Explaining Inference Queries with Bayesian Optimization

by

Brandon Lockhart

B.Sc., University of Victoria, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2021

Declaration of Committee

Name:	**Brandon Lockhart**
Degree:	**Master of Science**
Thesis title:	**Explaining Inference Queries with Bayesian Optimization**
Committee:	**Chair:** Mo Chen Assistant Professor, Computing Science
	Jiannan Wang Supervisor Associate Professor, Computing Science
	Martin Ester Committee Member Professor, Computing Science
	Oliver Schulte Examiner Professor, Computing Science

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 work, 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 three real-world datasets.

Keywords: ML Explanation; SQL Explanation

Table of Contents

List of Tables

List of Figures

Chapter 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
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 for the unexpected result. 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 [49, 37]. This method is known as a *provenance* or *intervention*-based explanation [33].

Specifically, there are two types of explanations in the intervention-based setting: fine-grained (a set of tuples) and coarse-grained (a predicate) [32]. This work focuses on coarse-grained explanation. Predicates, unlike sets of tuples, provide a comprehensible explanation and identify common properties of the input tuples that cause the unexpected result. For the above example, it may return a predicate like `sex` = `'female' AND` $20 \leq$ `age` ≤ 25

	SQL Explanation	**Inference Query Explanation**	
	[49, 37, 36, 1]	**Rain [50]**	**BOExplain**
Inference Data	Supported	Supported	Supported
Training Data	Not Supported	Supported	Supported
Source Data	Not Supported	Not Supported	Supported
Explanation Type	Coarse-grained	Fine-grained	Coarse-grained
Methodology	White-box	White-box	Black-box

Table 1.1: Comparison of BOExplain and existing approaches.

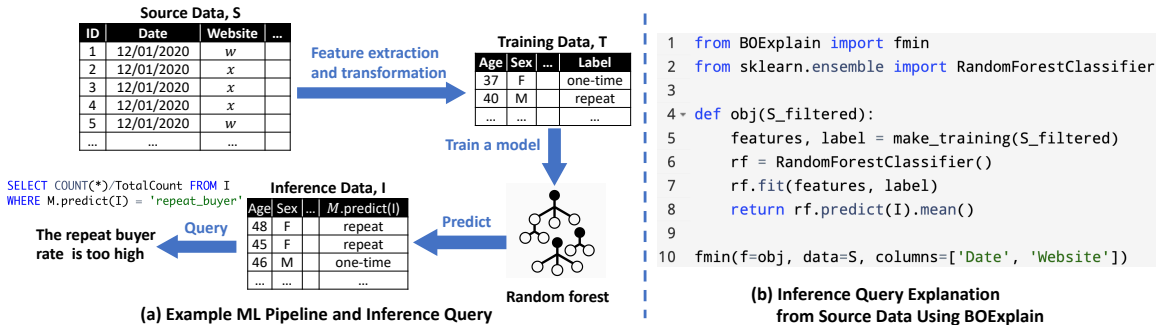

Figure 1.1: An illustration of using BOExplain to generate an explanation from source data in an ML pipeline.

which suggests that if the young female customers are removed from the inference data, the query result would look normal. Then, the data scientist can look into these customers more closely and conduct further investigation.

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

Example 1. *CompanyX creates an ML pipeline (Figure 1.1(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` $\in [t_1, t_2]$*, 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 ($t_1 \leq$* `Date` $\leq t_2$ `AND Website` $= w$*).*

Unfortunately, existing SQL explanation approaches [49, 37, 36, 1] are ill-equipped to address this setting (Table 1.1) 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 [50] 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 [23] 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 less amenable to data science programs (Figure 1.1(b)) that heavily incorporate UDFs.

As a first approach towards addressing the above limitations, and to diverge from existing white-box explanation approaches [49, 37, 36, 1, 50], this thesis explores a black-box approach towards inference query explanation. BOExplain models inference query explanation as a hyperparameter tuning problem and adopts Bayesian optimization (BO) to solve it. In ML, hyperparameters (e.g., the number of trees, learning rate) control the training process and are tuned in an "outer-loop" that surrounds the model training process. Hyperparameter tuning seeks to find the best hyperparameters that maximizes some model quality measure (e.g., validation score). BOExplain treats predicate constraints (e.g., t_1, t_2, w in Example 1) as hyperparameters, and the goal is to assign the optimal value to each constraint. By defining a metric that evaluates a candidate explanation's quality, (e.g., the decrease of the repeat buyer rate), BOExplain finds the constraint values that correspond to the highest quality predicate.

A black-box approach offers a number of advantages for inference query explanation. In terms of **usability**, a data scientist can derive a predicate from any data involved in an ML pipeline rather than inference data only. Furthermore, its concise API design is similar to popular hyperparameter tuning libraries, such as scikit-optimize [20] and Hyperopt [8], that many data scientists are already very familiar with. Figure 1.1(b) shows an example using BOExplain's API to solve Example 1. The data scientist wraps the portion of the program in an objective function obj whose input is the dataset to generate predicates for, and whose output is the repeat buyer rate that should be minimized. She also provides hints to focus on the Date and Website variables. See Section 4.2 for more details.

In terms of **adaptability**, a black-box approach can potentially be used to generate explanations for any data science workflow beyond inference queries. The current machine learning and analytics ecosystem is rapidly evolving. In contrast to white-box approaches, which must be carefully designed for a specific class of programs, BOExplain can more readily evolve with API, library, model, and ecosystem changes.

In terms of **effectiveness**, BOExplain builds on the considerable advances in BO by the ML community [43], to quickly generate high quality explanations. A secondary benefit is that BO is a progressive optimization algorithm, which lets BOExplain quickly propose an initial explanation, and improve it over time.

The key technical challenge is that existing BO approaches [10, 21, 46] cannot be naively adapted to explanation generation. In the hyperparameter tuning setting, categorical vari-

ables typically have very low cardinality (e.g., with 2-3 distinct values [34]). In the query explanation setting, however, a categorical variable can have many more distinct values. To address this, we propose a categorical encoding method to map a categorical variable into a numerical variable. This lets BOExplain estimate the quality of the categorical values that have not been evaluated. We further propose a warm start approach so that BOExplain can prioritize predicates with more promising categorical values.

In summary, this thesis makes the following contributions:

- We are the first to generate coarse-grained explanations from the training and source data to an inference query.
- We argue for a black-box approach to inference query explanation and discuss its advantages over a white-box approach.
- We propose BOExplain, a novel query explanation framework that derives explanations for inference queries using BO.
- We develop two techniques (categorical encoding and warm start) to improve BOExplain's performance on categorical variables.
- We show that BOExplain can generate comparable or higher quality explanations than state-of-the-art SQL explanation engines (Scorpion [49] and MacroBase [1]) on SQL-only queries.
- We evaluate BOExplain using inference queries on real-world datasets showing that BOExplain can generate explanations for different input datasets with a higher degree of explanation than random search.
- We implement BOExplain as a Python package and open-source it at `https://github.com/sfu-db/BOExplain`.

Thesis outline. First, the related work is discussed in Chapter 2. In Chapter 3, SQL explanation and the extension to inference query explanation is formally defined. Next, Bayesian optimization and the BOExplain framework is introduced in Chapter 4. In Chapter 5, we propose two techniques to improve BOExplain for categorical variables. Experimental results on synthetic and real-world datasets are given in Chapter 6. Finally, in Chapter 7, we conclude this work and propose possible future directions.

Chapter 2

Related Work

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

Query Explanation

BOExplain is most closely related to Scorpion [49] and the work of Roy and Suciu [37]. Both approaches define explanations as predicates. Scorpion uses a space partitioning and merging process to find the predicates, while Roy and Suciu [37] 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 DIFF operator [1], explanation-ready databases [36], and counterbalances [33]. Finally, a number of specialized systems focus on explaining specific scenarios, such as streaming data [4], map-reduce jobs [22], online transaction processing workloads [51], cloud services [38], and range-radius queries [40].

ML Pipeline Debugging

Rain [50] 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 [48] focuses on explaining systematic errors in a data generative process. Other systems debug the configuration of a computational pipeline [30, 24, 3, 52].

Bayesian Optimization

Bayesian optimization (BO) is used to optimize expensive black box functions (see [16, 43, 11, 29] 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 model is a Gaussian process (GP), but other models have been used, including random forests [21], neural networks [45], Student-t processes [42], and tree-structured Parzen estimators [10, 9]. Expected improvement (EI) [41] is the most common acquisition function; other functions include upper confidence bound [47] and probability of improvement [25].

Categorical Bayesian Optimization

A popular method for handling categorical variables in BO is to use one-hot encoding [18, 17, 15]. However, it does not scale well to variables with many distinct values [39]. BO may use tree-based surrogate models (e.g., random forests [21], tree Parzen estimators [10]) to handle categorical variables, however their predictive accuracy is empirically poor [17, 34]. Other work optimizes a combinatorial search space [5, 13, 35], and categorical/category-specific continuous variables [34]. These works consider only categorical variables or focus on categorical variables with a small number of distinct values, which is unsuitable for the query explanation setting.

Chapter 3

Problem Definition

In this chapter, we first define the SQL explanation problem, and subsequently describe the extension to inference query explanation.

3.1 Problem Definition in SQL Explanation

3.1.1 Query

We first define the supported queries. In this work, we focus on aggregation queries over a single table (the extension to multiple tables has been formalized in [37]). An *explainable query* is an arithmetic expression over a collection of SQL query results, as formally defined in Definition 1.

Definition 1 (Supported Queries)**.** *Given a relation R, an explainable query $Q = E(q_1, \ldots, q_k)$ is an arithmetic expression E over queries $q_1, \ldots, q_k$ of the form*

$$q_i = \textbf{SELECT } agg(\ldots) \textbf{ FROM } R \textbf{ WHERE } C_1 \textbf{ AND/OR } \ldots \textbf{ AND/OR } C_m$$

where agg is an aggregation operation and C_j is a filter condition.

Example 2. *Returning to the running example from Chapter 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*

$q_1 =$ *SELECT COUNT(*) FROM I WHERE M.predict(I)='repeat buyer'*
$q_2 =$ *SELECT COUNT(*) FROM I*

3.1.2 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 [37]. We use the notation $dir = low$ ($dir = high$) to indicate that Q is unexpectedly high (low).

Example 3. *In our running example, the user found the repeat buyer rate to be too high. Therefore, along with the query Q from Example 2, the user specifies $dir = low$ to indicate that Q should be lower.*

3.1.3 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*

$$p = l_1 \leq N_1 \leq u_1 \wedge \cdots \wedge l_n \leq N_n \leq u_n \wedge C_1 = c_1 \wedge \cdots \wedge C_m = c_m.$$

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

Example 4. *The source data in Figure 1.1 contains the variables Date and Website. An example explanation over these variables is*

$$12/01/2020 \leq Date \leq 12/10/2020 \wedge Website = w.$$

3.1.4 Objective Function

Next, we define the 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 $\sigma_{\neg p}(R)$ represent R filtered to contain all tuples that do not satisfy p. We apply the query to $\sigma_{\neg p}(R)$ and get the new query result. If the user specifies $dir = low$, 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 $obj(p, R, Q) \to \mathbb{R}$ applies Q on the relation $\sigma_{\neg p}(R)$.*

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 dir, and predicate space S, find the predicate*

$$p^* = \arg\min_{p \in S} obj(p, R, Q)$$

if $dir = low$ (use $\arg\max$ *if $dir = high$).*

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) or finding a trivial solution (e.g., returning only one record). 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 [49]. Since Q is an arithmetic expression over multiple queries, one of those queries may simply be the regularization term.

3.2 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[1]. The query processing pipeline is as follows (Figure 1.1(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 from the model predictions as well as the input data, and 2) in 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 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. Note that when R is the updated inference data $\mathcal{P}$ is a no-op , and the inference query explanation problem degrades to the SQL explanation problem. The extended objective function is defined in Definition 5.

[1]In general, any intermediate dataset is acceptable, however, we focus on these three due to their prevalence and for simplicity.

Definition 5 (Objective Function)**.** *Given a subset of an ML pipeline $\mathcal{P}$, a predicate p, relation R, and query Q, the objective function $obj(p, R, \mathcal{P}, Q) \to \mathbb{R}$ feeds $\sigma_{\neg p}(R)$ through $\mathcal{P}$, and then applies Q on the inference data.*

Finally, we define the inference query explanation problem.

Definition 6 (Inference Query Explanation Problem)**.** *Given a relation R, query Q, direction dir, pipeline $\mathcal{P}$, and predicate space S, find the predicate*

$$p^* = \arg\min_{p \in S} obj(p, R, Q, \mathcal{P})$$

if $dir = low$ (use $\arg\max$ *if $dir = high$).*

Chapter 4

The BOExplain Framework

This chapter introduces Bayesian optimization (BO), presents the BOExplain framework, and describes the API design of the Python implementation of BOExplain.

4.1 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 *sequential* means that BO is an iterative algorithm and *model-based* means that BO builds surrogate models to estimate the behavior of f. The term *Bayesian* in Bayesian optimization refers to the fact that a prior model of the objective function is used.

4.1.1 Tree-structured Parzen Estimator (TPE)

The tree-structured Parzen estimator [10, 9] (TPE) is a type of Bayesian optimization that uses kernel density estimation to approximate a black-box function f, and the Expected Improvement [41] 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. Figure 4.1 shows an example iteration of TPE. We next formally define the TPE algorithm.

Initially, n_{init} samples are selected uniformally at random from the search space $\mathcal{X}$, 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) [44]. Assume for now that the search space is one-dimensional, i.e., $\mathcal{X} = [a, b] \subset \mathbb{R}$. Rather than model the posterior probability $p(y \mid x)$ directly (where $y = f(x)$), TPE exploits Bayes' rule, $p(y \mid x) \propto p(x \mid y)p(y)$, and models

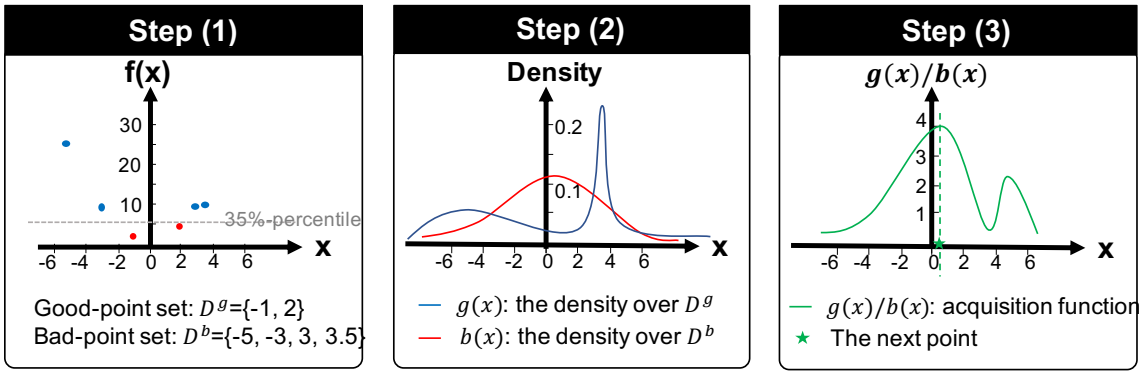

Figure 4.1: Suppose TPE has observed six points $D_6 = \{(-5, 25), (-3, 9), (-1, 1), (2, 4), (3, 9), (3.5, 9.25)\}$. This figure illustrates how TPE finds the next point to evaluate ($\gamma = 35\%$).

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

TPE uses the prior and likelihood models to derive the Expected Improvement [41] (EI) acquisition function. As the name suggests, EI involves computing how much improvement the objective function is expected to achieve over some threshold y^* by sampling a given point. Formally, EI under some model M of f is defined as

$$EI_{y^*}(x) = \int_{-\infty}^{\infty} \max\{y^* - y, 0\} p_M(y \mid x) dy. \tag{4.1}$$

For TPE, it follows from Equation 4.1 that

$$EI_{y^*}(x) \propto \left(\gamma + \frac{b(x)}{g(x)}(1 - \gamma)\right)^{-1} \tag{4.2}$$

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

Algorithm 1: Tree-structured Parzen Estimator

Input: $f, \mathcal{X}, n_{\text{init}}, n_{\text{iter}}, n_{EI}, \gamma$

Output: The best performing point found by TPE

Initialize: Select n_{init} points uniformally at random from $\mathcal{X}$, and create $D_{n_{\text{init}}} = \{(\mathbf{x}_i, f(\mathbf{x}_i))\}_{i=1}^{n_{\text{init}}}$

for $t \leftarrow n_{init}$ **to** n_{iter} **do**

  Determine the γ-quantile point, y^*

  Split D_t into D_t^g and D_t^b based on y^*

  for $i \leftarrow 1$ **to** d **do**

    Estimate $g(x)$ on the ith dimension of D_t^g

    Estimate $b(x)$ on the ith dimension of D_t^b

    Sample n_{EI} points from $g(x)$

    Find the sampled point x_{t+1}^i with highest $g(x)/b(x)$

  end

  Update $D_{t+1} \leftarrow D_t \cup \{(\mathbf{x}_{t+1}, f(\mathbf{x}_{t+1}))\}$

end

return $(\boldsymbol{x}, y) \in D_{n_{iter}}$ *with the best objective function value*

samples n_{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 1.

Categorical Variables

TPE models categorical variables by using categorical distributions rather than kernel density estimation. Consider a categorical variable with four distinct values: `Website` $\in \{w_1, w_2, w_3, w_4\}$. To build $g($`Website`$)$, TPE estimates the probability of w_i based on the fraction of its occurrences in D^g; the distribution is smoothed by adding 1 to the count of occurrences for each value. For instance, if the occurrences are 2, 0, 1, 0, then the distribution $g($`Website`$)$ will be $\{P(w_1), P(w_3), P(w_3), P(w_4)\} = \{3/7, 1/7, 2/7, 1/7\}$.

4.2 Our Framework

In this section, we describe the BOExplain framework.

4.2.1 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 $A_1, A_2, \ldots, A_n$.

If A_i is numerical (e.g., age, date), two parameters are created that serve as the bounds on the range constraint. Specifically, the parameters $A_{i_{\text{min}}}$ and $A_{i_{\text{length}}}$ define the lower bound

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

Table 4.1: Example inference data.

and the length of the range constraint, respectively. $A_{i_{\text{min}}}$ and $A_{i_{\text{length}}}$ 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 5. *Suppose the user defines a predicate space over* `State` *and* `Age` *in Table 4.1.* BOExplain *creates three parameters: one categorical parameter for* `State` *with domain {AZ, FL}, and two numerical parameters for* `Age` *with domains* $[40, 48]$ *and* $[0, 8]$*, respectively.*

4.2.2 BOExplain Framework

Figure 4.2 walks through the BOExplain framework. In step ⓪, the user provides an objective function *obj*, a relation S, and predicate variables $A_1, \ldots, A_n$ (Figure 1.1(b), line 10). Step ① creates the parameters and their domains. Step ② runs one iteration of TPE, starting with the parameters from step ①, and outputs a predicate. Steps ③ and ④ 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 1.1(b). Once it is executed, BOExplain first creates three parameters: $\texttt{Date}_{\text{min}}$, $\texttt{Date}_{\text{length}}$, and `Website` along with the corresponding domains. Then, it iteratively calls TPE to propose predicates (e.g., "$12/01/2020 \leq$ `Date` $\leq$ $12/02/2020$ `AND Website` $= w$"). BOExplain obtains S_filtered by removing the tuples that satisfy this predicate from S. Next, it applies `obj`$(\cdot)$ to S_filtered which will rerun the pipeline (Figure 1.1(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.

4.2.3 Why Is TPE Suitable For Query Explanation?

Recent work [6, 27, 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

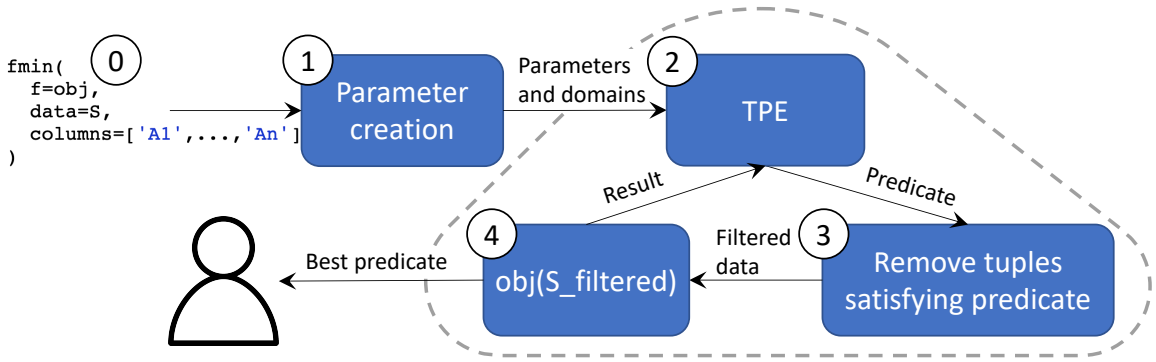

Figure 4.2: The BOExplain framework.

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-weigh similar values (e.g., $9, 16$) since the prior distribution will have lower weight in that region.

This property tends to hold in query explanation, because similar predicates tend to have similar objective values: similar predicates define similar sets of tuples, and removing similar sets of tuples from the dataset usually results in comparable 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` $\in [10, 21]$. If the former has a poor objective value, the latter two may be pruned.

Another consideration is that TPE (and BO in general) is generally used for computationally expensive black-box functions, which is not necessarily the case for inference queries. Even still, we find that the cost of updating the surrogate models and maximizing the EI is negligible compared to the cost of evaluating the inference query. Moreover, TPE is a derivative-free optimization algorithm, which is necessary for inference queries.

4.2.4 Other Implementations of BO

The most common implementation of BO is using Gaussian process (GP) [16] for the surrogate model and the expected improvement (EI) [41] acquisition function. There are two reasons we chose to use TPE rather than GP+EI: First, GPs scale cubically in $|D_t|$ and linearly in the number of variables, whereas TPE scales linearly in $|D_t|$ and linearly in the number of variables [10]. Second, GPs generally support categorical variables by a one-hot encoding method [17], which can be prohibitively expensive for high cardinality categorical columns.

Another approach is based on random forests which can handle numerical and categorical variables [21]. A random forest is trained on $D_t = \{(\mathbf{x}_i, y_i)\}_{i=1}^{t}$ to form the surrogate model. For the acquisition function, variants of the best performing previous configurations

and randomly selected configurations are evaluated by the surrogate model, and the configurations that perform the best are evaluated by the objective function. However, it has been shown that the predictive distribution of random forests is empirically poor [17, 34].

Ultimately, our interest is in applying BO for query explanation, and the specific implementation is not fundamental in our framework. TPE is used in the popular HT libraries Optuna [2] and Hyperopt [8, 7]. But other BO methods can be used for query explanation, and we leave the exploration of these to future work.

4.3 API Design

In this section, we detail the API design of our Python implementation of BOExplain, which is open-sourced at `https://github.com/sfu-db/BOExplain`. The design is inspired by hyperparameter tuning (HT) libraries. The API consists of two functions, `fmin` and `fmax`, which are respectively used to minimize and maximize the objective function. The user defines an objective function which is passed to `fmin` (or `fmax`) along with a pandas DataFrame, search variable names, and evaluation budget. A filtered DataFrame is returned from `fmin` (and `fmax`) with the tuples satisfying the best predicate found by BOExplain removed. The objective function takes a DataFrame as input and outputs a number.

Example 6. *We show how the BOExplain API can be used to derive explanations from the three data stages in the context of the running example. Figure 4.3b shows an objective function which constructs training data from the filtered source data, trains a random forest classifier, and returns the predicted repeat buyer rate. Since the data scientist found the repeat buyer rate to be unexpectedly high, she calls* `fmin` *and passes the objective function, source data S, variables Date and Website over which to produce a predicate, and the evaluation budget of* 100 *iterations of BO as parameters (Figure 4.3a).*

To derive an explanation from training or inference data, the corresponding subset of the ML pipeline simply needs to be put in the objective function. Figures 4.3c and 4.3d show the setup for deriving an explanation from training and inference data, respectively.

We will next compare BOExplain's API design with the design of a popular Python HT library scikit-optimize [20]. Figure 4.4 shows a simple HT example using scikit-optimize. Similar to `fmin`, the function `gp_minimize` takes an objective function as input which should be minimized. Although the design of BOExplain is inspired by HT libraries such as scikit-optimize, we made two changes to make it more user friendly for inference query explanation: (1) the objective function passed to `fmin` takes a DataFrame as input whereas the objective function passed to `gp_minimize` takes parameter assignments, (2) `fmin` takes a DataFrame and variable names as input whereas `gp_minimize` takes the domains of the parameters to be optimized. If BOExplain's objective function took a parameter assignment as input, the user would need to filter the DataFrame to consist of all tuples that do not

```
S_new = fmin(
  f=obj,
  data=S,
  columns=['Date', 'Website'],
  n_trials=100)
```

(a) Use the BOExplain function `fmin` to minimize the objective function.

```
def obj(S_filtered):
    features, labels = \
        make_training(S_filtered)
    rf = RandomForestClassifier()
    rf.fit(features, labels)
    return rf.predict(I).mean()
```

(b) Derive an explanation from source data, S.

```
T = make_training(S)
def obj(T_filtered):
    features = T_filtered[features]
    labels = T_filtered['label']
    rf = RandomForestClassifier()
    rf.fit(features, labels)
    return rf.predict(I).mean()
```

(c) Derive an explanation from training data, T.

```
features, labels = make_training(S)
rf = RandomForestClassifier()
rf.fit(features, labels)
I['pred'] = rf.predict(I)
def obj(I_filtered):
    return I_filtered['pred'].mean()
```

(d) Derive an explanation from inference data, I.

Figure 4.3: The BOExplain Python API.

satisfy the corresponding predicate inside the objective function, which could be difficult and error-prone. Also, we decided that BOExplain should define the parameters and compute their domains rather than the user since it is a well-defined process given the DataFrame variables (see Section 4.2.1).

```
from skopt import gp_minimize
def obj(x):
    return (x[0] + x[1]) ** 2
gp_minimize(func=obj, dimensions=[(0, 1), (0, 1)], n_calls=100)
```

Figure 4.4: An example of the scikit-optimize (skopt) API.

Chapter 5

Supporting Categorical Variables

In this chapter, we present our techniques to enable BOExplain to support categorical variables more effectively.

5.1 Individual Contribution Encoding

Recall from Section 4.1, TPE models numerical and categorical variables using kernel density estimation and categorical distributions, 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 $\text{domain}(C) = \{c_1, \ldots, c_n\}$. 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 7. *Suppose we would like an explanation from the inference data in Table 4.1. 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* $\rightarrow$ 1*, Artist* $\rightarrow$ 2*, Writer* $\rightarrow$ 3*.*

5.2 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 value, IC) as computed in Section 5.1, $S_{IC}^i = \{(c_j, IC(c_j))\}_{j=1}^{n_i}$, where n_i is the number of unique values in variable C_i. Next, we compute and materialize the d−ary Cartesian product and add the ICs for each combination $S_{IC} = S_{IC}^1 \times \cdots \times S_{IC}^d = \{((c_{i_1}, \ldots, c_{i_d}), IC(c_{i_1}) + \cdots + IC(c_{i_d})) \mid i_j \in \{1, \ldots, n_j\}\}$. The actual joint contribution is not computed because it is too computationally expensive: the number of invocations of the objective function is $O(m^c)$ for the joint contribution where m is the maximum number of distinct values across all categorical columns and c is the number of categorical columns, whereas our approach requires $O(mc)$ invocations. To see why adding the 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 $C_1 = c_1 \wedge C_2 = c_2$ 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 8. *The IC for values in the variable Occupation were computed in Example 7,* $S_{ic}^{Occupation} = \{(Athlete, 0.5), (Artist, 0.75), (Writer, 1)\}$*, and for the variable Sex we have* $S_{ic}^{Sex} = \{(F, 0.5), (M, 1)\}$*. Next, we compute the combined IC score for each combination of predicates:* $S_{IC} = \{((Athlete, F), 1), \ldots, ((Writer, M), 2)\}$*. Recall, 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 iteration of BO we evaluate the predicates* `Occupation` = *"Athlete"* $\wedge$ `Sex` = *"F" and* `Occupation` = *"Artist"* $\wedge$ `Sex` = *"F" respectively. Note that* `Occupation` = *"Athlete"* $\wedge$ `Sex` = *"F" is the best predicate, so adding IC scores can be useful at prioritizing good explanations.*

Algorithm 2: BOExplain

Input: Objective function obj, data S, variables $A_1, \ldots, A_n$
Output: A predicate and the corresponding objective value

1 **foreach** *categorical variable* C **do**
2 | Compute the IC of all unique values in C
3 **end**
4 **Create** the parameters and domains
5 Compute the predicted high quality combinations based on IC for the warm start
6 **Initialize TPE:** Perform n_{init} iterations using a warm start to create $D_{n_{\text{init}}} = \{(\mathbf{x}_i, obj(\sigma_{\neg \mathbf{x}_i}(S))\}_{i=1}^{n_{\text{init}}}$.
7 **for** $t \leftarrow n_{init}$ **to** n_{iter} **do**
8 | Split D_t into D_t^g and D_t^b based on the splitting threshold γ
9 | **for** $i \leftarrow 1$ **to** d **do**
10 | | Estimate $g(x)$ on the ith dimension of D_t^g
11 | | Estimate $b(x)$ on the ith dimension of D_t^b
12 | | Sample n_{EI} points from $g(x)$
13 | | Find the sample x_{t+1} with the highest $g(x)/b(x)$
14 | **end**
15 | Update $D_{t+1} \leftarrow D_t \cup \{(\mathbf{x}_{t+1}, obj(\sigma_{\neg \mathbf{x}_{t+1}}(S)))\}$
16 **end**
17 **return** $(\boldsymbol{x}, obj(\sigma_{\neg \boldsymbol{x}}(S))) \in D_{n_{iter}}$ *with the best objective value*

5.3 Putting Everything Together

We lastly present the full BOExplain algorithm in Algorithm 2. 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 sets based on γ. Next, from line 9, a value is selected for each parameter. In lines 10 and 11, distributions of the good and bad sets 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.

Chapter 6

Experiments

In the previous chapters, we presented a novel approach for explaining SQL and inference queries. In this chapter, we aim to experimentally evaluate the proposed approach. Specifically, we are interested in answering 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?

6.1 Experimental Settings

Baselines

For SQL-only queries, we compare BOExplain with the explanation engines Scorpion [49] and MacroBase [1, 4] which return predicates as explanations. For inference queries, no predicate-based explanation engines exist, so we compare with a random search baseline [6].

Scorpion [49] is a framework for explaining group-by aggregate queries. The authors define a function to measure the quality of a predicate, which can be implemented as BOExplain's objective function. Each continuous variable's domain is split into 15 equi-sized ranges as set in the original paper. We use the author's open-source code[1] to run the Scorpion experiments.

MacroBase [4] (later, the DIFF SQL operator [1]) is an explanation engine that considers combinations of variable-values pairs, similar to a `CUBE` query [19], as candidate explanations. The utility of each explanation is determined by one or more *difference metrics* each having a utility threshold; explanations that satisfy the utility threshold are outputted. In

[1] `https://github.com/sirrice/scorpion`

Section 2.3 in [1], the authors describe how to use the DIFF operator for Scorpion's influence metric. We implemented it in the author's open-source code[2]. The user needs to define how numerical variables are discretized; we tuned the bin size from 2 to 15 and report the best result.

In [1], MacroBase was shown to outperform other explanation engines including Data X-ray [48] and Roy and Suciu [36], and so we do not compare with these approaches.

Random search is a competitive method for hyperparameter tuning [6]. The parameters are chosen independently and uniformly at random for numerical and categorical variables from the domains described in Section 4.2.

Real-world Datasets and ML Pipelines

The following lists the three datasets and ML pipelines used in our experiments. The pipelines can be visualized in Figure 6.1, and we put a green box around the data where an explanation is derived in each pipeline. In Adult and House, an explanation is derived from training data, and in Credit an explanation is derived from source data.

Adult income dataset [14]. This dataset contains 32,561 rows and 15 human variables from a 1994 census. Figure 6.1(a) shows the pipeline where the data is prepared for modelling, and a random forest classifier is then used to predict whether a person makes over $50K a year. We split the data into 80% for training and 20% for inference.

House price prediction [12]. 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 6.1(b).

Credit card approval prediction[3]. The source data consists of two tables: `application_record` (438,557 rows, 18 variables), which contains information about previous applicants, and `credit_record` (1,048,575 rows, 3 variables), which contains information about the credit history of the applicants. The pipeline to prepare the data for modelling is given in Figure 6.1(c), 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.

Metrics

To measure the quality of an explanation, we plot the best objective function value achieved by each time point t. The systems are run for a fixed number of seconds rather than a fixed number of iterations since random search does not have the overhead of updating the

[2] `https://github.com/stanford-futuredata/macrobase`

[3] `https://www.kaggle.com/rikdifos/credit-card-approval-prediction`

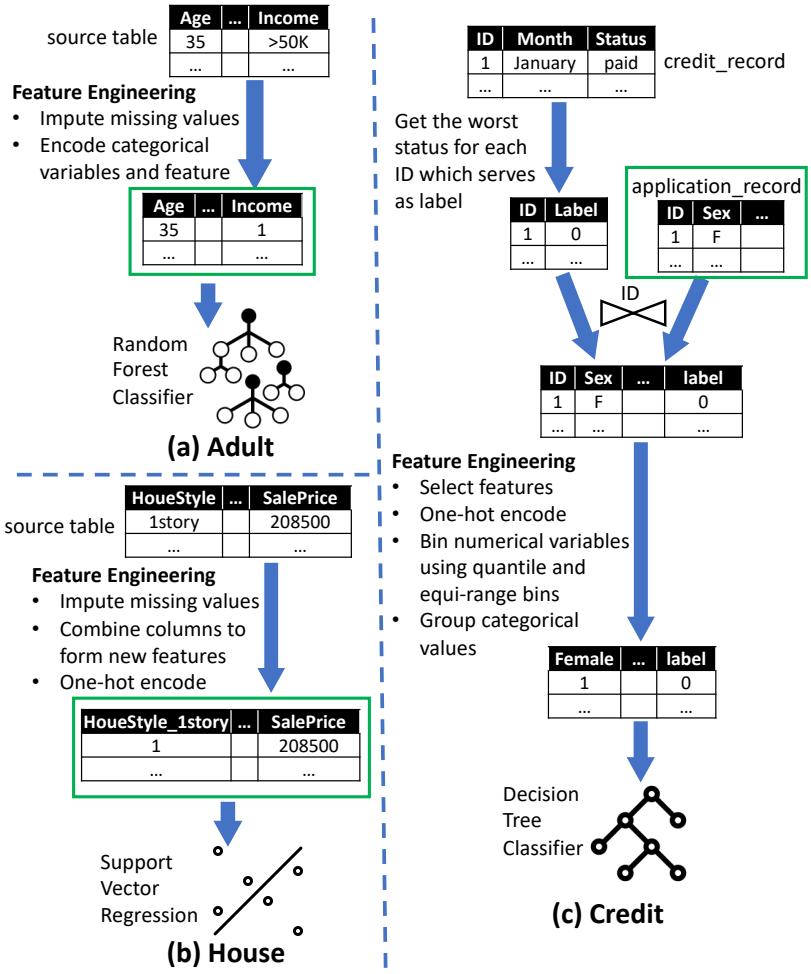

Figure 6.1: ML Pipelines for Adult, House, and Credit. The green box indicates where to generate an explanation.

models. The length of the run is chosen to be long enough for the systems to converge. 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 on the real-world datasets. We synthetically corrupt data defined by a predicate, and use that data as ground truth. Precision is the number of selected corrupted 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 and Random, each result is averaged over 10 runs.

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 6.3 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 $\gamma = 0.1$ for all experiments.

6.2 Explaining SQL-Only Queries

We compare BOExplain with Scorpion and MacroBase using the synthetic data and corresponding query from Scorpion's paper [49]. The dataset consists of a single *group by* variable A_d, an aggregate variable A_v, and search variables $A_1, \ldots, A_n$ with $\text{domain}(A_i) = [0, 100] \subset \mathbb{R}$, $i \in [n]$. A_d 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_v value in a holdout group is drawn from $\mathcal{N}(10, 10)$. 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 $A_v \sim \mathcal{N}(\mu, 10)$, and the outer cube contains 25% of the tuples in the group and $A_v \sim \mathcal{N}(\frac{\mu+10}{2}, 10)$, else $A_v \sim \mathcal{N}(10, 10)$. μ 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_v) FROM synthetic GROUP BY A_d. The arithmetic expression over the SQL query is defined in Section 3 of [49] 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 [49]. We used $n = 2$ and $n = 3$ since 3 is the maximum number of variables supported by MacroBase.

The results are shown in Figure 6.2. 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

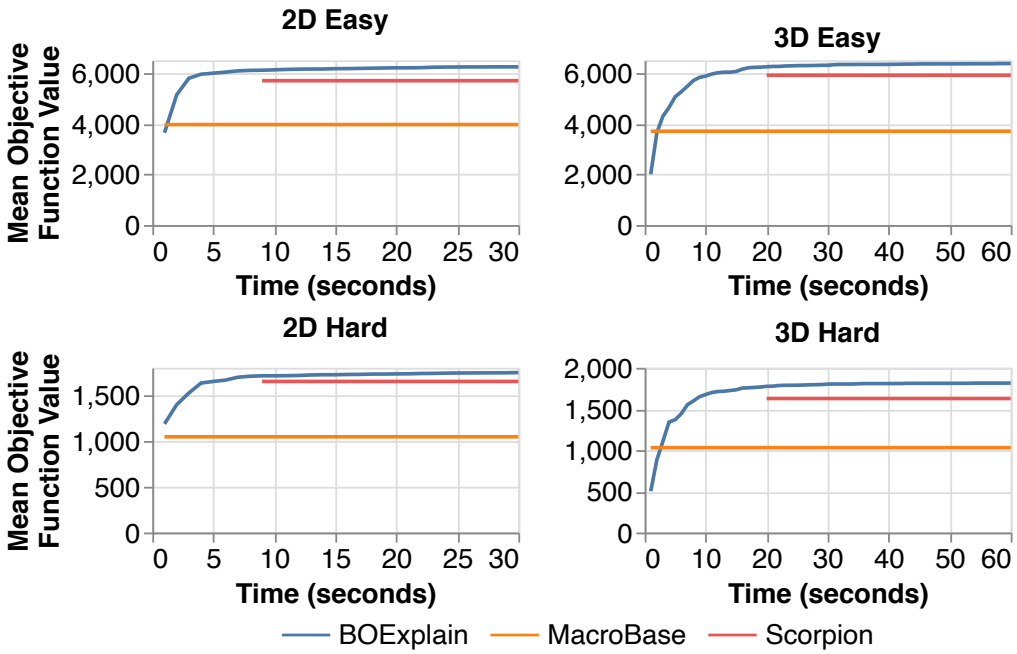

Figure 6.2: Performance comparison with Scorpion and MacroBase. The goal is to maximize the objective function.

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 work 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.

6.3 Explaining Inference Queries

We evaluate BOExplain's performance of explaining inference queries in various settings. We start with an experiment on Adult where an explanation is derived from training data (Section 6.3.1), then we investigate BOExplain's approach for categorical variables on House (Section 6.3.2), and finally we evaluate BOExplain in a complex ML pipeline on Credit, where an explanation is derived from source data (Section 6.3.3).

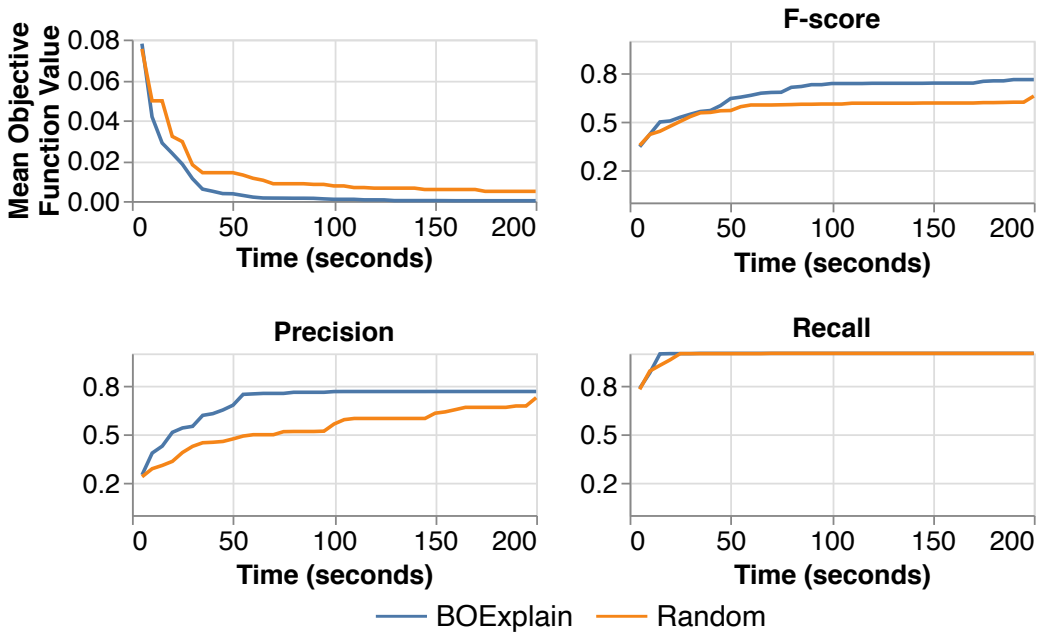

Figure 6.3: Adult: best objective function value, F-score, precision, and recall found at each 5 second increment, averaged over 10 runs. The goal is to minimize the objective function.

6.3.1 Explanation From Training Data

In this experiment, we create an error region by flipping the training labels on Adult where $8 \leq$ `Education-Num` $\leq 10 \wedge 30 \leq$ `Age` ≤ 40 which affects 16% of the training data. On the inference data, we query the average predicted value for the group Male. To asses whether BOExplain can accurately remove the corrupted data, we define the objective function to compute 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, and then define that the objective function should be minimized. We use the two numerical search variables Education-Num and Age which have domains $[1, 16]$ and $[17, 90]$, respectively, and the size of the search space is 1.4×10^6.

Each method is run for 200 seconds, and the results are shown in Figure 6.3. BOExplain on average achieves an objective function result lower than Random before 45 seconds compared to Random's result at 200 seconds. This shows that it is effective for BOExplain to exploit promising regions, whereas random search that just explores the space cannot find a good predicate as quickly. BOExplain also outperforms Random in terms of F-score and precision. The recall is high for both approaches since it is likely that a predicate is produced with large ranges that cover all of the corrupted tuples. On average, BOExplain (Random) completed 348.9 (295.6) iterations. BOExplain performed more iterations because as it exploited the promising region, it removed more training data than a random predicate, and so the model took less time to retrain.

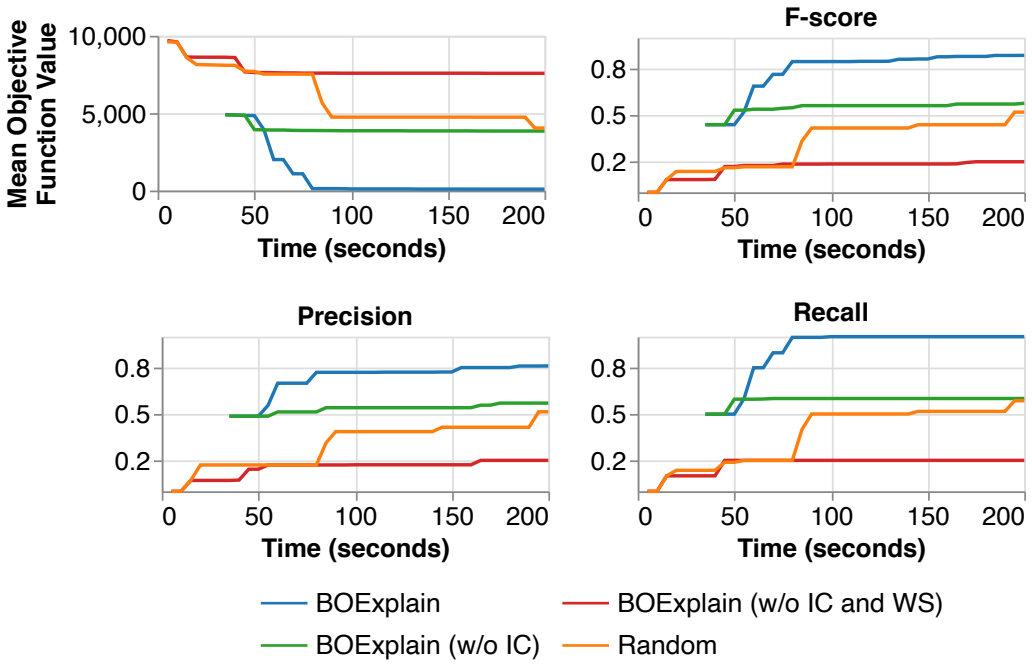

Figure 6.4: House: best objective function value, F-score, precision, and, and recall, found at each 5 second increment averaged over 10 runs. The goal is to minimize the objective function. (IC = Individual Contribution Encoding, WS = Warm Start)

6.3.2 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'` $\wedge$ `Exterior1st='VinylSd'` $\wedge$ $2000 \leq$ `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 Chapter 5. 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 5.2, 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 6.4. 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.

6.3.3 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 $-23000 \leq$ `DAYS_BIRTH` $\leq -17000 \wedge 2 \leq$ `CNT_FAM_MEMBERS` ≤ 3 to have a "bad" credit status, which affects 20.1% of the data. Corrupting the data 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 6.5. 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.

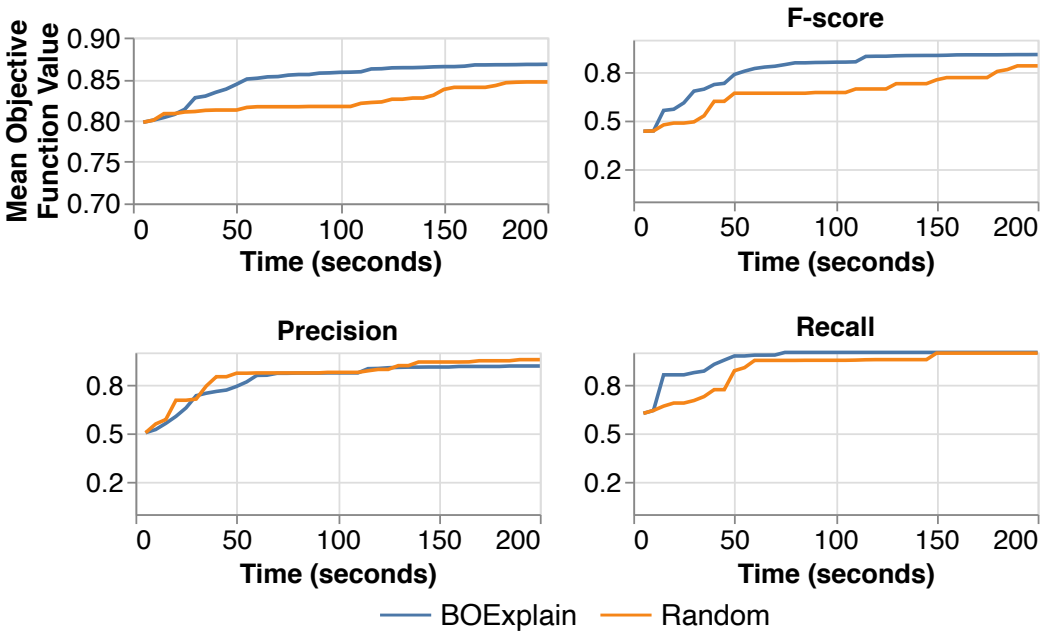

Figure 6.5: Credit: best objective function value, F-score, precision, and recall found at each 5 second increment, averaged over 10 runs. The goal is to maximize the objective function; larger values are better.

Chapter 7

Conclusion and Future Work

In this thesis, a novel framework for explaining inference queries using Bayesian optimization, called BOExplain, was proposed. 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 i) BOExplain can even outperform Scorpion and Macrobase for explaining SQL-only queries in certain situations, ii) the proposed IC and warm start techniques were effective, iii) BOExplain significantly outperformed random search for explaining inference queries.

Future Directions. There are several future directions for this work. First, we would like to make BOExplain support a richer set of explanations. For example, extending the equality constraint for categorical variables to a set containment clause would help to identify multiple categorical values that contribute to the unexpected query result. This extension would be difficult since it involves going from a constraint having c possibilities (c is the number of distinct values in the column) to 2^c possibilities, and determining a good relationship or notion of similarity between sets is nontrivial. For numerical variables, it would be useful to support explanations with multiple range constraints over the same variable. Also, currently BOExplain requires the user to specify the exact variables over which to derive an explanation; extending BOExplain to derive an explanation from a subset of variables would free the user from having to know the variables that can provide an explanation beforehand. A conditional hyperparameter optimization solution [26] could be applied where a parameter selects which variables appear in the predicate, and conditioned on the output of this parameter, the variable values are then selected.

It would be useful to explore further optimizations to BOExplain for categorical variables that have many distinct values. The cost of the individual contribution encoding — which requires evaluating the objective function for every distinct categorical value — may be prohibitively expensive. One direction is to use a data sample so that evaluating the objective

function requires less time. Alternatively, it may be possible in certain situations to consider only frequently occurring categorical values that are likely to make a significant impact on the objective function.

Finally, a future direction is to explore other optimization strategies such as Bayesian optimization using Gaussian Processes [46] or random forests [21], or a multi-armed bandit approach like HyperBand [28]. It would be particularly useful to model potential relationships between parameters, such as between the parameters for the lower and upper bounds on a range constraint, that TPE does not model.

Bibliography

[1] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, et al. Diff: a relational interface for large-scale data explanation. *The VLDB Journal*, pages 1–26, 2020.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[3] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.

[4] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556, 2017.

[5] Ricardo Baptista and Matthias Poloczek. Bayesian optimization of combinatorial structures. *arXiv preprint arXiv:1806.08838*, 2018.

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[7] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[8] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, volume 13, page 20. Citeseer, 2013.

[9] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123, 2013.

[10] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[11] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[12] Dean De Cock. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3), 2011.

[13] Aryan Deshwal, Syrine Belakaria, and Janardhan Rao Doppa. Scalable combinatorial bayesian optimization with tractable statistical models. *arXiv preprint arXiv:2008.08177*, 2020.

[14] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[15] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[16] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[17] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing*, 380:20–35, 2020.

[18] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[19] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[20] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. scikit-optimize/scikit-optimize: v0.5.2, March 2018.

[21] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[22] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Perfxplain: debugging mapreduce job performance. *arXiv preprint arXiv:1203.6400*, 2012.

[23] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, pages 1885–1894. PMLR, 2017.

[24] Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. Cadet: A systematic method for debugging misconfigurations using counterfactual reasoning. *arXiv preprint arXiv:2010.06061*, 2020.

[25] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. 1964.

[26] Julien-Charles Lévesque, Audrey Durand, Christian Gagné, and Robert Sabourin. Bayesian optimization for conditional hyperparameter spaces. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 286–293. IEEE, 2017.

[27] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR, 2020.

[28] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[29] Daniel James Lizotte. *Practical bayesian optimization*. University of Alberta, 2008.

[30] Raoni Lourenço, Juliana Freire, and Dennis Shasha. Bugdoc: A system for debugging computational pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2733–2736, 2020.

[31] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[32] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. Causality and explanations in databases. *Proc. VLDB Endow.*, 7(13):1715–1716, August 2014.

[33] Zhengjie Miao, Qitian Zeng, Boris Glavic, and Sudeepa Roy. Going beyond provenance: Explaining query answers with pattern-based counterbalances. In *Proceedings of the 2019 International Conference on Management of Data*, pages 485–502, 2019.

[34] Dang Nguyen, Sunil Gupta, Santu Rana, Alistair Shilton, and Svetha Venkatesh. Bayesian optimization for categorical and category-specific continuous inputs. In *AAAI*, pages 5256–5263, 2020.

[35] Changyong Oh, Jakub Tomczak, Efstratios Gavves, and Max Welling. Combinatorial bayesian optimization using the graph cartesian product. In *Advances in Neural Information Processing Systems*, pages 2914–2924, 2019.

[36] Sudeepa Roy, Laurel Orr, and Dan Suciu. Explaining query answers with explanation-ready databases. *Proc. VLDB Endow.*, 9(4):348–359, December 2015.

[37] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1579–1590, New York, NY, USA, 2014. Association for Computing Machinery.

[38] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1167–1178. IEEE, 2015.

[39] Binxin Ru, Ahsan Alvi, Vu Nguyen, Michael A Osborne, and Stephen Roberts. Bayesian optimisation over multiple continuous and categorical inputs. In *International Conference on Machine Learning*, pages 8276–8285. PMLR, 2020.

[40] Fotis Savva, Christos Anagnostopoulos, and Peter Triantafillou. Explaining aggregates for exploratory analytics. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 478–487. IEEE, 2018.

[41] Matthias Schonlau, William J Welch, and Donald R Jones. Global versus local search in constrained optimization of computer models. *Lecture Notes-Monograph Series*, pages 11–25, 1998.

[42] Amar Shah, Andrew Wilson, and Zoubin Ghahramani. Student-t processes as alternatives to gaussian processes. In *Artificial intelligence and statistics*, pages 877–885, 2014.

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

[44] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

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

[46] Jasper Roland Snoek. *Bayesian optimization and semiparametric models with applications to assistive technology*. PhD thesis, Citeseer, 2013.

[47] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

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

[49] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.

[50] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1317–1334, 2020.

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

[52] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 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*, pages 687–700, 2014.