Explaining Inference Queries with Bayesian Optimization

Brandon Lockhart* Jinglin Peng* Weiyuan Wu* Jiannan Wang* Eugene Wu†
Simon Fraser University* Columbia University†
{brandon_lockhart, jinglin_peng, youngw, jnwang}@sfu.ca ewu@cs.columbia.edu

ABSTRACT
Obtaining an explanation for an SQL query result can enrich the analysis experience, reveal data errors, and provide deeper insight into the data. Inference query explanation seeks to explain unexpected aggregate query results on inference data; such queries are challenging to explain because an explanation may need to be derived from the source, training, or inference data in an ML pipeline. In this paper, we model an objective function as a black-box function and propose BOExplain, a novel framework for explaining inference queries using Bayesian optimization (BO). An explanation is a predicate defining the input tuples that should be removed so that the query result of interest is significantly affected. BO — a technique for finding the global optimum of a black-box function — is used to find the best predicate. We develop two new techniques (individual contribution encoding and warm start) to handle categorical variables. We perform experiments showing that the predicates found by BOExplain have a higher degree of explanation compared to those found by the state-of-the-art query explanation engines. We also show that BOExplain is effective at deriving explanations for inference queries from source and training data on a variety of real-world datasets. BOExplain is open-sourced as a Python package at https://github.com/sfu-db/BOExplain

1 INTRODUCTION
Data scientists often need to execute aggregate SQL queries on inference data to inspect a machine learning (ML) model's performance. We call such queries inference queries, which can be seen as an SQL query whose expressions may perform model inference. Consider an inference dataset with four variables (customer_id, age, sex, M.predict(I)), where M.predict(I) represents a variable where each value denotes whether the model M predicts the customer will be a repeat buyer or not. Running the following inference query will return the number of female (predicted) repeat buyers:

SELECT COUNT(*) FROM InferenceData as I
WHERE sex = 'female' and M.predict(I) = 'repeat buyer'

If the query result is surprising, e.g., the number of repeat buyers is higher than expected, the data scientist may seek an explanation. One popular explanation method is to find a subset of the input data such that when this subset is removed, and the query is re-executed, the unexpected result no longer manifests [40, 52]. This method is higher than expected, but she wants to double check that the result is not due to a data error. In fact, it turns out that the source data contains errors during Date ∈ [t1, t2], when the website w had network issues; customers confirmed their transactions multiple times, which led to duplicate records in S. The training data extraction UDF was coded to label customers with multiple purchases as repeat buyers, and labelled all of the w customers during the network issue as repeats. The model erroneously predicts every website w customer as a repeat buyer, and thus leads to the high query result. Ideally, the data scientist could ask whether the source data contains an error, and an explanation system would generate a predicate (t1 ≤ Date ≤ t2 AND Website = w).

Unfortunately, existing SQL explanation approaches [1, 39, 40, 52] are ill-equipped to address this setting (Table 1) because they are based on analysis of the query provenance. Although they can generate a predicate explanation over the inference data, the provenance analysis does not extend across model training nor UDFs, which are prevalent in data science workflows. The recent system Rain [53] generates fine-grained explanations for inference queries. It relaxes the inference query into a differentiable function over the model's prediction probabilities, and leverages influence analysis [25] to estimate the query result's sensitivity to a training record. However, Rain returns training records rather than predicates, and estimating the model prediction sensitivity to group-wise changes to the training data remains an open problem. Further, Rain does not currently support UDFs and uses a white-box approach that is

| Methodology      | SQL Explain [1, 39, 40, 52] | Inference Query Explain |
|------------------|-----------------------------|-------------------------|
| Source Data      | Supported                    | Supported               |
| Training Data    | Not Supported                | Supported               |
| Inference Data   | Supported                    | Supported               |
| Explanation Type | Coarse-grained               | Fine-grained            |
|                  | Black-box                    | Black-box               |

Table 1: Comparison of BOExplain and existing approaches.

Generating an explanation from inference data can certainly help to understand the answer to an inference query. However, an ML pipeline does not only contain inference data but also training and source data. The following example illustrates a scenario where an explanation should be generated from source data.

Example 1.1. CompanyX creates an ML pipeline (Figure 1(a)) to predict repeat customers for a promotional event. CompanyX receives transaction records from several websites that sell their products and aggregates them into a source data table S. Next, the user defined function (UDF) make_training(·) extracts and transforms features into the training dataset T. Finally, a random forest model is fit to the training data, and the model is applied to the inference dataset I which updates it with a prediction variable, M.predict(I).

For validation purposes, the data scientist writes a query to compute the percentage of repeat buyers. The rate is higher than expected, but she wants to double check that the result is not due to a data error. In fact, it turns out that the source data S contains errors during Date ∈ [t1, t2], when the website w had network issues; customers confirmed their transactions multiple times, which led to duplicate records in S. The training data extraction UDF was coded to label customers with multiple purchases as repeat buyers, and labelled all of the w customers during the network issue as repeats. The model erroneously predicts every website w customer as a repeat buyer, and thus leads to the high query result. Ideally, the data scientist could ask whether the source data contains an error, and an explanation system would generate a predicate (t1 ≤ Date ≤ t2 AND Website = w).

Unfortunately, existing SQL explanation approaches [1, 39, 40, 52] are ill-equipped to address this setting (Table 1) because they are based on analysis of the query provenance. Although they can generate a predicate explanation over the inference data, the provenance analysis does not extend across model training nor UDFs, which are prevalent in data science workflows. The recent system Rain [53] generates fine-grained explanations for inference queries. It relaxes the inference query into a differentiable function over the model's prediction probabilities, and leverages influence analysis [25] to estimate the query result's sensitivity to a training record. However, Rain returns training records rather than predicates, and estimating the model prediction sensitivity to group-wise changes to the training data remains an open problem. Further, Rain does not currently support UDFs and uses a white-box approach that is
Figure 1: An illustration of using BOExplain to generate an explanation from source data in an ML pipeline.
where \( \text{agg} \) is an aggregation operation and \( C_j \) is a filter condition.

**Example 2.1.** Returning to the running example in Section 1, the user queries the predicted repeat buyer rate. This can be expressed as \( Q = q_1/q_2 \), an arithmetic expression over \( q_1 \) and \( q_2 \) where

\[
q_1 = \text{SELECT COUNT(\*) FROM I WHERE M.predict(I)='repeat buyer'} ;
q_2 = \text{SELECT COUNT(\*) FROM I}.
\]

**Complaint.** After the user executes a query, she may find that the result is unexpected and complain about its value. In this work, the user can complain about the result being too high or too low, as done in [40]. We use the notation \( \text{dir} = \text{low}(\text{dir} = \text{high}) \) to indicate that \( Q \) is unexpectedly high (low).

**Example 2.2.** In our running example, the user found the repeat buyer rate too high. Thus along with the query \( Q \) from Example 2.1, the user specifies \( \text{dir} = \text{low} \) to indicate that \( Q \) should be lower.

**Explanation.** After the user complains about a query result, BOExplain will return an explanation for the complaint. In this work, we define an explanation as a predicate over given variables.

**Definition 2 (Explanation).** Given numerical variables \( N_1, \ldots, N_n \) and categorical variables \( C_1, \ldots, C_m \), an explanation is a predicate \( p \) of the form

\[
p = l_1 \leq N_1 \& \cdots \& l_n \leq N_n \& C_1 = c_1 \& \cdots \& C_m = c_m.
\]

The set of all such predicates forms the predicate space \( S \).

**Example 2.3.** The source data in Figure 1 contains the variables Date and Website. An example explanation over these variables is

\[
12/01/2020 \leq \text{Date} \leq 12/10/2020 \& \text{Website} = \text{w}.
\]

**Objective Function.** Next we define our objective function. The goal of our system is to find the best explanation for the user’s complaint. Hence, we need to measure the quality of an explanation. For a predicate \( p \), let \( \sigma_{\rightarrow p}(R) \) represent \( R \) filtered to contain all tuples that do not satisfy \( p \). We apply the query to \( \sigma_{\rightarrow p}(R) \) and get the new query result. If the user specifies \( \text{dir} = \text{low} \), then the smaller the new query result is, the better the explanation is. Hence, we use the new query result as a measure of explanation quality. The objective function is formally defined in Definition 3.

**Definition 3 (Objective Function).** Given a predicate \( p \), relation \( R \), and query \( Q = E(q_1, \ldots, q_k) \), the objective function \( \text{obj}(p, R, Q) \rightarrow \mathbb{R} \) applies \( Q \) on the relation \( \sigma_{\rightarrow p}(R) \).

With the definition of objective function, the problem of searching for the best explanation is equivalent to finding a predicate that minimizes or maximizes the objective function.

**Definition 4 (SQL Explanation Problem).** Given a relation \( R \), query \( Q = E(q_1, \ldots, q_k) \), direction \( \text{dir} \), and predicate space \( S \), find the predicate \( p^* = \arg\min_{p \in S} \text{obj}(p, R, Q) \) if \( \text{dir} = \text{low} \) (use arg max if \( \text{dir} = \text{high} \)).

It may appear that minimizing the above objective function runs the risk of overfitting to the user’s complaint (perhaps with an overly complex predicate). However, a regularization term can be placed within the objective function—for instance, SQL explanation typically regularizes using the number of tuples that satisfy the predicate [52]. Since \( Q \) is an arithmetic expression over multiple queries, one of those queries may simply be the regularization term.

2.2 Extension to Inference Query Explanation

For inference query explanation, we focus on three input datasets that the user can generate explanations from: source, training, and inference. The query processing pipeline is as follows (Figure 1(a)):

1. Transform and featurize the source data into the training data.
2. Train an ML model over the training data.
3. Use the model to predict a variable from the inference dataset.
4. Issue a query over the inference dataset.

From the above workflow, we can find that there are two differences between SQL and inference query explanations: 1) the query for inference query explanation is evaluated on the inference data with model predictions, and 2) inference query explanation, the user may want an explanation for the input dataset at any step of the workflow (e.g., the source, training, or inference dataset), while SQL explanation only consider the query’s direct input.

We next formally define the scope of the errors that we seek to explain in Definition 5.

**Definition 5 (Scope of Errors).** This paper focuses on errors in the form of systematically mislabelled tuples that can be described using a predicate as defined in Definition 2.

We next extend the objective function from SQL explanation to inference query explanation. Let \( Q \) be the query issued by the user over the updated inference data, with the same form as in Definition 1. Let \( R \) be the data that we want to derive an explanation from (it can be source, training, or inference data) and \( p \) be an explanation (i.e., predicate) over \( R \). We measure the quality of \( p \) like in SQL explanation: filter the data by \( p \), then get the new query result. Note that for inference query explanation, the query is issued over the updated inference data. Hence, we define \( \mathcal{P} \) as the subset of the ML pipeline that takes as input the dataset \( R \) that we wish to generate an explanation from, and that outputs the updated inference data which is used as input to the SQL query. The extended objective function is defined in Definition 6.

**Definition 6 (Objective Function).** Given a subset of an ML pipeline \( \mathcal{P} \), a predicate \( p \), relation \( R \), and query \( Q \), the objective function \( \text{obj}(p, R, \mathcal{P}, Q) \rightarrow \mathbb{R} \) feeds \( \sigma_{\rightarrow p}(R) \) through \( \mathcal{P} \), and then applies \( Q \) on the inference data.

Finally, we define the inference query explanation problem.

**Definition 7 (Inference Query Explanation Problem).** Given a relation \( R \), query \( Q \), direction \( \text{dir} \), pipeline \( \mathcal{P} \), and predicate space \( S \), find the predicate \( p^* = \arg\min_{p \in S} \text{obj}(p, R, Q, \mathcal{P}) \) if \( \text{dir} = \text{low} \) (use arg max if \( \text{dir} = \text{high} \)).

We assume that an explanation in the form of Definition 2 that performs well under the objective function in Definition 7 is meaningful to the user. Hence, if \( \text{dir} = \text{low} (\text{high}) \), the predicate \( p^* \) that minimizes (maximizes) the objective function is considered optimal.

3 THE BOEXPLAIN FRAMEWORK

This section introduces Bayesian optimization (BO) and presents the BOExplain framework.

---

1In general, any intermediate dataset is acceptable, however we focus on these three due to their prevalence and to simplify the paper.
Figure 2: Suppose TPE has observed six points \( D = \{(-5, 25), (-3, 9), (-1, 1), (2, 4), (3, 9), (3.5, 9.25)\} \). This figure illustrates how TPE finds the next point to evaluate \((y = 35\%)\).

| Age | Sex | City | State | Occupation | M.predict() |
|-----|-----|------|-------|------------|-------------|
| 48  | F   | Mesa | AZ    | Athlete    | repeat      |
| 45  | F   | Mesa | FL    | Artist     | repeat      |
| 46  | M   | Mesa | AZ    | Writer     | one-time    |
| 40  | M   | Miami| FL    | Athlete    | repeat      |
| 42  | F   | Miami| FL    | Athlete    | repeat      |

Table 2: An illustration of parameter creation.

3.1 Background

Black-box optimization aims to find the global minima (or maxima) of a black-box function \( f \) over a search space \( \mathcal{X} \).

\[
x^* = \min_{x \in \mathcal{X}} f(x).
\]

BO is a sequential model-based optimization strategy to solve the problem, where \textit{sequential} means that BO is an iterative algorithm and \textit{model-based} means that BO builds surrogate models to estimate the behavior of \( f \).

Tree-structured Parzen Estimator (TPE). TPE [8, 10] is a popular BO algorithm. It first initializes by evaluating \( f \) on random samples from the search space. Then, it iteratively selects \( x \) from the search space using an \textit{acquisition function} and evaluates \( f(x) \). Let \( D = \{(x_1, f(x_1)), (x_2, f(x_2)), \cdots, (x_t, f(x_t))\} \) denote the set of samples evaluated in previous iterations. TPE chooses the next sample as follows:

1. Partition \( D \) into sets \( D^\gamma \) and \( D^b \), where \( D^\gamma \) consists of the set of \( \gamma \)-percentile points with the lowest \( f(x) \) values in \( D \), and \( D^b \) consists of the remaining points \((\gamma \text{ is a user-definable parameter)}\). Since the goal is minimize \( f(x) \), \( D^\gamma \) is called the \textit{good-point set} and \( D^b \) is called the \textit{bad-point set}. Intuitively, good points lead to smaller objective values than bad points.

2. Use Parzen estimators (a.k.a. kernel density estimators) to build a density model \( g(x) \) and \( b(x) \) over \( \mathcal{D}^\gamma \) and \( \mathcal{D}^b \), respectively. Intuitively, given an unseen \( x^* \) in the search space, the density models \( g(x^*) \) and \( b(x^*) \) can return the probability of \( x^* \) being a good and bad point, respectively. Note that separate density models \( g(x) \) and \( b(x) \) are constructed for each dimension of \( \mathcal{X} \).

3. Construct an acquisition function \( g(x)/b(x) \) and select \( x \) with the maximum \( g(x)/b(x) \) to evaluate in the next iteration. Intuitively, TPE selects a point that is more likely to appear in the good-point set and less likely to appear in the bad-point set.

Figure 2 illustrates the three steps. A complete introduction to TPE is given in Appendix A.

3.2 Our Framework

In this section, we describe the BOExplain framework.

Parameter Creation. Given a predicate space, we need to map it to a parameter search space (the parameters and their domains). Suppose a predicate space is defined over variables \( A_1, A_2, \cdots, A_n \). If \( A_i \) is numerical (e.g., age, date), two parameters are created that serve as bounds on the range constraint. Specifically, the parameters \( A_{i_{\text{min}}} \) and \( A_{i_{\text{max}}} \) define the lower bound and the length of the range constraint, respectively. \( A_{i_{\text{min}}} \) and \( A_{i_{\text{max}}} \) have interval domains \([\min(A_i), \max(A_i)]\) and \([0, \max(A_i) - \min(A_i)]\), respectively.

If \( A_i \) is categorical (e.g., sex, website), one categorical parameter is created with a domain consisting of all unique values in \( A_i \).

Example 3.1. Suppose the user defines a predicate space over \textit{State} and \textit{age} in Table 2. BOExplain creates three parameters: one categorical parameter for \textit{State} with domain \{AZ, FL\}, and two numerical parameters for \textit{age} with domains \{40, 48\} and \{0, 8\}, respectively.

BOExplain Framework. Figure 3 walks through the BOExplain framework. In step (1), the user provides an objective function \( obj \), a relation \( S \), and predicate variables \( A_1, \cdots, A_n \) (Figure 1(b), line 10). Step (2) creates the parameters and their domains. Step (2) runs one iteration of TPE, starting with the parameters from step (1), and outputs a predicate. Step (2) and (3) evaluate the predicate by removing those tuples from the input dataset, and evaluating \( obj \) on the filtered data. The result is passed to TPE for the next iteration, and possibly yielded to the user as an intermediate or final predicate explanation.

Consider the example code in Figure 1(b). Once it is executed, BOExplain first creates three parameters: \( \text{Date}_{\text{min}}, \text{Date}_{\text{max}}, \) and Website along with the corresponding domains. Then, it iteratively calls TPE to propose predicates (e.g., “12/01/2020 ≤ Date ≤ 12/02/2020 AND Website = ‘w’”). BOExplain obtains \( S_{\text{filtered}} \) by removing the tuples that satisfy this predicate from \( S \). Next, it applies \( obj(\cdot) \) to \( S_{\text{filtered}} \) which will rerun the pipeline (Figure 1(a)) to compute the updated repeat buyer rate. The predicate and the updated rate are passed to TPE to use when selecting the predicate on the next iteration. This iterative process will repeat until the time budget is reached. When the user stops BOExplain, or when the optimization has converged, the predicate with the lowest repeat buyer rate is returned.
Why Is TPE Suitable For Query Explanation? Recent work [7, 28, 31] has suggested that random search is a competitive strategy for hyperparameter tuning across a variety of challenging machine learning tasks. However, we find that TPE is more effective for query explanation because it is designed for problems where similar parameter values tend to have similar objective values (e.g., model accuracy). TPE can leverage this property to prune poor regions of the search space. As a trivial example, suppose a hyperparameter controls the number of trees in a random forest. If values 10, 12, 14 have resulted in a poor objective value, then TPE will down-weight similar values (e.g., 9, 16).

This property tends to hold in query explanation, because similar predicates tend to have similar objective values. For instance, we would expect that the predicate age $\in [10, 20]$ will exhibit a similar objective to age $\in [10, 19]$ and age $\in [10, 21]$; when the former has a poor objective value, the latter two may be pruned. When this property does not hold, BOExplain can still find the optimal predicate via the exploration component of BO. BO balances two components for selecting a point to evaluate: 1) exploration of the search space, and 2) exploitation of points similar to previously well-performing points. Exploration may be ineffective if similar predicates do not perform similarly under the objective function, but exploitation will still test unpromising predicates, thus eventually leading BO to the optimal predicate.

4 SUPPORTING CATEGORICAL VARIABLES
In this section, we present our techniques to enable BOExplain to support categorical variables more effectively.

4.1 Individual Contribution Encoding
Recall that in Section 3.1, TPE models numerical and categorical variables using kernel density estimation and categorical distribution, respectively. The advantage of kernel density estimation over a categorical distribution is that it can estimate the quality of unseen points based on the points that are close to them. To benefit from this advantage, we map a categorical variable to a numerical variable. We call this idea categorical encoding. In the following, we present our categorical encoding approach, called individual contribution (IC) encoding.

A good encoding method should put similar categorical values close to each other. Intuitively, two categorical values are similar if they have a similar contribution to the objective function value. Based on this intuition, we rank the categorical values by their individual contribution to the objective function value. Specifically, consider a categorical variable $C$ with domain$(C) = \{c_1, \ldots, c_k\}$. For each value $c_i$, we obtain the filtered dataset $\sigma_{C \neq c_i}(S)$ w.r.t. the predicate $C = c_i$. Next, the objective function is evaluated on the relation $\sigma_{C \neq c_i}(S)$ which returns a number. This number can be interpreted as the contribution of the categorical value on the objective function. After repeating for all values $c_i$, the categorical values are mapped to consecutive integers in order of their IC. BOExplain will then use a numerical rather than categorical variable to model $C$.

Example 4.1. Suppose we would like an explanation from the inference data in Table 2. Suppose the objective function value is the repeat buyer rate and the predicate space is defined over the Occupation variable. Note that the Occupation variable has the domain [Athlete, Artist, Writer]. The IC of Athlete is determined by removing the tuples where Occupation=“Athlete” and computing the objective function on the filtered dataset, which gives 0.5 (since only one of the two tuples in the filtered dataset is a repeat buyer). Similarly, the ICs of Artist and Writer are 0.75 and 1 respectively. Finally, we sort the categorical values by their objective function value and encode the values as integers: Athlete → 1, Artist → 2, Writer → 3.

4.2 Warm Start
We next propose a warm-start approach to further enhance BOExplain’s performance for categorical variables. Since an IC score has been computed for each categorical value, we can prioritize predicates that are composed of well performing individual categorical values. Rather than selecting $n_{init}$ points at random to initialize the TPE algorithm, we select the $n_{init}$ combinations of categorical values with the best combined score. More precisely, for a variable $C_i$, we consider the tuple pairs (variable IC) as computed in Section 4.1, $S^{IC}_{ij} = \{(c_j, IC(c_j))\}_{i \in \sigma}^{n_{init}}$, where $n_i$ is the number of unique values in variable $C_i$. Next, we compute the d–ary Cartesian product and add the ICs for each combination $\sum_{i=1}^{n_{init}} IC(c_i) + \sum_{i=1}^{n_{init}} IC(c_i) \in \{((c_1, \ldots, c_d), IC(c_1), \ldots, IC(c_d)) | i \in \{1, \ldots, n_i\}\}$.

Example 4.2. The IC for values in the Occupation variable were computed in Example 4.1. $S^{IC}_{Occupation} = \{(Athlete, 0.5), (Artist, 0.75), (Writer, 1)\}$, and for Sex we have $S^{IC}_{Sex} = \{(F, 0.5), (M, 1)\}$. Next we compute the combined IC score for each combination of predicates $\sum_{i=1}^{n_{init}} IC(c_i) \in \{((Athlete, F), (Athlete, M), (Writer, M), 2)\}$.

To see why adding ICs can be useful for prioritizing good predicates, suppose we want to minimize the objective function, and that $C_1 = c_1$ and $C_2 = c_2$ have small ICs. Then it is likely that $C_1 = c_1 \land C_2 = c_2$ has a small value. So we choose to sum the IC values as it encodes this property. Finally, we select $n_{init}$ valid predicates with the best combined IC score. Recall the user defines the direction that the objective function should be optimized. Therefore, we select the predicates with the smallest (largest) IC score if the objective function should be minimized (maximized). If the predicate also contains numerical variables, values are selected at random to initialize the range constraint parameters.

Example 4.3. Continuing with Example 4.2, recall that we want to minimize the objective function, so the smaller the combined IC score the better. Suppose $n_{init} = 2$, then on the first and second iterations of BO, we evaluate the predicates $Occupation = “Athlete” \land Sex = “F”$ and $Occupation = “Artist” \land Sex = “F”$ respectively. Note that $Occupation = “Athlete” \land Sex = “F”$ is the best predicate, so adding IC scores can prioritize good explanations.

4.3 Putting Everything Together
We lastly present the full BOExplain algorithm in Algorithm 1. First, the ICs for the categorical variables are computed in lines 1-3. Next, the parameters and domains are created in line 4. In line 5, the IC values are used to prioritize predicted high quality predicates, and in line 6 TPE is initialized for $n_{init}$ iterations with the predicted high quality predicates. Starting from line 7, we use a model to select the next points. In line 8, the previously evaluated points are split into good and bad groups based on $y$. Next, from line 9, a value is selected for each parameter. In lines 10 and 11, distributions of the good and bad groups are modelled, respectively. In line 12, points are sampled from the good distribution, and the sampled point with the largest expected improvement is selected as the next parameter value (line 13). In line 15, the objective function is evaluated based
on the parameter assignment, and the set of observation-value pairs is updated.

5 EXPERIMENTS

Our experiments seek to answer the following questions. (1) How does BOExplain compare to current state-of-the-art query explanation engines for numerical variables? (2) Are the IC encoding and warm start heuristics effective? (3) How effective is BOExplain at deriving explanations from source and training data? (4) Can BOExplain generate useful explanations for real corrupted datasets?

5.1 Experimental Settings

5.1.1 Baselines. For SQL-only queries, we compare BOExplain with the explanation engines Scorpion [52] and MacroBase [1, 5] which return predicates as explanations. For inference queries, no predicate-based explanation engines exist, so we compare with a random search baseline [7] and Hyperband [27]. Random search is a competitive method for hyperparameter tuning [7]. The parameters are chosen independently and uniformly at random from the domains described in Section 3.2.

5.1.2 Real-world Datasets and ML Pipelines. The following lists the five real-world datasets used in our experiments. We visualize the pipelines for House and Credit in Figure 5, and put a green box around the data where an explanation is derived in each pipeline.

- **House price prediction** [14]. This dataset was published already split into training (1460 rows) and inference (1459 rows) tables. It contains 79 variables of a house which are used to train a support vector regression model to predict the house price. The pipeline denoting how to prepare the data for modelling is given in Figure 5(a).
- **Credit card approval prediction** [3]. The source data consists of two tables: application_record (438,357 rows, 18 variables), which contains information about previous applicants, and credit_t_record (1,048,575 rows, 3 variables), which stores the applicants’ credit history. The pipeline to prepare the data for modelling is given in Figure 5(b), and a decision tree classifier is trained to predict whether a customer will default on their credit card payment. We set aside 20% of the data to use for the inference query, and 80% for training.
- **Amazon product reviews** [37]. This dataset contains 6928 reviews of Amazon products with ground truth and crowdsourced binary labels. 80% of the reviews are used for training with labels formed from the majority vote of the crowdsourced labels, and 20% for testing with the ground truth labels. We encode the reviews using Count Vectorization and train a support vector classifier.
- **German credit risk** [16]. This dataset contains 19 variables of 1000 bank customers with each customer labelled as having good or bad credit risk. We one-hot encode the categorical variables, do an 80-20 train-test split, and train an XGBoost classifier.
- **NYC yellow taxi dataset** [5]. This dataset contains taxi trip information for every yellow taxi trip in New York City. Following the setup in [4], we predict the hourly demand by region for the 20 most frequent regions, and use the features weekday, region, demand of previous 24 hours, and cosine/sine features to encode that hours are cyclical. We train an XGBoost regressor on data from January and February, 2020, and perform inference on data from March 2020.

5.1.3 Metrics. To measure the quality of an explanation, we plot the best objective function value achieved by each time point \( t \). For Scorpion and MacroBase, we plot the objective function value corresponding to their output predicate as a line that begins when the system finishes. To evaluate the effectiveness at identifying data errors, we measure the F-score, precision, and recall in the

---

1https://github.com/sirrice/scorpion
2https://github.com/standford-futuredata/macrobase
3https://www.kaggle.com/rkidfox/credit-card-approval-prediction
4https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
experiments on the House and Credit datasets. We synthetically corrupt data defined by a predicate, and use that data as ground truth. Precision is the number of selected corrected tuples divided by the total number of selected tuples. Recall is the number of selected corrupted tuples divided by the total number of corrupted tuples. F-score is the harmonic mean of precision and recall. For BOExplain, Random, and Hyperband, each result is averaged over 10 runs.

5.1.4 Implementation. BOExplain was implemented in Python 3.9. The code is open sourced at https://github.com/sfu-db/BOExplain. We modify the TPE algorithm in the Optuna library [2] with our optimization for categorical variables. The ML models in Section 5.3 are created with sklearn. The experiments were run single-threaded on a MacBook Air (OS Big Sur, 8GB RAM). In the TPE algorithm, we set \( n_{init} = 10 \), \( n_{ei} = 24 \), and \( y = 0.1 \) for all experiments.

5.2 Explaining SQL-Only Queries

To compare BOExplain, Scorpion, and MacroBase, we replicate the experiment from Section 8.3 of Scorpion's paper [52], using the same datasets, query, and objective function. Note that MacroBase explicitly aims to optimize Scorpion’s objective function, as described in Section 2.2 of [1]. The dataset consists of a single group by variable \( A_J \), an aggregate variable \( A_m \), and search variables \( A_1, \ldots, A_n \) with domain \((A_i) = [0, 100] \subseteq \mathbb{R} \), \( i \in [n] \). \( A_J \) contains 10 unique values (or 10 groups) each corresponding to 2000 tuples randomly distributed in the \( n \) dimensions. 5 groups are outlier groups and the other 5 are holdout groups. Each \( A_J \) value in a holdout group is drawn from \( N(0, 10) \). Outlier groups are created with two \( n \) dimensional hyper-cubes over the \( n \) variables, where one is nested inside the other. The inner cube contains 25% of the tuples and \( A_J \sim N(\mu, 10) \), and the outer cube contains 25% of the tuples in the group and \( A_J \sim N(\frac{\mu + 10}{2}, 10) \), else \( A_J \sim N(10, 10) \), \( \mu \) is set to 80 for the “easy” setting (the outliers are more pronounced), and 30 for the “hard” setting (the outliers are less pronounced). The query is SELECT SUM(\( A_J \)) FROM synthetic GROUP BY \( A_J \). The arithmetic expression over the SQL query is defined in Section 3 of [52] that forms an objective function to be maximized. The penalty \( c = 0.2 \) was used to penalize the number of tuples removed as described in Section 7 of [52]. We used \( n = 2 \) and \( n = 3 \) since 3 is the maximum number of variables supported by MacroBase.

The results are shown in Figure 4. BOExplain outperforms Scorpion and MacroBase in terms of optimizing the objective function in each experiment. This is because BOExplain can refine the constraint values of the range predicate which enables it to outperform Scorpion and MacroBase which discretize the range. The results are the same in the easy and hard settings. MacroBase performs poorly because the predicates formed by discretizing the variable domains into equi-sized bins, and computing the cube, do not optimize this objective function. This exemplifies a known limitation of MacroBase that binning continuous variables is difficult [1].

BOExplain also outperforms Scorpion in terms of running time. BOExplain achieves Scorpion’s objective function value in around half the time on each experiment.

Note. The focus of this paper is not on SQL-only queries, thus we did not conduct a comprehensive comparison with Scorpion and MacroBase. This experiment aims to show that a black-box approach (BOExplain) can even outperform white-box approaches (Scorpion and MacroBase) for SQL-only queries in some situations.

5.3 Explaining Inference Queries

In this section, we evaluate BOExplain’s efficacy at explaining inference queries from source and training data. In Section 5.3.1, we investigate BOExplain’s approach for categorical variables on
House, and in Section 5.3.2, we evaluate BOExplain in a complex ML pipeline on Credit, where an explanation is derived from source data.

### 5.3.1 Supporting Categorical Variables

In this experiment, we assess BOExplain’s method for handling categorical variables on House. The data is corrupted by setting the tuples satisfying Neighbourhood='CollgCr' ∧ Exterior1st='VinylSd' ∧ 2000 ≤ YearBuilt ≤ 2010 to have their sale price multiplied by 10, affecting 6.16% of the data. We query the average predicted house price and seek an explanation for why it is high. To assess BOExplain’s efficacy at removing the corrupted tuples, we define the objective function to minimize the distance between the queried result on the passed data and the result of the query issued on the data with the corrupted tuples removed. We use two categorical search variables Neighbourhood and Exterior1st which have 25 and 15 distinct values respectively, and one numerical search variable YearBuilt which has domain [1872, 2010]. The search space size is 7.25 × 10^6.

In this experiment, we compare three strategies for dealing with categorical variables. The first, BOExplain, is our algorithm with both of the IC encoding and warm-start (WS) optimizations proposed in Section 4. To determine whether encoding categorical values to integers based on IC and using a numerical distribution is effective, we consider a second approach, BOExplain (w/o IC), which uses the warm start optimization from Section 4.2, but uses the TPE categorical distribution to model the variables rather than encoding. The third, BOExplain (w/o IC and WS), is BOExplain without any optimizations.

Each method is run for 200 seconds, and the results are shown in Figure 6. The benefit of the warm start is apparent since BOExplain and BOExplain (w/o IC) outperform the other baselines much sooner. Also, BOExplain significantly outperforms BOExplain (w/o IC) which shows that encoding the categorical values, and using a numerical distribution to model the parameter, leads to BO learning the good region which can optimize the objective function when exploited. The F-score, precision, and recall also demonstrate how BOExplain can significantly outperform the baselines. In this experiment, BOExplain completed on average 274.3 iterations, whereas random completed 1148.4 iterations.

#### 5.3.2 Explanation From Source Data

In the last experiment, we derive an explanation from source data on Credit. We corrupt the source data by setting all applicant records satisfying −23000 ≤ DAYS_BIRTH ≤ −17000 ∧ 2 ≤ CNT_FAM_MEMBERS ≤ 3 to have a 'bad' credit status, which affects 20.1% of the data. Corruption decreases the accuracy of the model, and we define the objective function to increase the model accuracy. We derive an explanation from the source data table application_record with the variables DAYS_BIRTH and CNT_FAM_MEMBERS which have domains [-25201, -7489] and [1, 15], respectively, and the size of the search space is 7.06 × 10^10.

The experiment is run for 200 seconds, and the results are shown in Figure 7. On average, BOExplain completes 246.8 iterations and random search completes 319.6 iterations during the 200 seconds. BOExplain significantly outperforms Random at optimizing the objective function, as BOExplain on average attains an objective function value at 51 seconds that is higher than the average value Random attains at 200 seconds. This shows that exploiting promising regions can lead to better explanations, and that BOExplain is effective at deriving explanations from source data that passes through an ML pipeline. Although random search can find an explanation with high precision, BOExplain significantly outperforms Random in terms of F-score.

#### 5.4 Case Studies

To understand how BOExplain performs on real workloads, we present three case studies in this section. These case studies use BOExplain to derive an explanation from three real-world datasets under realistic settings. The derived explanations are insightful, which show BOExplain’s effectiveness in real-world applications.

**Crowdsourced Mislabels.** With the Amazon dataset, the inference accuracy is 93.75%. To investigate whether mislabelled reviews decrease the accuracy, we define the objective function to increase the accuracy. We derive a predicate over the variables Country and TextWordCount (the number of words in the review) in the source dataset. After running BOExplain for 60 seconds, the output predicate is Country = "Turkey" ∧ 101 ≤ TextWordCount ≤ 221, which increases the test accuracy to 95%. Upon further inspection, we found that the labelling accuracy of the training data is 93%, but the labelling accuracy of the tuples satisfying the returned predicate is 90%. Hence, BOExplain identified that labellers from Turkey were more likely to mislabel long reviews, which degraded the model.
Mean Objective Function Value

0.70

0.80

BOExplain

2020-03-31 23:00:00

When the tuples that satisfy this predicate are removed, the MSE on the inference data is 8706.13. In fact, the MSE of the tuples satisfying the predicate is 18151.93, which is much higher than the overall MSE of the inference data (10658.49). Therefore, BOExplain identified that long-term loans for new cars are greatly biased in favour of people over 25 years old.

Data Drift. With the NYC dataset, the mean-squared error (MSE) of taxi trip durations is 1801.03 on the training data and 10658.49 on the inference data. The objective function is to minimize the MSE on the inference data, and we search for an explanation over the inference data with the search variables Purpose and DurationInMonths (duration of the loan).

We ran BOExplain for 60 seconds, and the returned predicate is Purpose = “car (new)” ∧ 20 ≤ DurationInMonths ≤ 50. After removing the tuples satisfying this predicate, the predicted rate of having good credit risk is 74% and 68% for people over and under 25, respectively, which is significantly less biased. Moreover, in the training data, the overall good credit risk rate is 71% and 60% for people over and under 25, respectively, however for the tuples satisfying the predicates, the rates are 57% and 17%. Therefore, BOExplain identified that long-term loans for new cars are greatly biased in favour of people over 25 years old.

Bias. With the German credit dataset, the predicted rate of good credit risk for individuals 25 years old or older is 76%, and for individuals under 25 years old it is 57%, hence this dataset is biased. We define the objective function to minimize the predicted rate of good credit risk between those over and under 25 years old, and derive a predicate from the training data with the search variables Purpose and DurationInMonths (duration of the loan).

We ran BOExplain for 60 seconds, and the returned predicate is Purpose = ”car (new)” ∧ 20 ≤ DurationInMonths ≤ 50. After removing the tuples satisfying this predicate, the predicted rate of having good credit risk is 74% and 68% for people over and under 25, respectively, which is significantly less biased. Moreover, in the training data, the overall good credit risk rate is 71% and 60% for people over and under 25, respectively, however for the tuples satisfying the predicates, the rates are 57% and 17%. Therefore, BOExplain identified that long-term loans for new cars are greatly biased in favour of people over 25 years old.

Data Drift. With the NYC dataset, the mean-squared error (MSE) of taxi trip durations is 1801.03 on the training data and 10658.49 on the inference data. The objective function is to minimize the MSE on the inference data, and we search for an explanation over the inference data with the search variables Region and PickUpDateTime.

After running BOExplain for 30 seconds, the returned predicate is Region = "230" ∧ 2020-03-12 21:00:00 ≤ PickUpDateTime ≤ 2020-03-31 23:00:00. When the tuples that satisfy this predicate are removed, the MSE on the inference data is 8706.13. In fact, the MSE of the tuples satisfying the predicate is 18151.93, which is much higher than the overall MSE of the inference data (10658.49). Thus, BOExplain identified a region and time period that underwent significant drift as a result of the COVID-19 pandemic.

6 RELATED WORK

Our work is mainly related to query explanation, ML pipeline debugging, and Bayesian optimization.

Query Explanation. BOExplain is most closely related to Scorpion [52] and the work of Roy and Suciu [40]. Both approaches define explanations as predicates. Scorpion uses a space partitioning and merging process to find the predicates, while Roy and Suciu [40] use a data cube approach. Both systems make assumptions about the aggregation query’s structure in order to benefit from their white-box optimizations. In contrast, BOExplain supports complex queries, model training, and user-defined functions. Further, BOExplain is a progressive algorithm that improves the explanation over time. Variations of these ideas include the DPPF operator [1], explanation-ready databases [39], and counterbalances [33]. Finally, a number of specialized systems focus on explaining specific scenarios, such as streaming data [5], map-reduce jobs [24], online transaction processing workloads [54], cloud services [38], and range-radius queries [45].

Another related concept is the online analytical processing (OLAP) data cube [21] which is used to explore and discover insights about subsets of multidimensional queries. Much previous work has been dedicated to providing the user with more meaningful and efficient exploration of the data cube [42–44]. Other work has used the data cube concept to further understand the results of ML models [12, 13, 36]. However, BOExplain is different from the cube-based approaches in two aspects. First, BOExplain can generate explanations from not only inference data but also training and source data. Second, BOExplain does not need to discretize numerical variables.

ML Pipeline Debugging. Rain [53] is designed to resolve a user’s complaint about the result of an inference query by removing a set of tuples that highly influence the query result. In contrast, BOExplain removes sets of tuples satisfying a predicate, which can be easier for a user to understand. In addition, BOExplain is more expressive, and supports UDFs, data science workflows, and pre-processing functions. Data X-Ray [51] focuses on explaining systematic errors in a data generative process. Other systems debug the configuration of a computational pipeline [3, 26, 30, 55].

Optimization Algorithms. Bayesian optimization (BO) is used to optimize expensive black box functions (see [11, 18, 29, 47] for overviews). BO consists of a surrogate model to estimate the expensive, derivative-free objective function, and an acquisition function to determine the next best point. The most common surrogate models are Gaussian processes [46] (GP) and tree-structured Parzen estimators [8, 10] (TPE). We selected TPE since it scales linearly in the size of the set of evaluated points, whereas a GP scales cubically [10]. Other surrogate models include random forests [23] and neural networks [49]. Expected improvement [46] is the most common acquisition function.

Hyperband [27] is a bandit-based approach for hyperparameter tuning that uses adaptive resource allocation and early-stopping to speed up random search. We did not choose Hyperband as the optimization approach since a time budget needs to be specified before running the algorithm (whereas TPE can run progressively), and our categorical variable optimizations in Section 4 are designed for a sequential optimization algorithm, which Hyperband is not.

Categorical Bayesian Optimization. Categorical variables in BO are often handled by one-hot encoding [17, 19, 20]. However, this approach does not scale well to variables with many distinct values [41]. BO may use tree-based surrogate models (e.g., random forests [23], TPE [10]) to handle categorical variables, however their
predictive accuracy is empirically poor [19, 34]. Other work optimizes a combinatorial search space [6, 15, 35], and categorical/category-specific continuous variables [34]. These works only consider categorical variables or focus on categorical variables with few distinct values, which is unsuitable for query explanation.

7 CONCLUSION

In this paper, we proposed BOExplain, a novel framework for explaining inference queries using BO. This framework treats the inference query along with an ML pipeline as a black-box which enables explanations to be derived from complex pipelines with UDFs. We considered predicates as explanations, and treated the predicate constraints as parameters to be tuned. TPE was used to tune the parameters, and we proposed a novel individual contribution encoding and warm start heuristic to improve the performance of categorical variables. We performed experiments showing that a) BOExplain can even outperform Scorpion and Macrobase for explaining SQL-only queries in certain situations, b) the proposed IC and warm start techniques were effective, c) BOExplain significantly outperformed random search for explaining inference queries, and d) BOExplain generated useful explanations for real corrupted datasets.

REFERENCES

[1] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Arvind Ananthanarayan, John Shue, Erik Meijer, Xi Wu, et al. 2020. DIF: a relational interface for large-scale data explanation. The VLDB Journal (2020), 1–26.
[2] Takuya Akiba, Shotaor San, Toshikiho Yanase, Takero Ohita, and Masatsugu Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2623–2631.
[3] Cyrille Artho. 2011. Iterative delta debugging. International Journal on Software Tools for Technology Transfer 13, 3 (2011), 223–246.
[4] Lucas Baier, Marcel Hofmann, Niklas Kühl, Marisa Mohr, and Gerhard Satzger. 2020. Handling Concept Drifts in Regression Problems—the Error Interaction Approach. arXiv preprint arXiv:2004.00438 (2020).
[5] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Gerhard Weikum. 2018. Apache Arrow: columnar storage for all. In Proceedings of the 2017 ACM International Conference on Management of Data. 541–556.
[6] Ricardo Baptista and Matthias Poloczek. 2018. Bayesian optimization of combinational structures. arXiv preprint arXiv:1806.08588 (2018).
[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. The Journal of Machine Learning Research 13, 1 (2012), 281–305.
[8] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search. arXiv preprint arXiv:1306.1466 (2013).
[9] James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In Proceedings of the 12th Python in scientific conference, Vol. 13. Citeseer, 28.
[10] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithmics for hyper-parameter optimization. In Advances in Neural Information Processing Systems. 2546–2554.
[11] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization for explaining cost-sensitive queries in database systems. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. 5254–5255.
[12] Bee-Chung Chen, Lei Chen, Yi Lin, and Raghu Ramakrishnan. 2005. Prediction algorithms for explaining SQL queries. In Proceedings of the 29th International Conference on Very Large Data Bases. 982–993.
[13] Yeonoh Chung, Tim Kraska, Neilolis Polyzos, Ki Hyun Tae, and Steven Euijong Whang. 2019. Slice finder: Automated data slicing for model validation. In Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data. 115–123.
[14] Sudeepa Roy, Laurel Orr, and Dan Suciu. 2015. Explaining Query Answers with Slicetrace. The VLDB Journal (2015), 1–26.
[15] Aryan Deshwal, Syrine Belakaria, and Janardhan Rao Doppa. 2020. Scalable bayesian optimization with gaussian processes. Neurcomputing (2020), 29–35.
[16] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google optimizer: A service for black-box optimization. In Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining. 1487–1495.
[17] Jim Grey, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murut Venkatraj, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data mining and knowledge discovery, 1, 1 (1997), 29–53.
[18] Tim Head, MecHoder, Gilles Louppe, Jaroslaw Scharber, Echazar, Zé Vincent, cromartie, Christopher Sharp, and David Storl. 2019. scikit-optimize: a powerful approach to reinforcement learning. arXiv preprint v6.5.2. https://doi.org/10.5281/zenodo.1207017
[19] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization using information-based acquisition functions. JMLR, 12 (May 2011), 1929–1957.
[20] Sudeepa Roy, Laurel Orr, and Dan Suciu. 2015. Explaining Query Answers with Slicetrace. The VLDB Journal (2015), 1–26.
[46] Matthias Schonlau, William J Welch, and Donald R Jones. 1998. Global versus local search in constrained optimization of computer models. Lecture Notes-Monograph Series (1998), 11–25.

[47] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. Proc. IEEE 104, 1 (2015), 148–175.

[48] Bernard W. Silverman. 1986. Density estimation for statistics and data analysis. Vol. 26. CRC press.

[49] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. 2015. Scalable bayesian optimization using deep neural networks. In International conference on machine learning. 2171–2180.

[50] Jasper Roland Snoek. 2013. Bayesian optimization and semiparametric models with applications to assistive technology. Ph.D. Dissertation. Citeseer.

[51] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data x-ray: A diagnostic tool for data errors. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 1231–1245.

[52] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. Proc. VLDB Endow. 6, 8 (June 2013), 553–564. https://doi.org/10.14778/2536354.2536356

[53] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jianman Wang. 2020. Complaint-driven Training Data Debugging for Query 2.0. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 1317–1334.

[54] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In Proceedings of the 2016 International Conference on Management of Data. 1599–1614.

[55] Jiaqi Zhang, Lakshminarayan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuan Yuan Zhou. 2014. Encore: Exploiting system environment and correlation information for misconfiguration detection. In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems. 687–700.
APPENDIX

A TREE-STRUCTURED PARZEN ESTIMATOR

The tree-structured Parzen estimator [8, 10] (TPE) is a sequential model-based optimization algorithm that uses Gaussian mixture models to approximate a black-box function \( f \) and the Expected Improvement [46] acquisition function to select the next sample. At a high level, TPE splits the evaluated points into two sets: good points and bad points (as determined by the objective function). It then creates two distributions, one for each set, and finds the next point to evaluate which has a high probability in the distribution over the good points and low probability in the distribution over the bad points. We next formally define the algorithm.

Initially, \( n_{\text{init}} \) samples are selected uniformly at random from the search space \( X \), and subsequently a model is used to guide the selection to the optimal location. TPE models each dimension of the search space independently using univariate Parzen window density estimation (or kernel density estimation) [48]. Assume for now that the search space is one-dimensional, i.e., \( X = [a, b] \subseteq \mathbb{R} \). Rather than model the posterior probability \( p(y \mid x) \) directly, TPE exploits Bayes’ rule, \( p(y \mid x) \propto p(x \mid y)p(y) \), and models the likelihood \( p(x \mid y) \) and the prior \( p(y) \). To model the likelihood \( p(x \mid y) \), the observations \( D_t = \{(x_i, y_i) = f(x_i)\}_{i=1}^t \) are first split into two sets, \( D_t^g \) and \( D_t^b \), based on their quality under \( f \): \( D_t^g \) contains the \( y \)-quantile highest quality points, and \( D_t^b \) contains the remaining points. Next, density functions \( g(x) \) and \( b(x) \) are created from the samples in \( D_t^g \) and \( D_t^b \) respectively. For each point \( x \in D_t^g \), a Gaussian distribution is fit with mean \( x \) and standard deviation set to the greater of the distances to its left and right neighbor. \( g(x) \) is a uniform mixture of these distributions. The same process is performed to create the distribution \( b(x) \) from the points in \( D_t^b \).

Formally, for a minimization problem, we have the likelihood

\[
p(x \mid y) = \begin{cases} 
g(x) & \text{if } y < y^* \\
b(x) & \text{if } y \geq y^* 
\end{cases}
\]

where \( y^* \) is the \( y \)-quantile of the observed values. The prior probability is \( p(y < y^*) = y \).

TPE uses the prior and likelihood models to derive the Expected Improvement [46] (EI) acquisition function. As the name suggests, EI involves computing how much improvement each point is expected to achieve over some threshold \( y^* \) by sampling a given point. Formally, EI under some model \( M \) of \( f \) is defined as

\[
EI_{y^*}(x) = \int_{-\infty}^{\infty} \max(y^* - y, 0) \rho_M(y \mid x) dy.
\]  

(1)

For TPE, it follows from Equation 1 that

\[
EI_{y^*}(x) \propto \left( y + \frac{b(x)}{g(x)}(1 - y) \right)^{-1}
\]  

(2)

the proof of which can be found in [10]. This means that a point with high probability in \( g(x) \) and low probability in \( b(x) \) will maximize the EI. To find the next point to evaluate, TPE samples \( n_{\text{EI}} \) candidate points from \( g(x) \). Each of these points is evaluated by \( g(x)/b(x) \), and the point with the highest value is suggested as the next point to be evaluated by \( f \).

For a \( d \)-dimensional search space, \( d > 1 \), TPE is performed independently for each dimension on each iteration. The full TPE algorithm is given in Algorithm 2.

---

**Algorithm 2: Tree-structured Parzen Estimator**

**Input:** \( f, X, n_{\text{init}}, n_{\text{iter}}, n_{\text{EI}}, y \)

**Output:** The best performing point found by TPE

1. **Initialize:** Select \( n_{\text{init}} \) points uniformly at random from \( X \), and create \( D_{n_{\text{init}}} = \{(x_i, f(x_i))\}_{i=1}^{n_{\text{init}}} \)

2. For \( t \leftarrow n_{\text{init}} \) to \( n_{\text{iter}} \)

3. Determine the \( y \)-quantile point, \( y^* \)

4. Split \( D_t \) into \( D_t^g \) and \( D_t^b \) based on \( y^* \)

5. For \( i \leftarrow 1 \) to \( d \)

6. Estimate \( g(x) \) on the \( i \)-th dimension of \( D_t^g \)

7. Estimate \( b(x) \) on the \( i \)-th dimension of \( D_t^b \)

8. Sample \( n_{\text{EI}} \) points from \( g(x) \)

9. Find the sampled point \( x_{i+1} \) with highest \( g(x)/b(x) \)

10. Update \( D_{t+1} \leftarrow D_t \cup \{(x_{i+1}, f(x_{i+1}))\} \)

11. Return \( (x, y) \in D_{n_{\text{iter}}} \) with the best objective function value

---

**Figure 8:** ML Pipeline for the Adult dataset. In this experiment, an explanation is derived from the training data.

---

**B BOEXPLAIN WITH SKEWED DATA**

In this section, we evaluate BOExplain in the presence of skewed data using the Adult Census Income dataset [16]. This dataset contains 32,561 rows and 15 human variables from a 1994 census, and we use a random forest classifier to predict whether a person makes over $50K a year. The ML pipeline is visualized in Figure 8, where the green box indicates that an explanation is derived from the training data. We split the data into 80% for training and 20% for inference. To define the ground truth corrupted data, we flipped the labels of training data tuples that satisfy the predicate \( 8 \leq \text{Education-Num} \leq 10 \) or \( 30 \leq \text{Age} \leq 40 \), which affects 16% of the training data. On the inference data, we query the average predicted value for the group Male. To assess whether BOExplain can accurately remove the corrupted data, we define the objective function to minimize the distance between the query result on the passed data and the query result if executed on the data after filtering out the corrupted tuples. We use the two numerical search variables Education-Num and Age which have domains [1, 16] and [17, 90], respectively, to derive an explanation.

The distribution of the search variable Education-Num is given in Figure 10, and as we can see, some values occur much more frequently than others. Next, we will investigate whether similar predicates define sets of tuples with similar size. Define a predicate \([w, x]\) to be similar to a predicate \([y, z]\) if \( w = y \) and \( x = z + 1 \), or \( w = y + 1 \) and \( x = z \). For each pair of similar predicates, we compute the absolute value of the difference between the number of tuples that satisfy each predicate. The distribution is plotted in Figure 11. As there is a large representation close to zero, similar predicates usually define a similar set of tuples. However, sometimes this is not
Figure 9: Adult: best objective function value, F-score, precision, and recall found at each 5 second increment, averaged over 10 runs. The goal is to minimize the objective function.

Figure 10: Distribution of Education-Num

Figure 11: Distribution of the difference in the size of sets of tuples defined by similar predicates

the case. For example, predicate $7 \leq \text{Education-Num} \leq 9$ satisfies 10,501 more tuples than predicate $7 \leq \text{Education-Num} \leq 8$.

We ran BOExplain and random search for 200 seconds, and the results are shown in Figure 9. BOExplain on average achieves an objective function result lower than Random before 45 seconds compared to Random’s result at 200 seconds. BOExplain also outperforms Random in terms of F-score and precision. These results show that BOExplain can still perform well in the presence of skewed data.