23, 26, 28, 29, 32, 35, 38] have been proposed. Most of them focus on extracting augmented features from the training table itself. However, in practice, there is relevant information stored in other tables that can be used to augment the training table.

Example 1. *Consider a scenario for predicting a customer's next purchase. We want to use customer data from the past 12 months (August 1st, 2022 to July 31st, 2023) to predict whether a customer will purchase a Kindle in August 2023. We have a training table called User_Info and a relevant table called User_Logs (shown in Figure 1). The User_Info table contains only a limited number of potentially useful features (i.e. age and gender), so additional useful features (i.e. avgprice) should be extracted from the User_Logs table.*

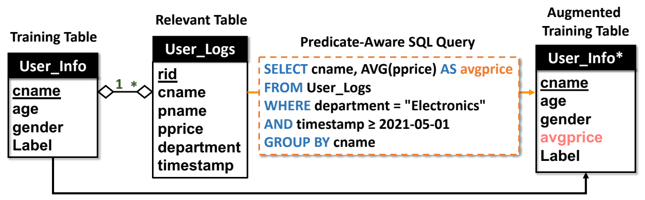

Figure 1: Feature augmentation with predicate-aware SQL queries.

However, we cannot simply add the columns of `User_Logs` to `User_Info` because the two tables have a one-to-many relationship. That is, each row in `User_Info` represents a customer, and a customer may have multiple purchases in `User_Logs`. To handle a one-to-many relationship table, data scientists typically write aggregation queries on relevant tables to extract features, which is time-consuming.

Example 2. *Continuing with Example 1, the relationship between User_Info and User_Logs is one-to-many with the foreign key cname. To predict whether a customer will purchase a Kindle in August 2023, a data scientist may believe that the amount a customer spent in the past is related to the likelihood that the customer will purchase a Kindle in the future. Consequently, she may write the following aggregation query to generate a feature:*

```
SELECT cname, AVG(pprice) AS feature FROM User_Logs
GROUP BY cname
```

To extract more useful features, the data scientist may need to write multiple queries by considering other aggregation functions such as COUNT and other columns in User_Logs like pname, which can be tedious and time-consuming.

Being able to *automatically* generate features from a one-to-many relationship table will facilitate various ML applications, such as customer churn prediction, next purchase prediction, and loan repayment prediction. The following example illustrates how Featuretools [22], widely used in the data science community, addresses this issue.

Example 3. *To relieve data scientists from the tedious task of writing SQL queries, Featuretools generates features automatically by constructing SQL queries in the following format:*

```
SELECT cname, agg(a) AS feature FROM User_Logs
GROUP BY cname
```

Here, agg *is an aggregation function such as SUM, AVG, and MAX, and* a *is an attribute in User_Logs used for aggregation. Each query result contains two columns. By joining the query result with the training table User_Info on cname, new features can be added to the training data.*

One significant limitation of Featuretools[22] is that it does not take predicates into account when generating queries. However, users' interests change rapidly, and purchases made on specific days like "Black Friday" or "Double 11" have little long-lasting impact on sales. Thus, extracting features by aggregating user behavior logs within a specific time slot, rather than using all logs, can be more helpful.

Example 4. *Continuing Example 2, Featuretools cannot automatically generate the following SQL query with predicates named predicate-aware SQL query:*

```
SELECT cname, AVG(pprice) AS avgprice FROM User_Logs
WHERE department = "Electronics"
AND timestamp ≥ 2023-07-01
GROUP BY cname
```

In fact, this is a useful feature. The more money a customer spends on "Electronics" products in a recent month, the more likely the customer will purchase a Kindle next month.

Our Methodology. Obviously, it is impossible to materialize all predicate-aware SQL queries because of two reasons: *(R1)* The number of SQL queries that can be constructed is huge even though the attribute combination in WHERE clause is fixed. *(R2)* There is not only one attribute combination that can form the WHERE clause. *Thus, can we directly find useful SQL queries (i.e. features) from the large search space?* Our key idea is to "learn" which areas in the search space are promising (or not promising), and then prune unpromising areas and generate SQL queries from promising areas. Based on this idea, we propose FeatAug, a predicate-aware SQL query generation framework. Given a training table and relevant table, FeatAug aims to automatically extract useful features from the relevant table by constructing predicate-aware SQL queries. FeatAug contains two components to filter out unpromising queries.

For R1, FeatAug needs to search for promising predicate-aware SQL queries by searching for the proper aggregation function, attributes for aggregation and values that can fill out the WHERE clause. Our key idea is to model the SQL query generation problem as a hyperparameter optimization problem. By modelling the correlation between the SQL queries (i.e. features) and their performance, we introduce an exploration-and-exploitation strategy to enhance the search process. Moreover, we also warm up the search process by transferring the knowledge of related tasks.

For R2, FeatAug need to search for promising attribute combinations in WHERE clause. Our key idea is to model the search space as a tree-like search space and greedily expand the tree by predicting the performance of each tree node. Note that each tree node represents an attribute combination. To reduce the long evaluation time of each tree node, we take the low-cost proxy to simulate the real evaluation score.

Contributions. We make the following contributions in this paper:

- We study a novel predicate-aware SQL query generation problem of automatic feature augmentation from one-to-many relationship tables motivated by real-world ML applications. We formally define the *Predicate-Aware SQL Query Generation* problem.
- We develop FeatAug, a predicate-aware SQL query generation framework to enable automatic feature augmentation from one-to-many relationship tables.
- We model the problem of searching for promising predicate-aware SQL queries as a hyperparameter optimization problem. We enhance the search process by introducing exploration-and-exploitation strategy and transferring the knowledge of related tasks.
- We model the search space of promising attribute combinations in WHERE clause as a tree-like space and greedily expand it by predicting the performance of each query template.
- The empirical results on four real-world datasets show the effectiveness of FeatAug on both traditional ML models and deep models. Compared to the popular Featuretools with the same number of generated features, FeatAug can get up to 9.32% AUC improvement on classification tasks and 0.1478 RMSE improvement on regression tasks.

2 RELATED WORK

Automatic Feature Augmentation. There are several existing efforts on automated feature augmentation [10, 11, 14, 17, 22, 23, 26, 28, 29, 32, 35, 38]. However, they complement our work and focus on different scenarios. Explorekit [23], FC-Tree [14], SAFE [35], LFE [32], Auto-Cross [29], Autofeat [17], OpenFE [38], and FETCH [26] work for the single table scenario. They automatically generate new features by applying different operators to existing features, which is orthogonal to the scenario FeatAug applies. ARDA [10] and AutoFeature [28] fit the scenarios with multiple relational tables by assuming that each table can be directly joined with the base table, i.e. has the one-to-one relationship with the base table. ARDA automatically joins the top relevant tables with the base table according to the relevant score it computed, While AutoFeature aims to filter out the effective feature set from the relevant tables that can be joined with the base table. Our work mainly considers the one-to-many relationship between the base table and the other relevant tables, which cannot be solved by directly joining. Featuretools [22] is a popular tool that automatically augments new features for the scenario with one-to-many relationship tables. It augments new features to the base table through SQL query generation, but their generated SQL queries do not include predicates in the WHERE clause, i.e. Featuretools is non-predicate-aware. In contrast, FeatAug is predicate-aware, i.e. FeatAug considers predicates in the WHERE clause when generating SQL queries. Tsfresh [11] automatically augments new features for single-table time-series data particularly, while FeatAug works for the multiple table scenario and border type of data compared to Tsfresh.

Hyperparameter Optimization. Hyperparameter Optimization has been extensively studied in the ML community [21]. Random Search [7] is one simple but effective method in this area. Bayesian Optimization (BO) is another popular methodology in this area. It uses a statistical model to estimate the expensive objective function. The common models are Gaussian Process (GP) [20, 34], Tree-structured Parzen Estimators (TPE) [6, 8], and Random Forest [18]. To speed up the search process of BO, Hyperband [25] and BOHB [13] are proposed with the parallel execution and the early-stopping ideas. Our work proposes a novel framework to bridge hyperparameter optimization and predicate-aware SQL query, i.e. feature generation, and may open up a new avenue for future research in automatic feature augmentation.

3 PROBLEM FORMULATION

In this section, we formulate two problems including *Predicate-Aware SQL Query Generation* and *Query Template Identification*.

3.1 Predicate-Aware SQL Query Generation

We formulate the Predicate-Aware SQL query generation problem for one-to-many relationship tables. Without losing generalization, we first define the problem under the scenario with one base table and one relevant table. In practice, there are more complex scenarios such as the *Deep-Layer Relationships*. By joining all the relevant tables into one, the *Deep-Layer Relationships* can also be represented by the aforementioned scenario.

Let D denote a *training table*, which has a unique identifier, a set of features, and a label. Let R denote a *relevant table* contains the foreign key referring to D's unique identifier. Firstly, we define *Query Template* used to generate effective predicate-aware SQL queries:

Definition 1 (Query Template). *Given a relevant table R with a set of attributes $Attr = \{A_1, A_2, \cdots, A_m\}$ where A_i denotes the i-th attribute of R, a query template w.r.t. R is a quadruple $T = (F, A, P, K)$, where F is a set of aggregation functions, $A \subseteq Attr$ is a set of attributes which can be aggregated, $P \subseteq Attr$ is a fixed attribute combination forming* WHERE *clause, and K is the foreign key attributes.*

Example 5. *Consider the relevant table in Example 1. Here is an example query template w.r.t. the table:*

$$T = \Big([SUM, AVG, MAX], [pprice], [department, timestamp], [cname]\Big)$$

where [SUM, AVG, MAX] is the aggregation function set F, A =[pprice] is the set of attributes which can be aggregated, [department, timestamp] is the fixed attribute combination that forms the WHERE *clause, and [cname] is the foreign key attribute between the base table D and the relevant table R.*

A query template belongs to a query template set which is defined in Definition 4. Different query templates in this query template set are with different attribute combinations in WHERE clause.

A query template represents a pool of candidate SQL queries. Definition 2 defines the query pool w.r.t. a given query template.

Definition 2 (Query Pool). *Given a relevant table R and a query template $T = (F, A, P, K)$, a query pool Q_T consists of a collection of predicate-aware SQL queries in the following form:*

```
SELECT k, agg(a) AS feature FROM R
WHERE predict(p_1) AND ... AND predict(p_w)
GROUP BY k
```

where $agg \in F$, $a \in A$, $p_i \in P$ for each $i \in [1, w]$ and $k \subseteq K$ is a subset of the foreign key attributes. If p_i is a categorical column, predicate(p_i) represents an equality predicate, i.e. $p_i = d$, where d is a value in the domain of p_i; if p_i is a numerical or datetime column, predicate(p_i) represents a range predicate, $d_{low} \leq p_i \leq d_{high}$, where d_{low} and d_{high} are two values in the domain of p_i ($d_{low} \leq d_{high}$). Note that the range-predicate definition includes one-sided range predicates.

Example 6. *Continue with Example 5, the query pool Q_T related to the query template T is composed of the predicate-aware SQL queries in the following form:*

```
SELECT cname, agg(a) AS feature FROM User_Logs
WHERE department = '?'
AND timestamp ≥ '?' AND timestamp ≤ '?'
GROUP BY cname
```

The SQL query in Example 4 is one query in the query pool Q_T.

For a predicate-aware SQL query $q \in Q_T$, let $q(R)$ denote the result table by executing q on R. Definition 3 defines the augmented training table, which adds the generated feature to the training table.

Definition 3 (Augmented Training Table). *Given a training table D and a query result table $q(R)$, the augmented training table D^q is defined as:*

```
SELECT D.*, q(R).feature
FROM D LEFT JOIN q(R) ON D.k = q(R).k
```

Example 7. *Continuing Example 4, after executing the query in Example 4, we get the query result table $q(User\_Logs) = (\underline{cname}, avgprice)$. We can get the augmented training table by joining User_Info with $q(User\_Info)$ and get $D^q = (\underline{cname}, age, gender, avgprice, label)$ with the following SQL query:*

```
SELECT User_Info.*, q(User_Logs).avgprice
FROM User_Info LEFT JOIN q(User_Logs)
ON User_Info.cname = q(User_Logs).cname
```

To evaluate the effectiveness of generated SQL query, i.e. feature, D^q can be split into a training set D^q_{train} and a validation set D^q_{valid}, where D^q_{train} is used to train an ML model and D^q_{valid} is used to evaluate model performance. The lower the model loss, the more effective the generated SQL query, i.e. feature.

The goal of *Predicate-Aware SQL Query Generation Problem* is to minimize the model loss by generating effective predicate-aware SQL queries, i.e. features. This implies an optimization problem. We formally define the *Predicate-Aware SQL Query Generation Problem* in Problem 1.

Problem 1 (Predicate-Aware SQL Query Generation). *Given a training table D, a relevant table R, a query template T and an ML model $\mathcal{A}$, the goal of predicate-aware query generation is to find the most effective query $q^* \in Q_T$ such that the model trained on $D^{q^*}_{train}$ and evaluated on $D^{q^*}_{valid}$ achieves the lowest loss, i.e.,*

$$q^* = \arg\min_{q \in Q_T} \mathcal{L}(\mathcal{A}(D^q_{train}), D^q_{valid}),$$

where $\mathcal{A}(D^q_{train})$ represents the model trained on the training set D^q_{train}. $\mathcal{L}(\cdot, \cdot)$ takes a model and the validation set D^q_{valid} as input, and returns the validation loss of the trained model.

3.2 Query Template Identification

In practice, users who are unfamiliar with the data often cannot provide explicit query templates for generating effective SQL queries. To deal with the more general scenario, we need to identify the query templates that are useful for generating effective SQL queries. We formally show the definition of *Query Template Set* and *the Effectiveness of Query Template*. Then we formally define the *Query Template Identification* problem in Problem 2.

Definition 4 (Query Template Set). *Given a relevant table R and an attribute set $Attr = \{A_1, A_2, \cdots, A_m\}$ where A_i denotes the i-th attribute of R, a query template set $\mathcal{S}$ w.r.t. R is a set including all possible query templates, i.e. $\mathcal{S} = \{(F, A, P, K) | \forall P \subseteq Attr\}$.*

Example 8. *Consider the query template T in Example 5. There are other query templates by differentiating P, i.e. [department, timestamp] in T. Here are other two example query templates:*

$$T_1 = \Big([SUM, AVG, MAX], [pprice], [pname, pprice], [cname]\Big)$$

$$T_2 = \Big([SUM, AVG, MAX], [pprice], [pname, department], [cname]\Big)$$

There are 2^6 different query templates, which can be constructed as the query template set $\mathcal{S}$.

Definition 5 (Effectiveness of Query Template). *Given a training table D, a relevant table R, a query template T and an ML model $\mathcal{A}$, the effectiveness of query template T is defined as,*

$$e_T = \mathcal{L}(\mathcal{A}(D^{q^*}_{train}), D^{q^*}_{valid})$$

where q^* is the most effective SQL query in Q_T.

Problem 2 (Query Template Identification). *Given a training table D, a relevant table R and a set of attributes $attr \subseteq Attr$, the query template set w.r.t. attr is $\mathcal{S}_{attr} = \{(F, A, P, K) | \forall P \subseteq attr\}$. The goal of query template identification is to recommend n query templates $T_1, T_2, \cdots, T_n \in \mathcal{S}_{attr}$, where $T_1, T_2, \cdots, T_n$ shows top-n effectiveness over all query templates in $\mathcal{S}_{attr}$.*

4 THE FEATAUG FRAMEWORK

In this section, we present the FeatAug framework. We first introduce the overview of FeatAug workflow. Without losing generalization, we exhibit more details of each part by taking the scenario with one base table and one relevant table as an example.

Workflow of FeatAug. The FeatAug workflow is shown in Figure 2. It includes two components named *SQL Query Generation* and *Query Template Identification.* The *SQL Query Generation* component aims to generate effective predicate-aware SQL queries. It takes one training table D, one relevant table R and one query template T as input. It outputs an effective predicate-aware SQL query q that can augment one feature into D by executing q on R. If users want to get β effective SQL queries in Q_T, they just need to call the *SQL Query Generation* component β times. The *Query Template Identification* component is optional in FeatAug. It deals with the situation that the users provide a set of attributes in R which may be useful for constructing promising query templates or even nothing. Given one training table D, one relevant table R and a set of attributes in R, the *Query Template Identification* component aims to figure out the most promising query templates, i.e. the most promising attribute combinations for constructing the WHERE clause. It outputs a set of promising query templates for the *SQL Query Generation* component to generate effective SQL queries.

SQL Query Generation. The *SQL Query Generation* component takes one training table D, one relevant table R and one query template T as input, and searches for the effective predicate-aware SQL query $q \in Q_T$. According to the definition of query pool (Definition 3), the *SQL Query Generation* problem can be modelled as the *Hyperparameter Optimization* problem. Then, the query pool Q_T can be searched iteratively for generating q. In this work, FeatAug utilizes *Bayesian Optimization* as the search strategy, which is commonly employed in the HPO area. At the start of the search process, FeatAug warms up the search process by transferring knowledge from relative tasks.

Query Template Identification. *Query Template Identification* component takes a training table D, a relevant table R, and a set of attributes in R as input. It constructs the query template set $\mathcal{S}$ according to the set of attributes and iteratively searches n promising query templates $T_1, T_2, \cdots, T_n$ which seems to include effective predicate-aware SQL queries in their query pools $Q_{T_1}, Q_{T_2}, \cdots, Q_{T_n}$.

At each iteration, FeatAug first draws the most promising sample of the query template T from the query template set $\mathcal{S}$. Then FeatAug evaluates predicate-aware SQL queries in the query pool Q_T. As Definition 4 shows, the effectiveness of T equals the evaluation result of the most effective SQL query $q* \in Q_T$. Note that the strategy of drawing the most promising T is determined by the search strategy FeatAug employed. In this work, FeatAug employs the beam-search idea for identifying the most promising query templates layer-by-layer. However, directly applying the beam-search idea is infeasible. We analyze the reason and make it practical in Section 6.

5 SQL QUERY GENERATION

In this section, we introduce the *SQL Query Generation* component in FeatAug. We first model the problem of searching for effective predicate-aware SQL queries in the query pool as the HPO problem. Then, we introduce a representative *Bayesian Optimization* algorithm for executing the search process in the query pool, i.e. *TPE* [6]. Finally, we introduce the strategy of warming up the search process.

5.1 SQL Query Generation as HPO

In Section 2, we define the *Query Pool* Q_T related to each *Query Template* T. Obviously, Q_T is our search space. We first map SQL queries in Q_T into a vector space $\mathcal{V}$, then we can model the *SQL Generation Problem* as *Hyperparameter Optimization Problem.*

Given a query template $T = (F, A, P, K)$ and a SQL query $q \in Q_T$, the corresponding *query vector* v_q consists of four parts: (1) a single element that represents the aggregation function selected from F. (2) a single element that represents the aggregation attribute selected from A. (3) a set of possible values for attributes in P forming WHERE clause. Suppose that P contains n categorical attributes and m numerical/datetime attributes. Note that we need 2 elements to represent a numerical/datetime attribute since a range predicate has a lower and upper bound. Then, the third part contains $(n + 2 * m)$ elements. If the query does not contain a predicate on some attribute, the corresponding element will be set to *None* meaning that the attribute is not selected. Otherwise, the actual value will be shown in the vector. (4) a set of possible values indicating the set of attributes k selected for GROUP BY clause. Note that k is the subset of the foreign key. If one attribute in the foreign key is selected, the corresponding element equals 1, otherwise 0. The fourth part contains $|K|$ elements. By indicating each element of v_q concretely, a query $q \in Q_T$ can be generated. All the *query vector* $v_q \in \mathcal{V}$, i.e. SQL query $q \in Q_T$, constructs the whole search space.

Example 9. *Considering Example 4, the query vector v corresponding to the SQL query in Example 4 is shown as follows:*

```
v = [1, 0, 4, '2023-05-01', None, 0]
```

In the query vector, the first element is set to 1, representing the AVG function whose index equals 1 is selected from the aggregation function set [SUM, AVG, MAX]. The second element is set to 0, representing the pprice attribute whose index equals 0 is selected from the aggregation attribute set [pprice]. The elements from the third place to the fifth place correspond to the attribute combination [department, timestamp] forming the WHERE *clause. The third element indicates*

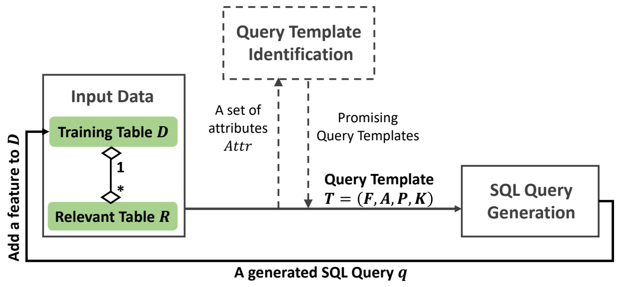
Query Template Identification
Input Data
Training Table D
1
*
Relevant Table R
A set of attributes $Attr$
Promising Query Templates
Query Template $T = (F, A, P, K)$
SQL Query Generation
Add a feature to D
A generated SQL Query q

Figure 2: Workflow of FeatAug.

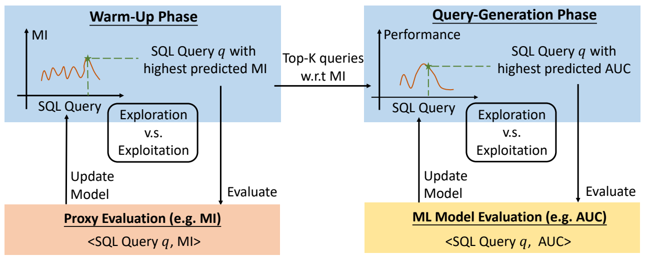
Warm-Up Phase
MI
SQL Query q with highest predicted MI
SQL Query
Exploration v.s. Exploitation
Update Model
Evaluate
Proxy Evaluation (e.g. MI)
<SQL Query q, MI>
Top-K queries w.r.t MI
Query-Generation Phase
Performance
SQL Query q with highest predicted AUC
SQL Query
Exploration v.s. Exploitation
Update Model
Evaluate
ML Model Evaluation (e.g. AUC)
<SQL Query q, AUC>

Figure 3: Workflow of SQL Query Generation Component. Mutual Information (MI) is taken as the low-cost proxy.

that the value of department is "4", i.e. the encoding of "Electronics". Since timestamp is a datetime attribute (occupying two elements in the vector), the fourth element represents the lower bound of timestamp and the fifth element represents the upper bound. The last element denotes the cname attribute whose index equals 0 is selected from the foreign key set [cname].

Mapping Q_T to $\mathcal{V}$ provides a natural analogy between the *SQL Generation Problem* and *Hyperparameter Optimization Problem.* In the HPO problem, a set of hyperparameters is also abstracted as a vector $[p_1, p_2, \cdots, p_i]$, and the value of $p_1, p_2, \cdots, p_i$ is picked from the domains of hyperparameters. The goal of the HPO problem is to search for the best vector that achieves the optimal metric. Meanwhile, the *Predicate-Aware SQL Query Generation Problem* also aims to find the most effective SQL queries $v_q \in \mathcal{V}$, i.e. features which lead to minimal validation loss of the ML model.

5.2 BO for SQL Query Generation

In the realm of *Bayesian Optimization (BO)*, the objective is to identify an optimal point $x*$ within a search space X, which maximizes an objective function f:

$$x^* = \arg\max_{x \in X} f(x)$$

Here, f serves as a black-box function lacking a straightforward closed-form solution. This framework is particularly popular in HPO, where x represents a set of hyperparameters and $f(x)$ quantifies the performance of a model governed by those HPs.

Sequential Model-Based Optimization (SMBO) is a well-established technique for tackling such problems. It treats f as an oracle, iteratively querying it to refine a Gaussian Process (GP) surrogate model. Due to the high computational cost of oracle evaluations, acquisition functions like Expected Improvement (EI) are employed to judiciously select subsequent query points. However, SMBO struggles with discrete points due to GP's inherent properties.

Tree-structured Parzen Estimator (TPE) addresses these limitations by utilizing Kernel Density Estimation (KDE) as its surrogate model. It partitions points into "good" and "bad" groups based on a quantile γ, and calculates EI as a function of the ratio $P_{good}(x)/P_{bad}(x)$. In multi-dimensional scenarios, separate KDE models are constructed for each dimension. There are certainly some other optimization approaches [6, 8, 18, 24, 34]. We adopt TPE in our study for three principal reasons: (1) it has an established reputation in the field of HPO [6, 8] (2) it is more efficient compared to SMBO [20] and SMAC [19] (3) TPE is good at optimizing both discrete and continuous hyperparameters.

Remark. The focus of this work is to demonstrate the applicability of HPO methodologies to the generation of predicate-aware SQL queries. Evaluations of alternative optimization strategies are reserved for future research.

5.3 Warm-up the Surrogate Model

The initialization phase of *TPE* is to draw a small sample of SQL queries (e.g., 20) from the query pool randomly, evaluate them, and fit initial KDEs according to evaluating results. Instead of randomly initializing the search process of TPE, we consider transferring the knowledge of the related low-cost tasks to strengthen the initialization, i.e. construct better *KDE*s at the start of *TPE* search. The knowledge-transferring idea can speed up the search process for predicate-aware SQL queries or even get better SQL queries. As shown in Figure 3, to incorporate the knowledge of related low-cost tasks into the search process, we propose to run TPE for two rounds. In the first round (Warm-Up Phase), we run TPE on the related low-cost task such as optimizing *MI* values, aiming to search for the SQL queries with high *MI* values. Then we select top-k SQL queries (e.g., 50) with the highest *MI* values, evaluate them, and use them to initialize the surrogate model (i.e. the KDEs) of the second round of TPE (Query-Generation Phase), which aims to search for the predicate-aware SQL query leads to the lowest validation loss of the ML model.

6 QUERY TEMPLATE IDENTIFICATION

In this section, we introduce the *Query Template Identification* component that identifies promising query templates when users only give a set of attributes which may include promising attribute combinations or even do not provide any possible set of attributes.

6.1 The Brute-Force Approach

The Brute-Force Query Template Identification. Assume $attr$ is a set of attributes from where we select a fixed attribute combination P to construct a query template as Definition 1 described, then the possible number of query templates equals the number of subsets of $attr$, which is $2^{|attr|}$. The brute-force approach of identifying n promising query templates is to calculate the effectiveness of all $2^{|attr|}$ query templates and select the query templates with n highest effectiveness. We denote $cost$ as the average cost of calculating the effectiveness of each query template $T \in \mathcal{S}_{attr}$. With the brute-force approach, the total cost of identifying promising query templates is $2^{|attr|} \cdot cost$. Because of the huge number of predicate-aware SQL queries in , obviously, it is impractical to search for global promising query templates with such an expensive cost.

6.2 The Beam Search Approach

To avoid the expensive calculation of the brute-force method and make the query template identification practical, inspired by *Beam Search* [31], we employ the *greedy* idea to explore the most promising part of a tree-structured search space in Figure 4. We first introduce how to map the search space of the query template into the tree-like search space, then we describe the search process and analyze the cost can be declined by utilizing the *Beam Search* idea.

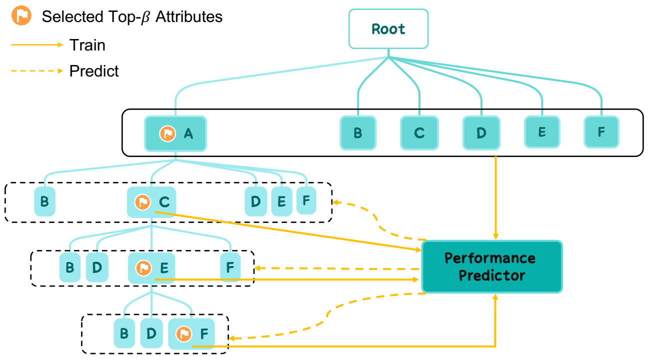

Figure 4: The illustration for the search space and the process of the Query Template Identification component. ($\beta = 1$)

The Tree-Like Search Space. The different subsets of *attr*, i.e. attribute combinations, construct a tree-structured search space shown in Figure 4. Each node depicts one possible attribute combination, i.e. one possible query template. Nodes in the first layer indicate the attribute combinations formed by only one attribute (e.g. $\{A\}, \{B\}, \cdots$). Nodes in the second layer indicate the attribute combinations formed by two attributes (e.g. $\{A, B\}, \{A, C\}, \cdots$). Obviously, the tree-structured search space in Figure 4 expands exponentially. If the relevant table is high-dimensional, it is crucial to find the search direction smarter.

The Identification Process with Beam Search. The basic idea of *Beam search* [31] is to only expand the top-β promising nodes in each layer. For example, Figure 4 shows one typical expansion with $\beta = 1$. Starting from the Root node, in the first layer, we get query templates formed by only one attribute (e.g. $\{A\}, \{B\}, \cdots$) and calculate their effectiveness. Then we pick up the top-1 node $\{A\}$ for the following expansion. In the second layer, we expand $\{A\}$ to $\{A, B\}, \{A, C\}, \cdots$, calculate their effectiveness and pick up the top-1 node to continue the expansion. The procedure is terminated the max *depth* for expansion is achieved. In Figure 4, we set the max $depth = 4$. Note that different query templates have different attribute combinations in the WHERE clauses. Thus, after the termination of the process in Figure 4, we get 6+5+4+3=18 query templates and their effectiveness. The n most promising query templates are identified by picking up query templates with n highest effectiveness. The identification process with *Beam Search* can decline the total cost of identifying promising query templates from $2^{|attr|} \cdot cost$ to $\sum_{i=1}^{depth} \beta \cdot (|attr| - i) \cdot cost$. For example, we calculate the effectiveness of 18 query templates in Figure 4 rather than $2^6 = 72$ query templates.

6.3 Optimizations for The Identification Process

However, the identification process described above is still not practical. That is because to get the top-β promising nodes in each layer, all nodes in this layer should be evaluated. The evaluation result of a node, i.e. the effectiveness of a query template T is determined by the most effective SQL query $q^* \in Q_T$ that minimizes the model validation loss. The optimal query q^* can be identified through exhaustive enumeration of Q_T by computing the actual validation loss associated with the ML model, which becomes computationally intensive when dealing with large training tables or complex models. To solve the above issue, we introduce a low-cost proxy to simulate the evaluation result of each node, i.e. the effectiveness of each query template. Instead of evaluating all nodes in each layer, we evaluate promising nodes by utilizing a performance predictor.

Optimization 1: Low-cost Proxy for Query Template Effectiveness. Instead of calculating the real validation loss of the ML model, considering a low-cost proxy to simulate the real validation loss is more practical. To address this, we use the low-cost proxy like *Mutual Information (MI)* to represent the real validation loss. MI is a well-established method in feature selection [9, 16, 33]. Given two random variables X and Y, MI measures the dependency between the two variables. The higher MI value indicates higher dependency. Note that the effectiveness of query template T equals the evaluation result of the most effective SQL query $q^* \in Q_T$, which leads to the lowest validation loss of the ML model compared to other SQL queries in Q_T. Thus, the *MI* between the feature generated by q^* and the labels can be the proxy of T's effectiveness.

Let us denote $cost_p$ as the average cost of calculating the proxy value of each query template $T \in \mathcal{S}_attr$, the low-cost proxy optimization can reduce the total cost of query template identification to $\sum_{i=1}^{depth} \beta \cdot (|attr| - i) \cdot cost_p$. Obviously, it is much cheaper because of the lower computation cost of the proxy compared to the evaluation with real ML model.

Optimization 2: Promising Query Templates Prediction. Even with the low-cost proxy, for selecting top-β nodes in each layer, we still need to evaluate all nodes (i.e. all query templates) in this layer. Thus, it is essential to cut off unpromising nodes prior to prevent redundant evaluations. A predictor can be trained to predict whether a node, i.e. query templates can produce effective predicate-aware SQL queries or not. For training this predictor, we first need to encode query templates, then collect training data and do inference layer-by-layer. We introduce the details of encoding query templates and predicting whether the query templates are promising in the following content.

(1) Encoding Query Templates. The difference among query templates is the different attribute combinations in the WHERE clause. Thus we take one-hot encoding to encode query templates. Take the attributes in Figure 4 as an example, there are six attributes {A, B, C, D, E, F} which can generate 2^6 possible query templates. If the WHERE clause of a query template T is composed by the attribute combination {A, C, E, F}, the encoding e_T=[1, 0, 1, 0, 1, 1].

(2) Predicting Promising Query Templates. As Figure 4 shows, we train the predictor by collecting the training data layer-by-layer. In the first layer, we get query templates formed by only one attribute and the proxy values of them. Thus we can get 6 training data in the first layer and train the predictor. Before evaluating query templates in the second layer, we first use the trained predictor to predict the proxy value of each node, i.e. query template. Then pick up the top-β query templates with top-β highest prediction scores and calculate their proxy values.

With the promising query template prediction, the total cost of query template identification can be finally reduced to $\sum_{i=1}^{depth} \beta \cdot cost_p$, which is much cheaper than the brute-force approach and the original beam search approach.

7 EXPERIMENTS

We conduct extensive experiments using real-world datasets to evaluate FeatAug. The experiments aim to answer main questions: (1) Can FeatAug benefit from the proposed optimizations? (2) Can FeatAug find more effective features compared to baselines on traditional ML models and deep models? (3) How does the performance of FeatAug change when the important settings change?

Table 1: Detailed information of datasets. "# of rows in R": the number of rows in relevant tables.

Dataset	# of Tables	# of rows in R	# of Train/Valid/Test
Tmall	3	6.5M	3.7w/1.2w/1.2w
Instacart	4	7.8M	3w/1w/1w
Student	2	1.6M	6k/2k/2k
Merchant	3	4.4M	3w/1w/1w

7.1 Experimental Settings

Datasets. We use the following 4 datasets including classification and regression tasks to conduct our experiments. The detailed information of the 4 datasets is shown in Table 1: *(1) Tmall* [1] is a repeat buyer prediction data aiming to predict whether a customer will be a repeat buyer of a specific merchant. This dataset includes three tables and we join the user profile table and the user behaviour table into one relevant table. *(2) Instacart* [2] aims to predict whether a customer will purchase a commodity which has "Banana" in its name. {{DQi: **process?**}} *(3) Student* [5] aims to use time series data generated by an online educational game to determine whether players will answer questions correctly. It contains two tables and we can directly consider the table containing the time series data as the relevant table. *(4) Merchant* [3] aims to recommend to users the merchant category they will buy in the next purchase. It contains three tables and we join the merchant information table and the historical transaction table into one relevant table.

Detailed Information of Query Templates. Table 2 shows the aggregation functions utilized by each dataset. It also shows the number of attributes for aggregation and the number of attributes in the relevant table that may be helpful for forming `WHERE` clause. The concrete names of attributes can be found in our technical report []. The group-by keys between the training and relevant table of each dataset are also shown in Table 2. With these information, query templates and the related query pools can be constructed for each dataset.

Approaches. We compare our FeatAug with three approaches. The first compared approach is *Featuretools* [22]. Note that *Featuretools* cannot construct predicate-aware SQL queries and it does not filter out any useless SQL queries (i.e. features) during the generation process. Thus, we combine the SQL query (i.e. feature) generation process of *Featuretools* with feature selectors and consider it as the second compared approach. In this paper, we choose *GBDT Selector* by considering its effectiveness [37]. The third compared approach is the random approach, which randomly picks up query templates and predicate-aware SQL queries, i.e. features.

- Featuretools materalizes all features with *Featuretools* without any feature selector.
- Featuretools + GBDT Selector first generates features with *Featuretools*, then uses these features to train a *Gradient Boosting Decision Tree (GBDT)* classifier and selects the features with top feature importances.
- Random first chooses query templates from the query template set randomly, then randomly searches predicate-aware SQL queries in each query pool of each query template.
- FeatAug first identifies promising query templates, then searches effective predicate-aware SQL queries, i.e. features in each query pool of each promising query template.

In our experiments, we utilize *Featuretools* and *Featuretools + GBDT Selector* to generate 40 features. We also use the random approach and FeatAug to generate 40 predicate-aware SQL queries, (i.e. features) by selecting 8 query templates and 5 predicate-aware SQL queries in each query pool related to each query template.

Table 2: Detailed information of query templates. "F": the aggregation functions. "# of A": the number of attributes for aggregation. "# of $attr$": the number of provided attributes that may be useful for forming `WHERE` clause. "K": the group-by keys between the training and relevant table. "# of T": the number of query templates.

Dataset	Tmall	Instacart	Student	Merchant
F	SUM, MIN, MAX, COUNT, AVG, COUNT_DISTINCT, VAR, VAR_SAMPLE, STD, STD_SAMPLE, ENTROPY, KURTOSIS, MODE, MAD, MEDIAN			
# of A	6	6	10	34
# of $attr$	5	8	10	15
K	user_id, merchant_id	user_id	session_id	merchant_id
# of T	2^5	2^8	2^{10}	2^{15}

Table 3: Ablation study of FeatAug. "NoQTI": without the Query Template Identification component. "NoWU": without the warm-up part in the SQL Query Generation component.

Dataset		Tmall	Instacart	Student	Merchant
Metric		AUC ↑	AUC ↑	AUC ↑	RMSE ↓
LR	**NoQTI**	0.5257	0.5000	0.5000	3.9855
	NoWU	0.5650	0.6354	**0.5935**	3.9549
	Full	**0.5749**	**0.6369**	**0.5935**	**3.9538**
XGB	**NoQTI**	0.5331	0.5000	0.5000	4.0176
	NoWU	0.5812	0.6794	**0.5782**	4.0084
	Full	**0.5898**	**0.6844**	**0.5782**	**4.0012**
RF	**NoQTI**	0.5325	0.5000	0.5000	**4.0063**
	NoWU	0.5526	0.6063	0.5582	4.0567
	Full	**0.5573**	**0.6248**	**0.5636**	4.0313
DeepFM	**NoQTI**	0.5284	0.5000	0.5000	3.9942
	NoWU	0.6186	0.7330	0.6303	3.9398
	Full	**0.6226**	**0.7364**	**0.6438**	**3.9277**

ML Models. We evaluate our proposed method using three traditional ML models including *Logistic Regression (LR), Random Forest (RF), XGBoost (XGB)* and one deep model *DeepFM* [15]. We choose the three traditional ML models based on the recent survey [4], which shows their effectiveness and popularity. We choose *DeepFM* because it is effective and widely used in the industry, particularly for recommendation systems and advertising. Choosing *DeepFM* as downstream ML models emphasizes the practical implications and potential benefits of FeatAug in real-world applications.

Metrics. For the classification datasets including *Tmall, Instacart* and *Student*, we evaluate the performance with *AUC*. For the regression dataset *Merchant*, we evaluate the performance using *RMSE*.

Implementation Details. For all the datasets, we set the ratio of train/valid/test as 0.6/0.2/0.2. Our experiments are conducted on one AWS EC2 r6idn.8xlarge instance (32 vCPUs and 256GB main memory) by default. All of the experiments are repeated five times and we report the average to avoid the influence of hardware, network and randomness.

7.2 Can FeatAug Benefit from The Proposed Optimizations?

In this section, we examine whether the proposed optimizations can benefit FeatAug.

Optimization 1: The Warm-up in SQL Generation. To study the benefit of the warm-up in the *SQL Generation* component, we drop the warm-up part in the *SQL Generation* component and Table 3 shows the performance gap w/o the warm-up. In our implementation of FeatAug, the process of the warm-up includes running TPE on the related low-cost task (i.e. optimizing MI value) for 200 iterations, selecting SQL queries with top-50 *MI* values and evaluating them to initialize the surrogate model. Then we run 40 iterations of

Table 4: Overall performance of FeatAug compared to baselines. "Featuretools": Featuretools without feature selector. "Featuretools+": Featuretools with GBDT Selector.

Dataset		Tmall	Instacart	Student	Merchant
Metric		AUC	AUC	AUC	RMSE
LR	**Featuretools**	0.5610	0.5679	0.5269	3.9677
	Featuretools+	0.5620	0.6100	0.5003	3.9678
	Random	0.5630	0.6021	0.5620	3.9804
	FeatAug	**0.5749**	**0.6369**	**0.5935**	**3.9538**
XGB	**Featuretools**	0.5568	0.6349	0.5730	4.0752
	Featuretools+	0.5494	0.6488	0.5736	4.0637
	Random	0.5848	0.5830	0.5575	4.0161
	FeatAug	**0.5898**	**0.6844**	**0.5782**	**4.0012**
RF	**Featuretools**	0.5000	0.5601	0.5205	**4.0160**
	Featuretools+	0.5000	0.5723	0.5262	4.0274
	Random	0.5572	0.6057	0.5432	4.0246
	FeatAug	**0.5573**	**0.6248**	**0.5636**	4.0313
DeepFM	**Featuretools**	0.5818	0.7001	0.5685	3.9840
	Featuretools+	0.6074	0.7085	0.5967	3.9327
	Random	0.5976	0.6449	0.6115	3.9817
	FeatAug	**0.6226**	**0.7364**	**0.6438**	**3.9277**

TPE with the warm-started surrogate model. For fair comparison, we do not simply drop the whole process above and run 40 iterations of TPE because the evaluating time of the top-50 SQL queries in the warm-up part cannot be neglected. Instead, we drop the warm-up part by only running 50+40=90 iterations of TPE. In most scenarios, the warm-up part can lead to better performance, which shows the effectiveness of transferring knowledge from the relevant tasks. An interesting observation is that the effectiveness of the warm-up part under different scenarios is different, which is highly related to datasets and the downstream ML models. Despite *MI*, there are also other static characteristics like *Spearman Correlation* are also alternatives of the proxy. We explore the effectiveness of different proxies in Section 7.4. Figuring out the proxy that contributes most to the warm-up prior is an interesting direction for future research.

Optimization 2: Query Template Identification To study the benefit of the query template identification component, we drop the query template identification part in FeatAug and Table 3 also shows the performance gap w/o the query template identification. Recall that for `Tmall`, `Instacart` and `Merchant` dateset, we provide a set of attributes which may be useful for forming the `WHERE` clause, while for `Student` dataset, we directly consider all the attributes in the relevant table. For dropping the *Query Template Identification* component, we take the same attribute sets for each dataset to construct a query template and search for effective SQL query in the related query pool. In 15 out of 16 scenarios, adding the query template identification component can improve the performance significantly, which shows the effectiveness of picking up the promising query templates. Without the query template identification, there is only one possible query template constructed by the set of attributes provided by users, which leads to an unpromising query pool.

7.3 Can FeatAug Find More Effective Features?

In this section, we compare FeatAug with baselines to figure out whether FeatAug can find more effective features.

We evaluate the performance of generated features on four datasets in Table 1 with four ML models, i.e. 16 scenarios in total. The result of the effectiveness experiment is shown in Table 4. We can see that FeatAug outperforms all baselines in 15 scenarios. The max AUC improvement on classification tasks achieves 10.14%, while the max RMSE improvement on regression tasks achieves 0.0740. The significant improvement on both traditional ML models and deep models shows the generalization of FeatAug. Note that *Featuretools* generates features by constructing all possible SQL queries without considering `WHERE` clause, while FeatAug generates features by considering both SQL queries with and without `WHERE` clause. If the predicate-aware SQL queries are useless, the performance of FeatAug would be approximate to or worse than *Featuretools*. However, FeatAug shows AUC improvement in most scenarios compared to *Featuretools*. Thus, extending the *Featuretools'* space for generating predicate-aware SQL queries is effective.

Table 5: Performance of FeatAug by varying the low-cost proxy. "SC": Spearman Correlation. "MI": Mutual Information. "LR": the Logistic Regression model.

Dataset		Tmall	Instacart	Student	Merchant
Metric		AUC ↑	AUC ↑	AUC ↑	RMSE ↓
LR	**SC**	0.5629	0.6168	**0.5935**	3.9623
	MI	**0.5749**	0.6369	**0.5935**	**3.9538**
	LR	0.5537	**0.6476**	0.5856	3.9756
XGB	**SC**	0.5854	0.6632	0.5772	**3.9943**
	MI	**0.5898**	**0.6844**	**0.5782**	4.0012
	LR	0.5888	0.6057	0.5517	4.0053
RF	**SC**	0.5549	0.6086	0.5687	**4.0230**
	MI	**0.5573**	0.6248	0.5636	4.0313
	LR	0.5396	**0.6670**	**0.5750**	4.0666
DeepFM	**SC**	0.6177	0.7266	0.6396	3.9464
	MI	**0.6226**	**0.7364**	**0.6438**	**3.9277**
	LR	0.6135	0.7269	0.6382	3.9799

7.4 In-Depth Analysis of FeatAug

In this section, we perform the in-depth analysis of the performance impact to FeatAug under different settings.

Varying Number of Query Template. Recall that when utilizing FeatAug to generate effective SQL queries, we pick up 8 promising query templates and search for 5 effective SQL queries in each query pool. It is interesting whether more query templates cause better performance. Figure 5 shows the trend of performance by varying the number of query templates. We show all the trends on our 4 datasets and 4 downstream ML models. We have three interesting observations.

Firstly, in most cases (9 out of 16 scenarios), the increase in the number of query templates brings performance improvement to downstream ML models. It shows that considering multiple query templates is more helpful than only a single query template, which matches what data scientists really do in practice. Secondly, there is no fixed number of query templates that fit all scenarios. For *Tmall* dataset with DeepFM model, FeatAug converges when the number of query templates is 7, while FeatAug converges at 3 when the dataset is *Instacart* with DeepFM model. Thirdly, the deep model i.e. *DeepFM* can get benefits easily from the increased number of query templates, while traditional ML models including *LR, XGB* and *RF* keep stable in most cases even the number of query templates increases. That is because the deep models can perform feature interaction automatically, and the increase of the number of query templates provides more opportunity for deep models to synthesize generated features into more informative features.

Varying The Low-cost Proxy. We explore the sensitivity of FeatAug by varying the low-cost proxy and recommend the low-cost proxy in practical scenarios. We consider three low-cost proxies including *Spearman Correlation (SC)*, *Mutual Information (MI)* and *Logistic Regression (LR)*. Given two random variables X and Y, *SC* calculates the monotonic dependency between X and Y. While *MI* is defined

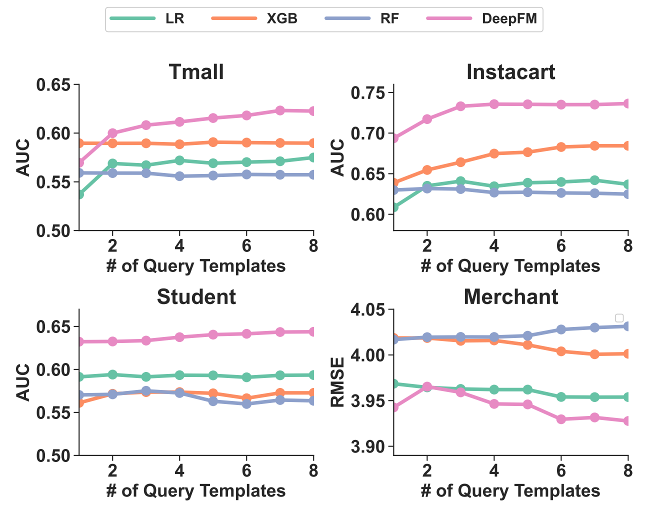

Figure 5: The trend of performance by varying the number of query templates.

as $I(X;Y) = H(X) - H(X|Y)$, where $H(X)$ is the entropy of X and $H(X|Y)$ is the conditional entropy for X given Y. *LR* takes the performance of *LR* model as the proxy of other ML models.

As we can see in Table 5, *SC* performs best in 2 out of 16 scenarios, *LR* performs best in 3 out of 16 scenarios, and *MI* is the most effective proxy in most cases, i.e. 11 out of 16 scenarios. The result suggests the entropies calculated in *MI* can simulate the performance of ML models well in both classification and regression tasks. Surprisingly, *SC* is competitive to *MI* in 10 out of 16 scenarios. Note that the AUC score represents the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. Thus the monotonic dependency that *SC* measures is helpful for getting higher AUC score. The RMSE score can also benefit from *SC*. However, *LR* proxy is not competitive with *LR* and *SC* in most cases, i.e. 10 out of 16 scenarios, suggesting that the performance of *LR* cannot represent the performance of other ML models well.

8 CONCLUSION

In this paper, we tackled the scalability challenge in feature discovery. We formalized it as an automated SQL query generation problem and proposed FeatAug, a novel framework that can automatically and efficiently search for the best query in a large query pool. The key insight is to model it as a hyperparameter tuning problem. We discussed how to extend TPE (a widely used hyperparameter tuning algorithm) to our problem. We observed that directly applying TPE in FeatAug cannot solve our problem well. We analyzed the reasons and proposed three optimizations to enhance FeatAug. We conducted extensive experiments and case studies using two real-world ML datasets to evaluate FeatAug and compare it with TPE and Random Search. The results show that FeatAug was able to discover explainable and effective features from a large search space (with millions of possible features), making scalable feature discovery from impossible to possible.

REFERENCES

[1] Ijcai-15 repeat buyers prediction dataset. https://tianchi.aliyun.com/dataset/dataDetail?dataId=42, 2015.
[2] Instacart market basket analysis. https://www.kaggle.com/c/instacart-market-basket-analysis, 2017.
[3] Elo merchant category recommendation. https://www.kaggle.com/competitions/elo-merchant-category-recommendation, 2018.
[4] State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021, 2021.
[5] Student dataset. https://www.kaggle.com/competitions/predict-student-performance-from-game-play, 2023.
[6] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.
[7] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.
[8] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.
[9] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján. Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. *The journal of machine learning research*, 13:27–66, 2012.
[10] N. Chepurko, R. Marcus, E. Zgraggen, R. C. Fernandez, T. Kraska, and D. R. Karger. ARDA: automatic relational data augmentation for machine learning. *Proc. VLDB Endow.*, 13(9):1373–1387, 2020.
[11] M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr. Time series feature extraction on basis of scalable hypothesis tests (tsfresh - A python package). *Neurocomputing*, 307:72–77, 2018.
[12] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.
[13] S. Falkner, A. Klein, and F. Hutter. BOHB: robust and efficient hyperparameter optimization at scale. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1436–1445. PMLR, 2018.
[14] W. Fan, E. Zhong, J. Peng, O. Verscheure, K. Zhang, J. Ren, R. Yan, and Q. Yang. Generalized and heuristic-free feature construction for improved accuracy. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pages 629–640. SIAM, 2010.
[15] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: A factorization-machine based neural network for CTR prediction. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.
[16] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.
[17] F. Horn, R. Pack, and M. Rieger. The autofeat python library for automated feature engineering and selection. In P. Cellier and K. Driessens, editors, *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, volume 1167 of *Communications in Computer and Information Science*, pages 111–120. Springer, 2019.
[18] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION'05 Proceedings of the 5th international conference on Learning and Intelligent Optimization*, pages 507–523, 2011.
[19] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.
[20] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *International Conference on Learning and Intelligent Optimization*, pages 281–298. Springer, 2010.
[21] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges.* Springer Nature, 2019.
[22] J. M. Kanter and K. Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Campus des Cordeliers, Paris, France, October 19-21, 2015*, pages 1–10. IEEE, 2015.
[23] G. Katz, E. C. R. Shin, and D. Song. Explorekit: Automatic feature generation and selection. In F. Bonchi, J. Domingo-Ferrer, R. Baeza-Yates, Z. Zhou, and X. Wu, editors, *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, pages 979–984. IEEE Computer Society, 2016.
[24] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.
[25] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.
[26] L. Li, H. Wang, L. Zha, Q. Huang, S. Wu, G. Chen, and J. Zhao. Learning a data-driven policy network for pre-training automated feature engineering. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net, 2023.
[27] G. Liu, T. T. Nguyen, G. Zhao, W. Zha, J. Yang, J. Cao, M. Wu, P. Zhao, and W. Chen. Repeat buyer prediction for e-commerce. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 155–164, 2016.

[28] J. Liu, C. Chai, Y. Luo, Y. Lou, J. Feng, and N. Tang. Feature augmentation with reinforcement learning. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 3360–3372. IEEE, 2022.

[29] Y. Luo, M. Wang, H. Zhou, Q. Yao, W. Tu, Y. Chen, W. Dai, and Q. Yang. Autocross: Automatic feature crossing for tabular data in real-world applications. In A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 1936–1945. ACM, 2019.

[30] R. Malhotra and D. K. Malhotra. Evaluating consumer loans using neural networks. *Omega*, 31(2):83–96, 2003.

[31] M. F. Medress, F. S. Cooper, J. W. Forgie, C. Green, D. H. Klatt, M. H. O'Malley, E. P. Neuburg, A. Newell, D. Reddy, B. Ritea, et al. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3):307–316, 1977.

[32] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. S. Turaga. Learning feature engineering for classification. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 2529–2535. ijcai.org, 2017.

[33] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8):1226–1238, 2005.

[34] M. Schonlau, W. J. Welch, and D. R. Jones. Global versus local search in constrained optimization of computer models. *Lecture Notes-Monograph Series*, page 11–25, 1998.

[35] Q. Shi, Y. Zhang, L. Li, X. Yang, M. Li, and J. Zhou. SAFE: scalable automatic feature engineering framework for industrial tasks. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1645–1656. IEEE, 2020.

[36] T. Vafeiadis, K. I. Diamantaras, G. Sarigiannidis, and K. C. Chatzisavvas. A comparison of machine learning techniques for customer churn prediction. *Simulation Modelling Practice and Theory*, 55:1–9, 2015.

[37] Z. E. Xu, G. Huang, K. Q. Weinberger, and A. X. Zheng. Gradient boosted feature selection. In S. A. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 522–531. ACM, 2014.

[38] T. Zhang, Z. Zhang, Z. Fan, H. Luo, F. Liu, W. Cao, and J. Li. Openfe: Automated feature generation beyond expert-level performance. *CoRR*, abs/2211.12507, 2022.

[39] M.-A. Zöller and M. F. Huber. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research*, 70:409–472, 4 2019.