Supporting User-Defined Functions on Uncertain Data

Thanh T. L. Tran[†], Yanlei Diao[†], Charles Sutton[‡], Anna Liu[†]
[†]University of Massachusetts, Amherst [‡]University of Edinburgh
{ttran,yanlei}@cs.umass.edu csutton@inf.ed.ac.uk anna@math.umass.edu

ABSTRACT

Uncertain data management has become crucial in many sensing and scientific applications. As user-defined functions (UDFs) become widely used in these applications, an important task is to capture result uncertainty for queries that evaluate UDFs on uncertain data. In this work, we provide a general framework for supporting UDFs on uncertain data. Specifically, we propose a learning approach based on Gaussian processes (GPs) to compute approximate output distributions of a UDF when evaluated on uncertain input, with guaranteed error bounds. We also devise an online algorithm to compute such output distributions, which employs a suite of optimizations to improve accuracy and performance. Our evaluation using both real-world and synthetic functions shows that our proposed GP approach can outperform the state-of-the-art sampling approach with up to two orders of magnitude improvement for a variety of UDFs.

1. INTRODUCTION

Uncertain data management has become crucial in many applications including sensor networks [9], object tracking and monitoring [22], severe weather monitoring [13], and digital sky surveys [2]. Data uncertainty arises due to a variety of reasons such as measurement errors, incomplete observations, and using inference to recover missing information [22]. When such data is processed via queries, its uncertainty propagates to processed results. The ability to capture the result uncertainty is important to the end user for interpreting the derived information appropriately. For example, knowing only the mean of a distribution for the result cannot distinguish between a sure event and a highly uncertain event, which may result in wrong decision making; knowing more about the distribution can help the user avoid such misunderstanding and ill-informed actions.

Recent work on uncertain data management has studied intensively relational query processing on uncertain data (e.g., [4, 7, 16, 19, 20, 23]). Our work, however, is motivated by the observation that real-world applications, such as scientific computing and financial analysis, make intensive use of *user-defined functions (UDFs)* that process and analyze the data using complex, domain-specific algorithms. In practice, UDFs can be provided in any form of external code, e.g., C programs, and hence are treated mainly as *black boxes* in traditional databases. These UDFs are often expensive to compute due to the complexity of processing. Unfortunately, the support for UDFs on uncertain data is largely lacking in today's data management systems. Consequently, in the tornado detection application [13], detection errors cannot be distinguished from true events due to the lack of associated confidence scores. In other applications such as computational astrophysics [2], the burden of characterizing UDF result uncertainty is imposed on the programmers: we observed that the programmers of the Sloan digital sky surveys manually code algorithms to keep track of uncertainty in a number of UDFs. These observations have motivated us to provide system support to automatically capture result uncertainty of UDFs, hence freeing users from the burden of doing so and returning valuable information for interpreting query results appropriately.

More concretely, let us consider two examples of UDFs in the Sloan digital sky surveys (SDSS) [2]. In SDSS, nightly observations of stars and galaxies are inherently noisy as the objects can be too dim to be recognized in a single image. However, repeated observations allow the scientists to model the position, brightness, and color of objects using continuous distributions, which are commonly Gaussian distributions. Assume the processed data is represented as $(objID, pos^p, redshift^p, ...)$ where pos and $redshift$ are uncertain attributes. Then, queries can be issued to detect features or properties of the objects. We consider some example UDFs from an astrophysics package [1]. Query Q1 below computes the age of each galaxy given its redshift using the UDF $GalAge$. Since $redshift$ is uncertain, the output $GalAge(redshift)$ is also uncertain.

```
Q1:  Select G.objID, GalAge(G.redshift)
     From Galaxy G
```

A more complex example of using UDFs is shown in query Q2, which computes the comoving volume of two galaxies whose distance is in some specific range. This query invokes two UDFs $ComoveVol$ and $Distance$ on uncertain attributes $redshift$ and pos respectively, together with a selection predicate on the output of the UDF $Distance$.

```
Q2:  Select G1.objID, G2.objID, Distance(G1.pos, G2.pos)
            ComoveVol(G1.redshift, G2.redshift, AREA)
     From Galaxy AS G1, Galaxy AS G2
     Where Distance(G1.pos, G2.pos) ∈ [l, u]
```

Problem Statement. In this work, we aim to provide a general framework to support UDFs on uncertain data, where the functions are given as black boxes. Specifically, given an input tuple modeled by a vector of random variables $\mathbf{X}$, which is characterized by a joint distribution (either continuous or discrete), and a univariate, black-box UDF f, our objective is to characterize the distribution of $Y = f(\mathbf{X})$. In the example of Q2, after the join between G_1 and G_2, each tuple carries a random vector, $\mathbf{X} = \{G_1.pos, G_1.redshift, G_2.pos, G_2.redshift, \ldots\}$, and

Proceedings of the VLDB Endowment, Vol. 6, No. 6

two UDFs produce $Y_1 = Distance(G_1.pos, G_2.pos)$ and $Y_2 = ComoveVol(G_1.redshift, G_2.redshift, \text{AREA})$.

Given the nature of our UDFs, exact derivation of result distributions may not be feasible, and hence approximation techniques will be explored. A related requirement is that the proposed solution must be able to meet user-specified accuracy goals. In addition, the proposed solution must be able to perform efficiently in an online fashion, for example, to support online interactive analysis over a large data set or data processing on real-time streams (e.g., to detect tornados or anomalies in sky surveys).

Challenges. Supporting UDFs as stated above poses a number of challenges: (1) UDFs are often computationally expensive. For such UDFs, any processing that incurs repeated function evaluation to compute the output will take a long time to complete. (2) When an input tuple has uncertain values, computing a UDF on them will produce a result with uncertainty, which is characterized by a distribution. Computing the result distribution, even when the function is known, is a non-trivial problem. Existing work in statistical machine learning (surveyed in [5]) uses regression to estimate a function, but mostly focuses on deterministic input. For uncertain input, existing work [12] computes only the mean and variance of the result, instead of the full distribution, and hence is of limited use if this distribution is not Gaussian (which is often the case). Other work [15] computes approximate result distributions without bounding approximation errors, thus not addressing user accuracy requirements. (3) Further, most of our target applications require using an online algorithm to characterize result uncertainty of a UDF, where "online" means that the algorithm does not need an offline training phase before processing data. Relevant machine learning techniques such as [12, 15] belong to offline algorithms. In addition, a desirable online algorithm should operate with high performance in order to support online interactive analysis or data stream processing.

Contributions. In this paper, we present a complete framework for handling user-defined functions on uncertain data. Specifically, our main contributions include:

1. An approximate evaluation framework (§2): We propose a carefully-crafted approximation framework for computing UDFs on uncertain data, including approximation metrics and objectives. These metrics, namely discrepancy and KS measures, are a natural fit of range queries and intuitive to interpret. While many approximation metrics exist in the statistics literature, our choices of the metrics and objectives combined allow us to provide new theoretical results regarding the error bounds of output distributions.

2. Computing output distributions with error bounds (§3 and §4): We employ an approach of modeling black-box UDFs using a machine learning technique called *Gaussian processes* (GPs). We choose this technique due to its abilities to model functions and quantify the approximation in such function modeling.

Given the GP model of a UDF and uncertain input, our contribution lies in computing output distributions with *error bounds*. In particular, we provide an algorithm that combines the GP model of a UDF and Monte Carlo (MC) sampling to compute output distributions. We perform an in-depth analysis of the algorithm and derive new theoretical results for quantifying the approximation of the output, including bounding the errors of both approximation of the UDF and sampling from input distributions. These error bounds can be used to tune our model to meet accuracy requirements. To the best of our knowledge, this work is the first to quantify output distributions of Gaussian processes.

3. An optimized online algorithm (§5): We further propose an *online* algorithm to compute approximate output distributions that satisfy user accuracy requirements. Our algorithm employs a suite of optimizations of the GP learning and inference modules to improve performance and accuracy. Specifically, we propose *local inference* to increase inference speed while maintaining high accuracy, *online tuning* to refine function modeling and adapt to input data, and an *online retraining* strategy to minimize the training overhead. Existing work in machine learning [12, 15, 14, 17] does not provide a sufficient solution to such high-performance online training and inference while meeting user-specified accuracy requirements.

4. Evaluation (§6): We conduct a thorough evaluation of our proposed techniques using both synthetic functions with controlled properties, and real functions from the astrophysics domain. Results show that our GP techniques can adapt to various function complexities, data characteristics, and user accuracy goals. Compared to MC sampling, our approach starts to outperform when function evaluation takes more than 1ms for low-dimensional functions, e.g., up to 2 dimensions, or when function evaluation takes more than 100ms for high-dimensional ones, e.g., 10 dimensions. This result applies to real-world expensive functions as we show using the real UDFs from astrophysics. For the UDFs tested, the GP approach can offer up to two orders of magnitude speedup over MC sampling.

2. AN APPROXIMATION FRAMEWORK

In this section, we first propose a general approximate evaluation framework, and then present a baseline approach based on Monte Carlo sampling to compute output distributions of UDFs.

2.1 Approximation Metrics and Objectives

Since UDFs are given as *black boxes* and have no explicit formula, computing the output of the UDFs can be done only through function evaluation. For uncertain input, computing the exact distribution requires function evaluation at all possible input values, which is impossible when the input is continuous. In this work, we seek approximation algorithms to compute the output distribution given uncertain input. We now present our approximation framework including accuracy metrics and objectives.

We adopt two distance metrics between random variables from the statistics literature [11]: the discrepancy and Kolmogorov–Smirnov (KS) measures. We choose these metrics because they are a natural fit of range queries, hence allowing easy interpretation of the output.

Definition 1 Discrepancy measure. *The discrepancy measure, $\mathcal{D}$, between two random variables Y and Y' is defined as:*

$$\mathcal{D}(Y, Y') = \sup_{a,b:a\le b} |\Pr[Y \in [a,b]] - \Pr[Y' \in [a,b]]|.$$

Definition 2 KS measure. *The KS measure (or distance) between two random variables Y and Y' is defined as:*

$$KS(Y, Y') = \sup_y |\Pr[Y \le y] - \Pr[Y' \le y]|.$$

The values of both measures are in $[0, 1]$. It is straightforward to show that $\mathcal{D}(Y, Y') \le 2KS(Y, Y')$. Both measures can be computed directly from the cumulative distribution functions (CDFs) of Y and Y' to capture their maximum difference. The KS distance considers all one-sided intervals, i.e., $[-\infty, c]$ or $[c, \infty]$, while the discrepancy measure considers all two-sided intervals $[a, b]$.

In practice, users may be interested only in intervals of length at least λ, an application-specific error level that is tolerable for the computed quantity. This suggests a relaxed variant of the discrepancy measure, as follows.

Definition 3 λ-discrepancy. *Given the minimum interval length λ, the discrepancy measure $\mathcal{D}_\lambda$ between two random variables Y and Y' is:*

$$\mathcal{D}_\lambda(Y, Y') = \sup_{a,b:b-a\ge\lambda} |\Pr[Y \in [a,b]] - \Pr[Y' \in [a,b]]|.$$

This measure can be interpreted as: for all intervals of length at least λ, the probability of an interval under Y' does not differ from that

under Y by more than $\mathcal{D}_\lambda$. These distance metrics can be used to indicate how well one random variable Y' approximates another random variable Y. We next state the our approximation objective, (ϵ, δ)-approximation, using the discrepancy metric; similar definitions hold for the λ-discrepancy and the KS metric.

Definition 4 (ϵ, δ)**-approximation.** *Let Y and Y' be two random variables. Then Y' is an (ϵ, δ)-approximation of Y iff with probability $(1-\delta)$, $\mathcal{D}(Y, Y') \leq \epsilon$.*

For query Q1, (ϵ, δ)-approximation requires that with probability $(1-\delta)$, the approximate distribution of `GalAge(G.redshift)` does not differ from the true one more than ϵ in discrepancy. For Q2, there is a selection predicate in the WHERE clause, which truncates the distribution of `Distance(G1.pos, G2.pos)` to the region $[l, u]$, and hence yields a tuple existence probability (TEP). Then, (ϵ, δ)-approximation requires that with probability $(1-\delta)$, (i) the approximate distribution of `Distance(G1.pos, G2.pos)` differs from the true distribution by at most ϵ in discrepancy measure, and (ii) the result TEP differs from the true TEP by at most ϵ.

2.2 A Baseline Approach

We now present a simple, standard technique to compute the query results based on Monte Carlo (MC) simulation. However, as we will see, this approach may require evaluating the UDF many times, which is inefficient for slow UDFs. This inefficiency is the motivation for our new approach presented in Sections 3–5.

A. Computing the Output Distribution. In recent work [23], we use Monte Carlo simulation to compute the output distribution of aggregates on uncertain input. This technique can also be used to compute any UDF $Y = f(\mathbf{X})$. The idea is simple: draw the samples from the input distribution, and perform function evaluation to get the output samples. The algorithm is as follows.

Algorithm 1 Monte Carlo simulation

1: Draw m samples $\mathbf{x}_1 \ldots \mathbf{x}_m \sim p(\mathbf{x})$.
2: Compute the output samples, $y_1 = f(\mathbf{x}_1)$, ..., $y_m = f(\mathbf{x}_m)$.
3: Return the empirical CDF of the output samples, namely Y', $\Pr(Y' \leq y) = \frac{1}{m} \sum_{i \in [1..m]} \mathbb{1}_{[y_i, \infty)}(y)$.

$\mathbb{1}(\cdot)$ is the indicator function. It is shown in [23] that if $m = \ln(2\delta^{-1})/(2\epsilon^2)$, then the output Y' is an (ϵ, δ)-approximation of Y in terms of KS measure, and $(2\epsilon, \delta)$-approximate in terms of discrepancy measure. Thus, the number of samples required to reach the accuracy requirement ϵ is proportional to $1/\epsilon^2$, which is large for small ϵ. For example, if we use the discrepancy measure and set $\epsilon = 0.02$, $\delta = 0.05$, then m required is more than 18000.

B. Filtering with Selection Predicates. In many applications, users are interested in the event that the output is in certain intervals. This can be expressed with a selection predicate, e.g., $f(\mathbf{X}) \in [a, b]$, as shown in query Q2. When the probability $\rho = \Pr[f(\mathbf{X}) \in [a, b]]$ is smaller than a user-specified threshold θ, this corresponds to an event of little interest and can be discarded. For high performance, we would like to quickly check whether $\rho < \theta$ for filtering, which in turn saves the cost from computing the full distribution $f(\mathbf{X})$.

While drawing the samples as in Algorithm 1, we derive a confidence interval for ρ to decide whether to filter. By definition we have $\rho = \int \mathbb{1}(a \leq f(\mathbf{x}) \leq b) p(\mathbf{x}) d\mathbf{x}$. Let $h(\mathbf{x}) = \mathbb{1}(a \leq f(\mathbf{x}) \leq b)$ and $\tilde{m}$ be the number of samples drawn so far ($\tilde{m} \leq m$). And let $\{h_i | i = 1 \ldots \tilde{m}\}$ be the samples evaluated on $h(\mathbf{x})$. Then, h_i are iid, Bernoulli samples, and ρ can be estimated by $\tilde{\rho}$, computed from the samples, $\tilde{\rho} = \frac{\sum_{i=1}^{\tilde{m}} h_i}{\tilde{m}}$. The following result, which can be derived from the *Hoeffding's inequality* in statistics, gives a confidence interval for ρ.

Remark 2.1 *With probability $(1-\delta)$, $\rho \in [\tilde{\rho} - \tilde{\epsilon}, \tilde{\rho} + \tilde{\epsilon}]$, where* $\tilde{\epsilon} = \sqrt{\frac{1}{2\tilde{m}} \ln \frac{2}{1-\delta}}$.

If the user specifies a threshold θ to filter low-probability events, and $\tilde{\rho} + \tilde{\epsilon} < \theta$, then we can drop this tuple from output.

3. EMULATING UDFS WITH GAUSSIAN PROCESSES

In the next three sections, we present an approach that aims to be more efficient than MC sampling by requiring many fewer calls to the UDF. The main idea is that every time we call the UDF, we gain information about the function. Once we have called the UDF enough times, we ought to be able to approximate it by *interpolating* between the known values to predict the UDF at unknown values. We call this predictor an *emulator* $\hat{f}$, which can be used in place of the original UDF f, and is much less expensive for many UDFs.

We briefly mention how to build the emulator using a statistical learning approach. The idea is that, if we have a set of function input-output pairs, we can use it as training data to estimate f. In principle, we could build the emulator using any regression procedure from statistics or machine learning, but picking a simple method like linear regression would work poorly on a UDF that did not meet the strong assumptions of that method. Instead, we build the emulator using a learning approach called *Gaussian processes (GPs)*. GPs have two key advantages. First, GPs are *flexible* methods that can represent a wide range of functions and do not make strong assumptions about the form of f. Second, GPs produce not only a prediction $\hat{f}(\mathbf{x})$ for any point $\mathbf{x}$ but also a *probabilistic confidence* that provides "error bars" on the prediction. This is vital because we can use this to adapt the training data to meet the user-specified error tolerance. Building an emulator using a GP is a standard technique in the statistics literature; see [15] for an overview.

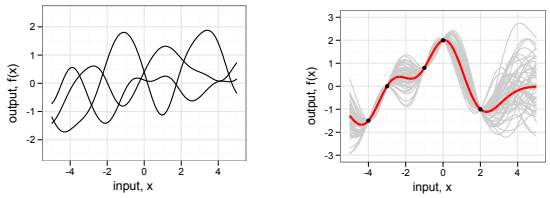
In this section, we provide background on the basic approach to building emulators. In Section 4, we extend to uncertain inputs and aim to quantify the uncertainty of outputs of UDFs. We then propose an online algorithm to compute UDFs and various optimizations to address accuracy and performance requirements in Section 5.

3.1 Intuition for GPs

We give a quick introduction to the use of GPs as emulators, closely following the textbook [18]. A GP is a distribution over functions; whenever we sample from a GP, we get an entire function for f whose output is the real line. Fig. 1(a) illustrates this in one dimension. It shows three samples from a GP, where each is a function $\mathbb{R} \to \mathbb{R}$. Specifically, if we pick any input $\mathbf{x}$, then $f(\mathbf{x})$ is a scalar random variable. This lets us get confidence estimates, because once we have a scalar random variable, we can get a confidence interval in the standard way, e.g., mean $\pm$ 2*standard_deviation. To use this idea for regression, notice that since f is random, we can also define conditional distributions over f, in particular, conditional distribution of f given a set of training points. This new distribution over functions is called the *posterior distribution*, and it is this distribution that lets us predict new values.

3.2 Definition of GPs

Just as the multivariate Gaussian is an analytically tractable distribution over vectors, the Gaussian process is an analytically tractable distribution over functions. Just as a multivariate Gaussian is defined by a mean and covariance matrix, a GP is defined by a *mean function* and a *covariance function*. The mean function $m(\mathbf{x})$ gives the average value $\mathbb{E}[f(\mathbf{x})]$ for all inputs $\mathbf{x}$, where the expectation is taken over the random function f. The covariance function $k(\mathbf{x}, \mathbf{x}')$



(a) Prior (b) Posterior

Figure 1: Example of GP regression. (a) prior functions, (b) posterior functions conditioning on training data

returns the covariance between the function values at two input points, i.e., $k(\mathbf{x}, \mathbf{x}') = \text{Cov}(f(\mathbf{x}), f(\mathbf{x}'))$.

A GP is a distribution over functions with a special property: if we fix any vector of inputs $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, the output vector $\mathbf{f} = (f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_n))$ has a multivariate Gaussian distribution. Specifically, $\mathbf{f} \sim \mathcal{N}(\mathbf{m}, K)$, where $\mathbf{m}$ is the vector $(m(\mathbf{x}_1) \ldots m(\mathbf{x}_n))$ containing the mean function evaluated at all the inputs and K is a matrix of covariances $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ between all the input pairs.

The covariance function has a vital role. Recall that the idea was to approximate f by interpolating between its values at nearby points. The covariance function helps determine which points are "nearby". If two points are far away, then their function values should be only weakly related, i.e., their covariance should be near 0. On the other hand, if two points are nearby, then their covariance should be large in magnitude. We accomplish this by using a covariance function that depends on the distance between the input points.

In this work, we use standard choices for the mean and covariance functions. We choose the mean function $m(\mathbf{x}) = 0$, which is a standard choice when we have no prior information about the UDF. For the covariance function, we use the *squared exponential* one, which in its simplest form is $k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 e^{-\frac{1}{2l^2}\|\mathbf{x}-\mathbf{x}'\|^2}$, where $\|\cdot\|$ is Euclidean distance, and σ_f^2 and l are its parameters. The signal variance σ_f^2 primarily determines the variance of the function value at individual points, i.e., $\mathbf{x} = \mathbf{x}'$. More important is the lengthscale l, which determines how rapidly the covariance decays as $\mathbf{x}$ and $\mathbf{x}'$ move farther apart. If l is small, the covariance decays rapidly, so sample functions from the result GP will have many small bumps; if l is large, then these functions will tend to be smoother.

The key assumption made by GP modeling is that at any point x, the function value $f(x)$ can be accurately predicted using the function values at nearby points. GPs are flexible to model different types of functions by using an appropriate covariance function [18]. For instance, for smooth functions, squared-exponential covariance functions work well; for less smooth functions, Matern covariance functions work well (where smoothness is defined by "mean-squared differentiability"). In this paper, we focus on the common squared-exponential functions, which are shown experimentally to work well for the UDFs in our applications (see §6.4). In general, the user can choose a suitable covariance function based on the well-defined properties of UDFs, and plug it into our framework.

3.3 Inference for New Input Points

We next describe how to use a GP to predict the function outputs at new inputs. Denote the training data by $X^* = \{\mathbf{x}_i^* | i = 1, \ldots, n\}$ for the inputs and $\mathbf{f}^* = \{f_i^* | i = 1, \ldots, n\}$ for the function values. In this section, we assume that we are told a fixed set of m test inputs $X = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m)$ at which we wish to predict the function values. Denote the unknown function values at the test points by $\mathbf{f} = (f_1, f_2, ..., f_m)$. The vector $(\mathbf{f}^*, \mathbf{f})$ is a random vector because each $f_{i:i=1\ldots m}$ is random, and by the definition of a GP, this vector simply has a multivariate Gaussian distribution. This distribution is:

$$\begin{bmatrix} \mathbf{f}^* \\ \mathbf{f} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X^*, X^*) & K(X^*, X) \\ K(X, X^*) & K(X, X) \end{bmatrix}\right), \quad (1)$$

where we have written the covariances as matrix with four blocks. The block $K(X^*, X)$ is an $n \times m$ matrix of the covariances between all training and test points, i.e., $K(X^*, X)_{ij} = k(\mathbf{x}_i^*, \mathbf{x}_j)$. Similar notions are for $K(X^*, X^*)$, $K(X, X)$, and $K(X, X^*)$.

Now that we have a joint distribution, we can predict the unknown test outputs $\mathbf{f}$ by computing the conditional distribution of $\mathbf{f}$ given the training data and test inputs. Applying the standard formula for the conditional of a multivariate Gaussian yields:

$$\mathbf{f} \,|\, X, X^*, \mathbf{f}^* \sim \mathcal{N}(\mathbf{m}, \Sigma), \text{where} \quad (2)$$
$$\mathbf{m} = K(X, X^*)K(X^*, X^*)^{-1}\mathbf{f}^*$$
$$\Sigma = K(X, X) - K(X, X^*)K(X^*, X^*)^{-1}K(X^*, X)$$

To interpret $\mathbf{m}$ intuitively, imagine that $m = 1$, i.e., we wish to predict only one output. Then $K(X, X^*)K(X^*, X^*)^{-1}$ is an n-dimensional vector, and the mean $\mathbf{m}(\mathbf{x})$ is the dot product of this vector with the training values $\mathbf{f}^*$. So $\mathbf{m}(\mathbf{x})$ is simply a weighted average of the function values at the training points. A similar intuition holds when there is more than one test point, $m > 1$. Fig. 1(b) illustrates the resulting GP after conditioning on training data. As observed, the posterior functions pass through the training points marked by the black dots. The sampled functions also show that the further a point is from the training points, the larger the variance is.

We now consider the complexity of this inference step. Note that once the training data is collected, the inverse covariance matrix $K(X^*, X^*)^{-1}$ can be computed once, with a cost of $O(n^3)$. Then given a test point $\mathbf{x}$ (or X has size 1), inference involves computing $K(X, X^*)$ and multiplying matrices, which has a cost of $O(n^2)$. The space complexity is also $O(n^2)$, for storing these matrices.

3.4 Learning the Hyperparameters

Typically, the covariance functions have some free parameters, called *hyperparameters*, such as the lengthscale l of the squared-exponential function. The hyperparameters determine how quickly the confidence estimates expand as test points move further from the training data. For example, in Fig. 1(b), if the lengthscale decreases, the spread of the function will increase, meaning that there is less confidence in the predictions.

We can learn the hyperparameters using the training data (see Chapter 5, [18]). We adopt maximum likelihood estimation (MLE), a standard technique for this problem. Let $\boldsymbol{\theta}$ be the vector of hyperparameters. The log likelihood function is $\mathcal{L}(\boldsymbol{\theta}) := \log p(\mathbf{f}^* | X^*, \boldsymbol{\theta}) = \log \mathcal{N}(X^*; \mathbf{m}, \Sigma)$; here we use $\mathcal{N}$ to refer to the density of the Gaussian distribution, and $\mathbf{m}$ and Σ are defined in Eq. (2). MLE solves for the value of $\boldsymbol{\theta}$ that maximizes $\mathcal{L}(\boldsymbol{\theta})$. We use gradient descent, a standard method for this task. Its complexity is $O(n^3)$ due to the cost of inverting the matrix $K(X^*, X^*)^{-1}$. Gradient descent requires many steps to compute the optimal $\boldsymbol{\theta}$; thus, retraining often has a high cost for large numbers of training points. Note that when the training data X^* changes, $\boldsymbol{\theta}$ that maximizes the log likelihood $\mathcal{L}(\boldsymbol{\theta})$ may also change. Thus, one would need to maximize the log likelihood to update the hyperparameters. In §5.3, we will discuss retraining strategies that aim to reduce this computation cost.

4. UNCERTAINTY IN QUERY RESULTS

So far in our discussions of GPs, we have assumed that all the input values are known in advance. However, our work aims to compute UDFs on uncertain input. In this section, we describe how

f, $\hat{f}$, $\tilde{f}$	true function, mean function of the GP, and a sample function of the GP, respectively.
f_L, f_S	upper and lower envelope functions of $\tilde{f}$ (with high probability)
Y, $\hat{Y}$, $\tilde{Y}$	output corresponding to f, $\hat{f}$, $\tilde{f}$, respectively.
Y_L, Y_S	output corresponding to f_L, f_S, respectively.
$\hat{Y}'$	estimate of $\hat{Y}$ using MC sampling. (Similarly for Y'_L and Y'_S)
$\tilde{\rho}$, $\hat{\rho}$	probability of $\tilde{Y}$ and $\hat{Y}$, in a given interval $[a, b]$.
ρ_U, ρ_L	upper and lower bounds of ρ (with high prob.).
$\tilde{\rho}'$, $\hat{\rho}'$, ρ'_U, ρ'_L	MC estimates of $\tilde{\rho}$, $\hat{\rho}$, ρ_U and ρ_L respectively.
n	number of training points.
m	number of MC samples.

Table 1: The main notation used in GP techniques.

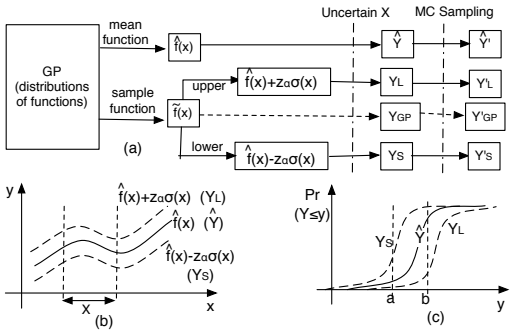


Figure 2: GP inference for uncertain input. (a) Computation steps (b) Approximate function with bounding envelope (c) Computing probability for interval [a, b] from CDFs

to compute output distributions using a GP emulator given uncertain input. We then derive theoretical results to bound the errors of the output using our accuracy metrics.

4.1 Computing the Output Distribution

We first describe how to approximate the UDF output $Y = f(\mathbf{X})$ given uncertain input $\mathbf{X}$. When we approximate f by the GP emulator $\hat{f}$, we have a new approximate output $\hat{Y} = \hat{f}(\mathbf{X})$, having CDF, $\Pr[\hat{Y} \leq y] = \int \mathbb{1}(\hat{f}(\mathbf{x}) \leq y)p(\mathbf{x})d\mathbf{x}$. This integral cannot be computed analytically. Instead, a simple, offline algorithm is to use Monte Carlo integration by repeatedly sampling input values from $p(\mathbf{x})$. This is very similar to Algorithm 1, except that we call the emulator $\hat{f}$ rather than the UDF f, which is a cheaper operation for long-running UDFs. The algorithm is detailed as below.

Algorithm 2 Offline algorithm using Gaussian processes

1: Collect n training data points, $\{(\mathbf{x}_i^*, y_i^*), i = 1..n\}$ by evaluating $y_i^* = f(\mathbf{x}^*)$
2: Learning a GP via training using the n training data points, to get $\mathcal{GP} \sim (\hat{f}(\cdot), k(\cdot,\cdot))$.
3: For uncertain input, $\mathbf{X} \sim p(\mathbf{x})$:
4: Draw m samples, $\mathbf{x}_1, ..., \mathbf{x}_m$, from the distribution $p(\mathbf{x})$.
5: Predict function values at the samples via GP inference to get $\{(\hat{f}(\mathbf{x}_i), \sigma^2(\mathbf{x}_i)), i = 1..m\}$
6: Construct the empirical CDF of $\hat{Y}$ from the samples, namely $\hat{Y}'$, $\Pr(\hat{Y}' \leq y) = \frac{1}{m}\sum_{i\in[1..m]} \mathbb{1}_{[\hat{f}_i,\infty)}(y)$, and return $\hat{Y}'$.

In addition to returning the CDF of $\hat{Y}'$, we also want to return a confidence of how close $\hat{Y}'$ is to the true answer Y. Ideally, we would do this by returning the discrepancy metric, $\mathcal{D}(\hat{Y}', Y)$. But it is difficult to evaluate $\mathcal{D}(\hat{Y}', Y)$ without many calls to the UDF f, which would defeat the purpose of using emulators. So instead we ask a different question, which is feasible to analyze. The GP defines a posterior distribution over functions, and we are using the posterior mean as the best emulator. The question we ask is *how different would the query output be if we emulated the UDF using a random function from the GP, rather than the posterior mean?* If this difference is small, this means the GP's posterior distribution is very concentrated. In other words, the uncertainty in the GP modeling is small, and we do not need more training data.

To make this precise, let $\tilde{f}$ be a sample from the GP posterior distribution over functions, and define $\tilde{Y} = \tilde{f}(\mathbf{X})$ (see Fig. 2a for an illustration for these variables). That is, $\tilde{Y}$ represents the query ouput if we select the emulator randomly from the GP posterior distribution. The confidence estimate that we will return will be an upper bound on $\mathcal{D}(\hat{Y}', \tilde{Y})$.

4.2 Error Bounds Using Discrepancy Measure

We now derive a bound on the discrepancy $\mathcal{D}(\hat{Y}', \tilde{Y})$. An important point to note is that there are two sources of error here. The first is the *error due to Monte Carlo sampling of the input* and the second is the *error due to the GP modeling*. In the analysis that follows, we bound each source of error individually and then combine them to get a single error bound. To the best of our knowledge, this is the first work to quantify the output distributions of GPs.

The main idea is that we will compute a *high probability envelope* over the GP prediction. That is, we will find two functions f_L and f_S such that $f_S \leq \tilde{f} \leq f_L$ with probability at least $(1-\alpha)$, for a given α. Once we have this envelope on $\tilde{f}$, then we also have a high probability envelope of $\tilde{Y}$, and can use this to bound the discrepancy. Fig. 2 (parts b & c) gives an illustration of this intuition.

Bounding Error for One Interval. To start, assume that we have already computed a high probability envelope. Since the discrepancy involves a supremum over intervals, we start by presenting upper and lower bounds on $\tilde{\rho} := \Pr[\tilde{Y} \in [a, b] \,|\, \tilde{f}]$ for a single fixed interval $[a, b]$. Now, $\tilde{\rho}$ is random because $\tilde{f}$ is; for every different function $\tilde{f}$ we get from the GP posterior, we get a different $\tilde{\rho}$.

For any envelope (f_S, f_L), e.g., having the form $\hat{f}(\mathbf{x}) \pm z\sigma(\mathbf{x})$ as shown in Fig. 2, define $Y_S = f_S(\mathbf{X})$ and $Y_L = f_L(\mathbf{X})$. We bound $\tilde{\rho}$ (with high probability) using Y_S and Y_L. For any two functions g and h, and any random vector $\mathbf{X}$, it is always true that $g \leq h$ implies that $\Pr[g(\mathbf{X}) \leq a] \geq \Pr[h(\mathbf{X}) \leq a]$ for all a. Putting this together with $f_S \leq \tilde{f} \leq f_L$, we have that

$$\tilde{\rho} = \Pr[\tilde{f}(\mathbf{X}) \leq b] - \Pr[\tilde{f}(\mathbf{X}) \leq a] \leq \Pr[f_S(\mathbf{X}) \leq b] - \Pr[f_L(\mathbf{X}) \leq a]$$

In other words, this gives the upper bound:

$$\tilde{\rho} \leq \rho_U := \Pr[Y_S \leq b] - \Pr[Y_L \leq a] \tag{3}$$

Similarly, we can derive the lower bound:

$$\tilde{\rho} \geq \rho_L := \max(0, \Pr[Y_L \leq b] - \Pr[Y_S \leq a]) \tag{4}$$

This is summarized in the following result.

Proposition 4.1 *Suppose that f_S and f_L are two functions such that $f_S \leq \tilde{f} \leq f_L$ with probability $(1-\alpha)$. Then $\rho_L \leq \tilde{\rho} \leq \rho_U$, with probability $(1-\alpha)$, where ρ_U and ρ_L are as in Eqs. 3 and 4.*

Bounding λ-discrepancy. Now that we have the error bound for one individual interval, we use this to bound the λ-discrepancy $\mathcal{D}_\lambda(\tilde{Y}, \hat{Y})$. Using the bounds of $\tilde{\rho}$, we can write this discrepancy as

$$\mathcal{D}_\lambda(\tilde{Y}, \hat{Y}) = \sup_{[a,b]} |\tilde{\rho} - \hat{\rho}| \leq \sup_{[a,b]} \max\{|\rho_L - \hat{\rho}|, |\rho_U - \hat{\rho}|\},$$

where the inequality applies the result from Proposition 4.1. This is progress, but we cannot compute ρ_L, ρ_U, or $\hat{\rho}$ exactly because they

Algorithm 3 Compute λ-discrepancy error bound

1: Construct the empirical CDFs, $\hat{Y}'$, Y'_S and Y'_L, from the output samples. Let $\mathcal{V}$ be the set of values of these variables.
2: Precompute $\max_{b\geq b_0}(\Pr[\hat{Y}' \leq b] - \Pr[Y'_L \leq b])$ and $\max_{b\geq b_0}(\Pr[Y'_S \leq b] - \Pr[\hat{Y}' \leq b])$ $\forall b_0 \in \mathcal{V}$.
3: Consider values for a, s.t. $[a, a+\lambda]$ lies in the support of $\hat{Y}'$. a is in $\mathcal{V}$, enumerated from small to large.
4: For a given a:
(a) Get $\Pr[\hat{Y}' \leq a]$, $\Pr[Y'_S \leq a]$, and $\Pr[Y'_L \leq a]$.
(b) Get $\max_{b\geq a+\lambda}(\Pr[Y'_S \leq b] - \Pr[\hat{Y}' \leq b])$.
Find smallest b_1 s.t. $\Pr[Y'_L \leq b_1] \leq \Pr[Y'_S \leq a]$, and then get $\max_{b\geq b_1}(\Pr[\hat{Y}' \leq b] - \Pr[Y'_L \leq b])$. This is done by using the precomputed values in Step 2.
(c) Compute $\max(\rho'_U - \hat{\rho}', \hat{\rho}' - \rho'_L)$ from the quantities in (a) and (b). This is the error bound for intervals starting with a.
5: Increase a, repeat step 4, and update the maximum error.
6: Return the maximum error for all a, which is ϵ_{GP}.

require integrating over the input $\mathbf{X}$. So we will use Monte Carlo integration once again. We compute Y'_L and Y'_S, as MC estimates of Y_L and Y_S respectively, from the samples in Algorithm 2. We also define (but do not compute) $\tilde{Y}'$, the random variable resulting from MC approximation of $\tilde{Y}$ with the same samples. An identical argument to that of Proposition 4.1 shows that

$$\mathcal{D}_\lambda(\tilde{Y}', \hat{Y}') = \sup_{[a,b]} |\tilde{\rho}' - \hat{\rho}'| \leq \sup_{[a,b]} \max\{|\rho'_L - \hat{\rho}'|, |\rho'_U - \hat{\rho}'|\} := \epsilon_{GP},$$

where adding a prime means to use Monte Carlo estimates.

Now we present an algorithm to compute ϵ_{GP}. The easiest way would be to simply enumerate all possible intervals. Because $\hat{Y}'$, Y'_S, and Y'_L are empirical CDFs over m samples, there are $O(m^2)$ possible values for ρ'_U, ρ'_L, and $\hat{\rho}'$. This can be inefficient for large numbers of samples m, as we observed empirically.

Instead, we present a more efficient algorithm to compute this error bound, as shown in Algorithm 3. The main idea is to (i) precompute the maximum differences between the mean function and each envelope function considering decreasing values of b (Step 2), then (ii) enumerate the values of a increasingly and use the precomputed values to bound $\tilde{\rho}'$ for intervals starting with a (Steps 3-5). This involves taking a pass through the m points in the empirical CDF of $\hat{Y}'$. Then for a given value of a, use binary search to find the smallest b_1 s.t. $\Pr[Y'_L \leq b_1] \leq \Pr[Y'_S \leq a]$. The complexity of this algorithm is $O(m \log m)$. More details are available in [24].

Combining Effects of Two Sources of Error. What we return to the users is the distribution of $\hat{Y}'$, from which $\hat{\rho}'$ can be computed for any interval. As noted, there are two sources of error in $\hat{\rho}'$: the GP modeling error and the MC sampling error. The latter arises from having $\hat{Y}'$, Y'_L, and Y'_S to approximate $\hat{Y}$, Y_L, and Y_S respectively. The GP error is from using the mean function to estimate $\tilde{\rho}$. We can combine these into a single error bound on the discrepancy:

$$\mathcal{D}_\lambda(\hat{Y}', \tilde{Y}) \leq \mathcal{D}_\lambda(\hat{Y}', \tilde{Y}') + \mathcal{D}_\lambda(\tilde{Y}', \tilde{Y}).$$

This follows from the triangle inequality that $\mathcal{D}_\lambda$ satisfies because it is a metric. Above we just showed that $\mathcal{D}_\lambda(\hat{Y}', \tilde{Y}') \leq \epsilon_{GP}$. Furthermore, $\mathcal{D}_\lambda(\tilde{Y}', \tilde{Y})$ is just the error due to a standard Monte Carlo approximation, which, as discussed in §2, can be bounded with high probability by, say, ϵ_{MC}, depending on the number of samples. Also, the two sources of error are independent. This yields the main error bound of this paper, which we state as follows.

Theorem 4.1 *If MC sampling is $(\epsilon_{MC}, \delta_{MC})$-approximate and GP prediction is $(\epsilon_{GP}, \delta_{GP})$-approximate, then the output has an error bound of $(\epsilon_{MC} + \epsilon_{GP})$ with probability $(1-\delta_{MC})(1-\delta_{GP})$.*

Computing Simultaneous Confidence Bands. Now we describe how to choose a high probability envelope, i.e., a pair (f_S, f_L) that contains $\tilde{f}$ with probability $1-\alpha$. We will use a band of the form $f_S = \hat{f}(\mathbf{x}) - z_\alpha\sigma(\mathbf{x})$ and $f_L = \hat{f}(\mathbf{x}) + z_\alpha\sigma(\mathbf{x})$. The problem is to choose z_α. An intuitive choice would be to choose z_α based on the quantiles of the univariate Gaussian, e.g., choose $z_\alpha = 2$ for a 95% confidence band. This would give us a *point-wise* confidence band, i.e., at any point $\mathbf{x}$, we would have $f_S(\mathbf{x}) \leq \tilde{f}(\mathbf{x}) \leq f_L(\mathbf{x})$. But we need something stronger. Rather, we want (f_S, f_L) such that the probability that $f_S(\mathbf{x}) \leq \tilde{f}(\mathbf{x}) \leq f_L(\mathbf{x})$ at all inputs $\mathbf{x}$ *simultaneously* is at least $1-\alpha$. An envelope with this property is called a *simultaneous confidence band*.

We will still use a band of the form $\hat{f}(\mathbf{x}) \pm z_\alpha\sigma(\mathbf{x})$, but we will need to choose a z_α large enough to get a simultaneous confidence band. Say we set z_α to some value z. The confidence band is satisfied if $Z(\mathbf{x}) := |\frac{\tilde{f}(\mathbf{x})-\hat{f}(\mathbf{x})}{\sigma(\mathbf{x})}| \leq z$ for any $\mathbf{x}$. Therefore, if the probability of $\sup_{\mathbf{x}\in\mathbf{X}} Z(\mathbf{x}) \geq z$ is small, the confidence band is unlikely to be violated. We adopt an approximation of this probability due to [3], i.e.,

$$\Pr[\sup_{\mathbf{x}\in\mathbf{X}} Z(\mathbf{x}) \geq z] \approx \mathbb{E}[\varphi(A_z(\mathbf{X})], \tag{5}$$

where the set $A_z(\mathbf{X}) := \{\mathbf{x} \in \mathbf{X} : Z(\mathbf{x}) \geq z\}$ is the set of all inputs where the confidence band is violated, and $\varphi(A)$ is the Euler characteristic of the set A. Also, [3] provides a numerical method to approximate Eq. (5) that works well for small α, i.e., high probability that the confidence band is correct, which is precisely the case of interest. The details are somewhat technical, and are omitted for space; see [3, 24]. Overall, the main computational expense is that the approximation requires computing second derivatives of the covariance function, but we have still found it to be feasible in practice. Once we computed the approximation to Eq. (5), we compute the confidence band by setting z_α to be the solution of the equation $\Pr[\sup_{\mathbf{x}\in\mathbf{X}} Z(\mathbf{x}) \geq z_\alpha] \approx \mathbb{E}[\varphi(A_z(\mathbf{X})] = \alpha$.

4.3 Error Bounds for KS Measure

The above analysis can be applied if the KS distance is used as the accuracy metric in a similar way. The main result is as follows.

Proposition 4.2 *Consider the mean function $\hat{f}(\mathbf{x})$ and the envelope $\hat{f}(\mathbf{x}) \pm z\sigma(\mathbf{x})$. Let $\tilde{f}(\mathbf{x})$ be a function in the envelope. Given uncertain input $\mathbf{X}$, let $\hat{Y} = \hat{f}(\mathbf{X})$ and $\tilde{Y} = \tilde{f}(\mathbf{X})$. Then $KS(\tilde{Y}, \hat{Y})$ is largest when $\tilde{f}(\mathbf{x})$ is at either the boundary of the envelope.*

Proof sketch. Recall that $KS(\tilde{Y}, \hat{Y}) = \sup_y |\Pr[\tilde{Y} \leq y] - \Pr[\hat{Y} \leq y]$. Let y_m correspond to the supremum in the formula of KS. Wlog, let $KS = \int(\mathbb{1}[\hat{f}(\mathbf{x}) \leq y_m] - \mathbb{1}[\tilde{f}(\mathbf{x}) \leq y_m])p(\mathbf{x})d\mathbf{x} > 0$. That is, for some $\mathbf{x}$, $\hat{f}(\mathbf{x}) \leq y_m < \tilde{f}(\mathbf{x})$. Now suppose there exists some $\mathbf{x}'$ s.t. $\tilde{f}(\mathbf{x}') < \hat{f}(\mathbf{x}')$, the KS distance would increase if $\hat{f}(\mathbf{x}') \leq \tilde{f}(\mathbf{x}')$. This means, KS becomes larger when $\tilde{f}(\mathbf{x}) \geq \hat{f}(\mathbf{x})$ for all $\mathbf{x}$; or, $\tilde{f}(\mathbf{x})$ lies above $\hat{f}(\mathbf{x})$ for all $\mathbf{x}$. Also, it is intuitive to see that among the functions that lie above $\hat{f}(\mathbf{x})$, $\hat{f}(\mathbf{x}) + z\sigma(\mathbf{x})$ yields the largest KS error, since it maximizes $\mathbb{1}[\hat{f}(\mathbf{x}) \leq y] - \mathbb{1}[\tilde{f}(\mathbf{x}) \leq y], \forall y$. (Similarly, we can show that if $KS = \int(\mathbb{1}[\tilde{f}(\mathbf{x}) \leq y_m] - \mathbb{1}[\hat{f}(\mathbf{x}) \leq y_m])p(\mathbf{x})d\mathbf{x} > 0$, KS is maximized if $\tilde{f}(\mathbf{x})$ lies below $\hat{f}(\mathbf{x})$ for all $\mathbf{x}$.) $\square$

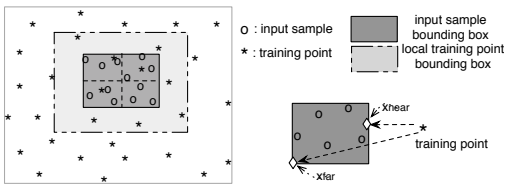


Figure 3: Choosing a subset of training points for local inference

As a result, let Y_S and Y_L be the output computed using the upper and lower boundaries $\hat{f}(\mathbf{x}) \pm z\sigma(\mathbf{x})$ respectively. Then, the KS error bound is $\max(KS(\hat{Y}, Y_S), KS(\hat{Y}, Y_L))$

We can obtain the empirical variables $\hat{Y}'$, Y_S', and Y_L' via Monte Carlo sampling as before. We also analyze the combining effects of the two sources of error, MC sampling and GP modeling, as for the discrepancy measure. We obtain a similar result: the total error bound is the sum of the two error bounds, ϵ_{MC} and ϵ_{GP}. The proof is omitted due to space constraints but available in [24].

5. AN OPTIMIZED ONLINE ALGORITHM

In Section 4.1, we present a basic algorithm (Algorithm 2) to compute output distributions when Gaussian processes model our UDFs. However, this algorithm does not satisfy our design constraints as follows. This is an offline algorithm since the training data is fixed and learning is performed before inference. Given an accuracy requirement, it is hard to know the number of training points, n, needed beforehand. If we use larger n, the accuracy is higher, but the performance suffers due to both the training cost $O(n^3)$ and the inference cost $O(n^2)$. We now seek an online algorithm that is robust to UDFs and input distributions in meeting accuracy requirements. We further optimize it for high performance.

5.1 Local Inference

We first propose a technique to reduce the cost of inference while maintaining good accuracy. The key observation is that the covariance between two points $\mathbf{x}_i$ and $\mathbf{x}_j$ is small when the distance between them is large. For example, the squared-exponential covariance function decreases exponentially in the squared distance, $k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{l^2}\}$. Therefore, the far training points have only small weights in the weighted average, and hence can be omitted. This suggests a technique that we call *local inference* with the steps shown in Algorithm 4. (We refer to the standard inference technique as *global inference*.)

Algorithm 4 Local inference

Input: Input distribution $p(\mathbf{x})$. Training data: $\{(\mathbf{x}_i^*, y_i^*), i = 1 \ldots n\}$, stored in an R-tree.

1: Draw m samples from the input distribution $p(\mathbf{x})$ and construct a bounding box for the samples.
2: Retrieve a set of training points, called X_L^*, that have distance to the bounding box less than a maximum distance specified by the local inference threshold Γ (discussed more below).
3: Run inference using X_L^* to get the function values at the samples. Return the CDF constructed from the inferred values.

Fig. 3 illustrates the execution of local inference to select a subset of training point given the input distribution. The darker rectangle is the bounding box of the input samples, and the lighter rectangle includes the training points selected for local inference.

Choosing the training points for local inference given a threshold. The threshold Γ is chosen so that the approximation error in $\hat{f}(\mathbf{x}_j)$, for all samples $\mathbf{x}_j$, is small. That is, $\hat{f}(\mathbf{x}_j)$ when computed using either global or local inference does not differ much. Revisit global inference as in Eq. 2. The vector $K(X^*, X^*)^{-1}\mathbf{y}^*$, called $\boldsymbol{\alpha}$, can be updated once the training data changes, and stored for later inference. Then, computing $\hat{f}(\mathbf{x}_j) = K(\mathbf{x}_j, X^*)K(X^*, X^*)^{-1}\mathbf{y}^* = K(\mathbf{x}_j, X^*)\boldsymbol{\alpha}$ involves a vector dot product. Note that the cost of computing this mean is $O(n)$; the high cost of inference $O(n^2)$ is due to computing the variance $\sigma^2(\mathbf{x}_j)$ (see §3.3 for more detail).

If we use a subset of training points, we approximate $\hat{f}(\mathbf{x}_j)$ with $\hat{f}_L(\mathbf{x}_j) = K(\mathbf{x}_j, X_L^*)\boldsymbol{\alpha}_L$. ($\boldsymbol{\alpha}_L$ is the same as $\boldsymbol{\alpha}$ except that the entries in $\boldsymbol{\alpha}$ that do not correspond to a selected training point are set to 0). Then the approximate error γ_j, for the sample j, is:

$$\begin{aligned}\gamma_j &\approx K(\mathbf{x}_j, X^*)\boldsymbol{\alpha} - K(\mathbf{x}_j, X_L^*)\boldsymbol{\alpha}_L \\ &= K(\mathbf{x}_j, X_{\bar{L}}^*)\boldsymbol{\alpha}_{\bar{L}} = \sum_{l \in \bar{L}} k(\mathbf{x}_j, \mathbf{x}_l^*)\alpha_l,\end{aligned}$$

where $X_{\bar{L}}^*$ are the training points excluded from local inference. Ultimately, we want to compute $\gamma = \max_j |\gamma_j|$, which is the maximum error over all the samples. The cost of computing γ by considering every j is $O(mn)$, as $j = 1 \ldots m$, which is high for large m.

We next present a more efficient way to compute an upper bound for γ. We use a bounding box for all the samples $\mathbf{x}_j$ as constructed during local inference. For any training point with index l, $\mathbf{x}_l^*$, let $\mathbf{x}_{near}$ be the closest point from the bounding box to $\mathbf{x}_l^*$ and $\mathbf{x}_{far}$ be the furthest point from the bounding box to $\mathbf{x}_l^*$ (see Fig. 3 for an example of these points). For any sample j we have:

$$k(\mathbf{x}_{far}, \mathbf{x}_l^*) \leq k(\mathbf{x}_j, \mathbf{x}_l^*) \leq k(\mathbf{x}_{near}, \mathbf{x}_l^*)$$

Next, by multiplying with α_l, we have the upper and lower bounds for $k(\mathbf{x}_j, \mathbf{x}_l^*)\alpha_l$. With these inequalities, we can obtain an upper bound γ_{upper} and lower bound γ_{lower} for $\gamma_j, \forall j$. Then,

$$\gamma = \max_j |\gamma_j| \leq \max(|\gamma_{upper}|, |\gamma_{lower}|)$$

Computing this takes time proportional to the number of excluded training points, which is $O(n)$. For each of these points, we need to consider the sample bounding box, which incurs a constant cost when the dimension of the function is fixed. After computing γ, we compare it with the threshold Γ. If $\gamma > \Gamma$, we expand the bounding box for selected training points and recompute γ until we have $\gamma \leq \Gamma$. Note that Γ should be set to be small compared with the domain of Y, i.e., the error incurred for every test point is small. In §6, we show how to set Γ to obtain good performance.

We mention an implementation detail to make the bound γ tighter, which can result in fewer selected training points for improved performance. We divide the sample bounding box into smaller non-overlapping boxes as shown in Fig. 3. Then for each box, we compute its γ, and then return the maximum of all these boxes.

Complexity for local inference. Let l be the number of selected training points; the cost of inference is $O(l^3 + ml^2 + n)$. $O(l^3)$ is to compute the inverse matrix $K(X_L^*, X_L^*)^{-1}$ needed in the formula of variance; $O(ml^2)$ is to compute the output variance; and $O(n)$ is to compute γ while choosing the local training points. Among the costs, $O(ml^2)$ is usually dominant (esp. for high accuracy requirement). This is an improvement compared to global inference, which has a cost of $O(mn^2)$, because l is usually smaller than n.

5.2 Online Tuning

Our objective is to seek an online algorithm for GPs: we start with no training points and collect them over time so that the function model gets more accurate. We can examine each input distribution on-the-fly to see whether more training points are needed given

an accuracy requirement. This contrasts with the offline approach where the training data must be obtained before inference.

To develop an online algorithm, we need to make two decisions. The first decision is *how many* training points to add. This is a task related to the error bounds from §4, that is, we add training points until the upper bound on the error is less than the user's tolerance level. The second decision is *where* the training points should be, specifically, what input location $\mathbf{x}_{n+1}$ to use for the next training point. A standard method is to add new training points where the function evaluation is highly uncertain, i.e., $\sigma^2(\mathbf{x})$ is large. We adopt a simple heuristic for this: we cache the Monte Carlo samples throughout the algorithm, and when we need more training points, we choose the sample $\mathbf{x}_j$ that has the largest predicted variance $\sigma^2(\mathbf{x}_j)$, compute its true function value $f(\mathbf{x}_j)$, and add it to the training data set. After that, we run inference, compute the error bound again, and repeat until the error bound is small enough. We have experimentally observed that this simple heuristic works well.

A complication is that when we add a new training point, the inverse covariance matrix gets bigger $K(X^*, X^*)^{-1}$, so it needs to be recomputed. Recomputing it from scratch would be expensive, i.e., $O(n^3)$. Fortunately, we can update it incrementally using the standard formula for inverting a block matrix (see [24] for details).

5.3 Online Retraining

In our work, the training data is obtained on the fly. Since different inputs correspond to different regions of the function, we may need to tune the GP model to best fit the up-to-date training data, i.e., to *retrain*. A key question is when we should perform retraining (as mentioned in §3.4). It is preferable that retraining is done infrequently due to its high cost of $O(n^3)$ in the number of training points and multiple iterations required. The problem of retraining is less commonly addressed in existing work for GPs.

Since retraining involves maximizing the likelihood function $\mathcal{L}(\boldsymbol{\theta})$, we will make this decision by examining the likelihood function. Recall also that the numerical optimizer, e.g., gradient descent, requires multiple iterations to find the optimum. A simple heuristic is to run training only if the optimizer is able to make a big step during its very first iteration. Given the current hyperparameters $\boldsymbol{\theta}$, run the optimizer for one step to get a new setting $\boldsymbol{\theta'}$, and continue with training only if $||\boldsymbol{\theta'} - \boldsymbol{\theta}||$ is larger than a pre-set threshold Δ_θ.

In practice, we have found that gradient descent does not work well with this heuristic, because it does not move far enough during each iteration. Instead, we use a more sophisticated heuristic based on a numerical optimizer, called Newton's method, which uses both the first and the second derivatives of $\mathcal{L}(\boldsymbol{\theta})$. Mathematical derivation shows that second derivatives of $\mathcal{L}(\boldsymbol{\theta})$ are:

$$\mathcal{L}''(\theta_j) = \frac{1}{2}\,\mathrm{tr}[(\frac{\partial K^{-1}}{\partial \theta_j}\mathbf{y}^*\mathbf{y}^{*T}K^{-1} + K^{-1}\mathbf{y}^*\mathbf{y}^{*T}\frac{\partial K^{-1}}{\partial \theta_j} - \frac{\partial K^{-1}}{\partial \theta_j})\frac{\partial K}{\partial \theta_j} + (K^{-1}\mathbf{y}^*\mathbf{y}^{*T}K^{-1} - K^{-1})\frac{\partial^2 K}{\partial \theta_j^2}],$$

where $\mathrm{tr}[\cdot]$ is the trace of a matrix. $\partial K/\partial\theta_j$ and $\partial^2 K/\partial\theta_j^2$ can be updated incrementally. (The details are shown in [24].)

5.4 A Complete Online Algorithm

We now put together all of the above techniques to form a complete online algorithm to compute UDFs on uncertain data using GPs. The main idea is, starting with no training data, given an input distribution, we use online tuning in §5.2 to obtain more training data, and run inference to compute the output distribution. Local inference in §5.1 is used for improved performance. When some training points are added, we use our retraining strategy to decide whether to relearn the GP model by updating its hyperparameters.

Algorithm 5 OLGAPRO: Compute output distribution using Gaussian process with optimizations

Input: Input tuple $\mathbf{X} \sim p(\mathbf{x})$. Training data: $\mathcal{T} = \{(\mathbf{x}_i^*, y_i^*), i = 1..n\}$; hyperparameters of the GP: $\boldsymbol{\theta}$. Accuracy requirement for the discrepancy measure: (ϵ, δ).

1: Draw m samples for $\mathbf{X}$, $\{\mathbf{x}_j, j = 1..m\}$, where m depends on the sampling error bound $\epsilon_{MC} < \epsilon$.
2: Compute the bounding box for these samples. Retrieve a subset of training points for local inference given the threshold Γ (see §5.1). Denote this set of training point $\mathcal{T}_\Gamma$.
3: **repeat**
4: Run local inference using $\mathcal{T}_\Gamma$ to get the output samples $\{(\hat{f}(\mathbf{x}_j), \sigma^2(\mathbf{x}_j)), j = 1..m\}$.
5: Compute the discrepancy error bound $\mathcal{D}_{upper}$ using these samples (see §4.2).
6: If $\mathcal{D}_{upper} > \epsilon_{GP}$, add a new training point at the sample with largest variance, i.e., $(\mathbf{x}_{n+1}^*, f(\mathbf{x}_{n+1}^*))$ (see §5.2), and insert this point into the training data index. Set $n := n + 1$.
7: **until** $\mathcal{D}_{upper} \leq \epsilon_{GP}$
8: **if** one or more training points are added **then**
9: Compute the log likelihood $\mathcal{L}(\boldsymbol{\theta}) = \log p(\mathbf{y}^*|X^*, \boldsymbol{\theta})$ and its first and second derivatives, and estimate δ_θ (see §5.3).
10: **if** $\delta_\theta \leq \Delta_\theta$ **then**
11: Retrain to get the new hyperparameters $\boldsymbol{\theta'}$. Set $\boldsymbol{\theta} := \boldsymbol{\theta'}$.
12: Rerun inference.
13: **end if**
14: **end if**
15: Return the distribution of Y, computed from samples $\{\hat{f}(\mathbf{x}_j)\}$.

Our algorithm, which we name OLGAPRO, standing for *ONline GAussian PROcess*, is shown as Algorithm 5. The objective is to compute the output distribution that meets the user-specified accuracy requirement under the assumption of GP modeling. The main steps of the algorithm involve: (a) Compute the output distribution by sampling the input and inferring with the Gaussian process (Steps 1-4). (b) Compute the error bound (Steps 5-7). If this error bound is larger than the allocated error bound, use online tuning to add a new training point. Repeat this until the error bound is acceptable. (c) If one or more training points have been added, decide whether retraining is needed and if so perform retraining (Steps 8-12).

Parameter setting. We further consider the parameters used in the algorithm. The choice of Γ for local inference in step 2 is discussed in §5.1). The allocation of two sources of error, ϵ_{MC} and ϵ_{GP} is according to Theorem 4.1, $\epsilon = \epsilon_{MC} + \epsilon_{GP}$. Then our algorithm automatically chooses the number of samples m to meet the accuracy requirement ϵ_{MC} (see §2 for the formula). For retraining, setting the threshold Δ_θ, mentioned in §5.3, smaller will trigger retraining more often but potentially make the model more accurate, while setting it high can give inaccurate results. In §6, we experimentally show how to set these parameters efficiently.

Complexity. The complexity of local inference is $O(l^3 + ml^2 + n)$ as shown in §5.1. Computing the error bound takes $O(m \log m)$ (see §4.2). And, retraining takes $O(n^3)$. The number of samples m is $O(1/\epsilon_{MC}^2)$, while the number of training points n depends on ϵ_{GP} and the UDF itself. The unit cost is basic math operations, in contrast to complex function evaluations as in standard MC simulation. This is because when the system converges, we seldomly need to add more training points, or to call function evaluation. Also, at convergence, the high cost of retraining can be avoided; the computation needed is for inference and computing error bounds.

Hybrid solution. We now consider a hybrid solution that combines our two approaches: direct MC sampling, and GP modeling and inference. The need for a hybrid solution arises since functions can vary in their complexity and evaluation time. Therefore, when given a black-box UDF, we explore these properties on the fly and choose the better solution. We can measure the function evaluation time while obtaining training data. We then run GPs to convergence, measure its inference time, and then compare the running times of the two approaches. Due to space constraints, the details of this solution are deferred to [24]. In §6, we conduct experiments to determine the cases where each approach can be applied.

5.5 Online Filtering

In the presence of a selection predicate on the UDF output, similar to the filtering technique for Monte Carlo simulation (§2), we also consider online filtering when sampling with a Gaussian process. Again, we consider selection with the predicate $a \leq f(\mathbf{x}) \leq b$. Let $(\hat{f}(\mathbf{x}), \sigma^2(\mathbf{x}))$ be the estimate at any input point $\mathbf{x}$. With the GP approximation, the tuple existence probability $\tilde{\rho}$ is approximated with $\hat{\rho} = \Pr[\hat{f}(\mathbf{x}) \in [a, b]]$. This is exactly the quantity that we bounded in §4.2, where we showed that $\tilde{\rho} \leq \rho_U$. So in this case, we filter tuples whose estimate of ρ_U is less than our threshold. Again, since ρ_U is computed from the samples, we can check this online for filtering decision as in §2.

6. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our proposed techniques using both synthetic functions and data with controlled properties, and real workloads from the astrophysics domain.

6.1 Experimental Setup

We first use synthetic functions with controlled properties to test the performance and sensitivity of our algorithms. We now describe the settings of these functions, input data and parameters used.

A. Functions. We generate functions (UDFs) of different shapes in terms of bumpiness and spikiness. A simple method is to use Gaussian mixtures [10] to simulate various function shapes (which should not be confused with the input and output distributions of the UDF and by no means favors our GP approach). We vary the number of Gaussian components, which dictates the number of peaks of a function. The means of the components determine the domain, and their covariance matrix determines the stretch and bumpiness of the function. We denote the function dimensionality d; this is the number of input variables of the function. We observe that in real applications, many functions have low dimensionality, e.g., 1 or 2 for astrophysics functions. For evaluation purposes, we vary d in a wider range of [1,10]. Besides the shape, a function is characterized by the evaluation time, T, which we vary in the range 1µs to 1s.

B. Input Data. By default, we consider uncertain data following Gaussian distributions, i.e., the input vector has distribution characterized by $\mathcal{N}(\boldsymbol{\mu}_I, \Sigma_I)$. μ_I is drawn from the given support of the function $[L, U]$. Σ_I determines the spread of the input distributions. For simplicity, we assume the input variables of a function are independent, but supporting correlated input is not harder—we just need to sample from the joint distributions. We also consider other distributions including exponential and Gamma. We note that handling other types of distributions is similar due to the same reason (the difference is the cost of sampling).

C. Accuracy Requirement. We use the discrepancy measure as the accuracy metric in our experiments. The user specifies the accuracy requirement (ϵ, δ) and the minimum interval length λ. λ is set to be a small percentage (e.g., 1%) of the range of the function. This requirement means that with probability $(1-\delta)$, for any interval of length at least λ, the probabilities of an interval computed from the approximate and true output distributions do not differ from each other by more than ϵ. For the GP approach, the error bound ϵ is allocated to two sources of error, GP error bound ϵ_{GP} and sampling error bound ϵ_{MC}, where $\epsilon = \epsilon_{GP} + \epsilon_{MC}$. We also distribute δ so that $1 - \delta = (1 - \delta_{GP})(1 - \delta_{MC})$.

Our default setting is as follows. The domain of function $[L, U] = [0, 10]$, input standard deviation $\sigma_I = 0.5$, function evaluation time $T = 1ms$, accuracy requirement $(\epsilon = 0.1, \delta = 0.05)$. The reported results are averaged from 500 output distributions or when the algorithm converges, whichever is larger.

6.2 Evaluating our GP Techniques

We first evaluate the individual techniques employed in our Gaussian process algorithm, OLGAPRO. The objective is to understand and set various internal parameters of our algorithm.

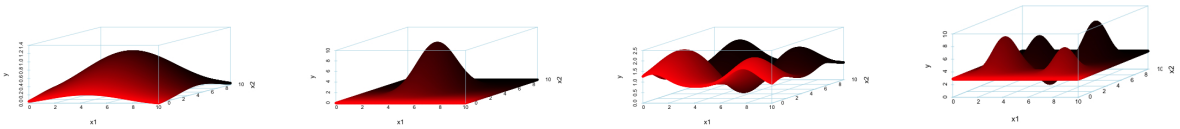
Profile 1: Accuracy of function fitting. We first choose four two-dimensional functions of different shapes and bumpiness (see Fig. 4). These functions are the four combinations between (i) one or five components, (ii) large or small variance of Gaussian components, which we refer to as $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, and $\mathcal{F}_4$. First, we check the effectiveness of GP modeling. We vary the number of training points n and run basic global inference at test points. Fig. 5(a) shows the relative errors for inference, i.e., $|\frac{\hat{f}(\mathbf{x}) - f(\mathbf{x})}{f(\mathbf{x})}|$, evaluated at a large number of test points. The simplest function $\mathcal{F}_1$ with one peak and being flat needs a small number of training points, e.g., 30, to be well approximated. In contrast, the most bumpy and spiky function $\mathcal{F}_4$ requires the largest number of points, $n > 300$, to be accurate. The other two functions are in between. This confirms that the GP approach can model functions of different shapes well, however the number of training points needed varies with the function. In the later experiments, we will show that OLGAPRO can robustly determine the number of training points needed online.

Profile 2: Behavior of error bound. We next test the behavior of our discrepancy error bound, which is described in §4.2 and computed using Algorithm 3. We compute the error bounds and measure the actual errors. Fig. 5(b) shows the result for the function $\mathcal{F}_4$, which confirms that the error bounds are actual upper bounds and hence indicates the validity of GP modeling. More interestingly, it shows how tight the bounds are (about 2 to 4 times of the actual errors). As λ gets smaller, more intervals are considered for the discrepancy measure; thus, the errors and error bounds, the suprema for a larger set of intervals, get larger. We test the other functions and observe the same trends. In the following experiments, we use a stringent requirement: setting λ to be 1% of the function range.

Profile 3: Allocation of two sources of error. We also examine the allocation of the user-specified error bound ϵ to the errors from GP modeling and MC sampling, ϵ_{GP} and ϵ_{MC}, as in Theorem 4.1. The details are omitted due to space constraints, but are discussed in [24]. In general, we set ϵ_{MC} to be 0.7ϵ for good performance.

In the next three experiments, we evaluate three key techniques employed in our GP approach. The default function is $\mathcal{F}_4$.

Expt 1: Local inference. We first consider our local inference technique as shown in §5.1. We compare the accuracy and running time of local inference with those of global inference. For now, we fix the number of training points to compare the performance of the two inference techniques. We vary the threshold Γ of local inference from 0.1% to 20% of the function range. Recall that setting Γ small corresponds to using more training points and hence similar to global inference. Our goal is to choose a setting of Γ so that local inference has similar accuracy as global inference while being faster. Figs. 5(c) and 5(d) show the accuracy and running time,



(a) Funct1 (b) Funct2 (c) Funct3 (d) Funct4

Figure 4: A family of functions of different smoothness and shape used in evaluation.

respectively. We see that for most of values Γ tested, local inference is as accurate as global inference while offering a speedup from 2 to 4 times. We repeat this experiment for other functions and observe that for less bumpy functions, the speedup for local inference is less pronounced, but the accuracy is always comparable. This is because for smooth functions, far training points still have a high weight in inference. In general, we set Γ about (0.05x function_range), which results in good accuracy and improved running time.

Expt 2: Online tuning. In §5.2, we proposed adding training points on-the-fly to meet the accuracy requirement. We now evaluate our heuristics of choosing samples with the largest variance to add. We compare it with two following heuristics: Given an input distribution, a simple one is to choose a sample of the input at random. Another heuristics is what we call "optimal greedy", which considers all samples, simulates adding each of them to compute a new error bound, and then picks the sample having the most error bound reduction. This is only hypothetical since it is prohibitively expensive to simulate adding every sample. For only this experiment, we assume that each input has 400 samples for "optimal greedy" to be feasible. We start with just 25 training points and add more when necessary. Fig. 5(e) shows the accumulated number of training points added over time (for performance, we restrict that no more than 10 points can be added for every input). As observed, our technique using the largest variance requires fewer training points, hence runs faster, than randomly adding points. Also, it is close to our "optimal greedy" while being much faster to be run online.

Expt 3: Retraining strategy. We now examine the performance of our retraining strategy (see §5.3). We vary our threshold Δ for retraining and compare this strategy with two other strategies: eager training when one or more training points are added, and no training. Again, we start with a small number of training points and add more using online tuning. Figs. 5(f) and 5(g) show the accuracy and running time respectively. As expected, setting Δ smaller means retraining more often and is similar to eager retraining, while larger Δ means less retraining. We see that setting Δ less than 0.5 gives best performance, as fewer retraining calls are needed while the hyperparameters are still good estimates. We repeat this experiment with other functions and see that conservatively setting $\Delta = 0.05$ gives good performance for this set of functions. In practice, Δ can be chosen in reference with the hyperparameter values.

6.3 GP versus Monte Carlo Simulation

We next examine the performance of our complete online algorithm, OLGAPRO (Algorithm 5). The internal parameters are set as above. We also compare this algorithm with the MC approach.

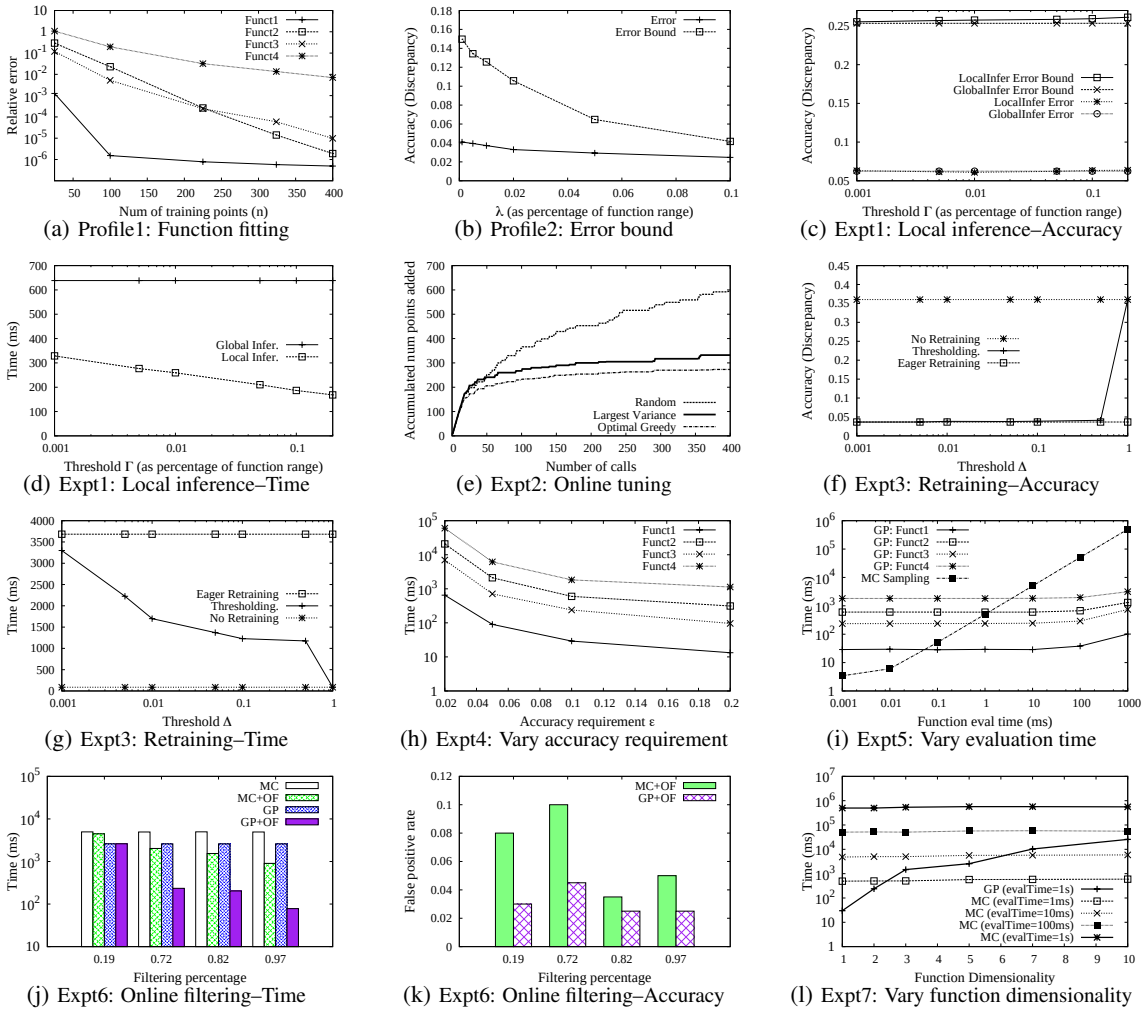
Expt 4: Varying user-specified ϵ. We run the GP algorithm for all four functions $\mathcal{F}_1$ to $\mathcal{F}_4$. We vary ϵ in the range of [0.02, 0.2]. Fig. 5(h) shows the running time for the four functions. (We verify that the accuracy requirement ϵ is always satisfied, and omit the plot due to space constraints.) As ϵ gets smaller, the running time increases. This is due to the fact that the number of samples is proportional to $1/\epsilon_{MC}^2$. Besides, small ϵ_{GP} requires more training points, hence higher cost for inference. This experiment also verifies the effect of the function complexity on the performance. A flat function like $\mathcal{F}_1$ needs much fewer training points than a bumpy, spiky function like $\mathcal{F}_4$, thus running time is about two orders of magnitude different. We also repeat this experiment for other input distributions including Gamma and exponential distributions, and observe very similar results, which is due to our general approach of working with input samples. Overall, our algorithm can robustly adapt to the function complexity and the accuracy requirement.

Expt 5: Varying evaluation time T. The tradeoff between the GP and MC approaches mainly lies in the function evaluation time T. In this experiment, we fix $\epsilon = 0.1$ and vary T from 1µs to 1s. Fig. 5(i) shows the running time of the two approaches for all four functions. Note that the running time for MC sampling is similar for all functions, hence we just show one line. As observed, the GP approach starts to outperform the sampling approach when function evaluation takes longer than 0.1ms for simple functions like $\mathcal{F}1$, and up to 10ms for complex functions like $\mathcal{F}4$. Also we note that our GP approach is almost insensitive to function evaluation time, which is only incurred during the early phase. After convergence, function evaluation occurs only infrequently. This demonstrates the applicability of the GP approach for long running functions.

This result also argues for the use of a hybrid solution as described in §5.4. Since the function complexity is unknown beforehand, so is the number of training points. The hybrid solution can be performed to automatically pick the better approach based on the function's complexity and evaluation time, e.g., the GP method is used for simple functions with evaluation time of 0.1ms or above, and for only complex functions with longer time.

Expt 6: Optimization for selection predicates. We examine the performance of online filtering when there is a selection predicate. As shown in §2 and §5.5, this can be used for both direct MC sampling and sampling with a GP. We vary the selection predicate, which in turn affects the rate that the output is filtered. We decide to filter output whose tuple existence probability is less than 0.1. Fig. 5(j) shows the running time. As seen, when the filtering rate is high, online filtering helps reduce the running time, by a factor of 5 and 30 times for MC and GP respectively. We observe that the GP approach has a higher speedup because besides processing fewer samples, it results in a GP model with fewer training points, or smaller inference cost. Fig. 5(k) shows the false positive rates, i.e., tuples should be filtered but are not during the sampling process. We observe that this rate is low, always less than 10%. The false negative rates are zero or negligible (less than 0.5%).

Expt 7: Varying function dimensionality d. We consider different functions with dimension d varying from 1 to 10. Fig. 5(l) shows the running time of these functions for both approaches. Since the running time using GP is insensitive to function evaluation time, we show only one line for $T = 1s$ for clarity. We observe that with GPs, high-dimensional functions incur high cost, because more training points are needed to capture a larger region. Even with a high dimension of 10, the GP approach still outperforms MC when the function evaluation time reaches 0.1s.



(a) Profile1: Function fitting (b) Profile2: Error bound (c) Expt1: Local inference–Accuracy

(d) Expt1: Local inference–Time (e) Expt2: Online tuning (f) Expt3: Retraining–Accuracy

(g) Expt3: Retraining–Time (h) Expt4: Vary accuracy requirement (i) Expt5: Vary evaluation time

(j) Expt6: Online filtering–Time (k) Expt6: Online filtering–Accuracy (l) Expt7: Vary function dimensionality

Figure 5: Experimental results including profiling and evaluating GP and MC approaches using synthetic data and functions

The results indicate that the hybrid approach is feasible by encoding general rules based on the known dimensionality and observed evaluation time. When the function is really fast, i.e., $T \leq 0.01ms$, MC sampling is a preferred solution. For most functions we see in our applications, which have low dimensionality, we use GP approach for better performance if functions have $T \geq 1ms$. For very high-dimensional functions, we use Monte Carlo approach.

6.4 A Case Study: UDFs in Astrophysics

In this experiment, we consider the application of our techniques in the astrophysics domain using real functions and data. We examined functions to compute various astrophysical quantities, available in a library package [1] and found eight functions computing a scalar value, which can all be incorporated into our framework; our algorithms treat them as black-box UDFs. Most of these UDFs are one and two-dimensional, usually have simple shapes but can be slow-running due to complex numerical computation. We chose three representative functions, as shown below, that vary in evaluation time across a range. (See §1 for the queries using these functions.)

FunctName	Dim.	EvalTime (ms)
AngDist	2	0.00298
GalAge	1	0.29072
ComoveVol	2	1.82085

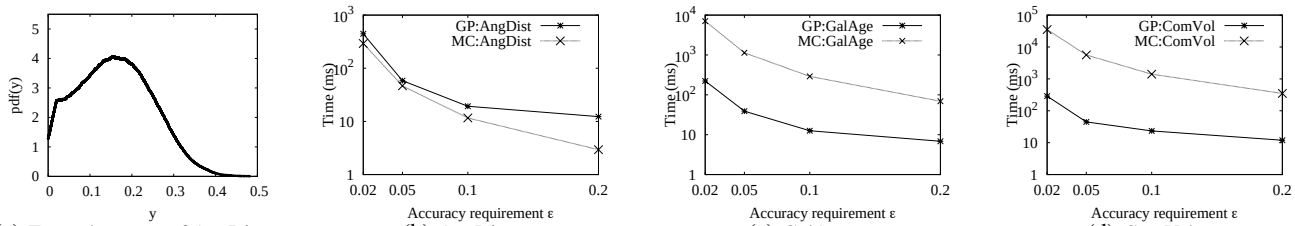
We use a real dataset from the Sloan digital sky survey (SDSS) project [21] and extract the necessary attributes for these functions, which are uncertain and characterized by Gaussian distributions.

We vary the accuracy requirement ϵ from 0.02 to 0.2. We verify that output distributions are non-Gaussian; an example output of $AngDist$ is shown in Fig. 6a. We compare the performance of Monte Carlo simulation with our algorithm OLGAPRO, which is shown in Fig. 6. (We verify that the errors are less than ϵ.)) Overall, we observe that these UDFs are generally smooth and non-spiky, hence do not need many training points. The 1D function $GalAge$ requires about 10 training points, while the two 2D functions, $AngDist$ and $ComoveVol$, require less than 40 points. OLGAPRO adds most training points for the first few input tuples; after that, it becomes stable. $AngDist$ is a quite fast function and OLGAPRO is somewhat slower than MC sampling. For the other two functions, $GalAge$ and $ComoveVol$, whose evaluation time is about 0.3 and 2 ms respectively, OLGAPRO outperforms MC sampling by one to two orders of magnitude. These results are consistent with those of the synthetic functions shown above and demonstrate the applicability of our techniques for the real workloads.

7. RELATED WORK

We discuss several broad areas of research related to our work.

Work on UDFs with deterministic input. Existing work, e.g.,[6], considers queries invoking UDFs that are expensive to execute and



(a) Example output of AngDist (b) AngDist (c) GalAge (d) ComVol

Figure 6: Results for real astrophysics functions and SDSS data

proposes reordering the execution of the predicates. Another work [8] considers functions that are iterative and computed by numerical methods, whose accuracy varies with computation cost. It proposes to adaptively execute these functions depending on given queries. However, this line of work considers only deterministic input, hence inherently different from our work.

Gaussian process regression with uncertain input. Regression using GPs has been well studied in the machine learning literature (see [18] for a survey). However, most of the existing work considers deterministic input and presents offline solutions, while we consider uncertain input in an online setting. One line of work from statistics most related to ours is [15], which uses GPs as emulators to computer code that is slow to run. It briefly mentions using sampling to handle uncertain input, but does not quantify the approximation, which we tackle by deriving error bounds. Further, we present an online algorithm with different novel optimizations. The prior work [12] computes only the mean and variance of the output. Since the UDF output in most cases is not Gaussian, this approach does not fully characterize the output distribution. In addition, [12] mainly considers input data following Gaussian distributions, while our work can support input of any distributions.

Optimizations of Gaussian processes. Existing lines of work [14, 17] propose optimizations for inference with GPs; however, they work in an offline manner and are not suitable for our online settings due to the lack of online tuning to obtain training data on the fly and retraining strategies to reduce the training overhead. Regarding inference only, the paper [14] suggests pre-dividing the function domain into fixed local regions corresponding to local models, then runs inference using the local models and combines the results by weighting them. This is different from our local inference technique since all training points are used, and hence can be inefficient for large training datasets. The work [17] has a more similar idea to ours by using sparse covariance matrices, which zeroes out low covariances, to reduce the number of training points under inference. However, it does not quantify the approximation errors as ours.

8. CONCLUSIONS AND FUTURE WORK

We have examined the problem of supporting user-defined functions on uncertain data. Given the black-box UDFs, we proposed two approaches: a simple Monte Carlo approach and a more complex, learning approach using Gaussian processes. For the GP approach, we presented new results to compute the output distributions and quantify the error bounds. We then presented an online algorithm that can adapt to user accuracy requirements, together with a suite of novel optimizations. Our evaluation using both real-world and synthetic functions shows that our proposed GP approach can outperform the MC approach with up to two orders of magnitude improvement for a variety of UDFs. For future work, we plan to study the support for a wider range of functions such as high-dimensional input and multivariate output, and consider to extend our techniques to allow for parallel processing for high performance.

Acknowledgements. This work was supported in part by the National Science Foundation under the grants IIS-1218524 and IIS-0746939, and by the Engineering and Physical Sciences Research Council [grant number EP/J00104X/1].

9. REFERENCES

[1] Idl astronomy library. `http://idlastro.gsfc.nasa.gov`.
[2] Sloan digital sky survey. `http://www.sdss.org`.
[3] R. J. Adler. Some new random field tools for spatial analysis. In *Stochastic enviromental research and risk assessment*, 2008.
[4] L. Antova, et al. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, 2008.
[5] C. M. Bishop. *Pattern recognition and machine learning*. Springer-Verlag New York, Inc., 2009.
[6] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *ACM TODS*, pages 87–98, 1996.
[7] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4), pages 523–544, 2007.
[8] M. Denny and M. J. Franklin. Adaptive execution of variable-accuracy functions. In *VLDB*, pages 547–558, 2006.
[9] A. Deshpande, et al. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.
[10] G. McLachlan and D. Peel *Finite Mixture Models*. Wiley-Interscience, 2000.
[11] A. L. Gibbs and F. E. Su. On choosing and bounding probability metrics. *International Statistical Review*, 70, pages 419–435, 2002.
[12] A. Girard, et al. GP priors with uncertain inputs - application to multiple-step ahead time series forecasting. In *NIPS*, 529–536, 2003.
[13] J. F. Kurose, et al. An end-user-responsive sensor network architecture for hazardous weather detection, prediction and response. In *AINTEC*, pages 1–15, 2006.
[14] D. T. Nguyen and J. Peters. Local Gaussian process regression for real-time model-based robot control. In *Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 380 –385, 2008.
[15] A. O'Hagan. Bayesian analysis of computer code outputs: A tutorial. In *Reliability Engineering and System Safety*, 2006.
[16] L. Peng, et al. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 4(11), pages 1169–1180, 2011.
[17] A. Ranganathan and M. H. Yang. Online sparse matrix Gaussian process regression and vision applications. In *ECCV*, 468–482, 2008.
[18] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. MIT Press, 2009.
[19] P. Sen, et al. Exploiting shared correlations in probabilistic databases. In *VLDB*, pages 809–820, 2008.
[20] S. Singh, et al. Database support for probabilistic attributes and tuples. In *ICDE*, pages 1053–1061, 2008.
[21] A. S. Szalay, et al. Designing and mining multi-terabyte astronomy archives: The Sloan digital sky survey. In *SIGMOD*, pp. 451–462, 2000.
[22] T. Tran, et al. Probabilistic inference over RFID streams in mobile environments. In *ICDE*, pages 1096–1107, 2009.
[23] T. T. L. Tran, et al. Claro: Modeling and processing uncertain data streams. *VLDB J.*, pages 651–676, 2012.
[24] T. T. L. Tran, et al. Supporting user-defined functions on uncertain data. UMass technical report, 2012. Available at `http://www.cs.umass.edu/~ttran/udf_tr.pdf`