FeatAug: Automatic Feature Augmentation From One-to-Many Relationship Tables

Danrui Qi
Simon Fraser University
Vancouver, Canada
dqi@sfu.ca

Weiling Zheng
Simon Fraser University
Vancouver, Canada
weiling_zheng@sfu.ca

Jiannan Wang
Simon Fraser University
Vancouver, Canada
jnwang@sfu.ca

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00146

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* [1] 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.

***Index Terms*—automatic feature augmentation, automatic feature engineering, data preparation, one-to-many relational tables**

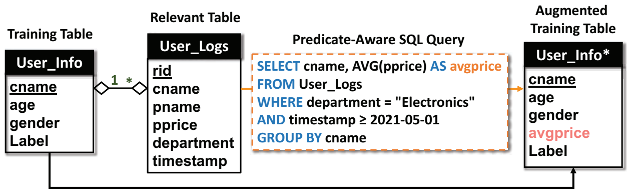
I. INTRODUCTION

Machine learning (ML) can be applied to tackle a variety of important business problems in the industry, such as customer churn prediction [2], next purchase prediction [3], and loan repayment prediction [4]. While promising, the success of an ML project highly depends on the availability of good features [5]. 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.

A. Motivation

Due to the handcrafted feature augmentation being time-consuming, many automatic feature augmentation methods including [1], [6]–[16] 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.



```
SELECT cname, AVG(pprice) AS avgprice
FROM User_Logs
WHERE department = "Electronics"
AND timestamp ≥ 2021-05-01
GROUP BY cname
```

Fig. 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. 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. 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 [1] 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 is 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.

B. Chanllenges and 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.

C. Our 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 attribute combination.
- The empirical results on four real-world datasets show the effectiveness of FEATAUG on both traditional and deep ML models. Compared to the popular *Featuretools* and other baselines with the same number of generated features, FEATAUG can get up to 10.74% AUC improvement on classification tasks and 0.0740 RMSE improvement on regression tasks.

II. Related Work

In this section, we present the related work of this paper from four perspectives including automatic feature augmentation, feature selection in automatic feature augmentation, data enrichment and hyperparameter optimization.

Automatic Feature Augmentation. There are several existing efforts on automated feature augmentation [1], [6]–[17]. However, they complement our work and focus on different scenarios. Explorekit [6], FC-Tree [11], SAFE [12], LFE [13], Auto-Cross [7], Autofeat [8], OpenFE [9], and FETCH [10] work for the single table scenario. They automatically generate new features by applying unray operators, binary operators and feature crossing operations to existing features, which is orthogonal to the scenario FEATAUG applies. ARDA [14] and AutoFeature [15] fit the scenarios with multiple relational tables by assuming that each table can be directly joined with the training table, i.e. the one-to-one relationship tables. 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 tables, which cannot be solved by directly joining. Featuretools [1] is a popular tool that automatically augments new features for one-to-many relationship tables. It augments new features to the training table through generating SQL queries by using aggregation functions such as SUM, COUNT and MIN without considering predicates. In contrast, FEATAUG is predicate-aware, i.e. FEATAUG considers predicates in the `WHERE` clause when generating SQL queries.

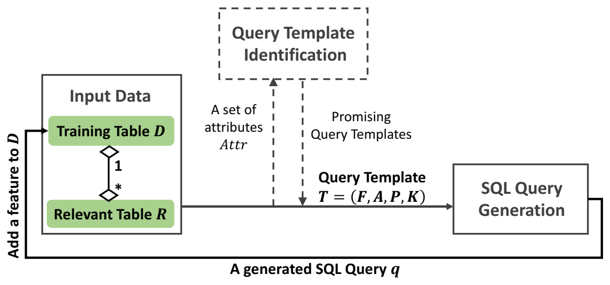

Query Template Identification
Input Data
Training Table D
Relevant Table R
1
*
A set of attributes $Attr$
Promising Query Templates
Query Template $T = (F, A, P, K)$
SQL Query Generation
Add a feature to D
A generated SQL Query q

Fig. 2: Workflow of FEATAUG·

Feature Selection in Automatic Feature Augmentation. Feature selection [18], [19] aims to only keep effective features and filter out features that have little or even negative impact on the performance of the downstream ML model. Several automatic feature augmentation methods follow the expand-and-reduce framework and utilize different feature selection strategies. FC-Tree [11] and SAFE [12] use the information gain such as mutual information to select useful features. AutoCross [7] and AutoFeat [8] use the improvement of a linear regression model to evaluate whether a feature is effective. LFE [13] and ExploreKit [6] use meta-features to train an ML model and predict whether a new coming feature is effective. AutoFeature [15] defines a reward function, i.e. the improvement of an XGBoost model to measure the impact of adding a feature on the performance of a downstream ML model. Different from the previous methods, FEATAUG filters out useless features by identifying promising query templates and promising search areas in query pools prior.

Data Enrichment. Data enrichment aims to augment a local table with new attributes extracted from external data, such as Web Tables [20]–[22] and Deep Web [23]–[25]. In Deep Web, the knowledge from deep web (i.e., a hidden database) are progressively crawled through a keyword-search API to enrich a local table. These data enrichment methods do not measure the practical impact of enriched data on the particular ML model i.e. they are not model-aware. While FEATAUG is a feature augmentation method that is model-aware, i.e. the enrichment goal of FEATAUG is to maximize the performance of the downstream ML model.

Hyperparameter Optimization. Hyperparameter Optimization has been extensively studied in the ML community [26]. Random Search [27] is one simple but effective method in this area. Bayesian Optimization (BO) is another popular methodology in this area using one surrogate model to establish the correlation between hyperparameters and the downstream ML model performance. Establishing these surrogate models is often expensive. The commonly used surrogate models include Gaussian Process (GP) [28], [29], Tree-structured Parzen Estimators (TPE) [30], [31], and Random Forest [32]. To speed up the search process of BO, Hyperband [33] and BOHB [34] are proposed with the parallel downstream ML model fitting 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.

III. PROBLEM FORMULATION

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

A. 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. It is easy to be extended to more complex scenarios. For the *Deep-Layer Relationships* [1], it can be represented by the aforementioned scenario by joining all the tables into one relevant table. For the scenario with multiple relevant tables, it can be represented by multiple scenarios with one base table and one relevant table.

Let D denote a *training table*, which has a primary key, a set of features, and a label. Let R denote a *relevant table* contains the foreign key referring to D's primary key. 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:

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

where [`SUM, AVG, MAX`] is the aggregation function set F, A =[`pprice`] is the set of attributes for aggregation, [`department, timestamp`] is the fixed attribute combination that forms the `WHERE` clause, and [`cname`] is the foreign key attribute between D and R.

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, $d_{low} \le p_i \le d_{high}$, where d_{low} and d_{high} are two values in the domain of p_i ($d_{low} \le d_{high}$). Note that the range-predicate definition includes one-sided range predicates.

Example 6: Continuing Example 5, the query pool Q_T related to the query template T is composed of predicate-aware SQL queries in the following form. And the SQL query in Example 4 is one query in Q_T.

```
SELECT cname, agg($a$) AS feature
FROM User_Logs
WHERE department = '?'
AND timestamp $\ge$ '?' AND timestamp $\le$ '?'
```

```
GROUP BY cname
```

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 that adds the generated feature in $q(R)$.

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`)= (<u>`cname`</u>, `avgprice`). We can get the augmented training table by joining `User_Info` with q(`User_Info`) and get D^q=(<u>`cname`</u>, `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*.

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.

B. 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. Thus, we formally define the *Query Template Identification* problem.

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:

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

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

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 $\mathcal{S}_{attr}$.

IV. The FeatAug Framework

In this section, we present the overview of FeatAug framework. Then we exhibit more details of each part in FeatAug. 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. Figure 2 illustrates the FeatAug framework. It takes a training table and a relevant table that has one-to-many relationship with the training table as input. In the *SQL Query Geneartion* component, FeatAug iteratively searches for the effective queries (i.e. features) in the query pool. Then in the *Query Template Identification Component*, FeatAug iteratively searches n promising query templates which seem to include effective predicate-aware SQL queries in their query pools.

1) 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 a training table D, a relevant table R and a query template T as input. It outputs an effective predicate-aware SQL query q that can augment one feature into D. If users want to get multiple effective SQL queries in Q_T, they just need to call the *SQL Query Generation* component multiple times. However, in practice, users who are unfamiliar with the input data often cannot specify the attribute combination P in T explicitly. Instead, they can only provide a set of attributes in R which may construct promising query templates or even nothing. Then the *Query Template Identification* component is optional in FeatAug. It deals with the situation that the users cannot provide explicit query templates. 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 n most promising query templates, i.e. n most promising attribute combinations for constructing the `WHERE` clause. It outputs a set of promising query templates as the input of the *SQL Query Generation* component to generate effective SQL queries.

2) The SQL Query Generation Component: The SQL Query Generation component takes a training table D, a relevant table R and a 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 *Predicate-Awarer SQL Query Generation* problem can be modelled as the *Hyperparameter Optimization* problem. Thus 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.

3) The Query Template Identification Component: The Query Template Identification Component takes a training table D, a relevant table R, and a set of attributes $attr$ in R as input. It constructs the query template set $\mathcal{S}_{attr}$ and iteratively searches n promising query templates which seem to include effective predicate-aware SQL queries in their query pools. At each iteration, FEATAUG first draws the most promising sample of the query templates from $\mathcal{S}_{attr}$. Then FEATAUG evaluates predicate-aware SQL queries in the related query pools. As Definition 4 shows, the effectiveness of query templates is determined by the evaluation result of the most effective SQL query in their query pool. Note that the strategy of drawing the most promising query templates 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 VI.

V. SQL QUERY GENERATION

In this section, we introduce the *SQL Query Generation* component in FEATAUG. We first model the *Predicate-Aware SQL Query Generation* problem as the HPO problem. Then, we introduce a representative *Bayesian Optimization* algorithm for executing the search process in the query pool, i.e. *TPE* [31]. Finally, we introduce the strategy of warming up the search process.

A. 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 attribute for aggregation 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, the third part contains $(n+2*m)$ elements because we need 2 elements to represent the range predicate of a numerical/datetime attribute. If the query does not contain a predicate on some attribute, the corresponding element will be set to *None*. 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 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.

B. 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:

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

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 hyperparameters.

BO framework [28] treats f as an oracle and iteratively queries 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, the GP surrogate model 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. The boundray between good points and bad points is denoted by γ which is some quantile of observed evaluations. γ typically equals to 10%-15%, i.e. 10%-15% points are good points. Then it calculates EI as a function of the ratio

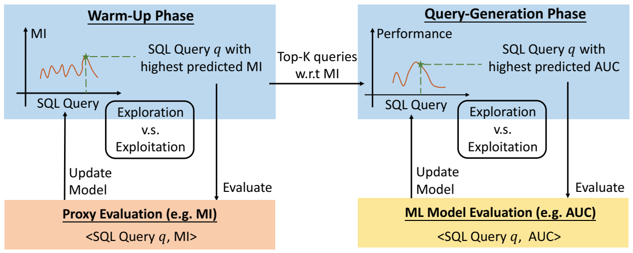

Warm-Up Phase
MI
SQL Query q with highest predicted MI
SQL Query
Exploration v.s. Exploitation
Update Model
Evaluate
Proxy Evaluation (e.g. MI)
<SQL Query q, MI>
Top-K queries w.r.t MI
Query-Generation Phase
Performance
SQL Query q with highest predicted AUC
SQL Query
Exploration v.s. Exploitation
Update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

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

$P_{good}(x)/P_{bad}(x)$. In multi-dimensional scenarios, a specific KDE model is constructed for each dimension. There are certainly some other optimization approaches [29]–[32], [35]. We adopt TPE in our study for three principal reasons: (1) it has an established reputation in the field of HPO [30], [31] (2) it is more efficient compared to other BO approaches such as SMBO [28] and SMAC [36] (3) TPE is good at optimizing both discrete and continuous hyperparameters.

Remark. The focus of this work is to demonstrate the applicability of HPO methods to the generation of predicate-aware SQL queries. We choose the popular TPE to achieve this goal. There exists other HPO methods such as SMAC [36] and BOHB [34]. It will be interesting to investigate which HPO method is better in the future study.

C. Warm-up the Surrogate Model

1) Potential Issues of TPE: At the initial stage of *TPE*, it randomly draws a sample of SQL queries from the search space to identify a promising area and exploit this area by selecting the queries around this area. Our problem has a large search space and an expensive evaluation, thus directly applying *TPE* requires a large number of iterations to identify promising areas, which could be very expensive. This issue will be further exaggerated when the size of training data is large.

2) Our Solution - Warm-up the Surrogate Model: 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.

VI. Query Template Identification

In Section V, we introduce how to generate effective queries when the query template is fixed. In a practical scenario, users often do not know the detailed correlation between the training table and the relevant table. Thus, the attributes provided by users for forming predicates may not be the promising attribute combination for generating effective SQL queries. To further promise the generalization of FeatAug, we try to identify the effective query template when users cannot provide explicit query templates. In this section, we introduce the *Query Template Identification* component that identifies promising query templates when users cannot provide explicit query templates.

A. The Brute-Force Approach

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 for 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. Note that the cost of calculating the effectiveness of each query template $T \in \mathcal{S}_{attr}$ is different. That is because the size of Q_T is different and the execution cost of each SQL query $q \in Q_T$ varies. Thus we denote $cost$ as the maximum cost of calculating the effectiveness of each query template $T \in \mathcal{S}_{attr}$. With the brute-force approach, the maximum cost of identifying promising query templates is $2^{|attr|} \cdot cost$. Because of the huge number of predicate-aware SQL queries in the query pool related to each query template, obviously, it is impractical to search for global promising query templates with such an expensive cost.

B. 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* [37], we map the search space of the query template into the tree-like search space, and employ the *greedy* idea to explore the most promising part of a tree-structured search space in Figure 4.

1) The Tree-Structured 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.

2) The Identification Process with Beam Search.: The basic idea of *Beam search* [37] 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 when 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

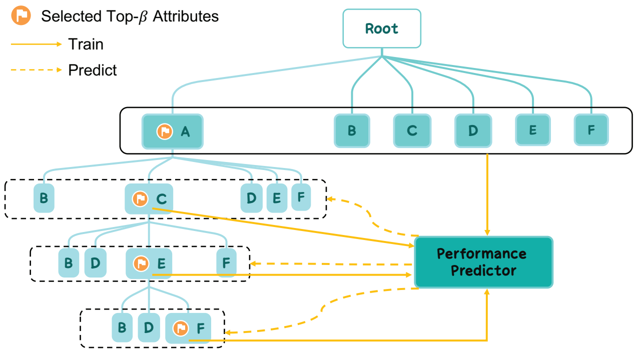


Fig. 4: The illustration for the search space and the process of the Query Template Identification component. ($\beta = 1$)

query templates are identified by picking up query templates with n highest effectiveness from all the 18 query templates. The identification process with *Beam Search* can decline the maximum cost of identifying promising query templates from $2^{|attr|} \cdot cost$ to $\left(|attr| + \sum_{i=2}^{depth} \beta \cdot (|attr| - i)\right) \cdot cost$. For example, we calculate the effectiveness of 18 query templates in Figure 4 rather than $2^6 = 72$ query templates.

C. 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.

1) 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 [19], [38], [39]. 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 maximum cost of calculating the low-cost proxy value for each query template $T \in \mathcal{S}_{attr}$. The low-cost proxy optimization can reduce the maximum cost of query template identification to $\left(|attr| + \sum_{i=2}^{depth} \beta \cdot (|attr| - i)\right) \cdot cost_p$. Obviously, it is much cheaper because $cost_p << cost$.

2) 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.

- **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].
- **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 maximum cost of query template identification can be finally reduced to $\left(|attr| + \sum_{i=2}^{depth} \beta\right) \cdot cost_p$, which is much cheaper than the brute-force approach and the original beam search approach.

TABLE I: Detailed information of datasets. "# of Tables": the number of tables included in each dataset. "# 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

VII. 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?

A. Experimental Settings

1) Datasets: We use the following 4 datasets including classification and regression tasks to conduct our experiments.

TABLE II: 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}

The detailed information of the 6 datasets is shown in Table I:

- *Tmall* [40] 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.
- *Instacart* [41] aims to predict whether a customer will purchase a commodity which has "Banana" in its name. It contains four tables. We join the historical order table, the product table and the department table into one relevant table.
- *Student* [42] 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.
- *Merchant* [43] 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.

2) Detailed Information of Query Templates: Table II shows the aggregation functions (F) utilized by each dataset. It also shows the number of attributes for aggregation (# of A) and the number of attributes in the relevant table that may be helpful for forming `WHERE` clause (# of $attr$). The concrete names of attributes can be found in our technical report [44]. The group-by keys (K) between the training and relevant table of each dataset are also shown in Table II. With these information, query templates and the related query pools can be constructed.

3) Baselines: We compare our FEATAUG with a variety of typical solutions. For all datasets, the first compared approach is *Featuretools* [1]. 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 also consider them as the compared approaches. In this paper, we choose seven feature selectors by considering the feature selection approaches in Section II. Another 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 + LR / GBDT Selector** first generates features with *Featuretools*, then uses these features to train a *Logistic Regression* or a *Gradient Boosting Decision Tree (GBDT)* classifier and selects the features with top feature importances.
- **Featuretools + MI / Chi2 / Gini Selector** first generates features with *Featuretools*, then uses *Mutual Information (MI)* or *Chi-square (Chi2)* or *Gini index (Gini)* to evaluate the correlation between the features and the labels. Finally, the features with top correlations are selected. Note that *Chi2* and *Gini* are only suitable for classification tasks, and *MI* is suitable for both classification and regression tasks.
- **Featuretools + Forward Selector** first generates features with *Featuretools*. Then, in each iteration, the *Forward Selector* adds the feature causing the highest improvement of the downstream ML model performance into the training table.
- **Featuretools + Backward Selector** first generates features with *Featuretools*. Then, in each iteration, the *Backward Selector* removes the feature degrading the downstream ML model performance most.
- **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.

In our experiments, we utilize *Featuretools* and *Featuretools + Selectors* 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.

4) 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* [45]. We choose the three traditional ML models based on the recent survey [46], which shows their effectiveness and popularity. *LR* and *RF* are the two most popular ML models. *XGB* is a tree-based model which takes the first popularity of complex ML models. 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.

5) Metrics: For the classification datasets including *Tmall, Instacart* and *Student*, we evaluate the performance with *AUC*, where the receiver operator characteristic (ROC) is a probability curve displaying the performance over a series of thresholds and AUC is the area under the ROC curve. For the regression dataset *Merchant*, we evaluate the performance using *RMSE*.

6) Implementation Details: For all the datasets, we set the ratio of train/valid/test as 0.6/0.2/0.2. We develop FEATAUG based on the TPE implementation in the Hyperopt library [47]. The traditional ML models we used in experiments are constructed using Scikit-Learn library [48]. The code is written in Python 3.8.10. 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.

TABLE III: Overall performance of FEATAUG compared to baselines on datasets with one-to-many relationship tables. "FT": Featuretools without any feature selector. "FT+": Featuretools with the selector showing the highest performance..

Dataset		Tmall	Instacart	Student	Merchant
Metric		AUC	AUC	AUC	RMSE
LR	**FT**	0.5610	0.5679	0.5269	3.9677
	FT+	0.5641	0.6100	0.5846	3.9670
	Random	0.5630	0.6021	0.5620	3.9804
	FeatAug	**0.5749**	**0.6369**	**0.5935**	**3.9538**
XGB	**FT**	0.5568	0.6349	0.5730	4.0752
	FT+	0.5587	0.6507	**0.5821**	4.0637
	Random	0.5848	0.5830	0.5575	4.0161
	FeatAug	**0.5898**	**0.6844**	0.5782	**4.0012**
RF	**FT**	0.5000	0.5601	0.5205	**4.0160**
	FT+	0.5028	0.5723	0.5369	4.0179
	Random	0.5572	0.6057	0.5432	4.0246
	FeatAug	**0.5573**	**0.6248**	**0.5636**	4.0313
DeepFM	**FT**	0.5818	0.7001	0.5685	3.9840
	FT+	0.6074	0.7130	0.5967	3.9327
	Random	0.5976	0.6449	0.6115	3.9817
	FeatAug	**0.6226**	**0.7364**	**0.6438**	**3.9277**

B. Can FeatAug Find More Effective Features for One-to-Many Relationship Tables?

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 with one-to-many relationship tables in Table I with four ML models, i.e. 16 scenarios in total. The effectiveness of FEATAUG is summarized in Table III, highlighting its superiority over baselines in 14 scenarios with a maximum AUC increase of 10.14% for classification tasks and a maximum RMSE reduction of 0.0740 for regression tasks. This demonstrates FEATAUG's broad applicability across both traditional ML and deep learning models. 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 performance improvement in most scenarios compared to *Featuretools*. Due to space constraints, Table III presents only the result of *Featuretools* with the selector showing the highest performance. For the full table, refer to [44].

It is notable that FEATAUG outperforms *Random* in classification tasks, achieving average AUC improvements of 1.07% on the *Tmall* dataset, 6.17% on the *Instacart* dataset, and 2.62% on the *Student* dataset across various ML models. For regression tasks, it recorded an average RMSE improvement of 0.1848 on the *Merchant* dataset across various ML models. These results demonstrate the effectiveness of Bayesian Optimization in identifying more efficient predicate-aware SQL queries compared to random search.

C. How does FeatAug perform for Single Table and One-to-One Relationship Tables?

In this section, we extend the evaluation of FEATAUG's effectiveness to datasets with single table and one-to-one relationship tables. Note that the datasets with single table can also be transferred to the scenario with one-to-one relationship tables by duplicating itself as the relevant table.

TABLE IV: Detailed information of *Covtype* and *Household* datasets.

Dataset	# of Tables	# of rows in R	# of Train/Valid/Test
Covtype	1	50K	3w/1w/1w
Household	1	9.5K	5.7k/1.9k/1.9k

TABLE V: Detailed information of query templates of *Covtype* and *Household* datasets.

Dataset	Covtype	Household
F	SUM, MIN, MAX, COUNT, AVG, COUNT DISTINCT, VAR, VAR SAMPLE, STD, STD SAMPLE, ENTROPY, KURTOSIS, MODE, MAD, MEDIAN	
# of A	54	123
# of $attr$	10	20
K	data_index	data_index
# of T	2^{10}	2^{20}

For the scenario with single table, we choose *Covtype* [49] dataset from UCI Machine Learning Repository [50], which is used in the single table feature augmentation works [9], [17]. *Covtype* [49] aims to predict forest cover in four Colorado wilderness areas. This dataset includes only one table and we take itself as the relevant table. For the scenario with one-to-one relationship tables, we choose *Household* [51] dataset which is been used by *Featuretools* to show their demo [52], [53]. *Household* [51] aims to classify the families' poverty level by considering the observable household attributes. It contains only one table. We keep 5 features in the training table and put other 137 features into the relevant table. The statistical information of these two datasets are shown in Table IV and detailed information of query templates of these two datasets are shown in Table V.

For datasets with single table and one-to-one relationship tables, two additional baselines dealing with these scenarios, i.e. *ARDA* [14] and *AutoFeature* [15] are also compared:

- **ARDA** heuristically uses a random injection-based feature augmentation to search good feature subsets from the relevant table.
- **AutoFeature** is a RL-based automatic feature augmentation method. In each iteration, AutoFeature utilizes *Multi-armed Bandit (MAB)* or *Deep Q Network (DQN)* to predict the next action, i.e. the next feature to augment.

We evaluate the performance of generated features on two datasets in Table IV with three ML models, i.e. 6 scenarios in total. That is because the two datasets are multi-class datasets and *DeepFM* only works for binary classification tasks. The result of the effectiveness experiment is shown in Table VI. We can see that FEATAUG outperforms all baselines in 4 scenarios. The *F1* score improvement shows that FEATAUG can also work correctly and well for the datasets with single table and the one-to-one relationship tables.

D. Can FeatAug Benefit from The Proposed Optimizations?

In this section, we examine whether the proposed optimizations can benefit FEATAUG. The two main optimizations we proposed in this paper are the warm-up part in the *SQL Generation* component and the *Query Template Identification* component. The results are shown in Table VII.

1) Can FEATAUG *Benefit from 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*

TABLE VI: Overall performance of FEATAUG compared to baselines on dataset with one-to-one relationship tables. "FT+": Featuretools with the selector showing the highest performance.

Dataset		Covtype	Household
Metric		F1 ↑	F1 ↑
LR	**FT**	0.1681	**0.2378**
	FT+	0.1248	0.2356
	AutoFeat-MAB	0.2688	0.1424
	AutoFeat-DQN	0.1930	0.2161
	ARDA	0.2275	0.2020
	Random	0.2492	0.2112
	FeatAug	**0.3084**	0.2159
XGB	**FT**	0.7582	0.2718
	FT+	0.5067	0.2782
	AutoFeat-MAB	0.7766	0.2927
	AutoFeat-DQN	0.7766	0.2453
	ARDA	0.6422	0.2375
	Random	**0.7800**	0.2666
	FeatAug	0.7769	**0.3024**
RF	**FT**	0.6289	0.2444
	FT+	0.5067	0.2782
	AutoFeat-MAB	0.7814	0.2278
	AutoFeat-DQN	0.6884	0.2371
	ARDA	0.6573	0.2639
	Random	0.7964	0.2616
	FeatAug	**0.8074**	**0.3003**

Generation component and Table VII 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 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.

2) Can FEATAUG *Benefit from Query Template Identification?:* To study the benefit of the query template identification component, we drop the query template identification part in FEATAUG and Table VII 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.

TABLE VII: Ablation study of FEATAUG. "NoQTI": FEATAUG without the Query Template Identification component. "NoWU": FEATAUG without the warm-up part in the SQL Query Generation component. "Full": FEATAUG with both the warm-up part in the in the SQL Query Generation component and the Query Template Identification 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**

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.

3) Can the Query Template Identification Component Benefit from the Two Optimizations?: To study whether the two optimizations in Section VI can indeed speed up the Query Template Identification Component, we drop each optimization and Figure 5 (a) shows the running time w/o the two optimizations. Without any optimizations, the initial Beam Search cannot complete query template identification in 6 hours on any dataset, which indicates that the initial Beam Search is very time-consuming. With only *Low-cost Proxy Optimization*, the Query Template Identification component can be completed in 2 hours on *Tmall, Student* and *Merchant* datasets, and in 5 hours on *Instacart* dataset, which already speeds up the initial Beam Search. With all optimizations, the Query Template Identification component can be speed up **1.4x - 2.8x** compared to the method with only *Low-cost Proxy Optimization*. We also explore whether adding the two optimizations will hurt the performance of FEATAUG severely. Figure 5 (b) - (e) shows the comparison results. For all the four datasets and all the four downstreaming ML models (in total 16 scenarios), adding the *Promising Query Template Prediction Optimization* hurts little performance of FEATAUG. The comparison results reveals that the *Promising Query Template Prediction Optimization* can cut off unpromising query templates prior precisely and fast, and keep promising query templates to produce effective predicate-awared SQL queries.

E. In-Depth Analysis of FeatAug

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

1) 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 6 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 improve-

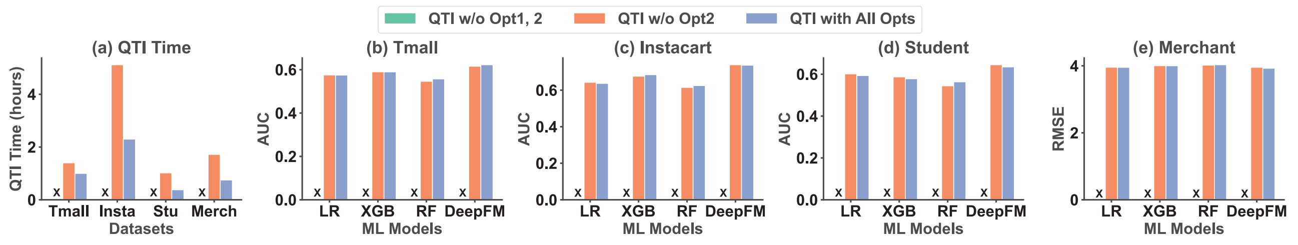


Fig. 5: The ablation study of two optimization in the Query Template Identification component. (a): the running time of the Query Template Identification component w/o the two optimizations. "X" means that the program cannot complete in 6 hours. (b) - (e): the performance comparison among FEATAUG with different Query Template Identification components.

TABLE VIII: Performance of FEATAUG by varying the low-cost proxy. "SC" takes the Spearman Correlation as the low-cost proxy. "MI" takes the Mutual Information as the low-cost proxy. "LR" takes the the Logistic Regression model as the low-cost proxy.

Dataset	Metric	LR			XGB			RF			DeepFM		
		SC	MI	LR	SC	MI	LR	SC	MI	LR	SC	MI	LR
Tmall	AUC ↑	0.5629	**0.5749**	0.5537	0.5854	**0.5898**	0.5888	0.5549	**0.5573**	0.5396	0.6177	**0.6226**	0.6135
Instacart	AUC ↑	0.6168	0.6369	**0.6476**	0.6632	**0.6844**	0.6057	0.6086	0.6248	**0.6670**	0.7266	**0.7364**	0.7269
Student	AUC ↑	**0.5935**	**0.5935**	0.5846	0.5772	**0.5782**	0.5517	0.5687	0.5636	**0.5750**	0.6396	**0.6438**	0.6382
Merchant	RMSE ↓	3.9623	3.9538	**3.9756**	**3.9943**	4.0012	4.0053	**4.0230**	4.0313	4.0666	3.9464	**3.9277**	3.9799

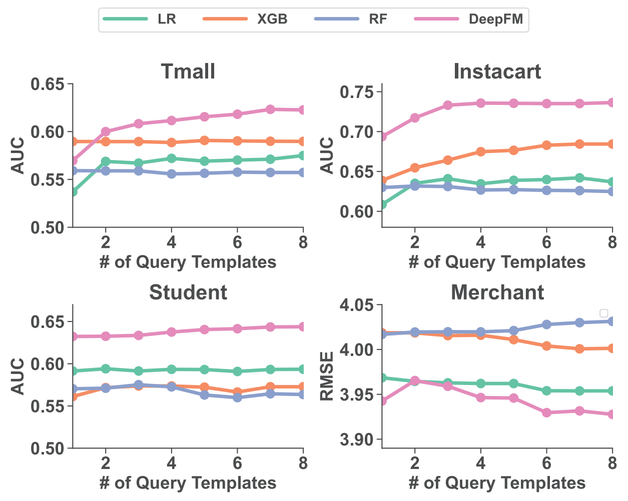


Fig. 6: The trend of performance by varying the number of query templates.

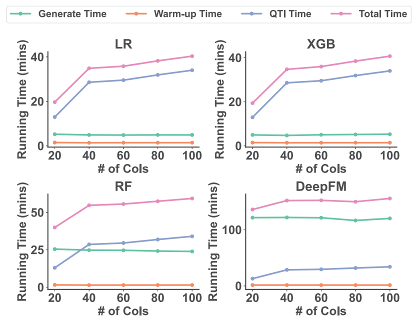
ment to downstream ML models. It shows that considering multiple query templates is more helpful than only a single query template, which matches what data scientists really do in practice. Secondly, there is no fixed number of query templates that fit all scenarios. For *Tmall* dataset with DeepFM model, 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.

2) 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: (1) *Spearman's Correlation (SC)*: Given two variables X and Y, Spearman's Correlation measures the strength and direction of the monotonic relationship between them. (2) *Mutual Information (MI)*: Given two random variables X and Y, MI measures the dependency between the two variables. (3) *Logistic Regression (LR)*: *LR* takes the performance of *LR* model as the proxy of other ML models.

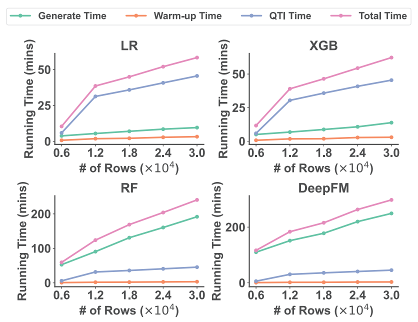
As we can see in Table VIII, *SC* performs best in 2 out of 16 scenarios, *LR* performs best in 3 out of 16 scenarios, and *MI* is the most effective proxy in most cases, i.e. 11 out of 16 scenarios. The result suggests the entropies calculated in *MI* can simulate the performance of ML models well in 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.

F. Scalability Analysis of FeatAug

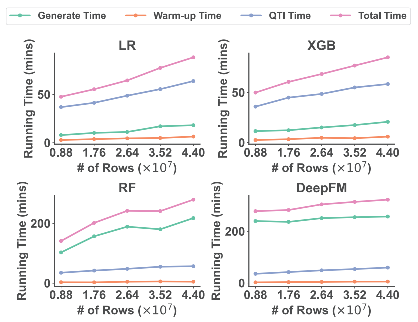
As described in Section IV, the FEATAUG framework includes two components: the SQL Query Generation Component and the Query Template Identification component. Moreover, the SQL Query Generation Component includes the warm-up phase and the query-generation phase. Thus, in the scalability experiments, we split the running time into three parts: QTI Time, Warm-up Time and Generate Time. In this section, we investigate the impact of each part's running time by varing the number of columns in the relevant table R, the number of rows in the training table D and the number of rows in the relevant table R.



(a) The running time of FEATAUG by varying the number of columns in relevant table R on *Student*.



(b) The running time of FEATAUG by varying the number of rows in training table D on *Merchant*.



(c) The running time of FEATAUG by varying the number of rows in relevant table R on *Merchant*.

Fig. 7: Scalability analysis of FEATAUG. "QTI Time": the running time of the Query Template Identification component. "Warm-up Time": the running time of the warm-up phase in the SQL Query Generation component. "Generate Time": the running time of the query-generation phase in the SQL Query Generation component.

1) Varying Number of Columns in Relevant Table R: Since the datasets we utilized do not include wide relevant table with larger than 20 columns, we increase the number of columns by duplicating the original datasets horizontally. For example, we duplicate the *Student* dataset by 13 times to generate a new dataset *Student-Wide* with 130 columns. Because of the space limitation, we present the trend of running time on the *Student-Wide* dataset, which is shown in Figure 7a. Results for other datasets can be found in our technical report [44]. We have two interesting observations from Figure 7a.

Firstly, the Query Template Identification Time does not strictly increase by linear when the number of columns in R increases. Recall that given $attr$, which is a set of attribute from where we select promising attribute combinations (i.e. query templates), the cost of calculating the effectiveness of each query template $T \in \mathcal{S}_{attr}$ is different. That is because the size of Q_T is different for each $T \in \mathcal{S}_{attr}$ and the execution cost of each SQL query $q \in Q_T$ varies. Our cost analysis in Section VI.C indicates **the maximum cost** of query template identification **increase linearly** with the increased number of columns in R. However in practice, even though the time cost does not increase linearly when the column number in R increases, it is still reasonable. Secondly, the Warm-up Time and the Generate Time keeps stable no matter how the number of columns changes. The Warm-up Time is mainly determined by the number of iterations used for warming-up when the query template is fixed. The Generate Time includes the model training time, which depends on the complexity of ML model and the training data.

2) Varying Number of Rows in Training Table D: Because of the space limitation, we present the trend of running time on *Merchant* dataset by varing the number of rows in training table D, which is shown in Figure 7b. Results for other datasets can be found in our technical report [44]. We have three interesting observations from Figure 7b.

Firstly, the Warm-up time increases linearly when the number of rows in D increases. That is because the Warm-up Phase runs TPE on the related low-cost task such as optimizing MI values and the calculation time of MI values is impacted linearly by the number of rows in training table D. Secondly, the Query Template Identification time increases linearly after the number of rows in D is greater than 12k. Thirdly, the increase of the Generate Time is not always linear w.r.t. the increase of the number of rows in D. That is because the Generate Time is mainly occupied by the ML model training time, which depends on the complexity of ML model and the training data. The LR model employs the *Ordinary Least Squares (OLS)* has linear time complexity w.r.t. the number of rows in D, while the time complexity of other models w.r.t the number of rows in D is not linear.

3) Varying Number of Rows in Relevant Table R: Due to space constraints, we show the running time trend for the *Merchant* dataset by varying the number of rows in table R in Figure 7c, with additional dataset results in our technical report [44]. Two key observations emerge from Figure 7c

Firstly, the Warm-up time and the Query Template Identification time increases linearly when the number of rows in R increases. That is because the two phases run TPE on the related low-cost task such as optimizing MI values. When the number of rows in training table D keeps stable, the running time of the two phases is impacted by SQL query execution time, which linearly increases when the number of rows in R increases. Secondly, the increase of the Generate Time is not always linear w.r.t. the increase of the number of rows in R. That is because the Generate Time is mainly occupied by the ML model training time, which depends on the complexity of ML model and the training data.

VIII. CONCLUSION

In this paper, we address the challenge of automatic feature augmentation from one-to-many relationship tables to enhance ML model performance. Our framework, FEATAUG, uses predicate-aware SQL queries for feature set enrichment. We discussed how to extend a widely used Hyperparameter Optimization (HPO) algorithm TPE to our problem and enhance it by warming up the search process. Additionally, we propose a method to identify promising query templates with beam search idea. We reduce the beam search cost through a low-cost effectiveness proxy and ML predictions. Through experiments on four real-world datasets, FEATAUG outperforms the *Featuretools* framework, demonstrating its ability to generate more effective features. Future work will consider mixed database relationships, broadening its real-world application.

References

[1] 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*. IEEE, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/DSAA.2015.7344858

[2] 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*, vol. 55, pp. 1–9, 2015.

[3] 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*, 2016, pp. 155–164.

[4] R. Malhotra and D. K. Malhotra, "Evaluating consumer loans using neural networks," *Omega*, vol. 31, no. 2, pp. 83–96, 2003.

[5] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[6] G. Katz, E. C. R. Shin, and D. Song, "Explorekit: Automatic feature generation and selection," in *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, F. Bonchi, J. Domingo-Ferrer, R. Baeza-Yates, Z. Zhou, and X. Wu, Eds. IEEE Computer Society, 2016, pp. 979–984. [Online]. Available: https://doi.org/10.1109/ICDM.2016.0123

[7] 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 *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds. ACM, 2019, pp. 1936–1945. [Online]. Available: https://doi.org/10.1145/3292500.3330679

[8] F. Horn, R. Pack, and M. Rieger, "The autofeat python library for automated feature engineering and selection," in *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, ser. Communications in Computer and Information Science, P. Cellier and K. Driessens, Eds., vol. 1167. Springer, 2019, pp. 111–120. [Online]. Available: https://doi.org/10.1007/978-3-030-43823-4_10

[9] T. Zhang, Z. Zhang, Z. Fan, H. Luo, F. Liu, W. Cao, and J. Li, "Openfe: Automated feature generation beyond expert-level performance," *CoRR*, vol. abs/2211.12507, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.12507

[10] 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. [Online]. Available: https://openreview.net/pdf?id=688hNNMigVX

[11] 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*. SIAM, 2010, pp. 629–640. [Online]. Available: https://doi.org/10.1137/1.9781611972801.55

[12] 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*. IEEE, 2020, pp. 1645–1656. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00146

[13] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. S. Turaga, "Learning feature engineering for classification," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 2529–2535. [Online]. Available: https://doi.org/10.24963/ijcai.2017/352

[14] 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.*, vol. 13, no. 9, pp. 1373–1387, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1373-chepurko.pdf

[15] 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*. IEEE, 2022, pp. 3360–3372. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00317

[16] 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*, vol. 307, pp. 72–77, 2018. [Online]. Available: https://doi.org/10.1016/j.neucom.2018.03.067

[17] D. Qi, J. Peng, Y. He, and J. Wang, "Auto-fp: An experimental study of automated feature preprocessing for tabular data," in *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, L. Tanca, Q. Luo, G. Polese, L. Caruccio, X. Oriol, and D. Firmani, Eds. OpenProceedings.org, 2024, pp. 129–142. [Online]. Available: https://doi.org/10.48786/edbt.2024.12

[18] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[19] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[20] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri, "Infogather: entity augmentation and attribute discovery by holistic matching with web tables," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 97–108.

[21] M. J. Cafarella, A. Halevy, and N. Khoussainova, "Data integration for the relational web," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1090–1101, 2009.

[22] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang, "A hybrid machine-crowdsourcing system for matching web tables," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 976–987.

[23] P. Wang, R. Shea, J. Wang, and E. Wu, "Progressive deep web crawling through keyword queries for data enrichment," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 229–246.

[24] L. Zhao, Q. Li, P. Wang, J. Wang, and E. Wu, "Activedeeper: a model-based active data enrichment system," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2885–2888, 2020.

[25] P. Wang, Y. He, R. Shea, J. Wang, and E. Wu, "Deeper: A data enrichment system powered by deep web," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1801–1804.

[26] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[27] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html

[28] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy, "Time-bounded sequential parameter optimization," in *International Conference on Learning and Intelligent Optimization*. Springer, 2010, pp. 281–298.

[29] M. Schonlau, W. J. Welch, and D. R. Jones, "Global versus local search in constrained optimization of computer models," *Lecture Notes-Monograph Series*, p. 11–25, 1998. [Online]. Available: http://www.jstor.org/stable/4356058

[30] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 1. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 115–123. [Online]. Available: http://proceedings.mlr.press/v28/bergstra13.html

[31] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., vol. 24. Curran Associates, Inc., 2011. [Online]. Available: https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

[32] 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*, 2011, pp. 507–523.

[33] 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.*, vol. 18, pp. 185:1–185:52, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-558.html

[34] S. Falkner, A. Klein, and F. Hutter, "BOHB: robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1436–1445. [Online]. Available: http://proceedings.mlr.press/v80/falkner18a.html

[35] 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*, vol. 18, no. 1, pp. 6765–6816, 2017.

[36] 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*. Springer, 2011, pp. 507–523.
[37] 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*, vol. 9, no. 3, pp. 307–316, 1977.
[38] 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*, vol. 27, no. 8, pp. 1226–1238, 2005.
[39] 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*, vol. 13, pp. 27–66, 2012.
[40] "Ijcai-15 repeat buyers prediction dataset," https://tianchi.aliyun.com/dataset/dataDetail?dataId=42, 2015.
[41] "Instacart market basket analysis," https://www.kaggle.com/c/instacart-market-basket-analysis, 2017.
[42] "Student dataset," https://www.kaggle.com/competitions/predict-student-performance-from-game-play, 2023.
[43] "Elo merchant category recommendation," https://www.kaggle.com/competitions/elo-merchant-category-recommendation, 2018.
[44] "Feataug: Automatic feature augmentation from one-to-many relationship tables (technical report)," https://github.com/sfu-db/FeatAug/blob/main/FeatAug(Technical_Report).pdf, 2023.
[45] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: A factorization-machine based neural network for CTR prediction," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 1725–1731. [Online]. Available: https://doi.org/10.24963/ijcai.2017/239
[46] "State of Data Science and Machine Learning 2021." https://www.kaggle.com/kaggle-survey-2021, 2021.
[47] J. Bergstra, D. Yamins, D. D. Cox *et al.*, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in science conference*, vol. 13. Citeseer, 2013, p. 20.
[48] "Scikit-learn: Machine learning in python," https://scikit-learn.org/stable/, 2023.
[49] J. Blackard, "Covtype Dataset," UCI Machine Learning Repository, 1998, DOI: https://doi.org/10.24432/C50K5N.
[50] M. Kelly, R. Longjohn, and K. Nottingham, "The uci machine learning repository," https://archive.ics.uci.edu, 2024.
[51] "Household dataset," https://www.kaggle.com/competitions/costa-rican-household-poverty-prediction/overview, 2018.
[52] "Demos of featuretools," https://www.featuretools.com/demos/, 2018.
[53] "Demo of featuretools on household dataset," https://github.com/alteryx/predict-household-poverty, 2018.